

EvolveDB - Documentation for End-User

Author: Torben Eckwert, M.Sc.
E-Mail: torben.eckwert@zdh.thm.de

Subject Area: Computer Science

Supervisors: Prof. Dr. rer. nat. Michael Guckert
Prof. Dr. ing. Gabriele Taentzer

December 5, 2024 , Wetzlar



Philipps



Universität
Marburg



FORSCHUNGSCAMPUS MITTELHESSEN

Contents

1	Introduction	1
2	Prerequisites and Installation	1
2.1	Installation via Eclipse Installer (Recommended)	1
2.1.1	Getting Started with the Oomph Installer	1
2.2	Manual installation	3
3	EvolveDB	4
3.1	Reverse Engineering	5
3.2	Restructuring	9
3.2.1	Change Column Type	10
3.2.2	Add New Columns	11
3.2.3	Change Default Values	12
3.2.4	Reduce Column Size	14
3.2.5	Change the Multiplicity of an Association	14
3.3	Model Comparison	16
3.3.1	SiLift	16
3.3.2	Model Compoarison with EvolveDB	18
3.4	Forward Engineering	23
3.4.1	Migration Model	23
3.4.2	Breaking & Resolvable	26
3.4.3	CHANGE_1N_INTO_NM	26
3.4.4	Migration Strategies:	27
3.4.5	Not Automatically Resolvable	28
3.5	Generate Migration Scripts	29
4	Migration Scripts	32
4.1	History Table	36

1 Introduction

EvolveDB is an Eclipse-based framework designed for managing schema evolution in MySQL databases. Users can define evolution steps by freely editing a database model extracted through reverse engineering. EvolveDB automatically analyzes the differences between the current schema and the evolved model, generating a corresponding data migration script. This user manual includes installation instructions and introductory tutorials on how to use EvolveDB as an end user.

2 Prerequisites and Installation

EvolveDB is a plug-in for recent versions of the [Eclipse Modeling Tools](#) (last tested with version 2022-06). It is an open-source project licensed under the Apache License 2.0. The complete source code is freely accessible on our [GitHub repository](#).

2.1 Installation via Eclipse Installer (Recommended)

Setting up a development environment can be a time consuming task. Such a setup can include:

- the installation of certain plug-ins
- the configuration of certain preferences
- cloning certain Git repositories
- importing projects from these Git repositories

Manually performing these steps is not only tedious but also prone to errors. Fortunately, the Eclipse Installer simplifies this process. As a standalone application, it is designed to install Eclipse products and keep them up-to-date. The underlying technology, Oomph, extends these capabilities to automate the provisioning of a pre-configured and customized Eclipse IDE. Oomph allows developers to automate their setup process using a setup profile. A setup profile is a configuration file that contains the tasks required to configure your Eclipse IDE precisely as needed. To streamline your workflow, our GitHub repository includes an Oomph setup file, located in the documentation folder.

2.1.1 Getting Started with the Oomph Installer

Download the Eclipse Installer from the official [Eclipse website](#). Select the installer package appropriate for your platform and extract it. To use a setup file, launch the installer and switch to Advanced Mode. From there, you can load the provided setup file to automate the configuration of your Eclipse environment, saving time and reducing the likelihood of errors.

2 Prerequisites and Installation

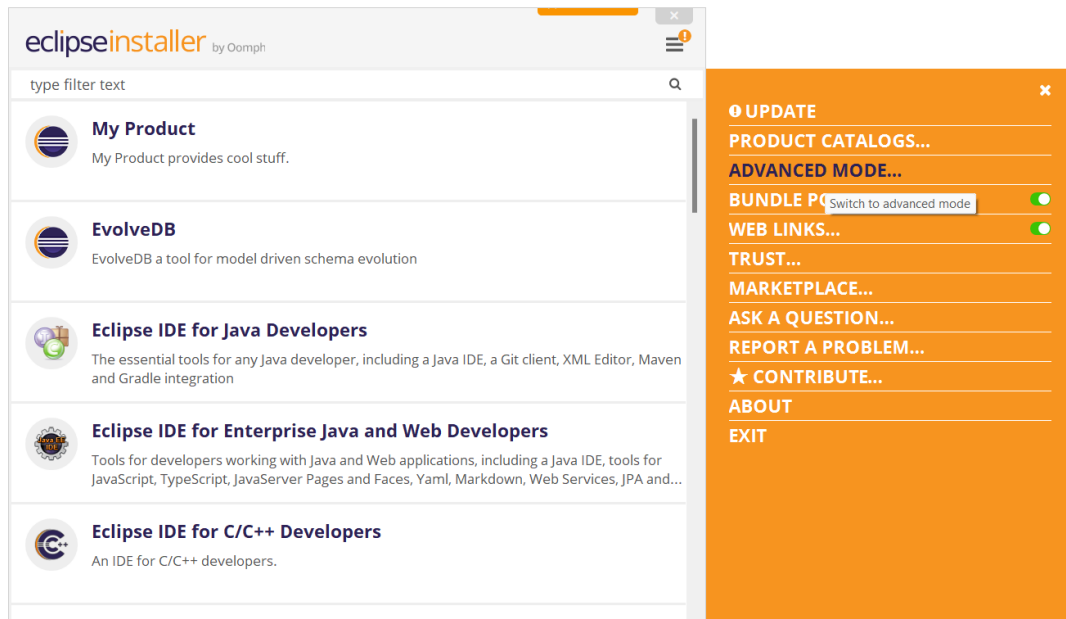


Figure 1: Eclipse installer: Advanced Mode

Out of the box the installer provides certain default categories containing different profiles, e.g., for Eclipse Projects. Additional user setups can be added by using the green plus button besides the filter text. After adding the custom EvolveDB setup file, the Eclipse installer creates an Eclipse environment with EvolveDB and all necessary plugins installed.

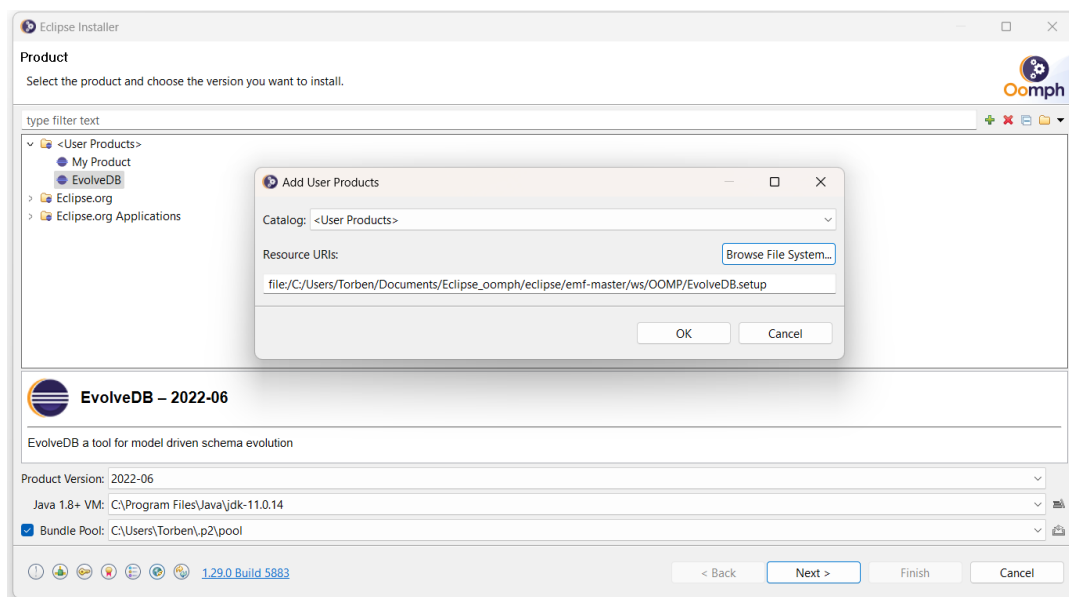


Figure 2: Eclipse installer: Additional user setup

The Eclipse Installer comes with several default categories, each containing various profiles tailored for Eclipse projects. These profiles provide a convenient starting point for setting up

development environments. To extend the available options, additional user-defined setup files can be added by clicking the green + button next to the filter text field. Once you add the custom EvolveDB setup file from our repository, the Eclipse Installer automatically creates a fully configured Eclipse environment. This setup includes EvolveDB and all required plug-ins, ensuring you are ready to start working immediately.

2.2 Manual installation

Important: EvolveDB currently requires Java SE 11 for proper operation. Using newer or older Java versions may result in unexpected behavior. To configure the required Java version, update the -vm option in the eclipse.ini file.

Before installing EvolveDB, we need to install some additional Eclipse plugins. The following list contains the mandatory plugins.

- [Sirius](#) is an Eclipse project which allows you to easily create your own graphical modeling workbench by leveraging the Eclipse Modeling technologies, including EMF and GMF.
- [Eclipse OCL](#) is available via the eclipse marketplace.

If all requirements are fulfilled, EvolveDB can be installed via the menu item **Help** ⇒ **Install New Software...** (fig. 3).

We used the [MySQL Server](#) version 8.0.28 for this tutorial. When using older versions, compatibility problems may occur.

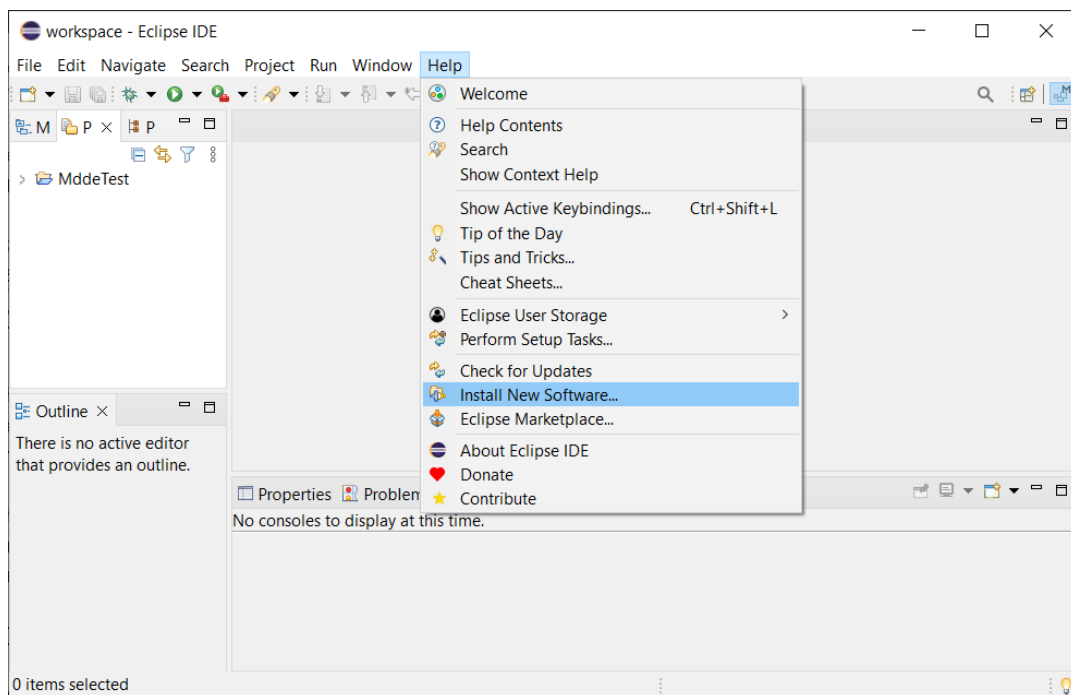


Figure 3: Eclipse: Install New Software...

The repository contains multiple categories (fig. 4). For the following tutorial, all elements must be installed.

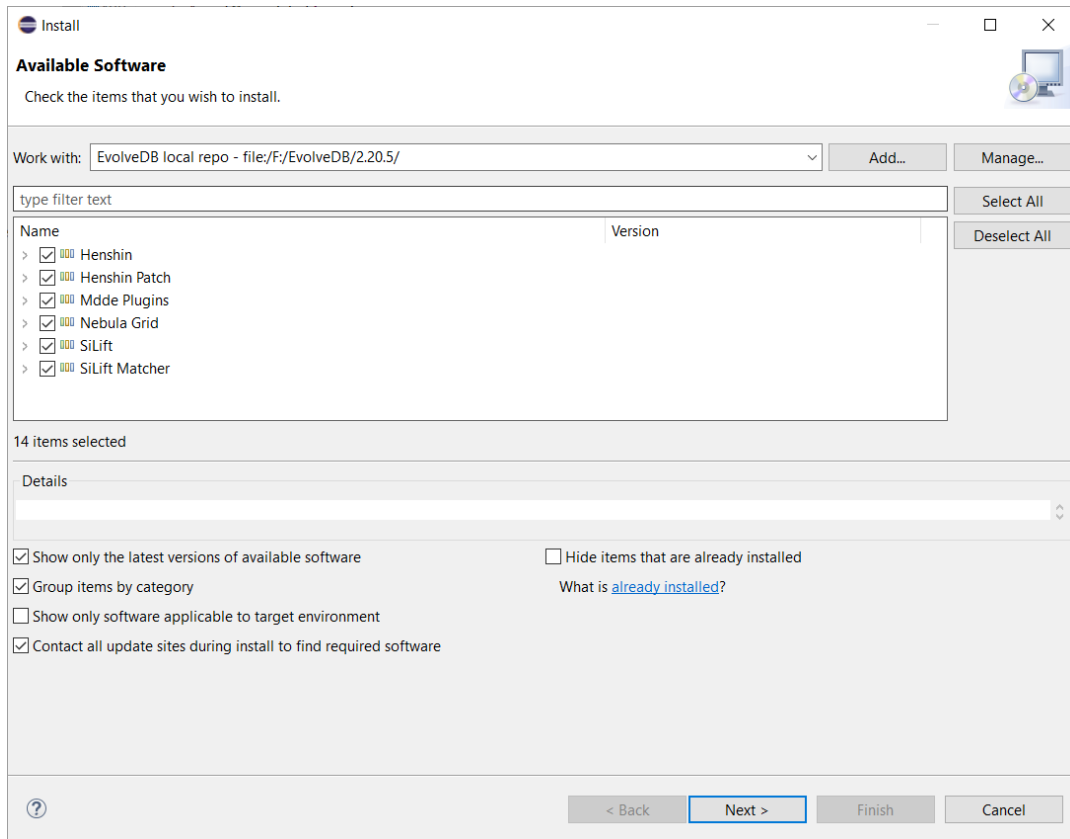


Figure 4: Eclipse: EvolveDB local repository

3 EvolveDB

EvolveDB employs a model-driven reengineering process for schema evolution, which consists of three key phases: reverse engineering, restructuring, and forward engineering.

- **Reverse Engineering:** The process begins with the creation of a model representing the existing database schema. This model is derived through reverse engineering, providing a structured view of the database.
- **Restructuring:** In this phase, the extracted model is edited to define the desired changes, resulting in a new version of the schema model. EvolveDB then analyzes the differences between the original schema and the evolved model.
- **Forward Engineering:** Using the calculated delta between the two model versions, EvolveDB generates SQL migration scripts. These scripts facilitate the migration of both the schema and the associated data, ensuring the database remains consistent.

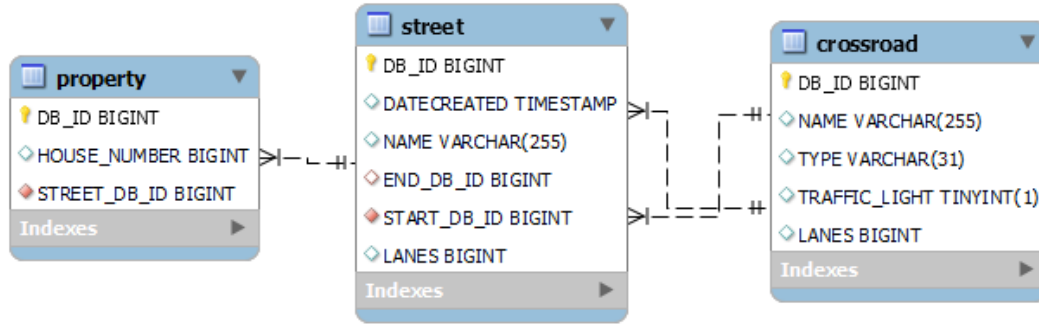


Figure 5: MySQL database schema

The use of EvolveDB for schema evolution is illustrated through a running example. The starting point is an existing MySQL database, as shown in Figure 5. The schema models traffic routes, represented by the table *street*. Each street has a starting point, which can be an intersection or a traffic circle, represented by the table *crossroad*. Multiple roads can originate from or terminate at a single node. Additionally, properties located along traffic routes are captured in the table *property*. Throughout this tutorial, we will apply a series of changes to the database schema and its associated data:

- In the table *street*, the data type of the column *DATECREATED* is changed from *TIMESTAMP* to *DATETIME*.
- Two new columns, *LONGITUDE* : *Integer* and *LATITUDE* : *Integer*, are added to the table *crossroad*.
- The size of the *NAME* column in the table *street* is reduced from 255 to 40.
- The single-valued association between *property* and *street* is transformed into a multi-valued association, requiring the creation of a new cross-reference table.
- A new default value of 1 is set for the column *lanes* in the table *crossroad*.

3.1 Reverse Engineering

In the first step, we extract a database model through reverse engineering. To do this, open the **Eclipse Project Explorer** and create a new project. Once the project is created, select it, then right-click to open the context menu. From the context menu, choose **EvolveDB** ⇒ **Create MDDE Model...** (Figure 6).

3 EvolveDB

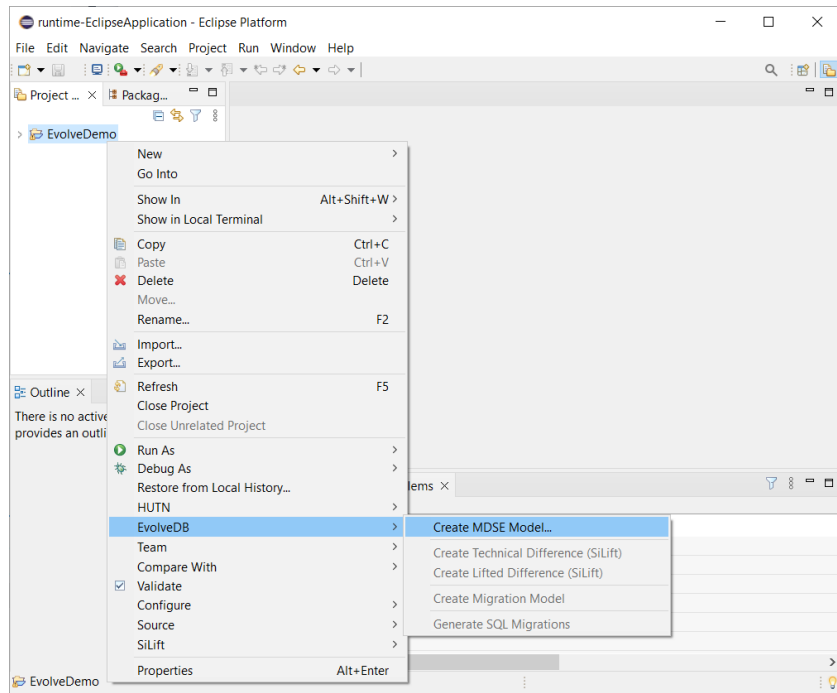


Figure 6: Start reverse engineering

A wizard dialog opens, consisting of several pages. On the first page, you need to select the data source. (fig. 7)

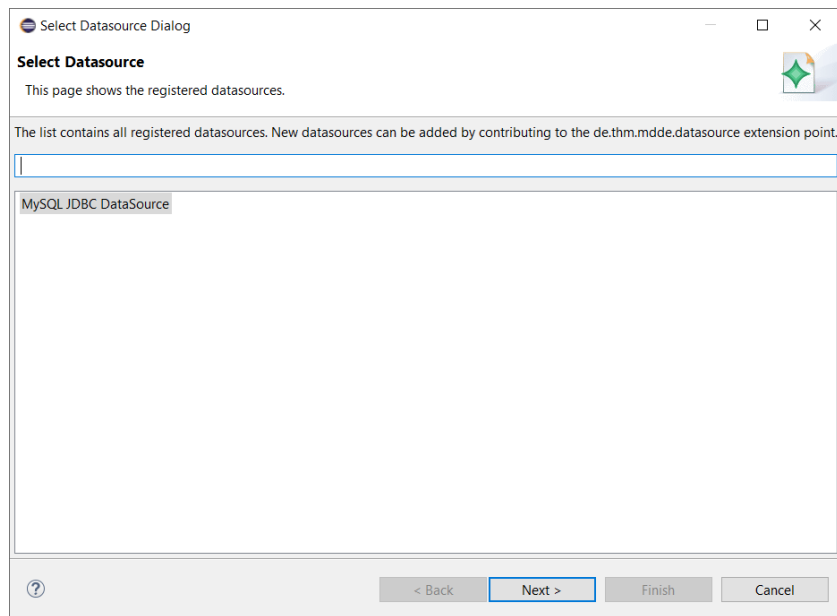


Figure 7: Select data source

EvolveDB does not include database drivers by default. When using EvolveDB for the first

time, you must download the required JDBC driver. Driver files can either be downloaded automatically through the tool or obtained manually. If you choose to download the driver manually, place the files in the designated driver directory. By default, this directory is located in the Eclipse installation folder. The driver location can be customized through the Eclipse preferences. If EvolveDB cannot locate a suitable driver, it will prompt you with a download driver dialog to assist in obtaining the necessary files. (fig. 8)

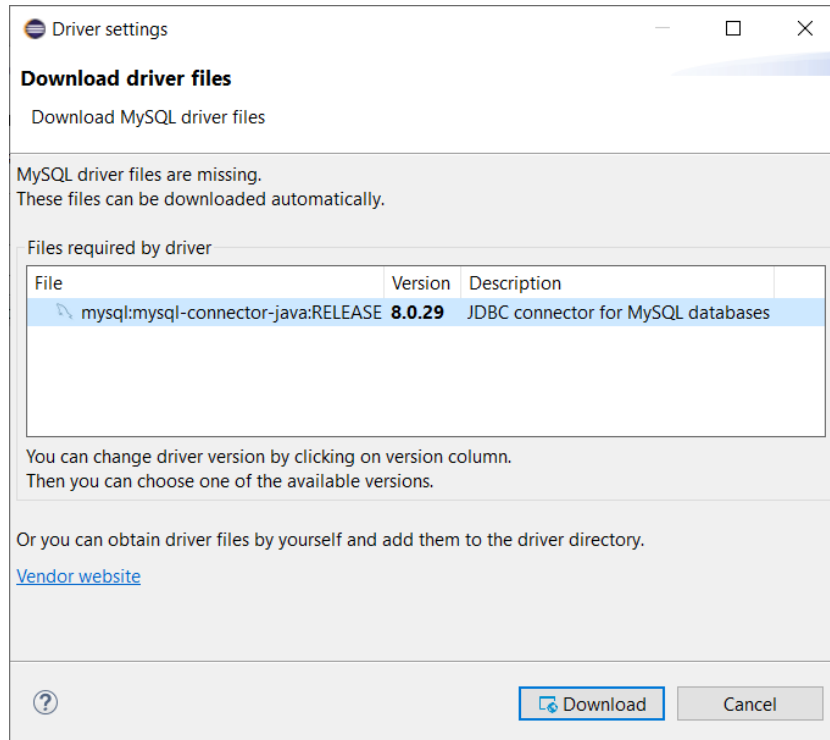
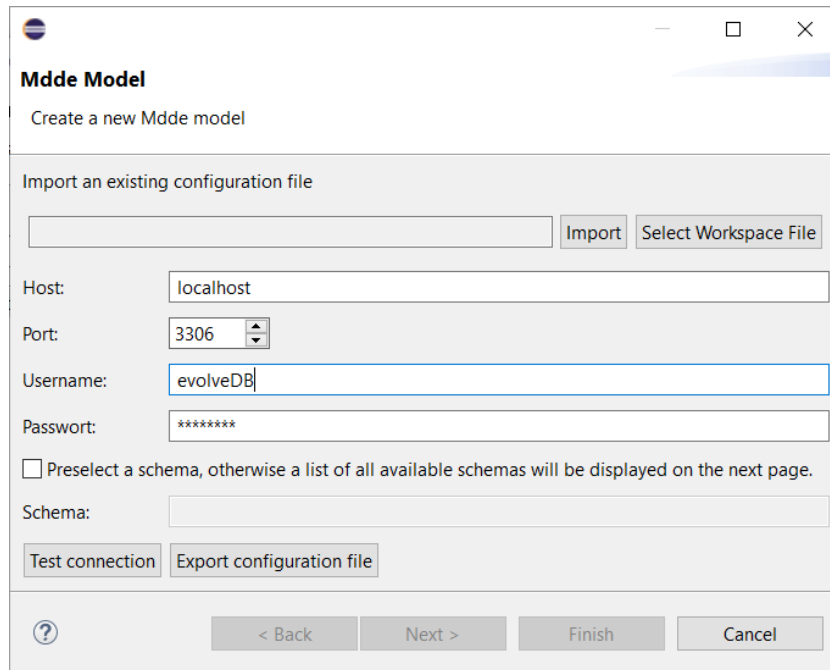


Figure 8: Download JDBC driver

After loading the driver class, you need to provide the connection information (Figure 9). Optionally, you can save the connection details (excluding the password) as an XML configuration file. This configuration file can be imported when creating a new model, eliminating the need to re-enter the connection information (Figure 11). Once the connection information is entered, you must test the connection. If the test is successful, you can proceed to the next step. The following page displays a list of all available database schemas, unless a schema was preselected on the first page (Figure 10). On the final page of the dialog, specify the name and storage location for the new model. After completing these steps, click Finish to close the dialog. The newly created model will automatically open in the editor. Additionally, a new folder named `genModel` is created at the specified storage location, containing a copy of the model. This second model is crucial for the comparison process after the restructuring phase and must not be altered.



Mdde Model
Create a new Mdde model

Import an existing configuration file

Host:

Port:

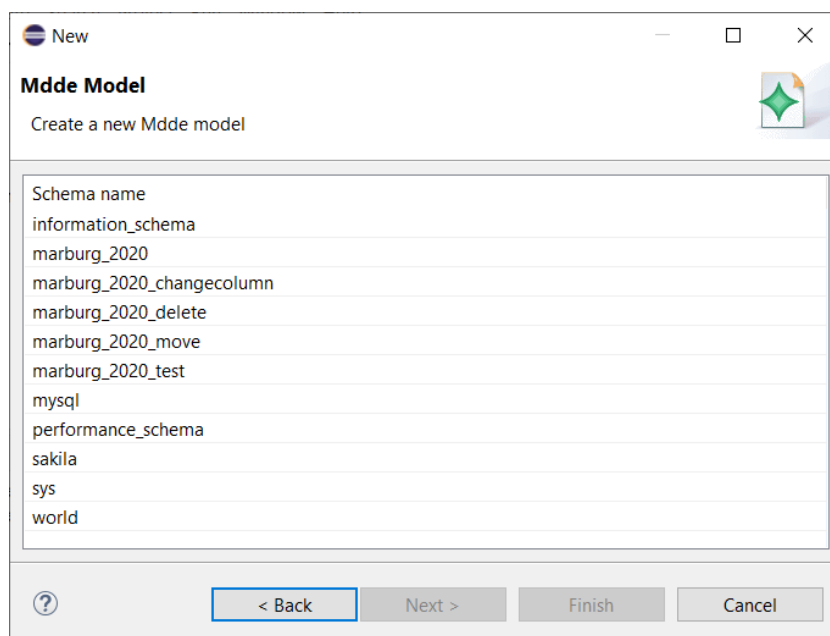
Username:

Password:

☐ Preselect a schema, otherwise a list of all available schemas will be displayed on the next page.

Schema:

Figure 9: Insert connection data



Mdde Model
Create a new Mdde model

Schema name

- information_schema
- marburg_2020
- marburg_2020_changecolumn
- marburg_2020_delete
- marburg_2020_move
- marburg_2020_test
- mysql
- performance_schema
- sakila
- sys
- world

Figure 10: Select database schema

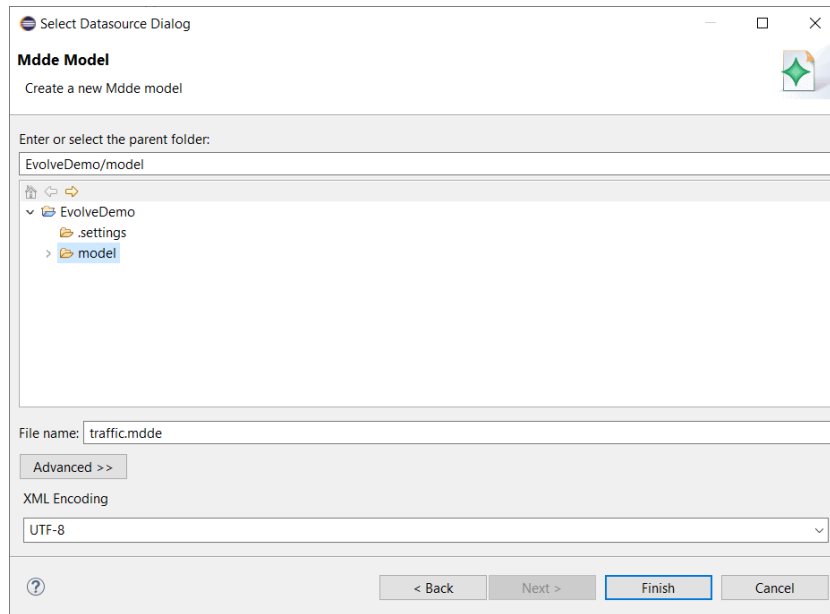


Figure 11: Choose file location

3.2 Restructuring

In the previous step, we created a database model through reverse engineering. The newly created model was automatically opened in the Eclipse editor (Figure 12).

The editor is divided into two sections:

- **Left Pane:** A tree-based editor for navigating the model.
- **Right Pane:** A detailed view displaying the properties of the selected element.

Each element in the model corresponds to a specific element in the database. Expanding an element reveals its child elements. With the model loaded in the editor, you can now begin editing it.

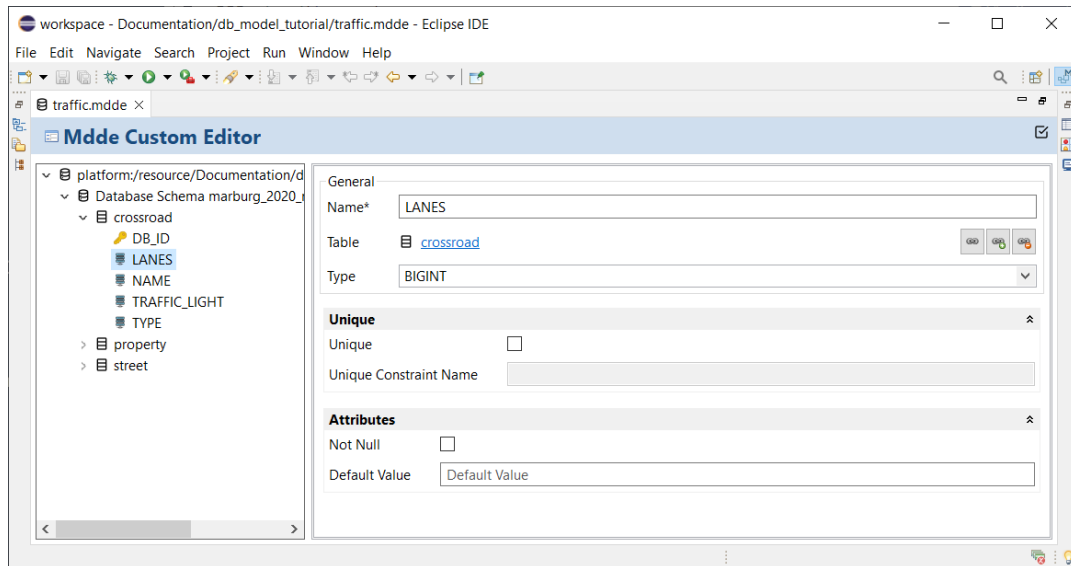


Figure 12: MDDE model editor

3.2.1 Change Column Type

To modify the column type, follow these steps:

1. Select the column *DATECREATED* in the table *street* within the tree-based editor.
2. The attributes of the selected column will appear in the properties view on the right side of the editor.
3. Locate the *Type* attribute and change its value from *TIMESTAMP* to *DATETIME*.

3.2.2 Add New Columns

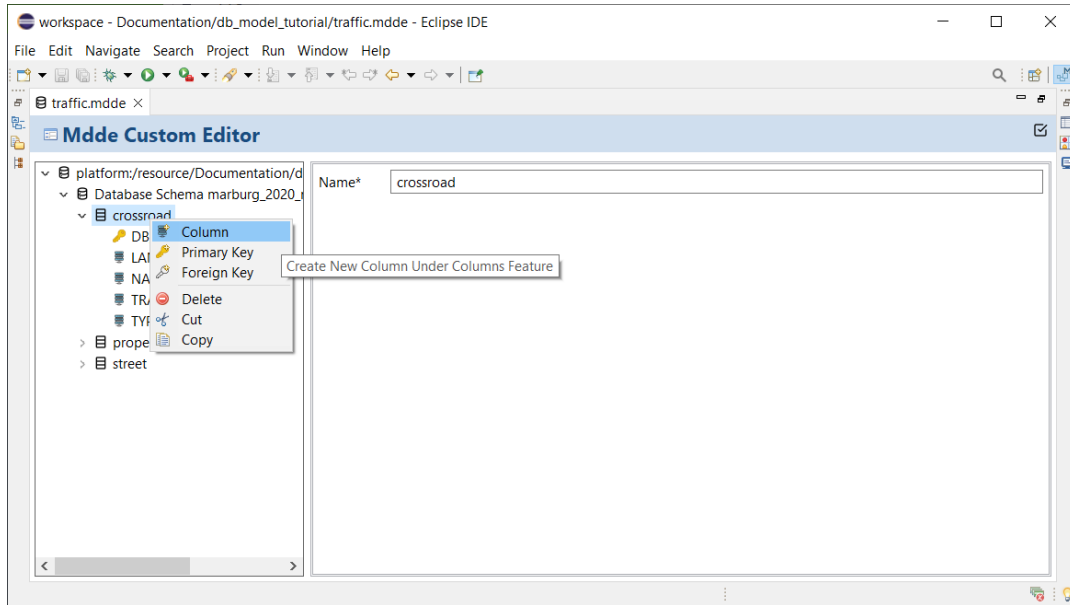


Figure 13: Context menu

In this step, we will add two new columns for longitude and latitude to the *crossroad* table:

1. Select the *crossroad* table in the tree-based editor and open the context menu with a right-click (Figure 13).
2. Choose **Create Column** from the menu.
3. The new column will open in the editor (Figure 14). Fields marked in red are mandatory.

For the first column:

- Name the attribute *LATITUDE*.
- Set the data type to *Integer*.
- Note: Columns of type *Integer* do not have a *Size* attribute, so this field is hidden.

Leave the other attributes unchanged (Figure 15).

Repeat the process to create the *LONGITUDE* column, following the same steps.

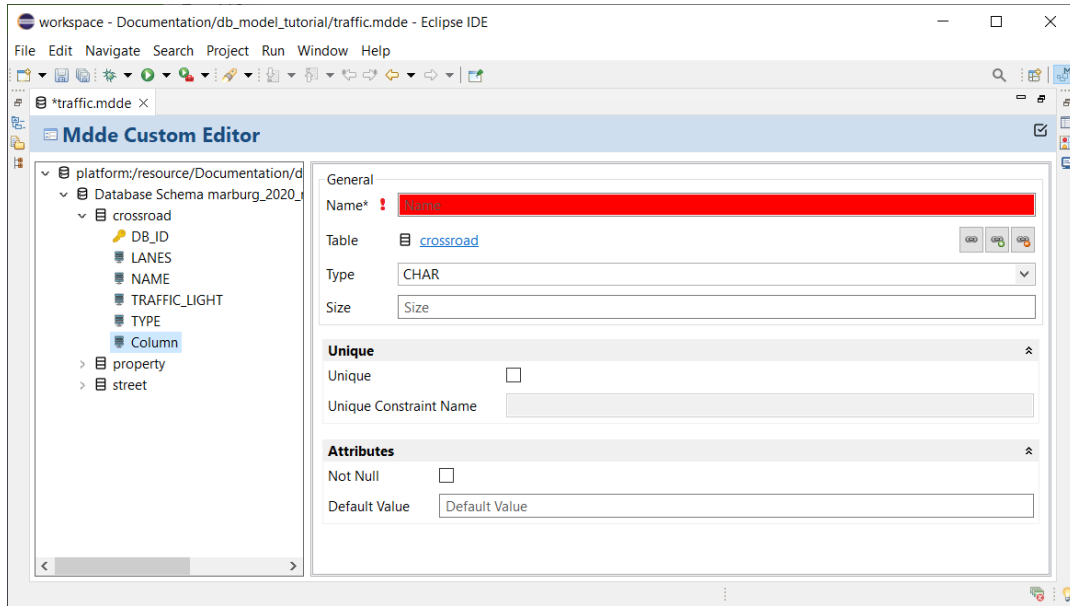
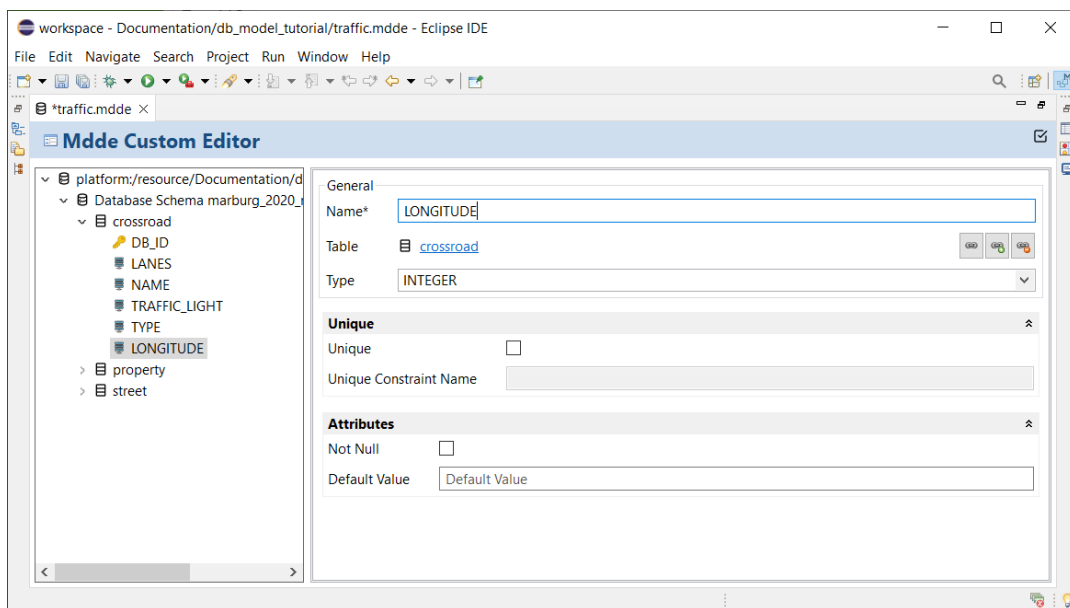


Figure 14: Mandatory fields are marked in red

Figure 15: The *Size* attribute is not visible

3.2.3 Change Default Values

Open the *crossroad* table in the tree viewer and select the column *LANES*. Set the default value to 1. At first glance, this modification might seem no different from a name or type change. You may wonder why this particular change is highlighted and explained in this tutorial. The reason lies in one of EvolveDB's key strengths, which will be detailed in the following section.

Explanation: Default-Value, Size and Validation In a MySQL database, each column can have a default value. However, this default value must adhere to specific constraints:

1. **Type Compatibility:** The default value must match the column's data type. For example, in the case of the *LANES* column, which uses a numeric data type, the default value can only consist of digits.
2. **Value Range:** The default value must fall within the allowable range for the data type.

For the *Integer* data type, the size is fixed, and EvolveDB hides the size attribute for such columns. The *Integer* data type supports values ranging from -2,147,483,648 to 2,147,483,647. If you specify a default value outside this range or enter invalid characters, an error message will be displayed during model validation (Figure 16). Validation can be triggered manually by clicking the validation button in the top-right corner of the editor.

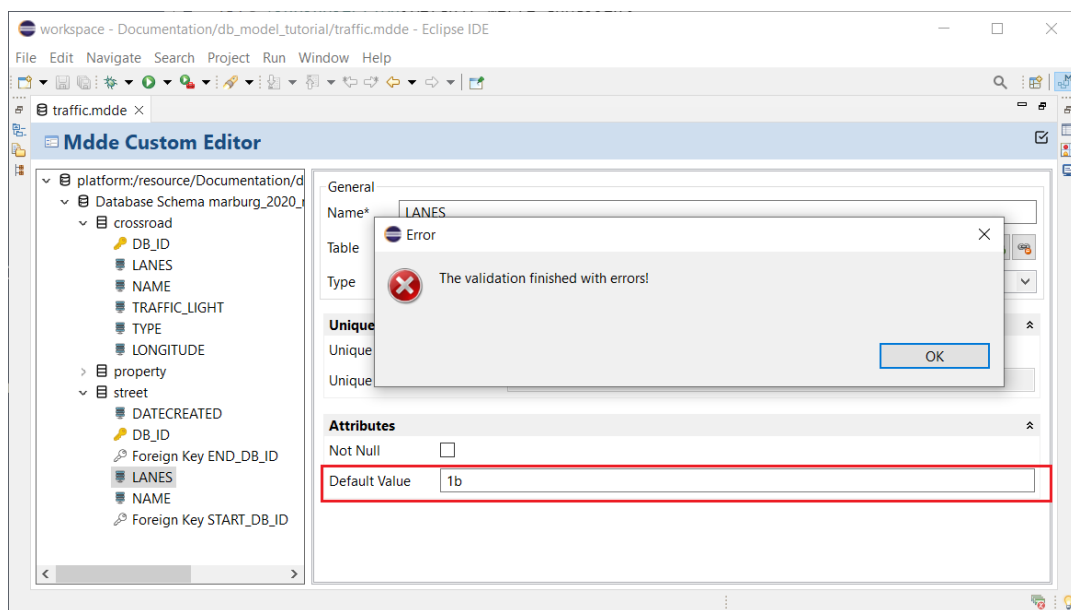


Figure 16: The default value is invalid.

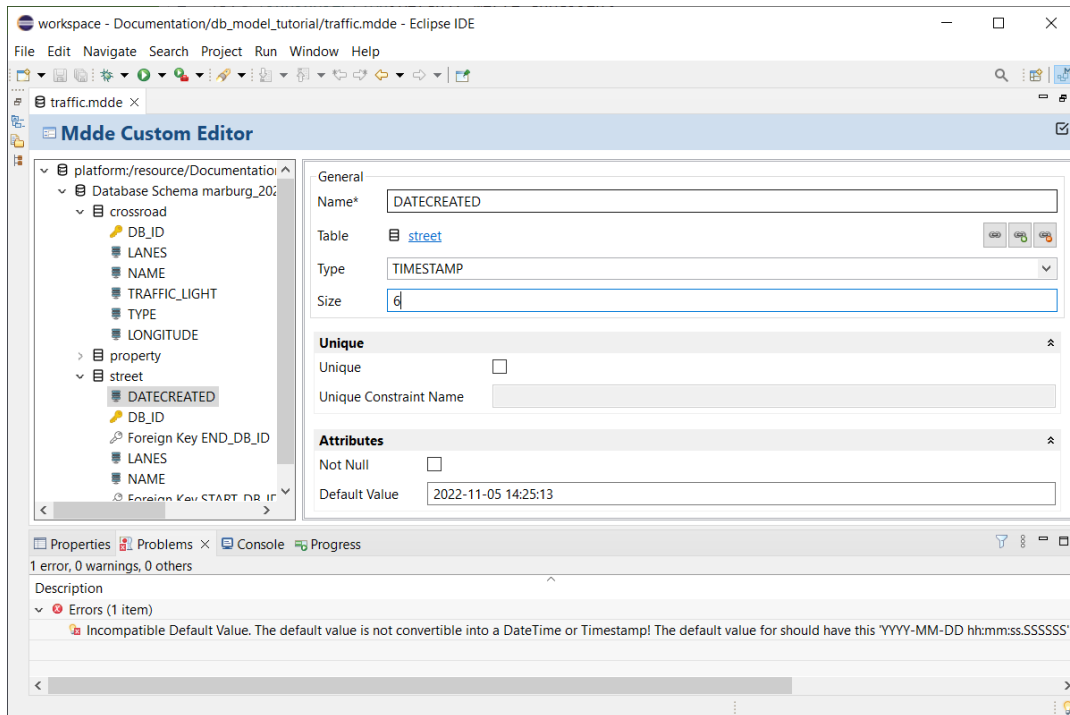


Figure 17: Default-value for a timestamp column

Let us consider another example: MySQL supports fractional seconds for the *TIME*, *DATE-TIME*, and *TIMESTAMP* data types, with a precision of up to six digits (microseconds). When validating the default value for a column using one of these data types, the default value must align with the specified *Size* attribute (Figure 17). In summary, even a simple modification to the default value or *Size* attribute requires comprehensive validation. These validations ensure that only permissible values are entered, maintaining the integrity and compatibility of the database schema.

3.2.4 Reduce Column Size

Select the *NAME* column in the *street* table. This column has the data type *VARCHAR*. Unlike numeric data types such as *INTEGER*, *VARCHAR* does not have a fixed size. Consequently, the *Size* attribute is visible and can be set to a value within the range of 0 to 255. Currently, the size is set to 255. Reduce this value to 40.

3.2.5 Change the Multiplicity of an Association

Right-clicking on the database schema in the tree-based editor opens the context menu. From this menu, you can add new tables (Figure 18). As discussed in previous sections, EvolveDB performs extensive validations. These validations apply not only to individual columns or attributes but also to entire tables and the model as a whole. For instance, after adding a new table named *property_street*, the model validation will detect several errors (Figure 19).

3 EvolveDB

These errors occur because the new table does not yet contain any columns, rendering the model invalid.

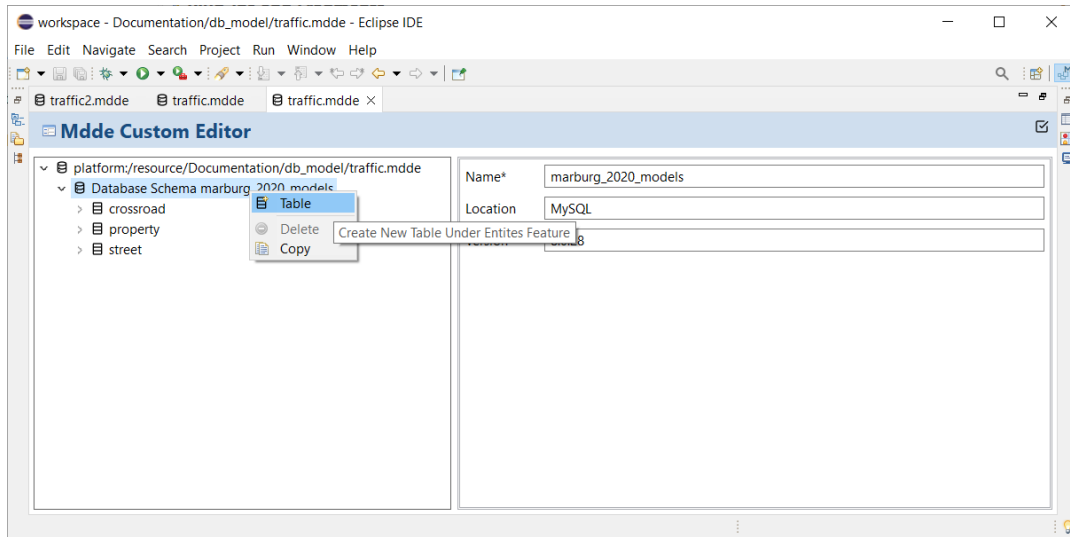


Figure 18: Add new table

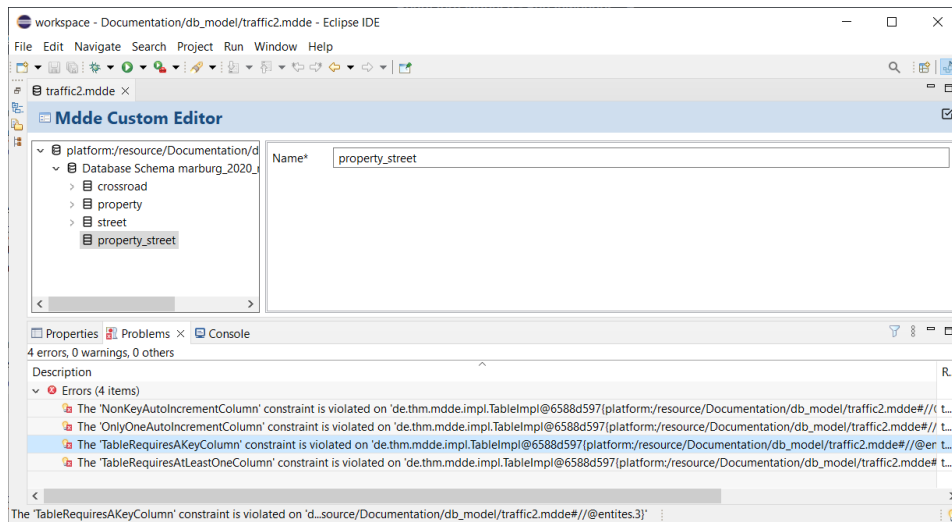
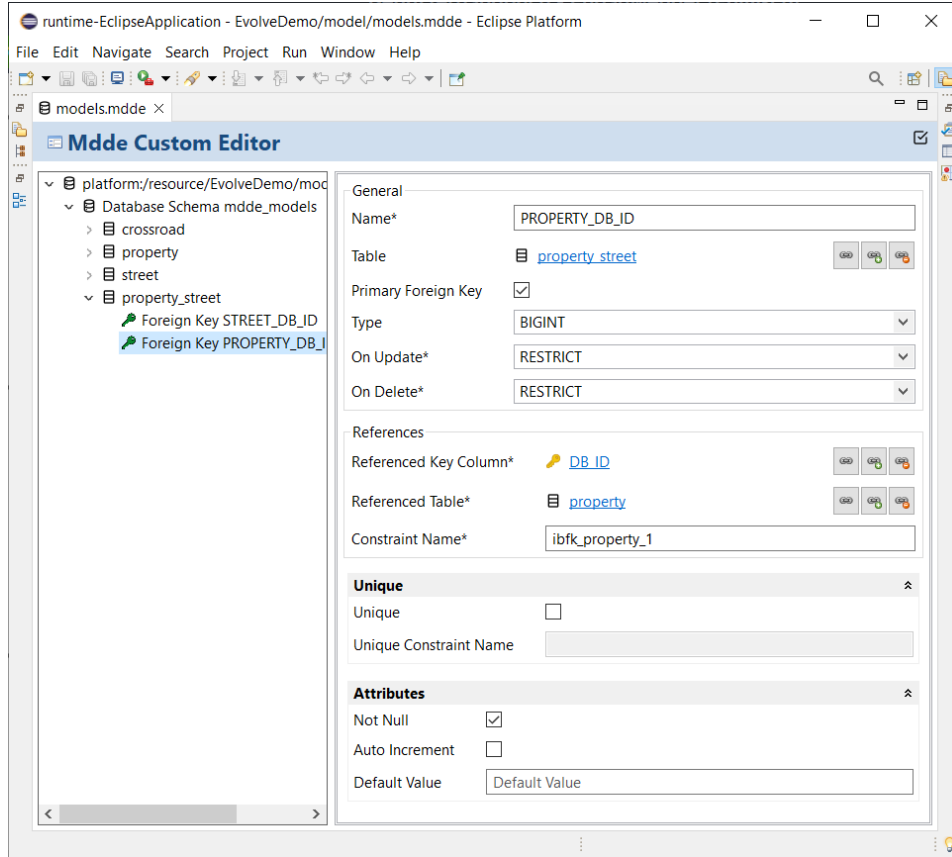


Figure 19: Model validation

Every table must have at least one column and a primary key. To address validation errors, double-click on the error message to select the model element causing the issue. Next, add two foreign keys to the *property_street* table. These foreign keys will reference the *street* and *property* tables. (Optionally, the *STREET_DB_ID* column can be moved from the *property* table to the *property_street* table.) Afterward, the old foreign key *street_db_id* in the *property* table can be safely deleted. Figure 20 illustrates the result. Once the model passes validation successfully, you can proceed to compare the model versions in the next step.

Figure 20: Table *property_street* with foreign keys

3.3 Model Comparison

EvolveDB analyzes the differences between the two model versions. Initially, a sequence of atomic differences is automatically identified. These differences are then (semi-)automatically mapped to predefined migration operations, which represent higher-level model evolution steps. Finally, these migration operations are applied to the original database schema to transform it into the target schema. For the comparison process, EvolveDB leverages the SiLift framework. SiLift is a versatile model comparison environment capable of semantically lifting low-level differences in EMF-based models into representations of user-level edit operations. This approach ensures that changes are interpreted and applied at a higher semantic level, facilitating accurate and reliable migrations.

3.3.1 SiLift

SiLift's approach can best be compared to a four-step pipeline, as shown in Figure 21. The two model versions, e.g. *models.mdde* and *models2.mdde* serve as input:

1. **Matching:** The task of a matcher is to identify the corresponding elements from model A and model B, i.e., the elements that match in both models. SiLift provides four different matchers by default. One of them is EMFCompare.

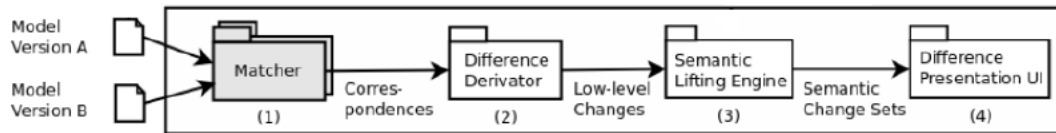


Figure 21: SiLift processing pipeline

2. **Difference Derivation:** Based on the correspondences, the difference derivator calculates a technical difference (low-level difference) between the model versions. All objects and references for which no correspondence exists must either have been added to model B or removed from model A.

```

Δ platform:/resource/Documentation/db_model_tutorial/test.symmetric
├─ Δ Symmetric-Difference: trafficV2.mdde <-> traffic.mdde
│   ├── ✖ Remove-Object: STREET_DB_ID
│   ├── ✖ Remove-Reference: table (STREET_DB_ID -> property)
│   ├── ✖ Remove-Reference: referencedTable (STREET_DB_ID -> street)
│   ├── ✖ Remove-Reference: referencedKeyColumn (STREET_DB_ID -> DB_ID)
│   ├── ➕ Add-Object: LATITUDE
│   ├── ➕ Add-Reference: table (LATITUDE -> crossroad)
│   ├── ➕ Add-Object: LONGITUDE
│   ├── ➕ Add-Reference: table (LONGITUDE -> crossroad)
│   ├── ➕ Add-Object: property_street
│   ├── ➕ Add-Reference: schema (property_street -> marburg_2020_complex)
│   ├── ➕ Add-Reference: columns (property_street -> STREET_DB_ID)
│   ├── ➕ Add-Reference: columns (property_street -> PROPERTY_DB_ID)
│   ├── ➕ Add-Object: STREET_DB_ID
│   ├── ➕ Add-Reference: table (STREET_DB_ID -> property_street)
│   ├── ➕ Add-Reference: referencedTable (STREET_DB_ID -> street)
│   ├── ➕ Add-Reference: referencedKeyColumn (STREET_DB_ID -> DB_ID)
│   ├── ➕ Add-Object: PROPERTY_DB_ID
│   ├── ➕ Add-Reference: table (PROPERTY_DB_ID -> property_street)
│   ├── ➕ Add-Reference: referencedTable (PROPERTY_DB_ID -> property)
│   ├── ➕ Add-Reference: referencedKeyColumn (PROPERTY_DB_ID -> DB_ID)
│   ├── ➕ Add-Reference: entites (marburg_2020_complex -> property_street)
│   ├── Δ Attribute-Value-Change: DATECREATED -> DATECREATED
│   ├── Δ Attribute-Value-Change: NAME -> NAME
│   ├── ✖ Remove-Reference: referencedBy (DB_ID -> STREET_DB_ID)
│   ├── ➕ Add-Reference: referencedBy (DB_ID -> STREET_DB_ID)
│   ├── ➕ Add-Reference: columns (crossroad -> LATITUDE)
│   ├── ➕ Add-Reference: columns (crossroad -> LONGITUDE)
│   ├── Δ Attribute-Value-Change: LANES -> LANES
│   ├── ✖ Remove-Reference: columns (property -> STREET_DB_ID)
│   ├── ➕ Add-Reference: referencedBy (DB_ID -> PROPERTY_DB_ID)
│   └─ > Matching (trafficV2.mdde <-> traffic.mdde)
├─ > platform:/resource/Documentation/db_model_tutorial/genModel/trafficV2.mdde
└─ > platform:/resource/Documentation/db_model_tutorial/traffic.mdde

```

Figure 22: Technical difference between models.mdde and models2.mdde

3. **Semantic Lifting:** The previously calculated technical difference contains all changes between the models. Then these changes are semantically lifted. For this purpose, the low-level changes are grouped into semantic change sets with the help of recognition rules. Each semantic change set represents an editing operation performed by the user. Often, user editing operations consist of more than one low-level change, because even basic edit operations that appear simple from a users point of view may lead to many low-level changes. While atomic rules cover the creation, deletion, moving of elements, and the changing of attribute values, the complex editing rules are usually composed of atomic and other complex rules.
4. **Difference Presentation UI:** The result is a symmetric difference model which contains the matching, the atomic changes and the resulting semantic change sets.

3.3.2 Model Comparison with EvolveDB

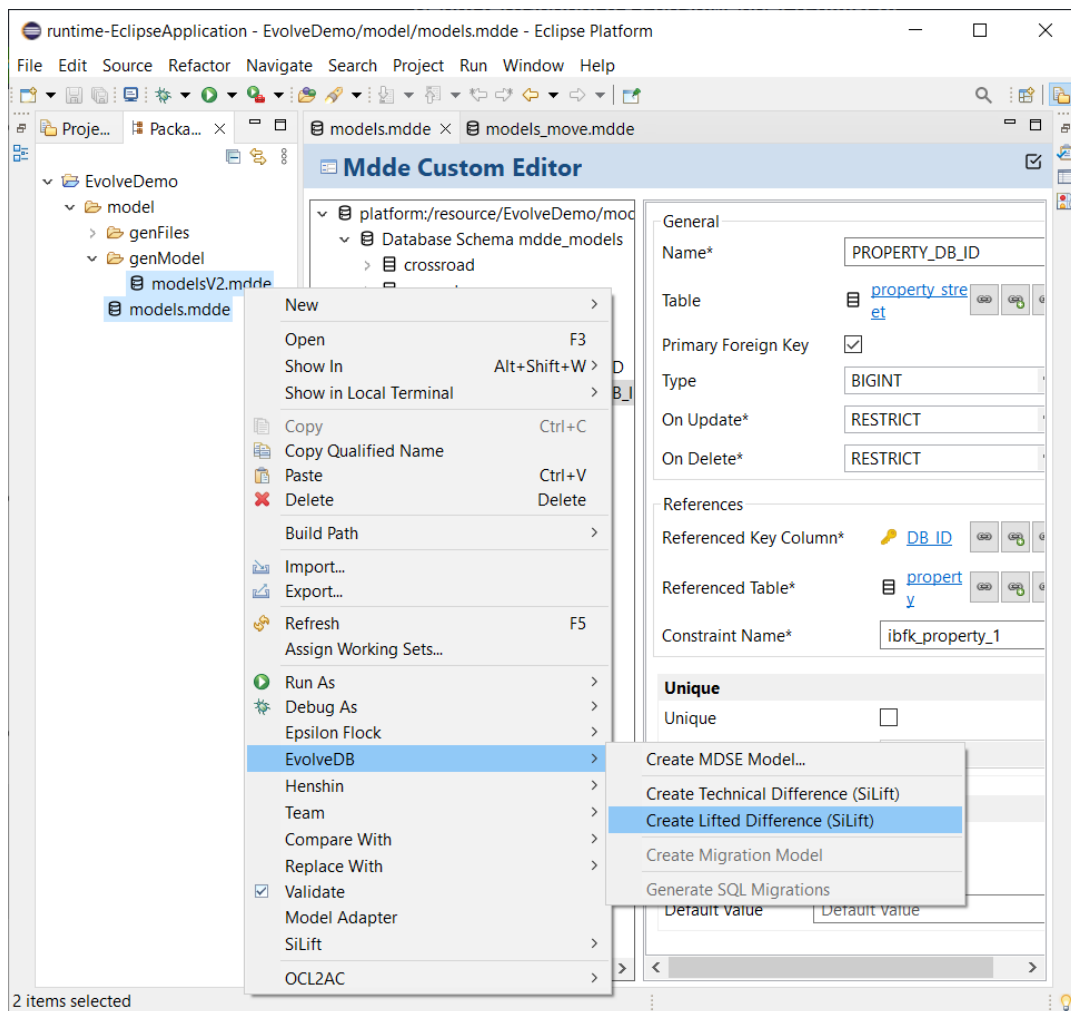


Figure 23: Start comparison

To compare the two model versions created in the previous chapter, follow these steps:

1. In the **Package** or **Project Explorer**, select the two **mdde** files you want to compare.
2. Right-click to open the context menu and choose **EvolveDB Create Lifted Difference (SiLift)** (Figure 23).

A multi-page wizard dialog will appear.

On the first page of the wizard, select the model comparison direction. Ensure that the arrow points to the edited model version (Figure 24).

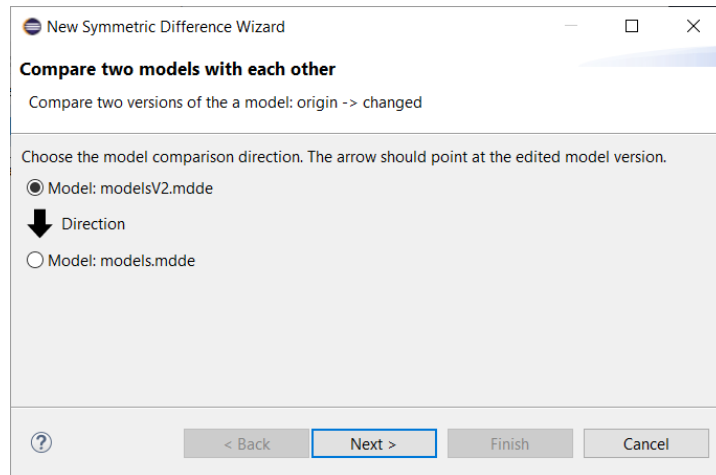


Figure 24: Choose model comparison direction

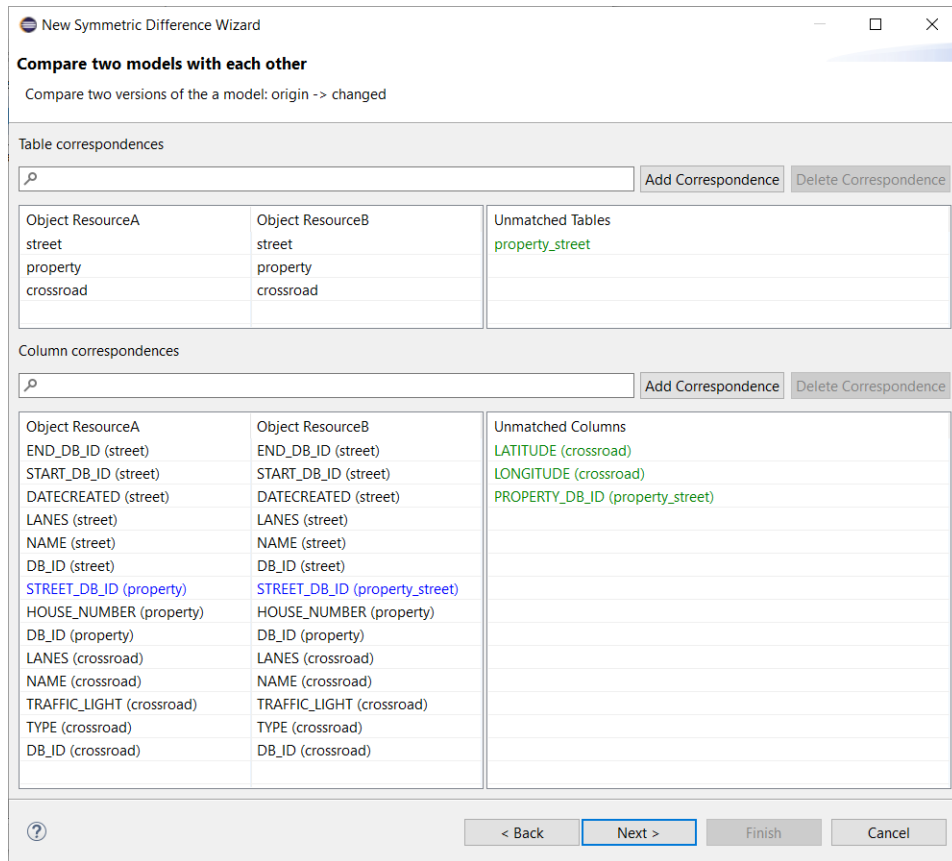
On the second page of the wizard (Figure 25), the matches identified by *SiLift* are displayed.

- The **upper table** lists the corresponding tables from Model A and Model B. Unmatched tables are highlighted to indicate their status:
 - Tables added to Model B are marked in **green**.
 - Tables removed from Model A are marked in **red**.
- The **lower table** displays the corresponding columns. Columns that have been moved or renamed are highlighted in **blue**. For example, the foreign key *STREET_DB_ID* has been moved to the table *property_street*.

If a correspondence is incorrectly identified, you can manually delete or add correspondences to ensure accuracy.

On the final page of the wizard dialog (Figure 26), you need to specify a file name for the symmetric difference model. Additionally, you can choose to create a migration model at this stage. Since the migration model will be required later in the process, it is advisable to create it now. The migration model will be stored in the same location as the symmetric difference model for consistency.

3 EvolveDB



New Symmetric Difference Wizard

Compare two models with each other

Compare two versions of the a model: origin -> changed

Table correspondences

Search:

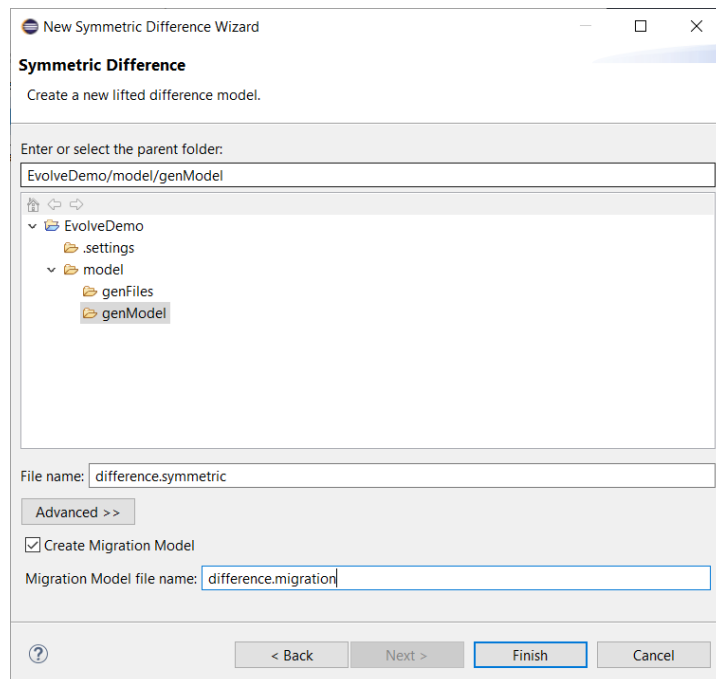
Object ResourceA	Object ResourceB	Unmatched Tables
street	street	property_street
property	property	
crossroad	crossroad	

Column correspondences

Search:

Object ResourceA	Object ResourceB	Unmatched Columns
END_DB_ID (street)	END_DB_ID (street)	LATITUDE (crossroad)
START_DB_ID (street)	START_DB_ID (street)	LONGITUDE (crossroad)
DATECREATED (street)	DATECREATED (street)	PROPERTY_DB_ID (property_street)
LANES (street)	LANES (street)	
NAME (street)	NAME (street)	
DB_ID (street)	DB_ID (street)	
STREET_DB_ID (property)	STREET_DB_ID (property_street)	
HOUSE_NUMBER (property)	HOUSE_NUMBER (property)	
DB_ID (property)	DB_ID (property)	
LANES (crossroad)	LANES (crossroad)	
NAME (crossroad)	NAME (crossroad)	
TRAFFIC_LIGHT (crossroad)	TRAFFIC_LIGHT (crossroad)	
TYPE (crossroad)	TYPE (crossroad)	
DB_ID (crossroad)	DB_ID (crossroad)	

Figure 25: Choose model comparison direction



New Symmetric Difference Wizard

Symmetric Difference

Create a new lifted difference model.

Enter or select the parent folder:

EvolveDemo

- .settings
- model
 - genFiles
 - genModel

File name:

☒ Create Migration Model

Migration Model file name:

Figure 26: Choose file location

The symmetric difference model encapsulates the matching, atomic changes, and the resulting semantic change sets. Figure 27 illustrates the symmetric difference model for our example, opened in the symmetric difference model editor. When a semantic change set is expanded, all associated atomic changes are displayed. In our example (Figure 27), the complex operator *CHANGE_1N_INT0_NM*, which represents the multiplicity change, comprises 21 atomic changes.

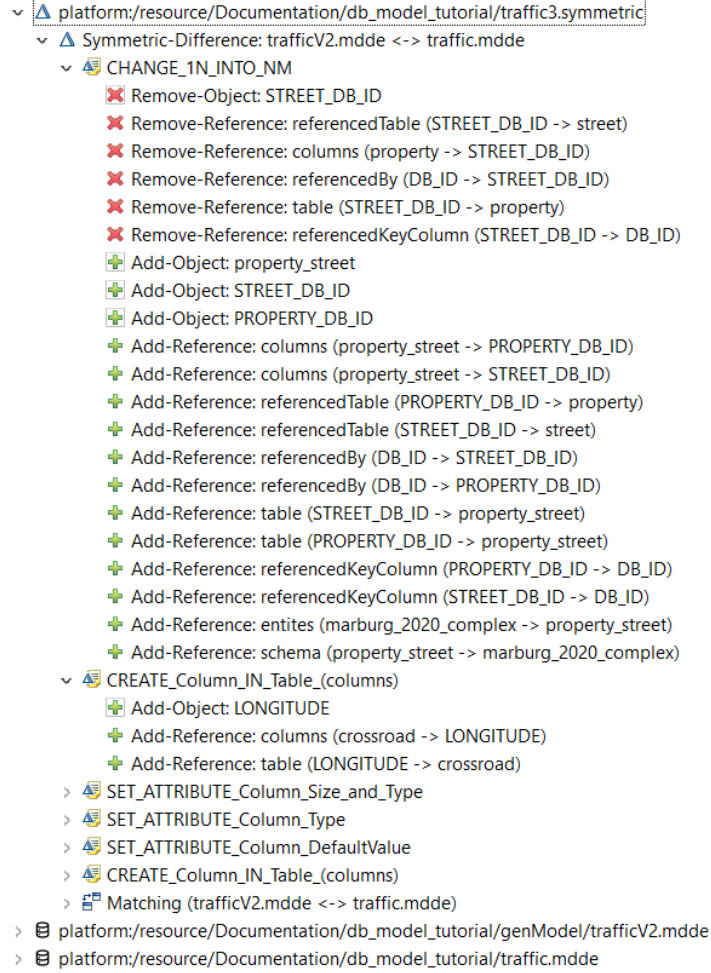


Figure 27: Lifted difference between traffic.mdde and traffic2.mdde

Recalling the changes we made earlier, the semantic change sets can be mapped to the corresponding editing operations as follows:

- **SET_ATTRIBUTE_Column_Size_and_Type:** Reduce the column *size* of *NAME* in the table *street* from 255 to 40.
- **CREATE_Column_IN_Table_(columns):** Create the new column *LONGITUDE* in the table *crossroad*.

- **CREATE_Column_IN_Table_(columns)**: Create the new column *LATITUDE* in the table *crossroad*.
- **CHANGE_1N_INTO_NM**: Transform the single-valued association between *property* and *street* into a multi-valued association. This operator includes both the creation of the new cross-reference table and the deletion of the old foreign key.
- **SET_ATTRIBUTE_Column_Type**: Change the column type of *DATECREATED* from *TIMESTAMP* to *DATETIME*.
- **SET_ATTRIBUTE_Column_DefaultValue**: Update the default value for the column *LANES*.

All currently supported operators are listed in table 1.

Name	Classification
Rename Element	VALUE CHANGE
Make ForeignKey a PrimaryForeignKey	VALUE CHANGE
Set foreign key on update constraint	VALUE CHANGE
Set foreign key on delete constraint	VALUE CHANGE
Set foreign key constraint name	VALUE CHANGE
Make column unique	VALUE CHANGE
Set column unique constraint name	VALUE CHANGE
Set column type	VALUE CHANGE
Set column size	VALUE CHANGE
Set column not null	VALUE CHANGE
Set column default value	VALUE CHANGE
Set column auto-increment attribute	VALUE CHANGE
Drop primary key	REDUCE
Drop table	REDUCE
Drop foreign key	REDUCE
Drop column	REDUCE
Create primary key	INCREASE
Create foreign key	INCREASE
Create table	INCREASE
Create column	INCREASE
Create many-to-many table	STRUCTURAL CHANGE
Move column	STRUCTURAL CHANGE
Dissolve many-to-many table	STRUCTURAL CHANGE
Join table	STRUCTURAL CHANGE

Table 1: Supported edit operations

3.4 Forward Engineering

In the next step, we will transform the edit operations into a SQL migration script. Each editing operation falls into one of the following categories:

- **Non-breaking:** These changes can be resolved automatically, with no impact on existing data. Most additions belong to this category.
- **Breaking and resolvable:** These changes disrupt the conformance of existing data but can be automatically adapted without user intervention.
- **Breaking and unresolvable:** These changes disrupt the conformance of existing data and cannot be automatically resolved. They require user input to determine how the data migration should be handled. For instance, if a column size is reduced, the user must specify how to manage data entries that exceed the new size limit.

Changes in the first category do not require data migration and are thus not relevant to the migration process. The second category involves changes that necessitate data migration but can be resolved without user input. The third category requires user attention to provide additional information essential for migrating the affected data.

3.4.1 Migration Model

The symmetric difference model created in the previous section is solely intended to represent the differences between the two model versions. However, it is not suitable for adding the additional information required for migration. To address this, the difference.symmetric model must be converted into a migration model (if this was not already done in the previous step). To perform this conversion:

Select the symmetric difference model in the Project or Package Explorer. Right-click to open the context menu and choose **EvolveDB** ⇒ **Create Migration Model...** (Figure 28). The migration model will be saved in the same folder as the symmetric difference model. If the migration model does not appear immediately, refresh the project or folder to make it visible.

3 EvolveDB

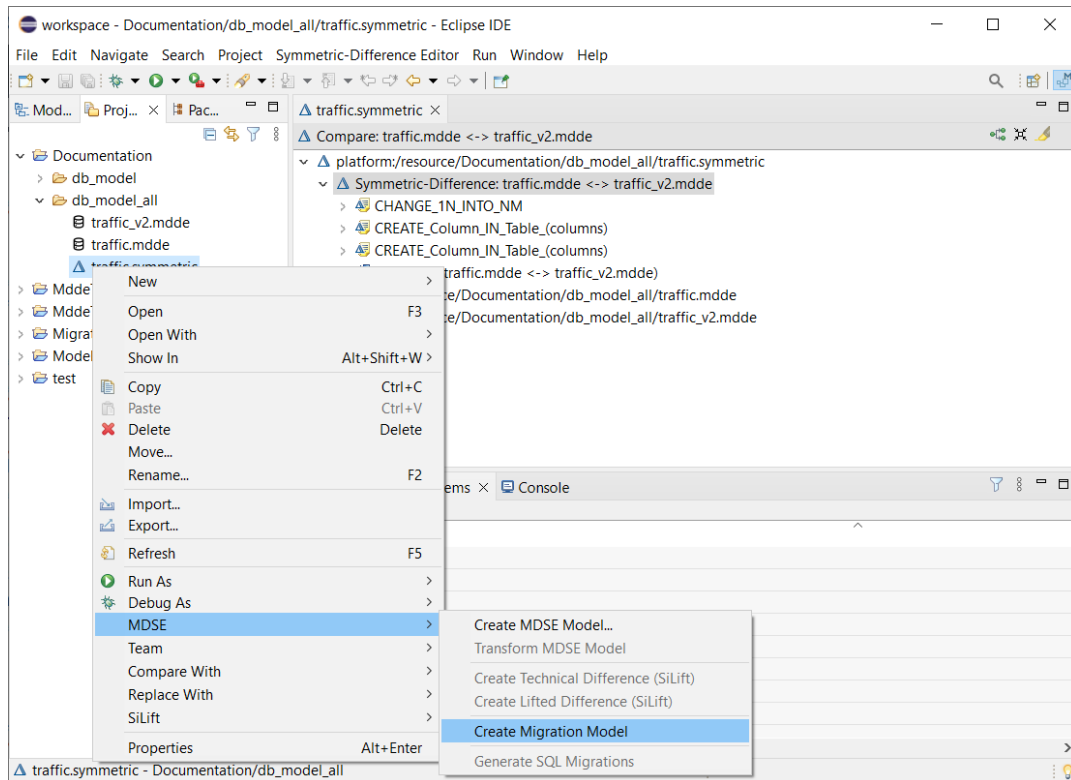


Figure 28: Create migration model

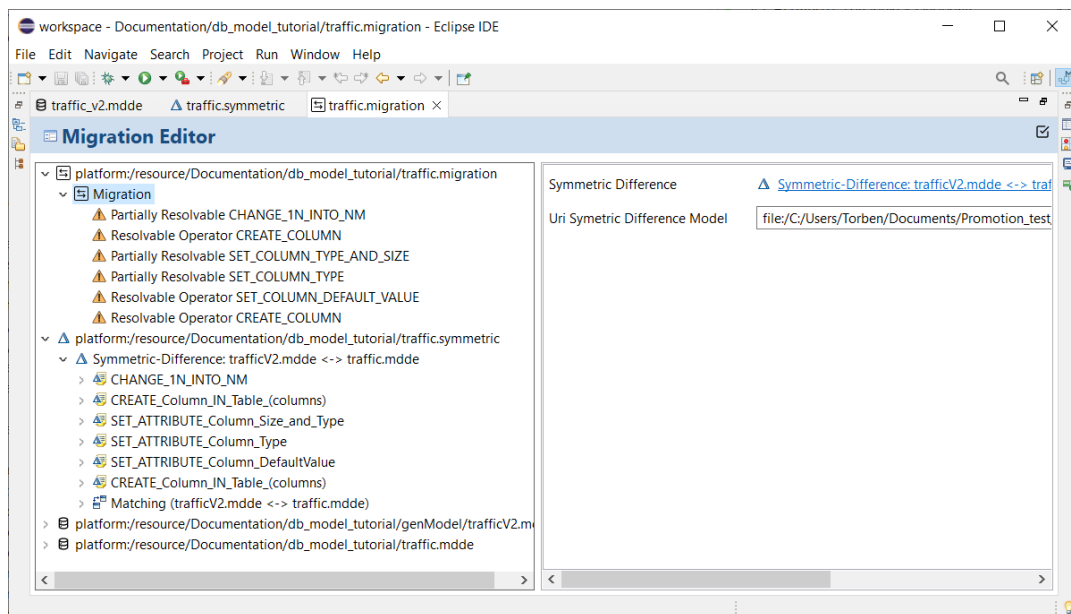


Figure 29: Migration model editor

The new model should open automatically in the **Migration Editor**. This editor is divided

into two sections:

- **Left Pane:** A tree-based navigation area to explore the model.
- **Right Pane:** Displays the details of the currently selected element.

The migration model references all previously created models, resulting in four models listed in the navigation view: the two versions of the traffic model, the difference model, and the migration model (Figure 29). During the transformation process, operators are classified based on the categorization discussed in the previous section:

- **Resolvable:** These operators are part of the *non-breaking* category and do not impact existing data. For instance, the *CREATE_COLUMN* operator falls into this group.
- **Partially Resolvable:** These operators are part of the *breaking and resolvable* category.
- **Not Automatically Resolvable:** These operators belong to the *breaking and unresolvable* category and require user intervention.

This classification ensures a clear understanding of the migration operations and their potential impact on the database schema and data.

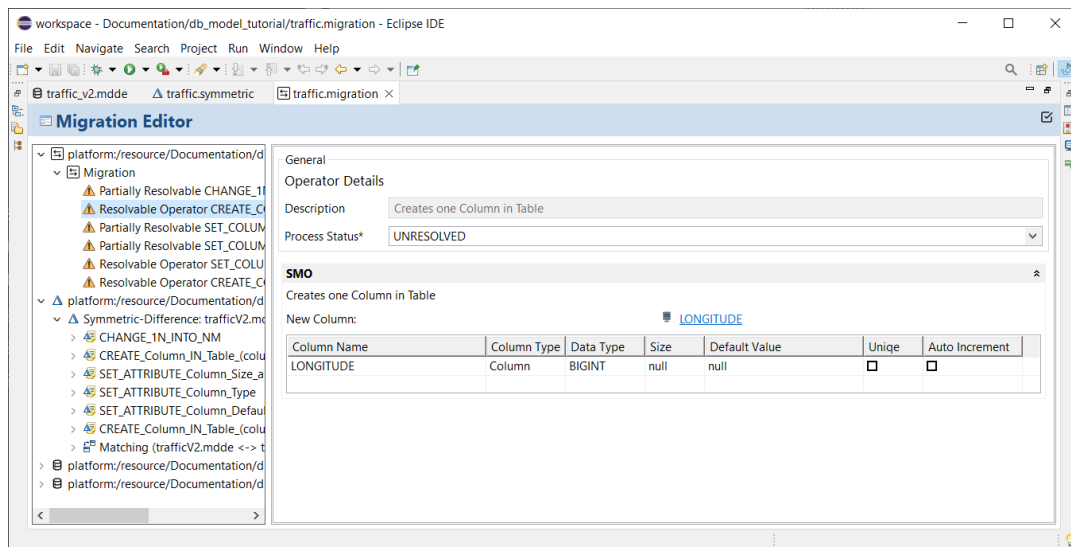


Figure 30: Resolvable operator Create Column

When selecting an operator, its details are displayed in the right-hand pane of the editor (Figure 30). Every operator should be reviewed by the developer to ensure accuracy, as not all change operations are fully covered yet.

Each operator is assigned a **process status**, which can be modified using the drop-down menu beneath the operator's description. The three possible statuses are:

- **UNRESOLVED:** This is the default value. When the migration model is created, all operators are initially marked as unresolved. This status indicates that the operator has not yet been reviewed or confirmed by the user.
- **RESOLVED:** This status signifies that the operator has been reviewed and approved by the user. Operators marked as resolved will be included when generating the migration scripts.
- **IGNORE:** If an operator is deemed incorrect or irrelevant, the status can be set to *IGNORE*. This eliminates the need to repeat the comparison or recreate the migration model. Operators with this status will be excluded from subsequent processes.

Operators marked as *UNRESOLVED* are highlighted with a warning icon in the tree-based editor. Once a status of *RESOLVED* or *IGNORE* is selected, the warning icon is replaced by a green checkmark, indicating that the operator has been processed (Figure 31).

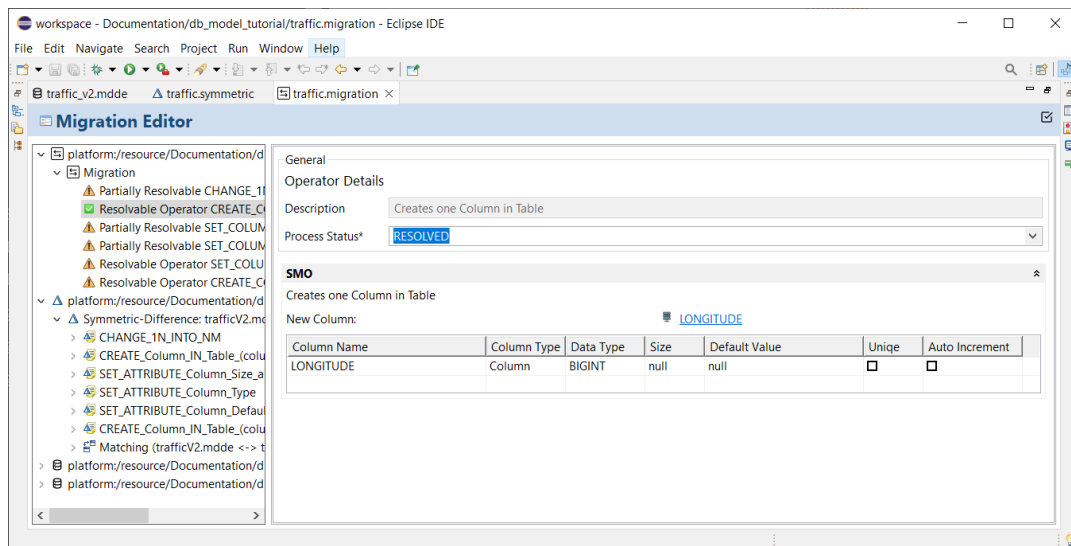


Figure 31: Operator was marked as resolved

3.4.2 Breaking & Resolvable

All operators requiring data migration typically belong to the *Breaking & Resolvable* group. For these operators, it is essential to specify how, and whether, EvolveDB should migrate the data. To facilitate this, EvolveDB provides various migration strategies that depend on the specific change operator. Lets explore this in more detail with three examples, starting with the following:

3.4.3 CHANGE_1N_INTO_NM

Changing the multiplicity between *property* and *street* involves several operations (Figure 33):

1. A new table is created containing two foreign keys.

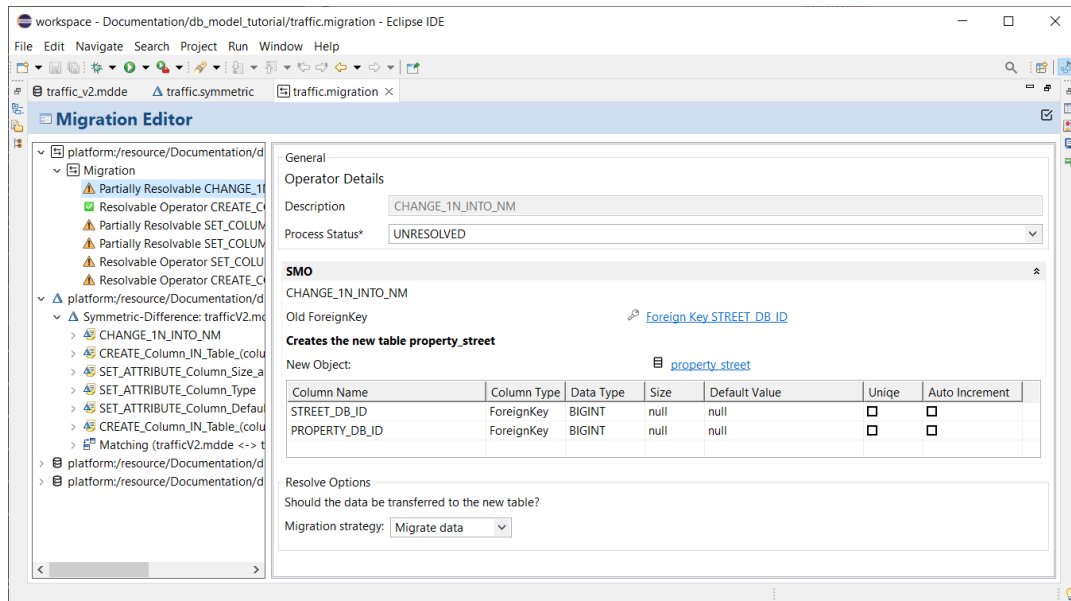


Figure 32: Partially Resolvable Operator

2. Existing data is transferred to the new table.
3. The old foreign key is deleted.

EvolveDB offers two migration strategies for this change. You can select the desired strategy from a dropdown menu (Figure 33):

- **Migrate Data:** Transfers the existing data to the newly created table, ensuring no data is lost.
- **Do Not Migrate Data:** Skips data migration. Selecting this option will result in data loss unless handled manually.

In this case, the choice of strategy is straightforward, with clear implications for data preservation.

SET_COLUMN_SIZE: In this example, we reduced the size of the *VARCHAR* column from 255 to 40 (Figure 34). If the column contains values exceeding the new size, the migration process will fail with an error. To address this issue, EvolveDB provides several migration strategies based on the column's constraints and properties:

3.4.4 Migration Strategies:

1. **No Violating Data:** No values in the column exceed the new constraint, so no additional actions are required.
2. **If the Column Has a Default Value and No Unique Constraint:**

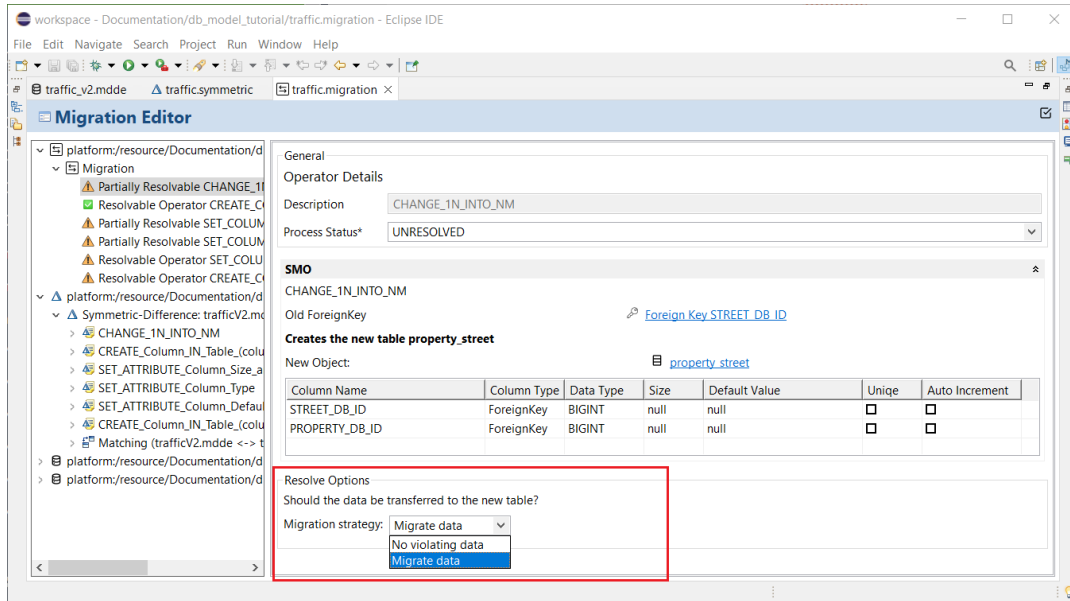


Figure 33: Migration strategy

- **Set Column to Default Value:** All values in the column are replaced with the default value.
- **Set Violating Rows to Default Value:** Only values exceeding 40 characters are replaced with the default value.

3. If the Column Is Nullable:

- **Set Column to Null:** All values in the column are replaced with NULL.
- **Set Violating Rows to Null:** Only values exceeding 40 characters are replaced with NULL.

SET_COLUMN_TYPE: In the third example, we examine a type change. Type changes fall into the second group (Breaking Resolvable) because existing data might be incompatible with the target type. EvolveDB automatically verifies the compatibility between the old and new data types. Based on the compatibility check, different migration strategies are provided. In our example (Figure 35), we changed the column type from `TIMESTAMP` to `DATETIME`. Since these two data types are compatible, no migration strategy needs to be selected in this case.

3.4.5 Not Automatically Resolvable

-TBD-

3 EvolveDB

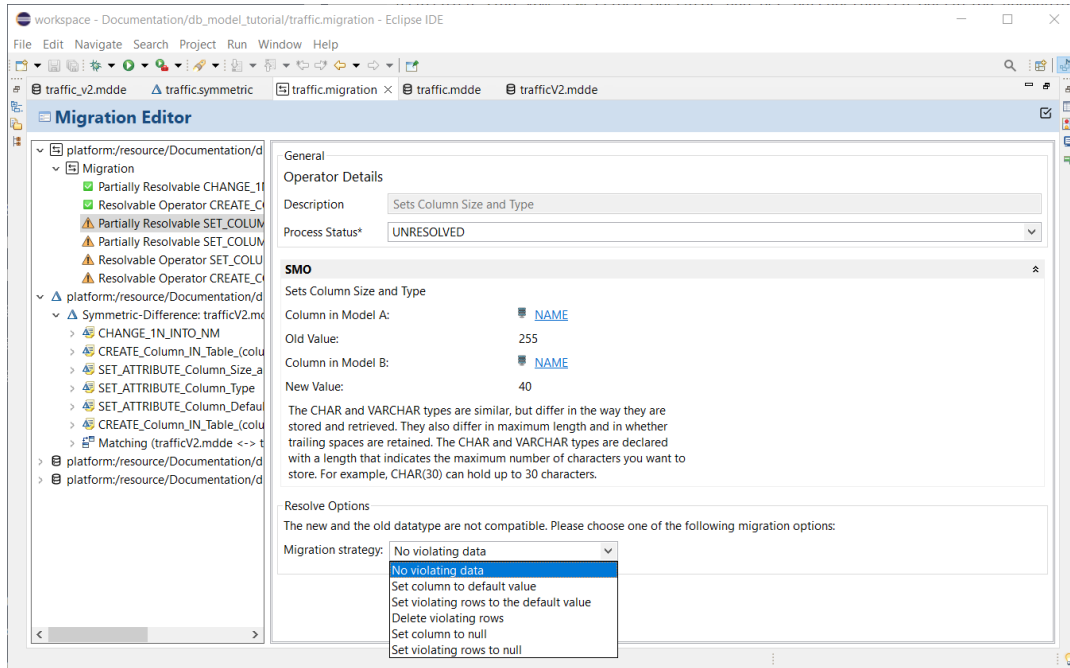


Figure 34: Migration strategy

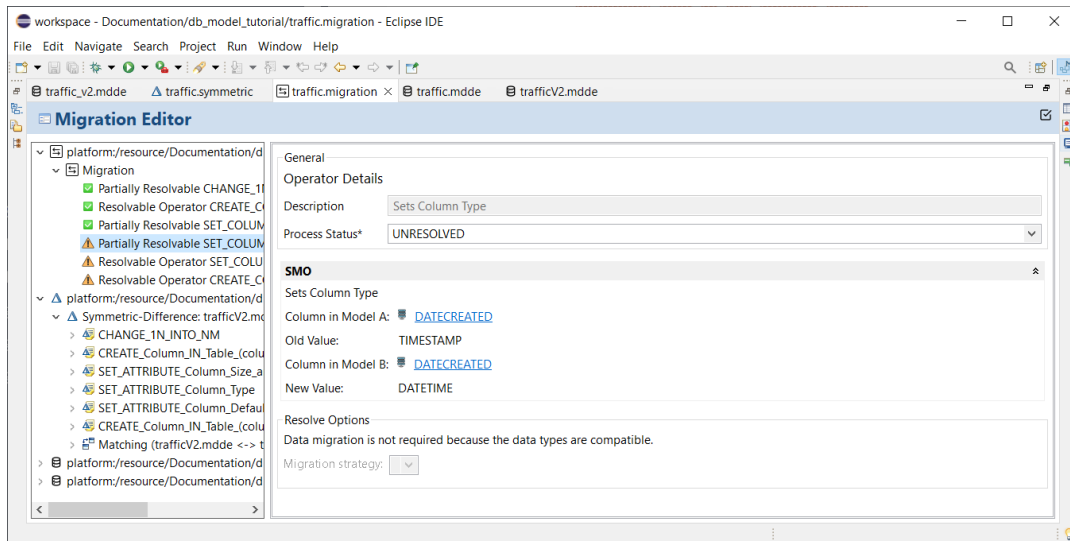


Figure 35: Migration strategy

3.5 Generate Migration Scripts

After selecting a migration strategy and updating the process status, the next step is to generate the migration script.

1. In the **Package Explorer**, select the migration model.

2. Right-click to open the context menu and choose **MDSE** ⇒ **Generate SQL Migrations** (Figure 36).
3. Alternatively, you can start the migration script generation directly from the toolbar in the **Migration Editor**.

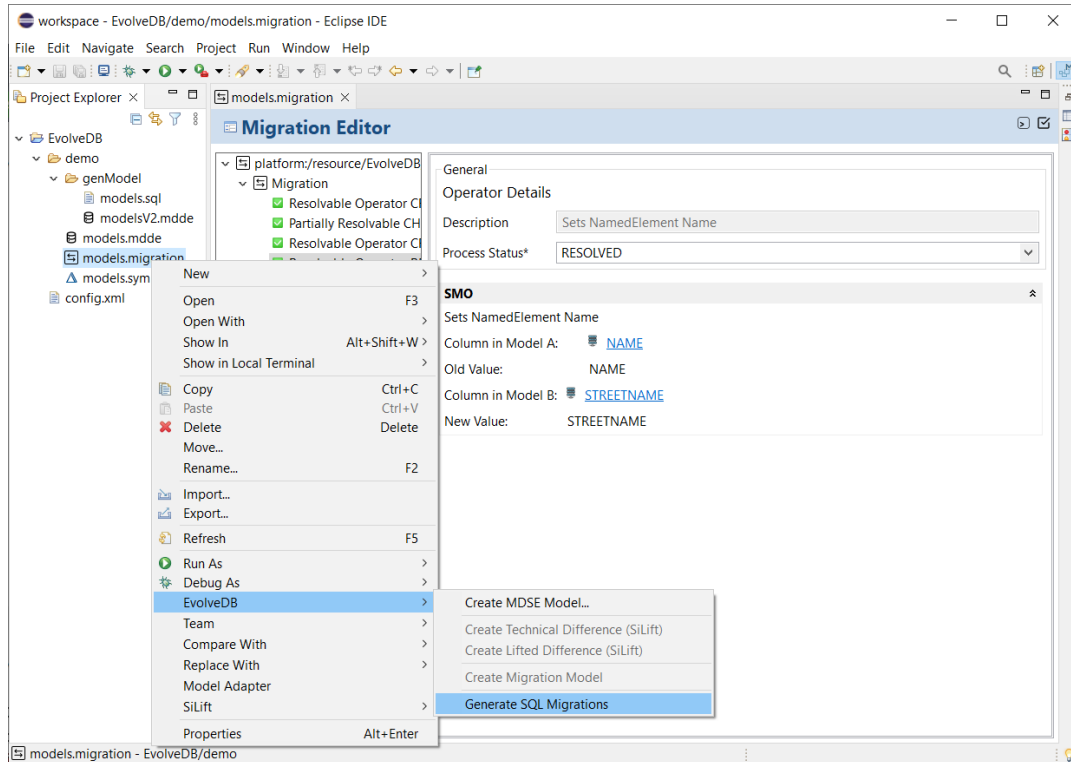


Figure 36: Create SQL migration script

A wizard dialog opens to guide the generation process. On the first page of the dialog, you must select the generator. Since SQL commands can vary depending on the database system or its version, EvolveDB provides an extension point for generators to accommodate these differences. In the default installation, only the MySQL-compatible generator is available (Figure 37).

On the second page of the dialog, you need to select the location where the SQL script will be stored (Figure 38). Currently, only projects within the workspace can be chosen as storage locations. Click **Finish** to start the generation process. If the generated file does not appear immediately, refresh the project to make it visible.

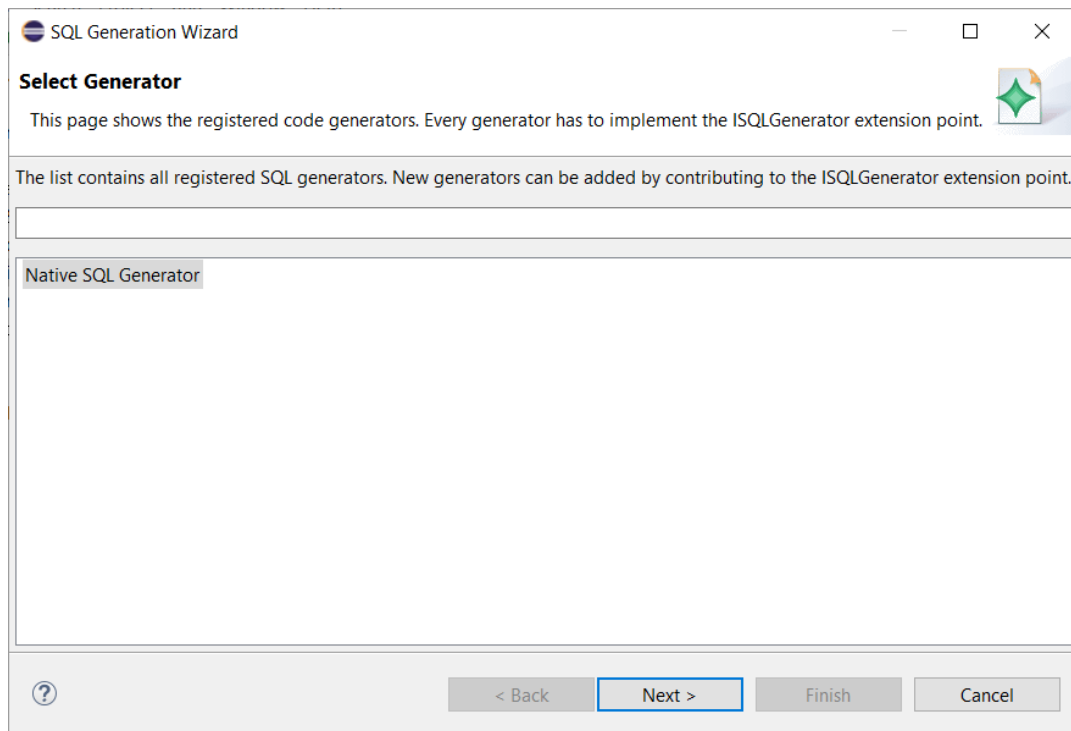


Figure 37: Select generator

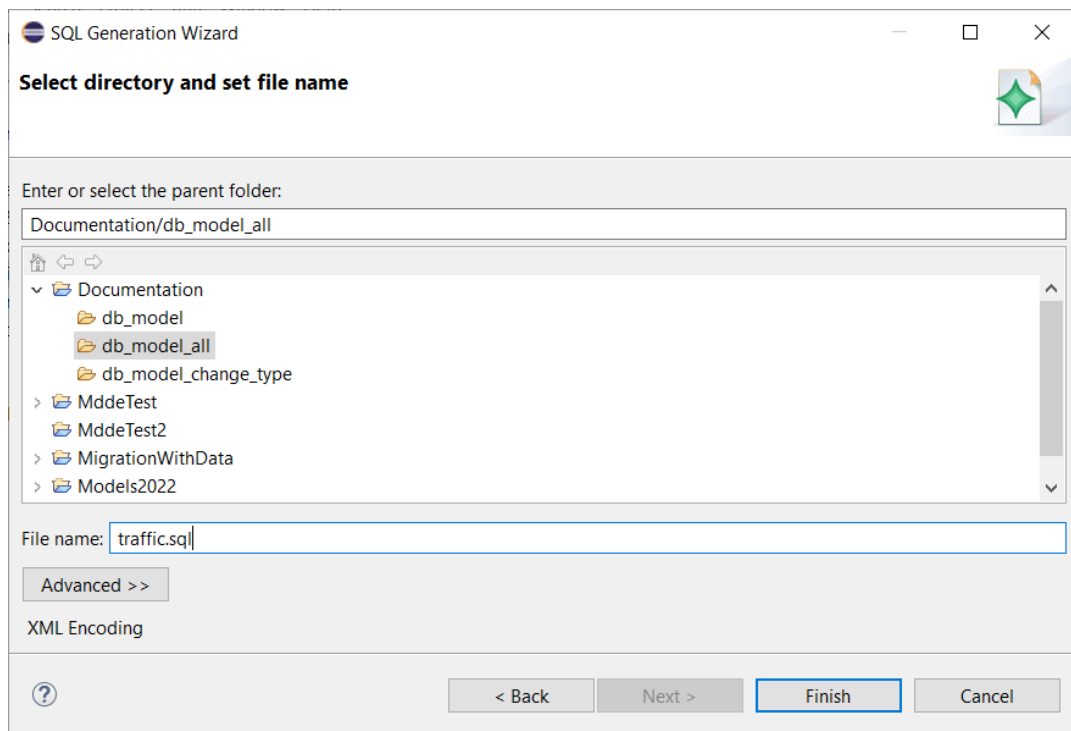


Figure 38: Choose file location

4 Migration Scripts

In this chapter, we will examine the generated migration script. Listing 1 presents the script generated using the model from our running example (Figure 29).

```

USE marburg_2020_models3;
START TRANSACTION;

-- Creates an history table for deleted and updated values
CREATE TABLE IF NOT EXISTS 'marburg_2020_models3'.'
    mdde_history' (
        'DB_ID' BIGINT NOT NULL AUTO_INCREMENT,
        'TABLENAME' VARCHAR(255) NOT NULL,
        'COLUMN_DB_ID' BIGINT NOT NULL,
        'COLUMN_NAME' VARCHAR(255) NOT NULL,
        'VALUE' BLOB NULL,
        'CHANGEDATE' DATETIME NOT NULL,
        PRIMARY KEY ('DB_ID'));

-- Add the new column LONGITUDE in Table crossroad
ALTER TABLE 'crossroad'
ADD COLUMN 'LONGITUDE' BIGINT;

-- Change default value of lanes
ALTER TABLE 'crossroad' CHANGE COLUMN 'LANES' 'LANES' BIGINT
    NULL DEFAULT 1 ;

-- Add the new column LATITUDE in Table crossroad
ALTER TABLE 'crossroad'
ADD COLUMN 'LATITUDE' BIGINT;

-- Change column type of datecreated
ALTER TABLE 'street' CHANGE COLUMN 'DATECREATED' '
    DATECREATED' DATETIME(0) NULL ;

-- Create Table property_street
CREATE TABLE IF NOT EXISTS property_street (
    'STREET_DB_ID' BIGINT NOT NULL,
    'PROPERTY_DB_ID' BIGINT NOT NULL
    ,PRIMARY KEY('STREET_DB_ID' , 'PROPERTY_DB_ID' ),
    CONSTRAINT 'property_crossroad_ibfk2'
    FOREIGN KEY ('STREET_DB_ID')
    REFERENCES 'street'('DB_ID')
    ON DELETE RESTRICT

```

4 Migration Scripts

```
ON UPDATE RESTRICT, 37
CONSTRAINT 'property_crossroad_ibfk3' 38
FOREIGN KEY ('PROPERTY_DB_ID') 39
REFERENCES 'property'('DB_ID') 40
ON DELETE RESTRICT 41
ON UPDATE RESTRICT 42
); 43
44
BEGIN; 45
SET @safe_mode = @@SQL_SAFE_UPDATES; 46
SET SQL_SAFE_UPDATES = 0; 47
48
-- Migrate data to the new table 49
INSERT INTO 'property_street' (PROPERTY_DB_ID , STREET_DB_ID 50
)
SELECT DB_ID, STREET_DB_ID FROM property WHERE STREET_DB_ID 51
IS NOT NULL; 52
53
SET SQL_SAFE_UPDATES = @safe_mode; 53
COMMIT; 54
-- If executing the script fails, we suggest a rollback. 55
56
-- Drop foreign key in property 57
ALTER TABLE 'property' DROP FOREIGN KEY 'ibfk_street'; 58
ALTER TABLE 'property' DROP COLUMN 'STREET_DB_ID'; 59
60
-- Find violating rows 61
SET @sql_mode = @@SESSION.sql_mode; 62
set @@SESSION.sql_mode = ''; 63
DROP TEMPORARY TABLE IF EXISTS my_temp_id_table; 64
CREATE TEMPORARY TABLE my_temp_id_table 65
    SELECT DB_ID from street v where LENGTH('NAME') > 40; 66
set @@SESSION.sql_mode = @sql_mode; 67
68
BEGIN; 69
SET @safe_mode = @@SQL_SAFE_UPDATES; 70
SET SQL_SAFE_UPDATES = 0; 71
72
-- Insert violating values into the history table 73
INSERT INTO 'marburg_2020_models3'.'mdde_history' 74
('TABLENAME', 75
'COLUMN_DB_ID', 76
'COLUMN_NAME', 77
'VALUE', 78
```

```

'CHANGEDATE') 79
SELECT 'street', DB_ID, 'NAME', NAME, now() from street s 80
    where LENGTH('NAME') > 40; 81
-- Set violating rows to the default value 82
UPDATE 'street' SET 'NAME' = 'unnamed_road' where DB_ID in ( 83
    Select DB_ID from my_temp_id_table); 84
SET SQL_SAFE_UPDATES = @safe_mode; 85
COMMIT; 86
-- If executing the script fails, we suggest a rollback. 87
88
DROP TEMPORARY TABLE IF EXISTS my_temp_id_table; 89
90
-- Change column size of name 91
ALTER TABLE 'street' CHANGE COLUMN 'NAME' 'NAME' VARCHAR(40) 92
    NULL DEFAULT 'unnamed_road';
COMMIT; 93
-- If executing the script fails, we suggest a rollback. 94

```

Listing 1: Generated SQL code

Applying the script to the original schema (Figure 5) results in the migrated schema shown in Figure 39.

The following changes are reflected in the new schema:

- The new columns *LATITUDE* and *LONGITUDE* have been added to the *crossroad* table.
- The original foreign key relationship between the *property* and *street* tables has been moved to the new cross-reference table *property_street*.
- The data type of the *DATECREATED* column was updated to *DATETIME*.
- The size of the *NAME* column in the *street* table was reduced to 40.

4 Migration Scripts

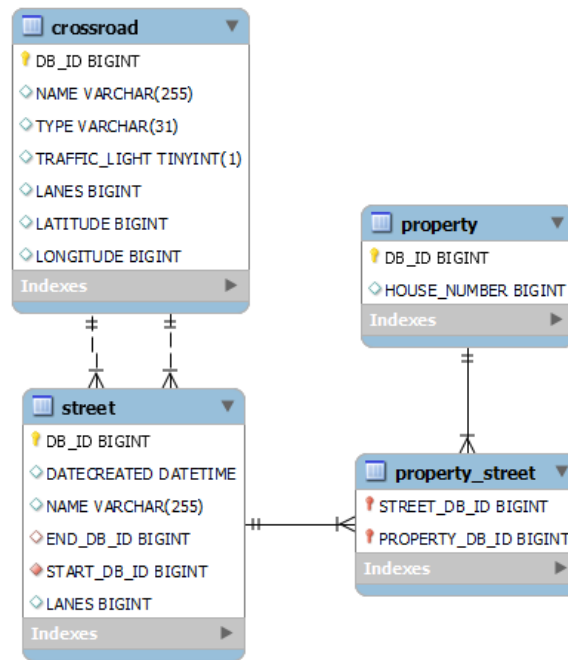


Figure 39: Das migrierte Datenbankschema

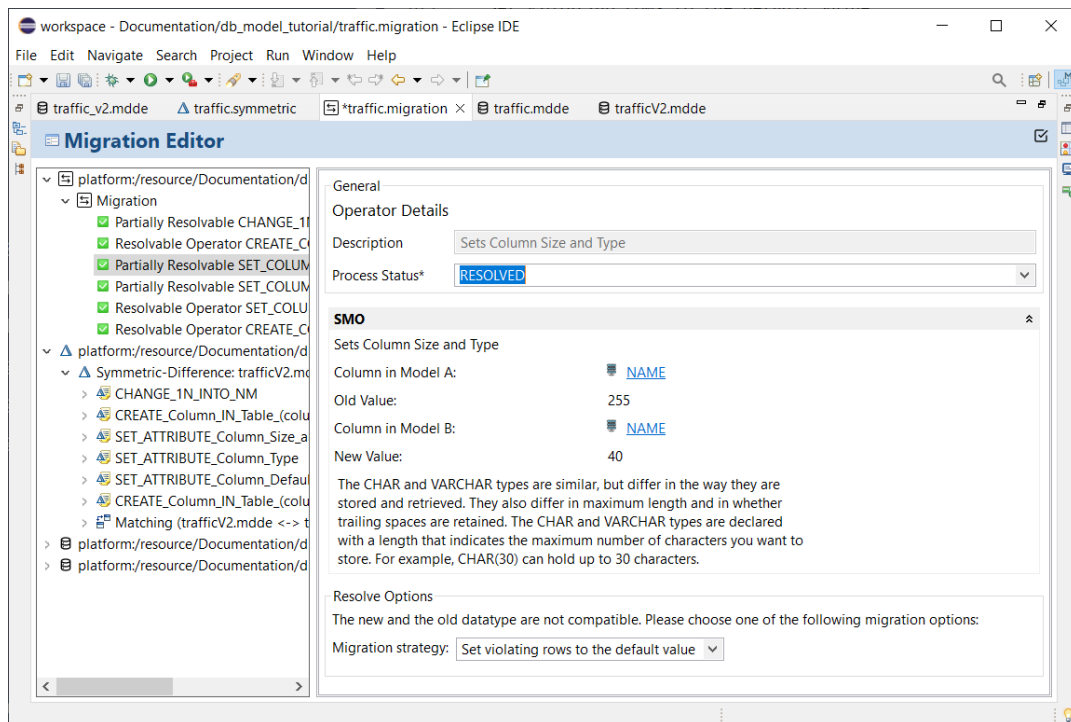


Figure 40: Migrationsstrategie mit History Tabelle

4.1 History Table

As mentioned earlier, EvolveDB provides default migration options to handle data transformations effectively. For instance, in our running example, we migrated data from the old foreign key to the newly created table. In cases such as the one described in Section 3.4.2, where a migration strategy must be selected to handle breaking changes, EvolveDB includes safeguards to prevent data loss. Specifically, EvolveDB creates a history table to store any values overwritten by the chosen migration strategy. In our example, we opted to set all violating values to the default value 'unnamed road' (Figure 40). These overwritten values were inserted into the history table, ensuring no data was permanently lost during the migration.

```

-- Creates an history table for deleted and updated values
CREATE TABLE IF NOT EXISTS 'marburg_2020_models3'.'
    mdde_history' (
        'DB_ID' BIGINT NOT NULL AUTO_INCREMENT,
        'TABLENAME' VARCHAR(255) NOT NULL,
        'COLUMN_DB_ID' BIGINT NOT NULL,
        'COLUMN_NAME' VARCHAR(255) NOT NULL,
        'VALUE' BLOB NULL,
        'CHANGEDATE' DATETIME NOT NULL,
        PRIMARY KEY ('DB_ID'));

-- Create temporary table
SET @sql_mode = @@SESSION.sql_mode;
set @@SESSION.sql_mode = '';
DROP TEMPORARY TABLE IF EXISTS my_temp_id_table;
CREATE TEMPORARY TABLE my_temp_id_table
    SELECT DB_ID from street v where LENGTH('NAME') > 40;
set @@SESSION.sql_mode = @sql_mode;

-- Find violating rows
BEGIN;
SET @safe_mode = @@SQL_SAFE_UPDATES;
SET SQL_SAFE_UPDATES = 0;
INSERT INTO 'marburg_2020_models3'.'mdde_history'
('TABLENAME',
'COLUMN_DB_ID',
'COLUMN_NAME',
'VALUE',
'CHANGEDATE')
SELECT 'street', DB_ID, 'NAME', NAME, now() from street s
    where LENGTH('NAME') > 40;

-- Set violating rows to the default value

```

```

UPDATE 'street' SET 'NAME' = 'unnamed_road' where DB_ID in ( 33
    Select DB_ID from my_temp_id_table);
SET SQL_SAFE_UPDATES = @safe_mode; 34
COMMIT; 35
36
-- If executing the script fails, we suggest a rollback. 37
DROP TEMPORARY TABLE IF EXISTS my_temp_id_table; 38
39
-- Change column type and size of name 40
ALTER TABLE 'street' CHANGE COLUMN 'NAME' 'NAME' VARCHAR(40) 41
    NULL DEFAULT 'unnamed_road';

```

Listing 2: Generated SQL code

The history table (see Figure 41) is created during the migration process, between lines three and twelve of the script. This table is used to store any values that violate the new constraints and are overwritten during the migration. After the migration, all values in the street table exceeding the 40-character limit are transferred to the history table. Table 2 displays the content of the history table after the migration. In this case, one value exceeded the 40-character limit and was replaced in the street table with the default value unnamed road. Table 3 shows the content of the street table after the migration, reflecting the applied changes.

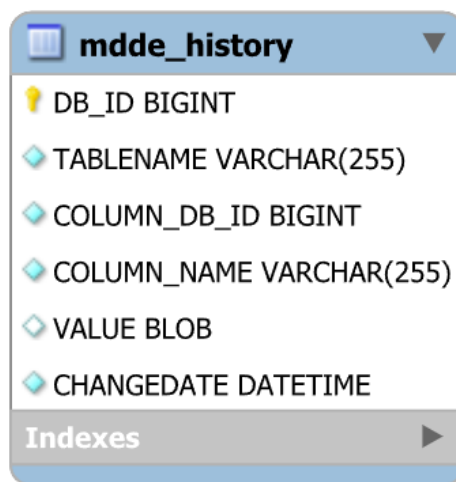


Figure 41: History table

DB_ID	TABLENAME	COL..._ID	COL..._NAME	VALUE	CHANGEDATE
1	street	2	NAME	BLOB	2022-05-20 15:41:57

Table 2: History table after executing the migration script.

4 Migration Scripts

DB_ID	DATECREATED	NAME	END_DB_ID	START_DB_ID	LANES
1	1976-05-20 15:41:57	Bahnhofstrasse	2	1	2
2	2019-08-12 13:25:57	unnamed road	2	3	3
3	1999-10-02 17:05:41	Seltersweg	3	1	1

Table 3: Street table after executing the migration script.