UNIVERSITY OF CINCINNATI

Date: 21-May-2010

I, Suma P Snehalatha, hereby submit this original work as part of the requirements for the degree of:

Master of Science

in Computer Engineering

It is entitled:

Logical Modeling of ETL Processes Using XML

Student Signature: Suma P Snehalatha

This work and its defense approved by:

Committee Chair: Karen Davis, PhD

Karen Davis, PhD

6/18/2010
Logical Modeling of ETL Processes Using XML

A thesis submitted to the

Division of Graduate Studies and Research
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of Electrical and Computer Engineering
of the College of Engineering and Applied Science
May 2010

by

Suma P. Snehalatha
B.Tech., Cochin University of Science and Technology, 2004

Thesis Advisor and Committee Chair: Dr. Karen C. Davis
Abstract

Data warehouses are an integral component for business decision making when a large volume of data has to be analyzed. Data in data warehouses come from sources including but not limited to relational databases, legacy systems, and flat files. The data from these sources have to be processed to conform to the data model and type of the data warehouse. Extract, Transform and Load (ETL) tools are used to take data from multiple sources to populate a data warehouse.

Mapping the source data to the warehouse schema is a complicated task and requires business analysts and ETL professionals to work together. The analyst identifies the data of interest for business decision making. It is the task of the ETL programmer to identify the sources and the operations required to convert the source data to conform to the data model of the data warehouse. In a typical ETL scenario, the business data requirements are identified and documented by an analyst. The ETL programmers identify the sources and operations to be performed to meet the requirements and implement them in an ETL tool. Our work focuses on a scenario where ETL requirements are documented in natural language text or diagrams and the ETL programmer converts these to an ETL process.

In this thesis, we propose a platform-independent logical model using XML for ETL processes and develop an implementation to generate program code from the logical model with minimal programmer intervention. For proof of concept, we choose an ETL tool, SAS, and generate a program from a logical specification. Automating the conversion process from the logical model to ETL process helps to greatly reduce the coding time spent in developing ETL programs. The logical model serves as an internal representation for an ETL process. Since it is platform-independent, it can serve as a common model for specifying logical ETL processes for different target ETL tools.
Table of Contents

Chapter 1: Introduction ............................................................................................................................. 1
  1.1 Motivating Example.......................................................................................................................... 2
  1.2 General Research Objective........................................................................................................... 4
  1.3 Specific Research Objectives ......................................................................................................... 4
  1.4 Research Methodology ................................................................................................................. 5
  1.5 Contributions of the Research ........................................................................................................ 5
  1.6 Overview of the Thesis .................................................................................................................... 6

Chapter 2: ETL Logical Modeling ............................................................................................................ 7
  2.1 Overview of ETL Modeling ............................................................................................................ 7
  2.2 Research Literature on ETL Logical Modeling ........................................................................... 8
  2.3 Feature Summary and Comparison of ETL Logical Modeling Tools ......................................... 10
  2.4 Conclusion ...................................................................................................................................... 11

Chapter 3: Identifying Basic ETL Operators ......................................................................................... 12
  3.1 ETL Activities .................................................................................................................................. 12
  3.2 ETL Operators ................................................................................................................................... 12
    3.2.1 Filter........................................................................................................................................ 13
    3.2.2 Join......................................................................................................................................... 17
    3.2.3 Format .................................................................................................................................... 18
    3.2.4 Aggregate .............................................................................................................................. 22
    3.2.5 Union .................................................................................................................................... 24
    3.2.6 Sort ...................................................................................................................................... 25
    3.2.7 Copy ..................................................................................................................................... 25
  3.3 Conclusions ...................................................................................................................................... 27

Chapter 4: Logical Modeling of Operators Using XML ...................................................................... 28
  4.1 Logical Modeling of ETL Processes .............................................................................................. 28
  4.2 Describing Source and Target Schemas ......................................................................................... 29
    4.2.1 Source Schema Representation ............................................................................................. 29
    4.2.2 Target Schema Representation ............................................................................................. 29
  4.3 Describing Mappings ....................................................................................................................... 30
    4.3.1 Mapping between Operators ................................................................................................. 30
    4.3.2 Mapping within Operators .................................................................................................... 31
  4.4 Logical Modeling of ETL Operators .............................................................................................. 31
    4.4.1 Filter .................................................................................................................................... 31
    4.4.2 Join ...................................................................................................................................... 32
    4.4.3 Format ................................................................................................................................... 33
    4.4.4 Aggregate .............................................................................................................................. 34
    4.4.5 Union ................................................................................................................................... 36
    4.4.6 Sort .................................................................................................................................... 36
    4.4.7 Copy ................................................................................................................................... 37
  4.5 Conclusions ...................................................................................................................................... 38

Chapter 5: Converting the Logical Model to an ETL Program ............................................................... 39
  5.1 Overview of SAS Programming ..................................................................................................... 39
  5.2 Generating SAS Code from an ETL Requirement ........................................................................... 40
List of Tables

Table 2.1 Comparison of features of OLTP and OLAP systems .................................. 7
Table 2.2 Comparison of different approaches to ETL logical modeling ..................... 10

Table 3.1 Summary of operator implementation in Orchid [DHWRZ08] and ARKTOS II [VS05] ......... 27

Table 5.1 Process table for the ETL requirement in Figure 1.1 ..................................... 48
Table 5.2 Entity table for the ETL requirement in Figure 1.1 ........................................ 49
Table 5.3 Relationship table for the ETL requirement in Figure 1.1 ............................. 50
Table 5.4 Attribute table for the ETL requirement in Figure 1.1 ................................. 51
Table 5.5 Mapping table for the XML document in Figure 4.1 ................................. 52
Table 5.6 Terminology mapping between the XML model and SAS model .................. 53
Table 5.7 SAS_Entity table entries for the ETL requirement in Figure 1.1 ....................... 54
Table 5.8 SAS_Relationship table entries for the ETL requirement in Figure 1.1 .......... 55

Table 6.1 Process table entry for the ETL requirement in Figure 6.15 .......................... 76
Table 6.2 Entity table entries for the ETL requirement in Figure 6.15 .......................... 76
Table 6.3 Relationship table entries for the ETL requirement in Figure 6.15 .................. 76
Table 6.4 Attribute table entries for the ETL requirement in Figure 6.15 ....................... 77
Table 6.5 Mapping table entries for the ETL requirement in Figure 6.15 ....................... 78
Table 6.6 SAS_Entity table entries for the ETL requirement in Figure 6.15 .................... 78
Table 6.7 SAS_Relationship table entries for the ETL requirement in Figure 6.15 .......... 79
Table 6.8 Process table entry for the ETL requirement in Figure 6.16 ......................... 80
Table 6.9 Entity table entries for the ETL requirement in Figure 6.16 ......................... 80
Table 6.10 Relationship table entries for the ETL requirement in Figure 6.16 ............... 80
Table 6.11 Attribute table entries for the ETL requirement in Figure 6.16 ..................... 81
Table 6.12 Mapping table entries for the ETL requirement in Figure 6.16 ..................... 81
Table 6.13 SAS_Entity table entries for the ETL requirement in Figure 6.16 ................. 82
Table 6.14 SAS_Relationship table entries for the ETL requirement in Figure 6.15 .......... 82
## List of Figures

Figure 1.1 Block diagram of an example ETL process ................................................................. 3

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Sample filter operation</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Filter stage representation in OHM [DHWRZ08]</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Sample join operation</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Join stage representation in OHM [DHWRZ08]</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Example for format operation</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>Transform stage representation in OHM [DHWRZ08]</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Modify stage representation in OHM [DHWRZ08]</td>
<td>20</td>
</tr>
<tr>
<td>3.8</td>
<td>Example for aggregate operator</td>
<td>22</td>
</tr>
<tr>
<td>3.9</td>
<td>Aggregator stage representation in OHM [DHWRZ08]</td>
<td>23</td>
</tr>
<tr>
<td>3.10</td>
<td>Example for union operator</td>
<td>24</td>
</tr>
<tr>
<td>3.11</td>
<td>Example for sort operator</td>
<td>25</td>
</tr>
<tr>
<td>3.12</td>
<td>Example for copy operator</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Traversal of XML document: operator input not available</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Traversal of XML document: resuming from game_sales_asia</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Traversal of XML document: input needs to be written to dataset</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Traversal of XML document: input needs to be written to dataset</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Traversal of XML document: PROC SORT step input and output written to dataset</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Traversal of XML document: PROC MEANS step output written to dataset</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>The four parts of the Multilevel Data Dictionary [ACB05]</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>Two ER schema instances [ACB05]</td>
<td>46</td>
</tr>
<tr>
<td>5.9</td>
<td>Multilevel dictionary representation of the ER schemas in Figure 5.8 [ACB05]</td>
<td>47</td>
</tr>
<tr>
<td>5.10</td>
<td>Algorithm to generate SAS code from SAS process tables</td>
<td>56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction screen of the application</td>
<td>62</td>
</tr>
<tr>
<td>6.2</td>
<td>Generate XML screen</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Generate XML screen: Add Source block</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>Generate XML screen: Add Target block</td>
<td>65</td>
</tr>
<tr>
<td>6.5</td>
<td>Generate XML screen: Add Filter block</td>
<td>66</td>
</tr>
<tr>
<td>6.6</td>
<td>Generate XML screen: Add Join block</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>Generate XML screen: Add Format block</td>
<td>68</td>
</tr>
<tr>
<td>6.8</td>
<td>Generate XML screen: Add Aggregate block</td>
<td>69</td>
</tr>
<tr>
<td>6.9</td>
<td>Generate XML screen: Add Sort block</td>
<td>70</td>
</tr>
<tr>
<td>6.10</td>
<td>Generate XML screen: Add Union block</td>
<td>71</td>
</tr>
<tr>
<td>6.11</td>
<td>Generate XML screen: Add Copy block</td>
<td>72</td>
</tr>
<tr>
<td>6.12</td>
<td>Generate XML model tables screen</td>
<td>73</td>
</tr>
<tr>
<td>6.13</td>
<td>Generate SAS model tables screen</td>
<td>74</td>
</tr>
<tr>
<td>6.14</td>
<td>Generate SAS ETL code Screen</td>
<td>74</td>
</tr>
<tr>
<td>6.15</td>
<td>Block diagram of ETL requirement to classify customers</td>
<td>75</td>
</tr>
<tr>
<td>6.16</td>
<td>Block diagram of ETL requirement to combine part-supplier data</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Data warehouses allow enterprises to store relevant business information for analysis purposes. The systems that hold and manage data for analysis and decision making are called Online Analytical Processing (OLAP) Systems. OLAP systems hold a huge volume of data that is mostly summarized. OLAP systems are different from traditional database management systems, generally referred to as Online Transaction Processing (OLTP) systems, in the kind of data held and the nature of the operations performed. While OLTP systems are used to manage highly normalized data that facilitate transaction processing, OLAP systems hold summarized, historical data for use in analysis and decision making in business. This basic difference in nature and purpose of the data makes it difficult to use traditional database management tools to store and query OLAP data.

OLAP systems use a specialized set of tools to extract, process, and load data, as well as to generate reports. Since data for data warehouses typically come from OLTP systems, it has to be processed to conform to the requirements of the OLAP system. The tools for acquiring, manipulating, and storing OLAP data are collectively referred to as a data warehouse system. The application programs used to acquire data from heterogeneous sources, and process and load it into the data warehouse are called Extract, Transform, Load (ETL) tools. Examples of ETL tools are Informatica Powercenter 8 [IP08], Ab Initio [A02], IBM InfoSphere Datastage [D08] and Base SAS [S05]. The data retrieval applications for analysis are called reporting tools. IBM Cognos Powerplay [CP08] and SAP Business Objects [BO07] are some examples of commercially available reporting tools. A good data warehouse design can reduce the complexity of the reporting processes, but this requires substantial effort in the ETL process design and implementation.

ETL programming is one of the most challenging aspects of a data warehouse system. An ETL tool acts as an interface to convert the source data coming from OLTP systems to conform to the logical model of the data warehouse. This involves cleaning the source data to eliminate spurious data values, transforming the data to suit the data model of the data warehouse, and loading it into the data warehouse. A current practice for developing ETL programs is to specify the details of the ETL processing in natural language text and then a programmer converts these to ETL programs manually.

The time and complexity involved in the documentation to program conversion processes can be reduced if the ETL processing requirements are specified in a user-friendly logical model and the model is
transformed to the target ETL tool format that can then be refined by the ETL programmer. Orchid [DHWRZ08] is a proprietary software system that represents schema mapping and ETL jobs in a common internal representation. Orchid can convert schema mapping to ETL jobs and vice versa for IBM’s ETL tool InfoSphere Datastage. Simitsis and Vassiliadis [SV08] propose a logical model for ETL processes and a method to covert an ETL conceptual model to the logical model. The conceptual model [PSS02] has a template for frequently used ETL activities and can represent relationships between attributes and activities. The conceptual model can be employed in the early stages of ETL design. The logical model is used to capture detailed semantics of the ETL design including relationships between the attributes of the data flows. Vassiliadis and Simitsis [VS05] propose an internal representation to capture the logical design of ETL processes in a declarative programming language, LDL. The internal representation captures the ETL activities and relationships, but the use of declarative programming makes it less readable for business users. Munoz et al. [MMT09] proposes a platform-independent conceptual model for ETL processes and platform specific logical models for each ETL platform. The users are required to enter ETL requirements in UML Activity Diagrams and transformation rules associated with each ETL platform can be applied to convert the conceptual model to ETL programs. This stipulates that the business users be familiar with UML activity diagram notations which is not often the case.

We propose an application to input ETL requirements and an internal representation of the logical model in a well-known and widely used format that can interface to any ETL tool. The next section discusses a sample ETL scenario and our approach to modeling and transforming it to an ETL process.

1.1 Motivating Example
As an example ETL scenario, let us consider the sales data of a videogame product firm. The sales information is maintained in an OLTP system to keep account of the sales for the stores and to process exchanges or returns. Hence the OLTP system is designed to give a quick response time to insert, delete, update, and access individual transactions. Also, the OLTP system does not maintain data beyond a few months, while analyzing previous years’ sales provides valuable insights to customer behavior. Owing to the volume of transactions per minute and geographical location of stores, the sales data is stored in two different database servers in tables `game_sales_asia` and `game_sales_europe`. Querying these tables to summarize data for business analysis will slow down the system substantially. Figure 1.1 gives the block diagram representation of the processing required to convert the OLTP source data to conform to the analysis requirements. The next paragraph describes the analysis requirements of the data warehouse and summarizes the data processing required to achieve this.
Figure 1.1 Block diagram of an example ETL process

Figure 1.1 gives the block diagram of the ETL processing to convert the data in OLTP source tables to conform to the table structure of the data warehouse tables. The blocks on left side of the diagram represent the source tables in the OLTP system. Source tables game_sales_asia and game_sales_europe have identical structure and hold the sales records corresponding to the respective geographical regions. The source table exchange_rate holds the exchange rate of various currencies against the US dollar (USD). The source table store holds the location and manager information for each store.

The two blocks on the rightmost side of the diagram represent the tables game_sales_asia and game_sales_europe, referred to as fact tables in the data warehouse. A fact table in a data warehouse holds information about the attributes in an OLTP transaction such as price of an item, quantity, and discount. In a data warehouse, calculations such as sum and average are applied on these attributes, referred to as measures, to analyze the performance of a business. Though individual transaction details are not required for analysis, occasional reporting requirements necessitate storing the data in the table sales_detail_fact with detailed transaction information. The table sales_fact holds data for regular reporting. The sales offers and promotions are applied depending on a city to city basis; hence the reporting requirement is mostly to view the sales in each city. To achieve this, the data in the fact table sales_fact is stored as the sales volume per city, state or country, in a day. This also shrinks the size of the frequently queried fact table substantially compared to storing individual transactions, as the summarized table contains only one row per store per product per day. All blocks between the source and target represent operations performed on the data.
The analysis processing of the data in *game_sales_asia* and *game_sales_europe* is identical and so they are combined, indicated by the operator block `union`. Sometimes there are promotional offers in which a trial version of a game is given with the purchase of some product. These transaction entries identified by a zero amount sale are not required in the fact table and are to be filtered out, as given in the block `filter out dummy records`. The output tables *sales_fact* and *sales_detail_fact* contain *amount* in USD while the transaction may be in a foreign currency, so all amounts have to be converted by joining it with the table *exchange_rate*. This operation is indicated by the block `join on currency`. A copy of this data is written to the table *sales_detail_fact*. The splitting of the data into two flows is given by block `copy input`. To get the location of the store, the ETL flow is joined with an OLTP table *store* that contains information about store location. This data is sorted and aggregated based on *gamecode*, *city* and *date*, and the resulting summarized output is loaded in table *sales_fact*. This example is used in Chapters 3, 4 and 5 to explain our approach, and to illustrate logical modeling and conversion to an ETL process.

Our work proposes a logical model that allows business users and programmers to specify the data flow from source to target along with transformations using Extensible Markup Language [X966] and to generate ETL processes from it. Extensible Markup Language (XML) is a markup language that allows users to define their own tags to store and transfer data. XML is widely used to represent data and is easy to understand for technical and business users. Specifying operations with XML tags gives a common representation for ETL activities that is simple and powerful enough for both business and technical users. The logical representation can be used to semi-automatically generate ETL programs. The business users who enter the ETL requirements are not expected to know the program constructs of the target platform, and hence having a one-to-one correspondence between each function in the target ETL platform and the XML model is not required. To keep the model simple, we allow the users the flexibility to enter the requirements in plain text for complex operations. When the ETL code is generated, these operations are left as comments that can be reviewed by an ETL programmer to complete the program. The logical model can be converted to ETL processes by creating a mapping for each ETL tool. In our work, we illustrate the logical model to ETL process translation by implementing a dictionary-based approach to convert the logical model to Base SAS [S05] programs.

**1.2 General Research Objective**

Our work presents a logical model for ETL processes and a mapping algorithm that semi-automatically generates ETL programs from the model.

**1.3 Specific Research Objectives**

The implementation of a logical model requires the following steps:
A. Identify research problems in ETL logical modeling.
B. Identify the operations to be represented in the logical model.
C. Create a model to represent operations of an ETL process.
D. Implement an algorithm to generate an ETL process from the logical model.

1.4 Research Methodology
In order to achieve the above goals, we do the following steps:
A. Review the literature on ETL processing and modeling.
B. Identify the operators to be modeled in the following two steps:
   i. Survey current ETL tools and study their operators.
   ii. Identify a set of components that can capture the functionality of the operators provided by ETL tools.
C. Define the logical model equivalent to each component.
D. Convert the logical model to an ETL process in the following steps:
   i. Select a model management approach that can be used to generate ETL programs from the logical model.
   ii. Choose an ETL platform to generate the ETL processes.
   iii. Define the models and model mapping algorithm.
   iv. Write the code generating procedure to convert the logical model to ETL process leveraging the model mapping approach.

1.5 Contributions of the Research
We identify core differences between OLTP and OLAP systems and identify the need for separate tools for OLAP data processing. The expected contributions of the research are to identify a logical representation of the activities done in ETL processing tools and implement a programmatic solution leveraging a model management approach to convert the logical model to ETL processes. The definition of a logical model for ETL processes provides a common representation for ETL activities. To achieve this, we identify a set of ETL operators that can represent typical activities in an ETL process. The grammar rules to represent these operators have to be defined. Once the logical model is defined, we propose to semi-automate the process of creating SAS programs from the logical model. The programming constructs in the target ETL platform that are very specific to that language like statistical functions are not converted to the equivalent target functions to keep the model simple. This gives the logical model more flexibility in specifying complex data manipulation functions, which are converted to comments in the target language that can be modified by an experienced programmer. Our research
provides the foundation for generating code for ETL processes across ETL tools from the common logical representation.

1.6 Overview of the Thesis

Chapter 2 discusses the concepts of ETL logical modeling and analyzes some logical modeling approaches proposed in the literature. Steps involved in modeling and converting the processes to a target ETL platform are also discussed. Chapter 3 identifies frequently used operators in ETL processes and reviews the current research approaches to model the operators. Chapter 4 discusses our approach to creating a logical model for ETL processes. The proposed logical model for each of the operators identified in the previous chapter is defined and data flow through operators is represented as a mapping between operators. Chapter 5 explains traversing the logical model for code generation and presents the algorithm to generate ETL code. Chapter 6 gives a case study of creating a logical process in XML and generating a target SAS process for an example ETL scenario. Chapter 7 offers conclusions and future work.
Chapter 2: ETL Logical Modeling

This chapter discusses the concepts of ETL modeling and processing in current research. Section 2.1 gives an overview of ETL logical modeling. Section 2.2 discusses the steps involved in ETL logical modeling and converting a logical representation to an ETL process for a specific tool. Sections 2.3 and 2.4 discuss the research literature on ETL logical modeling offers a comparison between approaches.

2.1 Overview of ETL Modeling

Extraction, transformation and loading consume approximately 70% of the resources required to implement and maintain a data warehouse [KC04]. The data for a data warehouse comes from the transaction data generated over a period of time in OLTP systems. But the OLTP systems cannot directly populate a data warehouse because the nature and purpose of a data warehouse is quite different from that of an OLTP system. This arises from the fact that the data elements of interest during a transaction will not be the same as the data elements of interest from a business analysis point of view. Also, the OLTP operations are critical for daily business, so it cannot be slowed down by querying and summarizing a large number of records. Table 2.1 summarizes the difference between a data warehouse and an OLTP system.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Data warehouse</th>
<th>OLTP system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Facilitate business decision making</td>
<td>Facilitate business operations</td>
</tr>
<tr>
<td>Nature of data stored</td>
<td>Summarized</td>
<td>At the level of detail of the operation performed</td>
</tr>
<tr>
<td>Volume of data stored</td>
<td>Huge, in the range of terabytes</td>
<td>Much less compared to data warehouses</td>
</tr>
<tr>
<td>Users</td>
<td>Business analysts, managers</td>
<td>Anyone who uses the system to do a transaction or request information.</td>
</tr>
<tr>
<td>Sources</td>
<td>Heterogeneous, can include anything from text files and relational databases to world wide web</td>
<td>Mostly user inputs and data in the same DBMS</td>
</tr>
<tr>
<td>Operations performed</td>
<td>Mostly querying, less frequent inserts, rare deletions and updates</td>
<td>Insert, delete, update, select</td>
</tr>
<tr>
<td>Level of normalization</td>
<td>Generally denormalized</td>
<td>Normalized to at least 3NF</td>
</tr>
<tr>
<td>Query response time</td>
<td>A few seconds to hours</td>
<td>Crucial to have response time in the order of seconds</td>
</tr>
</tbody>
</table>

Table 2.1 Comparison of features of OLTP and OLAP systems

An ETL process acts as an interface between the transaction data from OLTP systems and the data warehouse. Data requirements for the data warehouse are identified and documented by a business
The requirements are analyzed by ETL professionals and enriched with the technical details of the source and target data models. This source and target information and the operations that are to be performed on the source data to conform to the target requirement constitute the ETL logical model. Logical modeling and ETL programming are a continuous process as the source data model and the target data requirements are subject to change over a period of time. In our work, semi-automating the conversion from logical model to ETL process is expected to reduce the time spent on programming every time there is a major data model change, which necessitates the creation of new ETL processes.

2.2 Research Literature on ETL Logical Modeling

In this section we review the research literature on logical modeling of ETL processes. The review focuses on user-interfaces, internal representation, and functionality and interoperability with ETL tools. We discuss three different approaches to model ETL logical processes in the research community.

Deßloch et al. [DHWRZ08] propose a tool called Orchid that implements a model to integrate ETL processes and schema mappings. Orchid follows a three-layer approach in which there is an external layer, intermediate layer, and abstract layer to represent mappings and ETL processes. The external layer represents the ETL and mapping systems in a product-specific manner. In the intermediate layer, the semantics of the ETL process are captured with emphasis on the ETL operations relevant to Orchid. This layer is product specific, hence there is a separate intermediate representation for each platform supported by Orchid. The abstract layer, called the Operator Hub Model (OHM) is common to both ETL systems and mapping systems as well as for all platforms supported. The OHM is based on relational algebra and is represented graphically as a directed graph of abstract operator nodes.

Orchid is a part of the Datastage ETL system in IBM and hence is not available for modification in the research community. Orchid supports the logical modeling and ETL generation for IBM ETL and mapping tools only. The OHM is a customized implementation of relational algebra while our work uses XML as the common representation format. Since the external and intermediate layers are platform-dependent in Orchid, two separate implementations are required for each platform supported by Orchid. OHM has a graphical interface and an internal representation while the development of the graphical interface and conversion from that to XML is not covered in our work. However, the graphical interface and logical mapping can be developed as the same tool in our work and hence eliminates the need for separate mapping tools. Both Orchid and our work do not cover the operations involved in data cleaning.
Simitsis and Vassiliadis [VS05] propose a framework for ETL logical modeling. The framework is represented in three layers and uses LDL to describe the semantics of the processes. The meta-model layer provides a single metaclass for all ETL activities. The template layer contains specializations of the meta-model class. The schema layer captures all of the information of the ETL scenario. A template for ETL activities is introduced and a graphical interface for the same is implemented in a tool called ARKTOS II. Each ETL activity is defined by a *name*, *parameter list*, *expression* and *mapping*. A *name* is a unique identifier for the activity. A *parameter list* is used to assign values to constants and specify dynamic mappings and conditions. An *expression* is a declarative statement that specifies the operations performed by the ETL activity. A *mapping* is the binding of input attributes to output attributes with possible modifications such as adding calculated columns and adding and dropping attributes. The definition of the templates for ETL activities and readability is enhanced by the use of macros to represent a group of input, output or parameter attributes.

While the graphical interface of ARKTOS II eases the logical modeling process, the internal representation does not operate with any of the ETL tools available. The logical model in our approach is converted to SAS code semi-automatically as an example to demonstrate its interoperability with ETL tools. Moreover, ARKTOS II captures the syntactic complexities of ETL processes while our model uses simple XML tags to represent everything from data types to the most complicated ETL activities. This eliminates the need to learn any new programming language separately for logical modeling. Similar to our approach, incremental loading of the data warehouse left as a separate task.

Jörg and Deßloch [JD08] discuss a method for generating ETL processes for incremental loading. The approach is to rewrite the OHM instances discussed in Orchid [DHWRZ08] such that they capture the semantics of incremental loading. The paper proposes a model for changed data as:

\[
D_{\text{new}} = D_{\text{old}} \uplus \Delta D \uplus \oplus D \ominus \nabla D \ominus \Delta D
\]

where \(D_{\text{old}}\) is the old dataset, \(D_{\text{new}}\) is the new dataset, \(\Delta D\) denotes the tuples that have been inserted, \(\nabla D\) denotes the tuples that have been deleted, \(\oplus D\) denotes the old data in the updated tuples, \(\ominus D\) the new data in the updated tuples and \(\ominus\) and \(\uplus\) are the multiset difference and union operators, respectively. The procedure for converting the incremental OHM instances to ETL processes is not discussed. The incremental variant of the aggregation operator is not discussed.

The model for changed data is based on the assumption that the changed data is identified at the source. In practice, this is just one of the approaches for incremental loading and not always used because of the
overhead in identifying changed data at the source. Another approach is to maintain a data table or file or records that are expected to change in the next load and input records are compared against it after all ETL operations. ETL platforms and target databases offer efficient programming constructs to compare against the target table to identify records as inserts or updates. The incremental OHM instances are not useful in these scenarios as the changed data is determined after all ETL operations. Our work does not generate incremental processes automatically from existing non-incremental ETL requirements.

The next section offers a summary of the features discussed here.

### 2.3 Feature Summary and Comparison of ETL Logical Modeling Tools

In this section we compare our research work with other ETL logical modeling tools and approaches. The evaluation of the approaches is based on the following parameters.

1. The internal representation of the approach (widely used).
2. Interoperability with ETL tools.
3. The provision of a graphical editor to represent the logical model.
4. The provision to convert from the ETL process to logical model.
5. The provision to generate incremental ETL processes.
6. The ability to express data cleaning operations.

Table 2.2 summarizes these features using the following notations: ○: partial compliance, ●: full compliance, and -: non-compliance.

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>Widely used</th>
<th>ETL tool interface</th>
<th>Graphical editor</th>
<th>ETL process to logical model conversion</th>
<th>Incremental processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orchid</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>-</td>
</tr>
<tr>
<td>ARKTOS II</td>
<td>○</td>
<td>-</td>
<td>●</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Jörg and Deßloch</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>●</td>
</tr>
<tr>
<td>Our approach</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 2.2 Comparison of different approaches to ETL logical modeling**

Table 2.2 gives a comparison of the features of the different approaches to ETL logical modeling. The development of a graphical interface for logical modeling is beyond the scope of our work while Orchid and ARKTOS II have this facility. Our approach and Orchid can automatically generate ETL processes.
from the logical model. Our work generates SAS ETL processes from the logical model while Orchid generates Datastage processes from the logical model. Orchid is part of a larger system of IBM’s business intelligence tools and hence is not available for logical modeling independent of these tools. XML has the advantage of being widely used for information exchange on the web and it is more accessible to a wider audience compared to the internal representations of Orchid and ARKTOS II. Jörg and Deßloch [JD08] propose a method to generate incremental ETL processes from existing ones, but it does not discuss how to convert this representation to any ETL programming language. Data cleaning is beyond the scope of all of the above approaches.

In summary, a few of the previous approaches provide the facility for a graphical editor for logical modeling while our approach is focused on an internal representation of a logical model in a well-known and widely used format that can interface to any ETL tool. Our research implements an interface to convert this logical model to ETL processes. A possible future extension to our work is to implement a graphical representation of logical model and develop incremental variants of the ETL processing components.

2.4 Conclusion

This chapter gives an outline of ETL logical modeling. The previous approaches to logical modeling in comparison to our work are discussed and a summary of these approaches is given. In this chapter we give a summary of the steps in ETL logical modeling and generating ETL processes. The next chapter discusses methodologies to convert each ETL activity to its logical model equivalent.
Chapter 3: Identifying Basic ETL Operators

This chapter discusses the principles of ETL logical modeling. We identify a set of activities that are frequently used in ETL and the operators that perform these activities. The functionality and parameters required for each of operator is defined. Section 3.1 discusses some of the commonly used ETL activities. Section 3.2 describes the operators used for these activities in detail and the current research approach to model them. Conclusions are offered in Section 3.3.

3.1 ETL Activities

An ETL activity is a set of operations on one or more data inputs. Some activities have functions associated with them that manipulate the data values while others select, project, combine, or order data. There can be a set of parameters associated with the activity depending on the operation it performs. The activities are broadly classified based on the operations they perform. ETL tools generally allow more flexibility in this classification by overloading an operator with more than one activity or providing multiple operators for the same activity. Our discussion is limited to a minimal set of activities that perform a single operation.

One of the most frequent ETL activities is filtering. An incoming flow of data records is validated based on certain criteria and records that do not match the criteria are filtered out. Another activity is join, in which multiple flows of data are combined to give a single flow based on matching column values. An ETL join is identical to a SQL join. Changing column values, adding calculated columns, applying mathematical functions on data, changing the format of the data, grouping records based on column values and generating a single record per group, performing a union on multiple flows, and sorting and copying data to multiple flows are some other ETL activities. An ETL operator is a logical abstraction of the activities, with a name and set of parameters associated with each activity. The next section discusses some of the commonly used ETL operators, the input parameters, and examples.

3.2 ETL Operators

This section gives an overview of the commonly used ETL operators, the input parameters, if any, and examples for each. The logical modeling approach for the operators in Orchid [DHWRZ08] and ARKTOS II [VS05] are also discussed. ARKTOS II uses a declarative programming language, LDL, to define the operators. The LDL notations used to represent the operators are explained with the operator definitions every time a new notation is used.
3.2.1 Filter

A filter operator takes a logical expression that evaluates to true or false for each record in an input flow of data. The output of this operation is only those records that satisfy the filter condition. The filter condition is a mandatory parameter for the filter operator. An example filter operation is depicted in Figure 3.1.

<table>
<thead>
<tr>
<th>Game Code</th>
<th>Store Code</th>
<th>Currency</th>
<th>Amount</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA12</td>
<td>K999</td>
<td>EUR</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>EA33</td>
<td>K999</td>
<td>EUR</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EA21</td>
<td>P022</td>
<td>VND</td>
<td>92500</td>
<td>1</td>
</tr>
<tr>
<td>EA21</td>
<td>P023</td>
<td>VND</td>
<td>98900</td>
<td>1</td>
</tr>
<tr>
<td>EA33</td>
<td>P023</td>
<td>VND</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EA07</td>
<td>C129</td>
<td>CLP</td>
<td>5000</td>
<td>1</td>
</tr>
</tbody>
</table>

![FILTER CONDITION: Amount != 0](image)

Each record in the input data is validated against the filter condition, $\text{Amount} \neq 0$ and the records that evaluate to false are rejected. For example, the first record has a value 15 for amount that evaluates to true for the filter expression. Hence, this record is carried to the output. The value of amount for the second record is 0 and so the condition check evaluates to false. Hence this record is rejected.

In Orchid [DHWRZ08], a filter activity is represented using an OHM filter operator. The ETL tool Orchid utilizes, Datastage, has a filter activity, referred to as a filter stage, that has more functionalities than the basic filter operation including splitting the input into multiple output flows and projecting columns. Splitting the input into multiple data flows is achieved with the OHM operator `split_following` the input data. Column projection facility in the filter stage is represented as a filter followed by a `basic-project` operator in OHM. Hence OHM has a combination of operators to represent a FILTER stage in Datastage. Figure 3.2 shows the OHM representation of the FILTER stage in Datastage.
The Filter stage in Datastage allows copying the input into multiple flows and testing each flow for a different condition. Hence a Filter stage can have more than one output flow. An OHM Split operator is used to copy the input into multiple flows. The columns in each flow can be projected using a Basic Project. The filter conditions associated with each of the FILTER(i) operators is stored in array effectiveWhere[] and projected columns corresponding to each BASIC PROJECT(i) operator for the flows are stored in a two dimensional array derivations[][] . Each Basic Project operator needs an array to store the projected columns hence derivations is a two-dimensional array. The Filter stage also has a rowOnlyOnce mode operation when there are multiple flows within the same Filter stage. If it is set to true, the filter conditions are evaluated serially in the order of the output flows specified using the integer pos. If a row is evaluated to true for one condition, it will not be processed in the subsequent flows.

In ARKTOS II [VS05], ETL operators are represented as design templates in LDL. A design template is an LDL construct for frequently used ETL operators. A filter operator in ARKTOS II is divided into six types: (1) selection, (2) not null, (3) primary key violation, (4) foreign key violation, (5) unique value, and (6) domain mismatch. Since all of these select a set of rows from a given input, we discuss only the selection filter operator. The representation for the selection template in ARKTOS II is given below.

**Selection**

```latex
\begin{align*}
  &\ a_{\text{out}}(\text{OUTPUT\_SCHEMA}) \leftarrow \\
  &\ a_{\text{in}}(\text{INPUT\_SCHEMA}), \\
  &\ \text{expr}(\text{PARAMS}), \\
  &\ \text{DEFAULT\_MAPPING} \end{align*}
```

where OUTPUT\_SCHEMA, INPUT\_SCHEMA, PARAMS and DEFAULT\_MAPPING are macros defined to enhance the readability of the template. OUTPUT\_SCHEMA and INPUT\_SCHEMA macros
expand to give the full definition of the output and input attributes, respectively. PARAMS is the macro to specify the set of filter conditions. DEFAULT_MAPPING is the propagation of all the input attributes. The expansion of these macros is given below.

```
DEFINE INPUT_SCHEMA as
  [i<arityOf(a_in)] {A_IN_$i$},
  [i=arityOf(a_in)] {A_IN_$i$}
DEFINE OUTPUT_SCHEMA as
  [i<arityOf(a_out)] {A_OUT_$i$},
  [i=arityOf(a_out)] {A_OUT_$i$}
DEFINE PARAMS as
  [i<arityOf(PARAM)] {PARAM[$i$]},
  [i=arityOf(PARAM)] {PARAM[$i$]}
DEFINE DEFAULT_MAPPING as
  [i<arityOf(a_out)] {A_OUT_$i$= A_IN_$i$},
  [i=arityOf(a_out)] {A_OUT_$i$= A_IN_$i$}
```

The macros use loops to define templates with unknown number iterations. Since the number of input fields and output fields or expressions in a condition depend on the specific ETL process being modeled, the general loop template use a function arityOf() to fetch the number of fields in the input/output or expressions in the condition. A variable beginning with the @ symbol is used to denote user-supplied values. A $i$ symbol is appended to a variable, say, A_OUT_, to create a series of variable names such as A_OUT_1 and A_OUT_2, where the value of i is known only at runtime. For the example in Figure 3.1, the arityOf function returns 5 as there are five input fields. Hence the macro INPUT_SCHEMA can be written as:

```
DEFINE INPUT_SCHEMA as
  [i<5] {A_IN_$i$},
  [i=5] {A_IN_$i$}
```

This macro produces five variables by appending the value of the loop variable i to A_IN_. The above macro definition for INPUT_SCHEMA for 5 input fields expands to:

A_IN_1, A_IN_2, A_IN_3, A_IN_4 and A_IN_5.

These variables are then assigned the input field names; hence the macro INPUT_SCHEMA for the example in Figure 3.1 represents:
GAME_CODE, STORE_CODE, CURRENCY, AMOUNT, QUANTITY.

Proceeding this way we get the output schema and list of filter conditions from macros OUTPUT_SCHEMA and PARAMS, respectively. The macro definition for DEFAULT_MAPPING is first expanded to fetch the value of arityOf function, which is equal to 5 for a_out. The macro definition after this step is given below.

```
DEFINE DEFAULT_MAPPING as
    [i<5] {A_OUT_$i$= A_IN_$i$},
    [i=5] {A_OUT_$i$= A_IN_$i$}
```

This produces five assignment statements to map the input fields to the output fields directly as given below.

```
A_OUT_1 = A_IN_1
A_OUT_2 = A_IN_2
A_OUT_3 = A_IN_3
A_OUT_4 = A_IN_4
A_OUT_5 = A_IN_5
```

Since each of these variables is assigned the field name of the input and output, the macro DEFAULT_MAPPING in Figure 3.1 represents:

```
GAME_CODE = GAME_CODE
STORE_CODE = STORE_CODE
CURRENCY = CURRENCY
AMOUNT = AMOUNT
QUANTITY = QUANTITY
```

With these macro definitions, the example in Figure 3.1 can be expanded as below.

```
a_out (GAME_CODE, STORE_CODE, CURRENCY, AMOUNT, QUANTITY) ←
a_in(GAME_CODE, STORE_CODE, CURRENCY, AMOUNT, QUANTITY),
expr(AMOUNT!=0),
GAME_CODE = GAME_CODE
STORE_CODE = STORE_CODE
CURRENCY = CURRENCY
AMOUNT = AMOUNT
QUANTITY = QUANTITY.
```

In this filter instance, a_out and a_in represent the output and input schemas, respectively, in Figure 3.1 and expr represents the list of condition expressions, which is amount!=0 in this case. The mapping from the input fields to output fields is given as a set of assignments with output field name on the left side and
input field name on the right. Since the field names or mapping do not change in a filter operation, the field names do not change from input to output.

### 3.2.2 Join

A join operator is similar to a SQL join, with options to do outer, inner, and left/right outer joins. The join operator in many ETL tools allows a user to specify if the incoming data is sorted on the join key or not. This option reduces the execution time if the data flow is already sorted. Most ETL joins do not allow a Cartesian product as a join key is a mandatory parameter for an ETL join. The join type is an optional parameter, which is an inner join by default. Figure 3.2 shows a sample join operation.

![Sample join operation diagram](image)

The input field `currency` is the key for this join operation. The first row in the `game_sales` input has a `currency` value `EUR`. To join this with the `exchange_rate` input, we search the exchange rate table for a row with `currency` value `EUR`. Since there is a match, a single row with all the fields from `game_sales` and `exchange_rate` is passed to the output. The second row in the `game_sales` input has a `currency` value...
To join this with the *exchange_rate* input, we search the exchange rate table for a row with *currency* value *ABC*. Since there is no match, this row does not appear in the output.

Since a Data stage join operator can take multiple input flows and allows the projection of columns, its corresponding OHM representation has multiple inputs and a JOIN operator followed by an optional BASIC PROJECT operator. The generic representation of join operator in OHM [DHWRZ08] is given in Figure 3.4.

![Figure 3.4 Join stage representation in OHM [DHWRZ08]](image)

ARKTOS II [VS05] identifies join as an ETL activity, but its design template is not discussed.

### 3.2.3 Format

A format operator allows changing the structure of the data flow by adding calculated columns. It also allows modifying data values and dropping columns. Many of the facilities provided by the format operator are sometimes offered in other operators that modify column values. For example, ETL tools may allow creating calculated columns and modifying column values in a join operator. An ETL format operator may also function as a filter, allowing users to filter data records and manipulate them in the same operator. Figure 3.3 gives a sample format operation.
This format operation updates the value of *amount* as *amount*\**exchange_rate* and drops the field *exchange_rate* from the output data flow. For example, the first row in the input has value 15 for *amount* and 1.41070 for *exchange_rate*. The value for *amount* in the output record is 21.15, the product of 15 and 1.41070. Also, the field *exchange_rate* does not appear in the output record.

An operator similar to format discussed in Orchid [DHWRZ08] is transform. A transform operator takes a subset of the input rows and columns. It is not mentioned if transform allows modifying data values. Another similar operator in Orchid is modify, which allows dropping columns. The OHM representation of transform stage is given below.
The transform stage allows optionally splitting the input into multiple copies and applying a selection condition on each flow. Hence the OHM representation for transform stage has the input followed by optional split and select operators, followed by project operator. The modify stage is a special case of the transform stage and it allows only to project columns from the input. The OHM representation of modify stage is given below.

![Diagram of Transform Stage Representation in OHM](image)

**Figure 3.6 Transform stage representation in OHM [DHWRZ08]**

The operators similar to format in ARKTOS II [VS05] are projection and function application. Projection allows projecting out columns from input. Function application allows adding calculated columns to the data field applying a set of pre-defined functions on one or more input fields. The representation for each of these is given below.

**Function Application**

\[
a_{\text{out}}(\text{OUTPUT_SCHEMA},@\text{OUTFIELD}) \leftarrow \\
a_{\text{in}}(\text{INPUT_SCHEMA}),
\]

\[
@\text{FUNCTION} \ (\text{PARAMS}, \ @\text{FunOutFIELD}),
\]

\[
@\text{OUTFIELD}=\@\text{FunOutFIELD},
\]

DEFAULT_MAPPING

where OUTPUT_SCHEMA and INPUT_SCHEMA are the macros for input and output schema definition. @FUNCTION is a user instantiated function applied to the columns specified in PARAMS macro and the function value is assigned to FunOutFIELD. DEFAULT_MAPPING is the propagation of all the input attributes. The output schema is propagation of all input attributes and the calculated column. For the example in Figure 3.5, if we consider the operation \(\text{amount} \times \text{exchange\_rate}\) as a function applied to field amount, function application can be expanded as follows.
a_out(GAME_CODE, STORE_CODE, CURRENCY, AMOUNT, QUANTITY, EXCHANGE_RATE)
\arrowleft a_in(GAME_CODE, STORE_CODE, CURRENCY, AMOUNT, QUANTITY, EXCHANGE_RATE),
@FUNCTION (amount*exchange_rate, AMOUNT ),
AMOUNT = AMOUNT,
DEFAULT_MAPPING.

Since the field amount is already there in the input, the output schema is the same as the input schema in this example. Function application cannot be used to drop columns; hence the operation in Figure 3.5 cannot be completed with function application alone. The projection design template in ARKTOS II [VS05] is given below.

**Projection**

a_out(OUTPUT_SCHEMA)\arrowleft

a_in1(INPUT_SCHEMA),
PROJECT_MAPPING

DEFINE PROJECT_MAPPING as

[i<arityOf(@PROJECTED_FIELDS)] {A_OUT_$i$= @PROJECTED_FIELDS[$i$],}
[i=arityOf(@PROJECTED_FIELDS)] {A_OUT_$i$= @PROJECTED_FIELDS[$i$]}

where OUTPUT_SCHEMA and INPUT_SCHEMA are the macros for input and output schema definition. PROJECT_MAPPING is the propagation of only those attributes specified by the user. The \texttt{arityOf} function returns 5 as the user inputs five columns to be projected to the output. The PROJECT_MAPPING macro expanded with the above information is given below.

DEFINE PROJECT_MAPPING as

[i<5] {A_OUT_$i$= @PROJECTED_FIELDS[$i$]},
[i=5] {A_OUT_$i$= @PROJECTED_FIELDS[$i$]}

The loop iterates five times to assign the output field names as the projected columns specified in the user input. The expanded PROJECT_MAPPING macro is given below.

GAME_CODE = GAME_CODE
STORE_CODE = STORE_CODE
CURRENCY = CURRENCY
AMOUNT = AMOUNT
QUANTITY = QUANTITY

The projection operation for dropping the field \texttt{exchange_rate} expanded with the definition for PROJECT_MAPPING is given below.
\[
\text{a\_out(GAME\_CODE, STORE\_CODE, CURRENCY, AMOUNT, QUANTITY)\leftrightarrow}
\]
\[
\text{a\_in(GAME\_CODE, STORE\_CODE, CURRENCY, AMOUNT, QUANTITY, EXCHANGE\_RATE),}
\]
\[
\text{GAME\_CODE = GAME\_CODE}
\]
\[
\text{STORE\_CODE = STORE\_CODE}
\]
\[
\text{CURRENCY = CURRENCY}
\]
\[
\text{AMOUNT = AMOUNT}
\]
\[
\text{QUANTITY = QUANTITY}
\]

Here \( a\_out \) and \( a\_in \) have the output and input schemas, respectively. Since \( exchange\_rate \) is dropped from the output schema, it is not listed in \( a\_out \) or the source to target mapping.

### 3.2.4 Aggregate

The aggregate operator gives summarized data records for a group of records. The records are grouped based on the aggregation key and a single record per group is output. Some ETL tools allow specifying an expression as an aggregation key. The input is required to be sorted on the aggregation key or else a sort on the aggregation key is done before aggregation. Hence a sort parameter is required to indicate if the data is already sorted. Figure 3.4 gives an example illustration of an aggregate operator.

![Figure 3.8 Example for aggregate operator](image)

The aggregation keys for this example are \( city \) and \( gamecode \). Each unique combination of \( city \) and \( gamecode \) in the input is treated as a group and a single output row is generated for each group. For fields \( amount \) and \( quantity \), the values in the group are added to produce the output. Consider the first row in the input. There is only one record in the input for \( city\_gamecode \) combination, \( PARIS-EA12 \), hence the aggregation operation does not change the record and it is passed as is to the output. For the second unique \( city\_gamecode \) combination, \( EA21-HANOI \), there are two input records, hence the value of \( amount \) in the output row is the sum of \( amount \) in the second and third input rows and the value of \( quantity \) in the output row is the sum of \( quantity \) in the second and third input rows.
The aggregator stage in Datastage is represented in OHM as given below. The OHM group operator is similar to the relational group operator and performs aggregation and duplicate elimination on a given field. The Datastage aggregator also allows dropping columns hence the basic project operator is included optionally. The OHM representation for Datastage aggregator is given below.

![Diagram](image.png)

**Figure 3.9 Aggregator stage representation in OHM [DHWRZ08]**

The design template for the aggregate operator in ARKTOS II [VS05] is given below.

**Aggregation**

```plaintext
a_out(ALL_GROUPERS, @AggrFunction<@Field>) ←
  a_in(INPUT_SCHEMA)
  DEFINE ALL_GROUPERS AS
    [i<arityOf(@GROUPERS)] {@GROUPERS[$i$],}
    [i=arityOf(@GROUPERS)]{@GROUPERS[$i$]}
```

where `INPUT_SCHEMA` is the macro for input schema definition and `ALL_GROUPERS` is the macro for defining the group attributes. The output schema is the propagation of all group attributes and the result of the aggregation function applied on attributes specified at runtime. For the example in Figure 3.8, there are two keys for aggregation; hence the function `arityOf` returns 2. The macro `ALL_GROUPERS` is expanded as given below.

```plaintext
DEFINE ALL_GROUPERS AS
  [i<2] {@GROUPERS[$i$],}
  [i=2] {@GROUPERS[$i$]}
```

The loop iterates twice to create two grouping variables and the aggregation keys are assigned to these variables. The aggregation template instance for the example in Figure 3.8 is given below.

```plaintext
a_out(CITY, GAMECODE, SUM<AMOUNT>, SUM<QUANTITY>) ←
  a_in(CITY, GAMECODE, AMOUNT, QUANTITY)
```

Here `a_out` and `a_in` represent the input and output schemas. Since the output schema contains only the keys and aggregation functions applied to input fields, `a_out` lists `city`, `sum`, and the `sum` function applied to `amount` and `quantity`.
3.2.5 Union

A union operator combines data from two or more input flows and gives a single output flow. The input flows are expected to have identical column names and data types. Some ETL tools give the option to preserve the order of the data records in the input. A union operator has no input parameters. Figure 3.9 gives the example for a union operation.

```
<table>
<thead>
<tr>
<th>Game Code</th>
<th>Store Code</th>
<th>City</th>
<th>Amount</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA21</td>
<td>P022</td>
<td>HANOI</td>
<td>5.18</td>
<td>1</td>
</tr>
<tr>
<td>EA21</td>
<td>P023</td>
<td>HANOI</td>
<td>5.53</td>
<td>1</td>
</tr>
<tr>
<td>EA07</td>
<td>C129</td>
<td>KUWAIT</td>
<td>10.44</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Game Code</th>
<th>Store Code</th>
<th>City</th>
<th>Amount</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA10</td>
<td>T102</td>
<td>LONDON</td>
<td>30.16</td>
<td>1</td>
</tr>
<tr>
<td>EA10</td>
<td>T115</td>
<td>LONDON</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>EA12</td>
<td>K999</td>
<td>PARIS</td>
<td>21.16</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Figure 3.10 Example for union operator

The union operation appends the contents of the second input to the first input. Both the inputs have identical schema and output schema is same as the input schema. The first three records in the output come from the first input and the last three records come from the second input.

Union is a basic operator in OHM and hence its Datastage representation is not discussed in Orchid [DHWRZ08]. Even though union is classified as a binary operator in ARKTOS II [VS05] its design template is not discussed.
3.2.6 Sort

A sort operator sorts incoming data records based on a key value. Because ETL programs typically have a long execution time and ETL operators generally preserve the order of records throughout the data flow, it is not desirable to let the operators do the sorting whenever an ordered record set is required, such as in join and aggregate. If the same flow of data goes into multiple operators that need sorted data on the same key, it is better to pre-sort the data before feeding it to the operators. A sort operator serves this purpose. Figure 3.11 gives an example sort operator.

![Table Example for Sort Operator](image)

In this example, the input is sorted on ascending values of key, gamecode. A sort algorithm is executed on the records based on gamecode and the output records are ordered such that any record in the output has a lower alphabetical and numerical value for gamecode than the record below it. The particular sort algorithm used depends on the ETL application being used.

The sort operator is not discussed in Orchid [DHWRZ08]. The ARKTOS II [VS05] sort as a template activity under file operations category, but its design template is not discussed.

3.2.7 Copy

A copy operator reads incoming data records and writes it to two separate data flows. A copy operator is required when different operations are performed on the same data to create multiple output data. If the input of the copy operator comes from an operator, the data needs to be written to memory for further processing. Most ETL applications do this implicitly for each copy operator, while some others like SAS requires this to be done by the programmer. Figure 3.12 gives an example copy operator.
Figure 3.12 Example for copy operator

In this example, the input is written to memory and each downstream operator gets a copy of the input data. Copy is a basic operator in OHM and hence its Datastage representation is not discussed in Orchid [DHWRZ08]. The copy operator is not discussed in ARKTOS II [VS05].

Table 3.1 summarizes the ETL operators discussed and their implementation in Orchid [DHWRZ08] and ARKTOS II [VS05]. To denote the functionality implemented we use the following notations: ●: the operator has an implementation, and - : the operator does not have an implementation.
Table 3.1 Summary of operator implementation in Orchid [DHWRZ08] and ARKTOS II [VS05]

<table>
<thead>
<tr>
<th>ETL Operator</th>
<th>Orchid</th>
<th>ARKTOS II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Join</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Format</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Aggregate</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Union</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sort</td>
<td>-</td>
<td>●</td>
</tr>
<tr>
<td>Copy</td>
<td>●</td>
<td>-</td>
</tr>
</tbody>
</table>

Our work implements a logical model that incorporates the operators in Table 3.1 and generates the ETL code for each operator.

3.3 Conclusions

This chapter discusses a set of commonly used ETL operators. The purpose of each is explained with the help of examples. The parameters required by each are also identified. The next chapter discusses logical modeling using XML to specify the operators identified here.
Chapter 4: Logical Modeling of Operators Using XML

This chapter discusses logical modeling for ETL processes using XML. Section 4.1 gives an overview of logical modeling in a format that facilitates automated processing. Section 4.2 discusses logical modeling using XML. Section 4.3 gives the XML description of the source and target schemas involved in the ETL process. Section 4.4 discusses the XML description of the ETL operators and fields. Section 4.5 discusses the logical model for ETL operators discussed in Chapter 3. Section 4.6 offers conclusions.

4.1 Logical Modeling of ETL Processes

The layout of an ETL requirement can be deployed as a linearly serializable process [S03]. A process that is linearly serializable can be represented as a series of discrete operations. Our logical model considers an ETL process as a flow of data from one or more sources to targets through the ETL operators. The data type of the source and target are not represented in this model, as the logical modeling of the operators is independent of data types at this abstraction level. We assume the input data is read into the ETL platform and that it models the operators associated with the transformation. The source and target are identified by name and have a set of associated attributes. The size and data type of the source or target fields are not considered in the discussion as our focus is on modeling the process and not the data. Each ETL operator is associated with a name and a set of parameters. The name and parameters are represented as values of XML tags.

XML is a set of rules that enables users to define their own markup languages [G08]. XML is widely used to store and transport data. Using XML, a user can specify custom tags to structure data. The main features of XML that make it widely used are [G08]:

1. XML can store and carry information.
2. It can be easily, extended, adapted and reused.
3. Since XML files are simple text files, they can be shared between disparate systems and organizations.
4. XML is non-proprietary.

XML provides a structuring mechanism for data that makes it an ideal tool for logical modeling. XML tags can be defined to identify ETL operators and their parameters. The source and target schemas can also be specified in XML. The next section describes the XML representation of the data source and targets in an ETL process.
4.2 Describing Source and Target Schemas

This section describes the XML representation of the source schemas. Each source and target in an ETL process is identified by its name. Section 4.3.1 describes the XML representation of a data source. Section 4.3.2 describes the XML representation of a target schema.

4.2.1 Source Schema Representation

For source data, the name of the source is identified as an attribute, name, of the tag SOURCE. The input fields are described within the tag IN_SCHEMA. Each input field is specified in a tag FIELD and the field name is specified as an argument of FIELD. The operator that takes the data from the source is specified within the tag OUT. The tokens specified in italics are replaced by user defined values. The generic XML representation of a source schema is given below.

```
<SOURCE name = "schema_name">
  <OUT> operator_name </OUT>
  <IN_SCHEMA>
    <FIELD name = "field1_name"/>
    <FIELD name = "field2_name"/>
    ...
    <FIELD name = "fieldN_name"/>
  </IN_SCHEMA>
</SOURCE>
```

The XML representation of the source game_sales given in Figure 2.1 is given below.

```
<SOURCE name = "game_sales">
  <OUT> promo_items_filter </OUT>
  <IN_SCHEMA>
    <FIELD name = "game_code"/>
    <FIELD name = "store_code"/>
    <FIELD name = "currency"/>
    <FIELD name = "amount"/>
    <FIELD name = "quantity"/>
  </IN_SCHEMA>
</SOURCE>
```

4.2.2 Target Schema Representation

For target data, the name of the target is identified as an attribute, name, of the tag TARGET. The input fields are described within the tag OUT_SCHEMA. Each input field is specified in a tag FIELD and the field name is specified as an argument of FIELD. The operator that supplies data to the target is specified within the tag OUT. The tokens specified in italics are replaced by user defined values. The generic XML representation of a target schema is given below.

```
<TARGET name = "target_name">
  <OUT> operator_name </OUT>
  <OUT_SCHEMA>
    <FIELD name = "field1_name"/>
    <FIELD name = "field2_name"/>
    ...
    <FIELD name = "fieldN_name"/>
  </OUT_SCHEMA>
</TARGET>
```
The XML representation of the target `sales_fact` given in Figure 2.1 is given below.

```xml
<TARGET name = "sales_fact">
  <IN> promo_items_filter </IN>
  <OUT_SCHEMA>
    <FIELD name = "game_code" />
    <FIELD name = "city"/>
    <FIELD name = "amount"/>
    <FIELD name = "quantity"/>
    <FIELD name = "date">
  </OUT_SCHEMA>
</TARGET>
```

4.3 Describing Mappings

This section discusses the mapping description between operators and within the operators in an ETL process. Section 4.4.1 describes representing the mapping between operators in the ETL process. Section 4.4.2 describes representing the mapping between attributes of the source and target data flows within an operator.

4.3.1 Mapping between Operators

The mapping between operators is specified by tags `IN` and `OUT` within the XML representation of the operator. The name of the operator or source that feeds data to the operator is specified within the tags `<IN>` and `</IN>`. The name of the operator or target to which data flows from the operator is specified within `<OUT>` and `</OUT>`. The generic representation for a mapping specification for an operator `operator1` is given below.

```xml
<operator_type name = operator1>
  <IN> operator2 <IN>
  <OUT> operator3 <OUT>
</operator_type>
```

Operator type can be any of the operators discussed in Chapter 3. The input data for `operator1` comes from `operator2` and output data flows to `operator3`. There can be other tags within `operator_type` depending on
the type of operation it does. A detailed discussion of the tags other than IN and OUT is given in Section 4.5.

4.3.2 Mapping within Operators
Mapping within operators is specified within the XML representation of the operator by tags MAPPING and FIELD. There is a MAPPING tag associated with each operator and a FIELD tag associated with each output field. Each FIELD tag has an attribute name to specify the name of the field. The value of the FIELD tag is an expression expr that takes one or more fields as input from the input data flow. Only those operators that change the schema of the input field have a MAPPING tag. Consequently JOIN, FORMAT and AGGREGATE require a MAPPING tag and FILTER, UNION and SORT do not require it. The generic representation of a mapping specification from input to output within an operator operator1 is given below. Data is assumed to be input from operator2 and output to operator3.

```
<operator_type name = operator1>
  <IN> operator2 </IN>
  <OUT> operator3 </OUT>
  <MAPPING>
    <FIELD name = name1> expr1 </FIELD>
    <FIELD name = name2> expr2 </FIELD>
    .
    .
    <FIELD name = nameN> exprN </FIELD>
  </MAPPING>
</operator_type>
```

The detailed description of other tags in this example is given in the next section where the XML representation of the ETL operators from Chapter 3 are discussed.

4.4 Logical Modeling of ETL Operators
This section describes the XML representation of ETL operators. Each operator is identified by a name. The position of the operator in the ETL process is identified by tags IN and OUT. Other tags for operators are the parameter tags specific to the operator and the mapping tags for those operators that change the structure of the data flow.

4.4.1 Filter
A filter operator has two mandatory parameters, the filter name and condition. Since a filter does not change the structure of data, there is no mapping associated with filter. The XML representation of filter is given below.

```
<FILTER name= operator_name>
  <IN> source </IN>
  <OUT> target </OUT>
```
<COND> filter_condition </COND>
</FILTER>

The filter condition is specified within the tag \textit{COND}. The mapping action of a filter is to propagate all input attributes, hence mapping is not specified in the XML representation of Filter. The \textit{operator_name} is a parameter supplied by the user to identify the operator. It is unique for an operator within an ETL process. Spaces are not allowed in the operator name. The XML representation of the filter operation in Figure 3.1 is given below.

\begin{verbatim}
<FILTER name = "promo_items_filter">
  <IN> union_region_sales </IN>
  <OUT> join_on_currency </OUT>
  <COND> amount != 0  </COND>
</FILTER>
\end{verbatim}

Argument \textit{name} of the XML tag \textit{FILTER} has a value that is unique within the XML document. Here \textit{promo_items_filter} is a unique name to identify the operator, as there could be multiple filters within the same XML document. The tag \textit{IN} has the value of the name argument of the operator or source whose output data is input to \textit{promo_items_filter}. The tag \textit{OUT} has the value \textit{join_on_currency}, the name argument of the operator that receives input from \textit{promo_items_filter}. The tag \textit{COND} holds the condition \textit{amount! = 0} based on which the input records are filtered.

\subsection*{4.4.2 Join}

The mandatory parameter for a join operator is the join key. The optional parameters for join are join type (inner join by default) and sort type which specifies if the inputs are already sorted on the join attributes. The join key is specified by the \textit{KEY} tag. Mapping of the attributes have to be represented in join as the output schema is different from the input schema. The XML representation for join is given below.

\begin{verbatim}
<JOIN name= operator_name >
  <IN > source </IN>
  <IN > source </IN>
  <OUT> target </OUT>
  <KEY>key1 </KEY >
  <KEY>key2 </KEY >
  ...
  <KEY>keyn </KEY >
  <MAPPING>
    <FIELD name = name1 expression= expr1 </FIELD>
    <FIELD name = name2 expression =expr2 </FIELD>
    ...
    <FIELD name = nameN expression= exprN </FIELD>
  </MAPPING>
</JOIN>
\end{verbatim}
The XML representation of the join in Figure 3.2 is given below.

```xml
<JOIN name="join_on_currency">
    <IN> promo_items_filter </IN>
    <IN> exchange_rate </IN>
    <OUT> convert_to_dollar </OUT>
    <KEY> currency </KEY>
    <MAPPING>
        <FIELD name="game_code" expression="game_code"/>
        <FIELD name="store_code" expression="store_code"/>
        <FIELD name="currency" expression="currency"/>
        <FIELD name="amount" expression="amount"/>
        <FIELD name="quantity" expression="quantity"/>
        <FIELD name="exchange_rate" expression="exchange_rate"/>
    </MAPPING>
</JOIN>
```

Here *join_on_currency* is the unique name to identify the join operator within the XML document. The output of operator *promo_items_filter* and data source *exchange_rate* are the inputs to be joined and hence these are specified using two *IN* tags. Tag *OUT* has the value *convert_to_dollar*, the name of the operator that receives input from *join_on_currency*. Tag *KEY* holds the name of the column from *promo_items_filter* and data source *exchange_rate* based on which the join is performed. Since the join is based on a single column, there is only one *KEY* tag. The *MAPPING* tag has six *FIELD* tags in it as six columns are mapped to the output. Each *name* argument gives the name of the output column and *expression* argument gives the input column or expression. For example, output column *game_code* is mapped from input column *game_code* hence the first *FIELD* tag has value *game_code* for arguments *name* and *expression*.

### 4.4.3 Format

A format operator allows manipulating column values and adding and dropping columns. Both of these can be done within the *MAPPING* tag. Manipulating data values can be done within the *FIELD* tag as an expression. Dropping columns can be done by not specifying the mapping for the field. A column can be added by adding a new *FIELD* tag in the mapping and specifying its value as a constant or expression. The generic XML representation for format is given below.

```xml
<FORMAT name=operator_name >
    <IN> source </IN>
    <OUT> target </OUT>
    <MAPPING>
```
The XML representation of the format operator in Figure 2.1 is given below.

```xml
<FORMAT name="convert_to_dollar">
  <IN> join_on_currency </IN>
  <OUT> copy_input </OUT>
  <MAPPING>
    <FIELD name="game_code" expression="game_code"/>
    <FIELD name="store_code" expression="store_code"/>
    <FIELD name="currency" expression="currency"/>
    <FIELD name="amount" expression="amount*exchange_rate"/>
    <FIELD name="quantity" expression="quantity"/>
  </MAPPING>
</FORMAT>
```

Here `convert_to_dollar` is the unique name to identify the format operator within the XML document. The output of operator `join_on_currency` is the input to be formatted specified using `IN` tag. The tag `OUT` has the value `copy_input`, the name of the operator that receives input from `convert_to_dollar`. The `MAPPING` tag has five `FIELD` tags in it as five columns are mapped to the output. Each `name` argument gives the name of the output column and the value of the tag gives the input column or expression. For example, output column `game_code` is mapped from input column `game_code` hence the first `FIELD` tag has value `game_code` for name and expression. Output column `amount` is mapped from input columns `amount` and `exchange_rate` using an operation, hence its `FIELD` tag has value `amount` for argument name, and value `amount*exchange_rate` for argument expression.

### 4.4.4 Aggregate

The mandatory parameter for an aggregate operator is the aggregate key. The aggregation can be based on multiple key fields. Hence there are one or more `KEY` tags to specify the aggregation key. Mapping of the attributes have to be represented in `AGGREGATE` as the output schema is different from the input schema. Each field value other than the key field has to be a summary operation on the non-key fields. The summary operations allowed are `sum` for sum, `avg` for average, `max` for maximum, `min` for minimum and `count` for the total number of observations in that group. The XML representation for aggregate is given below.

```xml
<AGGREGATE name=operator_name>
  <IN> source </IN>
```
<OUT> target </OUT>
<KEY> key_field1 </KEY>
<KEY> key_field2 </KEY>
.
.
<KEY> key_fieldN </KEY>

<MAPPING>
  <FIELD name = "key_field1" expression = "key_field1" />
  <FIELD name = "key_field2" expression = "key_field2" />
  .
  .
  <FIELD name = "key_fieldN" expression = "key_fieldN" />
  <FIELD name = "field1" expression = "summary(field1)" />
  <FIELD name = "field2" expression = "summary(field2)" />
  .
  .
  <FIELD name = "fieldN" expression = "summary(fieldN)" />
</MAPPING>
</AGGREGATE>

The XML representation of the aggregate operator in Figure 2.1 is given below.

<AGGREGATE name= "aggregate_input">
  <IN> sort_input </IN>
  <OUT> sales_fact </OUT>
  <KEY> gamecode </KEY>
  <KEY> city </KEY>
  <KEY> date </KEY>

  <MAPPING>
    <FIELD name = "game_code" expression = "game_code"/>
    <FIELD name = "city" expression = "city"/>
    <FIELD name = "date" expression = "date"/>
    <FIELD name = "amount" expression = "sum(amount)"/>
    <FIELD name = "quantity" expression = "sum(quantity)"/>
  </MAPPING>
</AGGREGATE>

Here aggregate_input is the unique name to identify the join operator within the XML document. The output of operator sort_input is the input to this operator and is specified using IN tag. Tag OUT has the value sales_fact, as the output of the aggregate operator is written to this table. The tag KEY holds the name of the column based on which the aggregation is performed. Since the aggregation is based on three columns, there are three KEY tags. The MAPPING tag has five FIELD tags in it as five columns are mapped to the output. The key columns are mapped directly and have the same value for name and expression. For example, output column game_code is mapped from input column game_code hence the first FIELD tag has value game_code for arguments name and expression. The summarized columns have
a summary function applied to the input. For example, the mapping of column \( quantity \) has name \( quantity \) and expression \( \text{sum}(quantity) \).

### 4.4.5 Union

A union operator gathers data records of the same format from multiple sources. Since a union operator requires only the sources and target to perform the operation, the mandatory parameters are operator name and input and output information. Since there are two inputs for the union operator, there will be two \( \text{IN} \) tags. As the data format cannot be changed in this operator, the \( \text{MAPPING} \) tag is not specified in the XML representation of union. The generic XML representation for union is given below.

```xml
<UNION name= operator_name >
  <IN > source1 </IN>
  <IN > source2 </IN>
  .
  .
  <IN > sourceN </IN>
  <OUT> target </OUT>
</UNION>
```

The XML representation of the union operator in Figure 2.1 is given below.

```xml
<UNION name= "union_region_sales">  
  <IN > game_sales_asia </IN>
  <IN > game_sales_europe </IN>
  <OUT> promo_items_filter </OUT>
</UNION>
```

The union operator \( \text{union_region_sales} \) takes input from two source blocks, so there are two \( \text{IN} \) tags with value \( \text{game_sales_asia} \) and \( \text{game_sales_europe} \). The output of the operator is the input to filter operator \( \text{promo_items_filter} \), so the \( \text{OUT} \) tag has value \( \text{promo_items_filter} \).

### 4.4.6 Sort

A sort operator sorts records from a single source based on a given key. Since a sort operator requires only the source, target and sort key to perform the operation, the mandatory parameters are operator name, input and output information and key. The default sort order is ascending. The sorting can be done on multiple fields, hence there can be more than one \( \text{KEY} \) tag. As the data format cannot be changed in this operator, \( \text{MAPPING} \) tag is not specified in the XML representation of sort. The generic XML representation for sort is given below.

```xml
<SORT name= operator_name >
  <IN > source </IN>
</SORT>
```
<OUT> target </OUT>
<KEY> key_field1 </KEY>
<KEY> key_field2 </KEY>
.
.
<KEY> key_fieldN </KEY>
<SORT>

The XML representation of the example sort operator in Figure 2.1 is given below.

<SORT name=“sort_input”>
   <IN > join_on_storecode </IN>
   <OUT> aggregate_input </OUT>
   <KEY> gamecode </KEY>
   <KEY> city </KEY>
   <KEY> date </KEY>
<SORT>

The sort operator sort_input takes input from the join operator join_on_storecode, so the IN tag of the operator has value join_on_storecode. The OUT tag has value aggregate_input, the aggregate operator that needs the sorted input. The data is sorted on three fields, hence there are three KEY tags. Each KEY tag holds the value of one sort field.

### 4.4.7 Copy

A copy operator receives data input from a single source and writes it to two output flows. Since a copy operator requires only the source and targets, the mandatory parameters are operator name, and input and output information. There are two OUT tags as the data is written to two flows. As the data format cannot be changed in this operator, MAPPING tag is not specified in the XML representation of copy. The generic XML representation for copy is given below.

<COPY name= operator_name >
   <IN > source </IN>
   <OUT> target1 </OUT>
   <OUT> target2 </OUT>
<COPY>

The XML representation of the example copy operator in Figure 2.1 is given below.

<COPY name=“copy_input”>
   <IN > convert_to_dollar </IN>
   <OUT> sales_detail_fact </OUT>
   <OUT> join_on_storecode </OUT>
<COPY>
The copy operator `copy_input` takes input from the format operator `convert_to_dollar`, so the `IN` tag of the operator has value `convert_to_dollar`. Since the data is written to two output flows, there are two `OUT` tags. The first `OUT` tag has value `sales_detail_fact`, the name of the fact table that holds the detailed transaction information. The second `OUT` tag has value `join_on_storecode`, the name of the operator that takes input from `copy_input`.

The complete XML representation of the ETL requirement in Figure 1.1 is given in Appendix A.

### 4.5 Conclusions

This chapter discusses the logical modeling of ETL processes. The XML representation of data sources and targets, mapping and the commonly used ETL operators are defined. The parameters required for each ETL operator identified in Chapter 3 are represented as XML tags. Sample XML code for the examples in the last chapter is also given. The next chapter discusses converting the logical model to ETL process in a target environment.
Chapter 5: Converting the Logical Model to an ETL Program

This chapter discusses converting the logical model from Chapter 4 to an ETL program. For proof of concept, we choose SAS as the target ETL platform. We propose an algorithm to traverse the operators in the logical model. We implement the algorithm in an application that converts an ETL requirement specified in XML to an internal tabular representation that is then converted into SAS code. The advantages of this approach are discussed later in this chapter. Section 5.1 gives an overview of SAS programming. Section 5.2 describes the strategy to traverse the XML document to identify the order of converting the operators to the ETL program. Section 5.3 discusses the implementation of the conversion strategy to generate SAS code from the XML logical model document. Section 5.4 discusses the SAS code for each operator. Section 5.5 offers conclusions.

5.1 Overview of SAS Programming

SAS is a group of software products to support data management operations such as data entry, retrieval, reporting, ETL, statistical analysis and data mining. Some of the SAS products are Base SAS, SAS Enterprise Guide, SAS ACCESS, SAS CONNECT and SAS Web Report Studio. Most of these products are abstractions of the SAS language referred to as the Base SAS programming language, with a GUI that converts the user input to Base SAS programs. SAS has a built-in file structure and program constructs to access and manipulate the files. The file structure is called dataset and is analogous to a table in SQL. We focus on reading SAS datasets as input and converting the transformation logic from the XML document to SAS code. The program constructs required to implement the transformation logic are explained below.

A SAS program has two components: data step and proc step. A data step is a set of statements to read data, create datasets, and do user defined calculations. Data steps begin with the keyword DATA followed by an output dataset name. Data steps can read data input from users, datasets, or files from external vendors such as relational tables or text files. We assume the inputs are already read into SAS datasets and hence focus on reading data from datasets and applying transformation logic from the XML document. A dataset is read into a data step using the keyword SET. A simple data step that reads the dataset game_sales_asia mentioned in Figure 2.1 is given below.

```sas
DATA game_sales_asia;
SET game_sales;
IF storecode = 'A321' THEN storecode= 'A371';
RUN;
```
This data step reads dataset `game_sales_asia` and changes the `storecode` value ‘A321’ to ‘A371.’ The name mentioned next to the keyword `DATA` is the name of the output dataset and the name mentioned next to keyword `SET` is the name of the input dataset. Since the input and output datasets are same in this example, SAS simply overwrites the original `game_sales_asia` dataset. The keyword `RUN` denotes the end of the `game_sales_asia` data step. There are other ways to end a data step that are discussed later in this chapter.

Thus a data step can be used to read datasets, create datasets and manipulate the input values as per requirement. The second component of SAS language, the `proc step` is a more powerful but restrictive set of SAS statements. A proc step can perform only predefined operations on the input data such as sorting or summarizing. Proc steps begin with the keyword `PROC` followed by a predefined procedure name. Procedures generally have only a set of possible statements [DS03]. These statements provide options to execute the procedure on the input data. An example for a proc step that sorts the `game_sales_asia` dataset is given below.

```
PROC SORT DATA = game_sales_asia OUT = sorted_game_sales_asia;
   BY storecode;
RUN;
```

This procedure sorts the `game_sales_asia` dataset by the field `storecode`. The keyword `DATA` followed by an equal sign gives the input dataset name and `OUT` followed by an equal sign gives the dataset to which the sorted data should be written. To overwrite `game_sales` with the sorted data, the same dataset name `game_sales_asia` can be used for the input as well as the output datasets. The keyword `RUN` indicates the end of the proc step, but the end of a `proc` step can be denoted in other ways like the beginning of another proc step or a data step. We use proc step for implementing the sort and aggregate operators. The data and proc step features required to do ETL transformations are explored in detail Section 5.3. The next section discusses the algorithm to traverse the XML document to generate SAS code.

### 5.2 Generating SAS Code from an ETL Requirement

The XML document represents the ETL processing logic with each source, target and operator as child nodes of the root. Since ETL programs treat data as a flow from source to target, identifying the order in which operations are performed is important to generate the SAS code. The sources in the logical model can be identified using the `SOURCE` tag. Traversing from the source, the flow can be identified by following the `IN` and `OUT` tags in source/operator blocks. But there are situations in which code generation cannot be immediately performed because all the inputs an operator requires should be datasets while the XML document gives the value of the `IN` tag as the name of an operator. This can be
demonstrated using the example in Figure 2.1. Traversing the XML document from source store, we reach the join operator join_on_storecode. The SAS code for a join operation requires two datasets as inputs while we have a dataset and the output of an operator in the XML document. Hence the code for the join operator can be generated only after all the operations preceding that input is performed and the output written to a dataset. This scenario is shown in Figure 5.1.

The fully shaded box indicates an already traversed XML node and the box with hatched shading indicates a node whose SAS code cannot be generated until the preceding operations are performed and the output data written to a dataset. At this point, we backtrack using the IN tags to find the source dataset of the preceding operator(s). Starting from join_on_storecode, we follow its first IN tag and visit the corresponding node convert_to_dollar. Since this is not a source node, we further visit nodes join_on_currency, promo_items_filter and union_region_sales to reach the source game_sales_asia. Since this is a source node, the code generation resumes from this point. This scenario is illustrated in Figure 5.2.
Figure 5.2 Traversal of XML document: resuming from game_sales_asia

From game_sales_asia, we reach union_region_sales. Union operator requires two datasets as input. Since the IN nodes of this union operator, game_sales_asia and game_sales_europe are of type source, the corresponding SAS code is generated. The OUT tag of union_region_sales is then followed to reach the filter operator, promo_items_filter. The code for filter also is immediately generated. Following the OUT tag of promo_items_filter, we reach the join operator join_on_currency which requires two datasets as input while the XML document gives the first input as an operator. This scenario is demonstrated in Figure 5.3.

Figure 5.3 Traversal of XML document: input needs to be written to dataset

The shaded circle indicates that the output of the previous operator should be written to a dataset for the code generation of the next operator. Once this is done, the SAS code for join_on_currency is generated. Following the OUT tag of the join operator, we reach the format operator convert_to_dollar, and its SAS code is generated. The OUT tag of convert_to_dollar leads us to copy_input. Since the same data flow
needs to be input two different flows, the data needs to be written to a dataset here. This scenario is illustrated in Figure 5.4.

Figure 5.4 Traversal of XML document: input needs to be written to dataset

The output of the SAS code generated so far is written to a dataset and this temporary dataset replaces the copy operator. It then needs to be written to the target dataset sales_detail_fact and operator \textit{join_on_storecode}. Since \textit{join_on_storecode} has two datasets as input at this point, its code is generated. The succeeding operator is of type sort. As we discussed earlier in Section 5.1, sort is a predefined operation that is implemented in a proc step. A proc step requires a dataset as input and can write the output into a dataset. Hence the output of the join operator is written to a dataset and the SAS code for the sort operator is generated and output written to another dataset. This is illustrated in Figure 5.5.

Figure 5.5 Traversal of XML document: PROC SORT step input and output written to dataset

The \textit{OUT} node of the sort operator points to \textit{aggregate_input}. As mentioned in Section 5.1, aggregation is implemented as a proc step in SAS. The output of the aggregate operator is of type target and hence it can directly be written to dataset \textit{game_sales_fact}. This is illustrated in Figure 5.5.
Since there are no more nodes to be visited, the traversal ends at this point and the generated code is the SAS equivalent of the XML document. The next section discusses the programmatic implementation of the conversion from the logical model to ETL program.

### 5.3 Converting the XML Document to ETL Application Code

In this section we discuss the implementation of an application program to process the XML logical model and generate ETL programs from it. We leverage a model management approach \[B03\] to achieve this. A model management system is a generic approach that treats models and mappings as a set of abstractions. This enables the transformation from one model to another using the concepts the models share. A model management approach is preferred over a direct translation program from one model to another for the following reasons \[B03\]:

1. It is a generic approach that can be applied to multiple models.
2. It abstracts the common set of concepts that different models share and enables to create mappings between models.

We utilize a multilevel dictionary model \[ACB05\] to represent the XML document in a way that facilitates the conversion from the XML representation of ETL processes to programs in a target ETL platform. A relational table-based representation provides the following benefits.

1. Organizing model information in tables facilitates automated processing of models.
2. A relational table-based representation can simplify the model creation, mapping and generation operations by utilizing the functionalities of SQL, which is the standard language for relational data processing.
3. The approach can be extended to include more models by plugging in new mappings.
We design the table representation of the XML model and implement an application that generates SAS process from the model. The application is implemented in three stages:

i. define and populate the table-based representation of the XML process model,
ii. define and populate the table-based representation of the SAS process model, and
iii. generate the SAS ETL code from the SAS tables.

Subsection 5.3.1 explains the approach we adopt for managing the ETL process model. Subsection 5.3.2 discusses our implementation of the table-based representation of the ETL XML model. Subsection 5.3.3 discusses our implementation of the table-based representation of the ETL SAS model and introduces the algorithm to populate the SAS process tables from the XML process tables. Subsection 5.3.4 discusses the algorithm to generate the SAS code from the SAS \textit{M model} tables.

5.3.1 Dictionary-based Representation for Model Management

Atzeni et al. [ACB05] propose a four-part dictionary to handle and coordinate models. The authors recognize that a visible dictionary in a relational database structure to represent the models eases the translation between models. The dictionary is organized into four parts as shown in Figure 5.6.

![Figure 5.7 The four parts of the Multilevel Data Dictionary [ACB05]](image)

Instances of models are represented in the \textit{M model}, the lower left part of the dictionary. The table representation of our XML logical model for a particular ETL requirement (instance) would come under the \textit{M model}. The lower right part of the figure, the \textit{SM} or the \textit{supermodel} part, unifies the sets of tables in the \textit{M model} by grouping similar constructs from different \textit{M models} into the same table. The upper left part of the figure, the \textit{mM} or the \textit{metamodel} part describes the \textit{M model}; the upper right part of the figure, the \textit{mSM} or the \textit{meta-supermodel}, describes the \textit{SM} part. Our focus is on the \textit{M Model} as we need to represent the ETL XML model and SAS model as a set of dictionary tables to facilitate translation from one to the other.
To explain the *M Model* dictionary tables, consider the Entity-Relationship diagram instances in Figure 5.8. We refer to these as ER schemas to avoid confusing with the dictionary model. The first ER schema describes a conceptual model of a *product-supplier-department* relation. The relationship *supply* relates the entities *product*, *supplier* and *department*. A supplier is identified by an id *code*, has a name *SName* and belongs to a city. Hence the entity supplier has attributes *code*, *SName* and *City*. Similarly, entity product has attributes *ProdNo* and *Name*, and entity department has attributes *DeptNo* and *DeptName*. A supplier can supply one or more product to one or more departments. This is indicated by cardinality *(I, N)* next to the entity supplier. The cardinality of entities product and department in the supply relationship is *(0, N)*. The second ER diagram shows a similar relationship of employees to a group. We explain how the first schema is represented in the multilevel dictionary tables for ER schema and show the data in the table for the first and second schemas, to illustrate how multiple ER schemas are represented in the same set of tables.

![Diagram](image)

**Figure 5.8 Two ER schema instances [ACB05]**

The multilevel dictionary model uses five tables to capture the information in an ER schema. The dictionary tables for the ER schema and the table entries for the ER schemas in Figure 5.8 are given in Figure 5.9.
The field OID has the primary key for each table. The table Schema contains one entry for each ER schema represented in the dictionary format. The field Name associates a name to each ER schema. Since the original ER diagrams in Figure 5.8 do not have names, the Name column has values 1\textsuperscript{st} ER Schema and 2\textsuperscript{nd} ER Schema. Each entity in the ER schema has an entry in the Entity table. The field Schema specifies the OID of the schema to which the entity belongs. The supplier entity belongs to the first ER schema and hence the field name has value Supplier and schema has value s1. The Relationship table has an entry for each relationship in the ER schema and its associated OID. The relation supply associates entities product, supplier and department in the first schema. The Relationship table has one row for this relation with name Supply and schema s1. The table AttributeOfEntity has an entry for each attribute of an entity in the ER schema. The field type gives the data type of the attribute as integer or string and field isKey gives a boolean value to indicate if the field contributes to the entity’s key. This information is typically not represented in an ER schema and requires domain knowledge to populate the fields. The field schema, abbreviated as Sch gives the OID of the attribute’s schema, and the field Entity has the OID of the attribute’s entity. For the attribute Code in the supplier entity in the Figure 5.8 a row is entered into the AttributeOfEntity table with value e1 for Entity, and Code for Name and s1 for Schema. The domain knowledge of the ER schema gives the values int for type and true for isKey. The
ComponentOfRelationship table associates each relationship to its participating entities. The supply relationship in the first ER schema links entities product, supplier and department. Hence there are three rows in the ComponentOfRelationship table with value r1 for field Relationship. The field isOpt is true if the entity is optional in the relationship, i.e., if minimum cardinality of 0 is allowed for that entity in the relation. The minimum cardinality of supplier is 1 in the first ER schema and hence isOpt is false. The field isFunct is true for an entity in a relationship if the maximum cardinality allowed for that entity is 1. The maximum cardinality for supplier in relationship supply is N and hence isFunct is false for entity supplier in relationship supply. Attributes isOpt and isFunct are used to represent the cardinality of the relationship between entities in an ER data model. A relationship only means the flow of data from one operator to another in our model, hence attributes isOpt and isFunct are not used in our model.

5.3.2 Defining and Populating the Dictionary-based Representation of the XML Process Model
We adapt the dictionary-based format to represent the XML and SAS model tables. Since an ETL logical model describes a process and the steps involved in it rather than the data description, the M model dictionary tables for the XML logical model do not include the data types of the source or target data. Similar to the Schema table in the ER, we have a Process table that has a row for each ETL process represented by the tables. The process table for the XML logical model with an entry for the ETL requirement discussed in Figure 4.1 is given below.

<table>
<thead>
<tr>
<th>PROCESS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Name</td>
</tr>
<tr>
<td>P1</td>
<td>Process1</td>
</tr>
</tbody>
</table>

Table 5.1 Process table for the ETL requirement in Figure 1.1
Here PID is the primary key for the table and Name is an identifier for the specific ETL process. We implement it to have value ProcessN for the n
th XML document represented in the dictionary format, hence the ETL requirement in Figure 1.1 is entered as Process1 in this table.

Similar to the Entity table in the ER schema, we have an Entity table to represent each source, operator or target block in the XML document. An Entity table has fields EID, Name, Process and Type. The Type field is not part of the original M model [ACB05] schema and is added to capture the information in the ETL process. It is used to indicate if the block represents a source, target or operator node in the XML document. The Name field is populated with the value of the attribute name in the XML document and the Type field is populated with the XML block tag. Table 5.2 gives the Entity table entry of the ETL requirement in Figure 4.1.
Table 5.2 Entity table for the ETL requirement in Figure 1.1

<table>
<thead>
<tr>
<th>EID</th>
<th>Name</th>
<th>Process</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>game_sales_asia</td>
<td>P1</td>
<td>SOURCE</td>
</tr>
<tr>
<td>E2</td>
<td>game_sales_europe</td>
<td>P1</td>
<td>SOURCE</td>
</tr>
<tr>
<td>E3</td>
<td>Store</td>
<td>P1</td>
<td>SOURCE</td>
</tr>
<tr>
<td>E4</td>
<td>Exchange_rate</td>
<td>P1</td>
<td>SOURCE</td>
</tr>
<tr>
<td>E5</td>
<td>sales_fact</td>
<td>P1</td>
<td>TARGET</td>
</tr>
<tr>
<td>E6</td>
<td>union_region_sales</td>
<td>P1</td>
<td>UNION</td>
</tr>
<tr>
<td>E7</td>
<td>promo_items_filter</td>
<td>P1</td>
<td>FILTER</td>
</tr>
<tr>
<td>E8</td>
<td>join_on_currency</td>
<td>P1</td>
<td>JOIN</td>
</tr>
<tr>
<td>E9</td>
<td>join_on_storecode</td>
<td>P1</td>
<td>JOIN</td>
</tr>
<tr>
<td>E10</td>
<td>convert_to_dollar</td>
<td>P1</td>
<td>FORMAT</td>
</tr>
<tr>
<td>E11</td>
<td>sort_input</td>
<td>P1</td>
<td>SORT</td>
</tr>
<tr>
<td>E12</td>
<td>Aggregate_on_city</td>
<td>P1</td>
<td>AGGREGATE</td>
</tr>
</tbody>
</table>

The process value for all the entities is P1 since all the entities belong to the same process. The source block with name game_sales_asia is the first entry in the table, and hence has value E1 for column EID. The name of the entity is the same as the value of the argument name for the block SOURCE in the XML document, hence game_sales_asia is entered for column Name. The type of the entity is the tag of the XML block corresponding to the entity and is source for entity game_sales_asia. The rest of the ETL operators from the XML document are similarly entered in the Entity table.

We adapt the Relationship and ComponentOfRelationship tables in the ER schema to represent the relationship between data inputs, operators and output. The relationships in an ER schema are named and has a meaning attached to it, and so two separate tables are required to represent each relationship. The Relationship table holds the name and schema of each relation the ComponentOfRelationship table holds the cardinality information of the relationship, so the Relationship table is a level of detail of the schema table and ComponentOfRelationship table is a level of detail of the Relationship table. But in an ETL requirement, a relationship between entities just means the flow of data from an input/source to a target/operator and does not require a name. Hence we use a single table, Relationship, to show the data flow between entities. The relationship table has primary key RID. The table has a row for each IN/OUT tag in the XML document. For each entity in the Entity table, the EID is entered in the Entity field of the Relationship table. The IN and OUT tag values for the EID is retrieved from the XML document and is entered in the table, with field Pointer indicating whether its an IN or OUT tag, and field Value indicating the value of the tag. Table 5.3 gives the Relationship table entry of the ETL requirement in Figure 4.1.
Table 5.3 Relationship table for the ETL requirement in Figure 1.1

For each entry in the Entity table, we search the XML document for IN/OUT tags. For the first entry in the Entity table in Figure 5.2, there is only one OUT tag, with value `union_region_sales`. We enter this information in the Relationship table with value R1 for RID. The EID for the row comes from the Entity table and has value E1. The Pointer field has value `OUT` as his is the OUT tag of the entity. The field Value has the value of the OUT tag in the XML document, `union_region_sales`. If there are multiple IN/OUT tags for the same entity, there will be multiple entries for the same EID in the Relationship table. For example, a join operator has two inputs and one output, so entity `join_on_currency` with EID E8 has two IN tags and one OUT tag. Correspondingly, there are three rows in the Relationship table with EID E8. The Relationship table is populated similarly with the IN and OUT tag values of the XML document.

Similar to the AttributeOfEntity table in the ER Schema, we have an Attribute table that stores the attribute information of entities. Field AID is the primary key of the table. The field Entity indicates which
entity the attribute belongs to. For source and target blocks the *Attribute* table stores the column names of
the data entity. For join and aggregate operators, the *Attribute* table stores the key fields of the operation.
For the filter operator, the *Attribute* table stores the filter condition. Table 5.4 gives the *Attribute* table
entry of the ETL requirement in Figure 4.1.

<table>
<thead>
<tr>
<th>AID</th>
<th>Entity</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>E1</td>
<td>game_code</td>
</tr>
<tr>
<td>A2</td>
<td>E1</td>
<td>store_code</td>
</tr>
<tr>
<td>A3</td>
<td>E1</td>
<td>currency</td>
</tr>
<tr>
<td>A4</td>
<td>E1</td>
<td>amount</td>
</tr>
<tr>
<td>A5</td>
<td>E1</td>
<td>quantity</td>
</tr>
<tr>
<td>A6</td>
<td>E1</td>
<td>date</td>
</tr>
<tr>
<td>A7</td>
<td>E2</td>
<td>game_code</td>
</tr>
<tr>
<td>A8</td>
<td>E2</td>
<td>store_code</td>
</tr>
<tr>
<td>A9</td>
<td>E2</td>
<td>currency</td>
</tr>
<tr>
<td>A10</td>
<td>E2</td>
<td>amount</td>
</tr>
<tr>
<td>A11</td>
<td>E2</td>
<td>quantity</td>
</tr>
<tr>
<td>A12</td>
<td>E3</td>
<td>store_code</td>
</tr>
<tr>
<td>A13</td>
<td>E3</td>
<td>manager</td>
</tr>
<tr>
<td>A14</td>
<td>E3</td>
<td>city</td>
</tr>
<tr>
<td>A15</td>
<td>E4</td>
<td>currency</td>
</tr>
<tr>
<td>A16</td>
<td>E4</td>
<td>exchange_rate</td>
</tr>
<tr>
<td>A17</td>
<td>E5</td>
<td>game_code</td>
</tr>
<tr>
<td>A18</td>
<td>E5</td>
<td>city</td>
</tr>
<tr>
<td>A19</td>
<td>E5</td>
<td>amount</td>
</tr>
<tr>
<td>A20</td>
<td>E5</td>
<td>quantity</td>
</tr>
<tr>
<td>A21</td>
<td>E5</td>
<td>date</td>
</tr>
</tbody>
</table>

Table 5.4 Attribute table for the ETL requirement in Figure 1.1

Entity *E1* corresponds to source *game_sales_asia* and has six columns, hence there are six rows in the
*Attribute* table for *E1*. The first column in *game_sales_asia* is *game_code* and is entered in the table with
*AID* A1 and field *game_sales_asia*. The other columns in source and target entities are populated
similarly. Entity *E7* corresponds to filter operator *filter_promo_items*, so the filter condition *amount!=0* is
entered for field *Value*. Entity *E8* corresponds to join operator *join_on_currency* and the join key
*currency* is entered for field *Value*.

Since the original M table is designed to capture the information of a data model, none of the above tables
can capture the mapping information of columns in the join and format operators. The mapping of
columns is an important functionality in an ETL process and the XML model requires a separate table to accommodate the column mapping information. We use a *Mapping* table to hold the mapping data of the join and format operators. The primary key of the *Mapping* table is *MID* and has a foreign key *Entity* to the *Entity* table. The *Mapping* field gives the input field or expression that is mapped to an output column and the *Target* field gives the name of the output field. A row is entered in the *Mapping table* for each *FIELD* tag in the join and format blocks of XML document. The value of the argument *expression* in the *FIELD* tag corresponds to the column *Mapping* and the value of the argument *name* in the *FIELD* tag corresponds to the column *Target* in the table. Table 5.5 gives the *Mapping* table entry of the ETL requirement in Figure 4.1.

<table>
<thead>
<tr>
<th>MID</th>
<th>Entity</th>
<th>Mapping</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>E8</td>
<td>game_code</td>
<td>game_code</td>
</tr>
<tr>
<td>M2</td>
<td>E8</td>
<td>store_code</td>
<td>store_code</td>
</tr>
<tr>
<td>M3</td>
<td>E8</td>
<td>currency</td>
<td>currency</td>
</tr>
<tr>
<td>M4</td>
<td>E8</td>
<td>amount</td>
<td>amount</td>
</tr>
<tr>
<td>M5</td>
<td>E8</td>
<td>quantity</td>
<td>quantity</td>
</tr>
<tr>
<td>M6</td>
<td>E8</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>M7</td>
<td>E8</td>
<td>exchange_rate</td>
<td>exchange_rate</td>
</tr>
<tr>
<td>M8</td>
<td>E9</td>
<td>game_code</td>
<td>game_code</td>
</tr>
<tr>
<td>M9</td>
<td>E9</td>
<td>city</td>
<td>city</td>
</tr>
<tr>
<td>M10</td>
<td>E9</td>
<td>amount</td>
<td>amount</td>
</tr>
<tr>
<td>M11</td>
<td>E9</td>
<td>quantity</td>
<td>quantity</td>
</tr>
<tr>
<td>M12</td>
<td>E9</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>M13</td>
<td>E10</td>
<td>game_code</td>
<td>game_code</td>
</tr>
<tr>
<td>M14</td>
<td>E10</td>
<td>store_code</td>
<td>store_code</td>
</tr>
<tr>
<td>M15</td>
<td>E10</td>
<td>amount*currency</td>
<td>amount</td>
</tr>
<tr>
<td>M16</td>
<td>E10</td>
<td>quantity</td>
<td>quantity</td>
</tr>
<tr>
<td>M17</td>
<td>E10</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>M18</td>
<td>E12</td>
<td>game_code</td>
<td>game_code</td>
</tr>
<tr>
<td>M19</td>
<td>E12</td>
<td>city</td>
<td>city</td>
</tr>
<tr>
<td>M20</td>
<td>E12</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>M21</td>
<td>E12</td>
<td>amount</td>
<td>amount</td>
</tr>
<tr>
<td>M22</td>
<td>E12</td>
<td>quantity</td>
<td>quantity</td>
</tr>
</tbody>
</table>

Table 5.5 *Mapping table for the XML document in Figure 4.1*

Entity *E8* corresponds to join operator *join_on_currency* and has seven columns mapped to the output. For the first column, the value of argument *name* in the tag is *game_code* and *expression* is *game_code*. This is entered into the table with *MID M1, Entity E8, Mapping game_code* and *Target game_code*. The other *FIELD* tags in the join and format operators are populated similarly into the *Mapping* table.
5.3.3 Defining and Populating the Dictionary-based Representation of the SAS Process Model

The SAS process model tables are similar to the XML process tables, as there is an equivalent operator in SAS for each operator in the XML model. We use a terminology mapping table that maps constructs in the XML model to constructs in the SAS model, and leverage this table to implement a program that automatically populates the SAS tables from the XML tables. Table 5.6 gives the terminology mapping between the XML model and SAS model.

<table>
<thead>
<tr>
<th>XML model</th>
<th>SAS model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Dataset</td>
<td>Direct mapping</td>
</tr>
<tr>
<td>Target</td>
<td>Dataset</td>
<td>Direct mapping</td>
</tr>
<tr>
<td>Filter</td>
<td>Filter</td>
<td>Direct mapping, implemented in data step</td>
</tr>
<tr>
<td>Join</td>
<td>Join</td>
<td>Direct mapping, implemented in data step</td>
</tr>
<tr>
<td>Format</td>
<td>Format</td>
<td>Direct mapping, implemented in data step</td>
</tr>
<tr>
<td>Aggregate</td>
<td>Aggregate</td>
<td>Direct mapping, implemented in proc step</td>
</tr>
<tr>
<td>Union</td>
<td>Union</td>
<td>Direct mapping, implemented in data step</td>
</tr>
<tr>
<td>Sort</td>
<td>Sort</td>
<td>Direct mapping, implemented in proc step</td>
</tr>
<tr>
<td>Copy</td>
<td>Temp_data</td>
<td>Replaced by temporary dataset created</td>
</tr>
<tr>
<td></td>
<td>Temp_data</td>
<td>Created for sort, aggregate, join and union</td>
</tr>
<tr>
<td>Process</td>
<td>Process</td>
<td>Identical</td>
</tr>
<tr>
<td>Entity</td>
<td>SAS_Entity</td>
<td>Generated from XML entity table</td>
</tr>
<tr>
<td>Relationship</td>
<td>SAS_Relationship</td>
<td>Generated from XML relationship table</td>
</tr>
<tr>
<td>Attribute</td>
<td>Attribute</td>
<td>Identical</td>
</tr>
<tr>
<td>Mapping</td>
<td>Mapping</td>
<td>Identical</td>
</tr>
</tbody>
</table>

Table 5.6 Terminology mapping between the XML model and SAS model.

The source and target are identical elements in the XML and SAS model and hence are directly mapped to the SAS model. All operators in the XML model except copy have the same meaning in the XML and SAS models and hence have a direct mapping. Filter, format, join and union are operators that are implemented in a data step while sort and aggregate are implemented in a proc step, as explained in Section 5.1. The copy operator does not have a direct implementation in SAS and hence is replaced by a temporary dataset, of type *temp_data*. Hence the name and type values for the copy operator in the entity table is replaced by *temp_data* and a temporary dataset name created at run-time in the SAS table. For each operator that requires a dataset input or output but has a non-dataset input or output in the XML model, a temporary dataset is created in the SAS model and the SAS *entity* and *relationship* tables updated accordingly. Since the entity and relationship tables are modified for the SAS model, we create two new tables, *SAS_Entity* and *SAS_Relationship for the SAS model*. The remaining tables from the XML model are directly used in the SAS model. The entity table entries in the SAS table for the example in Figure 1.1 are given below.
Table 5.7 SAS_Entity table entries for the ETL requirement in Figure 1.1

As explained in Section 5.1, temporary datasets of type temp_data are created for copy, join, sort and aggregate operators. Of the four temporary datasets created, temp_1 replaces the copy operator and temp_2, temp_3, and temp_4 are created for operators join_on_currency and sort_input, as given in Figure 5.6. The entries in the SAS_Relationship table for these datasets are given in Table 5.8.
The dataset temp_1 replaces copy_input in the entity table, hence all occurrences of copy_input in the relationship table is replaced by temp_1 in the SAS_Relationship table. Datasets temp_2, temp_3, temp_4 are inserted in the SAS_Entity table, so the value column in the SAS_Relationship table is updated with the temporary datasets for each entity that has an IN or OUT pointer to the temporary datasets. New entries R27-R32 are inserted to give the IN and OUT values for temporary datasets E15, E16 and17. The next section explains converting the SAS process model tables to ETL code.

5.3.4 Converting from SAS Model Tables to ETL Code

This section gives an overview of the ETL process generation from the SAS process tables. The methodology to construct the SAS code segments from the SAS tables is explained. The algorithm to generate the code for each operator is given in Appendix B. We leverage the example ETL scenario in Figure 1.1 and the tables process in Table 5.1, SAS_Entity in Table 5.7, SAS_Relationship in Table 5.8, attribute in Table 5.4 and mapping in Table 5.5 to discuss the SAS code.

We use arrays process_later[] and available[] to store the name of the temporary datasets generated by the ETL code. When processing starts, the EID of all the temporary datasets created by the process are

<table>
<thead>
<tr>
<th>RID</th>
<th>Entity</th>
<th>Pointer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R14</td>
<td>E9</td>
<td>OUT</td>
<td>convert_to_dollar</td>
</tr>
<tr>
<td>R15</td>
<td>E10</td>
<td>IN</td>
<td>temp_1</td>
</tr>
<tr>
<td>R16</td>
<td>E10</td>
<td>IN</td>
<td>store</td>
</tr>
<tr>
<td>R17</td>
<td>E10</td>
<td>OUT</td>
<td>temp_2</td>
</tr>
<tr>
<td>R18</td>
<td>E11</td>
<td>IN</td>
<td>join_on_currency</td>
</tr>
<tr>
<td>R19</td>
<td>E11</td>
<td>OUT</td>
<td>temp_1</td>
</tr>
<tr>
<td>R20</td>
<td>E12</td>
<td>IN</td>
<td>convert_to_dollar</td>
</tr>
<tr>
<td>R21</td>
<td>E12</td>
<td>OUT</td>
<td>sales_detail_fact</td>
</tr>
<tr>
<td>R22</td>
<td>E12</td>
<td>OUT</td>
<td>join_on_storecode</td>
</tr>
<tr>
<td>R23</td>
<td>E13</td>
<td>IN</td>
<td>temp_2</td>
</tr>
<tr>
<td>R24</td>
<td>E13</td>
<td>OUT</td>
<td>temp_3</td>
</tr>
<tr>
<td>R25</td>
<td>E14</td>
<td>IN</td>
<td>temp_3</td>
</tr>
<tr>
<td>R26</td>
<td>E14</td>
<td>OUT</td>
<td>sales_fact</td>
</tr>
<tr>
<td>R27</td>
<td>E15</td>
<td>OUT</td>
<td>sort_input</td>
</tr>
<tr>
<td>R28</td>
<td>E15</td>
<td>IN</td>
<td>join_on_storecode</td>
</tr>
<tr>
<td>R29</td>
<td>E16</td>
<td>IN</td>
<td>sort_input</td>
</tr>
<tr>
<td>R30</td>
<td>E16</td>
<td>OUT</td>
<td>aggregate_input</td>
</tr>
<tr>
<td>R31</td>
<td>E17</td>
<td>OUT</td>
<td>join_on_currency</td>
</tr>
<tr>
<td>R32</td>
<td>E17</td>
<td>IN</td>
<td>promo_items_filter</td>
</tr>
</tbody>
</table>

Table 5.8 SAS_Relationship table entries for the ETL requirement in Figure 1.1
added to process_later[]. The processing starts from the SAS_Entity table with an entity of type source. The OUT pointer of each source is checked for the operator that has the source as input and its SAS code is generated. When an operator has multiple sources and the other source is a temporary dataset that is not created yet, the source is added to the list process_later. When a temporary dataset is generated as output of a proc step or data step, it is added to the list available. Once all the sources in entity table are processed this way and SAS code generated, the process_later list is checked for sources. For each source that is input to an operator that has all inputs available, its SAS code is generated and the source is removed from the process_later list. For each temp_data in the process_later list that is present in the list available, the SAS code is generated and the temp_data is removed from the process_later list. When there are no more elements in the process_later list, the SAS code generation is completed. The algorithm is given in Figure 5.10.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>gen_SAS_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Generates SAS code for a given process</td>
</tr>
<tr>
<td>Input</td>
<td>PID</td>
</tr>
<tr>
<td>Output</td>
<td>True if method completed successfully, False otherwise</td>
</tr>
<tr>
<td>Result</td>
<td>A file is created with the given PID and contains the SAS code for that process.</td>
</tr>
</tbody>
</table>

**Pseudo-code:**
1. Set available[] to empty
2. For each row of type temp_data, add its eid to process_later[]
3. For each eid of type source in entity
   3.1 Check the OUT operator in relationship
   3.2 If both inputs for the operator are available
      3.2.1 Generate its SAS code
      3.2.2 If the output dataset is of type temp_data add it to available[]
   3.3 Else add the source eid to process_later[]
4. For each dataset in process_later[] if the eid exists in available[]
   4.1 Check the OUT operator in relationship
   4.2 If both inputs for the operator are available
      4.2.1 Generate its SAS code
      4.2.2 If the output dataset is of type temp_data add it to available[]

**Figure 5.10 Algorithm to generate SAS code from SAS process tables**

The expected SAS output for each operator is explained in the next section.

**5.4 SAS Code for ETL Operators**

This section discusses the SAS code for each operator. The methodology to construct the SAS code segments from the hash data structure is explained. We leverage the example ETL scenario in Figure 2.1 and XML code segments in Sections 4.5.1–4.5.6 to discuss the SAS code.
5.4.1 Filter
The filter operator is implemented as a single statement in a SAS data step. SAS filter code requires only one mandatory parameter, the filter condition. The filter statement is an IF statement with an expression which if evaluated to true passes the record to the output flow. The records for which the filter expression evaluates to false are filtered out. The code for the filter example discussed in Section 4.5.1 in context of the traversal algorithm in Section 5.2 is given below. We assume that the output of the union operator is written to a temporary dataset temp1.

```
DATA temp1;
SET temp1;
IF amount!=0;
RUN;
```

Since the input and output dataset names are the same, the contents of the input dataset will be replaced with the subset data with non-zero amount values. The filter condition amount!=0 is specified in the IF statement.

5.4.2 Join
The join is implemented as a SAS data step as it needs multiple parameters and condition checks. We use a MERGE statement and BY statement in the data step to do the join operation. The MERGE statement gives the source names and BY statement gives the key values. The key values from the attribute table are combined to get a list of keys. For join, the data step needs two datasets as inputs, and the code is generated only when both inputs are available in datasets. The fields that are mapped to the output are specified in the KEEP statement. The SAS code for the join example discussed in Section 4.5.2 is given below.

```
DATA temp2;
MERGE temp1 exchange_rate;
BY currency;
KEEP gamecode storecode currency amount quantity exchange_rate;
RUN;
```

We assume the output of the filter operator is written to temp1. The output of the join is written to dataset temp2. The join key of the operation is currency and is specified in the BY statement.

5.4.3 Format
The format operator is implemented as a SAS data step. The mapping of each field is done accessing the mapping table. Columns that have a direct mapping from source to target need not be assigned values in the data step, as the default behavior of the data step is to carry all columns directly to the target. A KEEP statement is used to select the input columns that are mapped directly to the output data flow. For
columns that have calculations performed to derive the target field value, the mapping expression is parsed to verify the correctness of the expression. Expressions that cannot be parsed successfully are commented in the SAS code for manual programmer revision later. The SAS code for the format example in Section 4.5.3 is given below.

\begin{verbatim}
DATA temp2;
SET temp2;
  amount = amount * exchange_rate;
KEEP gamecode storecode quantity date;
RUN;
\end{verbatim}

We assume the input of `join_on_currency` is written to dataset `temp2` and modify `temp2` to create the output of the format operation. The expression to calculate `amount` is parsed successfully and is added to the output.

### 5.4.4 Aggregate

The aggregate operator is implemented as a `proc-means` procedure. A `proc-sort` procedure is also generated before the aggregate step as `proc-means` requires sorted input. The SAS code for `sort` is discussed in Section 5.4.6. The default behavior of any `proc` step is to print the output data to the screen. To disable this, we use the `NOPRINT` option. Unlike the data step, where the output dataset name is the name of the data step itself, the `proc` step needs an explicit `OUTPUT` statement to write the output to a dataset. The `proc-means` procedure generates a row with summarized data on all rows, in addition to summarized data for each key. This row is generally not required in ETL requirements, so we use option `NWAY` to omit this row from the output dataset.

The input dataset name is given in the `proc-means` statement. A `CLASS` statement is used to specify the aggregate key. A `VAR` statement is used to give a list of columns that need to be summarized. The summary function that needs to be applied to the columns is determined by parsing the expression in the summary column mapping. The summary functions applied to the columns in the `VAR` statement and their target mappings are specified in the `OUTPUT` statement. In the XML notation, we use `sum(input_column)` for sum, `avg(input_column)` for average, `min(input_column)` for minimum, `max(input_column)` for maximum and `count(input_column)` for the number of observations. The corresponding SAS functions are `SUM(input_column)`, `MEAN(input_column)`, `MIN(input_column)`, `MAX(input_column)` and `N(input_column)`. Contrary to the usual assignment syntax, the target column name is specified on the right side of the assignment symbol `=`. e.g., `MEAN(input_column) = output_column`. If the same summary function has to be applied to multiple columns, all of the column
names can be specified as arguments to the same summary function, e.g., $\text{MEAN}(\text{input\_column1 input\_column2}) = \text{output\_column1 output\_column2}$.

The aggregate-key fields are carried as is to the target by default. The SAS code for the aggregate example discussed in Section 4.5.4 is given below.

```sas
PROC MEANS NWAY NOPRINT DATA = temp4;
CLASS gamecode city date;
VAR amount quantity;
OUTPUT OUT = sales_fact SUM(amount quantity) = amount quantity;
RUN;
```

We assume the output of the sort operator is written to dataset `temp4` and is taken as input for the aggregate operator. The aggregate keys `gamecode, city` and `date` for the operator `aggregate_input` are given in the `CLASS` statement. The summarized fields are specified in the statement `$\text{SUM(amount quantity)} = \text{amount quantity}$.

5.4.5 Union

The union operator is implemented as a data step. Since a union operation simply appends one dataset to the other, it does not need the input datasets to be sorted. As discussed in Section 4.5.5, a union operator requires both inputs to be of type source. If both the inputs are available in datasets, the appending operation can be done in a `SET` statement followed by the input dataset names. The SAS code for the union example discussed in Section 4.5.5 is given below.

```sas
DATA temp1;
SET game_sales_asia game_sales_europe;
RUN;
```

We assume the output of the union operation is written to temporary dataset `temp1`. The input datasets `game_sales_asia` and `game_sales_europe` are specified in the `SET` statement.

5.4.6 Sort

The sort operator is implemented as a `proc-sort` procedure. The `proc-sort` procedures is generated as part of the code generation for operators aggregate and join, and can also be used explicitly to sort a dataset. `Proc-sort` requires the input to be of type source. The input dataset name is specified in the `proc-sort` statement. The `BY` statement specifies the sort keys in the sort procedure. The default sort operation is by ascending key values. To sort a column in descending order, the `DESCENDING` keyword is used following the sort key. The SAS code for the sort example discussed in Section 4.5.6 is given below.

```sas
PROC SORT DATA = temp3 OUT = temp4;
```
BY gamecode;
RUN;

We assume the output of the `join_on_storecode` operator is written to temporary dataset `temp3` and the output of the sort operator is written to `temp4`. The sort-key `gamecode` is specified in the `BY` statement.

The complete SAS code for the ETL requirement in Figure 1.1 is given in Appendix C.

5.5 Conclusion
In this chapter we discuss the methodology to populate the XML and SAS tables in our dictionary model from the XML document as well as the procedure to generate ETL code from the SAS tables. The next chapter illustrates the parsing and code generation application along with case studies.
Chapter 6: Implementation and Case Studies

In this chapter we discuss the software development platform and application program to implement the ideas developed in the previous chapter. We illustrate the software functionality using two case studies. Section 6.1 gives an overview of the software development platform used. Section 6.2 discusses functionalities of the application that accept user input to create an XML model, populate the XML and SAS model tables and generate ETL code from it using the example in Figure 1.1. Section 6.3 illustrates the application using two case studies. Section 6.4 offers conclusions.

6.1 Development Platform

We develop the application program using Visual Studio 2008 Integrated Development Environment (IDE), C#.NET and Oracle Database 11g. We use Visual Studio 2008 IDE as the development environment for C#.NET. We use C#.Net as the programming language to develop the Graphical User Interface (GUI), to create the ETL XML document and generate the XML model tables from the XML document. The backend database is implemented in Oracle 11g to store the multilevel dictionary tables and also to implement procedures to generate the SAS model table data and SAS ETL code. We use ADO.NET classes to connect and communicate the GUI with the database.

Visual Studio 2008 IDE is a development environment that provides code development and debugging facilities for various programming languages. C#.NET is an object oriented programming language that provides namespaces and classes for XML data manipulation and database interaction. A namespace is a way of grouping classes into a hierarchy. The .NET framework provides various built-in namespaces to process XML data and manage database operations. We use namespaces System.Xml and Oracle.DataAccess.Client to create the XML document and parse it to populate the XML tables in Oracle database. ADO.NET is a set of classes in the .NET framework to access and manipulate data objects.

6.2 Implementation Features

We describe the features of the application program that implements our research work in this section. Each screen in the GUI is presented and its functionality is explained. We use the example ETL requirement in Figure 1.1 to illustrate the application and generate the data in Tables 6.1– 6.8.

6.2.1 Application Start Screen

The first screen in the application is the start page that gives an introduction to the application and provides buttons to choose an action. The screenshot of the first page is given in Figure 6.1.
6.2.2 Generating the XML Document

The *Generate XML Document* button in the start screen opens up a form that provides options to specify ETL requirements to generate the XML document. This screen provides the user-interface to enter data sources, targets and operator information into the XML document. Clicking on each button presents the user with a form to input its corresponding parameters. The screenshot of this page is given in Figure 6.2.
Figure 6.2 Generate XML screen

This screen enables the user to input data and operator blocks into the XML document. Clicking on the Add Source button presents the form to enter source name, output block name and columns. The screenshot of the form to add the source block game_sales_asia for the ETL requirement in Figure 1.1 is given in Figure 6.3.
Figure 6.3 Generate XML screen: Add Source block

Clicking on the Done button enters the corresponding block in the XML document and goes back to the generate XML screen. To add a target block, the user clicks on the Add Target button and is presented with the form to enter target block parameters. The screenshot of the form to add the target block game_sales for the ETL requirement in Figure 1.1 is given in Figure 6.4.
Figure 6.4 Generate XML screen: Add Target block

Once all the fields are entered, the user clicks the Done button to add the block to the XML document and go back to the generate XML screen. To add a filter operator block, the user clicks on the Add Filter button and is presented with the form to enter filter operator parameters. The screenshot of this form to enter the filter operator `promo_items_filter` for the ETL requirement in Figure 1.1 is given in Figure 6.5.
After inputting the parameter values for the filter operation, the user clicks the *Done* button to add the block to the XML document and go back to the generate XML screen. The *Add Join* button allows to add the parameters for join operator. The user is presented with a form to enter the parameters for the join. The screenshot of this form to enter the join operator `join_on_currency` for the ETL requirement in Figure 1.1 is given in Figure 6.6.
Clicking on the *Done* button enters the corresponding block in the XML document and directs the user to the generate XML screen. To add a format block, the user clicks on the *Add Format* button and is presented with the form to enter the parameters for aggregate operator. The screenshot of this form to enter the format operator for the ETL requirement in Figure 1.1 is given in Figure 6.7.
The Done button enables the user to enter the block into the XML document and go back to the generate XML screen. The Add Aggregate button is used to enter an aggregate operator to the XML document. When the user clicks on this button, the form to enter the operator parameters is presented. The screenshot of this form to enter the aggregate operator aggregate_input for the ETL requirement in Figure 1.1 is given in Figure 6.8.
Once all parameters are entered the user clicks the Done button to enter the corresponding block in the XML document and go back to the generate XML screen. The Add Sort button enables the user to input a sort operator by presenting the form to input the required parameters. The screenshot of this form to enter the sort operator sort_input for the ETL requirement in Figure 1.1 is given in Figure 6.4.
The *Done* button enables the user to enter the block into the XML document and go back to the generate XML screen. The *Add Union* button is used to enter a union operator to the XML document. When the user clicks on this button, the form to enter the union operator parameters is presented. The screenshot of this page enter the union operator *union_region_sales* for the ETL requirement in Figure 1.1 is given in Figure 6.10.
After inputting the parameter values for the union operator, the user clicks the *Done* button to add the block to the XML document and go back to the generate XML screen. The *Add Copy* button allows addition of a copy operator to the XML document. The user is presented with a form to enter the parameters for the copy operators. The screenshot of this page to enter the copy operator *copy_input* for the ETL requirement in Figure 1.1 is given in Figure 6.11.
The remaining source, target and operator blocks can be entered to the XML document in a similar fashion. Once the complete ETL requirement is entered, the user clicks the Back to Main Menu button to exit the screen and go to the main menu.

### 6.2.3 Generating the XML Model Tables

The Generate XML Model Tables button in the start screen opens up a form that provides options to generate XML model tables for a given XML document. This screen provides the user-interface to browse and select an XML document. After selecting the XML document, clicking the Load XML Tables button calls the subroutine to parse the XML document and generate the XML model tables. This screen also allows browsing the generated tables. The screenshot of this page is given in Figure 6.12.
6.2.4 Generating the SAS Model Tables

The Generate SAS Model Tables button in the start screen opens up a page that provides options to generate SAS model tables for a given Process_ID. This screen has a dropdown box to select a Process_ID. After selecting the Process_ID, clicking the Load SAS Tables button calls the PL/SQL procedure load_sas_tables which loads the SAS model tables from the XML model tables. The screenshot of this page is given in Figure 6.13.
6.2.5 Generating the SAS ETL Code

The Generate SAS ETL code button in the start screen opens a page that provides options to generate SAS code for a given Process_ID. This screen has a dropdown box to select a Process_ID. After selecting the Process_ID, clicking the Submit button calls the PL/SQL procedure gen_SAS_code which creates the SAS code from the SAS model tables and writes it to a file. The screen also allows browsing the generated file. The screenshot of this page is given in Figure 6.13.

Figure 6.14 Generate SAS ETL code Screen
6.3 Case Studies

In this section we present two case studies and illustrate the XML document, dictionary tables and ETL code generated using the application. The case studies address a set of ETL requirements from two different industry domains. Subsection 6.3.1 explains an ETL requirement from banking domain where customer and account information is used to classify customers [DHWRZ08]. Subsection 6.3.2 explains an ETL requirement from manufacturing domain where information on parts supplied by suppliers is loaded to the data warehouse [S03].

6.3.1 Case Study: Banking Domain

We explain an ETL requirement in banking domain [DHWR08] and present its block diagram. The XML document created by the application, the XML and SAS model tables and the target ETL code generated by the application is presented.

Figure 6.15 Block diagram of ETL requirement to classify customers

Figure 6.15 gives the block diagram of the ETL processing to convert the data in OLTP source tables to conform to the table structure of the data warehouse tables. The blocks on left side of the diagram represent the source tables customers and accounts in the OLTP system. These tables are used to load tables big_customers and other_customers depending on the customer’s total balance. Only accounts that are not loan accounts are used to calculate total balance, so a filter on account type field type is applied to remove loan accounts from the data flow. This is joined with the customer table on field cust_id and the output flow is aggregated on cust_id. The aggregated data is copied into two flows. The first flow is filtered for total_balance greater than $10,000 to load the big_customers table. The second flow is filtered for total_balance less than or equal to $10,000 to load the other_customers table.

The Generate XML Document screen is used to enter this requirement into the XML document. The XML document generated by the application is given in Appendix A. Once the XML document is created, the Generate XML Model Tables screen is used to parse the document and load the XML model tables. The Process table entry for the document is given in Table 6.1.
<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>Process2</td>
</tr>
</tbody>
</table>

Table 6.1 Process table entry for the ETL requirement in Figure 6.15

The Entity table entries for the document are given in Table 6.2.

<table>
<thead>
<tr>
<th>EID</th>
<th>Name</th>
<th>Process</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E15</td>
<td>customers</td>
<td>P2</td>
<td>source</td>
</tr>
<tr>
<td>E16</td>
<td>accounts</td>
<td>P2</td>
<td>source</td>
</tr>
<tr>
<td>E17</td>
<td>big_customers</td>
<td>P2</td>
<td>target</td>
</tr>
<tr>
<td>E18</td>
<td>other_customers</td>
<td>P2</td>
<td>target</td>
</tr>
<tr>
<td>E19</td>
<td>filter_on_type</td>
<td>P2</td>
<td>filter</td>
</tr>
<tr>
<td>E20</td>
<td>filter_big_cust</td>
<td>P2</td>
<td>filter</td>
</tr>
<tr>
<td>E21</td>
<td>filter_other_cust</td>
<td>P2</td>
<td>filter</td>
</tr>
<tr>
<td>E22</td>
<td>join_on_cust_id</td>
<td>P2</td>
<td>join</td>
</tr>
<tr>
<td>E23</td>
<td>filter_big_cust</td>
<td>P2</td>
<td>copy</td>
</tr>
<tr>
<td>E24</td>
<td>aggregate_on_cust_id</td>
<td>P2</td>
<td>aggregate</td>
</tr>
</tbody>
</table>

Table 6.2 Entity table entries for the ETL requirement in Figure 6.15

The Relationship table entries for the document are given in Table 6.3.

<table>
<thead>
<tr>
<th>RID</th>
<th>Entity</th>
<th>Pointer</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R27</td>
<td>E15</td>
<td>OUT</td>
<td>join_on_cust_id</td>
</tr>
<tr>
<td>R28</td>
<td>E16</td>
<td>OUT</td>
<td>filter_on_type</td>
</tr>
<tr>
<td>R29</td>
<td>E17</td>
<td>OUT</td>
<td>filter_big_cust</td>
</tr>
<tr>
<td>R30</td>
<td>E18</td>
<td>OUT</td>
<td>filter_other_cust</td>
</tr>
<tr>
<td>R31</td>
<td>E19</td>
<td>IN</td>
<td>accounts</td>
</tr>
<tr>
<td>R32</td>
<td>E19</td>
<td>OUT</td>
<td>join_on_cust_id</td>
</tr>
<tr>
<td>R33</td>
<td>E20</td>
<td>IN</td>
<td>copy_input</td>
</tr>
<tr>
<td>R34</td>
<td>E20</td>
<td>OUT</td>
<td>big_customers</td>
</tr>
<tr>
<td>R35</td>
<td>E21</td>
<td>IN</td>
<td>copy_input</td>
</tr>
<tr>
<td>R36</td>
<td>E21</td>
<td>OUT</td>
<td>other_customers</td>
</tr>
<tr>
<td>R37</td>
<td>E22</td>
<td>IN</td>
<td>customers</td>
</tr>
<tr>
<td>R38</td>
<td>E22</td>
<td>IN</td>
<td>accounts</td>
</tr>
<tr>
<td>R39</td>
<td>E22</td>
<td>OUT</td>
<td>aggregate_on_cust_id</td>
</tr>
<tr>
<td>R40</td>
<td>E23</td>
<td>IN</td>
<td>aggregate_input</td>
</tr>
<tr>
<td>R41</td>
<td>E23</td>
<td>OUT</td>
<td>filter_big_cust</td>
</tr>
<tr>
<td>R42</td>
<td>E23</td>
<td>OUT</td>
<td>filter_other_cust</td>
</tr>
<tr>
<td>R43</td>
<td>E24</td>
<td>IN</td>
<td>join_on_cust_id</td>
</tr>
<tr>
<td>R44</td>
<td>E24</td>
<td>OUT</td>
<td>copy_input</td>
</tr>
</tbody>
</table>

Table 6.3 Relationship table entries for the ETL requirement in Figure 6.15
The *Attribute* table entries for the document are given in Table 6.4.

<table>
<thead>
<tr>
<th>AID</th>
<th>Entity</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>A36</td>
<td>E15</td>
<td>cust_id</td>
</tr>
<tr>
<td>A37</td>
<td>E15</td>
<td>name</td>
</tr>
<tr>
<td>A38</td>
<td>E15</td>
<td>zip</td>
</tr>
<tr>
<td>A39</td>
<td>E15</td>
<td>age</td>
</tr>
<tr>
<td>A40</td>
<td>E15</td>
<td>credit_risk</td>
</tr>
<tr>
<td>A41</td>
<td>E15</td>
<td>years</td>
</tr>
<tr>
<td>A42</td>
<td>E16</td>
<td>balance</td>
</tr>
<tr>
<td>A43</td>
<td>E16</td>
<td>acct_id</td>
</tr>
<tr>
<td>A44</td>
<td>E16</td>
<td>cust_id</td>
</tr>
<tr>
<td>A45</td>
<td>E16</td>
<td>type</td>
</tr>
<tr>
<td>A46</td>
<td>E17</td>
<td>cust_id</td>
</tr>
<tr>
<td>A47</td>
<td>E17</td>
<td>name</td>
</tr>
<tr>
<td>A48</td>
<td>E17</td>
<td>total_balance</td>
</tr>
<tr>
<td>A49</td>
<td>E17</td>
<td>age_group</td>
</tr>
<tr>
<td>A50</td>
<td>E17</td>
<td>end_date</td>
</tr>
<tr>
<td>A51</td>
<td>E17</td>
<td>country</td>
</tr>
<tr>
<td>A52</td>
<td>E18</td>
<td>cust_id</td>
</tr>
<tr>
<td>A53</td>
<td>E18</td>
<td>name</td>
</tr>
<tr>
<td>A54</td>
<td>E18</td>
<td>total_balance</td>
</tr>
<tr>
<td>A55</td>
<td>E18</td>
<td>age_group</td>
</tr>
<tr>
<td>A56</td>
<td>E18</td>
<td>end_date</td>
</tr>
<tr>
<td>A57</td>
<td>E18</td>
<td>country</td>
</tr>
<tr>
<td>A58</td>
<td>E19</td>
<td>type!=='L'</td>
</tr>
<tr>
<td>A59</td>
<td>E20</td>
<td>total_balance&gt;10000</td>
</tr>
<tr>
<td>A60</td>
<td>E21</td>
<td>total_balance&lt;=10000</td>
</tr>
<tr>
<td>A61</td>
<td>E22</td>
<td>cust_id</td>
</tr>
<tr>
<td>A62</td>
<td>E24</td>
<td>cust_id</td>
</tr>
</tbody>
</table>

**Table 6.4 Attribute table entries for the ETL requirement in Figure 6.15**

The *Mapping* table entries for the document are given in Table 6.5.

<table>
<thead>
<tr>
<th>MID</th>
<th>Entity</th>
<th>Mapping</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>M23</td>
<td>E22</td>
<td>cust_id</td>
<td>cust_id</td>
</tr>
<tr>
<td>M24</td>
<td>E22</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>M25</td>
<td>E22</td>
<td>balance</td>
<td>balance</td>
</tr>
<tr>
<td>M26</td>
<td>E22</td>
<td>if age&lt;22 'A' else 'B'</td>
<td>age_group</td>
</tr>
<tr>
<td>M27</td>
<td>E22</td>
<td>date()+years*12</td>
<td>end_date</td>
</tr>
</tbody>
</table>
The Generate SAS Model Tables screen is used to generate the SAS model tables for the requirement from Tables 6.1-6.5. The SAS_Entity table entries for the document are given in Table 6.6.

The SAS_Relationship entries for the document are given in Table 6.7.
The *Generate SAS ETL Code* screen is used to generate the ETL code for the requirement. The generated SAS code is given in Appendix B. The next subsection discusses an ETL requirement in manufacturing domain.

### 6.3.2 Case Study: Manufacturing Domain

We explain an ETL requirement in manufacturing domain [S03] and present its block diagram. The XML document created by the application, the XML and SAS model tables and the target ETL code generated by the application is presented.

**Figure 6.16 Block diagram of ETL requirement to combine part-supplier data**

Figure 6.16 gives the block diagram of the ETL processing to convert the data in OLTP source tables *partsupp1* and *partsupp2* to conform to the table structure of the data warehouse table *partsupp*, combine them and write to table *partsupp*. The blocks on left side of the diagram represent the source tables *partsupp1* and *partsupp2* in the OLTP system. The source table *partsupp1* does not have the *date* field required in *partsupp*, so it is added using a format operator. The source table *partsupp2* has an additional field *department* that needs to be removed from the data flow. Also, the field *cost* has cost of parts in *Dollar* which needs to be converted to *Euro*. This is achieved using a format operator. The output
of the two format operators have the same schema and are combined and loaded to the data warehouse table `partsupp`.

The Generate XML Document screen is used to enter this requirement into the XML document. The XML document generated by the application is given in Appendix A. Once the XML document is created, the Generate XML Model Tables screen is used to parse the document and load the XML model tables. The process table entry for the document is given in Table 6.8.

<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>Process3</td>
</tr>
</tbody>
</table>

**Table 6.8 Process table entry for the ETL requirement in Figure 6.16**

The Entity table entries for the document are given in Table 6.9.

<table>
<thead>
<tr>
<th>EID</th>
<th>Name</th>
<th>Process</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E25</td>
<td>partsupp1</td>
<td>P3</td>
<td>source</td>
</tr>
<tr>
<td>E26</td>
<td>partsupp2</td>
<td>P3</td>
<td>source</td>
</tr>
<tr>
<td>E27</td>
<td>union_partsupp</td>
<td>P3</td>
<td>union</td>
</tr>
<tr>
<td>E28</td>
<td>add_date</td>
<td>P3</td>
<td>format</td>
</tr>
<tr>
<td>E29</td>
<td>convert_to_euro</td>
<td>P3</td>
<td>format</td>
</tr>
</tbody>
</table>

**Table 6.9 Entity table entries for the ETL requirement in Figure 6.16**

The Relationship table entries for the document are given in Table 6.10.

<table>
<thead>
<tr>
<th>RID</th>
<th>Entity</th>
<th>Pointer</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R45</td>
<td>E25</td>
<td>OUT</td>
<td>add_date</td>
</tr>
<tr>
<td>R46</td>
<td>E26</td>
<td>OUT</td>
<td>convert_to_euro</td>
</tr>
<tr>
<td>R47</td>
<td>E27</td>
<td>IN</td>
<td>add_date</td>
</tr>
<tr>
<td>R48</td>
<td>E27</td>
<td>IN</td>
<td>convert_to_euro</td>
</tr>
<tr>
<td>R49</td>
<td>E27</td>
<td>OUT</td>
<td>Partsupp</td>
</tr>
<tr>
<td>R50</td>
<td>E28</td>
<td>IN</td>
<td>partsupp1</td>
</tr>
<tr>
<td>R51</td>
<td>E28</td>
<td>OUT</td>
<td>union_partsupp</td>
</tr>
<tr>
<td>R52</td>
<td>E29</td>
<td>IN</td>
<td>partsupp2</td>
</tr>
<tr>
<td>R53</td>
<td>E29</td>
<td>OUT</td>
<td>union_partsupp</td>
</tr>
</tbody>
</table>

**Table 6.10 Relationship table entries for the ETL requirement in Figure 6.16**

The Attribute table entries for the document are given in Table 6.11.
Table 6.11 Attribute table entries for the ETL requirement in Figure 6.16

<table>
<thead>
<tr>
<th>AID</th>
<th>Entity</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>A63</td>
<td>E25</td>
<td>pkey</td>
</tr>
<tr>
<td>A64</td>
<td>E25</td>
<td>suppkey</td>
</tr>
<tr>
<td>A65</td>
<td>E25</td>
<td>qty</td>
</tr>
<tr>
<td>A66</td>
<td>E25</td>
<td>cost</td>
</tr>
<tr>
<td>A67</td>
<td>E26</td>
<td>pkey</td>
</tr>
<tr>
<td>A68</td>
<td>E26</td>
<td>department</td>
</tr>
<tr>
<td>A69</td>
<td>E26</td>
<td>suppkey</td>
</tr>
<tr>
<td>A70</td>
<td>E26</td>
<td>qty</td>
</tr>
<tr>
<td>A71</td>
<td>E26</td>
<td>cost</td>
</tr>
<tr>
<td>A72</td>
<td>E26</td>
<td>date</td>
</tr>
</tbody>
</table>

Table 6.12 Mapping table entries for the ETL requirement in Figure 6.16

<table>
<thead>
<tr>
<th>MID</th>
<th>Entity</th>
<th>Mapping</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>M34</td>
<td>E28</td>
<td>pkey</td>
<td>Pkey</td>
</tr>
<tr>
<td>M35</td>
<td>E28</td>
<td>suppkey</td>
<td>suppkey</td>
</tr>
<tr>
<td>M36</td>
<td>E28</td>
<td>sysdate</td>
<td>date</td>
</tr>
<tr>
<td>M37</td>
<td>E28</td>
<td>qty</td>
<td>qty</td>
</tr>
<tr>
<td>M38</td>
<td>E28</td>
<td>cost</td>
<td>cost</td>
</tr>
<tr>
<td>M39</td>
<td>E29</td>
<td>pkey</td>
<td>pkey</td>
</tr>
<tr>
<td>M40</td>
<td>E29</td>
<td>suppkey</td>
<td>suppkey</td>
</tr>
<tr>
<td>M41</td>
<td>E29</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>M42</td>
<td>E29</td>
<td>qty</td>
<td>qty</td>
</tr>
<tr>
<td>M43</td>
<td>E29</td>
<td>cost*0.5</td>
<td>cost</td>
</tr>
<tr>
<td>M33</td>
<td>E24</td>
<td>country</td>
<td>country</td>
</tr>
</tbody>
</table>

The Mapping table entries for the document are given in Table 6.12.

The Generate SAS Model Tables screen is used to generate the SAS model tables for the requirement from Tables 6.8-6.12. The SAS_Entity table entries for the document are given in Table 6.13.
### SAS_ENTITY

<table>
<thead>
<tr>
<th>EID</th>
<th>Name</th>
<th>Process</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>E31</td>
<td>partsupp1</td>
<td>P3</td>
<td>source</td>
</tr>
<tr>
<td>E32</td>
<td>partsupp2</td>
<td>P3</td>
<td>source</td>
</tr>
<tr>
<td>E33</td>
<td>union_partsupp</td>
<td>P3</td>
<td>union</td>
</tr>
<tr>
<td>E34</td>
<td>add_date</td>
<td>P3</td>
<td>format</td>
</tr>
<tr>
<td>E35</td>
<td>convert_to_euro</td>
<td>P3</td>
<td>format</td>
</tr>
<tr>
<td>E36</td>
<td>temp_1</td>
<td>P3</td>
<td>temp_data</td>
</tr>
<tr>
<td>E37</td>
<td>temp_2</td>
<td>P3</td>
<td>temp_data</td>
</tr>
</tbody>
</table>

Table 6.13 SAS_Entity table entries for the ETL requirement in Figure 6.16

The SAS_Relationship entries for the document are given in Table 6.14.

### SAS_RELATIONSHIP

<table>
<thead>
<tr>
<th>RID</th>
<th>Entity</th>
<th>Pointer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R55</td>
<td>E30</td>
<td>IN</td>
<td>aggregate_input</td>
</tr>
<tr>
<td>R56</td>
<td>E30</td>
<td>OUT</td>
<td>copy_input</td>
</tr>
<tr>
<td>R57</td>
<td>E31</td>
<td>OUT</td>
<td>add_date</td>
</tr>
<tr>
<td>R58</td>
<td>E32</td>
<td>OUT</td>
<td>convert_to_euro</td>
</tr>
<tr>
<td>R59</td>
<td>E33</td>
<td>IN</td>
<td>temp_1</td>
</tr>
<tr>
<td>R60</td>
<td>E33</td>
<td>IN</td>
<td>temp_2</td>
</tr>
<tr>
<td>R61</td>
<td>E33</td>
<td>OUT</td>
<td>partsupp</td>
</tr>
<tr>
<td>R62</td>
<td>E34</td>
<td>IN</td>
<td>partsupp1</td>
</tr>
<tr>
<td>R63</td>
<td>E34</td>
<td>OUT</td>
<td>temp_1</td>
</tr>
<tr>
<td>R64</td>
<td>E35</td>
<td>IN</td>
<td>partsupp2</td>
</tr>
<tr>
<td>R65</td>
<td>E35</td>
<td>OUT</td>
<td>temp_1</td>
</tr>
<tr>
<td>R66</td>
<td>E36</td>
<td>OUT</td>
<td>temp_2</td>
</tr>
<tr>
<td>R67</td>
<td>E36</td>
<td>IN</td>
<td>add_date</td>
</tr>
<tr>
<td>R68</td>
<td>E37</td>
<td>OUT</td>
<td>union_partsupp</td>
</tr>
<tr>
<td>R69</td>
<td>E37</td>
<td>IN</td>
<td>convert_to_euro</td>
</tr>
</tbody>
</table>

Table 6.14 SAS_Relationship table entries for the ETL requirement in Figure 6.15

The Generate SAS ETL Code screen is used to generate the ETL code for the requirement. The generated SAS code is given in Appendix B.

### 6.4 Conclusion

In this chapter we discussed the software development platform used to implement this thesis work and illustrated the functionalities of the application. We presented two case studies and generated the logical model tables and SAS code for each. The next chapter offers the contributions and future extension to our work.
Chapter 7: Contributions and Future Work

This chapter summarizes the research contributions of this thesis and identifies the future scope of this research.

7.1 Research Contributions

The research contributions of this thesis are given below.

1. We review the literature on ETL processing and ETL process modeling. We identify the differences between OLTP and OLAP systems and provide a comparison between them.
2. We survey current research approach in ETL modeling and study their operators. We present a feature comparison between the different approaches in ETL modeling.
3. We identify a set of ETL activities that can capture the functionality of the operators identified in the literature review.
4. We define the XML logical model equivalent to each operation and implement an application that accepts user inputs to generate the XML logical model.
5. We survey the current ETL applications and choose SAS as the target application to convert the ETL requirement to for proof of concept.
6. We choose the multi-level data dictionary (MDD) approach to represent the ETL requirements in a table format and implement an application to convert the XML logical model to the table format. We leverage it to generate the SAS model tables. We implement a program to generate ETL code from the SAS model tables.
7. We choose C#.NET to develop the user-interface and Oracle to store and manipulate the MDD tables. We create an application the accepts user input to generate an XML model of ETL requirement, convert it to XML model tables, map the SAS model tables from the XML tables, and generate the SAS code for the ETL requirement.

To summarize, we identified a logical model for ETL processes and a mapping algorithm that semi-automatically generates ETL programs from the model. We identified some of the research problems in ETL logical modeling, defined the operations to be represented in an ETL logical model, created a model to represent these operations and implemented an algorithm to generate an ETL process from the logical model.
7.2 Future Work

Our contribution could be enhanced by implementing a methodology to generate the XML model from an existing ETL application code. When ETL applications that have little or no documentation need to be modified, reverse engineering the logical model from the application would enable the designer to generate the original logical model from the application. This logical model can be modified to accommodate the changes in the processing requirement and generate ETL code with minimal programmer intervention.

Another opportunity for enhancement is to develop an application that generates a graphical representation of the ETL requirements. Our work gives a point-and-click application that enables users to drag and drop ETL operators and specify the required parameters to create the XML logical model. Modifying the XML document would be easier if the process flow can be represented as a diagram. The XML document developed by our application could be enhanced by presenting the user with a block diagram-like representation that visualizes the flow of data between operators.

Our work could also be extended to include multiple ETL platforms. This enables the conversion of an ETL requirement to multiple applications, making the logical model a common representation that can be converted to any ETL platform programmatically.
References


Appendix A: XML Representation of Sample ETL Requirements

A.1 XML Representation of an ETL Requirement in a Retail Domain

The XML document below corresponds to the example given in Figure 1.1.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<ETL>

<SOURCE name = "game_sales_asia">
<OUT> union_region_sales </OUT>
<IN_SCHEMA>
  <FIELD name = "game_code" />
  <FIELD name = "store_code"/>
  <FIELD name = "currency"/>
  <FIELD name = "amount"/>
  <FIELD name = "quantity"/>
  <FIELD name = "date"/>
</IN_SCHEMA>
</SOURCE>

<SOURCE name = "game_sales_europe">
<OUT> union_region_sales </OUT>
<IN_SCHEMA>
  <FIELD name = "game_code" />
  <FIELD name = "store_code"/>
  <FIELD name = "currency"/>
  <FIELD name = "amount"/>
  <FIELD name = "quantity"/>
</IN_SCHEMA>
</SOURCE>

<SOURCE name = "store">
<OUT> join_on_storecode </OUT>
<IN_SCHEMA>
  <FIELD name = "storecode" />
  <FIELD name = "manager"/>
  <FIELD name = "city"/>
</IN_SCHEMA>
</SOURCE>

<SOURCE name = "exchange_rate">
<OUT> join_on_currency </OUT>
<IN_SCHEMA>
  <FIELD name = "currency" />
  <FIELD name = "exchange_rate"/>
</IN_SCHEMA>
</SOURCE>

<TARGET name = "sales_fact">
<IN> aggregate_input </IN>
<OUT_SCHEMA>
</TARGET>
</ETL>
```
<FIELD name = "game_code" />
<FIELD name = "city"/>
<FIELD name = "amount"/>
<FIELD name = "quantity"/>
<FIELD name = "date"/>
</OUT_SCHEMA>
</TARGET >

<TARGET name = "sales_detail_fact">
<IN>  copy_input </IN>
<OUT_SCHEMA>
<FIELD name =  "game_code" />
<FIELD name =  "store_code" />
<FIELD name = "amount"/>
<FIELD name = "quantity"/>
<FIELD name = "date"/>
</OUT_SCHEMA>
</TARGET >

<UNION name= "union_region_sales" >
<IN > game_sales_asia </IN>
<IN > game_sales_europe </IN>
<OUT> promo_items_filter </OUT>
</UNION>

<FILTER name = "promo_items_filter">
<IN> union_region_sales </IN>
<OUT> join_on_currency </OUT>
<COND> "amount != 0"  </COND>
</FILTER>

<JOIN name= "join_on_currency" >
<IN > promo_items_filter </IN>
<IN > exchange_rate </IN>
<OUT> convert_to_dollar </OUT>
<TYPE> inner </TYPE>
.KEY>currency</KEY>
<MAPPING>
<FIELD name = "game_code" expression="game_code"/>
<FIELD name = "store_code" expression = "store_code"/>
<FIELD name = "currency" expression = "currency" />
<FIELD name = "amount" expression = "amount" />
<FIELD name = "quantity" expression = "quantity" />
<FIELD name = "date" expression = "date" />
<FIELD name = "exchange_rate" expression = "exchange_rate"/>
</MAPPING>
</JOIN>

<FORMAT name= "convert_to_dollar" >
<IN> join_on_currency </IN>
<OUT> copy_input </OUT>
<MAPPING>
<FIELD name = "game_code" expression = "game_code" />
<FIELD name = "store_code" expression = "store_code" />
<FIELD name = "amount" expression = "amount*exchange_rate" />
<FIELD name = "quantity" expression = "quantity" />
</MAPPING>
</FORMAT>
<FIELD name = "date" expression = "date" />
</MAPPING>
</FORMAT>

<COPY name = "copy_input">
<IN > convert_to_dollar </IN>
<OUT> sales_detail_fact </OUT>
<OUT> join_on_storecode </OUT>
</COPY>

<JOIN name = "join_on_storecode">
<IN > copy_input </IN>
<IN > store </IN>
<OUT> sort_input </OUT>
<Type> inner </TYPE>
<Key> storecode </KEY>

<MAPPING>
<Field name = "game_code" expression = "game_code"/>
<Field name = "city" expression = "city"/>
<Field name = "amount" expression = "amount"/>
<Field name = "quantity" expression = "quantity"/>
<Field name = "date" expression = "date"/>
</MAPPING>
</JOIN>

<SORT name = "sort_input">
<IN > join_on_storecode </IN>
<OUT> aggregate_input </OUT>
<Key> gamecode </KEY>
<Key> city </KEY>
<Key> date </KEY>
</SORT>

<AGGREGATE name = "aggregate_input">
<IN > sort_input </IN>
<OUT> sales_fact </OUT>
<Key> gamecode </KEY>
<Key> city </KEY>
<Key> date </KEY>

<MAPPING>
<Field name = "game_code" expression = "game_code"/>
<Field name = "city" expression = "city"/>
<Field name = "date" expression = "date"/>
<Field name = "amount" expression = "sum(amount)"/>
<Field name = "quantity" expression = "sum(quantity)"/>
</MAPPING>
</AGGREGATE>
</ETL>
A.2 XML Representation of an ETL Requirement in a Banking Domain

The XML document below corresponds to the example given in Figure 6.15.

```xml
<?xml version="1.0"?>

<ETL>
  <SOURCE name="customers">
    <OUT>join_on_cust_id</OUT>
    <IN_SCHEMA>
      <FIELD name="cust_id" />
      <FIELD name="name" />
      <FIELD name="zip" />
      <FIELD name="age" />
      <FIELD name="credit_risk" />
      <FIELD name="years"/>
    </IN_SCHEMA>
  </SOURCE>

  <SOURCE name="accounts">
    <OUT>filter_on_type</OUT>
    <IN_SCHEMA>
      <FIELD name="balance" />
      <FIELD name="acct_id" />
      <FIELD name="cust_id" />
      <FIELD name="type" />
    </IN_SCHEMA>
  </SOURCE>

  <FILTER name="filter_on_type">
    <IN>accounts</IN>
    <OUT>join_on_cust_id</OUT>
    <COND>type != 'L'</COND>
  </FILTER>

  <JOIN name="join_on_cust_id">
    <IN>customers</IN>
    <IN>accounts</IN>
    <OUT>aggregate_on_cust_id</OUT>
    <MAPPING>
      <FIELD name="cust_id" expression="cust_id" />
      <FIELD name="name" expression="name" />
      <FIELD name="balance" expression="balance" />
      <FIELD name="age_group" expression="if age < 22 'A' else 'B'" />
      <FIELD name="end_date" expression="date() + years * 12" />
      <FIELD name="country" expression="if length(zip) = 5 'USA' else 'Canada'" />
    </MAPPING>
    <KEY>cust_id</KEY>
  </JOIN>

  <AGGREGATE name="aggregate_on_cust_id">
    <IN>join_on_cust_id</IN>
    <OUT>copy_input</OUT>
    <MAPPING>
      <FIELD name="cust_id" expression="cust_id" />
    </MAPPING>
  </AGGREGATE>
</ETL>
```
<FIELD name="name" expression="name" />
<FIELD name="total_balance" expression="sum(balance)" />
<FIELD name="age_group" expression="age_group" />
<FIELD name="country" expression="country" />
</MAPPING>

<KEY>cust_id</KEY>
</AGGREGATE>

<COPY name="filter_big_cust">
  <IN>aggregate_input</IN>
  <OUT>filter_big_cust</OUT>
  <OUT>filter_other_cust</OUT>
</COPY>

<FILTER name="filter_big_cust">
  <IN>copy_input</IN>
  <OUT>big_customers</OUT>
  <COND>"total_balance>10000"</COND>
</FILTER>

<FILTER name="filter_other_cust">
  <IN>copy_input</IN>
  <OUT>other_customers</OUT>
  <COND>"total_balance<=10000"</COND>
</FILTER>

<TARGET name="big_customers">
  <OUT>filter_big_cust</OUT>
  <OUT_SCHEMA>
    <FIELD name="cust_id" />
    <FIELD name="name" />
    <FIELD name="total_balance" />
    <FIELD name="age_group" />
    <FIELD name="end_date" />
    <FIELD name="country" />
  </OUT_SCHEMA>
</TARGET>

<TARGET name="other_customers">
  <OUT>filter_other_cust</OUT>
  <OUT_SCHEMA>
    <FIELD name="cust_id" />
    <FIELD name="name" />
    <FIELD name="total_balance" />
    <FIELD name="age_group" />
    <FIELD name="end_date" />
    <FIELD name="country" />
  </OUT_SCHEMA>
</TARGET>

</ETL>
A.3 XML representation of an ETL requirement in a Manufacturing Domain

The XML document below corresponds to the example given in Figure 6.16.

```xml
<?xml version="1.0"?>
<ETL>
  <SOURCE name="partsupp1">
    <OUT>add_date</OUT>
    <IN_SCHEMA>
      <FIELD name="pkey" />
      <FIELD name="suppkey" />
      <FIELD name="qty" />
      <FIELD name="cost" />
    </IN_SCHEMA>
  </SOURCE>

  <SOURCE name="partsupp2">
    <OUT>convert_to_euro</OUT>
    <IN_SCHEMA>
      <FIELD name="pkey" />
      <FIELD name="department" />
      <FIELD name="suppkey" />
      <FIELD name="qty" />
      <FIELD name="cost" />
      <FIELD name="date" />
    </IN_SCHEMA>
  </SOURCE>

  <FORMAT name="add_date">
    <IN>partsupp1</IN>
    <OUT>union_partsupp</OUT>
    <MAPPING>
      <FIELD name="pkey" expression="pkey" />
      <FIELD name="suppkey" expression="suppkey" />
      <FIELD name="date" expression="sysdate" />
      <FIELD name="qty" expression="qty" />
      <FIELD name="cost" expression="cost" />
    </MAPPING>
  </FORMAT>

  <FORMAT name="convert_to_euro">
    <IN>partsupp2</IN>
    <OUT>union_partsupp</OUT>
    <MAPPING>
      <FIELD name="pkey" expression="pkey" />
      <FIELD name="suppkey" expression="suppkey" />
      <FIELD name="date" expression="date" />
      <FIELD name="qty" expression="qty" />
      <FIELD name="cost" expression="cost*0.5" />
    </MAPPING>
  </FORMAT>

  <UNION name="union_partsupp">
    <IN>add_date</IN>
    <IN>convert_to_euro</IN>
    <OUT>partsupp</OUT>
  </UNION>
</ETL>
```
Appendix B: SAS Code for Sample ETL Requirements

B.1 SAS Code for an ETL Requirement in a Retail Domain

The SAS code below corresponds to the example given in Figure 1.1.

```sas
DATA temp1;
SET game_sales_asia game_sales_europe;
IF amount!=0;
RUN;

DATA temp2;
MERGE temp1 exchange_rate;
BY currency;
KEEP gamecode storecode currency amount quantity exchange_rate;
RUN;

DATA sales_detail_fact;
SET temp2;
RUN;

DATA temp2;
SET temp2;
amount = amount * exchange_rate;
KEEP gamecode storecode quantity date;
RUN;

DATA temp3;
MERGE temp1 store;
BY storecode;
KEEP gamecode city amount quantity date;
RUN;

PROC SORT DATA = temp3 OUT = temp4;
BY gamecode;
RUN;

PROC MEANS NWAY NOPRINT DATA = temp4;
CLASS gamecode city date;
VAR amount quantity;
OUTPUT OUT = sales_fact SUM(amount quantity) = amount quantity;
RUN;
```

B.2 SAS Code for an ETL Requirement in a Banking Domain

The SAS code below corresponds to the example given in Figure 6.15.

```sas
DATA temp1;
SET account;
```
IF type!='L';
RUN;

DATA temp2;
MERGE customers temp1;
BY cust_id;
KEEP cust_id name balance;
age_group= // if age<22 'A' else 'B';
end_date= // date()+years*12;
country= if length(zip)=5 'USA' else 'Canada';
RUN;

PROC MEANS NWAY NOPRINT DATA = temp2;
CLASS cust_id name age;
VAR balance;
OUTPUT OUT = temp3 SUM(balance) = total_balance;
RUN;

DATA big_customers;
SET temp3;
IF total_balance>10000;
RUN;

DATA other_customers;
SET temp3;
IF total_balance<=10000;
RUN;

**B.3 SAS Code for an ETL Requirement in a Manufacturing Domain**

The SAS code below corresponds to the example given in Figure 6.16.

DATA temp1;
SET partsupp1;
date = //sysdate;
RUN;

DATA temp2;
SET partsupp2;
cost=cost*.5;
RUN;

DATA partsupp;
SET temp1 temp2;
RUN;