Towards the Industrialization of Data Migration: Concepts and Patterns for Standard Software Implementation Projects

Klaus Haller

COMIT AG, Pflanzschulstr. 7, CH-8004 Zürich, Switzerland
klaus.haller@comit.ch

Abstract. When a bank replaces its core-banking information system, the bank must migrate data like accounts from the old into the new system. Migrating data is necessary but not a catalyst for new business opportunities. The consequence is cost pressure to be addressed by an efficient software development process together with an industrialization of the development. Industrialization requires defining the deliverables. Therefore, our data migration architecture extends the ETL process by migration objectives to be reached in each step. Industrialization also means standardizing the implementation, e.g. with patterns. We present data migration patterns describing the typical transformations found in the data migration application domain. Finally, testing is an important issue because test-case based testing cannot guarantee that not a single customer gets lost. Reconciliation can do so by checking whether each object in the old and new system has a counterpart in the other system.

Keywords: Data Migration, Patterns, ETL, Standard Software, ERP.

1 Motivation

In the last years, many Swiss banks replaced old, less-flexible and expensive-to-maintain core-banking systems with new ones like Avaloq or Finnova [1]. Replacing the systems requires not only setting up and customizing the new system but also migrating data like customers or accounts into the new system.[1] Data migration is necessary, but only performed once. Furthermore, it is not an enabler for business processes. Strict budgets are the consequence requiring an industrialization of the data migration development. Industrialization is often narrowed down to having a software development process like CMMI [2]. But industrialization also has a technical aspect, i.e. standardizing artifacts to be developed and concepts for constructing them.

[1] To prevent confusion we want to point out the difference between database migration and data migration. In data migration, application-related artefacts like triggers must not be migrated. Instead, data might have to be transformed to fit into the new system's database schema. In contrast, migrating a data*base* (e.g. from Microsoft SQL server to Oracle) demands not only to copy the data but also all application-related artefacts (triggers, constraints etc.).

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 63–78, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Typical examples are patters [3] or the three-tier-architecture for data-intensive applications [4]. Our vision is providing concepts allowing the industrialization of our application domain *data migration*.

The concepts in this paper reflect our experience with data migration as part of several Avaloq core-banking system implementation projects in Swiss banks. They are an outcome of COMIT's industrialization efforts for core-banking system implementation projects, the LeanStream initiative [5]. Up to now, our work on data migration concentrated on a migration infrastructure architecture [6] and on project management issues [7]. This paper complements our previous work by focusing on the industrialization of the development by equipping practitioners with blueprints for their implementations. The core concept is the ETL (extract, transform, load) process known from data warehousing [8]. We assign data migration specific objectives to each step. If different developers develop code (or use ETL tools) for the steps, the results might look completely different. However, we observed only very few different underlying patterns, which we compile in this paper. Developers should look at a migration problem in a project and remember immediately the right pattern(s) he or she has to adopt and use. By focusing on the data structure before and after the usage of the pattern, we characterize the patterns in a universally applicable way.

We organize the rest of our paper as following: Section 2 discusses related work followed by a presentation of our general migration architecture (Section 3). Section 4 explains the most important implementation techniques (language constructs and tools). Sections 5-7 describe the patterns for the different data migration steps, i.e. extract the data from the old system, transform it with respect to the new schema, and, finally, load the data into the new system. Detecting failures, especially lost data, is a major issue for data migration. We devote Section 8 to this challenge.

2 Related Work

Data migration is a practitioners' topic, i.e. only very few publications exist. However, the pioneering work comes from academia: the *butterfly approach* [9]. The butterfly approach provides a phase model with five steps: (i) analysis, (ii) development of the data mappings, (iii) building up a sample data set in the target system, (iv) migration of the system components to the target system without any data, and (v) step-by-step data migration. The key architectural element is a temporary message queue. Messages in the message queue are either "waiting" or "processed". The message queue has two operating modes. In the first mode, it processes messages in the state "waiting" respectively newly arriving messages from the old system for the migration. Processed messages switch their state to "processed" but do not leave the message queue yet. In the second mode, "processed" messages are released into the target system whereas newly arriving messages are stored in the queue in the mode "waiting". The message queue switches regularly between the two modes. The assumption is that the number of messages in the queue gets smaller and smaller.

The butterfly architecture suits well for batch processing with one or just a few message queues. The more interactive the processing and the more systems are coupled, the more difficult and expensive the butterfly approach becomes.

The most extensive work on *data migration project management* is a book written by Morris [10]. The author focuses mainly on project managers having to set-up and organize a migration project for the first time. He also provides a high-level overview about the most important technical issues. An Endava white paper has the same focus [11]. Shorter articles (e.g. [12, 13]) target the same audience, but only discuss very basic problems and pitfalls.

Tool descriptions focus on how to use (often target system specific) tools for data migration. Examples are a book about SAP's data migration tools [14] or explanations of how to migrate to a new product version or how to get away from a competitor's product [15, 16]. Furthermore, Carreira and Galhardas describe a specific language designed especially for data migration purposes [17].

Broadening the view, also data warehousing respectively the ETL process mentioned above are related [8]. Schema mapping [18], an area were tremendous research took place, is related due to the goal of mapping schemata with their attributes. Though we would have been more than happy to use (semi-)automatic techniques to reduce costs, the focus is too different. Companies buy new software to get additional functionality. This requires transforming and enriching the data being migrated from the old to the new system, which is difficult to automate.

3 Generic Migration Architecture

Each data migration project has to decide whether to follow the source-push or the target-pull paradigm [11]. **Target-pull** means migrating only data necessary for the target system, whereas **source-push** migrates *all* data of the old system into the new one. On a first glance, the latter one sounds appealing. One cannot forget any data because everything is in the target system. However, it is highly uneconomical. Usually only around 10% of the attributes have to be migrated. Users and application management understand and know these attributes well. Other attributes require more effort and an in-depth analysis of the application. Many attributes are there for pure technical reasons or to store intermediate results. So if the application is not very simple, it is too expensive to analyze every attribute. Target-pull, i.e. hunting (and migrating) attributes of the old system needed by the new one, is the option of choice. But certainly, it makes sense to have a copy of the old system in a read-only database before the old system is switched off.

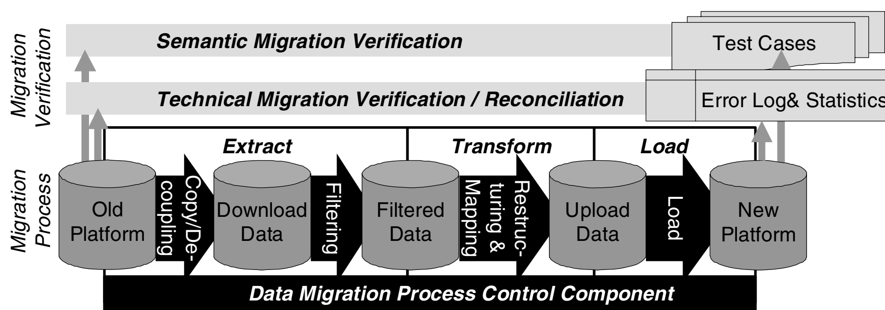


Fig. 1. Generic Migration Architecture

The centre of our data migration architecture in Figure 1 is an ETL process. The *extract step* aims firstly on decoupling the data migration project and the old system. It copies the data to a different server named download area. Thus, the project cannot affect the daily business of the bank. Secondly, the extract step identifies the data to be migrated and filters everything else. Customers, for example, not doing any business with the bank for years might not be of interest. The filtering is the one and only point where the decision is made whether an object is migrated or not. If an object passes this border, it must reach the target systems.

Next, the filtered data runs through the *transformation step*. If the database schemata of the old and new system differ, the data must be restructured in the transformation step. If the domain values are different (e.g. one system stores the currency as “USD” whereas the second one used the full name “US DOLLAR”), the transformation step accomplishes the mapping. Finally, after the transformation step, the data is *loaded* into the new system.

The ETL process illustrates the migration process of a single object type like customers or accounts. However, core-banking systems have many object types, i.e. there is one ETL process for each object type. Furthermore, often additional tasks must be performed like calculating statistics and histograms for the optimizer. Many processes and additional tasks, possibly to be performed in a certain order, make it too risky for a pure manual orchestration. Therefore, a *data migration process control component* stores the execution order. It does not necessarily perform the complete migration without manual intervention, but might automate certain steps.

After all data has been migrated, one verifies that the migrated data is correct and complete based on two complementary migration verification techniques. *Test cases* are selected sample objects, e.g. addresses and accounts of five typical and five very important customers. A tester checks manually all attributes like owner, IBAN, interest rate etc. in the old and in the new system for these objects (semantically migration verification). The second technique, *reconciliation*, is an automatic technical verification. It checks whether *all* objects have been migrated, but not whether all attributes are correct. It allows detecting e.g. five missing accounts out of ten millions.

4 Programming Paradigms

The generic data migration architecture assigns goals to each step. Fulfilling the goals can be done with different programming paradigms. The choice of the programming paradigm and the tool respectively programming language depends on each project’s situation. The source and target systems’ databases are relevant, knowledge in the project, availability of tools, the project duration etc. Therefore, we focus on the ideas of the three main paradigms (row-based implementation, set-oriented implementation, ETL tool). We discuss their different advantages on a qualitative level and provide concrete examples using PL/SQL respectively Oracle Warehouse Builder.

In the examples, the source schema has one table for natural persons (`OLD_PERSONS`) and one for juristic persons (`OLD_COMPANIES`). The target

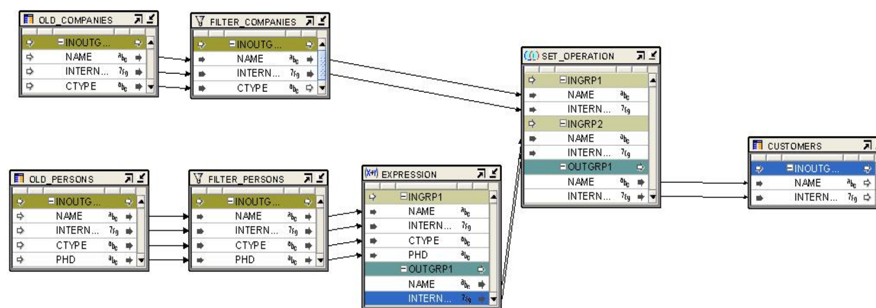
schema consists of one table CUSTOMERS. All rows of the natural and juristic persons tables are migrated if they represent customers (TYPE='Customer'). To illustrate transformations, natural persons having a Ph.D. (attribute PHD='+') get a "Dr." prefix to their names in the target table.

```
1   DECLARE
2     CURSOR c_old_persons
3      IS SELECT name, internal_id, phd FROM old_persons WHERE ctype = 'CUSTOMER';
4     CURSOR c_old_companies
5      IS SELECT name, internal_id FROM old_companies WHERE ctype = 'CUSTOMER';
6     newname varchar2(100);
7   BEGIN
8     FOR cp IN c_old_persons
9     LOOP
10     IF (cp.phd = '+') THEN newname := 'Dr. ' || cp.NAME;
11                        ELSE newname:=cp.name; END IF;
12     INSERT INTO customers (NAME, internal_id)
         VALUES (newname, cp.internal_id);
13    END LOOP;
14    FOR cc IN c_old_companies
15    LOOP
16     INSERT INTO customers (NAME, internal_id)
         VALUES (cc.NAME, cc.internal_id);
17  END LOOP;
18  COMMIT;
19  END;
```

Example 1: Row-oriented Implementation Paradigm using PL/SQL

```
1    BEGIN
2      INSERT INTO customers(name, internal_id)
3        SELECT CASE WHEN phd='+' THEN 'Dr. '||name ELSE name END, internal_id
4          FROM old_persons WHERE ctype='CUSTOMER'
5      UNION
6        SELECT name, internal_id
7          FROM old_companies WHERE ctype='CUSTOMER';
8      COMMIT;
9    END;
```

Example 2: Set-oriented Implementation Paradigm using PL/SQL



Example 3: ETL-Tool-based Implementation using Oracle Warehouse Builder

4.1 The Row-Oriented Implementation Paradigm

The row-oriented implementation paradigm specifies the migration in an imperative way using e.g. Java with JDBC or PL/SQL scripts. There is one script for each object type (addresses, persons etc.). The key idea is that each script has a loop enclosing the mapping (Example 1, lines 8-13 and 14-17). Inside the loop, a cursor accesses and processes one row per iteration of the loop. The actual implementation of the migration deals only with one source table row per iteration (lines 10-12 and line 16). The advantage of a row-based implementation is, firstly, that everyone familiar with imperative programming understands the concept. Secondly, the concept hides data-parallelism. Using cursors means that one does not have to consider the whole table at once but only the recent row. The ultimate benefit of having programming tasks with a lower complexity is that staffing the project becomes easier. However, some optimization possibilities are lost which a database optimizer might have otherwise.

4.2 The Set-Oriented Implementation Paradigm

The set-oriented implementation paradigm also uses an imperative programming model. Instead of hiding data-parallelism using cursors, it uses set-oriented SQL-statements like `SELECT`. In our example, all relevant data of table `OLD_PERSONS` respectively of table `OLD_COMPANIES` is selected and transformed in one statement (Example 2, lines 3-4 and 6-7). Complex transformations are more difficult to be implemented in a single step. Then, it might be wise implementing the transformation in more steps and storing intermediate results in temporary tables. The highly compact implementation allows database optimizers to execute the code more efficiently than row-oriented implementations. The disadvantage is the higher level of abstraction requiring programmers feeling comfortable with data set-oriented thinking.

4.3 ETL-Tool-Based Implementation Technique

ETL-tools often provide a visual programming language for defining data-flows. Data flows have one or more data sources. In Example 3 on the left, the tables `OLD_COMPANIES` and `OLD_PERSONS` are such sources. A data sink collects the result (table `CUSTOMERS`). Between the data sources and the data sink(s) operators can be placed for manipulating the data. In our example, the operators `FILTER_PERSONS` and `FILTER_COMANIES` filter objects not being customers. The operator `EXPRESSION` changes the names of persons depending on the `PHD` attribute. The companies thread and the customers thread come together at the UNION set operation implementing a `UNION`.

ETL tools provide a visual way of programming. The systems are very robust. However, complex migrations might require large data-flows which might be difficult to understand. The main obstacle against ETL tools is that learning them might take a long time if the knowledge does not already exist.

5 Extract Step Patterns

The extract step fulfils two goals. It downloads data from the productive system in the first sub-step. Afterwards, in the second sub-step, it filters the data. The purpose of the **download sub-step** is decoupling. A data migration project works on a separate project server, such that the project does not interfere with the daily operations on the old system. The decoupling requires downloading a copy of all possibly needed tables (e.g. customers, customer accounts, and banks in the example in Figure 2, but not account bookings). Generally, it is not wise to be too selective with the tables to be downloaded. Firstly, downloading a missing table later might only be allowed during dedicated service windows of the old system. Secondly, the data might become inconsistent. Assume one downloads all customer accounts on May 2nd and account balances on May 15th. If accounts are opened or closed between May 2nd and May 15th, there are suddenly accounts without account balances or vice versa. Such inconsistencies result in errors or testing problems. Thus, one missing table might require downloading a large number of tables to ensure consistency.

The **filtering sub-step** is conceptually important for the migration verification (see Section 8). Objects passing the filter must make it into the target system. This rule must be enforced strictly. Otherwise, it becomes difficult to decide whether an object was excluded on purpose or was forgotten. Such questions are especially difficult to answer if they arise weeks after the implementation.

The filtering allows excluding superfluous objects, e.g. customers who died ten years ago. Filtering also excludes objects to be migrated manually. Manual migration is more economical if there are only a few objects of a certain type (usually less then 100-1000). Also, some objects might already be in the target system. A core-banking system might e.g. already store all stock exchanges in a table. However, it is important to understand that no transformation takes place in the filtering step. But certainly, the object model in the old and new system might differ resulting in splitting object sets. Figure 2 illustrates the aspect. The old system stores all banks in one table. The new one distinguishes between the roles of banks. There are banks the bank does business with directly, e.g. because the bank has nostro accounts with them. Other banks are for reference purposes only, e.g. banks in Central Asia to which money could be sent by SWIFT. Thus, the banks of the old system are divided during the filtering into “business partner” banks and “reference data” banks.

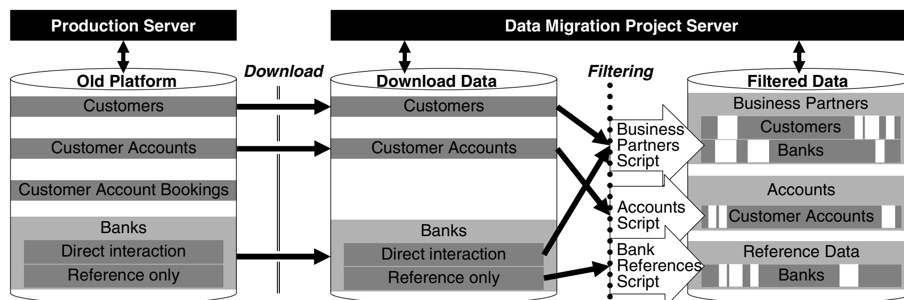


Fig. 2. Extract Step

The download sub-step copies tables and therefore does not need special patterns for the implementation. The filtering is more complex. In the following, we present the three main filtering patterns mostly needed in projects. Our presentation relies on the sample schema in Figure 3. The schema stores all accounts of the old system in table `T_ACCOUNT`. Customer accounts (in contrast to internal accounts) refer to their owner in table `T_CUSTOMER`. The third table `T_INTERESTRATE` stores the accounts' interest rates and how they changed over time. With the help of this sample schema, the three patterns are introduced quickly.

- **Attribute value based filtering.** The pattern decides whether a row is selected for each row independently of other rows or tables. One example is choosing all accounts from table `T_ACCOUNT` with `PRODUCT='SAVINGS ACCOUNT'` (Result Set 1 in Figure 3).
- **Selection table based filtering.** The pattern decides whether a row is selected based on information in a second table. The pattern determines a key for each of the rows in table one. In the second table, the pattern looks for rows having a matching key. Depending on the identified rows in the second table, the row of the first table passes the filter. An example is choosing all rows from table `T_ACCOUNT` having an owner with `BRANCH_ID=10` stored in table `T_CUSTOMER` or not having a customer as an owner (Result Set 2). It could be implemented e.g. based on a join condition like:

```
SELECT a.*
FROM T_ACCOUNT a LEFT OUTER JOIN T_CUSTOMER c
   ON a.OWNER_ID=c.CUSTOMER_ID
WHERE c.CUSTOMER_ID IS NULL OR c.BRANCH_ID=10
```

- **Aggregation based filtering.** Aggregation functions in SQL determine a value based on information in several rows, e.g. the highest value or the average. Similarly, aggregation based filtering decides whether a row is filtered based not only on the information of the row itself. It considers also other rows of the same table. A good example is choosing the latest interest rate for each account, i.e. the currently valid one (Result Set 3). Table `T_INTERESTRATE` stores two interest rates for account 1000765208, one valid from 6.8.2007, the other one from 1.1.2007. For choosing the actual valid interest rate, one must look at *all* interest rates of account 1000765208. Thus, the filter chooses the interest rate valid from 6.8.2007 and to skip the one from 1.1.2007.

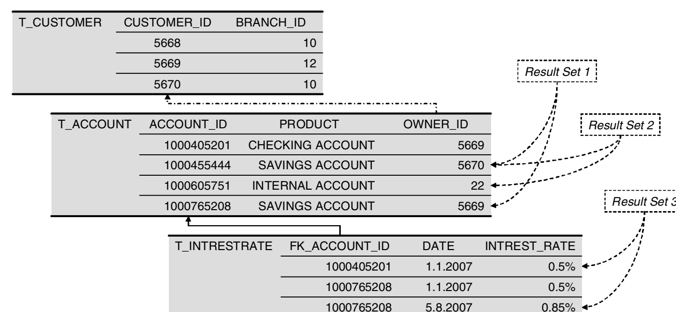


Fig. 3. Sample Tables for Filtering Patterns

6 Transformation Implementation Patterns

6.1 Pattern Group *Mapping*

Mapping is similar to working with a dictionary. You look for the value of the old system (e.g. "Germany" or "United States"). In the same row, but in a different column, you find the value for the new system ("DEU" and "USA"). The pattern group *mapping* provides two implementation patterns (Figure 4):

- **Mapping table.** A mapping table stores a value of the old system ("Germany") and the corresponding value of the new system ("DEU") in each row. Mapping tables are specified best as Excel sheets by experts with business knowledge. Then, the excel file is loaded into the database system. However, if the table is very small, it might make sense to use a CASE statement instead of a mapping table. Figure 4 provides a simple example based on the mapping table `MAP_COUNTRY`. Simple means that there is one attribute used for choosing the row (`NAME`), and one attribute is delivered back (`ISO_CODE_3`). The new value is determined by a join statement.

  ```
  SELECT c.CUSTOMER_ID, m.ISO_CODE_3
  FROM CUS_OLD c LEFT OUTER JOIN MAP_COUNTRY m
    ON c.nationality=m.name
  ```

- **Mapping function.** Some mappings are more complex and too difficult to be specified using a mapping table. A good example is temperature conversion from degree Celsius to Fahrenheit, where e.g. 3.21°C=(3.21*9/5+32)F or if the assets under management and the margin of a customer are mapped to a classification of the customer. In this situation, a mapping function is needed as represented by *f* in Figure 4. A corresponding mapping SQL statement would be:

  ```
  SELECT CUSTOMER_ID, f(CONTRIBUTION_MARGIN, ASSETS)
  FROM CUS_OLD
  ```

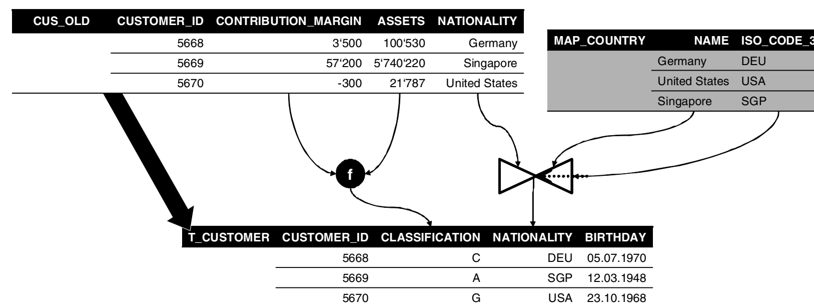


Fig. 4. Sample Tables *Mapping* Pattern Group

6.2 Pattern Group *Restructuring*

The old and the new system usually have different object models resulting in different database schemata. Restructuring patterns help transform existing data to fit into the database schema of the target system. The three main patterns are:

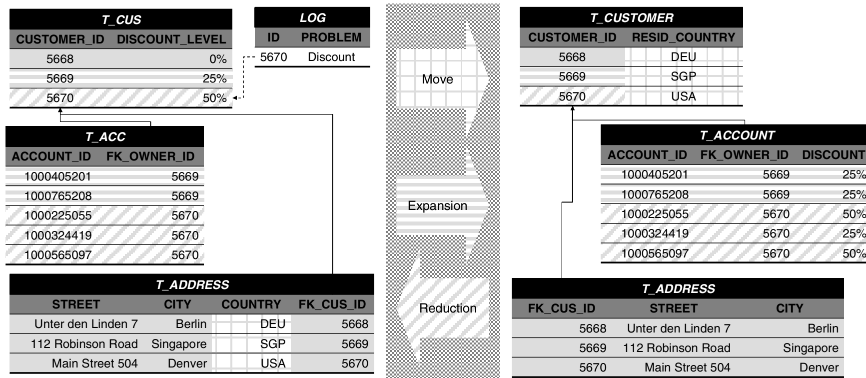


Fig. 5. Restructuring Pattern Group Examples

- **Simple Attribute Move.** The old and the new data schema store the same attribute in different tables. For example, the left schema in Figure 5 models the country of residence as address information and stores it in the address table `T_ADDRESS`. The right schema emphasizes the tax perspective. It stores the country of residence as customer information in table `T_CUSTOMER`. The simple attribute move pattern “moves” the information during the transformation step to a different table, i.e. from `T_ADDRESS` to `T_CUSTOMER`.
- **Expansion.** Both schemata have a semantically similar attribute but modeled on a different level of granularity. In Figure 5, the left schema provides a discount level for each customer (`T_CUS`). Each customer can get one discount level for all her bank charges, e.g. 0%, 50%, or even 100%. The right schema allows a more sophisticated fee modeling. Each account can have a different discount level. If the data from the left schema is migrated into the right one, the discount level information is expanded by copying the value into *each* account.
- **Reduction.** It is the opposite of expansion. The old system allows a more granular modeling than the new one. Thus, the migration is an approximation of the old data. Information gets lost. If the migration in Figure 5 takes place from right to left, customer 5670’s accounts have different discount levels in the right schema. But the customer can have only one in the left schema. Depending on the circumstances, it might be mandatory to log such loss of information (table `LOG`), because customers must be informed about changes. Thus, it is important not only to have a log table but also to have a process in place how to deal with such problems.

7 Load Patterns

When the data is transformed, the migration team loads the data into the target system. The implementation of the loading is the decision of the vendor. The vendor can choose from three patterns (Figure 6): the direct approach, the simple API one, and the workflow API one. The *direct approach* provides no API. All data is inserted

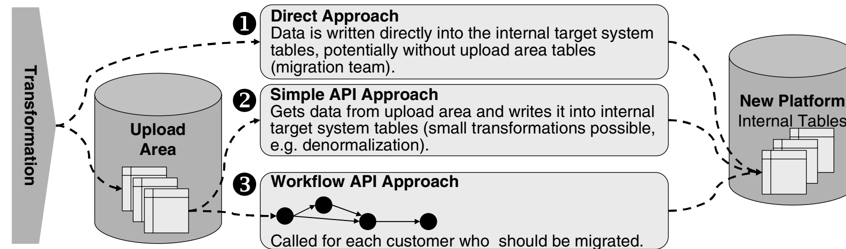


Fig. 6. Data Loading Approaches

directly into the internal tables. The *simple API approach* provides an upload area with API tables. The migration team inserts data into the API tables and invokes an API load procedure, which writes the data into the internal tables of the system. The *workflow-based API approach* also comes with an upload area with API tables. However, the API invokes the workflow separately for each object in the API table. The workflow is the same used e.g. by the GUI if new objects are entered manually.

Before we compare the patterns, we want to point out the vendor's dilemma. Customers are not willing to pay a premium for superior support for loading data during the migration. But the vendor risks his reputation if the project fails due to data migration problems. For a better understanding of the patterns, we compare them considering the dimensions in Table 1. *Error detection* considers whether the migration team gets feedback for each object whether it was migrated successfully. If not, a reason shall be given. *Conformity* compares data migrated by the data migration team and data manually entered via a GUI. The migrated data shall comply with the same requirements as manually entered data. *Vendor effort* rates the investment the vendor has to make. The *migration team training* addresses how much training the implementation team needs to work efficiently. The *migration team implementation effort* reflects the effort a trained team has for the implementation.

If the new system implements the **direct approach**, the core-banking system does not detect any migration errors. At most, some triggers or constraints might prevent the most severe mistakes. The conformity of migrated and manually entered data might be weak if the migration team does not implement exactly the same checks

Table 1. Load Step Strategies

	Technical Dimensions		Vendor Costs	Costs Migration Team	
	Error Detection	**Conformity**		**Team Training**	**Imple-mentation**
Direct Approach	No support	Not guaranteed, difficult to achieve	No effort	High, in-depth understanding of internal tables needed	High(est) due to the need to implement all checks
Simple API	Handled by API	Some confor-mity, but not guaranteed	High, if conformity desired	Low, requires good vendor documentation	Overhead for guaranteeing conformity
Workflow-based API	Handled by API	Guaranteed	Initial costs for framework, rest low	Low, requires good vendor documentation	No overhead for extra checks

applied to manually entered data respectively if not all restrictions are enforced by the database schema. However, the direct approach is the cheapest one for the vendor. It costs nothing. On the other side, the migration team needs much training (respectively learns by trial and error during the project, which is quite expensive). Also, the implementation is costly because the migration team has to implement many consistency checks.

The **simple API approach** means that the API copies the data from the API tables (possibly with some changes) into the internal tables of the system. The API can check for failures or non-compliances to the data model. The vendor either has to implement the same checks again he already uses for the GUI (high costs) or there is only a limited conformity guarantee. The benefit of an API for the migration team is that the team needs less training due to a clearly defined API. The migration team's implementation effort is restricted to missing conformity checks; therefore, it looses time by running into mistakes. The extra effort of the migration team depends how much the customization can change, because the changes require adopting the conformity checks or might be a source for mistakes.

If a vendor implements the **workflow-based API approach**, the workflows used for checking the consistency and inserting new data into the system are identical for data inserted via a GUI or data being migrated. The API uses existing workflows and returns already defined error messages. The vendor has initial costs for a framework. Afterwards, he has nearly no additional efforts no matter how many object types have to be considered. Also the migration team benefits from this approach. It has low training costs and gets data consistency guaranteed by the API.

8 Technical Migration Verification

A standard method for checking the functional correctness of applications is using *test cases*. In data migration projects, this means checking whether *all attributes of selected objects* are correctly migrated. Additionally, customers like banks or external auditors want to be sure that no data is lost. Every single customer, account, etc. must be checked. This is a task to be automated and usually termed technical migration verification or *reconciliation*. The focus is on checking relevant, *selected attributes of all objects*. Result is a reconciliation sheet. It is produced after each test data migration as a feedback for the migration team and after the final data migration. In the latter case, it enables the bank to decide whether the new system can replace the old one.

Based on our experience, we suggest that a *reconciliation sheet* consists of two parts, statistics and migration errors. Statistics provide an aggregated high level view, e.g. how many objects (accounts or also the sum of assets under management) exist in both systems and which only in one of the two. The migration errors part lists the "needles in a haystack". If three out of three million accounts are missing or have different attribute values, the error section lists keys identifying the wrong or missing objects together with the failure information ("object is missing" or "attribute BALANCE has different values").

We distinguish three patterns for deriving a reconciliation sheet (Table 2). The **top-down pattern** is the simplest one. It is used only if a project has not (or has yet not

Table 2. Reconciliation Strategies

Pattern	Idea	Identification	Usage Restrictions	Recon Sheet Section
Top-down	Counting, potentially grouped by	Object type level, based on table or characteristic attributes	No restrictions.	Statistics
Bottom-up equivalence	Comparing row by row	Key candidate are equivalent in both cases, attributes to be compared belong in both systems to the same object type	Key candidate attributes or attributes to be compared must not be involved in a restructuring	Comparison, results can be aggregated to statistics
Bottom-up fingerprint	Comparing aggregated row information	Aggregated rows have a common key attribute (but not a key for each row)	Useful in case of restructuring	Comparison, results can be aggregated to statistics

had) enough time to implement a sophisticated reconciliation. At least, it informs whether a large number of objects are missing. It creates the statistics section only by counting the objects in the old and new system, possibly considering subtypes. The accounts' reconciliation sheet in Figure 7 illustrates the aspect with the statistics for the accounts with subtype information (customer, nostro, etc.).

For identifying which single account got lost or has a wrong type, the comparison section of the reconciliation sheet must be created. The **bottom-up equivalence pattern** is one possibility. It creates a unique key for each row in the tables of the old and new system and looks whether there is a corresponding one in the other table. The attribute `ACCOUND_ID`, for example, is a good key for the tables `T_ACCOUNT_OLD` and `T_ACCOUNT_NEW`. The pattern can be implemented as following:

```
SELECT   o.ACCOUNT_ID, n.ACCOUNT_ID,
   CASE WHEN o.ACCOUNT_ID is not null AND n.ACCOUNT_ID is not null THEN 'OK'
        ELSE 'FAILED'
   END as match
FROM t_account_old o FULL OUTER JOIN t_account_new n ON n.account_id=o.account_id
```

It is mandatory to use a full outer join to identify rows in the old *or* the new system missing a counterpart in the other one. In our example, account 1000765208 exists only in the old system and 5000565097 is a phantom only existing in the new system. If the keys match, selected attributes are checked for correctness. The equivalence comparison includes *relevant and comparable* attributes. The only comparable attributes for accounts is the account type, which fails for account 9500000084. To get this result, we extend the matching join-condition as following:

```
SELECT   o.ACCOUNT_ID, n.ACCOUNT_ID,
        CASE WHEN o.ACCOUNT_ID is not null AND n.ACCOUNT_ID is not null THEN 'OK'
             ELSE 'FAILED'
         END as match,
        CASE WHEN o.ACCOUNT_TYPE = n.ACCOUNT_TYPE THEN 'OK'
             ELSE 'ERROR'
        END as equal
        FROM t_account_old o FULL OUTER JOIN t_account new n ON
n.account_id=o.account_id
```

In practice, the bottom-up equivalence pattern works well for 90-95% of the situations. The interest rates example in Figure 7 is one where it fails. A good reconciliation would use a pair <ACCOUNT_ID, LIMIT> as a key and compare the

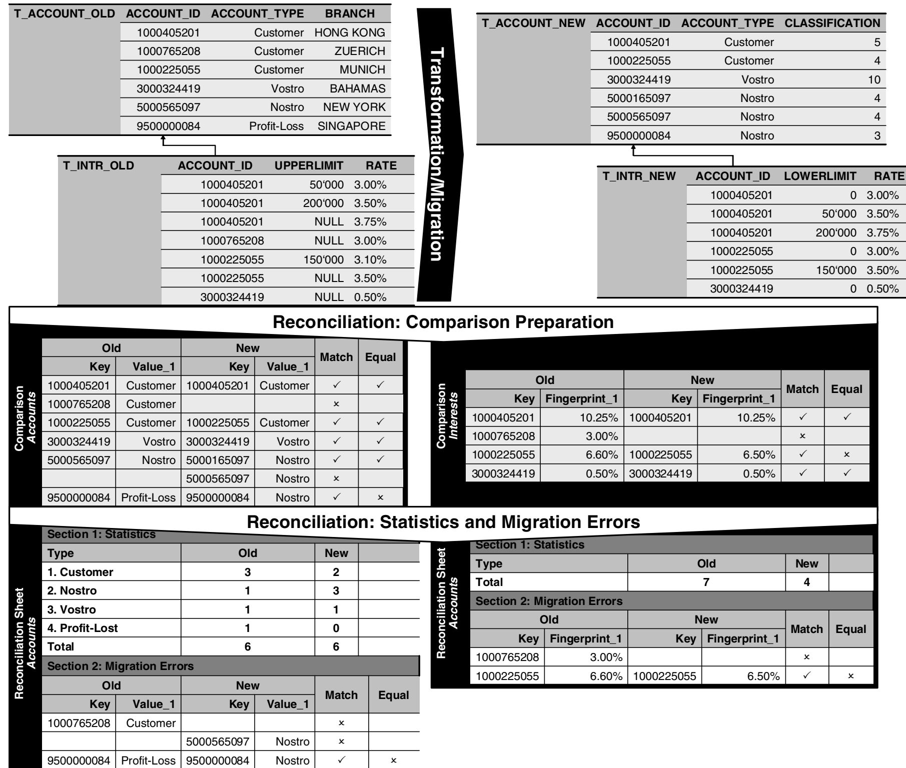


Fig. 1. Reconciliation Sheet Generation Process

interest rate as the most relevant attribute. This is not possible because the limit in the old system is an upper limit whereas the one in the new system a lower one. The worst thing one can do in such a situation is to copy the code used to transform the upper to a lower limit. If this is done, the reconciliation looks always perfect. The data migration step and the reconciliation have the same input and process the data in the same way. Thus, the results are the same no matter how wrong the transformation itself is. In such situations, the **bottom-up fingerprint pattern** helps. A fingerprint (a kind of hash value) is constructed using all relevant attributes, but it is not necessarily a semantically sensible piece of information.

We discuss now three sample fingerprints for the situation above. The simplest fingerprint is to look whether interests exist in the old and the new system for exactly the same accounts. Better would be option two, i.e. to look whether accounts with interests have always the same number of interests in both systems (like account 1000405201 having three ones). The third approach, which we used in our projects, is to sum up the interest rates for each account. It is semantically nonsense to calculate 3.00%+3.50%+3.75%=10.25% for account 1000405201. However, the rate information is included and the number of limits also influences the result. Our fingerprint does not guarantee that the limit - rate relationship is correct. However, such systematic failures should be detected by the manual migration verification,

which should include test cases with accounts with complex interest rate information. This fingerprint could be implemented as following:

```
SELECT o.ACCOUNT_ID, n.ACCOUNT_ID,
CASE WHEN o.ACCOUNT_ID is not null AND n.ACCOUNT_ID is not null THEN 'OK'
     ELSE 'FAILED'
END as match,
CASE WHEN o.FINGERPRINT= n. FINGERPRINT THEN 'OK'
     ELSE 'ERROR'
END as equal
FROM            (SELECT ACCOUNT_ID, SUM(RATE) as fingerprint
                      FROM T_INTR_OLD GROUP BY ACCOUNT_ID) o
FULL OUTER JOIN (SELECT ACCOUNT_ID, SUM(RATE) as fingerprint
                      FROM T_INTR_NEW GROUP BY ACCOUNT_ID) n
ON n.account_id=o.account_id
```

Data migration is often overlooked, but it is crucial for success when replacing an old by a new system. Our data migration architecture relies on an ETL process based data migration architecture. It defines clear objectives for the different ETL steps. Decoupling and filtering takes place in the extract step, mapping and restructuring data to fit into the schema of the target system follow in the transformation step. Getting the data into the target system with a feedback about the success takes place during the load step. Furthermore, we present the typical patterns developers find in their project such that they can rely on *simple* building blocks for their implementation. By also addressing the reconciliation challenge which is unique for data migration projects, all our concepts together form a blueprint for the implementation tasks in data migration projects. Companies can easily incorporate our work into their development processes. Thereby, they improve the standardization and industrialization of data migration in their projects.

References

1. Gabriel, C.: Plattform-Wechsel: Parforce-Übung mit weitreichenden Folgen, Schweizer Bank, Zürich (June 2007)
2. CMMI for Development, Version 1.2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006)
3. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Boston (2002)
4. Fraternali, P.: Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. ACM Computing Surveys 31(3) (September 1999)
5. LeanStream® – COMIT Implementationsmethodik, V. 3.0, Comit AG, Zürich (2007)
6. Haller, K.: Datenmigration bei Standardsoftware-Einführungsprojekten. Datenbank-Spektrum 8(25), 39–46 (2008)
7. Haller, K.: Data Migration Project Management and Standard. In: 5th Conference on Data Warehousing (DW 2008), St. Gallen, Switzerland. Lecture Notes in Informatics (2008)
8. Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 26(1), NY (1997)
9. Wu, B., Lawless, D., Bisbal, J., et al.: The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems. In: ICECCS, Como, Italy (1997)
10. Morris, J.: Practical Data Migration. British Computer Society, Swindon (2006)
11. Data Migration – The Endava Approach, White Paper, London (2006)
12. Burry, C., Mancusi, D.: How to plan for data migration, Computerworld, May 21 (2004)

13. Hudicka, J.R.: An Overview of Data Migration Methodology. Select Magazine, Independent Oracle Users Group, Chicago, IL (April 1998)
14. Willinger, J., Gradl, J.: Data Migration in SAP R/3. Galileo Press, Boston (2004)
15. Anavi-Chaput, V., et al.: Planning for a Migration of PeopleSoft 7.5 from Oracle/UNIX to DB2 for OS/390 (Red Book), IBM, Poughkeepsie, NY (2000)
16. Manek: Microsoft CRM Data Migration Framework (White Paper), Microsoft Corporation (2003)
17. Crreira, P., Galhardas, H.: Efficient development of data migration transformations. In: SIGMOD, Paris, France (2004)
18. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB Journal 10, 334–350 (2001)