International Journal of Soft Computing 10 (6): 431-436, 2015
ISSN: 1816-9503

Modified Data Transformation ETL Process for Integration of Business Systems and Creation of Data Warehouses

[1]V.N. Solovyev, [1]A.V. Prokofyev, [2]I.A. Khlamov and [1]R.G. Chesov
[1]Flexbby Solutions' LLC, 3 Zhukovskogo Str., Dolgoprudny, Moscow Region, Russia
[2]Moscow Institute of Physics and Technology (State University),
9 Institutsky Pereulok, Dolgoprudny, Russia

Abstract: This study describes the modified ETL process of transformation and import of data at which the intermediate stage of the loading data 'as is' in the target system was used. Then these data objects were transformed into target system data objects. The proposed transformation approach allows transforming the source objects into the target system objects in the background. The key advantage of the proposed approach is that a few target objects may be created from a single imported one or a few imported objects may be used for creation of a single target one which significantly facilitates the process of integration of business and analytical systems.

Key words: Data import, data transformation, python, BI, big data, API, ETL process

INTRODUCTION

One of the major issues that modern organizations face is rapid growth of structured and unstructured data that are critically important to making of managerial decisions. If we consider the IT-landscape of an enterprise the most of companies have a great number of information systems supporting automation of business processes related to data accumulation and processing (Bansal and Kagemann, 2015). For example, ERP Systems incorporating production management modules, finance, sales, marketing, clients' data, transaction accounting system assets, access rights management modules, etc., (Deneke *et al*., 2013).

On the other hand, increase in the automation level causes a number of problems while using this data the main cause of which consists in the fact that many processes that ae related from the managerial perspective are automated with the use of a few unrelated systems. Such patchwork automation method results in accumulation of data that are relevant to management in completely unrelated DB structures.

For example, the employees' data, their subordination, the regional structure of a geographically distributed company usually lay within a single system. The sales figures may appear in a few different systems. It is often the case that retail sales are managed by one system and online-sales by the second one, the channel sales by the third one. Such situation is observed in many organizations and this is primarily due to the fact that business processes evolve according to the market challenges that arise daily.

This is why many companies have to integrate new systems as the system integration rate is the main decision-making factor is.

It is often very difficult to adjust an already existing ERP System for current business tasks. This may be due to engineering imperfection of the architecture used, high requirements to the system accessibility or severe project risks.

In most cases the decision on implementation of a new system and integration thereof in the existing IT-landscape is made.

The main task that is solved at the implementation stage is searching for and use of the single valid data source and it often appears that there is no such source in the organization. As the result one has to use the tools of aggregation and transformation of source data. This situation implies certain difficulties:

- The structure of data of the system to be implemented does not match the source data structure
- Some portion of source data is kept in different systems
- In the system to be implemented the objects are formed with the use of logical operations of transformation of a few source objects

Corresponding Author: V.N. Solovyev, Flexbby Solutions' LLC, 3 Zhukovskogo Str., Dolgoprudny, Moscow Region, Russia

- High rate of data accumulation in the source systems (for example in the retail sales system) up to a few million transactions may be accumulated daily
- Changing the data structure in the source system (attribute extension, reduction)
- Storage of primary keys and searching for duplicates
- Keeping the history of objects imported their updates and changes
- Rate of data import in the target system

In this study, the approach is considered that uses the modified Extraction-Transformation-Loading (ETL) process. The transformed data from the corporate accounting system were further on used in the systemof management of indices of 'startup project' instructions.

The main task that was solved in this applied study was design of flexible tool that would allow performing transformation of the source data within the entire system life cycle. The architecture of the designed component of transformation had to support possible changes in the data structure in the source and target systems, high performance and scalability upon the increasing source data volume.

PROCEDURE

The process Extraction-Transformation-Loading (ETL) is being currently used primarily in BI Systems for design of a data warehouse (Master Data WareHouse) that are further on used by the company management for analysis of the current situation and taking managerial decisions (Simitsis and Vassiliadis, 2008; Karagiannis *et al*., 2013).

Speaking of the systems automating business processes, it did not suffice to use certain points of view and draw up analytics. In such systems, it is necessary to automate the processes that are based on the source data.

For example, the following may be referred to such systems: systems for calculation of the employees' bonuses based on execution of the variety of KPI being the result of processing of a huge amount of data accumulated in different systems. Therefore, the following purposes of use of transformed data may be specified:

- Real time analytics
- Post analytics
- Use of data for processes automation

The standard Extraction-Transformation Loading (ETL) process includes three main stages (Fig. 1):

Fig. 1: Extraction-transformation-loading process

Fig. 2: Modified extraction-transformation loading process

Extraction of data from sources (data extraction): For implementation of this process it is needed to provide the possibility of integration of the system and/or component that are in charge of the data extraction by the open integration interfaces. The interfaces used shall allow connecting through standard protocols and supporting common file formats.

Data modification (data transformation): For implementation of this process a component shall feature the developer interfaces API. API shall allow implementing the data transformation logic (for example, cleaning, verification, searching for duplicates, searching for the similar, storage of primary keys, keeping the history of downloads, keeping the history of changes, etc.). By virtue of the fact that the amount of the source data that have to be subject to transformation may increase non-linearly, the system shall feature the scalability mechanism that allows increasing the transformation performance significantly at a certain point.

Loading in the warehouse (data loading): This process represents loading of transformed data in the target data base (for example, creation of a DataWarehouse). In order to fulfill these tasks the system shall support different data bases.

However, such approach is used primarily for tasks of the data post analytics in BI Systems and creation of master data warehouses. This 'as is' approach does not suit our tasks that are close to real time analytics, business systems integration and process automation.

In this study, we proposed to introduce changes to the standard ETL-process. The designed process includes the following stages (Fig. 2):

- Extraction of source data (replication of source data)
- Loading source data in the intermediate storage of the target system
- Background transformation of source data in the target system objects

- Loading transformed data in the main storage of the target system

The proposed approach features variety of advantages. It allows significantly decreasing burden on the systems and sources and performing data uploading in the background. It allows reducing the time and computing resources significantly as simultaneous data uploading and transformation require much more computing capacities. Besides this approach provides such an advantage as asynchrony, we can transform data as they are demanded which may also reduce the general system load. Using this approach it is possible to implement a set of binary transformations that allow forming one target object from a few source ones which significantly simplifies the task of using objects located in different systems.

COMPONENT IMPLEMENTATION

The architecture of an implemented object consists of three main parts (Fig. 3): M_Importable ensures extraction and loading of source data in the DB of the target system while maintaining the source structure of objects and links, enables connection of various data sources (XML, SQL, TXT, CVS). Provides transformation at the level of representation of external objects for further use by the environment of the target system application server. Main functions of M_Importable object:

- DataMapping: representation of the external relational model in the system
- Minimal data transformation for the use by the target system environment (attribute types: dates, Boolean variables, etc.)
- Sets of possible primary keys and reference fields declarations the process of importing from pre-processed source data
- Tracking changes in imported objects (field hash sum)
- Deleting intermediate object upon successful completion of the import action

Transmutation is the object of binary transformation got at the input by the object formed with the use of M_Importable and gives the object of the business logic of the target system at the output. This object implements the applied project interface with the use of which the logic of transformation of the source object into the target one can be described. Main functions of the transmutation object:

- Relation (importable, object business logic); the relation is persistent (i.e., deletion thereof is undesirable upon successful import completion) may be considered as:
  - M:N (general case)
  - 1:N (for the fixed importable, a few objects of business logic of different classes may exist)
  - 1:1 (for the fixed Importable and class of the business logic object)

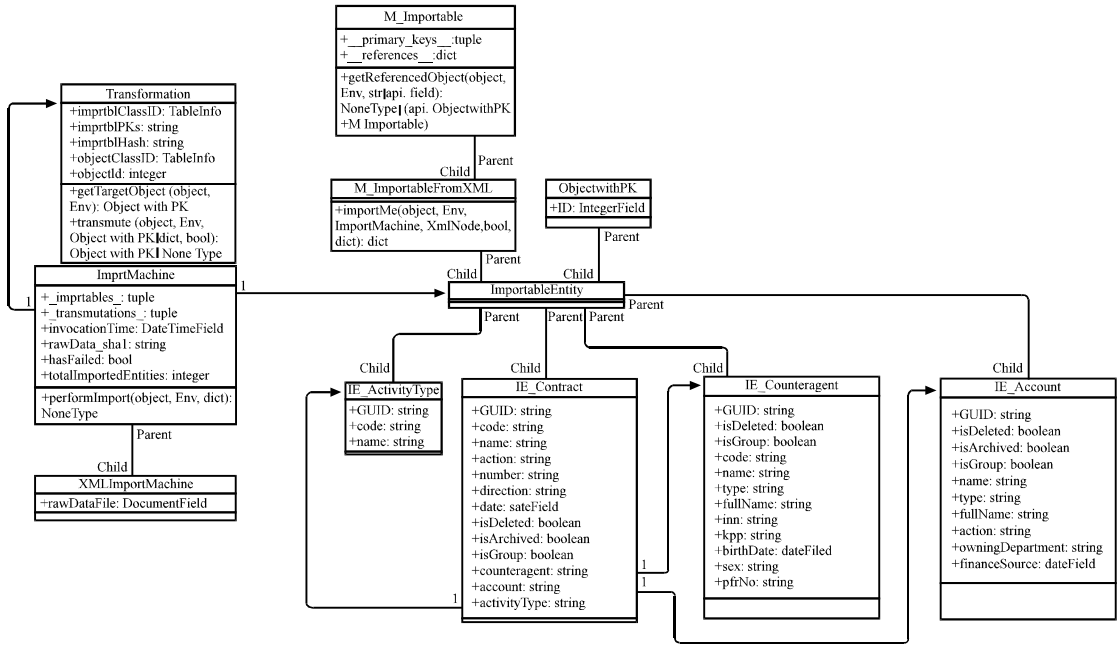

Fig. 3: Architecture of implemented objects (M_Importable, transmutation, ImportMachine)

- Logic of transformation from the specified (and single) class of importable into the specified (and single) class of business logic, i.e., transformation from the external relational model into the native one (with account of the business logic) including testing feasibility of such transformation (for example in an external relational model there may exist an object the existence of which at the same time is impossible within the native model)
- Related transformations (i.e., native relational model) are performed by means of creating an instance and calling another transmutation class (or the same one in case of hierarchical communication)
- Search (and use instead of creating a new one) by the already performed transformations for the specified Importable (with account of the hash sum, ref. M_Importable)
- Heuristic search by the already existing (added to the system in any other way beside import) target objects of business logic is used if searching by the existing transformations (previous paragraph) had no effect

ImportMachine provides auxiliary functions of the importing process for example, connection between the set of objects M_Importable and transmutation, data extraction, connection to sources, logging, process control. The main functions of the ImportMachine object:

- Grouping of Importable and Transmutation classes (i.e., the set of the simultaneously imported data)
- Receiving 'raw' data directly from the channel of communication with an external system (message queueing, DB, files)
- Pre-processing of the 'raw' data (for example, parsing and XML-representation in the object model supported by the language/standard library (for example, xml.etree for Python)); ensures close connection with the particular import mechanism (for example, from M_Importable From XML in case of XML import machine)
- Splitting the importing process into consecutive stages
- Import of pre-processed data (calling any importable possible)
- Transformation of all resulting Importable (calling all possible transmutation from the specified ones for the list of primary keys obtained as the result of the previous stage)
- Logging the importing process to provide debugging information to developers and system administrators
- On the part of the internal (and maybe external) API is the point of entry in the importing process of a specific data set by a specific data transmission channel from a specific external system

EXPERIMENTAL STUDIES

This approach was used for loading data from the external financial accounting system in the system for project startup instructions management.

The objects imported were financial transactions, contracts, counterparties. XML-files located at the message exchange server were used as the data source. Let's present performance of the contract import from the accounting system into the target one as an example. In the system the source contract is presented by the only single object 'Contract' while in the target system contract is represented by two objects 'ContactContract' contract with individuals and 'OrganizationContract' contract with legal entities. For creation of the import logic with the use of the proposed approach first the objects of the Contract type were imported, then they were transformed into two objects of the target system.

Declaration of the imported object in the Python language:

```
class IE_Contract (Object, M_ImportableFromXML):
    """Agreement"""
__primary_keys__ = (
    "GUID",
)
__references__ = {
    "parentGUID" : "IE_Contract", # dubious
    "account" : IE_Account
}
    def __init__(self):
    self.GUID = StringField("GUID")
    self.parentGUID = StringField("GUIDParent")
    self.action = StringField("Action")
    self.code = StringField("Code")
    self.name = StringField("Name")
    self.number = StringField("Number")
    self.direction = StringField("Direction")
    self.date = DateField("Date")
    self.isDeleted = BoolField("Note PometkaUdaleniya")
    self.isArchived = BoolField("Archive")
    self.isGroup = BoolField("ThisIsGroup")
    self.account = StringField("Account")
```

Declaration of logic of transformation of the imported object into the target object types:

```
class IE_Contract__Contract (Transmutation):
    __importable_class__ = IE_Contract
    __object_class__ = Contract
    def _getCounteragentImportable(self, env, importable):
        # TODO: abstract away obscure and leaky boilerplate below
        #       into M_Importable probably
        account_importable = IE_Account__BudgetCenter(env)._
getImportable(
                        env,
                        {"GUID" : importable.account.value()}
        )
        if (account_importable is None):
            return None
        counteragent_importable = IE_Counteragent__Organization(env)._
getImportable(
                        env,
```

```
                    {"GUID" : account_importable.counteragent.value()}
        )
        return counteragent_importable
    def _findTargetObject(self, env, importable):
        # avoid doubling manually created contracts having same thesis and
number
        c_iter = ObjectIterator(
            env,
            self.__object_class__.__name__,
            "thesis = %s AND docNum = %s" % (
                api.e(importable.name.value()),
                api.e(importable.number.value())
            )
        )
        if (len(c_iter) == 1):
            return next(c_iter)
    def _createTargetObject(self, env):
        contract = super(IE_Contract__Contract, self)._createTargetObject(env)
        contract.__create__(env, {})
        return contract
    def _transmute(self, env, importable, contract):
        contract.thesis = importable.name.value()
        contract.docNum = importable.number.value()
        contract.docDate = importable.date.value()
        return contract
class IE_Contract__OrganizationContract(IE_Contract__Contract):
    __object_class__ = OrganizationContract
#       compared to simply returning None from _transmute()
    # A: though, it is not possible to return None from _transmute; whether
it should?
    def _match(self, env, importable):
            counteragent_importable = self._getCounteragentImportable(env,
importable)
        return ((not counteragent_importable is None) and
                            (IE_Counteragent__Organization(env)._match(env,
counteragent_importable)))
    def _transmute(self, env, importable, contract):
        #
        org_contract = super(IE_Contract__OrganizationContract, self).\
                                _transmute(env, importable, contract)
        # find out referenced organization
            counteragent_importable = self._getCounteragentImportable(env,
importable)
                org = IE_Counteragent__Organization(env).transmute(env,
counteragent_importable)
        if (not org is None):
            org_contract.organizationID = org.id.value()
        #
        return org_contract
class IE_Contract__ContactContract(IE_Contract__Contract):
__object_class__ = ContactContract
def _match(self, env, importable):
            counteragent_importable = self._getCounteragentImportable(env,
importable)
        return ((not counteragent_importable is None)
and
(IE_Counteragent__ContactPerson(env)._match(env,counteragent_
importable)))
    def _transmute(self, env, importable, contract):
contact_contract = super(IE_Contract__ContactContract, self).\
                                _transmute(env, importable, contract)
        # find out referenced person
          counteragent_importable = self._getCounteragentImportable(env,
importable)
            person = IE_Counteragent__ContactPerson(env).transmute(env,
counteragent_importable)
            if (not person is None):
                contact_contract.contactPersonID = person.id.value()
        #
        return contact_contract

ImportMachine description for transformation of the source Contract object
into two target objects Contract_Contact and Organization_Contract
from. Contracts import IE_Contract, IE_Contract__OrganizationContract,
IE_Contract__ContactContract
from. Accounts import IE_Account__BudgetCenter
from.      Counteragents      import      IE_Counteragent__Organization,
IE_Counteragent__ContactPerson

class ImportMachine_Contracts(XMLImportMachine):
    __importables__ = (IE_Contract,) # this importable drives others, no
need to specify in here

    __transmutations__ = (
        IE_Contract__OrganizationContract,
        IE_Contract__ContactContract,
        IE_Account__BudgetCenter,
        IE_Counteragent__Organization,
        IE_Counteragent__ContactPerson,
    )
```

CONCLUSION

In the present paper the modified ETL (extraction-transformation loading) process for integration of two process control system was proposed. The proposed approach uses the intermediate stage of loading data 'as is' in the target system database. Introduction of this intermediate stage allows performing data transformation with the use of different logic. For example, forming a few objects in the target system from the source one; keeping the object transformation history and tracking changes in the source objects as well as performing repeat transformations. The proposed logic was implemented in the Python language. In order to support this logic three main components of the transformation system were implemented these are M_Importable ensures storage on the source object within the target system environment, Transmutation describes and supports binary transformation of the source object into the target one. ImportMachine supports connection of M_Importable objects with transmutation objects and general data loading logic, connection to sources, reading, etc. The proposed approach allows describing the structure of the source and target objects quite simply. The proposed component may be used as the intermediate link of integration of different systems. This approach has been already used for integration of two business process control systems in one of which the source object was a contract that was transformed into different business logic objects of the target system.

ACKNOWLEDGEMENT

The research was performed with support of the Ministry of Education and Science of the Russian Federation (work No. RFMEFI57914X0069).

REFERENCES

Bansal, S.K. and S. Kagemann, 2015. Integrating big data: A semantic extract-transform-load framework. Comput., 48: 42-50.

Deneke, W., W.N. Li and C. Thompson, 2013. Automatic composition of ETL workflows from business intents. Proceedings of the 16th International Conference on Computational Science and Engineering, December 3-5, 2013, IEEE, Sydney, NSW pp: 1036-1042.

Karagiannis, A., P. Vassiliadis and A. Simitsis, 2013. Scheduling strategies for efficient ETL execution. Inform. Syst., 38: 927-945.

Simitsis, A. and P. Vassiliadis, 2008. A method for the mapping of conceptual designs to logical blueprints for ETL processes. Decis. Support Syst., 45: 22-40.