Bachelor Thesis

Luciano Maiwald

Design and Implementation of a Library for Recurring ETL Imports of Reference Data in Ruby

Luciano Maiwald

Design and Implementation of a Library for Recurring ETL Imports of Reference Data in Ruby

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 30. April 2015

Luciano Maiwald

Thema der Arbeit
Design und Implementation einer Softwarebibliothek zum wiederkehrenden Import von Referenzdaten in Ruby

Stichworte
Synchronisation, Import, ETL, Extract, Transform, Load, Stammdaten, Datenmanagement

Kurzzusammenfassung
Viele der heutigen Informationssysteme benötigen Stammdaten von Drittanbietern für den reibungslosen Betrieb. In vielen Fällen müssen diese Daten beim Importieren an das Datenbankschema des Informationssystems angepasst werden. Wenn dieser Import zusätzlich auch regelmäßig (z.B. jede Nacht) erfolgt, ist es nötig die neuen Daten mit den bereits importierten zusammenzuführen, um die Konsistenz des Datenbestandes nicht zu verletzen. BeetleETL, die Softwarebibliothek, die in dieser Thesis erarbeitet und implementiert wird, ermöglicht es Nutzern diese Schematransformationen in SQL zu verfassen und übernimmt die Aufgabe des Zusammenführen der Daten.

Luciano Maiwald

Title of the paper
Design and Implementation of a Library for Recurring ETL Imports of Reference Data in Ruby

Keywords
synchronisation, Import, ETL, Extract, Transform, Load, Reference Data, Data Management

Abstract
Many of today's information systems require third party generated reference or master data in order to operate properly. Oftentimes this data needs to be transformed into a different database schema when imported into the system and in case this import happens regularly (e.g. every night), it is necessary to properly merge existing data with the newly imported dataset in order to ensure consistency. BeetleETL, the library designed and implemented in this thesis, relieves users of the complex task of merging data and enables them to express the import of third party entities if the form of SQL transformation queries.

Contents

1. Introduction

Many of today's information systems require third party generated reference or master data[1] in order to operate properly. Oftentimes this data needs to be transformed into a different database schema when imported into the system and in case this import happens regularly (e.g. every night), it is necessary to properly merge existing data with the newly imported dataset in order to ensure consistency.

Instead of using a local database dump or a remote database connection and referencing third party data directly from within the system, devising a dedicated piece of software to transform and merge the data into the system's schema guarantees a clear separation between the system's internal data and the data delivered by the third party.

An "*importer*" functions as an adapter: Changes by the third party to either the schema or the data have to be dealt with in the importer and do not affect the system directly. This enables the system to function even if the third party's data should become faulty or unavailable.

The goal of this thesis is to design and implement a software library (called *BeetleETL*) that helps users separate the business data transformation logic from the way the data is actually merged and imported into the system by generalising and abstracting common mandatory steps and making use of conventions.

1.1. Project Background

The underlying ideas for the library described in this thesis were taken from two projects developed at *mindmatters GmbH & Ko KG* which rely heavily on third party data, named *Epic-Relations* and *Mercury*. In both these projects the third party data was of such low quality that instead of maintaining the third party's entity-relationship model within the respective projects, it was decided to implement an importer to transform the data into a simplified schema with a more maintainable and consistent entity-relationship model, better suiting the applications' domains.

[1]Reference data usually only comprises of permissible values to be used by other data fields, while master data represents business objects, agreed on and shared by the enterprise [wik15]. Since the distinction between the two terms does not matter for concepts in this thesis, they will be used interchangeably.

1.2. Motivation

Understanding the concept of one-way synchronisation is simple: After synchronisation, a given **target** is identical to the **source** it has been synchronised with. Understanding the process of one-way synchronisation, however, requires understanding of multiple intricate operations and state transitions of the source's data.

Abstracting and encapsulating the complexities of any given process while exposing a simple interface helps software developers reason about their programs. Instead of being hidden by implementation detail, the intent of a program becomes clear.

The library developed in this thesis attempts to simplify the application of one-way synchronisation between tables of a relational database, enabling its users to synchronise database tables without having to know the details of the process. It is expected to be integrated into the *Mercury* project.

1.3. Extract, Transform, Load - ETL

The underlying process in *BeetleETL* for importing data is **extract, transform, load (ETL)** [wik14b]. For the purpose of this thesis these three phases are interpreted in the following way:

Extract encapsulates all steps necessary to retrieve third party data and ensure it is available to the following phases. This may include fetching the data over a network and dumping it into the **source** tables.

Transform contains all steps necessary to modify the **source** data in a way that it fits the **target** schema. These modifications may range from merely renaming column names to the complete restructuring of entities and their relations. In addition, transformations can also filter and remove invalid records completely. Other tasks include encoding values (mapping "M" to "male"), deriving newly calculated values (`total_price = quantity * unit_price`) or aggregating values[2].

Load applies the changes calculated during the transformation to the **target** database. Records are either created, updated, or, deleted accordingly.

[2]The *extract* and *transform* phases may not always be as clearly separable as stated here. There are scenarios where tasks that actually transform data may better be executed during the extract phase because of performance concerns.

1.4. Structure

Chapter 2 describes the problem domain of importing data from a third party source and showcases an algorithm to synchronise an information system's data with it.

Chapter 3 discusses considerations for designing a library that aids in the process of synchronising third party data, as well as its requirements.

Chapter 4 showcases the implementation of *BeetleETL*, highlighting its DSL and parallel execution model.

Chapter 5 closes with a summary of the thesis and explores three possible areas of improvement.

1.5. Disclaimer

1.5.1. PostgreSQL

The queries throughout this thesis have been devised and tested using the PostgreSQL, an open source object-relational database management system (DBMS). While the concepts themselves are universal and can be applied to any other DBMS, the queries depicted may use PostgreSQL specific notations and features. Differences to other DBMS' will be pointed out when necessary.

1.5.2. Importing vs. synchronising Data

Throughout the course of this thesis, there will be references to *importing data* and *synchronising data*. The process of *importing data* in and of itself can be understood as merely moving data from one location to another without regard to duplication and deletions.

In this thesis, however, it will be used synonymously with one-way synchronisation where data is expected to change in one location only and those changes will be applied to the other. Hence, for this thesis the result of importing data from a **source** to a **target** results in the **target** exactly mirroring the state of the **source** after applying business logic transformations.

1.5.3. Third Party Data & Information System

Importing data always implies moving data from a **source** to a **target**. For the purpose of illustration, the term **source** is used synonymously with *third party data*. The terms *information system*, *system* and, *application* always refer to the **target**, the environment being imported into.

2. Developing the Algorithm

This chapter introduces the concepts, data structures and algorithms used to import reference data from a third party source into an information system.

Given two distinct relational databases or sets of tables where one is the **source** (third party schema) and one is the **target** (information system schema), the task is to synchronise the **target** with the **source** while applying a set of defined transitions.

2.1. Separation of Primary Key Namespaces

One of the primary concerns when importing third party reference data is the separation of primary key namespaces.

Consider importing entities from a third party data source. While these entities may have uniquely identifying primary keys, using these primary keys to reference these entities in foreign key relationships on tables wholly owned by the information system introduces tight coupling to the third party data into the system. If the third party chose to restructure their data (e.g. rename tables or columns), this could have consistency breaking consequences for the information system. It may also introduce performance penalties when primary keys are poorly chosen by the third party.

This coupling may be an acceptable tradeoff when importing only one source of third party data. When more than one sources are being imported into a single table, however, their primary key namespaces may overlap, creating primary key collisions. These collisions could potentially break the system or at least prohibit the import of the colliding data. Fixing the consequences of such primary key collisions can become a very complex and time-consuming task and should therefore be avoided.

There are ways to counteract this coupling to third party data. The introduction of a separate mapping table is one of them: This table stores the primary keys of the third party data (referred to as `external_id` in this thesis) along with the primary key generated and assigned by the information system.

Another way is to add an `external_id` column directly to entity tables, automatically storing the third party namespace ID when the record is created by the import process.

In both cases however, the `external_id` can be used in subsequent applications of the import process to identify the record and calculate changes that have to be made to synchronise its data with the third party.

In the simplest case the `external_id` for a given entity is the primary key of its **source** table. When separating multiple **target** entities stored in a single **source** table, though, the `external_id` can become arbitrarily complex, using multiple columns in its calculation in order to uniquely identify all entities **source** table's record produces.

When importing data from multiple third party sources, an `external_id` column alone is not enough to uniquely identify records in subsequent applications of the import process. This scenario requires another column (either located in the aforementioned mapping table or entity table itself) that identifies the source of the entity, referred to as `external_source`.

2.2. An Introductory Example

The way *BeetleETL* synchronises data is best explained using examples and actual SQL statements. Listings 2.1 and 2.2 show SQL statements for two tables: `employees` and `managers`. Consider the `employees` table to be the **source** (e.g. third party data provider or legacy system) and the `managers` table to be the **target** (i.e. the information system being developed).

```sql
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name CHARACTER VARYING(255),
  last_name CHARACTER VARYING(255),
  manager BOOLEAN
);
```

Listing 2.1: `employees` table in the **source**

```sql
CREATE TABLE managers (
  id SERIAL PRIMARY KEY,
  name CHARACTER VARYING(255)
);
```

Listing 2.2: `managers` table in the **target**

Imagine a business requirement for the transformation of third party data was extracting managers from the `employees` table and merging first- and last name of every manager into a single field. Devising a query that imports only the specified records from the **source** into the information system is a trivial task. It would insert a new record with a newly generated, unique primary key into the `managers` table for every record in the `employees` table and would most likely look close to the SQL query depicted in listing 2.3.

```sql
INSERT INTO managers (name)
SELECT e.first_name || ' ' || e.last_name
FROM employees e
WHERE e.manager IS TRUE;
```

Listing 2.3: Trivial query for importing managers

However, if the **source** data was subject to change and it was necessary for the target to stay in sync with it, the aforementioned query would not be sufficient. Especially if system data relied on it with foreign key references. This query would produce duplicates of all records that had already been imported in a previous run.

This duplication could be accounted for by adding a new field to the `managers` table that stores the `employees`' primary key, as shown in listing 2.4, as well as separating the job of importing the data into multiple queries: One query to import records that were added to the **source** in the time between the last import and this one and therefore are not referenced in the `managers` table (listing 2.5), and another one to update existing records in case they were altered (listing 2.6).

```sql
CREATE TABLE managers (
  id SERIAL PRIMARY KEY,
  name CHARACTER VARYING(255),
  external_id INTEGER
);
```

Listing 2.4: Extended `managers` table in the **target**

```sql
INSERT INTO managers (name, external_id)
SELECT
  e.first_name || ' ' || e.last_name,
  e.id
FROM employees e
WHERE e.manager IS TRUE
  AND NOT EXISTS (
    SELECT 1
    FROM managers m
    WHERE m.external_id = e.id
  );
```

Listing 2.5: Import non-existing managers into the **target**

```sql
UPDATE managers m
SET
  name = e.first_name || ' ' || e.last_name
FROM employees e
WHERE m.external_id = e.id;
```

Listing 2.6: Update existing managers

This solution has two major problems: One being that if these two queries were to be run manually whenever necessary, the order in which they are run has implications on execution time. If the import of new records (listing 2.5) is run before the update of existing ones (listing 2.6), records that were just imported would be updated as well, resulting in obsolete and potentially time-consuming work. While this may sound like an easy enough problem to fix (e.g. using a function or script), the queries themselves do not convey their order-dependence, leaving it to developers to explicitly explain and record this implicit circumstance.

Another, and arguably more severe, problem with this solution is the duplication of business logic. Consider the way, managers' names are calculated from the employee data (`name = e.first_name || ' ' || e.last_name`) and imagine instead more complex calculations. The logic for this calculation would have to be duplicated in the inserting as well as the updating query, violating the DRY principle[1] and thus increasing maintenance work if things had to change.

It becomes clear that this approach will only serve the most basic synchronisation needs, but will not solve the problem of fully synchronising data from the **source** to the **target**: It duplicates business logic and introduces implicit order dependencies between its sub-tasks.

Further, it does not yet address the situation where data has been removed from the **source** by the third party and should also be removed in the information system. While this could easily be added, it underlines another downside to this approach: Importing a larger amount of tables results in an even larger number of different queries specific to the data at hand that have to be maintained.

2.3. Separating Business- & Import Logic

As the previous chapter shows, in order to synchronise system data with third party data, a new approach is necessary. One that properly separates business- and import logic. This section introduces the concept of a staging area where third party data is first gathered and then applied to the system's database in two separate phases.

In contrast to the method presented in the previous chapter, this method complies with the separation of phases described by the ETL process (see 1.3): Business logic describes measures necessary to convert data into the desired structure and hence is part of the **transform** phase in ETL. The logic to actually apply the changes to the system's data is part of the **load** phase.

Again, an example using actual SQL statements and data is best suited to explain synchronising data using stage tables. Given the `employees` **source** table (listing 2.7) and `managers` **target** (listing 2.8) table from the previous example:

[1]Don't repeat yourself (DRY) is a principle of software development, aimed at reducing repetition of information of all kinds. It is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." [wik14a]

```sql
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name CHARACTER VARYING(255),
  last_name CHARACTER VARYING(255),
  manager BOOLEAN
);
```

Listing 2.7: `employees` table in the **source**

```sql
CREATE TABLE managers (
  id SERIAL PRIMARY KEY,
  external_id INTEGER,
  name CHARACTER VARYING(255)
);
```

Listing 2.8: `managers` table in the **target**

In addition to the previous two tables a **stage** table is added to the database (listing 2.9). This table is defined by a schema that looks quite similar to the corresponding **target** table in that it contains the same identity columns (i.e. `id`, `external_id`) and payload columns (i.e. `name`).

```sql
CREATE TABLE stage_managers (
  id INTEGER,
  external_id INTEGER,
  name CHARACTER VARYING(255),
  transition CHARACTER VARYING(255)
);
```

Listing 2.9: `stage_managers` table in the **stage**

It also has two notable differences: Even though the **stage** table has an `id` column, it is neither automatically generated, nor used as the primary key. It also has an additional `transition` column, the use of which will become clear as the example progresses.

2.3.1. Transforming Data

With the necessary tables in place, the next step is to fill the **stage** table with transformed data, applying all the defined business logic. Table 2.1 shows the example data in the `employees`

source table.

id	first_name	last_name	manager
1	Rose	Salazar	TRUE
2	Paul	Lade	FALSE
3	Harold	Davis	TRUE
4	Gladys	Hill	TRUE
5	Ashley	Brown	FALSE
6	Amy	Castillo	FALSE

Table 2.1.: Source data in the `employees` table

Listing 2.10 shows the SQL query necessary to fill the `stage_managers` **stage** table. Apart from the `external_id`, it quite closely resembles the query devised in the introductory example (listing 2.3) and merely contains the business relevant transformations of only fetching managers and merging their first- and last name into a single field.

```sql
INSERT INTO stage_managers (name, external_id)
SELECT
  e.first_name || ' ' || e.last_name,
  e.id
FROM employees e
WHERE e.manager IS TRUE;
```

Listing 2.10: Moving and transforming data into `stage_managers`

The result of executing this query is shown in table 2.2.

id	external_id	name	transition
	1	Rose Salazar	
	3	Harold Davis	
	4	Gladys Hill	

Table 2.2.: Data in the `stage_managers` table

2.3.2. Calculating Changes

This section explains the necessity of the `stage_managers` table's `transition` column. Instead of applying changes directly to production data as in the introductory example, every record in the **stage** table is assigned one of the following transitions: CREATE, UPDATE, DELETE, REINSTATE. This transition then dictates how the record is applied to production data.

The following sections explain how the value of the transition field is calculated for CREATE, UPDATE and, DELETE. Though it is evident what its purpose is, REINSTATE is a special case that is not relevant to this example and will be explained at a later point.

In order to calculate the value for the `transition` field, the **stage** table's contents are compared to the **target** tables's contents. Table 2.3 shows the contents of the **target**'s `managers` table assumed for this example. This represents the information system's production data before the import process is started.

id	external_id	name
1	1	Rose Nelson
2	3	Harold Davis
3	6	Amy Castillo

Table 2.3.: Data in the **target**'s `managers` table

Creating Records

The value CREATE is assigned to the `transition` field of all records in the **stage** table that have no corresponding records in the **target** table based on their `external_id`. Listing 2.11 shows the query necessary to accomplish this.

```sql
UPDATE stage_managers AS stage
SET transition = 'CREATE'
WHERE NOT EXISTS (
  SELECT 1
  FROM managers AS target
  WHERE target.external_id = stage.external_id
);
```

Listing 2.11: Assigning CREATE to the `stage_managers` table

As with the transformation query in listing 2.10 resembling the initial import query from the introductory example, this query closely resembles that in listing 2.5. The **WHERE** condition for finding the relevant records is identical. However, it does not contain any of the business logic needed to transform the data.

Since the **target** table in this example does not contain a record for *Gladys Hill*, the record in `stage_managers` has to be updated to reflect that a new record has to be created in `managers`. Listing 2.4 shows the state of the `stage_managers` table after running this query.

id	external_id	name	transition
	1	Rose Salazar	
	3	Harold Davis	
	4	Gladys Hill	CREATE

Table 2.4.: Data in the `stage_managers` table after assigning CREATE

Updating Records

To update existing records in the **target** table, the corresponding **stage** table records are assigned the UPDATE transition. This transition is assigned to all records in the `stage_managers` table that contain values that are different from the corresponding **target** table `managers`.

In this example the `name` field is the only attribute used to discern whether **target** table records have to be updated. A more complex scenario would of course respect multiple fields in the comparison.

```sql
UPDATE stage_managers AS stage
SET transition = 'UPDATE'
WHERE EXISTS (
  SELECT 1
  FROM managers AS target
  WHERE target.external_id = stage.external_id
    AND target.name IS DISTINCT FROM stage.name
);
```

Listing 2.12: Assigning UPDATE to the `stage_managers` table

Comparing the **stage** and **target** tables from this example, it can be seen that *Rose Nelson* (`managers`) has changed her name to *Rose Salazar* (`stage_managers`). Listing 2.12 shows the

SQL query to assign the UPDATE transition value, table 2.5 shows the contents of the **stage** table after executing it.

id	external_id	name	transition
	1	Rose Salazar	UPDATE
	3	Harold Davis	
	4	Gladys Hill	CREATE

Table 2.5.: Data in the `stage_managers` table after assigning UPDATE

Deleting Records

In contrast to the previous steps, deleting records is not as straightforward. Because the contents of the **stage** tables can be understood as an execution plan for applying changes to the **target**, instead of updating existing records in `stage_managers`, deletion records or "*death certificates*"[2] are inserted.

Again, comparing the **stage** and **target** tables from this example reveals that there is no record of *Amy Castillo* in `stage_managers`. This means that the record of her has either been removed from the third party data or no longer conforms to the filters defined by business logic. Therefore it has to be removed from the information system's `managers` table, as well.

The query to add these deletion records to the **stage** table is shown in listing 2.13, the result of it in table 2.6.

```sql
INSERT INTO stage_managers (external_id, transition)
SELECT
  target.external_id,
  'DELETE'
FROM managers AS target
LEFT OUTER JOIN stage_managers AS stage
  ON (stage.external_id = target.external_id)
WHERE stage.external_id IS NULL;
```

Listing 2.13: Assigning DELETE to the `stage_managers` table

[2]The term was taken from Andrew S. Tanenbaum's explanation of propagation of deletions in epidemic algorithms for multicast communication in distributed systems [TvS06].

id	external_id	name	transition
	1	Rose Salazar	UPDATE
	3	Harold Davis	
	4	Gladys Hill	CREATE
	6		DELETE

Table 2.6.: Data in the `stage_managers` table after assigning DELETE

Note that for deletion records the actual data is not relevant and can be omitted. Only the `external_id` and the transition is required in order for the `managers` records to be deleted in the next phase.

2.3.3. Applying Changes

With all transitions fields calculated, the changes can be applied to production data. In order to do this, the **stage** table records first need to be mapped to their corresponding **target** records in `managers`. After that, another set of queries actually imports the data into the system.

Assigning System IDs

For **stage** table records that represent newly created records (i.e. have been assigned the CREATE transition), a primary key has to be procured from the **target** table's ID sequence[3]. This is the same sequence used to assign primary keys during a regular SQL **`INSERT`** so subsequent **`INSERT`**s do not produce primary key collisions.

For all other **stage** table records that have a corresponding record in their **target** table (i.e. with transitions UPDATE, DELETE and, REINSTATE), assigning system namespace IDs is as trivial as joining the two tables via their `external_id` columns.

Combining these two cases of assigning system namespace IDs results in the query depicted in listing 2.14).

[3]In PostgreSQL, this sequence is automatically generated during the creation of a table for every column of type `SERIAL`. This compares to the AUTO_INCREMENT option in MySQL databases and defining a trigger using a sequence in Oracle databases.

```sql
UPDATE stage_managers AS updatable
SET id = COALESCE(target.id, nextval('managers_id_seq'))
FROM stage_managers AS stage
LEFT OUTER JOIN managers AS target
  ON (stage.external_id = target.external_id)
WHERE updatable.external_id = stage.external_id;
```

Listing 2.14: Assigning IDs to **stage** table records

Listing 2.7 shows the result of assigning **target** table IDs to `stage_managers`.

id	external_id	name	transition
1	1	Rose Salazar	UPDATE
2	3	Harold Davis	
4	4	Gladys Hill	CREATE
3	6		DELETE

Table 2.7.: Data in the `stage_managers` table after assigning IDs

Loading Data

Actually loading the data into the **target** table is trivial after the transitions and **target** IDs are in place. Depending on the assigned transition, one of the following queries is used to load new records (listing 2.15), update existing records (listing 2.16) and, delete removed records (listing 2.17).

```sql
INSERT INTO managers (id, external_id, name)
SELECT id, external_id, name
FROM stage_managers
WHERE transition = 'CREATE';
```

Listing 2.15: Loading CREATE records

```sql
UPDATE managers AS target
SET name = stage.name
FROM stage_managers AS stage
WHERE stage.id = target.id
  AND stage.transition = 'UPDATE';
```

Listing 2.16: Loading UPDATE records

```sql
DELETE FROM managers As target
JOIN stage_managers AS stage
  ON (stage.id = public.id)
WHERE stage.transition = 'DELETE';
```

Listing 2.17: Loading DELETE records

Table 2.8 shows the contents of the **target**'s `managers` table after running these last queries. It has now been synchronised with the state dictated by the third party source data after transforming it using the given business logic, concluding this example.

id	external_id	name
1	1	Rose Nelson
2	3	Harold Davis
4	4	Gladys Hill

Table 2.8.: Data in the **target**'s `managers` table

2.3.4. Abstraction

This example shows that the information system's data can be synchronised with a third party data source while separating the concerns of applying business logic and synchronisation itself.

Business logic is encapsulated in a single query for all data transitions. Potential changes to business logic would only have to be made in the transformation query filling the **stage** table.

As for the import logic, take for instance the query to assign the UPDATE transition: The only characteristics tying it to the specific **stage** and **target** tables of this example are the names of the tables and the list of attributes to compare.

Listing 2.18 shows the original query from the example, listing 2.19 the same query for another table and with a different set of attributes.

```
UPDATE stage_managers As stage
SET transition = 'UPDATE'
WHERE EXISTS (
  SELECT 1
  FROM managers AS target
  WHERE target.external_id = stage.external_id
    AND target.name IS DISTINCT FROM stage.name
);
```

Listing 2.18: Original query for assigning UPDATE

```
UPDATE stage_departments AS stage
SET transition = 'UPDATE'
WHERE EXISTS (
  SELECT 1
  FROM departments AS target
  WHERE target.external_id = stage.external_id
    AND
      (target.name, target.address, target.organisation)
      IS DISTINCT FROM
      (stage.name, stage.address, stage.organisation)
);
```

Listing 2.19: Query for assigning UPDATE to a different entity

This demonstrates that with the exception of business logic transformations, every step of the process can be generalised to work with any entity of third party data and is not specific to the tables and data in this example. Once the import data has been moved into the **stage**, all the following operations on the data are very similar.

In addition, the process outlined in this example is idempotent. Any further iterations of applying the process with the same third party data would produce the same state of synchronisation between system data with the given third party data.

This can be observed in the previous example in the fact that the record of *Harold Davis* is not changed during the course of it. The record already existed in the **target** table and was not changed in the process of importing, because it was not changed in the third party data.

Summary of the Process

In summary, the complete process of importing an entity from the **source** to the **target** comprises of the four following steps:

1. Transform the data using rules defined by business logic and move the filtered data into the corresponding **stage** table (`transform`).
2. Calculate the transitions necessary for every record in the stage table by comparing the differences of payload columns of the **stage** and **target** tables (`table-diff`).
3. Assign the corresponding system-namespace primary key to each record in the **stage** table (`assign-ids`).
4. Apply the calculated changes to the **target** table (`load`).

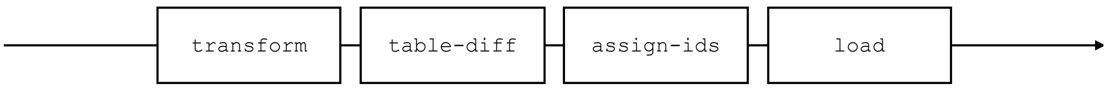

Figure 2.1.: Import process for a single entity

2.4. Deleting and Reinstating Data

In the previous example data that has been assigned the transition DELETE in the **stage** table is actually deleted from the **target** database. While this may be desirable in some circumstances, it introduces the risk of breaking consistency when data owned by the information system still references deleted records via foreign key relations.

Also, if records were removed from the third party source or the application of business logic transformations were to filter out previously existing records (as it was the case with the record of *Amy Castillo* in the previous example) and then reinstated in a subsequent execution of the import process, a new **target** table record would be created for it.

Staying with the example, if a set of entities in the information system still referenced *Amy Castillo*'s record after its deletion and changes to the third party data reinstated her into the `managers` table, all existing references to her previous records would not be updated and still contain a foreign key reference to the deleted record.

As a solution, instead of actually deleting records from the **target** tables when their corresponding **stage** table entries have been assigned the DELETE transition, they can be merely marked as deleted using a flag. This practice is called *logical* or *soft deletion*. Consequently, the handling of deleted records is moved from the DBMS to the surrounding software stack of the information system. Business logic in the application layer would then determine whether a given record should for instance be presented to a user or included in calculations.

This change in behaviour, however, requires all queries dealing with the assignment of transitions to **stage** table records to incorporate a check for this flag (see listing 2.20 as an example). It also explains the purpose of the REINSTATE transition: Instead of creating a new record, the soft-deletion flag is removed from **target** table records when their corresponding **stage** table records have been assigned the REINSTATE transition. Listing 2.21 shows the query necessary to assign REINSTATE to `stage_managers` from the previous example.

```sql
UPDATE stage_departments AS stage
SET transition = 'UPDATE'
WHERE EXISTS (
  SELECT 1
  FROM departments AS target
  WHERE target.external_id = stage.external_id
    AND target.deleted_at IS NULL
    AND target.name IS DISTINCT FROM stage.name
);
```

Listing 2.20: Updated query for assigning UPDATE with soft-delete

```sql
UPDATE stage_managers AS stage
SET transition = 'REINSTATE'
WHERE EXISTS (
  SELECT 1
  FROM managers AS target
  WHERE target.external_id = stage.external_id
    AND target.deleted_at IS NOT NULL
);
```

Listing 2.21: Assigning REINSTATE to the `stage_managers` table

2.5. Importing Data with Foreign Key References

The examples depicted so far act on the assumption that third party data only contains a single entity. However, third party data might also consist of complex hierarchies, where for one imported entity a number of other entities have to be imported that refer to the former, by means of foreign key references.

All references from one entity to another in the third party data use primary keys from the third party's primary key namespace. Since these primary keys are translated into the information system's primary key namespace during the import, the referencing entities' foreign keys have to be translated as well.

Consider the following example: `organisations` and `departments` are to be imported into the information system and `departments` belong to `organisations`. Tables 2.9 and 2.10 show the data in the **source** tables for this example. In order to focus on the handling of foreign key references, no additional business logic transformations will be applied in this example.

id	name
1	Apple
2	Google

Table 2.9.: Contents of `organisations` in the **source**

id	organisation_id	name
1	1	iPhone
2	1	MacBook
3	2	Search
4	2	Gmail

Table 2.10.: Contents of `departments` in the **source**

2.5.1. Mapping Relations

Translating foreign key relations from one primary key namespace into another requires modifications to the **stage** table of the referencing entity on the one hand and modifications to the process of importing on the other.

As was necessary for the primary key, it is now necessary to resolve the third party's foreign key to the corresponding one in the information system's namespace. Similar to the pair of `id` and `external_id`, i.e. the entities' primary key in the information system's namespace and in the third party's namespace, respectively, a set of columns for foreign keys has to be added: One of the foreign key in the information system's namespace (`organisation_id`) and one for the third party's namespace (`external_organisation_id`).

This requires that the `transform` step for importing departments also imports the `external_organisation_id`, in addition to all previously defined columns. Listing 2.22 shows the transformation necessary to fill the **stage** table and table 2.11 shows the result of this query as well as the modified structure of `stage_departments`.

```sql
INSERT INTO stage_departments
  (name, external_id, external_organisation_id)
SELECT
  source.name,
  source.id,
  source.organisation_id
FROM departments AS source;
```

Listing 2.22: Moving data into `stage_departments`

id	external_id	organisation_id	external_organisation_id	name	transition
	1		1	iPhone	
	2		1	MacBook	
	3		2	Search	
	4		2	Gmail	

Table 2.11.: Data in `stage_departments` after transformation

Considering the data shown in table 2.12 is present in the `stage_organisations` table at the point in time where foreign key relations are mapped for departments. The foreign keys in `stage_departments` can then be assigned by using the query in listing 2.23. The result of executing this query is depicted in table 2.13.

id	external_id	name	transition
12	1	Apple	
34	2	Google	

Table 2.12.: Contents of `stage_organisations`

```sql
UPDATE stage_departments AS current
SET organisation_id = foreign.id
FROM organisations AS foreign
WHERE current.external_organisation_id = foreign.external_id;
```

Listing 2.23: Mapping relations for `stage_departments`

id	external_id	organisation_id	external_organisation_id	name	transition
	1	12	1	iPhone	
	2	12	1	MacBook	
	3	34	2	Search	
	4	34	2	Gmail	

Table 2.13.: Data in `stage_departments` after mapping relations

With the `organisation_id` column properly filled with the **target** namespace's foreign keys, the process can continue as it did in the previous example, by comparing the **stage** with its corresponding **target** table and the assignment of transitions (`table-diff`).

During this comparison the `organisation_id` column is treated as a regular payload column. The `external_organisation_id` column, however, is disregarded since it has no equivalent on the **target** table.

In general, for all dependencies that a given entity has to other entities, it requires a column for the dependency's foreign key in the **target** namespace (`dependency_id`) and a column for the dependency's foreign key in the **source** namespace (`external_dependency_id`) in its **stage** table to be present.

Further, in order to complete the import process for an entity with dependencies, it is required that the dependencies' import process has already completed the `assign-ids` step so that their **target** namespace primary key are known. This then enables the execution of the query shown in listing 2.23, adapted to every given dependency. After that the process of importing the entity can continue with the `table-diff` step.

2.6. Completing the Import Process

With the process resulting from the example in section 2.3 and the mapping of foreign keys in place, the complete process for importing third party entities is established and depicted in figure 2.2.

For entities without foreign key references, the `map-relations` step effectively results in a no-op, making the process applicable for entities with and without foreign key references, alike.

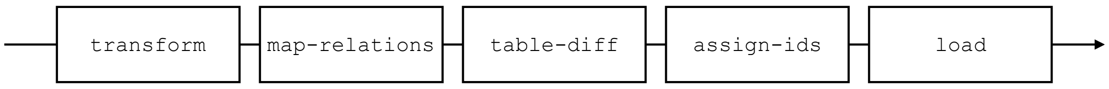

Figure 2.2.: Complete process of importing a single entity

The introduction of relationships between entities has also introduced order dependence into the process of importing multiple related entities. An entities dependencies must be imported before the entity itself can be imported.

3. Design of the Library

This chapter describes design considerations and requirements for *BeetleETL*, the library developed in this thesis.

3.1. Extracting functionality

Extracting specialised functionality from the ETL process requires decisions about what parts of the process can be generalised and abstracted and what parts are specific to the surrounding project. Further, it is important to consider what benefits are to be gained from extracting functionality at all. A small subtask of the process, for example converting Microsoft Access database files into CSV files, might not warrant the extraction into a library.

If the scope of the library's functionality is chosen too narrow, this might impede the usefulness of it. If it is too broad, it might be harder for potential users to understand what problems the library solves and what value it brings.

While all three phases of the ETL process are necessary to fully import data, the focus of this thesis is only on *transform* and *load*. **Source**-data is assumed to be available in the form of relational database tables. The task of procuring data and potentially converting various data storage formats like CSV, Excel or Microsoft Access is beyond the scope of this thesis.[1]

The **transform** phase, however, offers a lot potential for automation and optimisation because, as chapter 2 shows, apart from the user-defined business data transformations, most of the steps necessary to synchronise system data with third party data is identical regardless of the imported entities.

3.1.1. API Design

Finding a simple abstraction and a clear framing of the problem that the library is supposed to help solve, is an important part of designing its interface.

[1]It is also questionable, whether abstracting a large number of file format transformations is feasible at all and not best left to the users of the library.

The fact that *BeetleETL* is expected to synchronise only relational database tables within the same database, already simplifies the API design. *BeetleETL* is not designed to abstract all possible interpretations of the ETL process, but optimises one very specific case. This specificity allows the API to incorporate more assumptions on the environment in that the library will be used.

As section 2.6 shows, the process of importing entities is identical for every entity that is imported. This makes it suitable for automation, requiring users of the library only to specify what entities are to be imported.

As a result, the API can be designed in a way that emphasises this. Users only need to specify the transformations they wish to apply to the imported entities. The actual task of synchronisation is abstracted and hidden inside the library's implementation details.

This makes the API more declarative: Instead of implementing the algorithm manually for every imported entity, explicitly defining what tasks have to be executed in what order, users of the library define transformations that express the desired state of the data in the information system and the library handles the calculation and application of changes necessary to realise it.

3.2. Functional Requirements

Since *BeetleETL* is designed to be a specialised library, expected to only fulfil the single purpose of transforming and importing data into relational databases, the list of functional requirements contains only two items.

3.2.1. Synchronising Data

The main functional requirement for *BeetleETL* is synchronising data from one location to another while transforming said data using user-defined business logic transformations.

Success Scenario

Source data is present in a set of tables. Target tables contain either previously imported or no data from the source about to be imported. A set of transformation queries is defined for each entity that is expected to be imported. Invoking *BeetleETL* results in the target database tables reflecting the state of the source data after applying the defined transformations.

Failure Scenarios

BeetleETL is expected never to corrupt system data or break its consistency. The following list states a number of failures that are expected to result in the termination of the import process without altering system data at all. Even if any of these scenarios occurred for only one entity of an import process with multiple entities, none of the other entities' target tables should be altered.

1. A transformation query does not match its source table's schema.
2. A transformed set of data does not match the its target table's schema.
3. Applying the changes to the target tables violates any number of database constraints.

3.2.2. Enable Test-Driven Development

Another functional requirement is to enable users of the library to develop their transformation queries using test-driven development (TDD). Since TDD is the most used process for developing software at *mindmatters*, if the library is to replace parts of the *Epic-Relations* or *Mercury* projects, it must support this style of development.

For the development of transformation queries, it must be possible for users to define and assert expectations on the contents of the stage table in order to verify their correctness.

4. Implementation

This section showcases the implementation of *BeetleETL*. It is an implementation of the algorithm described in chapter 2.

4.1. Disclaimer

Because *BeetleETL* is a library intended to be used by developers, whenever *users* are mentioned this does not refer to users of the application that requires one-way synchronisation, but to users of the library (i.e. developers of the application).

The code listings in this chapter are shortened to contain only the parts that are relevant to explain the given functionality. These listings are annotated with the source code file in which the given part can be found.

4.2. Background

4.2.1. Test Driven Development

BeetleETL was developed following the practice of test driven development (TDD). This is a software development practice that conforms to the following process: The developer writes a test case that defines the desired functionality of a program. This test case functions as a criterion for the correctness of the code. The functionality is then implemented as straightforward as possible, continuously running the test, asserting whether the proposed solution is correct. When the test passes, the code produced is refactored to meet code quality standards agreed upon by the development team.

This cycle of *red*, *green*, *refactor*[1] is then repeated for every new piece of functionality or as more edge cases of a unit of code become apparent, continually improving the code quality of the program as more tests inform its structure and design.

[1]The colours red and green stem from the output of common test runners where failing tests are depicted red and passing tests green.

In addition to guiding implementation, the resulting test suite also aids in regression testing when introducing new functionality or refactoring larger parts of an application.

There are different kinds of tests at different levels of abstraction of the software being developed. High level acceptance tests assert the correctness of a larger part of the program. These tests, sometimes defined as user stories, can be transcribed directly from non-functional requirements. Lower level unit tests help drive the development of smaller components or single classes.

4.2.2. Schema Conventions

BeetleETL requires certain criteria for **target** tables to be met. These criteria are inspired by the conventions introduced by Ruby on Rails[2] and extended for *BeetleETL*.

Users of the library are expected to conform to the following rules for **target** table schemata:

- All primary keys columns are named `id` and of type `INTEGER`.
- All tables contain columns named `created_at`, `updated_at` and, `deleted_at` of type **`TIMESTAMP`**.
- All tables contain columns `external_id` and `external_source` columns. For the purpose of allowing custom `external_id`s, both these columns are expected to be of type `CHARACTER VARYING`.

4.3. Implementing the Algorithm

Because the algorithm described in chapter 2 is already explained in terms of SQL queries, *BeetleETL* can be viewed as a thin wrapper around a set of database queries, orchestrating the order in which they are executed.

As described in section 2.6, the import process of `transform`, `map-relations`, `table-diff`, `assign-ids` and, `load` is the same for every entity that is to be imported.

In *BeetleETL* these steps are therefore implemented using separate step classes. The classes themselves contain the blueprints for the queries necessary to execute the steps while concrete instances of them represent their respective functionality for a specific entity.

[2] *Ruby on Rails* is an open source full-stack web application framework written in Ruby that emphasises software engineering patterns like *convention over configuration* (CoC), *don't repeat yourself* (DRY), the *active record pattern*, and *model–view–controller* (MVC) [rub15]. It is used in both *Epic-Relations* and *Mercury*.

Listing 4.1 shows the `AssignIds` step class as an example. The `run` method contains a generalised version of the query introduced in section 2.3.3, augmented by methods that provide data for a specific entity.

```ruby
class AssignIds < Step

  def run
    database.execute <<-SQL
      UPDATE #{stage_table_name_sql} stage_update
      SET id = COALESCE(target.id, nextval('#{table_name}_id_seq'))
      FROM #{stage_table_name_sql} stage
      LEFT OUTER JOIN #{target_table_name_sql} target
        on (
          stage.external_id = target.external_id
          AND target.external_source = '#{external_source}'
        )
      WHERE stage_update.external_id = stage.external_id
    SQL
  end

end
```

Listing 4.1: `CreateStage` class for generating a stage tables.
`./lib/beetle_etl/steps/create_stage.rb`

The `target_table_name_sql`, `stage_table_name_sql`, `table_name` and, `external_source` methods are provided by the `Step` superclass. In order to change the values of these methods depending on the entity that is imported, this superclass also provides a constructor that takes the name of the imported entity's table as an argument.

For example, calling `AssignIds.new(:publishers)` would instantiate the `AssignIds` class for a *publishers* table and the `run` method of that instance would execute the query for the publishers' **stage** table.

4.3.1. Executing Steps

The most basic approach to executing the steps of the import process is running them in sequence. The order in which they can run is dictated on the one hand by the algorithm and on the other hand by the relationships between entities.

As discussed in section 2.5.1, an entity with dependencies can only complete its import process once all of its dependencies have at least completed the `assign-ids` step in order to be able to translate **source** namespace foreign keys into **target** namespace foreign keys. Before that, the entity's `map-relations` step cannot be executed.

Considering again the example of importing *organisations* and *departments* where departments belong to an organisation. The order in which the steps for importing these two entities could be executed is depicted in listing 4.2.

```ruby
def run_complete_import
  [:organisations, :departments].each do |entity|
    [
      Transform.new(entity),
      MapRelations.new(entity),
      TableDiff.new(entity),
      AssignIds.new(entity),
      Load.new(entity)
    ].each { |step| step.run }
  end
end
```

Listing 4.2: Example for executing steps in a predefined sequence

In this example, all steps for importing organisations are executed first and then all steps for importing departments. Adding new entities to the list only requires them to be added to the array of entities.

This solution, however, has a severe downside to it: The code in listing 4.2 only states that the entities have to be imported in that order. There is no information on the relationships between them.

In addition to *organisations* and *departments*, consider a third *employees* table were being imported. and their order was encoded in the array `[:organisations, :departments, :employees]`, it would not be possible to discern from the context of that file alone whether *employees* depended on *organisations*, *departments* or, both.

4.4. Domain Specific Language - DSL

Preventing misconfiguration and making dependencies explicit are the reasons why *BeetleETL* provides a DSL for defining entities' transformation queries and configuring their relationships (listing 4.3). This DSL helps gather all the necessary pieces of information for importing entities:

1. The name of the table that the entity is supposed to be imported into.
2. The payload columns that are supposed to be imported.

   This is necessary since it is possible to have columns on **target** tables that are not synchronised by *BeetleETL*, but are added by the application layer by way of users of the application. These additional columns are not to be included in the comparison of the `table-diff` step, so *BeetleETL* will only compare the columns specified.
3. The entity's foreign key references to other imported entities.
4. The SQL transformation query necessary to fill the stage tables[3].

```
import :departments do
  columns :name

  references :organisations, on: :organisation_id

  query <<-SQL
    INSERT INTO #{stage_table} (
      name,
      external_id,
      external_publisher_id,
      external_channel_id,
      ...
    )
    SELECT ...
  SQL
end
```

Listing 4.3: Example configuration file for importing `media` in *BeetleETL*

Apart from configuring and actually starting the import process, creating a file that defines the imported entities using this DSL is the only thing users have to do in order to import them.

Stating the foreign key relations of tables also clarifies the relationships between entities. This information is arguably more important when working on transformation queries than the order they are executed in. Instead of explicitly stating the order of execution and relationships

[3]The queries themselves are not abstracted in any way. There are ETL tools that abstract simple transformations, but the queries used in *Epic-Relations* and *Mercury* projects tend to be more complex, requiring multiple joins, nested SELECTs and Common Table Expressions. In these cases SQL is arguably the most practical language to write these queries in, especially if the development team is proficient in the use of SQL.

between entities being implicit, *BeetleETL*'s DSL allows relationships to be defined explicitly, making the order of execution implicit.

With the information provided, the necessary steps and, more importantly, their order can be calculated by *BeetleETL* itself.

The DSL also provides a number of helper methods that can be used to augment the transformation query. As an example, one such helper method (`stage_table`) is shown in listing 4.6. Its necessity will be explained in section 4.6.

4.4.1. Evaluating the DSL

Parsing of the DSL is implemented by evaluating the contents of the transformation file (listing 4.3) in a context in which the `import` method is defined. This is done by the `TransformationLoader` class' `load` method.

The `import` method takes two arguments: one for the name of the **target** table and a block containing the configuration for the imported table. For every invocation of the `import` method in the configuration file, its two arguments are appended to a list of `transformations` (listing 4.4).

```ruby
class TransformationLoader

  def initialize
    @transformations = []
  end

  def load
    File.open(BeetleETL.config.transformation_file, 'r') do |file|
      instance_eval file.read
    end

    @transformations.map do |(table_name, setup)|
      Transformation.new(table_name, setup)
    end
  end

  private

  def import(table_name, &setup)
    @transformations << [table_name, setup]
  end

end
```

Listing 4.4: *BeetleETL*'s transformation loader
`./lib/beetle_etl/dsl/transformation_loader.rb`

The `TransformationLoader`'s `load` method then instantiates an instance of the `Transformation` class for each item in the list of transformations. This `Transformation` instance (listing 4.5), in turn, evaluates the table's configuration block in the context of a `DSL` instance (listing 4.6) that defines the `columns`, `references` and `query` methods used in the example configuration file, depicted in listing 4.3.

```ruby
class Transformation

  attr_reader :table_name

  def initialize(table_name, setup)
    @table_name = table_name
    dsl = DSL.new(table_name)
    @evaluated = dsl.instance_exec(&setup)
  end

  def column_names
    @evaluated.column_names
  end

  def relations
    @evaluated.relations
  end

  def dependencies
    relations.values.to_set
  end

  def query
    @evaluated.query_string
  end

end
```

Listing 4.5: *BeetleETL*'s `Transformation` class
`./lib/beetle_etl/dsl/transformation.rb`

```ruby
class DSL

  attr_reader :column_names, :relations, :query_strings

  def initialize(table_name)
    @table_name = table_name
    @column_names = []
    @relations = {}
    @query_string = nil
  end

  def columns(*column_names)
    @column_names = column_names.map(&:to_sym)
  end

  def references(foreign_table, on: foreign_key)
    @relations[on] = foreign_table
  end

  def query(query)
    @query_string = query
  end

  # query helper methods

  def stage_table
    Naming.stage_table_name_sql(@table_name)
  end

  ...

end
```

Listing 4.6: *BeetleETL*'s `DSL` class
`./lib/beetle_etl/dsl/dsl.rb`

The result of evaluating the configuration file is a list of `Transformation` instances for every imported entity. Every instance contains all the necessary pieces of information for importing its entity: The business logic transformation query, the columns that are to be imported and, the relations to other entities.

4.5. Parallelism

As already explained in section 2.6, the order in which the steps of the import process are executed is is the same for every entity that is imported. The order of steps only becomes more intricate when multiple entities are imported during the same application of the import process that depend on one another.

There are, however, scenarios in which multiple entities are imported that have no dependencies on one another. If this is the case, certain steps can be executed in parallel, possibly increasing the execution time of the whole import process.

4.5.1. Calculating Dependencies

With the addition of declaring relationships between entities explicitly by the use of *BeetleETLs* DSL, instead of applying the steps in a user-defined order, the order in which they have to run can be calculated.

As mentioned in 4.3, all of *BeetleETL*'s step classes inherit from a generic `Step` class that provides common functionality. These step classes are expected to override the `dependencies` and `run` methods. The `dependencies` method returns a set of names of other steps that a particular step requires to execute before it can be executed itself. The `run` method implements the execution of the actual queries for that step.

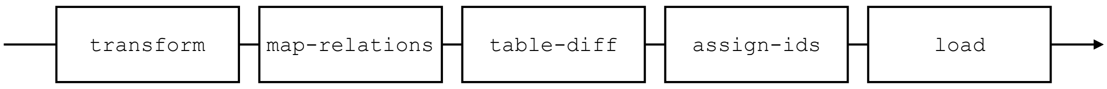

Figure 4.1.: Complete process of importing a single entity

Returning to the complete import process depicted in figure 4.1, defining the dependencies for some steps is trivial. The `assign-ids` step, for instance, can only run if the `table-diff` step for the same entity has been completed. Listing 4.7 shows the way this is implemented in *BeetleETL*.

To enable this, the `Step` class provides the `step_name` class method and the `name` instance method that return a unique name of the step for a specific entity imported. For example, calling `TableDiff.step_name(:publishers)` returns the string `"publishers: TableDiff"`, uniquely identifying the `table-diff` step for the `publishers` table.

```ruby
class AssignIds < Step

  def dependencies
    [TableDiff.step_name(table_name)].to_set
  end

  def run
    database.execute <<-SQL
      UPDATE #{stage_table_name_sql} stage_update
      SET id = COALESCE(target.id, nextval('#{table_name}_id_seq'))
      FROM ...
    SQL
  end

end
```

Listing 4.7: *BeetleETL*'s `AssignIds` class
`./lib/beetle_etl/steps/assign_ids.rb`

For the `map-relations` step the calculating list of dependencies is more complex. Apart from the `transform` step, preceding `map-relations` for the same entity, it also includes `assign-ids` steps for other imported entities, as discussed in section 2.5.1. Listing 4.9 shows the implementation for calculating dependencies for the `MapRelations` step.

```ruby
class MapRelations < Step

  def initialize(table_name, relations)
    super(table_name)
    @relations = relations
  end

  def dependencies
    result = Set.new([Transform.step_name(table_name)])
    result.merge @relations.values.map { |d| AssignIds.step_name(d) }
  end

  def run
    @relations.map do |foreign_key_column, foreign_table_name|
      database.execute <<-SQL
        UPDATE #{stage_table_name_sql} current_table
        SET #{foreign_key_column} = foreign_table.id
        FROM ...
      SQL
    end
  end

end
```

Listing 4.8: *BeetleETL*'s `MapRelations` class
`./lib/beetle_etl/steps/map_relations.rb`

For all other steps dependencies were dictated by the import algorithm and defined by *BeetleETL* itself. Relationships between entities, however, are defined by the user and therefore have to be validated.

When defining relationships between entities manually, there is a risk of users introducing cyclic dependencies. Therefore *BeetleETL* will check whether the all steps' dependencies can be met before execution starts and throw an `UnsatisfiableDependenciesError` if not.

4.5.2. Parallel Execution

BeetleETL implements parallelism by asynchronously scheduling database queries. Parallel execution of those queries itself is handled by the DBMS. Every query is handed off to the DBMS in its own thread and synchronisation between these threads is handled by a queue.

With all steps' dependencies in place, *BeetleETL* starts executing a list of steps using an instance of the `AsyncStepRunner` class. This instance implements the following algorithm:

1. Find all steps without dependencies or whose dependent steps are already completed.
2. Execute each of these steps in a separate thread.
3. When a step completes, push its name onto the queue.
4. Pop the step's name off of the queue and mark it as complete.
5. If there are still steps left to run, start again with item 1.

```ruby
class AsyncStepRunner

  def initialize(steps)
    @dependency_resolver = DependencyResolver.new(steps)
    @steps = steps

    @queue = Queue.new
    @completed = Set.new
    @running = Set.new
  end

  def run
    until all_steps_complete?
      runnables.each do |step|
        run_step_async(step)
        mark_step_running(step.name)
      end

      step_name = @queue.pop
      mark_step_completed(step_name)
    end
  end

  private

  attr_reader :running, :completed

  def run_step_async(step)
    Thread.new do
      step.run
      @queue.push step.name
    end
  end

  ...

end
```

Listing 4.9: Implementation of *BeetleETL*'s `AsyncStepRunner`
`./lib/beetle_etl/step_runner/async_step_runner.rb`

4.6. Generating Stage Tables

Chapter 2 established that the algorithm used in *BeetleETL* for synchronising data requires the use **stage** tables.

While providing the **source** and **target** tables, are within the user's responsibility, the concept of **stage** tables is an implementation detail of *BeetleETL*, an therefore must not be exposed to the user.

Because *BeetleETL*'s DSL requires users to state the table name, the names of the columns that are to be imported and, the foreign key relations for any entity, **stage** tables can be generated automatically, completely relieving the user of this responsibility.

By introspecting the **target** table, it is possible to procure the data type of every column specified, resulting in the following rules for **stage** tables:

1. It must contain an `id` column of type `INTEGER`.
2. It must contain an `external_id` column of type `CHARACTER VARYING`.
3. It must contain a `transition` column of type `CHARACTER VARYING`.
4. It must contain the specified payload columns of the same type as their corresponding **target** table columns.
5. For every foreign key relation it must contain a pair of `foreign_id` and `external_foreign_id` of types `INTEGER` and `CHARACTER VARYING`, respectively.

Listing 4.10 shows parts of *BeetleETL*'s `CreateStage` class for generating **stage** tables and demonstrates the application of the aforementioned set of rules.

For performance reasons it was chosen to define the **stage** tables as UNLOGGED. Since the data inside the **stage** tables is not crucial to the user's application, it does not have to be logged by the DBMS for recovery. In case of a database crash the import process cannot be continued and has to be started anew.

```ruby
class CreateStage < Step

  def initialize(table_name, relations, column_names)
    super(table_name)
    @relations = relations
    @column_names = column_names
  end

  def run
    database.execute <<-SQL
      CREATE UNLOGGED TABLE IF NOT EXISTS #{stage_table_name_sql} (
        id integer,
        external_id character varying(255),
        transition character varying(255),

        #{column_definitions}
      )
    SQL
  end

  private

  def column_definitions
    [
      payload_column_definitions,
      relation_column_definitions
    ].compact.join(", ")
  end

  ...

end
```

Listing 4.10: `CreateStage` class for generating a stage tables.
`./lib/beetle_etl/steps/create_stage.rb`

4.6.1. Generating Table Names

The generation of **stage** tables, however, introduces a new problem. Names for the **stage** tables have to be generated as well and that introduces the risk of name collisions. In order to remedy this risk the naming scheme depicted in listing 4.11 is applied. It generates a name unique to the `external_source` being imported for a given **target** table name. While collisions are still possible, with the inclusion of the **source**'s name and the digest they areis i very unlikely.

```ruby
module Naming
  def stage_table_name(table_name)
    digest = Digest::MD5.hexdigest(table_name)
    "#{external_source}-#{table_name}-#{digest}"[0, 63]
  end
end
```

Listing 4.11: Method for generating names for stage tables.
`./lib/beetle_etl/naming.rb`

The method shown in listing 4.11 is also used in the implementation of the `stage_table` helper method referred to in listing 4.3 of section 4.4. As listing 4.6 shows, it is available for users of the library to use in their transformation queries. This allows users to insert data into the **stage** table without knowing the name it has been assigned. The name is evaluated at runtime and substituted with the name of the actual **stage** table.

4.6.2. Dropping Stage Tables

BeetleETL will drop the generated **stage** tables after the import run. It even ensures that they are dropped if the process throws an exception.

4.7. Preventing Data Corruption

BeetleETL is designed to never corrupt user data, even if the application of the import process fails for any reason.

While all steps, including `CreateStage` and `DropStage`, inherit from the same `Step` class and define the `run` and `dependencies` methods, they are not all executed in the same instance of the `AsyncTaskRunner` class. In fact, only the steps necessary to transform the data and calculate the the necessary changes are run asynchronously within the same instance.

The `CreateStage` and `DropStage`, however, are executed separately and in sequence. Since creating and dropping database tables are very fast operations on a database, there is not much performance improvement to be gained by running these steps in parallel.

In case of `DropStage` there is a downside to running them along with the other steps: In case of an exception, they must be ensured to run in order not to pollute the user's database[4].

[4]The first approach to solving the problem of polluting users' databases with **stage** tables was the use of `temporary` tables. Unfortunately, however, this is not supported by the library that *BeetleETL* uses to dispatch SQL queries to the database. The `temporary` tables would not be visible to subsequent queries.

The `Load` steps executed separately, as well. Because actually applying changes to the **target** tables might fail, they are run inside a database transaction: Either all `Load` steps will be executed or none at all.

```ruby
class Import

  def run
    transformations.each do |t|
      CreateStage.new(t.table_name, t.relations, t.column_names).run
    end

    AsyncStepRunner.new(data_steps).run
    BeetleETL.database.transaction do
      AsyncStepRunner.new(load_steps).run
    end
  ensure
    transformations.each do |t|
      DropStage.new(t.table_name).run
    end
  end

  private

  def data_steps
    transformations.flat_map do |t|
      [
        Transform.new(t.table_name, t.dependencies, t.query),
        MapRelations.new(t.table_name, t.relations),
        TableDiff.new(t.table_name),
        AssignIds.new(t.table_name),
      ]
    end
  end

  ...

  def transformations
    @transformations ||= TransformationLoader.new.load
  end

end
```

Listing 4.12: *BeetleETL*'s `Import` class
`./lib/beetle_etl/import.rb`

4.8. Running Imports

In order to actually run *BeetleETL*, it has to be configured and executed. Listing 4.13 shows an example configuration.

```ruby
BeetleETL.configure do |config|
  config.transformation_file = '../example_transform.rb'
  config.database_config = database_config
  config.external_source = 'my_example_source'
  config.logger = Logger.new(STDOUT)
end

BeetleETL.import
```

Listing 4.13: Configuring and running *BeetleETL*

4.9. Testing

BeetleETL is expected to be usable in a development environment that values automated testing[5]. Therefore it must be possible for users to test their transformation queries.

Generating **stage** tables, however, introduces a number of complications: **Stage** tables only exist during the execution of the import process and cannot be accessed outside of it. Their names are unknown to the user, making it even more complicated to assert expectations on them. Also, the transformation query itself is defined using *BeetleETL*'s DSL and is not directly accessible to the user.

In order to enable automated testing, *BeetleETL* exposes a `Testing` module to its users. This module contains three helper methods that aid users in running tests for their transformation queries. Listing 4.14 shows an example for testing the transformation query for a *channels* entity.

[5]This is not limited to TDD, but can be applied in any kind of software development process that features automated testing.

```
describe "importing channels" do

  include BeetleETL::Testing

  it "inserts a channel with a new id" do
    with_stage_tables_for :channels do
      insert_into("source.gattungen").values(
        [ :id , :name        , :kuerzel ] ,
        [ 1   , "Gattung 1" , "G1"      ] ,
      )

      run_transformation :channels

      expect(stage_table_name(:channels)).to have_values(
        [ :id , :name        , :abbreviation , :external_id ] ,
        [ 1   , "Gattung 1" , "G1"           , "1"          ] ,
      )
    end
  end
end
```

Listing 4.14: Example for testing a transformation query using the `rspec` testing library and*BeetleETL*'s `Testing` module

The `with_stage_tables_for` method takes a list of entity names as arguments as well as a block and ensures the existence of these entities' stage tables during the execution of the block. After the block completes they are removed, restoring the initial state of the test environment.

Inside the block, the `run_transformations` and `stage_table_name` can then be used to run the transformation for a given table and returning the name of the given **stage** table, respectively.

5. Conclusion

BeetleETL provides a solution to the problem of one-way synchronisation. Given two sets relational database tables, it can duplicate the state of one to the other while applying business transformations.

The choice of *BeetleETL*'s particular interpretation of the ETL process narrows the scope of the library, allowing it to hide most of the intricacies of the synchronisation algorithm. Since the algorithm that *BeetleETL* uses for merging data is universal for all data that can be imported and can be automated, complete understanding of it is no longer required to make decisions about what data needs to be synchronised. If, for example, a new entity needs to be imported, this can be defined by simply adding a new `import` directive to the transformation file, instead of figuring out which queries need to be executed in what order.

The DSL underlines the simplicity of the metal model that *BeetleETL* establishes. It allows reasoning about import process on a higher level of abstraction: Users do not decide which steps of the import process need to be run, but instead decide what entities need to be imported. This allows a more declarative approach to one-way synchronisation where users define the desired state of a system and the library handles actually applying changes to the system so that it resembles this desired state.

The parallel execution model that *BeetleETL* implements also improves execution time when comparing it to a strictly sequential approach. At any given point during the application of the import process all steps that could possibly run in parallel are dispatched asynchronously to the DBMS. A comparison of execution times between *BeetleETL* and *Mercury*'s importer can be found in appendix A.

BeetleETL has been released as an open source library on `https://rubygems.org/` and can be installed using the `gem` or `bundler` package managers for Ruby. The source code can be found at `https://github.com/mawiald/beetle_etl`.

5.1. Areas of Improvement

5.1.1. Analytics

Even though *BeetleETL* provides logging for individual steps and aggregates the durations of all steps after each import run, there is potential for improving the library.

Durations of individual steps may be of interest in and of itself, but there is more value to it when comparing them to the durations of previous runs. Especially when using complex transformation queries, comparing runtimes of multiple applications of the import process can offer insights and suggestions as to what parts should be optimised.

Another helpful addition to the library would be statistics for changes in the imported entities. Seeing how many entities were created, updated and, deleted might be of interest to users of the application.

5.1.2. Validation Loop

Another possible improvement of *BeetleETL* would be the addition of a *validation loop*, a concept found in the *Epic-Relations* importer.

In addition to transforming and importing data, it features the concept of a *validation loop*. Because a large number of the entities imported are related to each other and validity of certain records depends on the existence of certain others records, it is not feasible to filter these entities in the transformation queries as is done in *BeetleETL*.

Instead, a *validation loop* checks a number of conditions for every record before they are loaded into the **target** tables and if these conditions are met, the record is deleted. If the deletion of this particular record then causes the conditions for deleting a referencing record to occur, that record is deleted in a subsequent run of the *validation loop*. This cycle is repeated until no more records are deleted, which states that the remaining records are valid.

5.1.3. SQL parser

One further area of improvement for *BeetleETL* lies in the DSL. Currently, users are required to supply the list of columns they wish to import for a given entity. Hence, the necessity to understand the concept of `external_id`s in order to use *BeetleETL* for importing complex graphs of related entities, complicates the mental model for synchronising data.

A possible solution to these problems is the use of an SQL parser. Using an SQL parser to analyse the user supplied transformation queries results in abstract syntax trees from which the imported columns could be extracted. This would make stating them obsolete, simplifying the library's interface.

Parsing the query could also be used to hide the existence of **stage** tables from the user. Users could simply formulate their queries as if they operated on the **target** tables and *BeetleETL* could replace the table in the AST.

The parsed query could also be transformed in a way that users would not have to insert data the `external_*_id` columns. It would instead be possible to define transformation queries solely in the source's primary key namespace using regular foreign key columns, simplifying the the API even further and leaving it to the library to generate the necessary queries.

Appendices

A. Performance Comparison

In addition to importing and merging data, the *Mercury* importer also contains extracting data from a Microsoft Access database file and dumping the data into the *source* tables. Therefore, all tests results include the duration of this extraction which varies between 3 and 4 minutes, depending on the amount of data being imported.

In order to compare the execution time of both *BeetleETL* and the *Mercury* importer, *BeetleETL* was integrated into the mercury importer, replacing the functionality.

A.1. Test Setup

The tests were run on an *Early 2011 MacBook Pro* with a 2GHz Core i7 processor and 8GB of DDR3 ram, running Mac OS X 10.10.2 with the internet connection disabled. Measurements for the duration of each import were taken using the unix `time` tool.

In order to run the test in a realistic environment, actual import data from the *Mercury* project was used. Each test run consists of executing *Mercury*'s full ETL process.

Mercury imports 14 entities using transformations of varying complexity, ranging from simple queries that merely move data to queries that require multiple joins and common table expressions.

A.2. Results

The tests were run importing three different Microsoft Access database files: 1.33GB, 1.33GB and, 1.42GB.

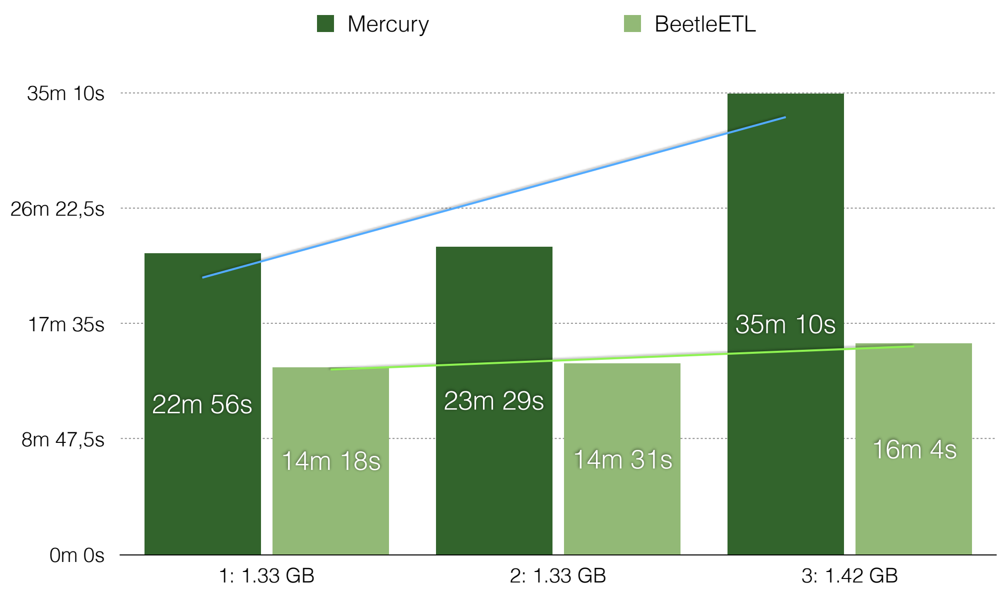

Figure A.1.: Importing three different sets of data

Bibliography

[rub15] Ruby on rails. http://rubyonrails.org/, April 2015.

[TvS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms, Second Edition.* Prentice Hall, 2006.

[wik14a] Don’t repeat yourself. http://en.wikipedia.org/wiki/Don%27t_repeat_yourself, October 2014.

[wik14b] Extract, transform, load. http://en.wikipedia.org/wiki/Extract,_transform,_load, October 2014.

[wik15] Master data. http://en.wikipedia.org/wiki/Master_data, April 2015.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 30. April 2015 Luciano Maiwald