FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Michal Vavrek

Evolution Management in NoSQL Document Databases

Department of Software Engineering

Supervisor of the master thesis:	Doc. RNDr. Irena Holubová, Ph.D.
Study programme:	Software Systems
Study branch:	Software Engineering

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague on May 11, 2018 Michal Vavrek

Title: Evolution Management in NoSQL Document Databases

Author: Michal Vavrek

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: NoSQL databases are widely used for many applications as a technology for data storage, and their usage and popularity rises. The first aim of the thesis is to research the existing approaches and technologies for schema evolution in NoSQL databases. Next, we introduce an approach for schema evolution in multi-model databases with a unified interface for the most common data models. The proposed approach is easy to use and covers the common migration scenarios. We have also implemented a prototype, optimized its read/write operations, and demonstrated its properties on real-world data.

Keywords: NoSQL, Evolution Management, Multi-model Databases

First of all I would like to express my gratitude to my supervisor Doc. RNDr. Irena Holubová, Ph.D. for her helpful advice, guidance, corrections and prompt responses, whenever I needed. Then I would like to thank Prof. Dr. Stefanie Scherzinger for helpful comments and advice. Last but not least, I would like to thank to my parents František and Hana, my girlfriend Anna, brother Jan, all the other family members and friends for their support, encouragement and especially for patience.

Contents

1. Introduction

Today, NoSQL databases [54] are widely used for many applications as a technology for data storage, and their popularity rises. NoSQL databases can be divided into multiple categories based on the provided data model: a key/value, a document, a graph, and a column-family. Most of NoSQL databases are schema-less. That means, we can store any structure of data in the database. In the same collection, we can have entities with various sets of properties. This fact gives developers freedom to add or remove properties on the fly without changes of the database schema. The database can freely evolve with new requirements of the application. The problem is, that it is not exactly true. The schema is always given by the application, because it has to be able to retrieve data and correctly load them to the application memory. As the application evolves the schema of persisted entities is evolving together with it. To keep the application and the database in sync we need a way to migrate persisted data.

With the increasing popularity of NoSQL databases, the need for schema evolution management tools rises, because manual migrations could be hard to manage by developers. It requires knowledge of the database, its query language, and when the database is scaled, then also its architecture, and the script itself. The script is created by developers so it has to be properly tested to avoid mistakes and ensure that all needed entities will be correctly migrated. The topic of the schema evolution is well-known mainly in the context of SQL databases. There are complex solutions for schema migration which in some cases include even query migration [58].

The original aim of this thesis was a research of existing approaches and technologies for schema evolution in NoSQL databases, and introducing a general approach for it. Nowadays, most of the databases (not only NoSQL) are extended to multi-model databases (MMDs). This fact is also predicted in Gartner's research [20]. They state that most of the modern databases will become multi-model and the schema evolution in MMD is an unresolved problem. In our research, we have not found any general approach for them so we have decided to focus the thesis on schema evolution in MMDs.

Introducing a general approach for schema evolution in MMDs which could be implemented in any MMD, and would be easy to work with, and which would yet cover common migration scenarios is very hard, if not impossible. To reach our goal, we decided to limit our focus on core migrations which are more common than complex ones and exclude query migration. Restrictions keep an introducing approach enough powerful to cover common migrations and it forms the basis for future works.

We introduce a language which provides a set of core migration operations that covers most common scenarios. The set allows developers to migrate data between entities from different models in the same way as between entities from the same model without any limitations. To ensure it, the operations have to be implementable in any data model of the MMD. It means, that the language provides a unified interface for all models which is simple enough to be implementable, but provides enough power to ensure all core migration operations.

1.1 Structure of the Thesis

The thesis is structured as follows.

- Chapter 1 introduces the goal of the thesis and describes its structure.
- Chapter 2 defines basic terms, which are used in the rest of the thesis. It also briefly introduces the XML, JSON, NoSQL databases and schema evolution.
- Chapter 3 compares existing approaches to schema evolution in NoSQL databases.
- Chapter 4 introduces multi-model databases, their structure, sample usage and multiple examples of real-world projects.
- Chapter 5 contains the main part of the thesis. In this chapter we describe conclusions of the existing research and propose our solution.
- Chapter 6 describes important details of the implementation of the prototype and used technologies.
- Chapter 7 contains the description of the experiments we executed, the results and their discussion.
- Chapter 8 summarizes the work and proposes suggestions for future work.

2. Definitions, Terms and Used Technologies

This chapter contains descriptions of the most important terms and constructs which we use throughout the thesis.

2.1 XML

XML (Extensible Markup Language, [67]) is a markup language designed for data storage and exchange. It is a markup language which means that XML documents are basically plain text documents augmented by certain marks that provide the structure and indicate the meaning of individual parts. XML defines rules for encoding documents in human and machine-readable format.

Figure 2.1 shows and example of an XML document which contains a tea record encoded as an XML document. Such a tea record has a root element *tea* with attribute with name *id*. The element *name* has a text content 'Silver Needle'.

2.2 JSON and BSON

JSON (JavaScript Object Notation, [31]) is a lightweight data-interchange format. It is easy for humans to read and write and it is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language [17].

JSON is built on two structures: a collection of key/value pairs and an ordered list of values (i.e. *array*). These structures are universal and supported in almost all modern programming languages. JSON supports the following set of basic types: *Number*, *String*, *Boolean*, *Array*, *Object* and *null*. Figure 2.2 shows a JSON document which consists of several keys. The field *id* has a value which

```
<?xml version="1.0" ?>
<tea id="0">
        <name>Silver Needle</name>
        <type>white</type>
        <price>10</price>
        <country>China</country>
        <!-- Comment: other items were omitted -->
</tea>
```

Figure 2.1: XML Example

```
{
  "tea": {
    "id": 0,
    "name": "Silver Needle",
    "type": "white",
    "suppliers": ["Oxalis", "Ahmad"],
    "description": {
          "preparation": "70-75",
      "color": "yellow-green"
    }
  }
}
```

Figure 2.2: JSON Example

is of type *Number*, the value of the field *suppliers* has type *Object*, the type of *description* is *Array* and the field *type* is of type *String*.

BSON (Binary JSON, [5]) is a binary format in which zero or more ordered key/value pairs are stored as a single entity. In general BSON is a binary-encoded serialization of JSON-like documents. BSON is designed to have a minimum spatial overhead which is useful for network transfers. However, the BSON representation can sometimes be bigger than the corresponding JSON file because of additional information for data traversability. BSON contains additional data like length of strings for faster traversal. It is designed to be fast to decode and encode.

2.3 NoSQL Databases

Traditional technologies like relational database management systems are usually not able to handle the size and velocity of modern applications which resulted in new technologies for batch processing, scaling, distributing and interactive processing of data. NoSQL is commonly translated as 'not only SQL' which are next generation databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable [54]. The horizontal scalability means adding/removing nodes to a distributed database system.

NoSQL databases provide mechanisms for storage and retrieval of data that is modeled in other ways than traditional relations. There is no single formal definition and the definitions vary. The original intention was modern web-scale databases. The movement began in early 2009 and is growing rapidly. Often more characteristics [54] apply such as: schema-free, easy replication support, simple API, eventually consistent – not ACID (atomicity, consistency, isolation, durability), a huge amount of data, fault-tolerant architecture and more.

There are several data models used in NoSQL systems. They differ in key/-value, document, column-family and graph databases. The first three models are oriented on aggregates. An aggregate is a unit of related data with a complex structure. We usually want to treat this data as one unit. Relational database management systems and graph databases have no concept of aggregates within their data model. These are considered aggregate ignorant. An aggregate ignorant model allows us to look at data in different ways so it is good when there is not a primary structure for manipulating data.

In general, NoSQL systems follow the definition of our data model. The graph model is questionable but later in the thesis, we will show that is can fit the definition too. All of them are able to store entities which consist of properties with values.

In the following sections we briefly describe each category of NoSQL database type.

2.3.1 Key/value Databases

This is the simplest type of NoSQL databases. It provides an associative array (a dictionary) addressed by a primary key. We would model it with two columns, e.g. *id* and *value* in a relational database. *Id* column stores the keys and the *value* contains the associated values. The structure of value can vary in different key/value databases. Most of them support structured values like ,e.g., arrays. Key/value databases usually support only basic operations: getting the value for the specified key, inserting the value for a key and deletion of a key/value pair. The simplicity provides great performance and great scalability, however it does not support complex queries.

The two most commonly used key/value databases [15] are Redis [60] and MemcachedDB [40]. Redis is an open-source in-memory key/value database with optional durability which supports different kinds of abstract data structures, such as string, arrays, maps, etc. MemcachedDB is open-source in-memory key/value database built on top of non-persistent memcached technology. MemcachedDB adds a persistent functionality. It provides a large distributed hash table across multiple machines and it is designed to store small chunks of arbitrary data (strings, objects). The difference between them is in complexity and provided functions for data manipulation. Parts of Redis were built in response to lessons learned from using MemcachedDB. Redis has more features than MemcachedDB and is, thus, more powerful and flexible.

This type of databases is commonly used for [61] storing session information of web applications, user profiles and preferences, shopping cart data etc. It is fast, as it works with a single object that contains all relevant information.

2.3.2 Document Databases

The main concept of document databases are documents which are stored, retrieved and managed. Documents are usually in JSON or XML format and they have hierarchical tree structure self-describing the contents. The databases are usually schema-less so the schema is defined by an application and can differ. Document databases contrast strongly with relational databases which are strongly typed during database creation and thus every instance of the data has the same format and it is often difficult to change the schema. Document databases get the type information from the data itself. They store the related data together and allow instances of the data to be different. This property makes it easier and more flexible to deal with changes. This fact also optimizes the space management, because each instance of data contains only required data. In case of relational databases, to add a new information to one instance of data we would have to add a new column with a default value to all instances and set the required value to the one instance.

The two most used document databases [15] are MongoDB [42] and Couchbase [10]. MongoDB is a open-source JSON-like document database with a focus on scalability and flexibility. It supports data indexing, queries and real time data aggregation. Couchbase (originally called Membase) is an open-source distributed JSON-like document database. It is designed to be scaled from a single machine to large-scale deployments in many machines. Couchbase also provides compatibility with memcached technology, so it can be used as a key/value database too.

Document databases are useful for [61] event logging, content management systems, blogging platforms, e-commerce applications, real-time analytics and more.

2.3.3 Column-Family Databases

This type of NoSQL databases is column-oriented. A column family is a collection of similar rows. Each row is a collection of columns associated with a key, i.e. it is a tuple (key/value pair), where the key is mapped to a set of columns. Column is a basic unit which consists of a triple: name-value pair and a timestamp. Data in a single column family are related and they are mostly accessed together by an application. In relational databases, a column family would be a table, each key/value pair being a row. In relational databases, each row must have the same columns whereas in column-family databases, this is not a necessary condition.

The two most commonly used column-family databases [15] are Cassandra [7] and HBase [25]. Cassandra is a scalable high-performing column-family database providing high availability without a single point of failure. Cassandra supports SQL-like query languege: Cassandra Query Language [8](CQL) which adds abstraction layer that hides the native syntax. HBase is optimized for Hadoop [2]

(a) Data in a relational database.

id	**name**	*price*	**type**
0	Silver Needle	10	white
1	Longjing	15	green

(b) Representation of data in a relational database.

0;Silver Needle;10;white
1;Longjing;15;green

(c) Representation of data in a column-family database.

0;1
Silver Needle;Longjing
10;14
white;green

Figure 2.3: Second meaning of column-family databases definition.

operations and random, realtime read/write access to data. It provides a fault-tolerant way of storing large quantities of sparse data (small amounts of information caught within a large collection of empty or unimportant data).

Column-family databases are suitable for [61] event logging, content management systems, blogging platforms etc.

We use the mentioned definition of the column-family database from the NoSQL world we are targeting but we have to mention also a second meaning of the column-family databases: The column-family databases store data in a column form instead of in rows as relational databases. Figure 2.3 shows an example of representing a table in this kind of column-family databases. This approach has a better performance for aggregation functions which are executed on top of multiple columns. Another benefit of this representation is if we need to change a value of a column for all rows.

2.3.4 Graph Databases

Graph databases store entities and relationships between them. Basic concepts are nodes and edges. Nodes can have properties (like, e.g., name) and edges have types (like, e.g., 'is ancestor of'). In a relational database we can model a single type of relationship whereas adding a new type of relationship requires changes in the existing schema. Nodes in graph databases can have many different types of relationships. The number of these relationships is not limited and relationships can be easily added or removed.

The two most commonly used graph databases [15] are Neo4j [52] and Giraph [21]. Neo4j is a graph database which provides ACID-compliant (Atomicity, Consistency, Isolation, Durability) transactions with native graph storage. It uses the Cypher Query Language [12] which is a declarative query language designed for expressive and efficient querying and updating of a property graph. Giraph is designed for processing of graphs on big data. It uses the MapReduce programming model [33] to data processing.

Graph databases are suitable for [61] connected data like social networks. Another area where this type of database is useful are routing, dispatch and location-based services.

2.3.5 Data Model and Query Model

For purpose of the thesis, we use a definition of a data model and a query model introduced in [63].

Data Model

The *data model* stores objects which are called *entities*. Each entity belongs to a *kind*, which is a group of semantically similar objects. Each entity has a unique key and value. The key is a tuple of its *kind* and *identifier* and value is a list of properties. Each entity property consists of a *name* and a *value*. The list of properties is an unordered set of properties with a unique name. The properties may be scalar, array of values or consist of nested entities.

For better understanding, we use the following representation for entities of the data model:

$(kind, id) = \{\text{list of properties}\}$

Query Model

We can manipulate entities based on their key and we can query all entities of a particular kind. We assume that a data store supports conjunctive queries with equality comparisons. This functionality is commonly provided by relational stores, document stores and column-family stores alike.

3. Existing Approaches and Technologies

This chapter presents an overview of existing approaches and technologies to managing the evolution of NoSQL databases. We are interested in recommended/best practices, scientific studies and existing solution for data evolution of NoSQL databases.

3.1 Basic Approach

The most popular and widely used way how to manage data evolution is coding and managing a set of transforming/migrating scripts. The developer has to create each script manually and apply it on each environment separately based on application version and his knowledge about the application.

This approach requires a good knowledge of the application, its architecture, and target NoSQL database, i.e., the a syntax of the used language, the database cluster topology, a possible database downtime, etc. These factors are potential sources of complications. For example, a script can contain errors or can convert data to an invalid format. That is the reason why this problem began to be studied.

One of the first references to schema evolution in NoSQL databases is presented in [68]. The problem is simplified to storing new versions of a record and ignores the needs for managing changes. The book states that NoSQL document databases support schema-less structures and thus accommodate flexible and continuing evolution. It presents the problem on CouchDB [1], where we can use default field *_rev* with a hash of revision to map versions from the database to the application.

Generally, we can apply the recommended *version* field for storing a version of a record. Each record can be extended by its version number that can be queried. The migration script can be created safer if it uses the field, because migration will be applied only on records with a specific version field. Unfortunately, the rest of the mentioned negative aspects like a good knowledge of the application and specific database remain.

3.2 NoSQL Schema Evolution in Standard ORM

Many standard ORMs (Object-Relational-Mappings) provide some basic functionality for schema evolution. We will introduce one of the most popular ORMs and explain their features for schema evolution. Sometimes ORMs for document databases are called Object-Document-Mappers (ODM) or Object-NoSQL Mappers, so we will make no difference between them.

3.2.1 Objectify

Objectify [55] is a Java data access API specifically designed for the Google Cloud Datastore [23]. The usage of Objectify is based on annotated Java classes and the database access point object `ObjectifyService.ofy`. Persisting classes have to be annotated as `@Entity` and contain property `@Id` as follows:

```java
@Entity
public class Person {
    @Id Long id;
    String name;
    String City;
}
```

The `ObjectifyService.ofy` service provides standard ORM functions like save, load or delete. The following example shows a few basic operations with entity `Person`:

```java
import static com.googlecode.objectify.ObjectifyService.ofy;

// Creates and saves a new entity
Person person = new Person("Michal", "Prague");
ofy().save().entity(person).now();

assert person.id != null;    // id was generated automatically

// Loads the entity
Person person1 = ofy().load().type(Person.class).id(person.id).now();

// Modifies it
person1.City = "Tabor";
ofy().save().entity(person1).now();

// Deletes it
ofy().delete().entity(person1).now();
```

Objectify has constructions for managing schema evolution. They allow a programmer to create scripts for migrating entities via annotated migration functions or it migrates entities at run-time. Bellow, we describe possible migrations and relevant operations when an entity is loaded and saved:

- **Adding or removing fields** – Fields can be added/removed to/from a class without any other code. The added field will be left with its default value when the class is initialized. For the removed field, the data in the database

```
@Entity
public class AppUser {
    @Id Long id;
    @AlsoLoad("phone")
    String phoneNumber;
    int birthDay;
    int birthMonth;
    int birthYear;

    void import(@AlsoLoad("birthdate") String birthdate) {
        // Date format dd-MM-YYYY
        String[] birthdateArray = birthdate.split(" ");
        birthDay = Integer.parseInt(birthdateArray[0]);
        birthMonth = Integer.parseInt(birthdateArray[1]);
        birthYear = Integer.parseInt(birthdateArray[2]);
    }
}
```

Figure 3.1: Example of data transformation for *AppUser* entity in Objectify.

will be ignored when the class is initialized and persisted then the entity will be saved without the field.

- **Renaming fields** – Fields can be also renamed by annotation `@AlsoLoad`. Objectify loads an entity from the database and the old renamed property is loaded to the annotated property during class initialization. For example, in Figure 3.1 class `AppUser` has renamed the field *phone* to *phoneNumber*. If both fields exist, Objectify throws an exception. When the class is saved, only *phoneNumber* is overwritten.

  A possible problem can occur with queries, because they do not know about the renaming. If we filter data by *phoneNumber*, we get only the migrated entities and if we filter by *phone*, we get only the old ones.

- **Transforming data** – Objectify also allows developers to transform the data through the migration function. The function must contain a single parameter with annotation `@AlsoLoad`. For example, Figure 3.1 shows how we can perform transformation of field *birthDate* of class `AppUser`. When the entity is saved again, it will have only properties *birthDay*, *birthMonth* and *birthYear*.

  If *birthDay*, *birthMonth*, *birthYear* and *birthDate* exists in the database, the results of migration are undefined.

- **Moving fields** – In the same way as data transformation above we can move

```
@Entity
public class Birthdate {
    @Id Long id;
    int birthDay;
    int birthMonth;
    int birthYear;
}

@Entity
public class AppUser {
    @Id Long id;

    @IgnoreSave int birthDay;
    @IgnoreSave int birthMonth;
    @IgnoreSave int birthYear;

    Key<Birthdate> userBirthDate

    @OnLoad void onLoad() {
        if (birthDay != null || birthMonth != null || birthYear !=
            null) {
             this.userBirthDate = ofy().save().entity(new Birthdate(
                birthDay, birthMonth, birthYear)).now();
             ofy().save().entity(this);
        }
    }
}
```
Figure 3.2: Example of moving fields for *AppUser* entity in Objectify.

a field from one class to another. For example, in Figure 3.2 birthdate fields are moved to a separate class `Birthdate`.

In general, Objectify provides a functionality for data manipulation:

- `@AlsoLoad` – loads a field with a given name and allows the developer to transform the data in methods.
- `@Ignore` – allows the developer to use fields that are not loaded or saved to the database.
- `@IgnoreLoad` – allows the developer to have save-only fields.
- `@IgnoreSave` – allows the developer to load data without saving them again.
- `@OnLoad` – allows the developer to execute an arbitrary code after all fields have been loaded.
- `@OnSave` – allows the developer to execute an arbitrary code before an entity is written to the database.

As shown above, Objectify provides powerful tools for schema evolution. The developer can add, remove, rename, transform or move properties as he wants. However, the developer still must write migration functions, manage changes manually and mainly cannot create a migration between more than two versions of an entity.

3.2.2 Morphia

Morphia [48] is a lightweight type-safe library for mapping Java objects to/from MongoDB. Morphia also uses annotations of Java classes and provides the same functions for data evolution management like Objectify. Morphia does not have more powerful functions than Objectify so we will not describe them again.

3.2.3 Hibernate OGM

Hibernate Object/Grid Mapper (OGM) [26] provides Java Persistence (JPA) support for NoSQL databases. It uses a standard ORM engine and persists entities in a NoSQL datastore instead of a relational database. OGM supports the following databases via database-specific dialects:

- Key/Value: Infinispan [29] and Ehcache [18]
- Document: MongoDB

```
using System.Collections.Generic;
using MongoRepository;
using MongoDB.Bson.Serialization.Attributes;

public class Customer: Entity  //Inherit from Entity!
{
    public string FirstName { get; set; }

    [BsonIgnore]
    public string LastName { get; set; }    //Ignore lastname

    [BsonElement("sx", Order = 1)]
    public string Sex { get; set; }         //Load sx to property Sex

    [BsonIgnoreIfNull]
    public List<Product> Products { get; set; } //Ignore when there
        are no items in list
}
```

Figure 3.3: Example of a MongoRepository entity object.

- Graph: Neo4j

And a few more are in progress, namely Redis, CouchDB, and Cassandra.

Hibernate OGM supports several ways for querying and returning Hibernate managed objects, namely:

- JP-QL (Java Persistence Query Language) which is converted into native backend queries,
- database native queries,
- full-text queries, using Hibernate Search (transparent indexer for fast full-text geolocation search) as index engine.

OGM provides the same annotations for data evolution management as Morphia and Objectify (i.e., `@IgnoreSave`, `@IgnoreLoad`, and `@OnLoad`).

3.2.4 MongoRepository

MongoRepository [47] is an implementation of a Repository pattern [41] on top of the official MongoDB C# driver [44]. The repository pattern separates the logic that retrieves the data and maps it to the model. The repository pattern wraps the functionality of ODM.

```
public class CustomerRepository : MongoRepository<Customer> {

    public List<Customer> GetCustomersWithOpenInvoices() {
        //...
    }

    public Customer GetCustomerWithLongestName() {
        //...
    }
}
```

Figure 3.4: Example of a MongoRepository repository object.

MongoRepository works with two types of classes. The first are `Entity` classes (for an example see Figure 3.3) which represent a database entity and the second one are `Repository` classes (for an example see Figure 3.4). The developer can access database entities through the prepared repository. The mentioned examples show a `Customer` entity and its repository.

MongoRepository also provides a few tools for schema evolution. In particular, we can add one of the following serialization attributes to class properties:

- `BsonIgnoreAttribute` – The property will not be initialized when the entity is loaded.
- `BsonIgnoreIfNullAttribute` – The property will not be initialized during the entity loading if the value is *null*.
- `BsonIgnoreExtraElementsAttribute` – If the entity contains extra properties which are not presented in class definition, the mapper will ignore them.
- `BsonElement` – Loads the defined entity to the property. The developer can use this functionality to rename entities.

Finally, if there is no way to use the attributes, the developer can code his own map function and define mapping rules. For example, the mapping function for loading the entity *sx* to property *Sex* from Figure 3.3:

```
BsonClassMap.RegisterClassMap<Customer>(cm => {
    cm.AutoMap();
    cm.GetMemberMap(c => c.Sex).SetElementName("sx").SetOrder(1);
});
```

The provided functions are not as powerful as Objectify functions. For example, we cannot easily specify rename operation because MongoRepository does not

ODM	Google Datastore	MongoDB	Redis	Neo4j	CouchDB	Cassandra
Objectify (Java)	✓	✗	✗	✗	✗	✗
Morphia (Java)	✗	✓	✗	✗	✗	✗
Hibernate OGM (Java)	✗	✓	✓	✓⋆	✓⋆	✓⋆
MongoRepository (C#)	✗	✓	✗	✗	✗	✗
Mongoid (Ruby)	✗	✓	✗	✗	✗	✗

⋆ **Currently under development**.

Table 3.1: Supported databases for ODMs.

provide the functionality `@AlsoLoad` as Objectify. The developer can still create migration scripts as we showed before and manage simple schema evolution.

3.2.5 Mongoid Evolver

Evolver [46] is a database schema evolution tool for Mongoid [45] which is a Ruby ODM framework for MongoDB. Project Evolver is focused on schema evolution in MongoDB, but unfortunately has no commit since 19th January 2013 or any other activity. The current documentation is very poor and its author did not reply to our questions about Evolver functionalities. So we cannot analyze Evolver in more detail.

To sum up, we prepared summary tables where we can see basic information about all mentioned ODMs. In Table 3.1 we can see all supported databases for each described ODM and its programming language. Next Table 3.2 shows a list of functionalities for schema evolution for the ODMs.

3.3 Managing Schema Evolution in NoSQL Data Stores

In [63], S. Scherzinger, M. Klettke and U. Störl introduce a cornerstone of many other studies – the general language for schema evolution in NoSQL databases and the **NoSQL Database Programming Language** (NoSQLDPL). They also

ODM	Also load	Ignore	Ignore load	Ignore save	On load	On save
Objectify	✓	✓	✓	✓	✓	✓
Morphia	✓	✓	✓	✓	✓	✓
Hibernate OGM	✓	✓	✓	✓	✓	✓
MongoRepository	✗	✓	✗	✗	✗	✗
Mongoid	?	?	?	?	?	?

Table 3.2: Schema evolution function for ODMs.

classify schema evolution into two strategies **eager** and **lazy**. Specifically, the work is focused on document and column databases.

3.3.1 NoSQL Schema Evolution Language

The language is based on the fact that NoSQL databases are schema-less, so there is no explicit schema. The structure is defined by individual application classes interpreted by object mappers. Mappers commonly map class names to entity kinds and class members to entity properties. Thus, each application defines somewhat consistently structured data without a fixed schema. In an agile setting, there are applications, which evolve rapidly their features, their classes and also their data. Under these assumptions, the authors define a set of declarative schema migration operations. The set covers a large share of the common schema evolution operations.

Figure 3.5 shows the syntax of the **NoSQL Schema Evolution Language** (NoSQLSEL) in Extended Backus-Naur Form (EBNF). As we can see, the operations can contain conditionals and even joins for specific data selection. This grammar contains two terminals: the property kinds (derived from **kname**) and the property names (**pname**). The language also uses property *version* (mentioned in Section 3.1) for detection of the current version of an entity.

Below, we show the evolution operations using a simple example. The language provides three basic operations which allow us to manage heterogeneous entities of the same kind:

- **Operation *add*** – Adds a property to all entities of a given kind. For example, operation **add person.visit = 0** adds a visit-counter *visit* to all entities *person* initialized to 0.

```
evolutionop ::= add | delete | rename | move | copy;

add ::= "add" property "=" value [selection];
delete ::= "delete" property [selection];
rename ::= "rename" property "to" pname [selection];
move ::= "move" property "to" kname [complexcond];
copy ::= "copy" property "to" kname [complexcond];

selection ::= "where" conds;
complexcond ::= "where" (joincond | conds | (joincond "and" conds));

joincond ::= property "=" property;
conds ::= cond {"and" cond};
cond ::= property "=" value;

property ::= kname "." pname;
kname ::= identifier;
pname ::= identifier;
```

Figure 3.5: Syntax of the NoSQLSEL in EBNF.

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*1*

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
visit	0
version	*2*

- **Operation *delete*** – Removes a property from all entities of a given kind. For example, operation **delete person.phone** deletes property *phone* from all entities *person*.

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
phone	00420777000000
version	*1*

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*2*

- **Operation *rename*** – Changes the name of the property for all entities of a given kind. For example, operation **rename person.mail to e-mail** renames property *mail* to *e-mail* for all entities *person*.

key	**(person, 42)**
name	John Doe
mail	john@doe.mail
version	*1*

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*2*

And there are also two dedicated functions for schema refactoring that affect two kinds of entities. These operations assume 1:N relationship between the affected entities. In case of N:M relationship the result of them is not defined. The problem of N:M relationship is that it cannot be defined as a safe operation. If there is an N:M relationship specified as a cross product then the execution order influences the migration result so the result is not predictable before execution of the operation.

The operations are:

- **Operation *move*** – Moves a property from one entity kind to another entity kind. For example, operation **move person.e-mail to comment where user.name = comment.owner** moves property *e-mail* from persons to all their comments.

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*1*

key	**(person, 42)**
name	John Doe
version	*2*

key	**(comment, 1991)**
topic	Charles University ...
text	It is the ...
owner	John Doe
version	*1*

key	**(comment, 1991)**
topic	Charles University ...
text	It is the ...
owner	John Doe
e-mail	john@doe.mail
version	*2*

- **Operation *copy*** – Copies a property from one entity kind to another entity kind. For example, operation **copy person.e-mail to comment where user.name = comment.owner** copies property *e-mail* from persons to all their comments.

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*1*

key	**(person, 42)**
name	John Doe
e-mail	john@doe.mail
version	*2*

key	**(comment, 1991)**
topic	Charles University ...
text	It is the ...
owner	John Doe
version	*1*

key	**(comment, 1991)**
topic	Charles University ...
text	It is the ...
owner	John Doe
e-mail	john@doe.mail
version	*2*

3.3.2 NoSQL Database Programming Language

Based on the NoSQLSEL the authors develop a generic NoSQLDPL. The language defines the typical operations on entities in NoSQL databases, and it is particularly modeled on the basis of the interfaces to the Google Datastore [23] and applies to document and column databases. We will not describe the language details. We will just introduce the main function.

Figure 3.6 defines language functions. In particular, Rule 3.1 creates a new entity with key κ. Initially, an entity does not have any properties. To set properties, we use Rule 3.2. The next Rule 3.3 adds a new property n with value v to the entity with key κ. Rule 3.4 adds a nested property as a property. Conversely, Rule 3.5 removes property n from the entity with key κ. Rule 3.6 persists the entity with key κ and replicates this entity to the database. Persisting of an entity replaces any other entity with the same name. Rule 3.7 deletes the entity with key κ from the database. Rule 3.8 retrieves an entity by its key from the database. Rule 3.9 retrieves all entities from the database that are of the specific kind c. We can also query with predicate $\emptyset$, as described by Rule 3.10.

3.3.3 NoSQL Schema Evolution Strategies

Next, the work introduces two main categories of NoSQL schema evolution strategies called eager migration and lazy migration.

The difference between the migrations is in the time when the migration is executed and the scope of affected entities.

Eager Migration

The eager migration is a batch job which migrates all affected entities. In such batch jobs, entities are fetched one-by-one from the database into an application where they are modified (migrated) and then written back to the database. The problem of the eager migration is when migrations happen while the application is in use and the data could be changed during the evaluation. Or the application should be suspended.

$$
\begin{aligned}
[\![new(\kappa)]\!](ds, as) &= (ds, as[\kappa \mapsto \emptyset]) && (3.1)\\
[\![new(\kappa, \pi)]\!](ds, as) &= (ds, as[\kappa \mapsto \pi]) && (3.2)\\
[\![setProperty(\kappa, n, v)]\!](ds, as \cup \{\kappa \mapsto \pi\}) &= (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto v])\}) && (3.3)\\
[\![setProperty(\kappa, n, \kappa')]\!](ds, as \cup \{\kappa \mapsto \pi\} \cup &\\
\{\kappa' \mapsto \pi'\}) &= (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \pi'])\} && (3.4)\\
&\quad \cup \{\kappa' \mapsto \pi'\})\\
[\![removeProperty(\kappa, n)]\!](ds, as \cup \{\kappa \mapsto \pi\}) &= (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \bot])\}) && (3.5)\\
[\![put(\kappa)]\!](ds, as \cup \{\kappa \mapsto \pi\}) &= (ds[\kappa \mapsto \pi], as \cup \{\kappa \mapsto \pi\}) && (3.6)\\
[\![delete(\kappa)]\!](ds, as) &= (ds[\kappa \mapsto \bot], as) && (3.7)\\
[\![get(\kappa)]\!](ds \cup \{\kappa \mapsto \pi\}, as) &= (ds \cup \{\kappa \mapsto \pi\}, as \cup [\kappa \mapsto \pi]) && (3.8)\\
[\![get(kind = c)]\!](ds, as) &= (ds, as[\{\kappa \mapsto \pi\\
&\quad \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c\}]) && (3.9)\\
[\![get(kind = c \wedge \emptyset)]\!](ds, as) &= (ds, as[\{\kappa \mapsto \pi\\
&\quad \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c\\
&\quad \wedge [\![\emptyset]\!](\kappa \mapsto \pi)\}]) && (3.10)
\end{aligned}
$$

Figure 3.6: NoSQL Database Programming Language [63]

```
public class Person {
        @Id Long id;
        @AlsoLoad("name") String fullName;
}
```

Figure 3.7: The Java class which represents entity *Person*.

Scripts written in the language migrates all entities of given kind κ with specified version v to the version $v + 1$ at once as a batch job.

The version property added to all entities makes the migration robust in case of interruptions. NoSQL data stores commonly offer very limited or none transaction support. So a large-scale migration cannot be performed as an atomic action. By restricting migrations to all entities of a particular version (using the where-clause), scripts may correctly recover from interrupts and migrate all remaining entities with version v.

Lazy Migration

The lazy migration allows co-existence of entities of the old and new schema. Whenever an entity is read into the application space, it can be migrated. Effectively, this will migrate only hot data that is still relevant to users. In other words, the lazy migration transforms only currently used data. However, each migration requires a custom code which should be evaluated when the entity is read. The lazy migration is a potential point of risk, because the data can by corrupted be incorrect application of migration scripts and there may be no way to undo the changes.

The NoSQLDPL can also implement common functions for lazy migration. The authors extend the language by self-explanatory constructs, such as if-statements or local variables and operation $hasProperty(\kappa, n)$ that tests whether the entity κ has a property by name n.

We will show a simple example from the paper which is adapted from Objectify documentation. Java class `Person` is mapped to database entity *person*. The entity has property *id* which is marked `@Id` and in earlier version it has property *name*, which is now renamed to *fullName*. Legacy entities do not yet have property *fullName*. When the entity is loaded into the application memory, the object mapper migrates property *name* to the *fullName*. Next time when the entity is persisted, its new version with property *fullName* will be stored. Figure 3.7 shows an implementation of class `Person`.

In the NoSQLDPL `@AlsoLoad` will be implemented as shown in Figure 3.8. The implementation checks if the entity has the property *name* and eventually sets the value of property *name* to the *fullName*. Finally, property *name* is removed.

```
Key p := ("Person"; id);
if hasProperty(p, name) do
        setProperty(p, fullName, getProperty(p, name));
        removeProperty(p, name)
od
```

Figure 3.8: The implementation of `@AlsoLoad` in the NoSQLDPL.

The authors do not explore lazy migration to a greater detail and leave it open for a future work. The current state of the language has one important limitation. It does not support nesting and unnesting of entities during migration operations.

3.4 Datalog-Based Protocol for Lazy Data Migration

In [66], S. Scherzinger, U. Störl and M. Klettke extend their work [63] (described in Section 3.3.1) about schema evolution in NoSQL databases. In the paper they propose a Datalog-based model:

- The model formalizes all structural variants of persisted entities in non-recursive Datalog with negation. The authors model a core set of migration operations for adding, removing, renaming, copying and moving properties.
- The model provides alternative evaluation strategies. Rule capturing put- and get-calls from the application are executed eagerly but rules capturing migration can be executed eagerly or lazily.
- The model leverages the rich body of work on Datalog evaluation algorithms. It also introduces a lazy migration protocol where all get-calls returns up-to-date entities. This feature allows to safely employ lazy migration in NoSQL application development, and deploy releases without downtime.

3.4.1 Datalog-Based Model

Datalog is a state-of-the-art formalism for data exchange [16]. The authors declare schema mapping and updates in Datalog and model the application accessing entities in a straightforward manner. Function $kind(id, p_1, ..., p_n)$ represents the current schema of a given *kind*. Each entity has a unique *id* and properties named p_1,...,p_n. The *kind* works like a placeholder for the name of the kind and this invariant is used in all next definitions below. For example, entity *Person* with properties *name* and *email* has *kind* function: $Person(id, name, email)$.

The next function $put(kind(id, p_1, ..., p_n))$ persists an entity and $get(kind, id)$ loads an entity having a specified identifier id.

Internally, the authors timestamp all entities and simulate a logical clock. The clock tracks the last update under the assumption that all actions are executed eagerly. The authors distinguish two kinds of Datalog rules for maintaining the state of the database. **Residual rules** model changes, such as writing an entity. A fact derived from a residual rule always holds true even in the presence of future actions. **Transient rules** compute intermittent facts required for the query predicate. These facts do not hold true in future and can be timely discarded.

The authors define a set of functions and a terminology. We will introduce them in a nutshell:

- For each action a_i an **action tuple** is defined as follows:

  $$AT(a_i) = (\Delta_i, R_i, T_i, p_i)$$

  where Δ_i is a set of Extensional Database (EDB) facts (i.e., entities put to the data store), R_i/T_i are residual/transient rules, and p_i is a query predicate which is empty for put-calls.

- A Datalog query is defined as follows:

  $$Q_n = (\Pi_n, p_n)$$

  where Π_n is a Datalog program which consists of rules $R_1 \cup ... \cup R_n \cup T_n$.

- A data instance D_n consists of all entities put to the data store, i.e., $D_n = \Delta_1 \cup ... \cup \Delta_n$

- An application of Datalog program Π_n to a data instance D_n is defined as $\Pi_n(D_n)$. It is a set of Intensional Database (IDB) facts that form logical consequences of $\Pi_n \cup D_n$.

- A function $ResDB(a)$ is a computation of residual entities after executing actions $a_1, ..., a_n$:

  $$\begin{aligned} ResDB(a_1) &= R_1(\Delta_1) \\ ResDB(a_n) &= R_n(ResDB(a_{n-1}) \cup \Delta_n) \end{aligned}$$

- The Datalog program can be defined incrementally:

  $$\Pi_n(D_n) = T_n(ResDB(a_n))$$

- For put-calls let ts be a new timestamp. We define AT, where the residual rule tracks which entities have become legacy entities, as:

$$\begin{aligned}
&\Delta = \{kind(id, p_1, ...p_n, ts)\},\\
&R = \{legacykind(id, ts) : -kind(id, p_1, ..., p_n, ts),\\
&\quad kind(id, s_1, ...s_n, nts), ts < nts.\},\\
&T = \emptyset,\\
&p = empty\ query
\end{aligned}$$

- For get-calls which request an entity, $kind(id, p_1, ...p_n) := get(kind, id)$, we consider ts be a new timestamp. We define AT:

$$\begin{aligned}
&\Delta = \emptyset,\\
&R = \{legacykind(id, ts) : -kind(id, p_1, ..., p_n, ts),\\
&\quad kind(id, s_1, ...s_n, nts), ts < nts.\},\\
&T = \{latestkind(id, ts) : -\\
&\quad kind(id, p_1, ..., p_n, ts), not\ legacykind(id, ts).,\\
&\quad getkind(id, p_1, ..., p_n) : -\\
&\quad kind(id, p_1, ..., p_n, ts), latestkind(id, ts).\}\\
&p = getkind(id, p_1, ..., p_n)
\end{aligned}$$

These basic operations are capable to work with one application release. Therefore, the functions work with one schema version. The paper defines function $kind[r](id, p_1, ..., p_n)$ for the release r. The new release $r + 1$ introduces a new schema version $kind[r + 1](id, p_1, ..., p_m)$. We always compile the residual rules for each kind occurring in the database:

$$\begin{aligned}
&legacykind[r](id, ts) : -\\
&\quad kind[r](id, p_1, ...p_n, ts),\\
&\quad kind[r](id, s_1, ...s_n, nts), ts < nts
\end{aligned}$$

$$\begin{aligned}
&latestkind[r](id, ts) : -\\
&\quad kind[r](id, p_1, ...p_n, ts), not\ legacykind[r](id, ts).
\end{aligned}$$

For each new release, we also declare a new schema version for other kinds $kind'[r](ID, A_1, ..A_k)$ that are not affected by the new schema version. We declare the function as a residual function with new timestamp ts:

$$\begin{aligned}&kind'[r+1](id, a_1, ..., a_k, ts) :- \\ &\quad kind'[r](id, a_1, ..., a_k, ots), latestkind'[r](id, ots).\end{aligned}$$

Now, we have specified all required functions for the schema evolution operations. Figure 3.9 shows the residual rules for the schema evolution operations. The rules for adding, removing and renaming properties are straightforward and trivial. In copying and moving properties, the functions always assume a 1:N relationship between the source and the target kind. The reason why authors assume the 1:N relationship is because then the result of the migration does not depend on the order of update of entities. However, if the relationship is N:M between the source and the target kind, then the execution order influences the migration result. The authors want safe migrations, so they assume the 1:N relationship.

3.4.2 Data Migration Protocols

The authors choose an algorithm for Datalog evaluation which provides put- and get-calls eager evaluation. As we have mentioned, schema migration can be executed eagerly or lazily. We assume that put- and get-calls are always evaluated eagerly, and therefore in an incremental bottom-up approach.

The defined functions and rules only generate non-recursive Datalog with negation, $Datalog^{\neg}_{non-rec}$. This fact allows us to evaluate rules bottom-up, in the order of their dependencies, or simpler, the timestamps assigned to them.

Eager Migration

Since the residual rules for data migration are within $Datalog^{\neg}_{non-rec}$, they may also be evaluated bottom-up and incrementally. A simple straightforward optimization for the eager approach is discarding all legacy entities. We can consider $kind[i](id, p_1, ..., p_n, ts_i)$ and $kind[j](id, s_1, ..., s_n, ts_j)$, we may discard the entities if $ts_i < ts_j$. Actually, many NoSQL databases follow an append-approach with put-calls and timestamps entities. A database has a store-internal garbage collector that discards legacy entities no longer needed.

Lazy Migration

Lazy migrations are triggered by get-calls and evaluated top-down. They derive only necessary facts from the residual rules. The lazy migration approach needs to hold on to legacy entities in several versions (until they are no longer needed). Thus, the approach needs a special-purpose garbage collector. The collector needs

Let $kind[r](id, p_1, ..., p_n)$ be the current schema, ts be a new timestamp associated with release r. For operation *copy* and *move* let $kind_S[r](id, s_1, ...s_n)$ and $kind_T[r](id, t_1, ..., t_m)$ be the current source and target schema.

- **add** $kind.p_{n+1} = v$, where p_{n+1} is a new property name with default value v:

$$\begin{aligned} & kind[r+1](id, p_1, ...p_n, v, ts) :- \\ & \quad kind[r](id, p_1, ..., p_n, ots), latestkind[r](id, ots). \end{aligned}$$

- **delete** $kind.p_i$:

$$\begin{aligned} & kind[r+1](id, p_1, ..., p_{i-1}, p_{i+1}, ..., p_n, ts) :- \\ & \quad kind[r](id, p_1, ..., p_n, ots), latestkind[r](id, ots). \end{aligned}$$

- **copy** $kind_S.s_i$ **to** $kind_T$ where $kind_S.id = kind_T.t_j$:

$$\begin{aligned} & kind_T(id_T, t_1, ..., t_m, s_i, ts) :- \\ & \qquad kind_T[r](id_T, t_1, ..., t_m, ts_T), latestkind_T[r](id_T, ts_T), \\ & \qquad kind_S[r](id_S, s_1, ..., s_n, ts_S), latestkind_S[r](id_S, ts_S), \\ & kind_T(id_T, t_1, ..., t_m, null, ts) :- \\ & \qquad kind_T[r](id_T, t_1, ..., t_m, ts_T), latestkind_T[r](id_T, ts_T), \\ & \quad not\ kind_S[r](id_S, s_1, ..., s_n, ts_S), \\ & \qquad id_s = t_j. \end{aligned}$$

- **move** $kind_S.s_i$ **to** $kind_T$ where $kind_S.id = kind_T.t_j$, with the same rules as for copy and the following rule:

$$\begin{aligned} & kind_S[r+1](id, s_1, ..., s_{i-1}, s_{i+1}, ..., s_n, ts) :- \\ & \quad kind_S[r](id, s_1, ..., s_n, ots), latestkind[r](id, ots). \end{aligned}$$

Figure 3.9: Residual rules for migrations declared in the schema evolution language

```
public class Player {
        @Id Long id;
        String name;
        Integer level;
}
```

Figure 3.10: Java object mapper class Player with property *level*.

to hold on to residual rules until they, cannot be used anymore, since all matching entities have meanwhile been garbage collected.

To sum up, the Datalog rules define clear semantics for data migration and a pair of correct evaluations (eager and lazy). These evaluations guarantee to obtain correct results. So far, the Datalog rules were not implemented in any distributed system. One of the reasons for it is reliability of assigning global timestamps in the distributed system which is crucial for the solution.

3.5 ControVol

ControVol [9] is a framework for controlled schema evolution in NoSQL application development which was presented by S. Scherzinger, T. Cerqueus and E. Cunha de Almeida [64]. ControVol is integrated into an IDE and statically checks Java object mapper class declarations against schema evolution history. It obtains information about history from the code repository. The plugin works with Objectify, Morphia or Hibernate OGM NoSQL mappers.

As the authors mention, ControVol is the first tool of its kind specifically designed for NoSQL data stores.

ControVol framework is a plugin for Eclipse IDE and it is capable of detecting schema evolution problems (mismatched data and schema) as a warning or also suggest quick fixes which can be automatically resolved by a developer. The core of the framework is its static type checking tool which checks if object mapper class declarations are compatible. ControVol requires access to code repository (Git), and thus it knows the history of all released class definitions. With this knowledge the framework checks each change of object mapper class declaration against the schema evolution history oh the class.

3.5.1 Schema Migration Warnings

ControVol is capable to detect a schema migration warning. For example it can be caused by renaming of a property of a class declaration in Figure 3.10 to a class declaration in Figure 3.11 when an application uses Objectify. The property

```
public class Player {
        @Id Long id;
        String name;
        Integer rank; // level is renamed
}
```

Figure 3.11: Java object mapper class Player with property *rank*.

level is renamed to *rank*. The authors use the notation inspired by the Machivelli system [6]. From the old object declaration (Figure 3.10) we derive the following mapping function:

$$\begin{aligned} Player_a : \{[\![login : String, name : String, level : Integer]\!]\} \\ \rightarrow \{[\![login : String, name : String, level : Integer]\!]\} \end{aligned}$$

The domain of this function specifies the set of safely loaded entities (i.e., no data loss or exception during class initialization). The codomain specifies persisted entities according to the object mapper class declaration. The mapper excepts the only player of the same type with properties *login*, *name* and *level*.

For the newer version according to the class declaration in Figure 3.11 we derived a similar mapping function which expects a property *rank* (instead of *level*):

$$\begin{aligned} Player_b : [\![login : String, name : String, rank : Integer]\!] \\ \rightarrow [\![login : String, name : String, rank : Integer]\!] \end{aligned}$$

Now the legacy players cannot be safely loaded according to the new class declaration because they are persisted with property *level*. As we can see, the codomain of $Player_a$ and the domain of $Player_b$ do not match. However, the application can load the persisted entities without run-time exception. The mapper initializes new property *rank* to zero and ignores old *level* one.

ControVol detects inconsistency of the older class codomain and the newer class domain as a potential pitfall and reports a warning in the Eclipse IDE.

3.5.2 Quick Fixes

ControVol is also capable to suggest so-called **quick fixes** to resolve warnings, which can be applied automatically (a few mouse clicks) by Eclipse IDE. For the example from Section 3.5.1 ControVol proposes these three quick fixes:

- Add Objectify annotation `@AlsoLoad("level")` to property *rank* that explicitly renames *level* to *rank*.

```
public class Player {
        @Id Long id;
        String name;
        @AlsoLoad("level") Integer rank;
}
```

Figure 3.12: Java object mapper class Player with property *level* renamed to *rank*.

- Add annotation `@Ignore` to property *level* that makes clear that *level* is removed.
- Restore property *level* which prevents losing its value (both properties *level* and *rank* will co-exist).

3.5.3 Recognized Annotations

A developer can use annotations (i.e., from Objectify) without ControVol help, so the framework must type check the annotations. ControVol checks all standard Objectify annotations (described in Section 3.2.1).

For instance, let us consider class declaration from Figure 3.12. The derived mapping function is:

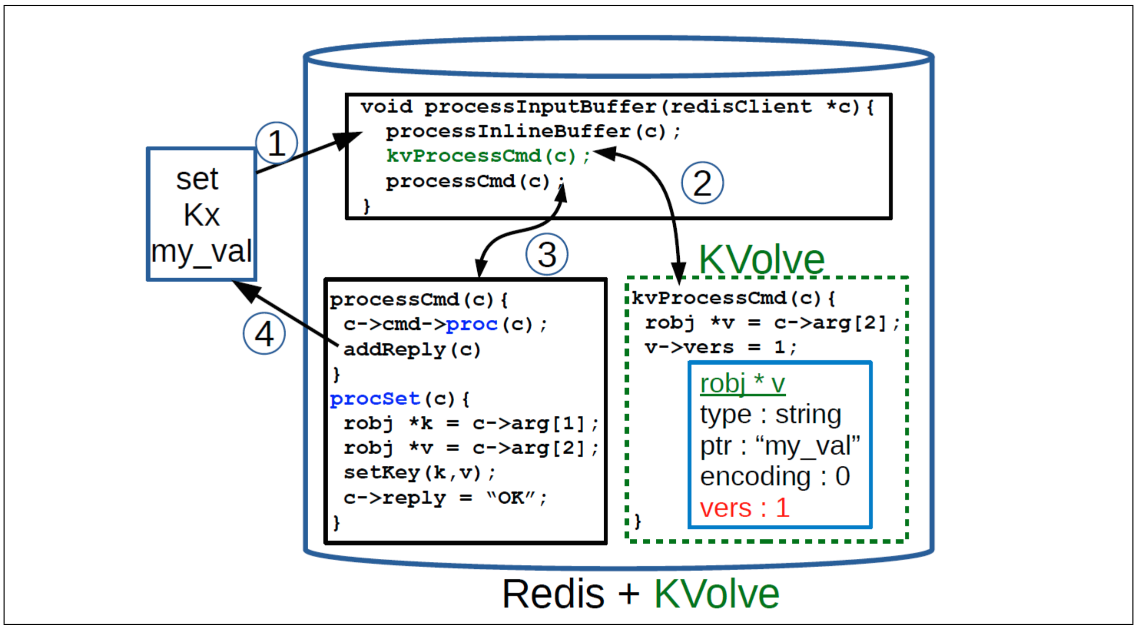
$$\begin{aligned} Player_c : & \llbracket login : String, name : String, level : Integer \rrbracket \\ & \rightarrow \llbracket login : String, name : String, rank : Integer \rrbracket \\ Player_c : & \llbracket login : String, name : String, rank : Integer \rrbracket \\ & \rightarrow \llbracket login : String, name : String, rank : Integer \rrbracket \end{aligned}$$

This object mapper class declaration loads both legacy and current player entities alike. ControVol does not report any warnings.

3.6 KVolve

KVolve is an extension to the key/value NoSQL database Redis developed and presented by K. Saur, T. Dumitras, and M. W. Hicks [62]. The authors use the terminology from the NoSQL Distilled [61] book and introduce a mechanism for **incremental migration** of key/value format records. The term incremental migration is a synonym for lazy migration (defined in Section 3.3.1).

The aim of KVolve is an implementation of lazy migration to Redis using an added version field *vers* which enables to detect possible migrations. The implementation of KVolve is a separate C library that is compiled into Redis



```
void processInputBuffer(redisClient *c){
  processInlineBuffer(c);
  kvProcessCmd(c);
  processCmd(c);
}
```

```
processCmd(c){
  c->cmd->proc(c);
  addReply(c)
}
procSet(c){
  robj *k = c->arg[1];
  robj *v = c->arg[2];
  setKey(k,v);
  c->reply = "OK";
}
```

```
kvProcessCmd(c){
  robj *v = c->arg[2];
  v->vers = 1;
}
```

Figure 3.13: Control Flow for Redis and KVolve

itself. The authors choose this approach as a solution with the best performance and it should be easy to maintain: KVolve adds the version field *vers* to the data structure and extends standard Redis request processing. KVolve supports 36 Redis commands and all of the main data structures (string, set, list, hash, and sorted set). The supported commands are the most commonly used ones. KVolve works by pre-processing incoming commands from the client before passing them to Redis.

KVolve processes incoming command in 4 steps. We will describe them on an example of command **set Kx value** as depicted in Figure 3.13:

1. The client issues command **set Kx value**.

2. Redis calls extension KVolve function *kvProcessCmd(c)* to pre-process the command (which might involve changes of data and version field).

3. Redis follows standard execution with *processCmd* function (which depends on the choice of command – in our example *procSet* is called because the client requested *set* command), and adds the object to the database, including changes to the version field set during the KVolve pre-processing.

4. Redis responds to the request, that it executed the *set* request.

The *kvProcessCmd(c)* function controls the migration. The function works based on the **version hash table**. The table contains information about the entity

```
void update_function (char ** key, void ** value , size t * val len )
    {
     json t *root, *arr , *ele , *price;
     int i; double pval; json_error_t error;
     root = json_loads((char*)*value, 0, &error);
     arr = json_object get(
                json_object_get(root, "order"), "orderItems");
     for (i = 0; i < json_array_size(arr); i++){
          ele = json_array_get(arr, i );
          price = json_object_get(ele, "price");
          json_object_set(ele, "discountedPrice",
          json_real(json_real_value(price ) - 3.0));
          json_object_set(ele, "fullPrice", price);
          json_object_del(ele, "price");
     }
     *value = json_dumps(root, 0);// Set the updated value
     * val len = strlen(*value );// Set the updated val length
}
```

Figure 3.14: Example of update function for JSON for KVolve

migrations and maps a namespace [1] to a record that involves current and previous versions and update functions that move data from one version to the next. It also contains information about client connections which declare an interest in the current version of this namespace (declares a head version).

The developer must specify migration rules which are specified as update functions from the old key/value format to the new one. KVolve does not support migrations in which a new key/value is created from several old key/values. These functions need to be created by a developer and uploaded to KVolve. Therefore, KVolve introduces the following function:

```
kvolve_upd_spec(
          old_namespace,
          new_namespace,
          old_version,
          new_version,
          migration_func)
```

The developer can specify the migration rule by executing a new Redis function *kvolve_upd_spec* and migrate namespace *order* from version 0 to 1 by the function from Figure 3.14:

```
kvolve upd spec("order","order", 0, 1, 1, update_function);
```

[1]Namespaces are commonly used in key/value stores and conceptually divide the kinds of objects in the database.

KVolve stores the information to the version hash table. Now when the client requests a record from *order* namespace of version 1, KVolve executes the *update_function* and returns the record.

In general we can describe three situations when the client calls *get K* command. Function *kvProcessCmd(c)* gets the object through the version hash table and checks the version field $vers$ of the record against the current client connection version $vers_{head}$ in the table:

- $vers < vers_{head}$ – KVolve migrates the record to the head version and stores the updated record. The old version of the key K is freed.
- $vers = vers_{head}$ – KVolve does nothing.
- $vers > vers_{head}$ – KVolve raises an exception and does not return any record. Outdated clients will not be permitted to reconnect with the old version of the software.

To sum up, KVolve can lazily migrate an object to its new version by the given update functions. The functions can use only the old key or value, and cannot use any other data in the database. KVolve can return older version of data if the data are not migrated yet and a client connection requests the older version. Otherwise, if the version of stored data is bigger than a requested version, it closes the connection with an exception.

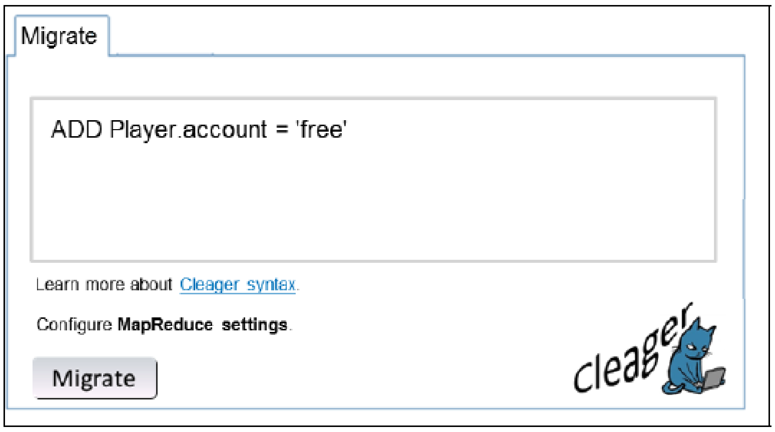
3.7 CLeager

CLeager is a tool for eager schema evolution in NoSQL document stores which was presented by S. Scherzinger, M. Klettke and U. Störl [65]. The CLeager framework offers a new and declarative alternative for eager data migration based on the NoSQLSEL which was introduced in Section 3.3.1.

The authors developed it as a stand-alone application which provides a console for migrations written in the language. The developer types a migration script into the CLeager console, and it executes the migration as MapReduce jobs, making use of massive parallelization of the Google Cloud Platform infrastructure. The MapReduce jobs can do schema modification operations, such as adding, deleting, renaming, moving or copying properties in batch.

Figure 3.15 shows an example of the CLeager console and a migration of Player entity. The migration adds a property *account* with a value *free* to all Player entities in a database.

CLeager has no relation/connection with the target application or an application code. When the developer wants to make changes in the class definition mapper, he must create them manually and also execute the CLeager migration.



ADD Player.account = 'free'

Learn more about Cleager syntax.
Configure MapReduce settings.
Migrate
cleager

```
def map(String key, JSON value) {
        if value is a Player object
        then value.addProperty("account", "free");
        put(key, value); // Persisting the change.
}
/* A map job is sufficient. */
```

Figure 3.15: Adding a property in the CLeager console and the MapReduce job for adding the property

3.8 Schema Evolution in XML-based Databases

A special kind of NoSQL databases is an XML database. For them, there already exist frameworks for schema evolution management. The works are not focused just on databases, but they are commonly focused on schema evolution of XML data.

In [58] the authors introduce principles of an evolution management framework. The main goal of the framework is to help a designer with the following problems:

- to make the required change easily and correctly,
- to identify all affected parts of the system,
- to make the respective changes of the affected parts semantically correct,
- to express the changes also syntactically correctly regarding the selected format,
- to integrate new schemas and discover relations to the current ones.

In the authors' previous papers, they introduce an idea of a five-level evolution management framework. In Figure 3.16 we can see the architecture of the framework. It is partitioned vertically and horizontally and in both cases its components

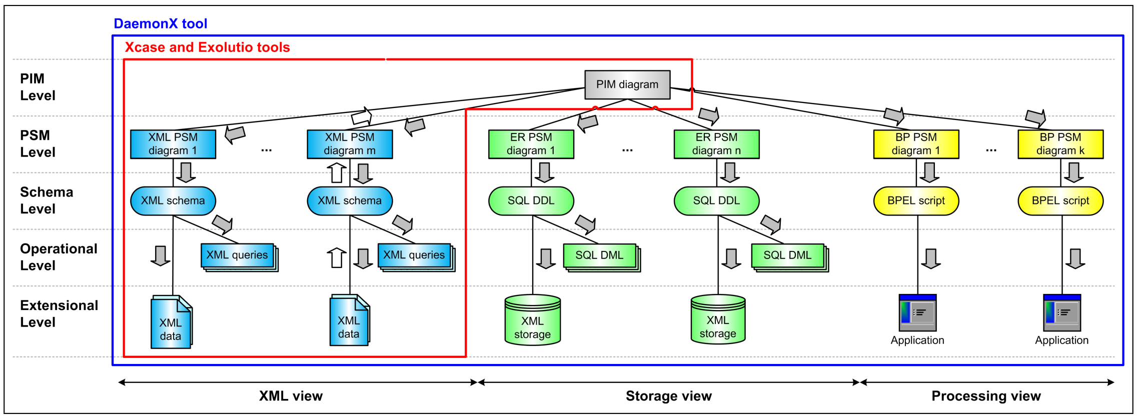


Figure 3.16: A five-level XML evolution architecture of DaemonX and eXolutio. The picture also shows scope of both Xcase and DaemonX frameworks.

are related and interconnected. The vertical partitioning identifies three views of the system. The authors depicted them as the most common and representative views:

- Blue part (●) – Covers an **XML view** of the data processed and exchanged in the system.
- Green part (●) – Represents a **storage view** of the system, e.g. relation view of the data.
- Yellow part (●) – Represents a **processing view** of the data, e.g. a chain of Web Services described using BPEL scripts.

The horizontal partitioning defines five levels. Each level represents a different view of an XML system and its evolution:

- **Platform-Independent Level** – contains **Platform-Independent Models** (PIMs). A schema in the PIM models real-world concepts and relationships between them without any details of their representation in, e.g., an XML data model. The authors use UML class diagrams for representation of the PIM.
- **Platform-Specific Level** – contains **Platform-Specific models** (PSM). A schema in the PSM describes how a part of the reality modeled by the PIM schema is represented. The PSM is created for each, e.g., XML schema. The authors use UML class diagrams extended for the purposes of XML modeling. The extension is needed because the standard UML constructs do not support several specifics of XML (such as hierarchical structure or distinction between XML elements and attributes).

- **Schema Level** – represents schemas that describe the structure of the instances, e.g. XML schemas or SQL/XML Data Definition Language (DDL).

- **Operational Level** – represents operations over the instances, e.g. XML queries over the XML data expressed in XQuery [70] or SQL/XML [24] queries over relations.

- **Extensional Level** – represents the particular instances that form the implemented system such as, e.g., XML documents, relational tables or Web Services that are components of particular business processes.

Change propagation in the defined hierarchy of models can be done semi-automatically. In [36] the authors introduce an algorithm for automatic revalidation of XML documents according to changes in the respective XML schema. The algorithm can validate two versions of a document (with structural modification) automatically; user interaction is required only where necessary (e.g. when a content must be added, etc.). The paper introduces a mechanism for generating an XSLT change script which can be applied to the old version of a document.

In the defined hierarchy there is no need to provide a mapping between PSMs, but only from every PSM to PIM. The paper defines two kinds of propagation:

- **Upwards Propagation** (UP) – For instance, if the change occurs in an XML schema, it is propagated to PSM schema and PIM schema.

- **Downwards Propagation** (DP) – Propagates changes from upwards propagation back to the affected all PSMs and bellow to the related parts of the system.

In their previous paper [51] the authors compares previously developed a schema evolution framework eXolutio and a framework XCase. XCase was the first framework which implements the described ideas and eXolutio follows the ideas and more generalizes them. DaemonX is the next step in the author's effort. In the following text, we will focus on the DaemonX framework.

3.8.1 DaemonX Framework

DaemonX implements the described five-level architecture. The framework is a plug-in-able tool for data and process modeling. The plug-ins are managed by the application core and define specific functionality like evolution processes between models.

The evolution process is managed by the DaemonX's evolution manager. The manager controls all evolution plug-ins which define upwards/downwards propagation from source to target model. The evolution plug-ins define so-called **evolution references** which define relationships between source and target models.

The evolution process starts with an analysis of an operation done in a source modeling plug-in. The result of the analysis is a collection of operations (from the evolution plug-in) which must be processed in the target models. These changes can be an input of another evolution plug-in (transitively related models).

The first DaemonX release contains the following model plug-ins:

- PIM Model – models the problem domain,
- XSEM [71] PSM Model – models XML data,
- UML Class Model – models general data structures,
- Relational Model – models relational data,
- BPMN [32] Model – models business processes.

And the release also contains the following evolution plug-ins:

- UML to UML,
- PIM to XSEM PSM,
- XSEM PSM to PIM,
- PIM to relational model.

DaemonX provides a GUI for system designers where they can use modeling plug-ins for data modifications. The framework also provides undo-redo support that allows them to safely revert changes. The undo-redo operation is not supported for each operation, so the GUI visualizes the operation with a different color.

To sum up, the authors of DaemonX created an extendable framework for schema evolution. The framework can be use for common systems and the presented version supports mainly XML systems. The user of the framework does not need to create migration scripts, because the framework is able to generate and semi-automatically apply them.

3.9 Summary

The tools and approaches that we mentioned in previous sections show us that there is no general approach to schema evolution for all NoSQL databases. The mentioned approaches cover different areas of databases and provide different functionality for schema evolution management. We can compare them from many perspectives but the following ones are more important to us to see their crucial differences.

3.9.1 Target Database of the Schema Evolution

As we mentioned at the beginning of this section, there is no general approach to schema evolution. We can divide the approaches to several categories by their target database. On one hand, we introduced technologies which are closely bound to the specific databases and, on the other hand, we mentioned the schema evolution language for NoSQL databases. These are completely different approaches to the problem.

We can divide them into two groups – tools which are bounded to specific database(s) (Objectify, Morphia, Hibernate OGM, MongoRepository, Mongoid Evolver, KVolve, ControVol) and tools and papers which solve schema evolution at an abstract layer without any specific database (DaemonX, NoSQLSEL). The first group contains well-defined approaches to the schema evolution which can be used by developers immediately but the provided functionalities are weaker and do not provide an extensive benefit to them. The second group provides powerful functionalities but the approaches are mainly theoretical and not ready for massive production usage, because they usually have just a prototype implementation.

3.9.2 Timing of the Schema Evolution

An important difference between the approaches is the time when the migration is executed. There are two main ways how to execute it – eagerly (CLeager) and lazily (KVolve). Also, there are abstract frameworks like DaamonX which do not care about the execution time, they solve the evolution in data models. The rest of tools like Objectify prove methods to perform the migration but time and correctness of the migration fully depends on a developer.

3.9.3 Provided Functionalities for Schema Evolution

The most important difference for developers is the list of provided functionalities. The introduced approaches and tools cannot be easily compared by their functionalities, because they deal with different abstract layer of schema evolution processing/management. Another problem of the comparison is that a part of them is a real-world solution and the rest of them is mainly a theoretical work.

At the end of Section 3.2 we compared all the mentioned standard ORM tools so we are not going to compare them again but we choose Objectify as a representative example of the ORM tools for the following comparison. The other representative example will be CLeager because it is an implementation of the NoSQLSEL.

Firstly, we compare these two representatives, because we need to find a mapping or similarities between the their operations. For recapitulation, here are the lists of them:

Objectify	@AlsoLoad @Ignore @IgnoreLoad @IgnoreSave @OnLoad @OnSave

CLeager	add delete rename move copy

We do not want to introduce a proper mapping algorithm between their operations but we want to find out if one can exists. CLeager is using a well defined language so we will try to find a theoretical way how to map Objectify functions. The operations *add* and *delete* are trivial, we can introduce a new field or remove an old one anytime without any annotations. To implement operation *rename*, we can use function `@AlsoLoad`. This function allows us to implement any data transformation with loaded data as renaming or the data transformation. Operations *move* and *copy* can be implemented using function `@OnLoad`. We can see the usage of these functions in Figure 3.1 and Figure 3.2. Other Objectify functions can be used during data processing, but they are not required for the basic set of functions for data evolution in a database.

ControVol is based on suggestions and quick fixes for mismatched data and schema against standard ORMs so we cannot compare operations as *add* or *remove*, because it is not the purpose of the tool.

KVolve works based on migration scripts which are uploaded directly to Redis database. It means that a developer can implement any transformation in language C with one exception. The developer cannot implement migration from several keys to one new key. KVolve uses a key/value database so operation *add/remove* does not make sense but the developer is able to *move*, *copy* or *rename* data.

DaemonX uses PIM to PSM (and back) transformations so the provided operations depend on the used models. In case of XML, it uses semi-automated migrations. When DaemonX detects an ambiguous migration, the developer is asked for an assistance. The result is that the developer is able to do any transformation on PIM and propagate it to XML PSM or back to the PIM.

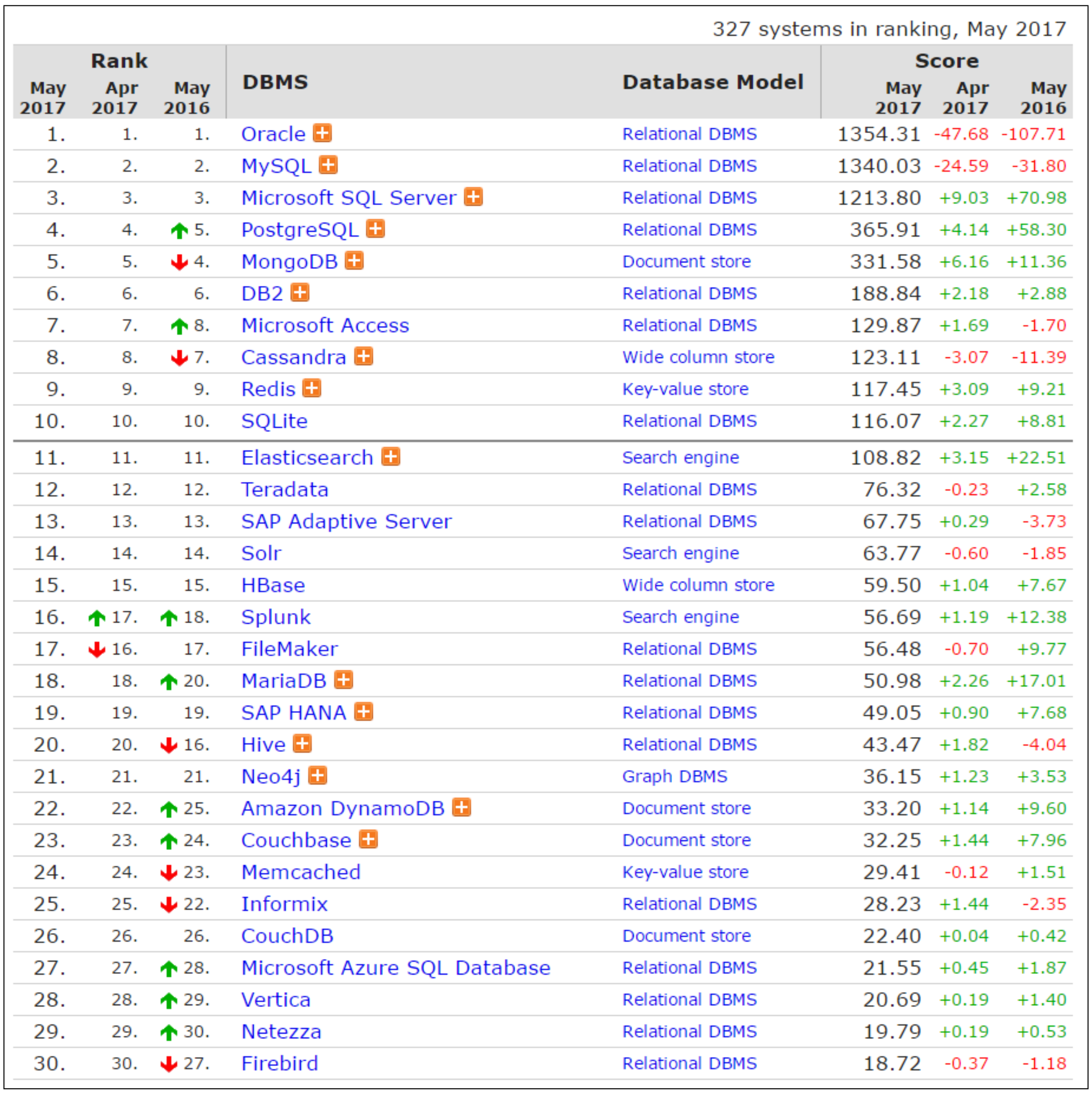
3.9.4 General Approach

As we mentioned before, currently there does not exists a general approach for NoSQL databases. The team of S. Scherzinger, M. Klettke and U. Störl introduced a few papers about the NoSQLSEL (see Section 3.3.1). The language is a good way how to introduce one common approach applicable to schema evolution. The authors introduced three basic operations *add*, *delete* and *rename*, and two dedicated operations *move* and *copy*. This set of five operations is powerful to manage standard schema evolution.

The language is focused on document databases and described mainly for JSON/BSON databases. The authors excluded key/value databases because they do not have any schema, so the evolution of schema and data is the responsibility of applications. The current definition of the language also does not support column and graph databases. The authors explain the challenges of column databases, but they do not explain the absence of graph databases. We understand the lack of support for the other types of NoSQL databases, because document databases are the most popular part of NoSQL and also the most commonly used ones. We can see this fact on the web page [15] which show us the most popular database engines. Figure 3.17 shows us a list of top 30 engines. Relational databases dominate in the list but if we exclude them, document databases will be the most popular ones.

The focus on JSON/BSON databases is not a critical limitation. We can easily propose an extension to XML databases, because they are based on a hierarchical data model too.

To sum up, a developer is able to execute basic operations like *add*, *delete*, *rename*, *move*, or *copy* with any introduced tool or theory. A difference in provided operations is mainly in the way how they are provided, i.e. whether the developer is able to modify code, model, or implement migration function.

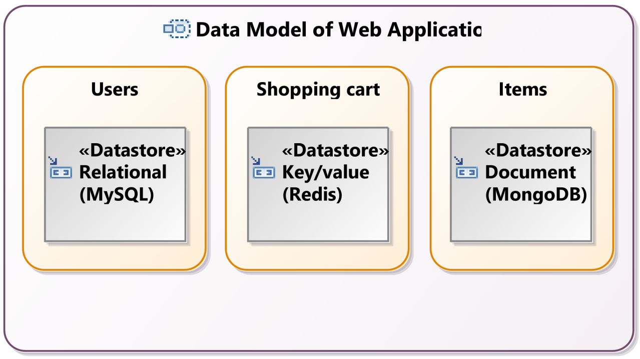

327 systems in ranking, May 2017

Rank			DBMS	Database Model	Score		
May 2017	Apr 2017	May 2016			May 2017	Apr 2017	May 2016
1.	1.	1.	Oracle	Relational DBMS	1354.31	-47.68	-107.71
2.	2.	2.	MySQL	Relational DBMS	1340.03	-24.59	-31.80
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1213.80	+9.03	+70.98
4.	4.	↑ 5.	PostgreSQL	Relational DBMS	365.91	+4.14	+58.30
5.	5.	↓ 4.	MongoDB	Document store	331.58	+6.16	+11.36
6.	6.	6.	DB2	Relational DBMS	188.84	+2.18	+2.88
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	129.87	+1.69	-1.70
8.	8.	↓ 7.	Cassandra	Wide column store	123.11	-3.07	-11.39
9.	9.	9.	Redis	Key-value store	117.45	+3.09	+9.21
10.	10.	10.	SQLite	Relational DBMS	116.07	+2.27	+8.81
11.	11.	11.	Elasticsearch	Search engine	108.82	+3.15	+22.51
12.	12.	12.	Teradata	Relational DBMS	76.32	-0.23	+2.58
13.	13.	13.	SAP Adaptive Server	Relational DBMS	67.75	+0.29	-3.73
14.	14.	14.	Solr	Search engine	63.77	-0.60	-1.85
15.	15.	15.	HBase	Wide column store	59.50	+1.04	+7.67
16.	↑ 17.	↑ 18.	Splunk	Search engine	56.69	+1.19	+12.38
17.	↓ 16.	17.	FileMaker	Relational DBMS	56.48	-0.70	+9.77
18.	18.	↑ 20.	MariaDB	Relational DBMS	50.98	+2.26	+17.01
19.	19.	19.	SAP HANA	Relational DBMS	49.05	+0.90	+7.68
20.	20.	↓ 16.	Hive	Relational DBMS	43.47	+1.82	-4.04
21.	21.	21.	Neo4j	Graph DBMS	36.15	+1.23	+3.53
22.	22.	↑ 25.	Amazon DynamoDB	Document store	33.20	+1.14	+9.60
23.	23.	↑ 24.	Couchbase	Document store	32.25	+1.44	+7.96
24.	24.	↓ 23.	Memcached	Key-value store	29.41	-0.12	+1.51
25.	25.	↓ 22.	Informix	Relational DBMS	28.23	+1.44	-2.35
26.	26.	26.	CouchDB	Document store	22.40	+0.04	+0.42
27.	27.	↑ 28.	Microsoft Azure SQL Database	Relational DBMS	21.55	+0.45	+1.87
28.	28.	↑ 29.	Vertica	Relational DBMS	20.69	+0.19	+1.40
29.	29.	↑ 30.	Netezza	Relational DBMS	19.79	+0.19	+0.53
30.	30.	↓ 27.	Firebird	Relational DBMS	18.72	-0.37	-1.18

Figure 3.17: The most used database engines [15]

4. Multi-model Databases

Most standard databases use a single model for storing data. The used model determines how the stored data can be used (i.e. queried, stored, organized or manipulated) and the single-model databases are well-optimized for their model.



Data Model of Web Applicatic
Users
«Datastore» Relational (MySQL)
Shopping cart
«Datastore» Key/value (Redis)
Items
«Datastore» Document (MongoDB)

Figure 4.1: An example of data model for an e-shop application

On the other hand, a lot of modern applications work with different types of data. A typical example of a modern application is an e-shop which sells products. This kind of application has to store at least information about users and products and it has to allow a user to buy products online through a shopping cart. Figure 4.1 shows data models for the application data and examples of possible databases for each kind of data. A problem for developers of the application is that they need three different databases. They need the knowledge of each of them and they have to integrate, manage and maintain three database technologies. This approach of using a variety of appropriate data models for different parts of the application is called **polyglot persistence** and it is described by Martin Fowler in [19]. In a nutshell, polyglot persistence is an idea which describes advantages and disadvantages of using multiple database technologies for storing divergent types of data based upon the way data are being used by an application.

The idea of polyglot persistence is one of the main reasons why there is a rising popularity of **Multi-Model Databases** (MMD). The MMDs provide a unified platform for applications which need to work with multi-model data. They provide most of the standard database functions such as the ability to query across all the models, indices or transaction support. Currently, there is no unified set of models which is supported by all MMDs. A typical set of supported models is a subset of the following data models:

- relational,

- column,
- key/value,
- document,
- graph.

This set covers most of the requirements of modern applications. The relational and column model can store tabular data in a structured form, the key/value model is suitable for hash tables, the document model stores semi-structured or object-like data, and the graph model is intended mainly for highly referential data.

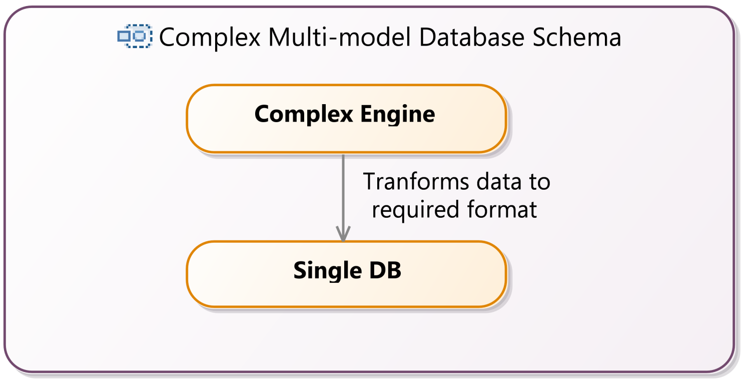
The MMDs are the way to solve the polyglot persistence problem and offer advantages of data modeling and managing without disadvantages of using different database technologies for each kind of data. The MMDs provide all models (a particular subset) in one database with unified query language and API for all supported models.

However, the key problem of the multi-model approach is that it increases system complexity for database operations. For example, the database has to pre-process all operations to serve them to the target model or manage all cross-model operations and all problems like transactivity.

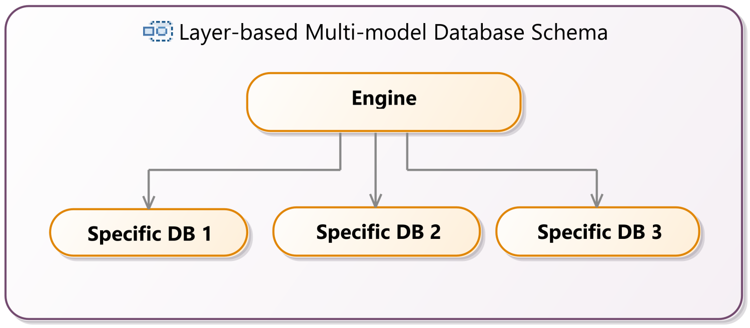
4.1 Architecture of MMD

Problems of cross-model operations can be partially solved by the architecture of the database. In general, there are two main ways how to support different models:

- **Complex engine** – The database transforms all supported data types to a single core model. Its engine has to pre-process all operations for the core model. For example, a document store (supporting JSON documents), a key/value store, and a graph store can all be successfully represented as JSON documents. Documents are straightforward, key/value pairs are stored in flat documents, and graph needs an extension which stores documents for each vertex and a document for each edge. An example of MMD which uses a complex engine is CouchBase [10].
- **Layered architecture** – The database supports different models via different layers on top of an engine. Data are stored in the relevant model. Each data model has its own component which communicates with the engine. An example of MMD which is based on layered architecture is Oracle Database 12c [4]. It distinguishes between relational data, XML data, and BLOBs (stored as documents). Each of the models is stored separately and


Complex Multi-model Database Schema
Complex Engine
Tranforms data to required format
Single DB

(a) A diagram of a MMD with a complex engine.



(b) A diagram of a MMD with a layered architecture.

Figure 4.2: A diagram of both types of MMDs.

using an own processing engine. On top of them, Oracle Database 12c uses a layer which can control all operations with data in the specific engines and properly split/merge operations and their data.

Figure 4.2 shows a schema of a MMD with complex engine and a MMD with layered architecture.

We decided to focus on the layered architecture because there is no need to introduce generic approach for specific complex engines. The engines often use one model to store all supported models, so they can use schema evolution for the single model and on top of it use transformation logic to support the required models. The top level transformation can differ based on the used model and has to be adjusted for the specific case.

The layer-based approach allows us to visualize our ideas easily and does not mix models together. Figure 4.3 shows two main layers of the layer-based approach: **model-specific** layer and **model-independent** layer. The engine of MMD is

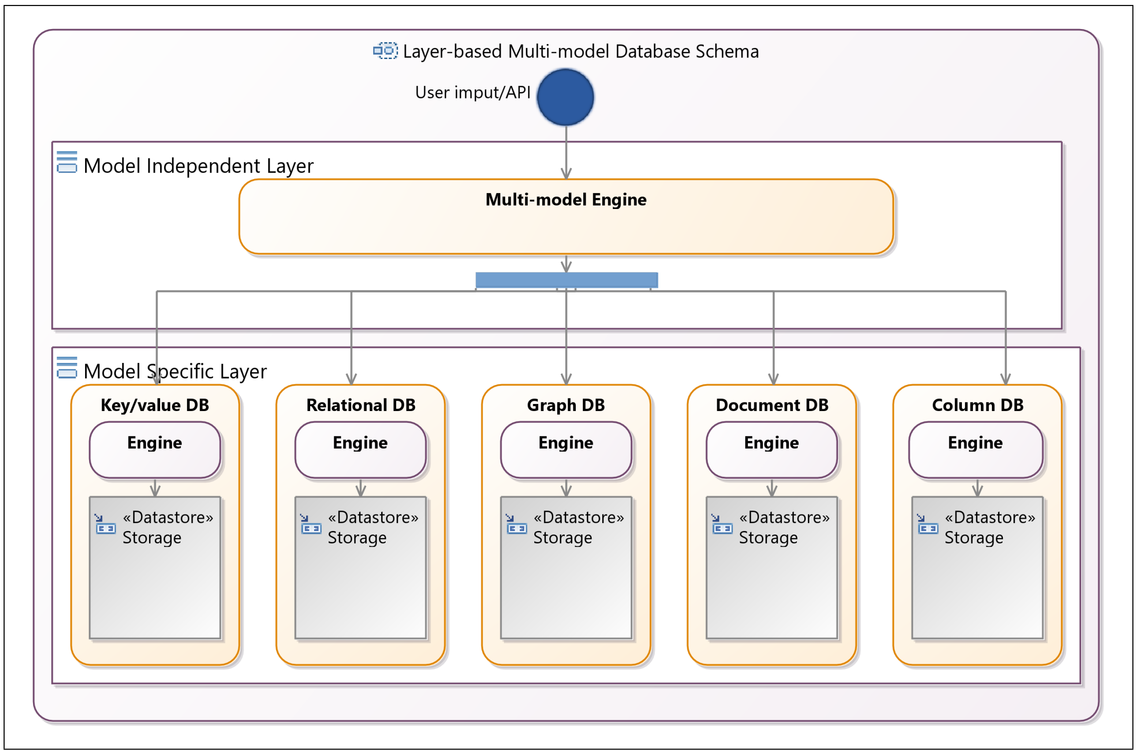


Figure 4.3: Schema of layer-based MMD with support of key/value, relational, graph, document and column models.

located in the model-independent layer. It is a facade for functions of the database as queries and distributes queries and commands to the respective models. Also, it collects data from them and creates the final result for the user. Each model from the model-specific layer provides the full functionality of the given kind of database.

Note that the concept of MMD in not new. We can find first notes about polyglot persistence and the idea of MMD in distributed database management systems (DBMS) which were studied in the early 80s and early 90s or in the idea of federation of relational engines [35].

4.2 Examples of MMDs

In [34] Jiaheng Lu and Irena Holubová study the current MMDs and introduce a detailed classification of them. They cite an article published by Gartner [20] that currently, all leading databases are going to become multi-model. Mainly they aim to support relational and NoSQL models. Table 4.1 shows examples of MMDs with the respective lists of supported models.

MMD	SQL	Key/value	Document	Graph	Text	Other
ArangoDB [3]	✗	✓	✓[a]	✗	✗	
CouchBase [10]	✓	✓	✓[a]	✗	✗	
CrateDB [11]	✓	✗	✓[c]	✗	✗	
MarkLogic [38]	✓	✗	✓[b]	✓[d]	✓	binary, geospatial
OrientDB [57]	✓	✓	✓[a]	✓	✓	binary, geospatial, reactive
Datastax [14]	✓	✓	✗	✓	✗	

[a] A JSON model.
[b] An XML model.
[c] A Lucene model.
[d] RDF plus OWL/RDFS.

Table 4.1: Examples of MMDs with the supported models.

4.3 Pros and Cons

The main advantage of an MMD is a single system for all required models. Developers have just one database to maintain, just one adapter for their application and also they have to learn details just about one database (such as query syntax, architecture or topology). The next advantage of MMDs is the cost of scaling-up. In the case when an application requires scaling of databases, developers can scale just the MMD technology instead of scaling all databases separately (scale a relational database, a document database, etc.).

The main disadvantage is the current state of the art of the databases. They are still relatively new and still being developed. The result is that they are immature and there are a lot of unsolved issues and challenges.

4.4 Open Challenges for MMDs

The biggest challenges of MMDs are as follows:

- **Inter-model queries** – The problem of queries in MMDs are references between different models. Each model has its own query engine and the challenge is how to connect these engines.

- **Inter-model transactions** – Transactions can be easily managed in particular models but an inter-model transaction is challenging mainly because of rollbacks and transaction persistence.

- **Schema and model evolution** – We have already mentioned challenges for NoSQL databases. Schema evolution in MMDs has to deal with inter-model evolution, inter-model references, and differences between schema evolution in each model.

5. Our Approach

This chapter describes our general approach for schema evolution in MMDs. We introduce a schema evolution language for MMDs and its set of supported operations, their meaning, and implementation in most common data models.

Currently, there exists no unified approach how to manage schema evolution in MMDs. We can find several approaches how to manage schema evolution in separate models, but to the best of our knowledge, there seems to exists no general approach to combined models.

In Chapter 3 we described several ways how to manage schema evolution in JSON and XML models. The XML model and the JSON model are both hierarchical so first, we will consider just JSON model as a representative of them.

In the context of NoSQL databases, we have found out that there is no general schema evolution approach in key/value databases except for KVolve which is a specific solution for Redis database. We cannot ignore them in MMDs because of cross-model schema evolution.

During our research, we have not found out any approach for schema evolution in graph databases.

5.1 Schema Evolution in MMDs

In our research, we mentioned DaemonX which has architecture similar to MMD. We can see a similarity with the PIM/PSM model of DaemonX where data can be stored in multiple databases or models. However, there is one big difference between DaemonX and MMDs which blocks us to use DaemonX to solve our problem: The current version of DaemonX does not allow us to store connected data across models but we can just duplicate them in multiple models.

To demonstrate the difference we will use an example from [58]. The authors of the paper explain DaemonX's evolution management on a model of an e-shop model design. Figure 5.1 shows PIM which contains the definition of entity *order* and it also shows three PSMs. The PSMs are targets of the PIM. The authors execute evolution which moves data attribute *address* from entity *User* to entity *Detail* and splits this attribute to new attributes *country*, *city*, *address* and *postCode*. They execute the evolution step by step in all three mentioned PSMs. In this example, we can see that DaemonX is able to handle a complex schema evolution of all three PSMs but its evolution algorithm is not able to handle migration which would, e.g., split data between multiple models. For the mentioned example it means that DaemonX can migrate attribute *address* only to one specific model. For a better understanding in the context of MMDs we can image that the exam-

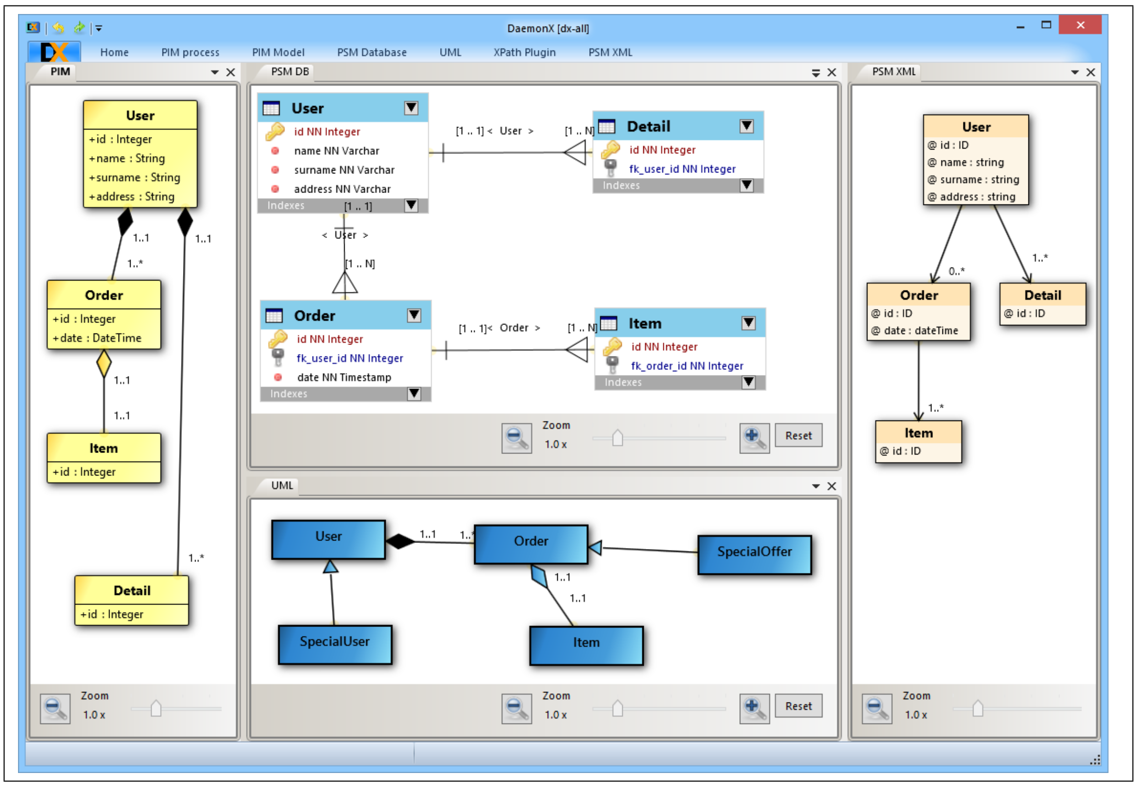


Figure 5.1: An example of model of eshop design in DaemonX [58].

ple is not using the XML and the DB PSMs, but it uses data models as PSMs, i.e., a document model and a relational model. The sample operation migrates data inside both models correctly. The problem is that we are not able to move data between PSMs. DaemonX is not able to remove all attributes *address* from the relational model and move them to the document model. It supports only migrations which do not distinguish target PSM.

With an non trivial effort it should be possible to extend DaemonX, but we decided not to use this way. The main reasons are that we want to focus on a general solution which can be implemented in any migration tool or MMD and the current complexity caused by different aim of DaemonX.

In this thesis we aim at a general solution of schema evolution in MMDs and the following models as the main representatives of MMD models:

- relational,
- column,
- graph,
- key/value, and
- JSON (and XML), i.e. document.

Since we focus on a general approach, we are not going to solve schema evolution between specific models one by one but we introduce a general model-independent solution.

5.1.1 Operations for Schema Evolution in MMDs

Our first goal is to introduce a set of operations for schema evolution in MMDs. In Section 3.9 we showed that most of the current NoSQL models provide similar operations for schema evolution. This fact allows us to use the set of operation as a common API for all specific models in the layered model.

Supported Operations by Specific Models

Our first goal is to define a common set of operations which can be supported by all data models. Our main requirement is that all functions have to be supported by the representatives of MMD models. In Section 3.3.1 we described the schema evolution language for NoSQL databases which covers most of the representatives. The language contains operations *add*, *delete*, *rename*, *move*, and *copy*. It is defined for document, column and key/value models.

```
{
  "vertex":[
    {
      "id": 0,
      "label": "Peter"
    },
    {
      "id": 1,
          "label": "George"
    },
    {
      "id": 2,
      "label": "John"
    }
  ],
  "edges":[
    {
      "id": 0,
      "source": 0,
      "target": 1,
      "label": "is friend of"
    },
    {
      "id": 1,
      "source": 2,
      "target": 0,
      "label": "hates"
    }
  ]
}
```

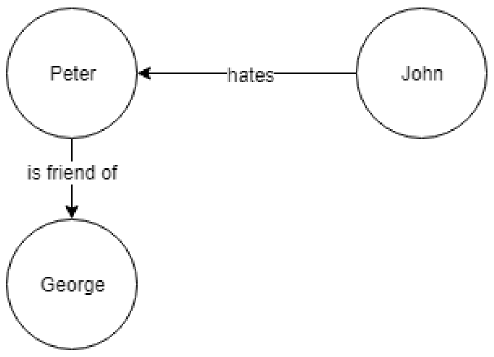


Figure 5.2: An example of representation of a graph model in the document model.

To cover all our representatives of MMD models we have to introduce an extension of the language for graph and relational model. We can consider the graph model as a document model which is created as a set of vertices and edges. A vertex is a document consisting of *id* and *label*. The edge is a set of an identificator of a source vertex (*source*) and an identificator of a target vertex (*target*) and *id* and property *label*. Figure 5.2 shows an example of a graph of relations between vertices *Peter*, *John*, and *George* which is represented in the document model.

With this representation, we can also use the language for the graph model. The only one which is remaining is the relational model. However, the whole NoSQLSEL is derived from the world of relational databases so it is not hard to extend it towards the relational model. We can consider that entities are represented as rows in a specific table and their properties are columns of the table. Each entity has a unique identification *id*. Below we show how the operations can be implemented in SQL:

- Operation *add entity.property* adds column *property* to table *entity*.

  ```
  ALTER TABLE entity ADD property;
  ```

- Operation *delete entity.property* deletes column *property* from table *entity*.

  ```
  ALTER TABLE entity DROP property;
  ```

- Operation *rename entity.property$_1$ to property$_2$* renames column *property$_1$* to *property$_2$* in table *entity*.

  ```
  ALTER TABLE entity CHANGE property₁ property₂;
  ```

- Operation *copy entity$_1$.property to entity$_2$.property* copies column *property* from table *entity$_1$* to *entity$_2$* as a sequence of commands:

  ```
  ALTER TABLE entity₂ ADD property;
  UPDATE entity₂
          SET property = (
                  SELECT property
                  FROM entity₁
                  WHERE entity₁.id = entity₂.id);
  ```

- Operation *move entity1.property to entity2.property* copies column *property* from table *entity1* to *entity2* and deletes column *property* from table *entity1*:

  ```
  ALTER TABLE entity₂ ADD property;
  UPDATE entity₂
          SET property = (
                  SELECT property
                  FROM entity₁
                  WHERE entity₁.id = entity₂.id);
  ALTER TABLE entity₁ DROP property;
  ```

Note that we do not focus on effectivity of this implementation, because our goal is to prove that it is possible to implement the language in a relational model.

We will call this extension of the language the **Database Schema Evolution Language** (DSEL). To implement schema evolution in the specific models we will use the DSEL for our main goal which is the **Multi-model Schema Evolution Language** (MMSEL). We decided to use this language as a commonly supported set of operations of specific models, because as the authors of the NoSQLSEL mentioned that these operations cover the most common schema migration tasks.

Another reason is that the DSEL is easy to implemented in all specific data models because it requires basic database functionality to store entities, remove entities, find entities by a key and conjunctive queries with equality comparisons by a key.

Before we introduce the MMSEL let us recapitulate the DSEL. It contains three main operations (now applicable on all considered data models):

- The *add* operation introduces a new property to all entities of a given kind with a specified default value.
- The *delete* operation removes a property from all entities of a given kind.
- The *rename* operation changes the name of a property for all entities of a given name.

All these operations affect all entities of one kind. For schema refactoring of existing entities we introduce the following operations:

- The *move* operation removes a property from one kind of entity and adds it to another one.
- The *copy* operation copies a property from one kind of entity to another one.

5.1.2 Multi-model Schema Evolution Language

Now we have a common interface supported by all our models so we can introduce the MMSEL. The functionality and provided operations by the language are crucial and they are our main goal. The logic of the language is executed in the model-independent layer which communicates with a model-specific layer through the common interface of the DSEL.

We decided to introduce the same set of operations as is provided by the DSEL, i.e. operations *add*, *delete*, *rename*, *move* and *copy* which will also support equality conditions by a key and joins (for moving and copying of data). Reasons are that this set of operations covers the most common cases of schema evolution as the authors of the NoSQLSEL claim. The second reason is the simplicity of the language. From the user perspective, the language is self-descriptive and thus easy to understand even without reading its specification which means that it is easy to learn it and use. The next reason is that its set of operations follows the common provided approach of schema evolution which we described in Section 3.9.3. On the other hand, we know about limitations of the NoSQLSEL which we mentioned during its presentation. Let us recapitulate quickly them. For operations *move* and *copy* the language requires relationship 1:N between the source property and its targets, otherwise the result of operations is not defined. The second limitation is that the language does not define the result of operations that add a new property to entities with the same key as already exists in the entities and the NoSQLSEL does not support nesting and unnesting of migrating entities. However, the benefits of the NoSQLSEL still outweigh its limitations and thus we use it for the MMSEL.

The result of this decision is that the MMSEL has the same limitations as the NoSQLSEL.

Our approach follows the idea of versioning of entities so it uses an extra numeric property *version* for all entities to detect their state. The property is incremented each time an entity is modified during any evolution. Moreover, the property makes the migration robust in case of interruptions by restricting migrations to all entities of a particular version. This can be specified using the where-clause and we may correctly recover from interrupts, even for move and copy operations.

The extension of the operations for MMDs requires specification of processing of commands in the model independent layer. The multi-model engine has to distinguish which models are affected by a given operation and propagate the operations to them. Let us introduce the operations in the layer-based MMD and specify the respective behavior.

We can divide the operations into two separate groups by the number of affected models:

- **Intra-model** operations – Operations which affect just one model. This is just operation *add*. As we can see from the definition of this function, it adds a new property so it cannot create any inconsistency between models or require data transformation between models. Figure 5.3 shows a simple activity diagram of processing operation *add*. In the diagram, we can see operation *add* requested by a client. The request is handled by the multi-model engine and delegated to the specific model. In case of operation *add* it cannot trigger any other operations.

- **Inter-model** operations – Operations which can affect multiple models. This is the rest of the mentioned operations: *copy*, *move*, *delete* and *rename*. The first two inter-model operations can trigger data transfer between models and all four operations can trigger reference changes in other models. We will use the following terminology for affected models: the **source model** is a specific model which contains original data and the **target model** is a specific model which receives the data. Figure 5.3 depicts an activity diagram when the multi-model engine receives operation *delete* or *rename* from a client and delegates it to a specific source model. These operations could trigger reference changes in multiple target models. These changes are handled by the engine. Figure 5.4 depicts an activity diagram when the engine

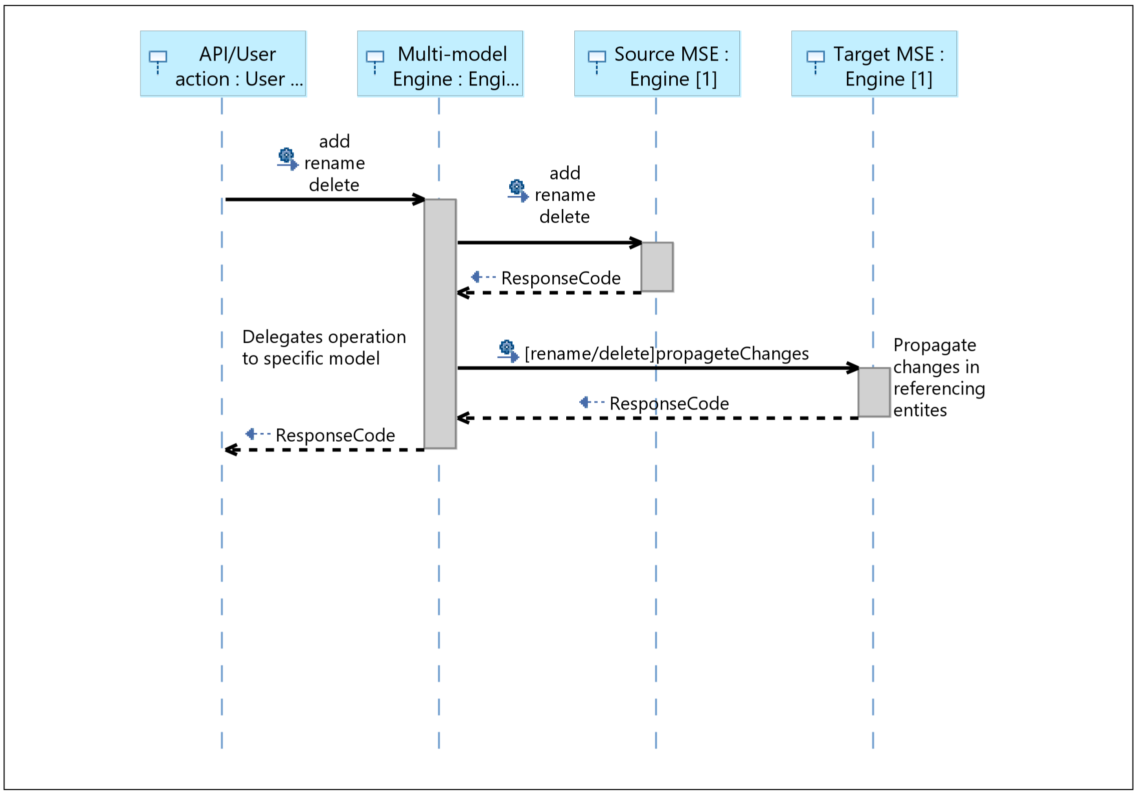

API/User action : User ...
Multi-model Engine : Engi...
Source MSE : Engine [1]
Target MSE : Engine [1]
add rename delete
add rename delete
ResponseCode
Delegates operation to specific model
[rename/delete]propageteChanges
Propagate changes in referencing entites
ResponseCode
ResponseCode

Figure 5.3: Interaction diagram of *add*, *delete* and *rename* operations.

receives operation *move* or *copy* from a client. The operation is delegated to a specific source model which contains the original data. The data are loaded to the engine. The engine sends them to the specific target model (which can be the same as the source model). Both operations can also trigger reference changes which are handled by the engine.

This classification can simplify our problem because intra-model operations affect only one model. It means that the multi-model engine in model independent layer propagates intra-model operations to specific target models which are already able to ensure correct data processing because we build the operations on top of existing operations for specific models. Intra-model operations just have to distinguish the right specific model and propagate the given requests.

The set of inter-model operations can affect multiple models. When the operations process entities within the same model (i.e., the source model equals to the target model), it is the same situation like in the case of an intra-model operation and the specific model can handle the given request. The problem is when entities should be moved from a source model to a different target model. In this case, the multi-model engine has to handle the logic. The engine gets all entities of the

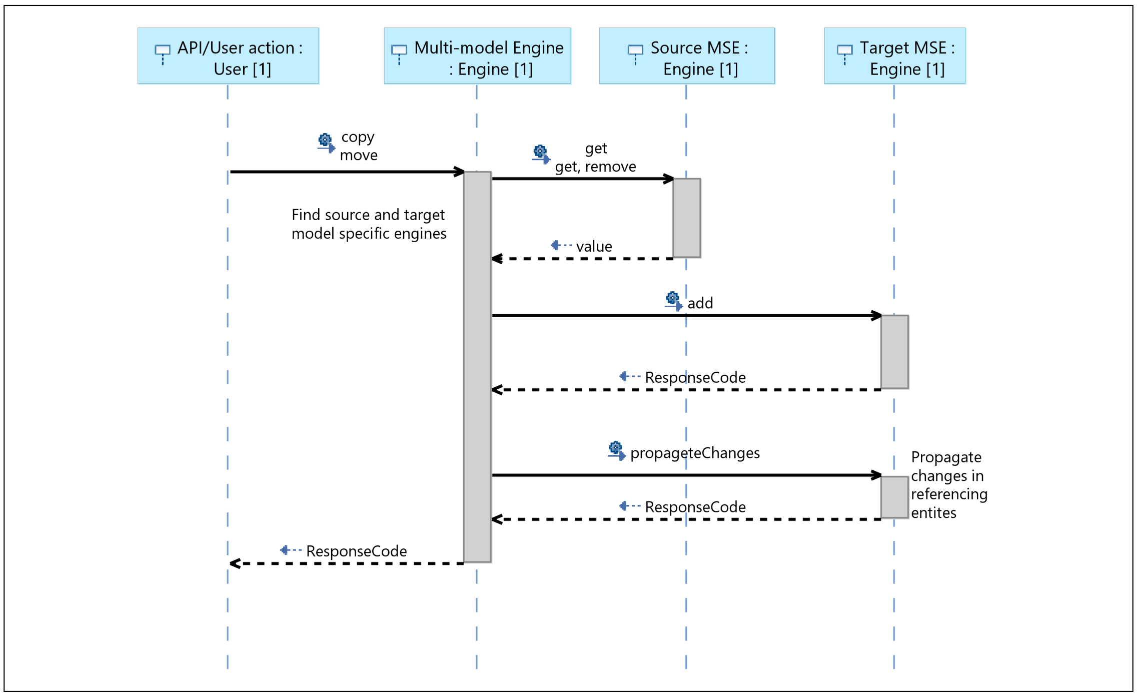

API/User action : User [1]
Multi-model Engine : Engine [1]
Source MSE : Engine [1]
Target MSE : Engine [1]
copy
move
get
get, remove
Find source and target model specific engines
value
add
ResponseCode
propageteChanges
Propagate changes in referencing entites
ResponseCode
ResponseCode

Figure 5.4: Interaction diagram of *move* and *copy* operations.

given kind from the source model and inserts them in the target model. In case of operation *move*, it has to delete them from the source model. The second problem of inter-model operations is propagation of changes to models which reference the changed entities. This logic has to be handled in the multi-model engine which is able to track all cross-model references. When the engine detects a change of referenced entity it also has to manage appropriate changes in the referencing entity. It can be renaming of a property in case of operation *rename* or deleting in case of *delete* or *move*.

The problematic of the reference evolution makes the schema evolution more complex. We have decided to ignore references at this moment and introduce a solution for schema evolution in MMDs without taking of references. We will extend our solution with references later. The reason is to make the evolution language clear and easy to understand. The result of this decision is that operations *rename* and *delete* are intra-model.

Now we have a high-level overview of operations of MMSEL and we can specify them in more details.

Syntax of MMSEL

Figure 5.5 shows the syntax of the MMSEL in the EBNF. The language allows us to use equality predicates between keys and joins. The conditions follow the

```
mmevolutionop ::= add | delete | rename | move | copy;

add ::= "add" property "=" value [selection];
delete ::= "delete" property [selection];
rename ::= "rename" property "to" pname [selection];
move ::= "move" property "to" kind [complexcond];
copy ::= "copy" property "to" kind [complexcond];

selection ::= "where" conds;
complexcond ::= "where" (joincond | conds | (joincond "and" conds));

joincond ::= property "=" property;
conds ::= cond {"and" cond};
cond ::= property "=" value;

property ::= mname "." kname ["." pname];
kind ::= mname "." kname;
mname ::= identifier;
kname ::= identifier;
pname ::= identifier;
```

Figure 5.5: Syntax of MMSEL in EBNF.

definition of the query model from Section 2.3.5. The result is that our language supports only conjunctive queries with equality comparisons between entity keys which are commonly provided by data stores. The property models (mname), the property kinds (kname) and the property names (pname) are terminals in the grammar. We will formally specify the semantics of all our operations later, because we want to build an intuition for this language first. Let us introduce the syntax of the language and behavior of each operation. If we compare the MMSEL with the NoSQLSEL, we extended its logic with the model identifier to distinguish the target model and an optional property name because not all models support properties (e.g. the key/value model). In general, we want to keep the language as similar as possible to make it easier for a developer to learn a simple extension of already known language.

To develop an intuition of the language and its operations we provide simple examples. We use the model of the web application from Chapter 4 which is depicted in Figure 4.1. It supports relational, key/value and document models.

Example. The original state of our MMD (Figure 5.6) contains a set of users represented in a relational model, a set of items represented in a document model and a set of carts represented in a key/value model. Users are stored in table *user(id, name, address)*, items are teas with compulsory properties *name*, *type* and *price*. We can also see optional properties *country* and *alias*. Shopping-carts are represented as key *cart:userId* and value corresponding to the list of pairs *productId, quantity.* (I.e., we consider that our key/value model supports a structured value which allows us to store the list of pairs.) The key/value model also contains other entities like current web application version and the name of the seller.

Now, we can execute examples of the MMSEL operations:

- Operation *add*: In the example bellow, we show application of operation *add* on an entity in the document model from our web application example. Operation **add document.tea.importer** = **"Tea Comp."** adds property *importer* with value *Tea Comp.* to all entities of kind *tea.*

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "version": 1
    }
  ]
}
```

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "importer": "Tea Comp.",
      "country": "China",
      "version": 2
    }
  ]
}
```

  We can also specify a selection predicate. Operation **add relational.users.canDeliver** = ***true*** **where relational.users.address** = **null** adds new column *canDeliver* on the basis of the value of property *relational.users.address*. The rest of the entities have an undefined value.

id	**name**	**address**	**v...**
1	Peter Parker	15010...	1
2	John Doe	null	1

id	**...**	**v...**	**canDeliver**
1	...	2	null
2	...	2	true

- Operation *delete*: Operation **delete document.tea.country** deletes property *country* from all entities *tea.*

(a) Data in document storage.

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China"
     },
     {
       "id": 1,
       "name": "Longjing",
       "type": "green",
       "price": 15,
       "alias": "Dragon's Well"
     },
     {
       "id": 2,
       "name": "Keemun",
       "type": "black",
       "price": 11
     }
  ]
}
```

(b) Data in relational storage.

id	name	address
1	Peter Parker	15010 NE 36th Street Redmond, WA 98052
2	John Doe	null

(c) Data in key/value storage.

key	value
cart:1	[1,5;0;2]
cart:2	[2,1]
appVersion	teaShop
seller	eTea Shop

Figure 5.6: An example of multi-model storage for a web application.

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "version": 1
    }
  ]
}
```

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "version": 2
    }
  ]
}
```

- Operation *rename*: Operation **rename relational.users.name to fullname** renames property *name* to *fullname* for all entities *tea* in the relational model.

id	name	address	v...
1	Peter Parker	15010...	1
2	John Doe	null	1

id	fullname	address	v...
1	Peter Parker	15010...	2
2	John Doe	null	2

- Operation *copy*: To copy a property from one model to a different one we use operation *copy*. Operation **copy keyValue.appVersion to document.tea** copies property *appVersion* for all entities *tea* in the document model.

key	value
cart:1	[1,5;0;2]
cart:2	[2,1]
appVersion	teaShop
seller	eTea Shop

key	value
cart:1	[1,5;0;2]
cart:2	[2,1]
appVersion	teaShop
seller	eTea Shop

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "version": 1
    }
  ]
}
```

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "appVersion": "teaShop",
      "version": 2
    }
  ]
}
```

- Operation *move*: The last example is operation *move*: Property *seller* from the key/value model is moved to each entity *tea* in the document model: **move keyValue.seller to document.tea**

key	value
cart:1	[1,5;0;2]
cart:2	[2,1]
appVersion	teaShop
seller	eTea Shop

key	value
cart:1	[1,5;0;2]
cart:2	[2,1]
appVersion	teaShop

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "version": 1
    }
  ]
}
```

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "price": 10,
      "country": "China",
      "seller": "eTea Shop",
      "version": 2
    }
  ]
}
```

5.1.3 Semantics of the MMSEL

Now, we can implement all multi-model schema evolution operations from Figure 5.5.

Semantics of DSEL

The semantics of the DSEL has to be discussed for each data model. The document and column models are defined by the implementation of these operations for NoSQL schema evolution introduced by S. Scherzinger, M. Klettke, and U. Störl in [63] (see Algorithm 5.1 and Algorithm 5.2). These operations use the NoSQLDPL. For the graph model, we consider representation as documents so the NoSQLDPL covers also this model. We therefore need a definition for the key/value and relational models.

Algorithm 5.1: Implementation of NoSQL schema evolution operations *add*, *delete*, and *rename* for the document and column models [63].

```
Let $c$ be a kind, let $n$ be a property name, and let $v$ be a property value
  from $Dom$. $\theta$ is a conjunctive query over properties.

add $c.n = v$ where $\theta$
foreach $element\ e\ of\ get(kind = c \wedge \theta)$ do
    setProperty($e$, $n$, $v$);
    setProperty($e$, $version$, getProperty($e$, $version$) + 1);
    put($e$);

delete $c.n$ where $\theta$
foreach $element\ e\ of\ get(kind = c \wedge \theta)$ do
    removeProperty($e$, $n$);
    setProperty($e$, $version$, getProperty($e$, $version$) + 1);
    put($e$);

rename $c.n$ to $m$ where $\theta$
foreach $element\ e\ of\ get(kind = c \wedge \theta)$ do
    setProperty($e$, $m$, getProperty($e$, $n$));
    removeProperty($e$, $n$);
    setProperty($e$, $version$, getProperty($e$, $version$) + 1);
    put($e$);
```

Algorithm 5.2: Implementation of NoSQL schema evolution operations *move*, and *copy* for the document and column models [63].

```
Let $c_1, c_2$ be kinds and let $n$ be a property name. Conditions $\theta_1$ and $\theta_2$ are
  conjunctive queries. $\theta_1$ has atoms of the form $c_1.m = v$, where $m$ is a
  property name and $v$ is a value from $Dom$. $\theta_2$ has atoms of the form
  $c_2.m = v$ or $c_1.a = c_2.b$, where $a$, $b$ and $m$ are property names. $v$ is a
  value from $Dom$.

move $c_1.n$ to $c_2$ where $\theta_1 \wedge \theta_2$
foreach element $e$ of get($kind = c_1 \wedge \theta_1$) do
    foreach element $f$ of get($kind = c_2 \wedge \theta_2$) do
        setProperty($f$, $n$, getProperty($e$, $n$));
        setProperty($f$, $version$, getProperty($f$, $version$) + 1);
        put($f$);
    setProperty($e$, $version$, getProperty($e$, $version$) + 1);
    removeProperty($e$, $n$);
    put($e$);

copy $c_1.n$ to $c_2$ where $\theta_1 \wedge \theta_2$
foreach element $e$ of get($kind = c_1 \wedge \theta_1$) do
    foreach element $f$ of get($kind = c_2 \wedge \theta_2$) do
        setProperty($f$, $n$, getProperty($e$, $n$));
        setProperty($f$, $version$, getProperty($f$, $version$) + 1);
        put($f$);
```

First, we define the operations for the key/value model. In this model, we do not have any properties as in the document model. We have only keys and values, i.e. entities and values. As a result, we have to exclude all conditions from this model because they are irrelevant. In Section 5.1.2 we mentioned that properties in the MMSEL are optional and we will use it right now. Algorithm 5.3 defines the behavior of these operations for the key/value model.

Let us explain the defined functions in the context of key/value model. The first defined function in the algorithm is operation *add*. In the first step it creates a new entity with the given key c_1 and value v in an application space. In the second and last step the created entity is stored in the database. The second defined function is *delete* which deletes an entity by the given key c_1 from the database. The last defined operation *rename* creates a new entity with the given key c_2 and a value of entity c_1 in the application space. Then the entity is stored in the database and the original entity is deleted. We choose this approach because we can add and remove the entity in all key/value databases, whereas renaming the key of an entity is not always supported.

Algorithm 5.3: Implementation of NoSQL schema evolution operations *add*, *delete*, and *rename* for the key/value model.

```
Let $c_1$, $c_2$ be kinds and $v$ a value from $Dom$.

add $c_1 = v$
new($c_1$, $v$);
put($c_1$);

delete $c_1$
delete($c_1$);

rename $c_1$ to $c_2$
new($c_2$, get($c_1$));
put($c_2$);
delete($c_1$);
```

The next consequence of missing properties is that the key/value model does not support operation *move*. We decided to exclude the operation because we cannot properly defined it without the support of properties in the model. It could be defined in the same way as operation *rename*, but we do not want to have two operations with the same behavior. For that reason, operation *move* in the key/value model does nothing. To be able to implement the operation with no functionality, we need to extend the NoSQLDPL. The new function is called *empty*. It does not modify the application space or even the database state. It means the function does not modify any data in application memory, even not in the database. We formally specify the function later with the implementation of MMSEL. Now, we introduce the remaining functions for the key/value model in Algorithm 5.4. Operation *move* calls the new function *empty*. The second operation *copy* creates a new entity with the given key c_2 and value of entity c_1 then the entity is put in the database.

Algorithm 5.4: Implementation of NoSQL schema evolution operations *move* and *copy* for the key/value model.

```
Let $c_1$, $c_2$ be kinds and $v$ a value from $Dom$.

move $c_1$ to $c_2$
empty();

copy $c_1$ to $c_2$
new($c_2$, get($c_1$));
put($c_2$);
```

The relational model is straightforward because we basically defined it in Section 5.1.1. Instead of transforming operations to the NoSQLDPL which is derived from the relational world, we used SQL to show the semantics of each function of the DSEL in the relational model.

Now, we have all models covered with our common interface language DSEL. So we covered the whole model-specific layer and we can deal with the MMSEL for the model-independent layer.

Implementation of MMSEL

The core logic of the MMSEL happens in the model independent layer. This layer is represented by a multi-model engine. The engine has to be able to distinguish between each model and communicate with them. In our solution the communication is provided by a unified interface of the DSEL. To be able to distinguish between the models we introduce the **Data Model Set** (DMS). The DMS is a set of data models presented in MMD which implements the DSEL. In our case the set is: $\{column, document, key/value, graph, relational\}$.

The engine has to be able to manipulate with the DMS. The most needed functionality is to choose the correct model based on its name. For easier access and management of the set, we introduce a **model key** δ: $\delta \in DMS$. We use the keys of the DMS for referencing the real-world models in the MMD. To create an abstract model of the MMD we follow the notation from [63] which uses the term **application state** for the current state of the application space. It is a non-persistent application memory. The second used term is **database state** which the current state of the database and it represents all stored data in our database at the moment. The main difference compared to the application state is that the database state is persistent.

Figure 5.7 defines the **Multi-Model Database Programming Language** (MMDPL) and its operations over the DMS. The MMDPL implements function

Let *dms* be a DMS, *ds* be a database state, *as* be an application state, δ be a model key, and Ω be a data model. Symbol $\perp$ denotes an undefined value.

$$[\![empty()]\!](dms, ds, as) = (dms, ds, as) \tag{5.1}$$
$$[\![addModel(\delta, \Omega)]\!](dms, ds, as) = (dms, ds, as\,[\delta \mapsto \Omega]) \tag{5.2}$$
$$[\![putModel(\delta)]\!](dms, ds, as \cup \{\delta \mapsto \Omega\}) = (dms\,[\delta \mapsto \Omega]\,, ds, as \cup \{\delta \mapsto \Omega\}) \tag{5.3}$$
$$[\![deleteModel(\delta)]\!](dms, ds, as) = (dms\,[\delta \mapsto \perp]\,, ds, as) \tag{5.4}$$
$$[\![getModel(\delta)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as\,[\delta \mapsto \Omega]) \tag{5.5}$$

Figure 5.7: Functions of the MMDPL for managing data models (Part 1)

empty for the key/value model (Rule 5.1). Function *empty* does not modify the application space or the database. In Rule 5.2 we introduce function $addModel(\delta, \Omega)$ which creates a new entity in application space with key δ which points to model Ω. To be able to store the created model we introduce function $putModel(\delta)$ in Rule 5.3. It stores the entity with key δ from application space to the DMS. Rule 5.4 defines function $deleteModel(\delta)$ which removes a model with key δ from the DMS. Now, we have the full control of models in MMD. Our main goal is the operation $getModel(\delta)$ (Rule 5.5) that we need for implementation of the evolution-management language. This function loads the model by the given key δ to the application space.

Note that we introduce the functions for the full control of models in the DMS, but there will be always five models and we are not going to create new models or remove any of them.

We also need to call specific schema evolution functions in specific models. To ensure it, we introduce a modified set of functions called MMDPL which extends the NoSQLDPL with DMS operations (Figure 5.8 and Figure 5.9). The difference is in operations for getting entities from the database and to save them in the database. The rest of the operations work in the application memory so there is no need to modify them, we just extend them with the DMS but their logic remains. Let us describe the changed functions in more detail. Rule 5.11 extends function $put(\delta, \kappa)$ by parameter δ to distinguish where the entity with the key κ is stored. For that purpose we introduce function $model(\kappa)$ which returns a model where the entity occurs. We use this approach to detect the affected model in all modified functions. In Rule 5.12 we extend function $delete(\delta, \kappa)$ by key of the model δ which contains the entity with key κ. Rule 5.13, Rule 5.14, and Rule 5.15 add parameter δ to function *get*. All modified functions *get* load entity/entities from the specified model by key δ to the application space. Rule 5.16 is also

Let *dms* be a DMS, *ds* be a database state, *as* be an application state, δ be a model key, and Ω be a data model. Let κ, κ' be entity keys. Let n, n' be property names, and let v be a property value. Symbol $\perp$ denotes an undefined value. Let π, π' be properties, i.e. mappings from property names to property values. *kind* : *Keys* $\mapsto$ *Kind* is a function that extracts the entity kind from a key. *model* : *Modelkeys* $\mapsto$ *Datamodel* is a function that extracts the entity model from a key. Θ is a conjunctive query, and c is a string constant.

$$[\![new(\kappa)]\!](dms, ds, as) = (dms, ds, as[\kappa \mapsto \emptyset]) \tag{5.6}$$

$$[\![new(\kappa, \pi)]\!](dms, ds, as) = (dms, ds, as[\kappa \mapsto \pi]) \tag{5.7}$$

$$\begin{aligned}[\![setProperty(\kappa, n, v)]\!]&(dms, ds, as \cup \{\kappa \mapsto \pi\}) = \\ &(dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto v])\})\end{aligned} \tag{5.8}$$

$$\begin{aligned}[\![setProperty(\kappa, n, \kappa')]\!]&(dms, ds, \\ as \cup \{\kappa \mapsto \pi\} &\cup \{\kappa' \mapsto \pi'\}) = \\ &(dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \pi'])\} \cup \{\kappa' \mapsto \pi'\})\end{aligned} \tag{5.9}$$

$$\begin{aligned}[\![removeProperty(\kappa, n)]\!]&(dms, ds, as \cup \{\kappa \mapsto \pi\}) = \\ &(dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \perp])\})\end{aligned} \tag{5.10}$$

$$\begin{aligned}[\![put(\delta, \kappa)]\!]&(dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto \pi\}) = \\ &(dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \pi \mid \\ &\quad model(\kappa) = \delta\}], as \cup \{\kappa \mapsto \pi\})\end{aligned} \tag{5.11}$$

Figure 5.8: Functions of the MMDPL (Part 2)

extended by model key δ to load the property from the specified model.

Now we have the MMDPL and we can focus on implementing of functions of the MMSEL from Section 5.1.2.

Algorithm 5.5 shows the implementation of operation *add*. The operation finds the correct model and checks if the model exists. In the next step the algorithm iterates all entities of the given kind c in the model m which match query θ and inserts property n with value v to them. Also it increments property version for all affected elements and stores them back to the database.

Let *dms* be a DMS, *ds* be a database state, *as* be an application state, δ be a model key, and Ω be a data model. Let κ, κ' be entity keys. Let n, n' be property names, and let v be a property value. Symbol $\perp$ denotes an undefined value. Let π, π' be properties, i.e. mappings from property names to property values. $kind : Keys \mapsto Kind$ is a function that extracts the entity kind from a key. $model : Modelkeys \mapsto Datamodel$ is a function that extracts the entity model from a key. Θ is a conjunctive query, and c is a string constant.

$$\begin{aligned}&[\![delete(\delta,\kappa)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as) = \\ &\qquad (dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \perp \mid model(\kappa) = \delta\}], as)\end{aligned} \tag{5.12}$$

$$\begin{aligned}&[\![get(\delta,\kappa)]\!](dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds \\ &\qquad as \cup [\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge model(\kappa) = \delta\}])\end{aligned} \tag{5.13}$$

$$\begin{aligned}&[\![get(\delta, kind = c)]\!](dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, \\ &\qquad ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \\ &\qquad \wedge model(\kappa) = \delta\}])\end{aligned} \tag{5.14}$$

$$\begin{aligned}&[\![get(\delta, kind = c \wedge \emptyset)]\!](dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, \\ &\qquad ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \\ &\qquad \wedge model(\kappa) = \delta \wedge [\![\emptyset]\!](\kappa \mapsto \pi)\}])\end{aligned} \tag{5.15}$$

$$\begin{aligned}&[\![getProperty(\delta,\kappa,n)]\!](dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto (\{n \mapsto v\} \cup \pi) \mid \\ &\qquad \kappa \mapsto (\{n \mapsto v\} \cup \pi) \in ds \wedge model(\kappa) = \delta\}) = v\end{aligned} \tag{5.16}$$

Figure 5.9: Functions of the MMDPL (Part 3)

Algorithm 5.5: Intra-model operation *add* of MMSEL

Legend: Let m be a model name, let c be a kind, let n be an optional property name, and let v be a property value from *Dom*. θ is a conjunctive query over properties.

```
add $m.c[.n] = v$ where $\theta$
if $getModel(m) \neq \perp$ then
    foreach element $e$ of $get(m, kind = c \wedge \theta)$ do
        setProperty($e$, $n$, $v$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        put($m$, $e$);
```

Algorithm 5.6 shows implementation of operations *delete* and *rename*. The core idea is the same as the idea of operation *add*. Both operations check the existence of the affected model m and then call the operation for the specific model. Operation *delete* removes property n_1 and operation *rename* creates a new property n_2 with value of property n_1 from all elements of kind c in model m which match query θ. Both operations increment property version for all affected elements and store them in the database.

Finally, Algorithm 5.7 defines an implementation of inter-model operations *move* and *copy*. Both operations check existence of models m_1 and m_2. In the next step both operations duplicate value of property n_1 for all elements e of kind c_1 in model m_1 which match query θ_1 to property n_1 of elements f of kind c_2 in model m_2 which match query θ_2. During the duplication the version property of f is incremented and elements are stored. Operation *move* also removes property n_1 from affected elements e and increments its version property.

Algorithm 5.6: Operations *delete* and *rename* of the MMSEL

Legend: Let m be a model name, let c be a kind, let n_1, n_2 be property names, and let v be a property value from *Dom*. θ is a conjunctive query over properties.

```
delete $m.c.n_1$ where $\theta$
if $getModel(m) \neq \perp$ then
    foreach element $e$ of $get(m, kind = c \wedge \theta)$ do
        removeProperty($e$, $n_1$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        put($m$, $e$);

rename $m.c.n_1$ to $n_2$ where $\theta$
if $getModel(m) \neq \perp$ then
    foreach element $e$ of $get(m, kind = c \wedge \theta)$ do
        setProperty($e$, $n_2$, getProperty($e$, $n_1$));
        removeProperty($e$, $n_1$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        put($m$, $e$);
```

Algorithm 5.7: Operations *move* and *copy* of the MMSEL

Legend: Let m_1, m_2 be model names, let c_1, c_2 be kinds, let n_1 be a property name, and let v be a property value from *Dom.* θ_1, θ_2 are conjunctive queries over properties where θ_1 has atoms of the form $m_1.c_1.n_1 = v$, where n_1 is a property name and v is a value from *Dom.* θ_2 has atoms of the form $m_2.c_2.n_2 = v$ or $m_1.c_1.a = m_2.c_2.b$, where a, b and n_2 are property names.

```
move $m_1.c_1.n_1$ to $m_2.c_2$ where $\theta_1 \wedge \theta_2$
if $(getModel(m_1) \neq \bot) \wedge (getModel(m_2) \neq \bot)$ then
    foreach element $e$ of $get(m_1, kind = c_1 \wedge \theta_1)$ do
        foreach element $f$ of $get(m_2, kind = c_2 \wedge \theta_2)$ do
            setProperty($f$, $n_1$, getProperty($e$, $n_1$));
            setProperty($f$, $version$, getProperty($f$, $version$) + 1);
            put($m_2$, $f$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        removeProperty($e$, $n_1$);
        put($m_1$, $e$);

copy $m_1.c_1.n_1$ to $m_2.c_2$ where $\theta_1 \wedge \theta_2$
if $(getModel(m_1) \neq \bot) \wedge (getModel(m_2) \neq \bot)$ then
    foreach element $e$ of $get(m_1, kind = c_1 \wedge \theta_1)$ do
        foreach element $f$ of $get(m_2, kind = c_2 \wedge \theta_2)$ do
            setProperty($f$, $n_1$, getProperty($e$, $n_1$));
            setProperty($f$, $version$, getProperty($f$, $version$) + 1);
            put($m_1$, $f$);
```

Optimized Implementation of MMSEL

Note that, we showed the most obvious implementation of the MMSEL which is easy to understand but less effective. Our implementation could be more effective if we used the option to execute schema evolution operations within specific models. The current implementations always load an entity to application memory which is in the model-independent layer. It means we always have to load an entity even during intra-model operations. We know that all our models implement DSEL so we can execute inter-model migrations inside a specific model. We can also optimize execution of potentiality inter-model operations. For example, when operations *move* or *copy* migrate data inside the same model, we can call a migration operation within the specific model. For the calling of migration operations in a specific model we use a self-explanatory pseudo code. In Algorithm 5.8 we optimize two intra-model operations *add* and *delete* and the inter-model operation *move*. The intra-model operations check existence of model m_1 and if it exists, then delegate operation to the specific model m_1. The algorithm also optimizes operation *move* which checks existence of both affected models m_1 and m_2. Then, in case of the model m_1 is equal to m_2, the operation is delegated to the specific model, else it executes the same non-optimized logic as we introduced in Algorithm 5.7. In Algorithm 5.9 we show the optimized intra-model operation *rename* and the inter-model operation *copy*. The operations follow the same logic as the previous optimized operations. In case of the intra-model operation it checks existence of model m_1 and if the model exists, then it delegates the operation to the specific model. The inter-model operations checks existence of models m_1 and m_2 and then, if they are equal, it delegates the operation *copy* to the specific model. If they are not equal, non-optimized logic is executed.

Algorithm 5.8: Optimized operations of the MMSEL (Part 1)

Legend: Let m_1, m_2 be model names, let c_1, c_2 be kinds, let n_1 be a property name, and let v be a property value from *Dom.* θ_1, θ_2 are conjunctive queries over properties where θ_1 has atoms of the form $m_1.c_1.n_1 = v$, where n_1 is a property name and v is a value from *Dom.* θ_2 has atoms of the form $m_2.c_2.n_2 = v$ or $m_1.c_1.a = m_2.c_2.b$, where a, b and n_2 are property names.

```
add $m_1.c_1[.n_1] = v$ where $\theta_1$
if $getModel(m_1) \neq \perp$ then
    $m_1$.execute(add $c_1[.n_1] = v$ where $\theta_1$);

delete $m_1.c_1.n_1$ where $\theta_1$
if $getModel(m_1) \neq \perp$ then
    $m_1$.execute(delete $c_1.n_1$ where $\theta_1$);

move $m_1.c_1.n_1$ to $m_2.c_2$ where $\theta_1 \wedge \theta_2$
if $(getModel(m_1) \neq \perp) \wedge (getModel(m_2) \neq \perp)$ then
    if $m_1 = m_2$ then
        $m_1$.execute(move $c_1.n_1$ to $c_2$ where $\theta_1 \wedge \theta_2$);
    else
        foreach element $e$ of $get(m_1, kind = c_1 \wedge \theta_1)$ do
            foreach element $f$ of $get(m_2, kind = c_2 \wedge \theta_2)$ do
                setProperty($f$, $n_1$, getProperty($e$, $n_1$));
                setProperty($f$, $version$, getProperty($f$, $version$) + 1);
                put($m_2$, $f$);
            setProperty($e$, $version$, getProperty($e$, $version$) + 1);
            removeProperty($e$, $n_1$);
            put($m_1$, $e$);
```

Algorithm 5.9: Optimized operations of the MMSEL (Part 2)

Legend: Let m_1, m_2 be model names, let c_1, c_2 be kinds, let n_1 be a property name, and let v be a property value from *Dom.* θ_1, θ_2 are conjunctive queries over properties where θ_1 has atoms of the form $m_1.c_1.n_1 = v$, where n_1 is a property name and v is a value from *Dom.* θ_2 has atoms of the form $m_2.c_2.n_2 = v$ or $m_1.c_1.a = m_2.c_2.b$, where a, b and n_2 are property names.

```
rename $m_1.c_1.n_1$ to $n_2$ where $\theta_1$
if $getModel(m_1) \neq \perp$ then
    $m_1$.execute(rename $c_1.n_1$ to $n_2$ where $\theta_1$);

copy $m_1.c_1.n_1$ to $m_2.c_2$ where $\theta_1 \wedge \theta_2$
if $(getModel(m_1) \neq \perp) \wedge (getModel(m_2) \neq \perp)$ then
    if $m_1 = m_2$ then
        $m_1$.execute(copy $c_1.n_1$ to $c_2$ where $\theta_1 \wedge \theta_2$);
    else
        foreach element $e$ of $get(m_1, kind = c_1 \wedge \theta_1)$ do
            foreach element $f$ of $get(m_2, kind = c_2 \wedge \theta_2)$ do
                setProperty($f$, $n_1$, getProperty($e$, $n_1$));
                setProperty($f$, $version$, getProperty($f$, $version$) + 1);
                put($m_1$, $f$);
```

Robustness of MMSEL

Our implementation of the MMSEL is also robust because of using the version property. We can introduce control mechanisms which can recover the database in case of a failure. The recovery means to finish the rest of interrupted migrations. The idea of the implementation is to have a persistent action log which contains all evolution operations executed in the MMD. The MMD logs all migrations start and end events with the current version of the migrating entity. The inconsistency can happen when the MMD is interrupted during execution of the migration (e.g. a shutdown of the database). The MMD checks if the log contains any start log without the end. If the log is found we can find all entities which are not migrated using a where condition with the version property number. If the version is the same as when the start event was logged it means it is not migrated yet.

Last but not least, note that we showed the eager implementation of the language, because it is easier to understand this scenario end-to-end. We keep the lazy implementation as an open challenge.

5.1.4 Reference Evolution in MMDs

We introduced operations of the MMSEL which, however, ignore possible references in MMD. It is not in the scope of the thesis but we have decided to suggest an idea of a solution how to integrate reference evolution.

We built our solution of schema evolution in MMDs on top of the NoSQLSEL. Unfortunately, the authors of the language do not consider references at all, because most of the NoSQL databases do not support references. Developers rather use data denormalization which creates duplicity of data or storing a key of the referenced entity and managing the reference in the application. Actually, the denormalization is a recommended approach and a best-practice for often read data. For the rest it is storing of a key in referencing entites. The recommendations are by authors of the database (e.g. [59]) or by users (e.g. [13]) to ensure better performance and usability. The former approach creates a full copy of the referenced data which is completely separated and independent from the source of the data. Future changes in the source data have to be propagated manually to all copies. We focus on the latter approach which uses keys as references to other entities. We can easily manage the stored reference keys and we do not have to copy whole entities.

First of all, we introduce what we mean by the reference. We consider the reference as a pointer from a property of a referencing entity to a property of a referenced entity. So the reference is a relation between two properties of existing entities: *source references target* where each one of the source and target indicates a triple of a model, entity, and property.

We have to describe what can happen during each operation within a model. We can split operations into two groups by affected references: **safe** and **unsafe**. Safe operations do not trigger any reference updates and unsafe operations can. Let us discuss them one by one:

- Operation *add*: The operation introduces a new property which cannot be referenced from any other entity. *Add* is safe and cannot trigger any reference changes.

- Operation *delete*: The operation removes a property which can be referenced which means that it is unsafe.

- Operation *rename*: The operation can change the name of a property which is referenced by another property but the reference remains unchanged. We just have to make sure that the references are updated to the new key. We use the key for managing references so operation *rename* is unsafe.

- Operation *move*: The operation deletes a property in a source entity and creates the same property in a target entity. Naturally, the reference still exists and has to be updated to the new entity. In common cases operation *move* does not change the meaning of data but it optimizes data structure. For that reason we keep the references to the moved property unchanged so they have to be updated. The operation *move* is unsafe.

- Operation *copy*: The operation creates a new property with a copied value of the source property. The copied property is independent and the referencing entity can still point to the same property. So *copy* is safe.

Our focus is on unsafe operations which can trigger changes in, i.e., *delete*, *rename*, and *move*. This fact is valid for the DSEL but also for the MMSEL. We want to suggest the idea of the solution for the MMSEL and the solution for the DSEL is kept as an open challenge. We do not want to change the defined behavior of the DSEL because it can theoretically affect a function which we used for the MMSEL and there could be required extra changes. The second reason is that implementation in the DSEL is considered as an improvement and our solution can work without it. When the evolution of references in the DSEL is formally proven, then we will be able to use the same performance improvements of propagation operations to the specific models as we did for operations in the MMSEL.

We focus our solution on the MMSEL. It means, our suggested solution is done in the model independent layer for all references which are changed in the same or different specific model.

Now, we have a database representation of references. It can be represented as a tuple $\{source, target\}$. We have to be able to persist the references and thus we need in MMD at least one model which is able to store the relations in a **relation space**. The relation space contains a set of the references. We suggest two simple ways how to implement the relation space:

- key/value model – When the model supports complex values as arrays, we can store directly a list of tuples. Let $source_1$ and $source_2$ be keys referencing properties, $target_1$ and $target_2$ be the keys of referenced properties. Then the following representation can be used in the key/value model:

key	**value**
__rel	$\{[source_1, target_1], [source_2, target_2], ...\}$

- document or relational model – We can use a more complex representation than in the key/value model. We can aggregate referencing properties for the same referenced property. In a database, there could be multiple entities

```
mmevolutionop |= reference;

reference ::= "reference" property "by" property [selection];
```

Figure 5.10: Extension of the MMSEL in EBNF with references.

referencing the same target and this fact increases effectiveness because we can find all affected properties in a single property. To fit the MMDPL we represent a reference as an entity with property p which specifies the referenced property of the entity, property k key of property in the MMD, and property s being an array of data about referencing entities (m as a model, k as a kind, and p as a property). An example of possible implementation in document model is shown below:

```
{
  "__rel": {
    "document.tea.type": [
        "m": "document",
        "k": "document.tea",
        "p": "type",
        "s": [{"m": "relational", "k":"users", "p":"favTea"},
            ...]
        ],
        ...
    }
}
```

We will use the second approach for its better performance and because most implementations of MMDs support the relational or document model. It is also easier to work with a structured representation in the MMDPL. On the other hand, when we need to remove a reference we have to go through all referenced properties anyway to check if it is not referenced by the removed property. We define property s as an array (Dom^+) and hence we can use standard operations for arrays, such as operation *push* to add to the array or *delete* to remove an element from the array.

Now we know, how a reference is represented and we are able to persist it. First of all in Figure 5.10 we define an extension of MMSEL which allows us to register a reference in a system. Operation *reference* $property_1$ *by* $property_2$ registers a reference from $property_2$ to $property_1$. Before we start with definition of the operation let us discuss the behavior. We use an example of the tea shop in Figure 5.6.

- A sequence of operations *reference* a property and *delete* the referenced property deletes also the referencing property. Operation **reference docu-**

ment.tea.type by relational.users.favTea and operation **delete document.tea.type** delete property *type* from all entities *tea* and also property *favTea* is deleted.

id	name	favTea	ver.
1	Peter Parker	white	1
2	John Doe	green	1

id	name	ver.
1	Peter Parker	2
2	John Doe	2

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "type": "white",
      "version": 1
    }, ...
  ]
}
```

```
{
  "tea":[
    {
      "id": 0,
      "name": "Silver Needle",
      "version": 2
    }, ...
  ]
}
```

The example shows us how the reference works and how it is processed during delete operation.

By default, references can allow us to create a chain of references. For the sake of simplicity we forbid them because instead of the chain the developer can use a direct reference to the source property. We can do it because the developer can achieve the same results with a reference to the final values of the chain. Including of chains will make the solution more complex. For example, there is a problem of possible circles and also splits and joins in the chain caused when a property references or is referenced by multiple properties. Because of these reasons, we keep the chain of references as an open challenge and it is not a part of our suggested solution.

During our study of reference migration, we discovered that where conditions make the solution much more difficult. It is caused by the nature of MMDs which allow a user to move just a part of properties. This behavior can split, delete or move completely an existing reference based on the affected set of the values. Since this area is beyond the scope of this thesis we introduce a solution for operations without conditions and keep it as an open challenge. Now, we can discuss a behavior when a referenced property is removed. We excluded the where conditions so we know that the deletion affects all referencing properties. We have two options what can happen with the referencing property: (1) set to default value, (2) delete the property. We decided to use the second approach because it is clear solution for the used models. The first approach has to define what should be the behavior when MMD contains entity without referencing property, then it has to define default values for all models, and the default value can be considered as a value of the property in an application so it can be a confusing for developers.

The next step defines operations for creating and managing references in the MMDPL. We need to have an opportunity to create a reference, store it, remove it and find it. Let **Reference Store Model** (RSM) be a store which is able to persists the reference entities. We use rhe RSM to extend the MMDPL and define functions which help us to implement reference management in the MMSEL. Figure 5.11 shows the extension of the MMDPL which provides functions for the mentioned operations. We introduced a special type of entities for the references which is stored in the RSM but we work with the type in the same way as with other database entities in the application space. The reference entities use a different set of keys to be able to easily distinguish between them in the application space. Rule 5.17 creates a new reference entity in the application space for the targeted key κ_1 which has to be filled in the application space. Rule 5.18 stores the entity ρ_1 in the RSM. Rule 5.19 loads a reference entity ρ_1 by the given database key κ_1 to the application space. Rule 5.20 loads all reference entities which are referenced by the given key c_1, n_1 of the model δ to the application space. Rule 5.21 removes the reference entity for the given database key κ_1 from the RSM. Rule 5.22 renames the reference entity for database key κ_1 to κ_2.

The set of operations allow us to create, delete, get, or rename a reference. In our use-cases, we need to be able to register a new reference, find out if the property is referenced by another one or if it is referencing another entity, and remove a reference.

Now, we can introduce operation *reference* and modified operations *delete*, *rename*, and *move*.

Let rs be a RSM, ρ_1,ρ_2 be keys in rs and η_1 its set of properties. Let dms be a DMS, ds be a database state, as be an application state. Let κ_1, κ_2 be entity keys. Let n_1 be property name. Let δ be a model key, c be a kind and ν be an array of triples of m, k, and p. Symbol $\perp$ denotes an undefined value. Let π_1 be properties, i.e. mappings from property names to property values. key : $RSM\ keys \mapsto model\ keys$ is a function that extracts the entity key from a reference store model key.

$$\begin{aligned}[\![newReference(\kappa_1)]\!](dms, ds, as, rs) = \\ (dms, ds, as[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}], rs)\end{aligned} \tag{5.17}$$

$$\begin{aligned}[\![putReference(\rho_1)]\!](dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs) = \\ (dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs[\rho_1 \mapsto \eta_1])\end{aligned} \tag{5.18}$$

$$\begin{aligned}[\![getReference(\kappa_1)]\!](dms, ds, as, rs) = \\ (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge key(\rho_1) = \kappa_1\}], rs)\end{aligned} \tag{5.19}$$

$$\begin{aligned}[\![getReferencedBy(\delta, c_1, n_1)]\!](dms, ds, as, rs) = \\ (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge \{''s'', \nu\} \in \eta_1 \\ \wedge\{''m'' : \delta,'' k'' : c,'' p'' : n_1\} \in \nu\}], rs)\end{aligned} \tag{5.20}$$

$$\begin{aligned}[\![deleteReference(\kappa_1)]\!](dms, ds, as, rs) = \\ (dms, ds, as, rs[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}])\end{aligned} \tag{5.21}$$

$$\begin{aligned}[\![renameReference(\kappa_1, \kappa_2)]\!](dms, ds, as, \\ rs \cup \{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \wedge key(\rho_1) = \kappa_1\}) = \\ (dms, ds, as, rs[\{\rho_2 \mapsto \eta_1 \mid key(\rho_2) = \kappa_2\}])\end{aligned} \tag{5.22}$$

Figure 5.11: Functions for management of references in the MMDPL.

Algorithm 5.10: Operation *reference* of the MMSEL and for reference evolution.

Legend: Let m_1, m_2 be model names, let c_1, c_2 be kinds, let n_1, n_2 be property names.

```
reference $m_1.c_1.n_1$ by $m_2.c_2.n_2$
if $(getModel(m_1) \neq \bot) \land (getModel(m_2) \neq \bot)$ then
    if $getReference((m_1.c_1, n_1)) = \bot$ then
        $r$ = newReference($(m_1.c_1, n_1)$);
        setProperty($r$, "m", $m_1$);
        setProperty($r$, "k", $(m_1.c_1, n_1)$);
        setProperty($r$, "p", $n_1$);
        setProperty($r$, "s", "[{"m": $m_2$, "k": $c_2$, "p": $n_2$}]");
    else
        $r$ = getReference($(m_1.c_1, n_1)$);
        setProperty($r$, "s", getProperty($r$, "s").push({"m": $m_2$, "k": $c_2$,
          "p": $n_2$}));
    putReference($r$);
```

Algorithm 5.10 defines operations *reference*. Operation *reference* checks if the RSM already contains an entity for the referenced property. If the entity is found, the referencing property is added to sources, else the function creates a new entity with a referencing entity and stores the changes in the RSM.

Algorithm 5.11: Modified operation *rename* of the MMSEL for reference evolution.

```
Legend: Let $m_1$ be a model name, let $c_1$ be a kind, let $n_1$, $n_2$ be a
 property names.

rename $m_1.c_1.n_1$ to $n_2$
if $getModel(m_1) \neq \perp$ then
    foreach element $e$ of $get(m_1, kind = c_1)$ do
        setProperty($e$, $n_2$, getProperty($e$, $n_1$));
        removeProperty($e$, $n_1$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        put($m$, $e$);
    renameReference(($m_1.c_1, n_1$),($m_1.c_1, n_2$));
    foreach element $g$ of $getReferencedBy(m_1, c_1, n_1)$ do
        foreach element $f$ of getProperty($g$, "s") do
            if getProperty($f$, "m") = $m_1 \wedge$ getProperty($f$, "k") = $c_1 \wedge$
             getProperty($f$, "p")=$n_1$ then
                getProperty($g$, "s").delete({"m": $m_1$, "k": $c_1$, "p": $n_1$});
                getProperty($g$, "s").push({"m": $m_1$, "k": $c_1$, "p": $n_2$});
        putReference($g$);
```

Algorithm 5.11 defines operation *rename*. Operation *rename* executes standard renaming and in the end it fixes the references. First, it tries to rename all references to the renamed property by *renameReference* function. Then it iterates through all references which are referenced by the renamed property and it fixes the new property name. At the end of the operation, all changes are saved to the RSM.

Algorithm 5.12: Modified operation *delete* of the MMSEL for reference evolution.

```
Legend: Let $m_1$ be a model name, let $c_1$ be a kind, let $n_1$ be a property name.

delete $m_1.c_1.n_1$
if getModel($m_1$)$\neq\perp$ then
    foreach element e of get($m_1, kind = c_1$) do
        removeProperty($e$, $n_1$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        put($m$, $e$);
    $r$ = getReference(($m_1.c_1, n_1$));
    if $r \neq\perp$ then
        foreach element f of getProperty(r, "s") do
            foreach element g of get(getProperty(f, "m"),getProperty(f, "k")) do
                removeProperty($g$, getProperty($f$, "p"));
                put(getProperty($f$, "m"),$g$);
        deleteReference(($m_1.c_1, n_1$));
        foreach element g of getReferencedBy($m_1, c_1, n_1$) do
            foreach element f of getProperty(g, "s") do
                if getProperty(f, "m")=$m_1\wedge$ getProperty(f, "k")= $c_1\wedge$ getProperty(f,"p")=$n_1$ then
                    getProperty($g$, "s").delete( {"m": $m_1$, "k": $c_1$, "p": $n_1$});
            putReference($g$);
```

Algorithm 5.12 defines operation *delete*. First, it deletes the property. In the next step the operation iterates through all properties which reference the removed property and removes them. When all references are removed, then also the entity of the reference is deleted. The last step is to remove the property itself from all reference entities which are referenced by it.

Algorithm 5.13: Modified operation *move* of the MMSEL for reference evolution.

```
Legend: Let $m_1$, $m_2$ be model names, let $c_1$, $c_2$ be kinds, let $n_1$ be a property name.

move $m_1.c_1.n_1$ to $m_2.c_2$
if (getModel($m_1$)$\neq \perp$) $\wedge$ (getModel($m_2$) $\neq \perp$) then
    foreach element $e$ of $get(m_1, kind = c_1)$ do
        foreach element $f$ of $get(m_2, kind = c_2)$ do
            setProperty($f$,$n_1$,getProperty($e$, $n_1$));
            setProperty($f$, $version$, getProperty($f$, $version$) + 1);
            put($m_2$, $f$);
        setProperty($e$, $version$, getProperty($e$, $version$) + 1);
        removeProperty($e$, $n_1$);
        put($m_1$, $e$);
    renameReference(($m_1.c_1, n_1$),($m_2.c_c, n_1$));
    foreach element $g$ of $getReferencedBy(m_1, c_1, n_1)$ do
        foreach element $f$ of getProperty($g$, "sources") do
            if getProperty($f$, "m")=$m_1 \wedge$ getProperty($f$, "k") = $c_1 \wedge$
              getProperty($f$, "p") = $n_1$ then
                getProperty($g$, "s").delete({"m": $m_1$, "k": $c_1$, "p": $n_1$});
                getProperty($g$, "s").push({"m": $m_2$, "k": $c_2$, "p": $n_1$});
        putReference($g$);
```

Algorithm 5.13 defines operation *move* which moves a property to a new entity and then renames all references to the new one. The last step is replacing the new name of the entity in the referenced properties in the RSM.

6. Implementation

We have implemented a prototype of the MMSEL as a part of the thesis to help us further demonstrate and validate the functionality of the language. In this chapter, we will briefly describe its architecture and how to use it. The application implements the MMSEL with reference extension and simulates the optimized version of operations when intra-model operations are executed inside the specific model.

6.1 Third Party Tools and Technologies

The application is based on the .NET framework 4.6.2 [53]. It is written in the C# language 7.0 in the Visual Studio IDE [69]. We created an abstract layered-based model of the MMD from MongoDB and MariaDB [37]. To be able to work with the databases in the code, we used the MongoDB Driver [44] and the MySQL Data [50]. For better visualization and control of databases we used the MongoDB Compass Community [43] for MongoDB and the MySQL Workbench 6.3 CE [49]. Git [22] served as the version control system.

6.2 Architecture

The architecture of the application can be split into three parts: model-independent part, model-specific part, and database part. Let us shortly describe each part.

6.2.1 Model-independent Layer

The model-independent layer is mainly responsible for parsing of user input, execution of the targeted evolution operation and communication with the affected models. The user input is parsed by our tokenizer which is also used for validation of the input. Our implementation does not support spaces in input strings so we replace them by underscore.

The next important role of the layer is the distribution of operations which happens in the multi-model engine. The engine distributes commands to the specific models.

6.2.2 Model-specific Layer

The model-specific layer contains adapters for each data model. Our abstract model of the MMD is built from key/value, document and relational models. Each one of them implements the DSEL interface to be able to simulate optimized operations and `Get` and `Put` for the rest of the logic.

6.2.3 Database Layer

The last layer contains data models which contain the evolving data. In our case, the layer is composed of MongoDB as a document model and it is also used as a key/value model to simplify the solution (no need to another data adapter or visualization tool). The second used database is MariaDB which represents a relational model. In our solution, both databases can be easily changed by providing different connection strings in `App.config`.

6.3 Application Usage

The prototype is a standard .NET application with command-line interface. It requires both databases to run and the respective connection strings. It can be run using the following command:

```
> MultiModelSchemaEvolution.exe
```

When the application starts, it waits for a user input. The input can be any command which matches MMSEL and is relevant to the used databases. The application informs the user about data processing, command validation, and possible errors. The application can be terminated by command *EXIT*.

7. Experiments

Now, we will demonstrate the benefits of our solution and results of running our implementation on experiments using a computer with the following configuration:

- a Inter Core i7 2.50GHz CPU
- 16GB of RAM memory
- 64-bit Windows 10 Home

7.1 Experiment Plan

We decided to split our experiment into two parts. First, we shortly describe the plan of execution of the experiments and then we will go through them in detail.

7.1.1 Proof of Concept

First, we will demonstrate that our implementation works as expected. We will do it on exactly the same example of the MMD which we used in the theoretical part and the same evolution commands, i.e., the example of the MMD of an e-shop with teas. The reason for it is that we will be able to simply check the initial state and the resulting state in the visualization tools. Plus it is easy for the reader to replicate our steps and check them by himself/herself.

The initial state of all experimental operations with the e-shop MMD is shown in Figure 5.6. Figure 7.1 shows the initial state of the MMD in the visualization tools for easier comparison with the future result states. The executed operations are the following:

- **Operation add:**

```
> add document.tea.importer = Tea_Comp.
```

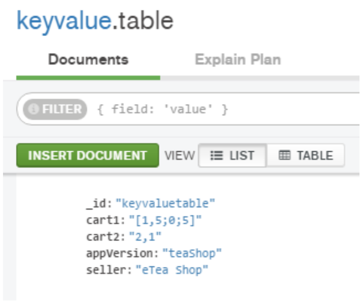
```
> add relational.users.canDeliver = true where relational.users.
    address = null
```

- **Operation delete:**

```
> delete document.tea.country
```

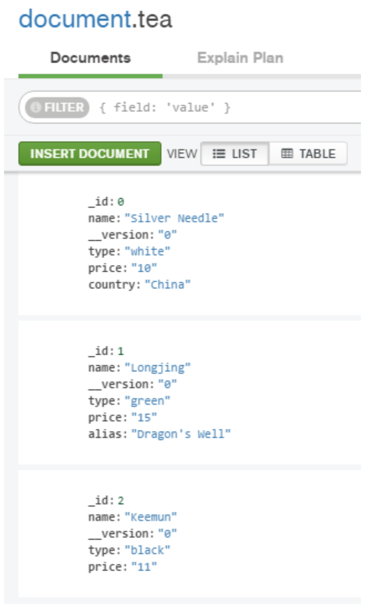
- **Operation rename:**

```
> rename relational.users.name to fullname
```



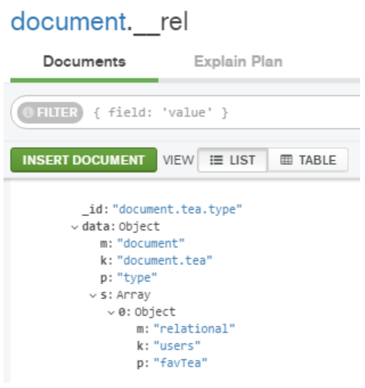
```
_id: "keyvaluetable"
cart1: "[1,5;0;5]"
cart2: "2,1"
appVersion: "teaShop"
seller: "eTea Shop"
```

(a) Data in the key/value model.



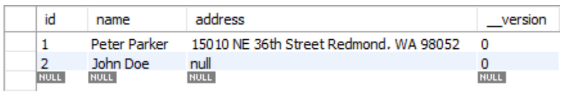
```
_id: 0
name: "Silver Needle"
__version: "0"
type: "white"
price: "10"
country: "China"

_id: 1
name: "Longjing"
__version: "0"
type: "green"
price: "15"
alias: "Dragon's Well"

_id: 2
name: "Keemun"
__version: "0"
type: "black"
price: "11"
```

(b) Data in the document model.



```
  _id: "document.tea.type"
v data: Object
    m: "document"
    k: "document.tea"
    p: "type"
  v s: Array
    v 0: Object
        m: "relational"
        k: "users"
        p: "favTea"
```

(c) References in the MMD.



id	name	address	__version
1	Peter Parker	15010 NE 36th Street Redmond. WA 98052	0
2	John Doe	null	0
NULL	NULL	NULL	NULL

(d) Data in the relational model.

Figure 7.1: The MMD of an e-shop in visualization tools.

- **Operation copy:**

```
> copy keyValue.appVersion to document.tea
```

- **Operation move:**

```
> move keyValue.seller to document.tea
```

- **Operation reference:**

```
> reference document.tea.type by relational.users.favTea
> delete document.tea.type
```

7.1.2 Real-world Data Experiments

One of the most famous real-world big data projects for experiments is the MediaWiki [39] project. It is an open-source web framework with the best-known representative Wikipedia, a world-wide popular collaborative encyclopedia. The next one is the Internet Movie Database (IMDb) [30] – an online database of information related to world films, TV programs, actors, fan reviews, and ratings, etc. Both of them are publicly accessible for non-commercial experiments and benchmarks.

We tried to use MediaWiki data but we faced troubles with their recommended import approach to the relational model. Based on our research it seems that there are changes in the relational schema which are not propagated to the data dumper. During the data import the dumper fails on the database constraints. It block us to import it and use it as a relational model. For this reason, we have decided to use IMDb data for our tests. The provided data are for a relational model and for a document model so the abstract MMD for the real-world experiments contains only two models: document and relational.

Data Preparation

The data for the relational model are available in [27] which also expains the meaning of them. The data for the document model can be downloaded as is described on [28]. We have not found any proper description of the data for document model so let us shortly introduce the high-level structure. The document model is represented using database *movies* which consists of the following collections:

- *Contributor* – A collection of contributors of the IMDb site.
- *Movie* – A collection of all movies in the IMDb data set.
- *MovieDoc* – A collection of documents describing movies.

- *MovieRole* – A collection of actors from movies with additional information like, e.g., role name.

The collections contain varying structures for different sub-types of entities like, e.g., properties for movies which are not presented for series. Because of this fact, we do not describe the structure in detail and we will go through specific data in the executed examples.

An important fact is that our structure is not the original structure of the IMDb. We used unrelated existing exports to the targeted models. As a consequence, the data are duplicated in both models but it is not a complication for our purpose. For example, in the relational model, the information about a title are stored in the *title_basics* table. In the document model we can find similar information about the title in the *Movie* collection.

We recommend readers to use the database from the proof of concept for testing of the MMSEL. The reason is that the preparation of the document model is a complex process. First, we had to set up a Unix-like virtual machine because the process of download executes a Java program with hard-coded settings. Downloading and extraction of data to the document model took approximately 35 hours.

For the relational model we used the MariaDB and for the document model, we used the MongoDB. To be able to use our implementation of MMSEL with IMDb we did only changes of connection strings to both databases.

Experimental Operations

The executed operations are the followings:

- **Operation add:**
  - Experiment Q1:

    ```
    > add document.MovieRole.RoleAge = unknown
    ```

  - Experiment Q2:

    ```
    > add relational.title_basics.country_restriction = null
      where relational.title_basics.isAdult = 1
    ```

- **Operation delete:**
  - Experiment Q3:

    ```
    > delete document.MovieRole.RoleAge where document.
      MovieRole.ContribClass = miscellaneous and document.
      MovieRole.__version = 1
    ```

  - Experiment Q4:

    ```
    > delete relational.title_basics.country_restriction
    ```

- **Operation rename:**
  - Experiment Q5:

    ```
    > rename document.MovieRole.RoleAge to AgeOfRole where
      document.MovieRole.__version = 1
    ```

  - Experiment Q6:

    ```
    > rename relational.title_basics.isAdult to isRestricted
      where relational.title_basics.genres = Adult and
      relational.title_basics.__version = 2
    ```

- **Operation copy:**
  - Experiment Q7:

    ```
    > copy document.Contributor.ContribBio to document.
      MovieRole where document.Contributor.ContribName =
      document.MovieRole.ContribName
    ```

  - Experiment Q9:

    ```
    > copy relational.name_basics.knownForTitles to document.
      MovieRole where relational.name_basics.primaryName =
      document.MovieRole.ContribName
    ```

- **Operation move:**
  - Experiment Q8:

```
> move relational.title_akas.language to relational.
    title_basics where relational.title_akas.titleId =
    relational.title_basics.tconst
```

  - Experiment Q10:

```
> move document.Contributor.ContribBio to relational.
    name_basics where document.Contributor._id = relational.
    name_basics.primaryName
```

- **Operation reference:**
  - Experiment Q11:

```
> reference relational.name_basics.primaryName by document.
    Contributor.ContribName
> rename document.Contributor.ContribName to PrimaryName
> delete relational.name_basics.primaryName
```

7.2 Execution of the Proof of Concept Experiments

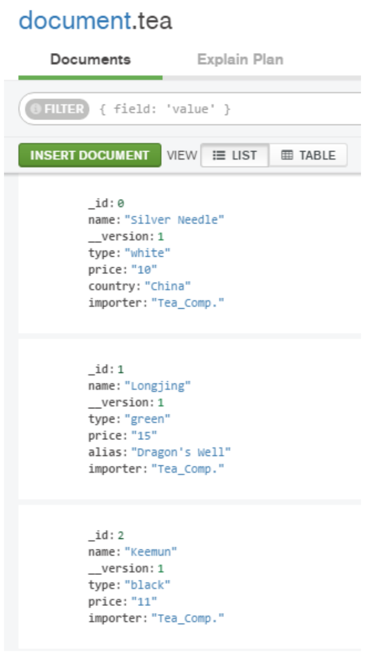
For each of the experiments, we will always introduce the executed operation, the resulting state, and compare the benefit of the usage of the MMSEL instead of a manual migration.

add document.tea.importer = Tea_Comp.

The operation adds a new property *importer* to all *tea* entities. We expect that the result will be the new property with value *Tea_Comp.* and an incremented version property.

The result of the operation is visualized in Figure 7.2. We can see the new property with the required value and an increased version. Now, we can compare what is the effort to do it manually. This operation is an intra-model one which means it is executed directly in the one model. One model means one database syntax. The example of the syntax of the command is shown in Figure 7.3.

At first sight, we can see that the manual operation is much more complex than the MMSEL operation. The second aspect is the manual command requires a knowledge of the MongoDB operation language.

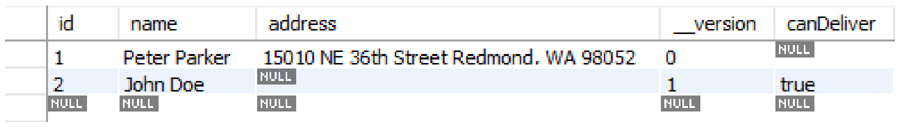


```
_id: 0
name: "Silver Needle"
__version: 1
type: "white"
price: "10"
country: "China"
importer: "Tea_Comp."

_id: 1
name: "Longjing"
__version: 1
type: "green"
price: "15"
alias: "Dragon's Well"
importer: "Tea_Comp."

_id: 2
name: "Keemun"
__version: 1
type: "black"
price: "11"
importer: "Tea_Comp."
```

Figure 7.2: The result of add operation of new property to the document model in visualization tool.

```
db.tea.find().forEach(function(item){
    db.tea.update({_id: item._id}, {$set: {importer: "Tea_Comp."}},
        true, true);
    db.tea.update({ id: item._id}, {$inc: {__version: 1}}, true, true
        );
});
```

Figure 7.3: An example of code which adds new property *importer* to all entities *tea* in the document model (MongoDB).



	id	name	address	__version	canDeliver
	1	Peter Parker	15010 NE 36th Street Redmond. WA 98052	0	NULL
	2	John Doe	NULL	1	true
	NULL	NULL	NULL	NULL	NULL

Figure 7.4: The visualization of the result of operation *add* of a new property to the relational model.

```
ALTER TABLE `users` ADD `canDeliver` VARCHAR(255);
UPDATE `users` SET `users`.`canDeliver` = 'true' WHERE `users`.`
    address` IS NULL;
IF NOT EXISTS(SELECT 1 FROM information_schema.COLUMNS WHERE
    TABLE_NAME = 'users' AND COLUMN_NAME = '__version')
         THEN
                   ALTER TABLE `users` ADD `__version` INT DEFAULT 0;
         END IF;
UPDATE `users` SET `{0}`.`__version` = `users`.`__version` + 1 WHERE
    `users`.`address` IS NULL;
```

Figure 7.5: The code which adds a new property *canDeliver* to all entities *users* in the relational model (MariaDB).

add relational.users.canDeliver = *true* where relational.users.address = *null*

The operation adds a new property *canDeliver* with value *true* to entities of the relational model which meet the where condition and it increases version property.

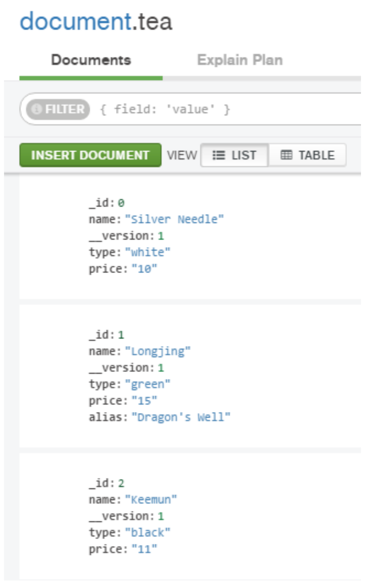
The result of the operation is visualized in Figure 7.4. The new property appeared for an entity which meets the where condition. This is an intra-model operation which affects only the relational model. Let us introduce the manual operation in Figure 7.5.

We showed the implementation of the manual command which controls the existence of version property. It makes the implementation more complex but based on our experiences it is often required in an enterprise environment. The manual solution consists of four SQL commands instead of one command in the MMSEL. It also means that the developer has to make sure that it will be executed in a single transaction.

delete document.tea.country

The operation removes property *country* for all entities *tea*. We expect that the property will be removed and the version property will be incremented.

The logic is similar as the logic of operation add to the document model. Figure 7.6 shows the result. Differences between manual approach and the MMSEL



```
_id: 0
name: "Silver Needle"
__version: 1
type: "white"
price: "10"

_id: 1
name: "Longjing"
__version: 1
type: "green"
price: "15"
alias: "Dragon's Well"

_id: 2
name: "Keemun"
__version: 1
type: "black"
price: "11"
```

Figure 7.6: The visualization of the result of operation *delete* of a property in the document model.

```
db.tea.find().forEach(function(item){
    db.tea.update({_id:item._id},{$unset:{country:1}},false,true);
    db.tea.update({_id:item._id},{$inc:{__version:1}},false,true);
});
```

Figure 7.7: The code which removes the property *country* from all entities *tea* in the document model (MongoDB).

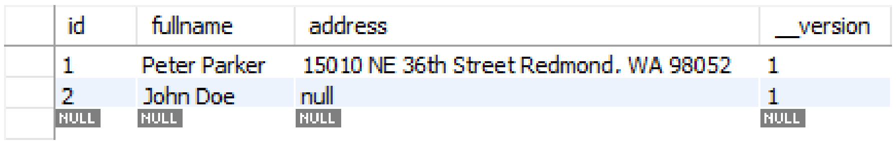
are in general similar as in the case of operation *add*. The manual implementation is presented in Figure 7.7.

rename relational.users.name to fullname

The operation renames property *name* of entity *users* to property *fullname*. The expected result is that the column of the table *users* should be renamed and the version is incremented.

Figure 7.8 shows the result of the operation and the SQL command in Figure 7.9 is the example of the manual migration operation.

The manual command is simple and it is very similar to the add command. The whole logic is in the alter the table. But let us consider the case that we add a where condition to the command. In the relational world, we cannot simply alter the table because we want to preserve some data in the original column. For



	id	fullname	address	__version
	1	Peter Parker	15010 NE 36th Street Redmond, WA 98052	1
	2	John Doe	null	1
	NULL	NULL	NULL	NULL

Figure 7.8: The visualization of the result of operation *rename* of a property in the relational model.

```
ALTER TABLE `users` CHANGE COLUMN `name` `fullname` VARCHAR(255);
IF NOT EXISTS(SELECT 1 FROM information_schema.COLUMNS WHERE
    TABLE_NAME='users' AND COLUMN_NAME='__version')
THEN
     ALTER TABLE `users` ADD `__version` INT DEFAULT 0;
END IF;
UPDATE `users` SET `users`.`__version`=`users`.`__version`+1;
```

Figure 7.9: The code which renames property *name* to *fullname* for all entities *users* in the relational model (MariaDB).

example, we will add a condition *where relational.users.name = "Peter Parker"*. Now the manual migration command can be seen in Figure 7.10.

After execution of this operation, both columns *name* and *fullname* co-exist. The simple change as adding a where condition makes a manual command more difficult than it would seem at first glance. The MMSEL hides this extra logic and a developer does not have to deal with it.

copy keyValue.appVersion to document.tea

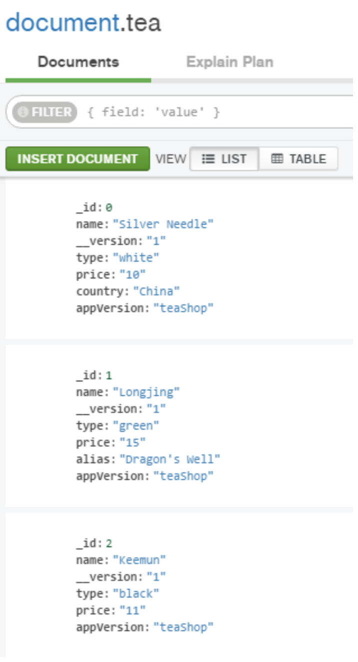
The operation copies property *appVesrion* from the key/value model to all entities *tea* in the document model. The operation is inter-model which means that the data transport is handled by the multi-model engine. The expected result is that property *appVersion* will appear in all entities *tea* in the document model. During the operation, the property field should be incremented.

The result is shown in Figure 7.11 and it is matching our expectations. Let us discuss how the manual migration can be done. We cannot introduce a code as we did for intra-model operations, because we need a mediator which will transform data between the models. This fact clearly shows one of the benefits of the MMSEL.

The effort to create a manual migration can be describe with psedo-code in Algorithm 7.1. The operations to get and set entities requires a knowledge of both affected models.

```
ALTER TABLE `users` ADD `fullname` VARCHAR(255);
UPDATE `users` SET `fullname`=`name` WHERE `users`.`name`="Peter␣
    Parker";
UPDATE `users` SET `name`=null WHERE `users`.`name`="Peter␣Parker"
IF NOT EXISTS(SELECT 1 FROM information_schema.COLUMNS WHERE
    TABLE_NAME='users' AND COLUMN_NAME='__version')
THEN
     ALTER TABLE `users` ADD `__version` INT DEFAULT 0;
END IF;
UPDATE `users` SET `users`.`__version`=`users`.`__version`+1 WHERE `
    users`.`name`="Peter␣Parker";
```

Figure 7.10: The code which renames property *name* to *fullname* for selected entities *users* selected by a where condition in the relational model (MariaDB).



```
_id: 0
name: "Silver Needle"
__version: "1"
type: "white"
price: "10"
country: "China"
appVersion: "teaShop"

_id: 1
name: "Longjing"
__version: "1"
type: "green"
price: "15"
alias: "Dragon's Well"
appVersion: "teaShop"

_id: 2
name: "Keemun"
__version: "1"
type: "black"
price: "11"
appVersion: "teaShop"
```

Figure 7.11: The visualization of the result of operation *copy* from the key/value model to the document model.

Algorithm 7.1: An example of copy *appVersion* property from the key/value model to the document model.

```
read appVersion property from the key/value model;
if property exists then
    foreach entity tea in the document model do
        get next entity;
        add property appVersion to the entity;
        increment the version property;
        store the entity;
```

move keyValue.seller to document.tea

The operation removes property *seller* from the key/value model and moves it to all entities *tea* in the document model. It is an inter-model operation so there is the same limitation as we discussed in case of operation *copy*. Figure 7.12 shows that our implementation works and we get a correct result where the entity is removed from key/value model and it is inserted into the document entities.

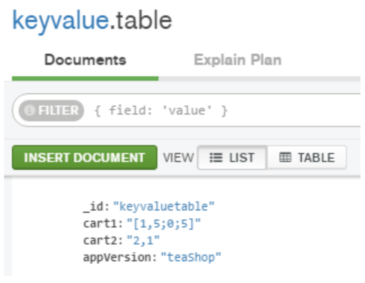
To see the idea of manual migration we introduce a pseudo-code in Algorithm 7.2. The algorithm copies the property to all target entities and deletes it when it is copied.

Algorithm 7.2: An example of move *seller* property from the key/value model to the document model.

```
read seller property from the key/value model;
if property exists then
    foreach entity tea in the document model do
        get next entity;
        add property seller to the entity;
        increment the version property;
        store the entity;
    remove seller from the key/value model;
```

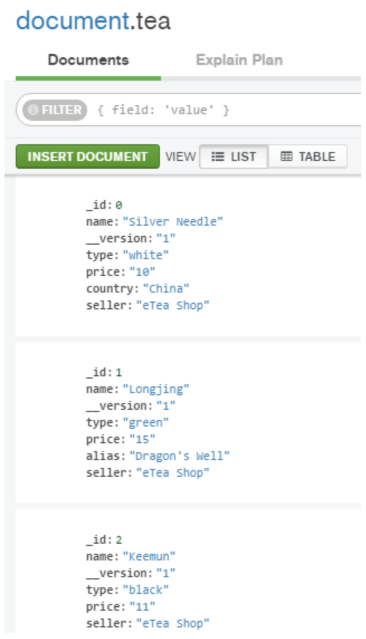
reference document.tea.type by relational.users.favTea && delete document.tea.type

The sequence of operations *reference* and *delete* of referenced entity should end up so that the referencing entity is removed too. Before we execute the operations we execute operation **add relational.users.favTea = 1** to add the column to the



```
_id: "keyvaluetable"
cart1: "[1,5;0;5]"
cart2: "2,1"
appVersion: "teaShop"
```

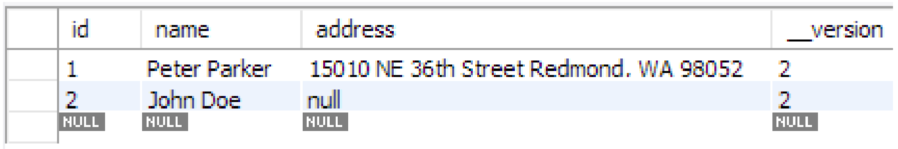
(a) Data in the key/value model.



```
_id: 0
name: "Silver Needle"
__version: "1"
type: "white"
price: "10"
country: "China"
seller: "eTea Shop"

_id: 1
name: "Longjing"
__version: "1"
type: "green"
price: "15"
alias: "Dragon's Well"
seller: "eTea Shop"

_id: 2
name: "Keemun"
__version: "1"
type: "black"
price: "11"
seller: "eTea Shop"
```

(b) Data in the document model.

Figure 7.12: The visualization of the result of operation *move* from the key/value model to the document model.



id	name	address	__version
1	Peter Parker	15010 NE 36th Street Redmond, WA 98052	2
2	John Doe	null	2
NULL	NULL	NULL	NULL

Figure 7.13: The visualization of the result of operation *delete* of a referenced property which was referenced from relational model.

relational model and the version property is changed to *1*. We execute the reference operation first and then the delete operation. Figure 7.13 shows the resulting state of the referencing entity. We do not show the result of the referenced entity because we have already shown that operation delete works.

In the result, we see that entities do not contain the property *favTea* which means that it was removed during the processing of operation delete. Also, the version field is incremented to value *2*.

To be able to perform this migration manually without the MMSEL the developer has to know which property is referencing the affected property. In Algorithm 7.3, we show a high-level pseudo-code of the manual migration. The complexity of the algorithm is hidden in the first step. The step to get all affected entities has to be able to get the information about references in the MMD. The second hidden complexity is that the entities can be stored in different models so the algorithm has to distinguish between them.

Algorithm 7.3: An example of processing of delete referencing property.

```
get all affected entities;
foreach affected entity do
    get next entity;
    remove the referencing property;
    increment the version property;
    store the entity;
```

Summary of the Proof of Concept

In the mentioned examples we showed that our implementation works and follows the definition of the MMSEL. On these simple examples of the e-shop MMD we demonstrated main advantages of the MMSEL as a unified language across all models, a reference handling, a unified get and put for all models, etc. We will summarize all benefits in Chapter 8.

7.3 Execution of IMDb Data Experiments

For the real-world experiments, we decided to split them into two categories: intra-model and inter-model. We will show both types of operations and review results and compare the usage of the MMSEL and a manual migration.

We decided to use a different approach for data preparation for real-world experiments. We gradually change the database state instead of using a fresh instance of the database before each operation. One of the reasons is the complexity of the MMD data restore and also we can simulate real demands by application of multiple migration operations.

We decided that we will describe experiments as real-world demands. This approach will make benefits of the language more visible and readers can easier imagine the usage of the MMSEL in practice.

For each experiment, we mention the number of affected entities and the number of targeted entities for each experiment. The number of affected entities means the number of entities which are changed during the execution of an operation. The number of targeted entities is the size of the set of entities which corresponds to the change request.

For better understanding we repeat the command for each experiment and we reference operations of each command by a shortcut Q[numer of experiment].

7.3.1 Intra-model Operations Experiments

The intra-model operations are *add*, *delete*, and *rename*. We choose to execute one operation of each kind for both models and we try to simulate real-world demands.

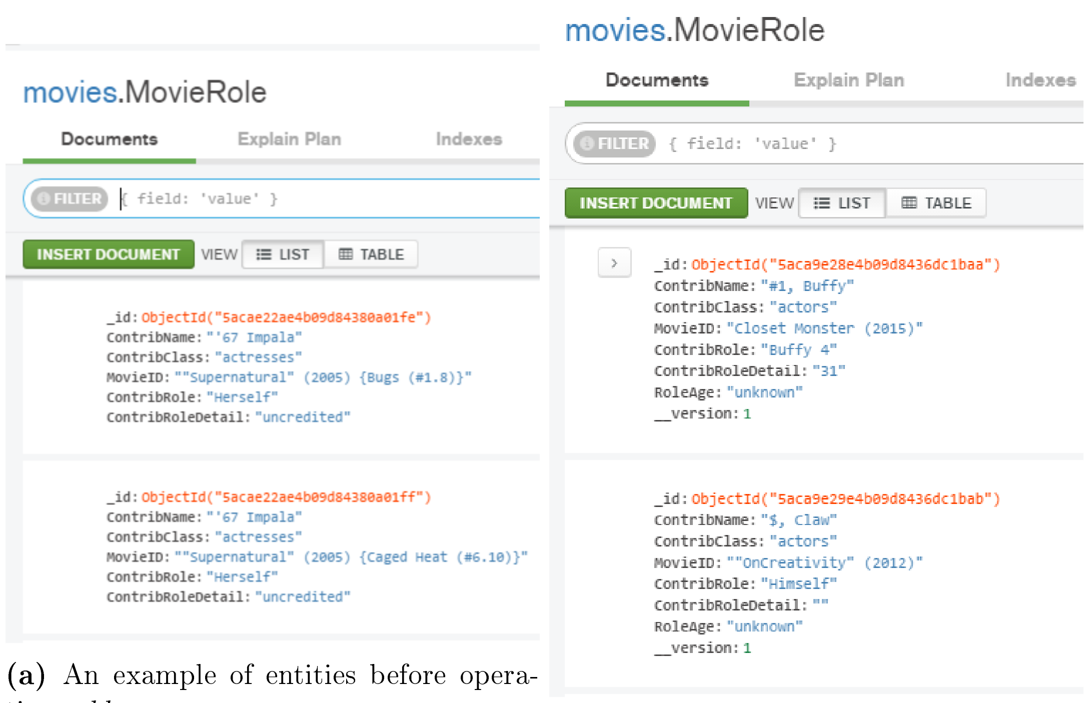
Experiment Q1

```
> add document.MovieRole.RoleAge = unknown
```

The first demand is to add a new property *RoleAge* to all entities *MovieRole* with default value *unknown*.

Figure 7.14 show examples of entities before and after execution of the operation. The operation targets 64,424,283 entities. Because the implementation of the MMSEL is eager, we are modifying all entities at once which takes a couple of minutes on the current computer setup.

Let us describe what happens during the execution of Q1 with data: All entities *MovieRole* have added a property *RoleAge* with the value *unknown* plus an automatically added version property with value *1*. We do not present an example of a manual migration because it is similar to migration script from the proof of concept section.



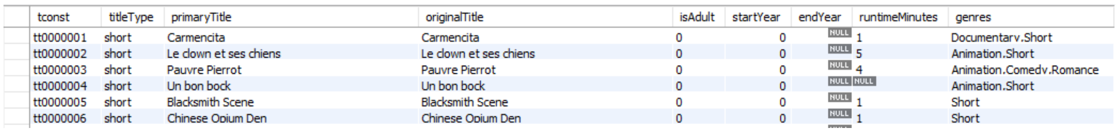
```
_id: ObjectId("5acae22ae4b09d84380a01fe")
ContribName: "'67 Impala"
ContribClass: "actresses"
MovieID: ""Supernatural" (2005) {Bugs (#1.8)}"
ContribRole: "Herself"
ContribRoleDetail: "uncredited"

_id: ObjectId("5acae22ae4b09d84380a01ff")
ContribName: "'67 Impala"
ContribClass: "actresses"
MovieID: ""Supernatural" (2005) {Caged Heat (#6.10)}"
ContribRole: "Herself"
ContribRoleDetail: "uncredited"
```

(a) An example of entities before operation *add*.

```
_id: ObjectId("5aca9e28e4b09d8436dc1baa")
ContribName: "#1, Buffy"
ContribClass: "actors"
MovieID: "Closet Monster (2015)"
ContribRole: "Buffy 4"
ContribRoleDetail: "31"
RoleAge: "unknown"
__version: 1

_id: ObjectId("5aca9e29e4b09d8436dc1bab")
ContribName: "$, Claw"
ContribClass: "actors"
MovieID: ""OnCreativity" (2012)"
ContribRole: "Himself"
ContribRoleDetail: ""
RoleAge: "unknown"
__version: 1
```

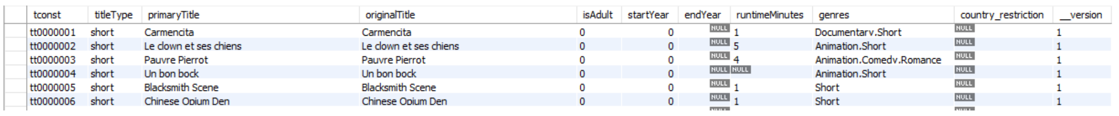
(b) An example of entities after operation *add*.

Figure 7.14: An example of the initial state of entities *MovieRole* and the state after operation *add*.



	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
	tt0000001	short	Carmencita	Carmencita	0	0	NULL	1	Documentarv.Short
	tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	0	NULL	5	Animation.Short
	tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	0	NULL	4	Animation.Comedv.Romance
	tt0000004	short	Un bon bock	Un bon bock	0	0	NULL	NULL	Animation.Short
	tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	0	NULL	1	Short
	tt0000006	short	Chinese Opium Den	Chinese Opium Den	0	0	NULL	1	Short

(a) An example of entities before operation *add*.



	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres	country_restriction	__version
	tt0000001	short	Carmencita	Carmencita	0	0	NULL	1	Documentarv.Short	NULL	1
	tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	0	NULL	5	Animation.Short	NULL	1
	tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	0	NULL	4	Animation.Comedv.Romance	NULL	1
	tt0000004	short	Un bon bock	Un bon bock	0	0	NULL	NULL	Animation.Short	NULL	1
	tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	0	NULL	1	Short	NULL	1
	tt0000006	short	Chinese Opium Den	Chinese Opium Den	0	0	NULL	1	Short	NULL	1

(b) An example of entities after operation *add*.

Figure 7.15: An example of the initial state of entities *title_baciscs* and a state after operation *add*.

To sum up, we add a new property to 64,424,283 entities and in advance, we introduce a second new version property to all of them which provides us a support for correct recovery from interrupts during the processing and zero downtime during the data migration.

Experiment Q2

```
> add relational.title_basics.country_restriction = null where
    relational.title_basics.isAdult = 1
```

The next requirement is to prepare a database structure for restricted countries in *title_basics* based on the property *isAdult*. The value will be filled by application users so we can choose a default value ourselves.

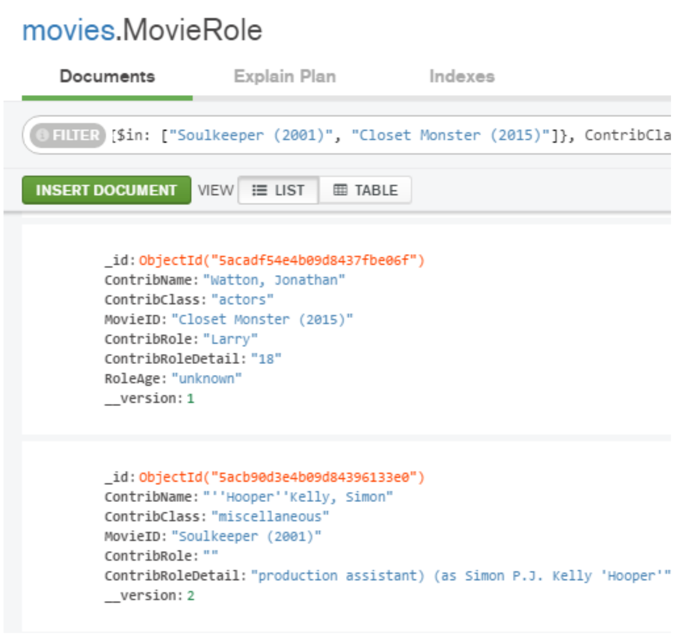
Figure 7.15 shows examples of entities before and after execution of experiment Q2. We can see that two new expected properties appeared. The operation sets the property *country_restriction* with value *null* to 149,636 entities where the property *isAdult* is *1* and the version property is added. Actually, the null value is set to all entities, because it is also a default value for an added column in the relational model. Thus, the real amount of affected entities is 4,920,457. Developers do not have to know this and mainly they do not have to take care of it because it is handled by the MMSEL.

The manual migration would use the same idea as we showed in Figure 7.5.

Experiment Q3

```
> delete document.MovieRole.RoleAge where document.MovieRole.
    ContribClass = miscellaneous and document.MovieRole.__version = 1
```

Following the previous requirement for the document model, we received a change request to remove the new property *RoleAge*, where *ContribClass* is *mis-*



```
[$in: ["Soulkeeper (2001)", "Closet Monster (2015)"]}, ContribCla
```

```
_id: ObjectId("5acadf54e4b09d8437fbe06f")
ContribName: "Watton, Jonathan"
ContribClass: "actors"
MovieID: "Closet Monster (2015)"
ContribRole: "Larry"
ContribRoleDetail: "18"
RoleAge: "unknown"
__version: 1
```

```
_id: ObjectId("5acb90d3e4b09d84396133e0")
ContribName: "''Hooper''Kelly, Simon"
ContribClass: "miscellaneous"
MovieID: "Soulkeeper (2001)"
ContribRole: ""
ContribRoleDetail: "production assistant) (as Simon P.J. Kelly 'Hooper'"
__version: 2
```

Figure 7.16: The result of operation *delete* of property *ContribClass* from entities *MovieRole*, where its value is *miscellaneous*.

cellaneous. Experiment Q3 deletes the property from all entities *MovieRole*, where the condition is fulfilled and the version of the entity is *1*.

Figure 7.16 presents the example of the migrated entity *MovieRole* against an entity which does not correspond to the where condition. Entities are migrated and only the affected entities have increased the version property. There is a disadvantage of this approach: entities co-exist with two different value of the version property, which means we cannot execute the next migration operations safely across all entities by its version field because it is not the same.

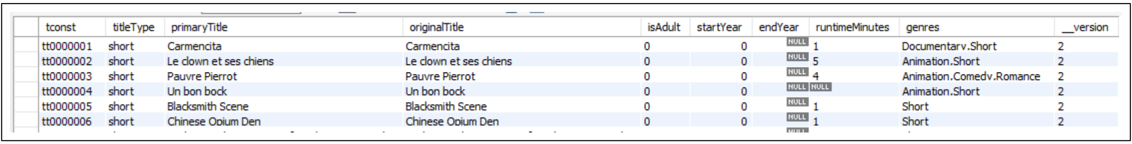
The operation deletes the property from 7,504,078 entities and it increases the version, too.

The manual migration is a little bit different than we showed before. The main difference is in the where condition. An example of code which does the manual migration is shown in Figure 7.17.

It is good to mention benefits of the MMSEL in this experiment. The first one is a simple filtering of targeted data without the knowledge of the MongoDB query language. The second benefit is the automated management of version to provide an opportunity for safer migrations.

```
db.MovieRole.find().forEach(function(item){
     db.MovieRole.update(
          {_id: item._id, ContribClass: "miscellaneous", __version: 1},
          {$unset: {RoleAge: 1}},
          false, true);
     db.MovieRole.update(
          { id: item._id, ContribClass: "miscellaneous", __version: 1},
          {$inc: {__version: 1}},
          false, true);
});
```

Figure 7.17: An example of code which deletes property *RoleAge* from all entities *MovieRole* in the document model.



	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres	__version
	tt0000001	short	Carmencita	Carmencita	0	0	NULL	1	Documentary.Short	2
	tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	0	NULL	5	Animation.Short	2
	tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	0	NULL	4	Animation.Comedy.Romance	2
	tt0000004	short	Un bon bock	Un bon bock	0	0	NULL	NULL	Animation.Short	2
	tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	0	NULL	1	Short	2
	tt0000006	short	Chinese Opium Den	Chinese Opium Den	0	0	NULL	1	Short	2

Figure 7.18: The result of operation *delete* of property *country_ restriction* from entities *title_ basics* from the relational model.

Experiment Q4

```
> delete relational.title_basics.country_restriction
```

The added property *country_ restiction* was not used so we received a request to remove it from all entities. Experiment Q4 removes the property from all entities. We added the property to entities where property *isAdult* is *1*. There is no need to specify it in operation *delete*. If the field is not set then there is nothing to remove and it is safe to execute it.

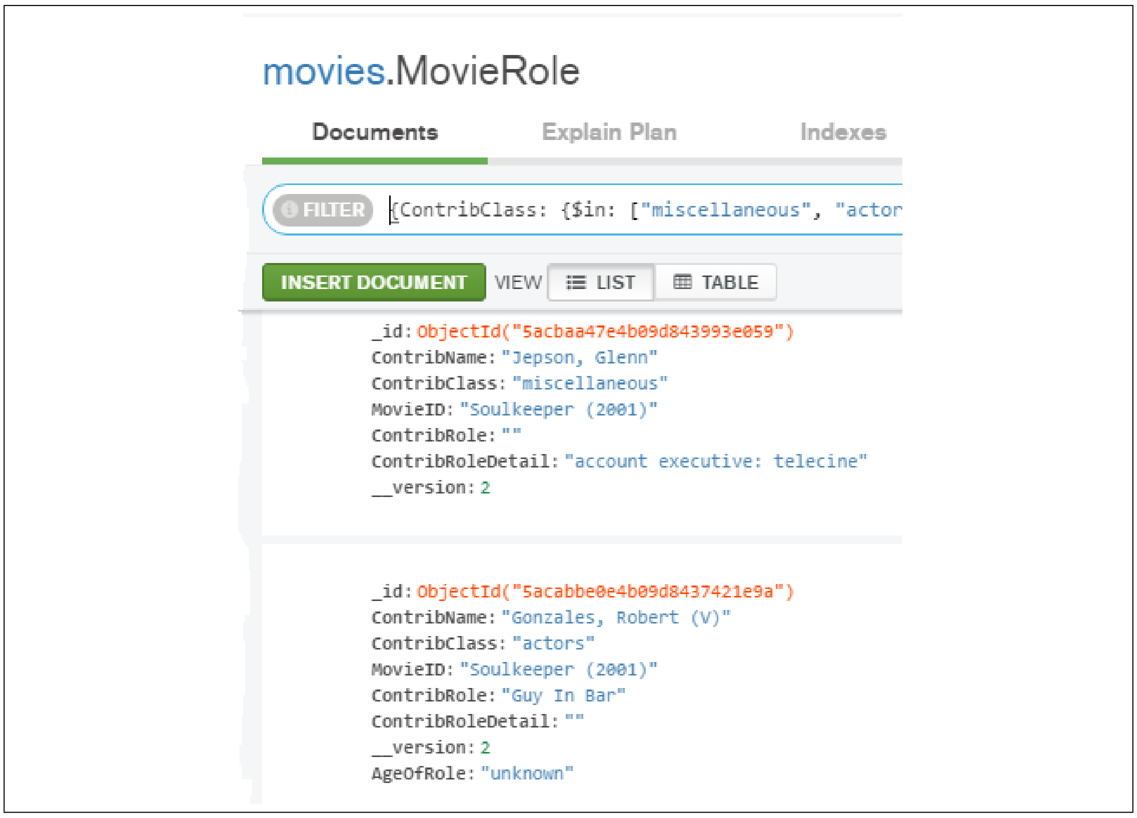
The manual migration will be similar as we showed in the proof of concept (see Section 7.2).

The result of the operation is displayed in Figure 7.18. The operation dropped column *country_ restriction* from the table and it changed the version property to *2*. From the application perspective, we removed the property from all 4,920,457 entities.

Experiment Q5

```
> rename document.MovieRole.RoleAge to AgeOfRole where document.
   MovieRole.__version = 1
```

The next experiment can happen during a data refactoring. The name of the property *RoleAge* is not self-describing so we have to change it to *AgeofRole*. In



```
{ContribClass: {$in: ["miscellaneous", "actor

_id: ObjectId("5acbaa47e4b09d843993e059")
ContribName: "Jepson, Glenn"
ContribClass: "miscellaneous"
MovieID: "Soulkeeper (2001)"
ContribRole: ""
ContribRoleDetail: "account executive: telecine"
__version: 2

_id: ObjectId("5acabbe0e4b09d8437421e9a")
ContribName: "Gonzales, Robert (V)"
ContribClass: "actors"
MovieID: "Soulkeeper (2001)"
ContribRole: "Guy In Bar"
ContribRoleDetail: ""
__version: 2
AgeOfRole: "unknown"
```

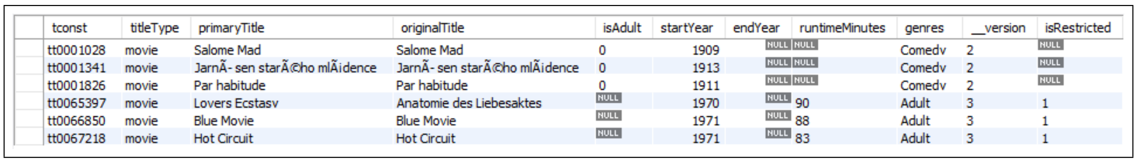
Figure 7.19: The result of operation *rename* of property *RoleAge* to *AgeOfRole* for entities *MovieRole* from the document model.

previous examples we introduced the field to all *MovieRole* entities and then we removed it from entities where *ContribClass = miscellaneous*. We have to remove from all entities where *ContribClass != miscellaneous* but the MMSEL does not support the not-equal comparison. Fortunately, it is not needed. We can execute experiment Q5 with version *1* for all entities. The rest of entities have version *2* and do not contain the property anyway.

The manual migration will not be much different than the one we showed in the proof of concept (see Section 7.2).

Figure 7.19 shows an example of migrated entities. The picture shows both types of entities: miscellaneous and non-miscellaneous. The first entity does not contain the changed *RoleAge* property and the version property is unchanged. The second type has property *RoleAge* changed to *AgeOfRole* and the version was incremented to version *2*. The value of the version property is the same in all entities *MovieRole* now so next operations can use the version property for safe execution.

The operation modified all 56,920,205 targeted entities.



tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres	__version	isRestricted
tt0001028	movie	Salome Mad	Salome Mad	0	1909	NULL	NULL	Comedv	2	NULL
tt0001341	movie	JarnÃ- sen starÃ©ho mlÃidence	JarnÃ- sen starÃ©ho mlÃidence	0	1913	NULL	NULL	Comedv	2	NULL
tt0001826	movie	Par habitude	Par habitude	0	1911	NULL	NULL	Comedv	2	NULL
tt0065397	movie	Lovers Ecstasv	Anatomie des Liebesaktes	NULL	1970	NULL	90	Adult	3	1
tt0066850	movie	Blue Movie	Blue Movie	NULL	1971	NULL	88	Adult	3	1
tt0067218	movie	Hot Circuit	Hot Circuit	NULL	1971	NULL	83	Adult	3	1

Figure 7.20: The result of operation *rename* of property *isAdult* to *isRestricted* for entities *title_ basics* from the relational model.

Experiment Q6

```
> rename relational.title_basics.isAdult to isRestricted where
    relational.title_basics.genres = Adult and relational.title_basics
    .__version = 2
```

The next experiment simulates a requirement to rename a subset of entities in the relational model. For movies where their gender is *Adult*, we have to rename property *isAdult* to *isRestricted* to make it more meaningful. Experiment Q6 renames the property for all movies with genre *Adult* and with the latest version.

In the database, we can also find movies with a genre like *Adult.Comedy* which we ignore in this experiment. The MMSEL do not support substring operation and it can be migrated with the specific genre string.

The manual migration will be similar to the one we showed in the proof of concept (see Section 7.2).

In Figure 7.20 we show an example of *Adult* and *Comedy* entities. There is a new column *isRestricted* which has a value only for entities *Adult*. The same entities do not have value in column *isAdult* and they have an incremented version property.

The operation renamed the property in 133,778 entities.

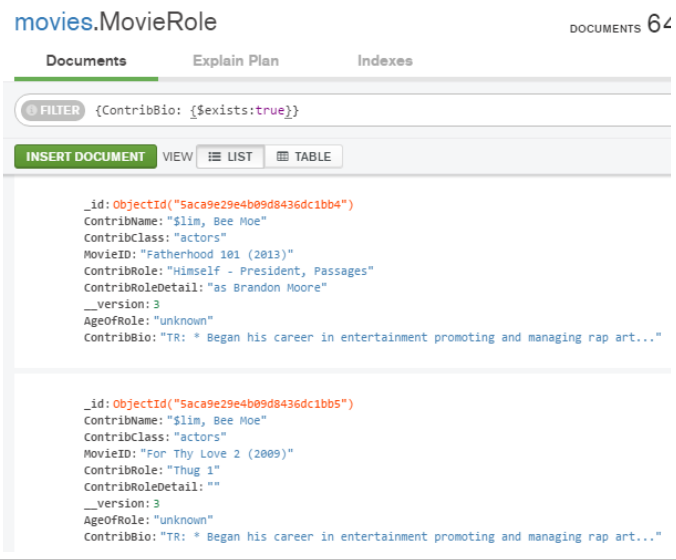
7.3.2 Inter-model Operations Experiments

The inter-model operations are *move* and *copy*. We also include operation *reference* into this section because we want to demonstrate the behavior of the inter-model reference propagation. We execute four experiments for inter-model operations and one experiment for references which consists of two sub-experiments.

Experiment Q7

```
> copy document.Contributor.ContribBio to document.MovieRole where
    document.Contributor.ContribName = document.MovieRole.ContribName
```

The next requirement is to have a property *ContribBio* from entities *Contributor* in *MovieRole* because it is often required when *MovieRole* entities are loaded



```
{ContribBio: {$exists:true}}
```

```
_id: ObjectId("5aca9e29e4b09d8436dc1bb4")
ContribName: "$lim, Bee Moe"
ContribClass: "actors"
MovieID: "Fatherhood 101 (2013)"
ContribRole: "Himself - President, Passages"
ContribRoleDetail: "as Brandon Moore"
__version: 3
AgeOfRole: "unknown"
ContribBio: "TR: * Began his career in entertainment promoting and managing rap art..."

_id: ObjectId("5aca9e29e4b09d8436dc1bb5")
ContribName: "$lim, Bee Moe"
ContribClass: "actors"
MovieID: "For Thy Love 2 (2009)"
ContribRole: "Thug 1"
ContribRoleDetail: ""
__version: 3
AgeOfRole: "unknown"
ContribBio: "TR: * Began his career in entertainment promoting and managing rap art..."
```

Figure 7.21: The result of operation *copy* of property *ContribBio* from entity *Contributor* to entity *MovieRole* in the document model.

and joining entities on each data load to get one extra property is too expensive. The denormalization in the document model is a common habit so we choose to copy the value. Experiment Q7 covers the described experiment. During the execution of this operation, we found out a limitation of MongoDB which is not able to handle big data (~6,000,000) in *distinct()* operation. The error message is *"errmsg": "exception: distinct too big, 16mb cap"*. For that reason, we modified our implementation to limit the query to 1000 entities.

Figure 7.21 shows the result of experiment Q7. We calculated the result numbers of the non-modified algorithm: The operation copied the property from 458,107 entities *Contributor* which contain the property to 23,594,251 entities *MovieRole* without loading them to the application memory. The version property for *MovieRole* was incremented during data processing too.

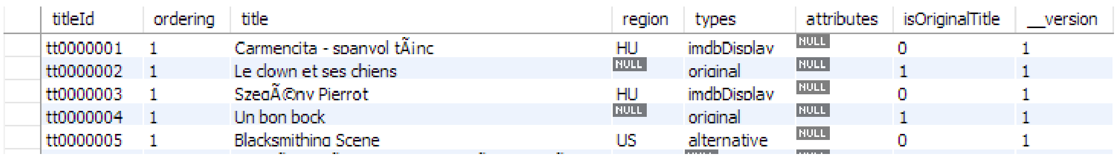
The manual migration is more complicated than migrations before so we prepared an example of the manual migration for MongoDB in Figure 7.22. The different part is the implementation of the where condition. The example selects all possible properties *ContribName* and matching *MovieRole*s and then it looks for entities *Contributor*. The operation is safe because it is a 1:N relationship.

Experiment Q8

```
> move relational.title_akas.language to relational.title_basics
   where relational.title_akas.titleId = relational.title_basics.
   tconst
```

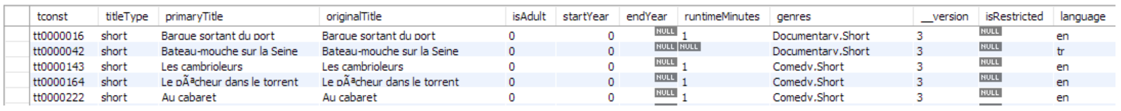
```
db.MovieRole.find({ContribName:{$in: db.Contributor.distinct(
    ContribName)}}).forEach(function(item){
        var value = "";
        db.Contributor.find({ContribName:item.ContribName}).forEach(
            function(u){value = u.ContribBio});
        db.MovieRole.update({_id:item._id},{$set:{ContribBio:value},
            $inc:{__version:1 },false,true);
});
```

Figure 7.22: An example of code which copies property *ContribBio* from *Contributor* to *MovieRole* in the document model.



titleId	ordering	title	region	types	attributes	isOriginalTitle	__version
tt0000001	1	Carmencita - spanvol tÃinc	HU	imdbDisplay	NULL	0	1
tt0000002	1	Le clown et ses chiens	NULL	original	NULL	1	1
tt0000003	1	SzegÃ©ny Pierrot	HU	imdbDisplay	NULL	0	1
tt0000004	1	Un bon bock	NULL	original	NULL	1	1
tt0000005	1	Blacksmithing Scene	US	alternative	NULL	0	1

(a) An example of entities *title_ akas* after operation *move.*



tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres	__version	isRestricted	language
tt0000016	short	Barque sortant du port	Barque sortant du port	0	0	NULL	1	Documentary,Short	3	NULL	en
tt0000042	short	Bateau-mouche sur la Seine	Bateau-mouche sur la Seine	0	0	NULL	NULL	Documentary,Short	3	NULL	tr
tt0000143	short	Les cambrioleurs	Les cambrioleurs	0	0	NULL	1	Comedy,Short	3	NULL	en
tt0000164	short	Le pÃªcheur dans le torrent	Le pÃªcheur dans le torrent	0	0	NULL	1	Comedy,Short	3	NULL	en
tt0000222	short	Au cabaret	Au cabaret	0	0	NULL	1	Comedy,Short	3	NULL	en

(b) An example of entities *title_ basics* after operation *move.*

Figure 7.23: An example of the final state of entities *title_ baciscs* and *title_ akas* after operation *move.*

The experiment request is to move property *language* from *title_ akas* to *title_ basics*. Experiment Q8 affects a single model so it will be executed as an intra-model operation within the model. We do not want to copy the property, because in the relational model it is usually not required to denormalize data.

In Figure 7.23 we show the final state after execution of experiment Q8. The operation moved the property from all 2,084,595 *title_ akas* to matching entities *title_ basics*. It also increments the version property for both types of entities. The whole operation is executed in the model so there is no entity loaded into the application memory.

Figure 7.24 shows an example of the manual migration with the version property management.

Experiment Q9

```
> copy relational.name_basics.knownForTitles to document.MovieRole
    where relational.name_basics.primaryName = document.MovieRole.
    ContribName
```

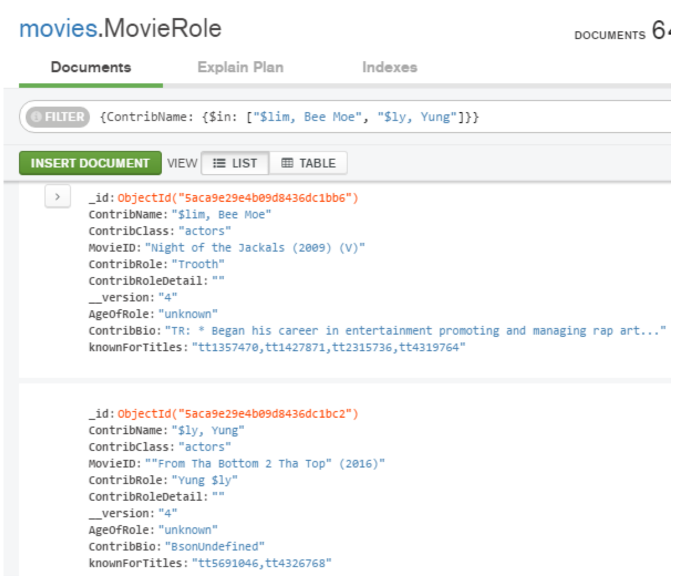
```
ALTER TABLE `title_basics` ADD `language` VARCHAR(255);
UPDATE `title_akas`,`title_basics` SET `title_basics`.`language` = `
    title_akas`.`language` WHERE `title_akas`.`titleId`= `title_basics
    `.`tconst`;
ALTER TABLE `title_akas` DROP `language`;
IF NOT EXISTS(SELECT 1 FROM information_schema.COLUMNS WHERE
    TABLE_NAME = 'title_basics' AND COLUMN_NAME = '__version')
THEN
        ALTER TABLE `title_basics` ADD `__version` INT DEFAULT 1;
END IF;
IF NOT EXISTS(SELECT 1 FROM information_schema.COLUMNS WHERE
    TABLE_NAME = 'title_akas' AND COLUMN_NAME = '__version')
THEN
        ALTER TABLE `title_akas` ADD `__version` INT DEFAULT 1;
END IF;
UPDATE `title_akas`,`title_basics` SET `title_basics`.`__version` = `
    title_basics`.`__version` + 1, `title_akas`.`__version` = `
    title_akas`.`__version` + 1 WHERE `title_akas`.`titleID`= `
    title_basics`.`tconst`;
```

Figure 7.24: An example of code which moves property *language* from *title_akas* to *title_basics* in the relational model.

The next experiment is a requirement to copy a property *knownForTitles* from entities *name_basics* in the relational model to *MovieRole* in the document model. To match relevant entities we use the name property as is shown in experiment Q9. Since we used separate exports to the relational and the document models, the property name has a different format. For example, in the document model the property name is *$lim, Bee Moe* and in the relational model it is *Bee Moe $lim*. In this situation, we cannot map entities so we decided to manually modify 2 properties name in the relational model to be able to execute the operation. The number of matching entities is not important for our experiments.

Figure 7.25 shows an example of the moved property. Entities contain the new property *knownForTitles* and on increased version property. Since we prepared 2 entities in the relational model we modified just 16 matching entities in the document model. So the total number of modified entities is 16 but the operation is inter-model so all entities have to be loaded to the multi-model engine. The number of loaded entities from the relational model is 8,534,771. From the document model it has to load 64,424,283. The total number is 72,959,054 loaded entities.

In this case, the manual migration is complex and we cannot easily show a script for it. The algorithm which we introduced in the proof of concept section (Algorithm 7.1) is the best way how to describe it. In other words, developers have to create a migration application which loads entities and transforms them.



```
{ContribName: {$in: ["$lim, Bee Moe", "$ly, Yung"]}}

_id: ObjectId("5aca9e29e4b09d8436dc1bb6")
ContribName: "$lim, Bee Moe"
ContribClass: "actors"
MovieID: "Night of the Jackals (2009) (V)"
ContribRole: "Trooth"
ContribRoleDetail: ""
__version: "4"
AgeOfRole: "unknown"
ContribBio: "TR: * Began his career in entertainment promoting and managing rap art..."
knownForTitles: "tt1357470,tt1427871,tt2315736,tt4319764"

_id: ObjectId("5aca9e29e4b09d8436dc1bc2")
ContribName: "$ly, Yung"
ContribClass: "actors"
MovieID: ""From Tha Bottom 2 Tha Top" (2016)"
ContribRole: "Yung $ly"
ContribRoleDetail: ""
__version: "4"
AgeOfRole: "unknown"
ContribBio: "BsonUndefined"
knownForTitles: "tt5691046,tt4326768"
```

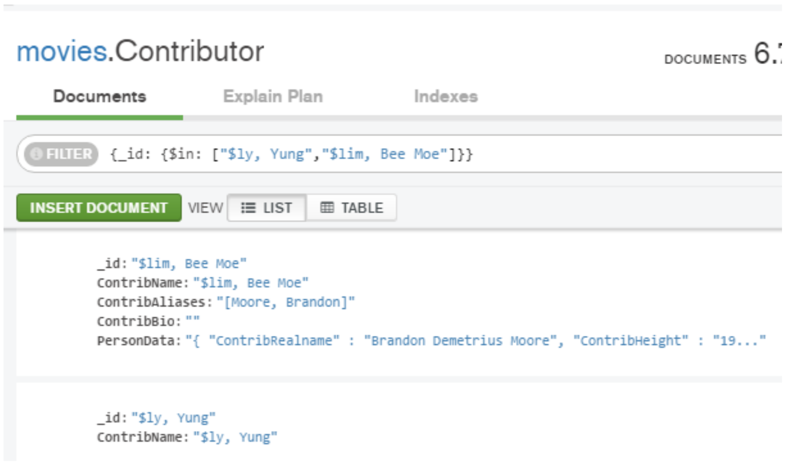
Figure 7.25: The result of operation *copy* of property *knownForTitles* from entities *name_ basics* in the relational model to entity *MovieRole* in the document model.

Experiment Q10

```
> move document.Contributor.ContribBio to relational.name_basics
    where document.Contributor._id = relational.name_basics.
    primaryName
```

This experiment moves property *ContribBio* from entities *Contributor* in the document model to entities *name_ basics* in the relational model. In the previous experiment we changed the name property to be able to match entities across models and we use this matching again.

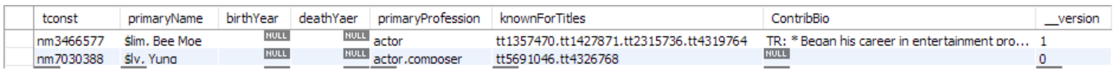
In Figure 7.26 we show the result of experiment Q10. The operation matches two entities in the document model and two entities in the relational model. But as we can see in the result, only one entity in the relational model contains the new property *ContribBio*. It is because only one entity in the document model had the property filled. The operation modifies two entities but it loads 6,693,529 entities *Contributor* and 8,534,771 entities *name_ basics* to the multi-model engine in the application memory. The total number of loaded entities is 15,225,300 .



```
{_id: {$in: ["$ly, Yung","$lim, Bee Moe"]}}
```

```
_id:"$lim, Bee Moe"
ContribName:"$lim, Bee Moe"
ContribAliases:"[Moore, Brandon]"
ContribBio:""
PersonData:"{ "ContribRealname" : "Brandon Demetrius Moore", "ContribHeight" : "19..."

_id:"$ly, Yung"
ContribName:"$ly, Yung"
```

(a) An example of entities *Contributor* after operation *move*.



	tconst	primaryName	birthYear	deathYaer	primaryProfession	knownForTitles	ContribBio	__version
	nm3466577	$lim. Bee Moe	NULL	NULL	actor	tt1357470.tt1427871.tt2315736.tt4319764	TR: * Began his career in entertainment pro...	1
	nm7030388	$lv. Yung	NULL	NULL	actor.composer	tt5691046.tt4326768	NULL	0

(b) An example of entities *name_ basics* after operation *move*.

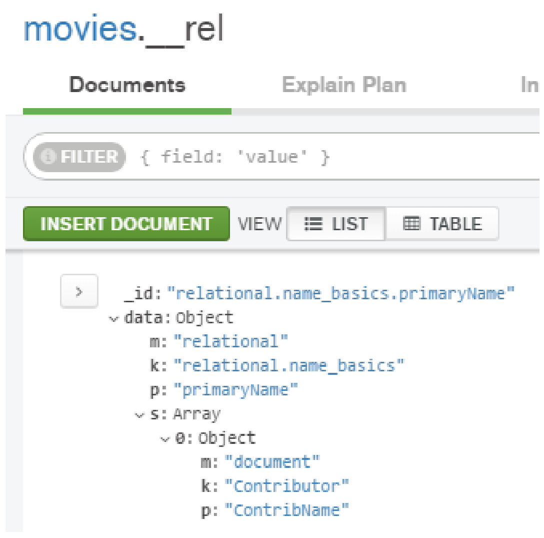
Figure 7.26: An example of the final state of entities *name_ basics* and *Contributor* after operation *move*.

Experiment Q11

```
> reference relational.name_basics.primaryName by document.
    Contributor.ContribName
> rename document.Contributor.ContribName to PrimaryName
> delete relational.name_basics.primaryName
```

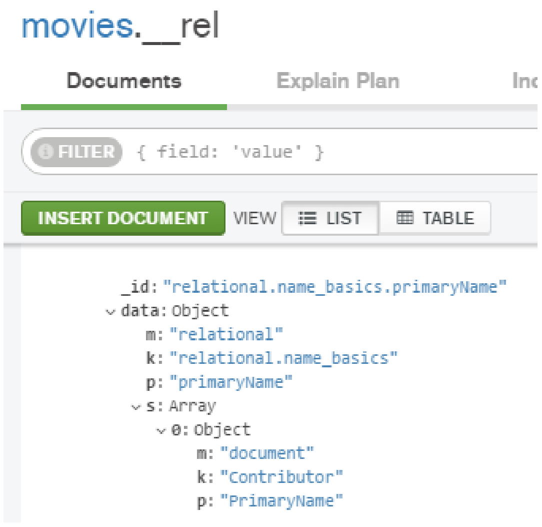
The last experiment focuses on references and it consists of two parts: renaming and deletion. The experiment simulates a real-world scenario when we have references in the MMD that have to evolve in time. In the experiment, we have to create a reference from *Contributor.ContibName* to the relational model specifically to *name_ basics.primaryName*. The next step is renaming of property *ContribName* to *PrimaryName* and, as the last step, we delete property *primaryName* from the relational entities. That triggers deletion of property *PrimaryName* which is referencing them from the document model. Experiment Q11 covers the experiment and it consists of three operations. The first one creates the reference, the second one changes the name of the referencing property and the last one deletes references property.

The first operation does not affect any entities so there is no need for manual migration. It creates a persistent reference in the reference store model. The second operation renames property *ContribName* in entities *Contributor*. The last operation of Q11 is the most interesting for us. If there was no reference from



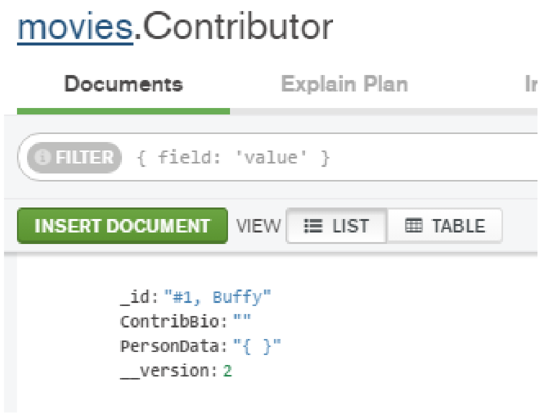
```
_id: "relational.name_basics.primaryName"
data: Object
    m: "relational"
    k: "relational.name_basics"
    p: "primaryName"
    s: Array
        0: Object
            m: "document"
            k: "Contributor"
            p: "ContribName"
```

(a) An example of a created reference.



```
_id: "relational.name_basics.primaryName"
data: Object
    m: "relational"
    k: "relational.name_basics"
    p: "primaryName"
    s: Array
        0: Object
            m: "document"
            k: "Contributor"
            p: "PrimaryName"
```

(b) An example of automatically changed reference after renaming the referencing property.



```
_id: "#1, Buffy"
ContribBio: ""
PersonData: "{ }"
__version: 2
```

(c) An example of the entity *Contributor* after deletion of the referenced property.

Figure 7.27: An example of the reference usage in the MMSEL.

the previous steps, then it would be operation *delete* which we described before. The reference makes it more complex especially for manual migration. Figure 7.27 shows the evolution of the persisted reference in the document model and an example of entity *Contributor* in the final state after deletion of the referenced property.

First of all, let us calculate numbers of affected and targeted entities of the operation *delete*. From the relational model, it deletes the property from 8,534,771 entities and then the referencing property from 6,693,529 entities in the document model (15,225,300 in total).

The problem of the manual migration is that developers have to know that there is the inter-model reference. Otherwise the data would get to an inconsistent state in the MMD. They have to know which models are affected and prepare the appropriate migration scripts for them. The MMSEL hides this logic and takes care of it.

Summary of Real-world Experiments

In each experiment, we have already mentioned some pros and cons of the specific approaches. We do not want to duplicate this information so we will focus on their common features.

Now, we can compare operations which would be needed in the real-world manual scenarios against the MMSEL. In the executed experiments, we always mentioned how a manual migration could be done. It is good to have an idea about it, but in a real-world usage of the MMD, it is not always possible. We can be limited for example by lack of commands for modifications of the specific model or the model language is not well-known by the developers. For that reason, we assume that migrations could be also executed by loading each entity to application memory of a migration application, modified, and then stored back to the MMD.

We summarize both approaches as the worst-case scenarios: (1) loading to the application memory, and (2) the best-case scenario when developers can execute an optimal migration script. In Table 7.1 we compare them with (3) the optimized implementation of the MMSEL which delegates intra-model operations to the affected model.

Let us briefly discuss the table. Operations Q1-Q6 are intra-model and Q7-Q10 are inter-model. Operation Q11 demonstrates usage of references in the MMSEL. For inter-model operations we do not provide information about the best-case scenarios because properties have to be loaded and transformed into the application memory anyway. In these cases, it would be the same as the worst-case scenarios.

For each approach, we calculated two important numbers. The first one is the number of read/write operations (R/W Op.) which is a number of entities which are required to load in the application memory to be able to execute the migration

Operation	**Target Ent.**	**R/W op.**	**Affected Ent.**	**R/W Op.**	**Affected Ent.**	**R/W Op.**	**Affected Ent.**
		Worst-case		**Best-case**		**MMSEL**	
Q1	64,424,283	64,424,283	64,424,283	0	64,424,283	0	64,424,283
Q2	149,636	149,636	4,920,457	0	4,920,457	0	4,920,457
Q3	7,504,078	7,504,078	7,504,078	0	7,504,078	0	7,504,078
Q4	4,920,457	4,920,457	4,920,457	0	4,920,457	0	4,920,457
Q5	56,920,205	56,920,205	56,920,205	0	56,920,205	0	56,920,205
Q6	133,778	133,778	133,778	0	133,778	0	133,778
Q7	23,594,251	24,052,358	23,594,251	0	23,594,251	0	23,594,251
Q8	4,169,190	7,005,052	7,005,052	0	7,005,052	0	7,005,052
Q9	16	72,959,054	16	N/A	N/A	72,959,054	16
Q10	2	15,225,300	8,534,771	N/A	N/A	15,225,300	8,534,771
Q11[a]	8,534,771 + 6,693,529[b]	15,225,300	15,225,300	0	15,225,300	0	15,225,300

[a] Numbers for the last delete operation.

[b] Entities reference the deleted property. The referencing properties are deleted during the operation.

Table 7.1: A summary of the worst-case manual migration.

operation. The second one is the number of affected entities after execution of the migration operation.

The table shows that all chosen approaches modify the same number of entities which is higher than the number of target entities. It is caused by the relational model where a new property always affects all entities of the kind.

In all experiments, worst-case scenarios have to load all needed entities to the application memory. The loading of entities from the MMD to the application memory is a very expensive operation from the performance point of view. If developers choose a worst-case scenario, the execution will be slow no matter if it is an intra- or an inter-model operation.

The best-case scenarios and the MMSEL scenarios execute intra-model operations directly in the affected model so we do not need to load entities into the application memory. The next fact is that the best-case scenarios always hit the minimum of affected entities but at the price of the complexity of the migration script, the knowledge of the migration languages of each model, etc.

The usage of the MMSEL is much easier and in our experiments, it is as effective as the best-case scenarios. But, for example, if we remove the where condition from operation Q5, renaming is the same but the version property could be updated in all entities and the MMSEL would be less effective than the best-case scenario. We said that it could happen because the MMSEL does not specify behavior for this use-case when an operation targets an entity without modifying the property.

To sum up, based on the result we can see that the MMSEL is much easier and understandable for developers than the usage of the best-case scenarios and the effectiveness is the same if we use the language correctly. Unlike the worst-case scenarios, it loads entities to the application memory only when it is needed for inter-model operations.

8. Conclusion

The goal of this thesis was to introduce a general approach for schema evolution in NoSQL databases which is reasonably complex yet would cover standard evolution requirements. We have researched, studied and evaluated a wide range of existing technologies and researches. Based on this we focused on general schema evolution in multi-model databases (MMDs) because have not found any existing approach or technology for them. We chose to use an existing proposal for NoSQL document databases (Section 3.3.1) and built our approach on top of it. We used the proposed language as a common interface for specific models. This decision gave us an idea of required operations and cornerstone for our solution but we had to accept its restrictions and limitations as well.

The main part of our effort consisted of research on induction of basic set of migration operations *add*, *delete*, *rename*, *move*, and *copy* for abstract model of MMDs (Section 5.1). The model consists of a model-independent layer and a model-specific layer. We introduced an idea of an extension of the multi-model schema evolution language (MMSEL) with inter-model references (Section 5.1.4). As a part of the solution we extended a basic approach and we optimized it so that it delegates intra-model operations to the target models. This behavior optimizes data transfer between model-specific and model-independent layers.

As the next step, we prepared an abstract prototype of the MMSEL (Chapter 6) to demonstrate and verify its benefits and usage. We used MongoDB and MariaDB for implementation of specific models. During the execution of experiments, we discovered some limitation of the prototype for big data (Section 7.3.2). Experimental results presented in the last chapter illustrate that, (1) the language is able to correctly migrate all targeted entities with the optimal number of read/write operations, (2) the syntax of our language is much simpler than manual migration scripts, (3) it enables execution of migrations safely based on the version field, and (4) the language provides an opportunity for reference management (both intra- and inter-model references). Naturally, there is still a space for improvements and extensions which will be discussed in the next section. However, as far as we found out, currently, there is no other solution for the schema evolution in MMDs.

8.1 Remaining Open Challenges in Multi-model Schema Evolution

Despite all our effort, there still remains space for further research, especially in the following areas.

8.1.1 Query Evolution in the MMD

The inter-query evolution is one of the biggest open challenges in MMDs because there is no unified query language. Partially we touched this problem when we suggested a solution of reference migration. The fact is that query migration is more complex and complicated. It requires better case analyses and introducing of the inter-model query language. Nowadays, Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux introduce a query language SQL++ [56]. The language is an extension of SQL and it unifies query languages for a multiple SQL and NoSQL databases. We suggest SQL++ as an adept for a unified query language for MMDs which can be used for the query evolution.

8.1.2 Schema Evolution in Graph Model

One of the possible improvements of our work can be a proper definition and implementation of the DSEL. We decided to transform the graph model into the document model. The used approach can decrease effectivity of graph operations which are executed on top of the document model and ban some native graph operations. Introducing of a solution for this open challenge can improve the performance of schema evolution in MMDs.

8.1.3 Lazy Implementation of Multi-model Schema Evolution

Our implementations of schema evolution in MMDs is eager. It can also be implemented as lazy. The benefit of lazy approach is that we do not have to migrate data which are not used. However, the problem of using lazy features in an uncontrolled manner is that we cannot predict when data are migrated and from which version.

8.1.4 Transactions in MMDs

In general, inter-model transactions have to be managed in the model independent layer by the multi-model engine which can delegate operations to the model specific layer. Each model can provide the full support of transactions (i.e. rollbacks, persistence, etc.). The engine should create a transaction on top of the model specific layer and control execution of operations. Currently, there is no general approach and the challenge remains open.

8.1.5 Reference Evolution in MMDs

We mentioned it in Section 5.1.4 that we have not finished references completely. We excluded chains of references which should be added to the MMSEL.

8.1.6 Validation of Operations *copy* and *move*

This is a problem inherited from the NoSQLSEL. The NoSQLSEL and MMSEL requires relationship 1:N for operations *copy* and *move* to ensure a defined result. The open challenge is how to verify if entities in the executing operation moves/copies are in the required relationship.

Bibliography

[1] Apache CouchDB, 2018. URL http://couchdb.apache.org/. [Online; Accessed 2-February-2018].

[2] Apache™ Hadoop, 2018. URL http://hadoop.apache.org/. [Online; Accessed 2-February-2018].

[3] ArangoDB, 2018. URL https://www.arangodb.com/. [Online; Accessed 2-February-2018].

[4] Bob Bryla and Kevin Loney. *Oracle Database 12C The Complete Reference*. McGraw-Hill Osborne Media, 1st edition, 2013. ISBN 0071801758, 9780071801751.

[5] BSON. Specification Version 1.1, 2018. URL http://bsonspec.org/spec.html. [Online; Accessed 2-February-2018].

[6] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1):30–76, March 1996. ISSN 0362-5915. doi: 10.1145/227604.227609. URL http://doi.acm.org/10.1145/227604.227609.

[7] Cassandra, 2018. URL http://cassandra.apache.org/. [Online; Accessed 2-February-2018].

[8] Cassandra Query Language, 2018. URL http://cassandra.apache.org/doc/latest/cql/. [Online; Accessed 2-February-2018].

[9] ControVol. ControVol, 2016. URL https://sites.google.com/site/controvolplugin/home. [Online; Accessed 18-April-2016].

[10] Couchbase, 2018. URL https://www.couchbase.com/. [Online; Accessed 2-February-2018].

[11] CrateDB, 2018. URL https://crate.io/. [Online; Accessed 2-February-2018].

[12] Cypher Query Language, 2018. URL https://neo4j.com/developer/cypher-query-language/. [Online; Accessed 2-February-2018].

[13] Dare Obasanjo's weblog. Building Scalable Databases: Denormalization, the NoSQL Movement and Digg, 2018. URL http://www.25hoursaday.com/weblog/2009/09/10/

BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.
aspx. [Online; Accessed 2-February-2018].

[14] DataStax, 2018. URL https://www.datastax.com/. [Online; Accessed 2-February-2018].

[15] DB Engines. DB Engines, 2017. URL https://db-engines.com/en/ranking. [Online; Accessed 05-May-2017].

[16] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 0124160441, 9780124160446.

[17] ECMAScript 2017 Language Specification. Ecma International, Geneva, 2018. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. [Online; Accessed 2-February-2018].

[18] Ehcache, 2018. URL http://www.ehcache.org/. [Online; Accessed 2-February-2018].

[19] Martin Fowler. Polyglot persistence. ONLINE, November 2011. URL https://martinfowler.com/bliki/PolyglotPersistence.html.

[20] Gartner. Magic Quadrant for Operational Database Management Systems, 2018. URL https://www.gartner.com/doc/3467318/magic-quadrant-operational-database-management. [Online; Accessed 2-February-2018].

[21] Giraph, 2018. URL http://giraph.apache.org/. [Online; Accessed 2-February-2018].

[22] Git, 2018. URL https://git-scm.com/. [Online; Accessed 2-February-2018].

[23] Google App Engine Datastore. Google App Engine Datastore Java API, 2016. URL https://developers.google.com/appengine/docs/java/datastore/. [Online; Accessed 18-April-2016].

[24] James Groff and Paul Weinberg. *SQL The Complete Reference, 3rd Edition.* McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2010. ISBN 0071592555, 9780071592550.

[25] HBase, 2018. URL https://hbase.apache.org/. [Online; Accessed 2-February-2018].

[26] Hibernate OGM. Hibernate ogm, 2016. URL `http://hibernate.org/ogm/`. [Online; Accessed 18-April-2016].

[27] IMDb Interfaces, 2018. URL `https://www.imdb.com/interfaces/`. [Online; Accessed 2-February-2018].

[28] Importing IMDB data into MongoDB: follow the instruction of MICHAEL HAVEY, 2018. URL `http://log4idea.blogspot.cz/2015/05/importing-imdb-data-into-mongodb-follow.html`. [Online; Accessed 2-February-2018].

[29] Infinispan, 2018. URL `http://infinispan.org/`. [Online; Accessed 2-February-2018].

[30] Internet Movie Database, 2018. URL `http://www.imdb.com/`. [Online; Accessed 2-February-2018].

[31] JSON. ECMA-404 The JSON Data Interchange Standard, 2018. URL `http://json.org/`. [Online; Accessed 2-February-2018].

[32] Matthias Kurz. Bpmn model interchange: The quest for interoperability. In *Proceedings of the 8th International Conference on Subject-oriented Business Process Management*, S-BPM '16, pages 6:1–6:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4071-7. doi: 10.1145/2882879.2882886. URL `http://doi.acm.org/10.1145/2882879.2882886`.

[33] Ralf Lämmel. Google's mapreduce programming model — revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2007.07.001. URL `http://www.sciencedirect.com/science/article/pii/S0167642307001281`.

[34] Jiaheng Lu and Irena Holubová. Multi-model data management: What's new and what's next? In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 602–605. OpenProceedings.org, 2017. ISBN 978-3-89318-073-8. doi: 10.5441/002/edbt.2017.80. URL `http://dx.doi.org/10.5441/002/edbt.2017.80`.

[35] Jiaheng Lu, Zhen Hua Liu, Pengfei Xu, and Chao Zhang. UDBMS: road to unification for multi-model data management. *CoRR*, abs/1612.08050, 2016. URL `http://arxiv.org/abs/1612.08050`.

[36] Jakub Malý, Irena Mlýnková, and Martin Nečaský. Xml data transformations as schema evolves. *Advances in Databases and Information Systems*, pages 375–388, 2011. doi: 10.1007/978-3-642-23737-9_27. URL http://dx.doi.org/10.1007/978-3-642-23737-9_27.

[37] MariaDB, 2018. URL https://mariadb.org/. [Online; Accessed 2-February-2018].

[38] MarkLogic, 2018. URL http://www.marklogic.com/. [Online; Accessed 2-February-2018].

[39] MediaWiki, 2018. URL https://www.mediawiki.org/wiki/MediaWiki. [Online; Accessed 2-February-2018].

[40] Memcached, 2018. URL https://memcached.org/. [Online; Accessed 2-February-2018].

[41] Microsoft. Repository pattern. ONLINE, jun 2016. URL https://msdn.microsoft.com/en-us/library/ff649690.aspx.

[42] MongoDB, 2018. URL https://www.mongodb.com/. [Online; Accessed 2-February-2018].

[43] MongoDB Compass, 2018. URL https://www.mongodb.com/products/compass. [Online; Accessed 2-February-2018].

[44] MongoDB Drivers, 2018. URL https://docs.mongodb.com/ecosystem/drivers/. [Online; Accessed 2-February-2018].

[45] Mongoid, 2018. URL https://github.com/mongodb/mongoid. [Online; Accessed 2-February-2018].

[46] Mongoid Evolver, 2018. URL https://github.com/mongoid/evolver. [Online; Accessed 2-February-2018].

[47] MongoRepository, 2018. URL https://github.com/RobThree/MongoRepository/. [Online; Accessed 2-February-2018].

[48] Morphia. Morphia, 2016. URL http://mongodb.github.io/morphia/. [Online; Accessed 18-April-2016].

[49] MySQL Workbench, 2018. URL https://www.mysql.com/products/workbench/. [Online; Accessed 2-February-2018].

[50] MySql.Data, 2018. URL https://www.nuget.org/packages/MySql.Data/. [Online; Accessed 2-February-2018].

[51] Martin Nečaský, Jakub Klímek, Jakub Malý, and Irena Mlýnková. Evolution and change management of xml-based systems. *Journal of Systems and Software*, 85(3):683 – 707, 2012. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2011.09.038. URL `http://www.sciencedirect.com/science/article/pii/S0164121211002524`. Novel approaches in the design and implementation of systems/software architecture.

[52] Neo4j, 2018. URL `https://neo4j.com/`. [Online; Accessed 2-February-2018].

[53] .NET Framework. Microsoft .NET Framework 4.6.2, 2018. URL `https://www.visualstudio.com/vs`. [Online; Accessed 2-February-2018].

[54] NoSQL Databases, 2018. URL `http://nosql-database.org/`. [Online; Accessed 2-February-2018].

[55] Objectify AppEngine. Migrating schemas. `https://github.com/objectify/objectify/wiki/SchemaMigration/`, 2016. [Online; Accessed 18-April-2016].

[56] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The sql++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. 05 2014.

[57] OrientDB, 2018. URL `http://orientdb.com/orientdb/`. [Online; Accessed 2-February-2018].

[58] Marek Polák, Martin Chytil, Karel Jakubec, Vladimír Kudelas, Peter Piják, Martin Nečaský, and Irena Holubová. Data and query adaptation using daemonx. *COMPUTING AND INFORMATICS*, 34(1), 2015. URL `http://www.cai.sk/ojs/index.php/cai/article/view/2040`.

[59] RavenDB. Denormalized References, 2018. URL `https://ravendb.net/docs/article-page/2.5/csharp/faq/denormalized-references`. [Online; Accessed 2-February-2018].

[60] Redis, 2018. URL `https://redis.io/`. [Online; Accessed 2-February-2018].

[61] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 2012. ISBN 0321826620. URL `http://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0321826620`.

[62] Karla Saur, Tudor Dumitras, and Michael W. Hicks. Evolving nosql databases without downtime. *CoRR*, abs/1506.08800, 2015. URL `http://arxiv.org/abs/1506.08800`.

[63] S. Scherzinger, M. Klettke, and U. Störl. Managing Schema Evolution in NoSQL Data Stores. *ArXiv e-prints*, August 2013.

[64] Stefanie Scherzinger, Thomas Cerqueus, and Eduardo Cunha de Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1464–1467. IEEE Computer Society, 2015. ISBN 978-1-4799-7964-6. doi: 10.1109/ICDE.2015.7113402. URL `http://dx.doi.org/10.1109/ICDE.2015.7113402`.

[65] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Cleager: Eager schema evolution in nosql document stores. In *BTW*, pages 659–662, 2015.

[66] Stefanie Scherzinger, Uta Störl, and Meike Klettke. A datalog-based protocol for lazy data migration in agile nosql application development. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 41–44, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3902-5. doi: 10.1145/2815072.2815078. URL `http://doi.acm.org/10.1145/2815072.2815078`.

[67] Michael Sperberg-McQueen, Tim Bray, François Yergeau, Eve Maler, and Jean Paoli. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[68] S. Tiwari. *Professional NoSQL*. EBL-Schweitzer. Wiley, 2011. ISBN 9781118167809. URL `https://books.google.cz/books?id=tv5iO9MnObUC`.

[69] Visual Studio IDE. Visual Studio Community 2017, 2018. URL `https://www.visualstudio.com/vs`. [Online; Accessed 2-February-2018].

[70] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007. ISBN 0596006349.

[71] XSEM, 2018. URL `http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=xsem`. [Online; Accessed 2-February-2018].

List of Figures

List of Tables

List of Algorithms

A. Attachments

A.1 DVD Content

This thesis contains an attached DVD with the source code of the prototype of the MMSEL, an electronic version of this document, and database exports of the initial state of the abstract MMD for the proof of concept experiments. The disc contains the following directories:

- doc – contains this text in PDF format,
- exports – contains exported data from initial state of the MMD from the proof of concept experiments,
- MultiModelSchemaEvolution – contains the complete source code in the form of a solution in Visual Studio 2017.