Mortadelo: Automatic Generation of NoSQL Stores from Platform-Independent Data Models

Alfonso de la Vega, Diego García-Saiz, Carlos Blanco, Marta Zorrilla, and Pablo Sánchez

Software Engineering and Real-Time Group, University of Cantabria, Santander (Spain)

Abstract

In the last decade, several NoSQL systems have emerged as a response to the scalability problems manifested by classical relational databases when used in Big Data contexts. These NoSQL systems appeared first as physical-level solutions, initially lacking any design methodologies. After this initial batch of systems, several design methodologies for NoSQL have been recently created. Nevertheless, most of these methodologies target just one NoSQL paradigm. In addition, as each methodology uses a different conceptual modeling approach, NoSQL database designers would need to remake conceptual models as they switch from one NoSQL paradigm to another. Moreover, most of these design processes provide just a set of design heuristics and guidelines that database designers need to apply manually, which can be a time-consuming and error-prone process. To overcome these limitations, this article presents *Mortadelo*, a model-driven NoSQL database design process where, from a high-level conceptual model, independent of any specific NoSQL paradigm, an implementation for a concrete NoSQL database system can be automatically generated. Moreover, this database generation process can be customized, so that some design trade-offs can be managed differently according to each context needs. We evaluated Mortadelo's capabilities by generating database implementations for several typical NoSQL case studies. In these cases, Mortadelo was able to generate implementations for the *Cassandra* and *MongoDB* NoSQL systems from the same conceptual data model. These implementations were similar to the ones generated by design methodologies specifically developed for a single paradigm. Therefore, design quality is not sacrificed by our approach in favor of generality.

Keywords: NoSQL, Database Design, Data Modeling, Model-Driven Engineering, Column Family Stores, Document Stores

1. Introduction

Nowadays, several kinds of software systems have pushed relational databases to their limits. Examples of these new kinds of applications are internet-scale applications, such as *Twitter* or *Amazon*; Internet of Things (IoT) applications, such as Smart Cities [1, 2]; Industry 4.0 systems [3, 4]; or Big Data systems [5, 6]. All these systems need to manage large volumes of data that are often distributed in several servers and assure low response times and high availability in the contexts of a high number concurrent requests. In these scenarios, relational databases have manifested different scalability problems [7].

In response to these limitations, a new generation of database management systems, denoted as *NoSQL systems* [8], started to offer some alternatives. Each one of these alternatives was designed for a specific purpose and following a different approach. So, NoSQL is not just a single alternative to relational databases, but a global term that comprises different database strategies, including, among others, document-oriented databases [9, 10], key-value stores [11, 12] or column family stores [13, 14]. Despite their differences, most NoSQL database systems rely on two common features: (1) they make use of *data denormalisation* to improve response times [15], and (2) they sacrifice some *ACID (Atomicity, Consistency, Isolation, and Durability)* properties [16, 7] to increase scalability, while providing other less restrictive properties but also useful in a best-effort approach, such as *eventual consistency* [17].

These NoSQL technologies emerged first at the implementation level and, consequently, they initially lacked well-defined design processes. Database design methodologies for relational databases [18], which are usually based on conceptual modelling notations such as ER (Entity-Relationship) [19] or UML (Unified Modeling Language) [20], revealed soon to be not enough for designing NoSQL databases. To take advantage of the benefits provided by data nesting and denormalisation, database designers need to take into account not only which data will be stored in the database, but also how these data will be accessed [21, 22]. In NoSQL systems, working with the same set of data, but with different data access patterns, might lead to different database implementations. This is due to the fact that, in many NoSQL systems, design decisions are driven by how data will be accessed. Traditional database design approaches do not provide an adequate support for these issues, mainly because they were created to satisfy other goals, e.g., the commented ACID properties [16].

To address this gap, several design methodologies for NoSQL systems have been created in the last years [23, 21, 24, 22]. Nevertheless, these approaches still present some limitations, which can be summarized as follows:

1. Each one of these approaches focuses on a concrete NoSQL paradigm, providing its own conceptual modelling languages and notations. This implies that the same con-

ceptual data model cannot be used to describe the same database in different NoSQL paradigms.

2. Most approaches describe how to design a NoSQL database by means of guidelines or heuristics that must be interpreted and applied manually by database designers. This can be an error-prone and time-consuming process. Just two approaches [21, 24] address design process automation and provide the basis for building CASE tools.
3. Those approaches that tackle automation often use the same strategy to transform the patterns they found at the conceptual level into constructs of the target database. Therefore, they neglect the existence of alternative strategies that might be more adequate in certain contexts, or when targeting different NoSQL paradigms, e.g., document-based or column family stores.

To overcome these limitations, we present *Mortadelo*, a model-driven development process for NoSQL database design. This process builds on previous work and goes one step further by being able to automatically generate implementations for different NoSQL paradigms from the same conceptual model. The generation process is achieved by means of model transformation and code generation techniques. Currently, we have created and implemented model transformation rules for supporting the generation of column family stores and document databases, but the framework could be extended to support other paradigms, such as key-value stores.

To evaluate the expressiveness and effectiveness of our approach, we used Mortadelo to model different case studies used as test-beds in the NoSQL literature. We compared the generated NoSQL databases with the databases obtained with state-of-the-art NoSQL design methodologies. The results of this evaluation process showed that, using Mortadelo, the same conceptual model can be transformed into either a column family database, implemented in Cassandra [25]; or a document database, expressed in MongoDB [9]. Moreover, the obtained databases were pretty similar to those generated by design methodologies devised specifically for one NoSQL paradigm. In some cases, our approach performed even better, and, in one case, our designs might not be as good as the ones generated by other approaches. Moreover, our approach offers several transformation alternatives, so the same conceptual model might be handled differently depending on each concrete context. This feature is scarcely supported by NoSQL design methodologies.

The remaining of this article is structured as follows. Section 2 presents the running example used throughout the paper, and introduces to the used NoSQL technologies. Next, in Section 3, related works are discussed. In Section 4, we detail the different phases of the transformation process followed by Mortadelo to generate NoSQL databases. Section 5 includes the evaluation of Mortadelo. Lastly, we expose our conclusions and future work in Section 6.

2. Background

To make this article self-contained, this section provides some background of the used technologies, i.e., column family and document data stores. Before describing these technologies, we introduce the running example that we used to illustrate the different concepts that appear in this work.

2.1. Running Example

We used as running example throughout this paper a database that represents an e-commerce platform. This database resembles the structure that can be found in existing online stores, such as Amazon.

Figure 1 shows a conceptual data model, in UML notation, for the e-commerce platform. As it can be seen, this platform manages *Products* and *Clients* as main entities. *Products* can belong to different *Categories*, and they can be supplied by different *Providers*. *Clients* make purchases of these products. Each *Purchase* has an associated shipping *Address* and, optionally, a *Bill*. Clients can request several products inside the same purchase. Each request is represented by a different *PurchaseLine*, that specifies how many items of a purchased product were acquired in that purchase.

The e-commerce application that clients use to make their purchases employs data from the conceptual model of Figure 1. These data are retrieved according to the following patterns:

Q1 Get all data of a *Product*, given a *productId*.

Q2 Get all data of a *Product*, including its *Categories*, given the product's *name*.

Q3 Get *name* and *price* of a set of *Products* from a *Category*, given the *name* of this category, and ordered by *price*.

Q4 Get the *Purchases* of a *Client* happening in the last three months of a given *year*, including the purchased *Products* and the postal code of the shipping *Address*.

Q5 Get the *Purchases* performed in a given *year*, including *Billing* data.

Q6 Get the *Purchases* performed in a given *year*, including their *postal code*.

Q7 Get the *Purchases* of a given *Client* in a concrete *month* of a *year*, including *Product* data.

In terms of adding data to the system (e.g. writing operations), it must be pointed out that new clients and products are scarcely added to the system when compared to other database operations such as reads; whereas new purchases are much more frequently added.

2.2. Column Family Stores

Column family stores, also known as extensible record stores [26], aim to improve scalability and read performance by promoting denormalization and the distribution of data in different physical locations or nodes. The main idea behind column family stores is that each query can be resolved by retrieving a well-located bundle of data, without having to perform complex operations to combine data spread in several of these nodes.

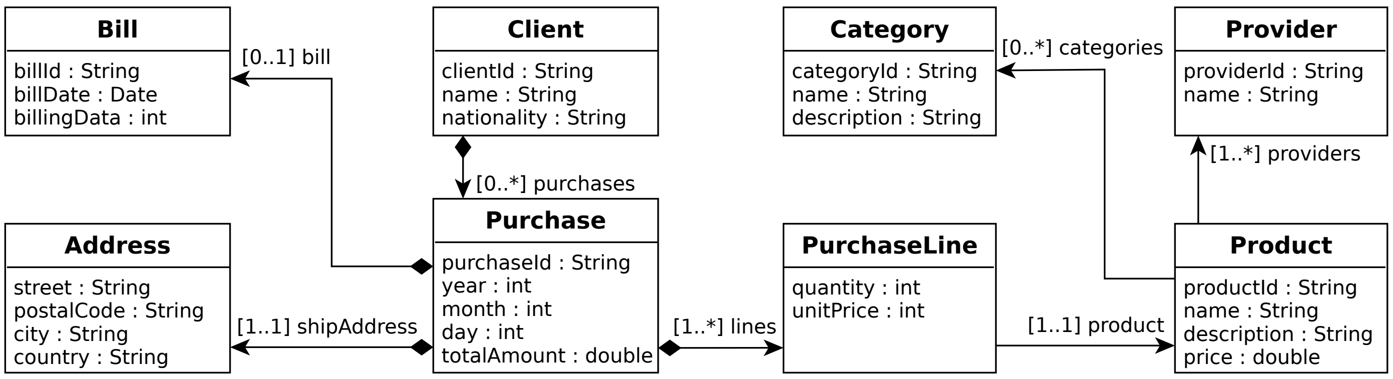


Figure 1: Conceptual data model of the e-commerce platform.

This kind of store organize data in tables, where each row contains a set of column names and values. Each row is identified by a *primary key*, which can be composed of one or more columns.

In column family stores, primary keys are selected not only to provide row uniqueness, but also to assure that some sets of rows are stored in the same physical node, so that these rows can be easily retrieved together when needed. For this purpose, column family stores like Cassandra [25] and ScyllaDB [27] often divide the primary key into two different subsets: the *partition key* and the *clustering key*. The *partition key* is the subset of the primary key used to distribute rows across physical nodes, ensuring that all rows with the same partition key are stored in the same node. The *clustering key* is used to sort rows, so that ranges of these rows can be easily retrieved. This sorting happens at partition-key level, this is, elements that share the same partition key value are sorted according to their clustering key. Clustering keys are optional, so there might be tables that do not define it.

Figure 2 illustrates a table for retrieving purchases made between two dates that exploits the benefits of these elements. In this case, the columns `year`, `month` and `purchaseId` are chosen as a primary key. The `purchaseId` column by itself ensures row uniqueness, and the other two columns are included in the primary key to improve performance. The `year` column is selected as partition key, ensuring that all purchases made in the same year are hosted in the same node. The `month` and `purchaseId` columns would be used as clustering key, so rows would be sorted first by *month* and then by *purchaseId*. This makes easier, for instance, to retrieve all purchases corresponding to the second semester of the last year.

For performance reasons, some column stores, such as *Cassandra*, limit query access to the set of rows associated to a single partition key, not being able to retrieve data from several partitions in a single query. For instance, for the column family of Figure 2, we could not retrieve purchases made in the last three years using a single query; we would need three queries instead. To ensure this constraint is satisfied, column family stores check that each query includes in the conditional clause a comparison by equality for all the columns that are part of the partition key. Other kind of comparisons, such as *greater than* operators, are not allowed for partition key columns. This constraint does not extend to clustering key columns, where these operators can be used, and are often used indeed. Moreover, for performance reasons, columns not included in the partition or clustering keys cannot be used for comparisons in the conditional clause of a query.

In our case, these constraints imply that all queries against the column family of Figure 2 must have a condition clause like *where year = x*, being *x* the value of a particular year. Queries retrieving purchases *before* or *after* a year would not be permitted, but queries retrieving purchases *before* or *after* a month inside a specific year are supported. For instance, we can compose and execute a query with a clause like *where year = 2018*

year	**month**	**purchaseId**	**amount**	**clientId**	...
2018	January	291376	30.44$	437120	...
		376291	27.00$	418320	...
	February	137629	10.44$	418320	...
		376291	17.00$	371204	...
2019	January	291376	72.00$	120437	...
		913762	57.46$	371204	...

Figure 2: Column family that stores purchases information by year and month. Double lines separate partitions (i.e. changes in *year*), and single lines indicate a change in the clustering key (*month* and *purchaseId*).

clientId	**year**	**purchaseId**	**amount**	...
120437	2019	291376	72.00$	...
371204	2018	376291	17.00$	...
	2019	913762	54.46$	...
418320	2018	376291	27.00$	...
		137629	10.44$	...
437120	2018	291376	30.44$	...

Figure 3: Table for storing purchases information by client and year. Double lines separate partitions (i.e. changes in *clientId*), and single lines indicate a change in the clustering key (*year* and *purchaseId*).

```
db.createCollection("Products", {
  validator: { $jsonSchema: {
    bsonType: "object",
    required: ["productId", "name",
               "price", "categories"],
    properties: {
      productId: {bsonType: "int"},
      name: {bsonType: "string"},
      price: {bsonType: "decimal"},
      categories: {bsonType: "array"
        items: {bsonType: "int"}}}
  }}})

db.createCollection("Categories", {
  validator: { $jsonSchema: {
    bsonType: "object",
    required: ["categoryId", "name",
               "description"],
    properties: {
      categoryId: {bsonType: "int"},
      name: {bsonType: "string"},
      description: {bsonType: "string"}}
  }}})
```

Figure 4: MongoDB normalized example: products point to their categories.

and month ≥ *July*, which will recover all purchases in the second semester of 2018.

Column family stores do not provide support for performing joins between tables, i.e., it is not possible to specify a query that gathers data from two or more tables at the same time. Therefore, if two queries share the same subset of data, and we want each one of these queries to be performed in a single operation to the database, this shared subset of data must be replicated in the tables that provide support for these queries. For instance, if purchases need to be recovered sometimes *by year and month* and in other occasions *by client and year*, we might need to replicate data about these purchases across two column families: the one already commented in Figure 2, and the one depicted in Figure 3, where *clientId* is the partition key, and the clustering key is formed by *year* and *purchaseId*. This data replication introduces *denormalization* [15] in the structure of the column families.

Nevertheless, this denormalization has a side effect: it makes insertion of new data, or updates of existing ones, more expensive. In the previous example, each time a purchase is incorporated into the database, it would need to be inserted in the two column families supporting the purchases *by year* and *by client* queries. In these cases, it might be better to maintain purchases in a single table, making insertions and updates faster, and resolving the *by year* and *by client* queries by launching several queries to different tables and then joining the results at the application level.

Finally, it might be worth to point out that the concept of *column family* might differ between specific NoSQL technologies. So, in systems like Cassandra and ScyllaDB, a column family is defined as a table with a partition key, a clustering key and a set of columns for each row. Nevertheless, for systems such as HBase [28] or BigTable [29], a column family

```
db.createCollection("Products", {
    validator: { $jsonSchema: {
    bsonType: "object",
    required: ["productId", "name",
               "price", "categories"],
    properties: {
      productId: {bsonType: "int"},
      name: {bsonType: "string"},
      price: {bsonType: "decimal"},
      categories: {
        bsonType: "array"
        items: {
          bsonType: "object",
          required: ["name", "description"],
          properties: {
            name: {bsonType: "string"},
            description: {bsonType: "string"}
}}}}}}})
```

Figure 5: MongoDB de-normalized: categories are nested in the products.

is a mechanism to group columns of a table into disjoint sets that are never retrieved together. Let us suppose that, for each purchase, we store data about the purchased product, the payment method and the delivery address. In addition, these data blocks are never retrieved together. In this case, we can group the columns related to these data blocks in column families. The main advantage of this column grouping mechanism is that these data blocks, since they are never retrieved together, can be stored in different physical locations, which might help increase performance. In the context of this work, we will use the term column family in the Cassandra sense, this is, a column family is a table with a partition key, a clustering key and a set of columns, which might vary in number and size.

2.3. *Document Stores*

Document-oriented databases [26] aim to improve performance by storing as single pieces of data hierarchies of objects that are most likely to be retrieved together. These object hierarchies are known as *documents*. Documents are grouped in *collections*, whose relational counterpart might be *tables*. A collection stores documents of the same entity, such as *Products* or *Clients*. A document is typically composed of key-value pairs, and it can contain other embedded documents.

For example, in MongoDB [9], which is probably the most popular document-oriented database, documents are stored using a JSON-like structure, known as BSON[1]. MongoDB allows specifying the structure of a collection by means of a JSON Schema[2], which is registered in a validator that controls the insertion of data in a collection.

Figure 4 shows the specification in MongoDB of two collections from our running example. In this case, *Products* and *Categories* entities have their own collections, so they are stored as separate documents. As it can be observed, among different objects, a *product* stores an array of references to the categories

[1]See BSON specification in http://bsonspec.org/
[2]https://json-schema.org/

it belongs to. So, if we wanted to retrieve a product along with data of its categories, we would need to perform a *join* operation between the two collections. However, these operations are expensive, and therefore they are often not recommended.

To avoid requiring a join operation, and since it is expected that a product does not belong to a high number of categories, and categories do not have any relationship with other entities, we might opt for embedding categories data in the products collection. If we do so, each time a product is retrieved, data about its categories are also returned, which avoids having to perform joins with information contained in another collection. A collection specification where categories are nested inside the data of a product is shown in Figure 5.

Finally, it is worth to mention that document-oriented databases, such as MongoDB, do not impose any constraint on which elements of a document can be included in a query, unlike it happens in column family stores. Therefore, any query can have equality and inequality conditions over any field of the collections of a document database.

3. Related Work

As NoSQL systems emerged, different approaches addressing the design problems of these systems were created. These approaches are summarized in Table 1. We briefly describe each one of these approaches.

Li [23] presented one of the very first works on NoSQL database design. They proposed a set of high-level heuristics for refactoring relational databases into *HBase* ones. To produce a NoSQL database using this work, we would need to create a relational database first, and then to transform it to HBase, which could be a tedious process as compared to generating a database directly from a conceptual data model.

de Lima and dos Santos Mello [30] describe a design methodology for transforming an *ER (Entity Relationship)* model [19] into a document-oriented logical model, and then to MongoDB code. Authors highlight the importance of knowing how data will be accessed for designing a NoSQL database because, depending on this, some mapping strategies might be more suitable than others. For this reason, authors complement ER models with information about the expected *application workload*, which is specified using a technique from the XML community [31].

Chebotko et al. [21] present a similar work, but focusing on *Cassandra*. These authors also rely on ER models, which are transformed into a logical model, called *Chebotko* diagrams, specifically designed by the authors themselves for representing column stores. To specify data access patterns, ER models are enriched with *ERQL (Entity-Relationship Query Language)* queries [34]. Authors provide a set of rules for transforming each application query into a column family. Based on these rules, authors developed a CASE tool to automate the design process for Cassandra databases.

In Chebotko et al. [21], query patterns are always mapped following the same strategy. Nevertheless, as pointed out by Mior et al. [22], several alternative strategies might exist. For instance, if we create a column family per query, data can get highly replicated, which increases read performance. On the other hand, updates, insertions and deletions become more expensive. Therefore, if these operations are frequent, data replication might not pay off. To deal with these issues, Mior et al. [22] present *NoSE (NoSQL Schema Evaluator)*, a tool that accepts as input an ER conceptual model, a set of data access patterns, some statistical information about query frequency and expected volume of data. With these inputs, NoSE calculates all candidate implementations for each query and builds a *Binary Integer Programming* (BIP) optimization problem. After solving it, an optimal NoSQL plan for implementing a column family database is obtained. NoSE does not generate database code, although this step would be trivial.

Atzeni et al. [35] proposed a uniform API, independent of any NoSQL paradigm, for accessing to NoSQL stores. In this API, pieces of data are identified by *paths expressions* such as */users/78913131/birth-date*, which would request the birth date of the user with the identifier *78913131*. Using this API, applications can be developed independently of the target NoSQL system being used. Therefore, this system might be changed without having to update the application code. Based on this work, Atzeni et al. [32] developed a design methodology that provides a set of heuristics for transforming object-oriented models into NoAM models and then into code. *NoAM (NoSQL Abstract Model)* is a platform-independent logical model for NoSQL databases developed by the authors themselves.

The mapping strategy is based in the concept of *aggregate*, instead of being query-driven as in previous work. *Aggregates* are coherent units of behavior and data, which have identical life cycles and are often accessed and modified together. In our running example, *Purchase*, *PurchaseLine*, *Address* and *Bill* might constitute an aggregate. Aggregates are transformed into *NoAM* blocks, which aim to be an abstraction of NoSQL constructs, such as *column families* or *documents*. Then, these blocks are transformed into constructs of a concrete technology. Since NoSQL databases are very heterogeneous, in order to be abstract, NoAM lacks important features of certain paradigms, such as the *clustering key* of column stores. So, when mapping a *NoAM* block to a column family, we might have problems for identifying the partition and clustering keys, because, among other issues, we do not know which fields would be used to retrieve the data.

Herrero et al. [33] focus on deciding whether certain pieces of data should be stored in NoSQL databases in the specific context of BigData applications with a high-variability in data. To make this decision, a conceptual data model, augmented with information of entities evolution likelihood, is firstly constructed. Entities are then classified by the database designer as *nested*, *heterogeneous* or *homogeneous*. Based on this information, a set of guidelines are provided to decide whether an entity should be stored in a NoSQL store. Authors also provide a set of generic, technology-independent heuristics that might help database designers when mapping entities to NoSQL stores. These heuristics are driven by the probability that two pieces of data are accessed together, which is specified by means of an *Affinity Matrix*. As in the case of NoAM, these heuristics

Work	Paradigm	Technology	Conceptual	Queries	Logical	Automated
Li [23]	Column Family	HBase	—	—	Relational	No
DeLima & Santos [30]	Document	MongoDB	EER	XML based-[31]	In-house notation	No
Chebotko et al. [21]	Column Family	Cassandra	ER	ERQL	Chebotko	Yes
Mior et al. [22]	Column Family	Cassandra	ER	Entity Graph	—	Yes
Daniel et al. [24]	Graph-based	Neo4J	UML	Gremlin	GrapdDB	Yes
Atzeni et al. [32]	Any	Any	OO	—	NoAM	No
Herrero et al. [33]	Any	Any	OO	Affinity Matrix	—	No

Table 1: NoSQL database design methodologies.

are too general and need to be complemented with technology-specific decisions. Besides, its objective is to improve evolution by mixing relational and NoSQL technologies, but not to generate optimized databases. Consequently, generated databases might not be optimal from the point of view of performance, although the performance penalty might be affordable when evolution is a key requirement.

Daniel et al. [24] introduce a model-driven process for transforming UML conceptual data models, including several constraints expressed in OCL (*Object Constraint Language*) [36], into a logical model for graph-based NoSQL databases, and then to code. In addition to database code, some extra code for checking integrity of OCL constraints is also generated. The transformation is driven by the conceptual model structure, and queries are not taken into account.

In summary, as it can be observed in Table 1, most design processes are specific for a particular NoSQL technology [23, 30, 21, 24, 22]. Therefore, NoSQL database designers need to change between methodologies when working with different NoSQL paradigms, which implies an extra effort. For instance, when designing hybrid NoSQL stores, which use several NoSQL paradigms at the same time, we might need to specify the same conceptual model and the same set of queries in different notations. Regarding approaches that aim to be generic [32, 33], both of them provide just high-level heuristics and do not tackle design process automation. These heuristics do not consider platform-specific particularities and, consequently, they might lead to inadequate database designs. For instance, column stores often create a column family per application query, whereas document databases try to use a document for answering several queries at the same time. As a consequence, for the same conceptual model and set of queries, the number of column families in a column store is often greater than the number of document in their document-oriented counterpart. However, generic approaches, such as provided by Atzeni et al. [37], might create databases with exactly the same number of column families as documents.

To overcome these limitations, we have developed *Mortadelo*, a model-driven process for NoSQL database design that, starting from a technology-agnostic conceptual data model, allows for the automatic generation of database implementations for several NoSQL technologies. Moreover, the transformation process can be easily customized by database designers, in order to use those strategies that best fit with each context needs.

4. Solution Description

We start by describing the general components of the transformation process defined by Mortadelo. Then, successive sections describe these components with more detail.

4.1. General Overview

Figure 6 shows the database design process that is followed when using Mortadelo. As introduced before, Mortadelo follows a model-driven approach. This means that Mortadelo needs to operate with well-defined models that conform to a metamodel. Mortadelo starts with the creation of a conceptual data model, which is used as input of a chain of model transformations that generates a database implementation for a selected NoSQL technology.

To design a NoSQL database, as previously commented and pointed out by other authors [21, 30, 22], a conceptual data model that specifies just which entities comprise the system and how they relate is not enough. In NoSQL systems, knowing how these entities will be retrieved and updated at runtime is key. Therefore, traditional data modeling languages, such as UML or ER, have been complemented by other authors with languages for specifying data access patterns, such as ERQL [34]. This implies handling two separate but interrelated models, where each model conforms to a different metamodel.

From a technical point of view, processing two separate but interrelated metamodels adds some accidental complexity that could be easily avoided if both metamodels were integrated in just one. Therefore, to get rid of this extra accidental complexity, we created the *Generic Data Metamodel (GDM)*, a metamodel for NoSQL database modeling where the structural and data access patterns views are integrated into the same metamodel. The GDM is intentionally platform-independent, so it can be used seamlessly as input for different NoSQL paradigms.

Thus, Mortadelo starts with the definition of a GDM model (Figure 6, left). We give more details about the GDM components in Section 4.2. Then, the transformation process receives as input a GDM model that has been verified to assess that it contains no mistakes. In step 1, a model-to-model (M2M) transformation translates this GDM model into a logical NoSQL specification by the application of a set of transformation rules. Due to the heterogeneity of NoSQL, Mortadelo defines a logical metamodel and an associated M2M transformation for each NoSQL paradigm. In the figure, two logical metamodels are shown: a column family data model, and a

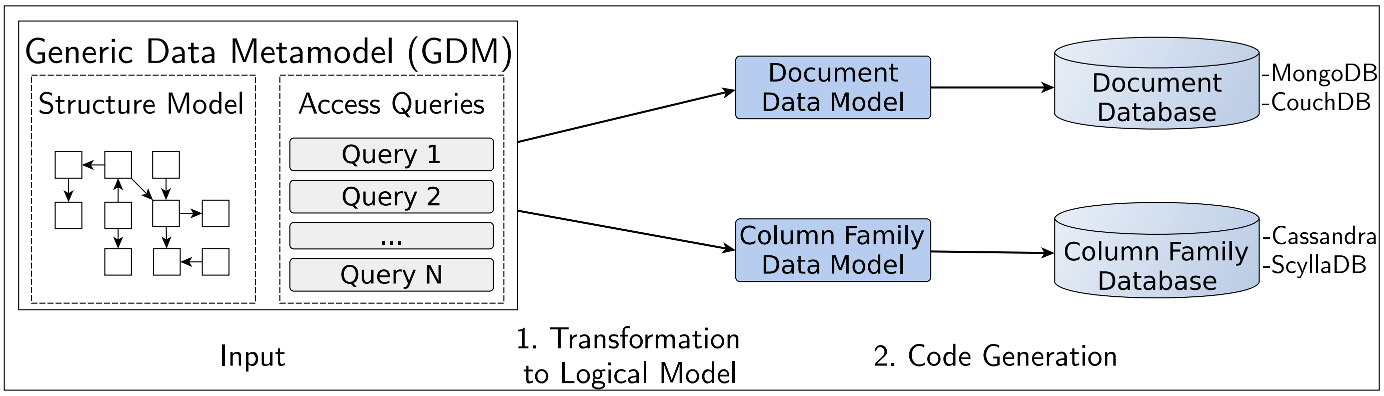


Figure 6: Transformation process of Mortadelo.

document data model. These metamodels are intermediate representations that contain information specific to the paradigm they represent. For instance, the column family data model allows defining the column families that should be instantiated in the final database. However, these specifications are still abstracted from any implementation details, which means that the logical model of a paradigm can be employed to represent technologies that belong to the same paradigm, e.g., Cassandra and ScyllaDB as column family-based technologies.

Lastly, the third step of the transformation process consists in a model-to-text (M2T) transformation. The obtained logical model from the M2M transformation of step 2 is used to automatically generate an implementation script for the targeted technology. Continuing with the previous example, a M2T generation from a column family data model could be performed to obtain a physical implementation for Cassandra, a database from this paradigm. An analogous example could be made for a document data model and a MongoDB implementation.

Next sections detail the GDM metamodel, and describe concrete examples of the transformation process for column family and document-based stores.

4.2. Generic Data Metamodel (GDM)

As mentioned in the previous section, we use instances of the *Generic Data Metamodel (GDM)* as input for Mortadelo. Figure 7 depicts this metamodel. This metamodel contains both the *Structure Model* and the *Access Queries* elements. Additionally, extra elements are included to allow *annotating* some elements of the GDM.

The *Structure Model* (Figure 7, left) is defined in a UML-like fashion. This is a well-known notation both in the modeling and database research areas, which presents adequate for the specification of the structure of domain data. Moreover, it is independent of any database technology, which is one of the requirements of the presented process. The data structure is defined by the specification of *entities*. These entities contain *features* of two kinds: (i) primitive *attributes*, which store values of a certain type; and (ii) *references* to other related entities. The references of an entity can have variable cardinality, e.g., 1, 2, 4 or unlimited.

The *Access Queries* (Figure 7, right) represent the requests that are going to be performed over the database. These queries are defined in the GDM over entities from the structure model. Queries are defined using a syntax structure inspired in SQL. However, oppositely to SQL, queries in our languages are specified against the entities of the GDM structural elements, and we can navigate through these entities by traversing their references.

A *Query* is executed over a main entity, captured by a *From* element. Any reference from that entity can be included in the query through an *Inclusion* element. Inclusions work in the same way as a conventional join of a relational SQL query. In addition, entities referenced by those that have been included previously can also be incorporated, i.e., inclusions can be recursively added as long as there are references available. The set of projection attributes that are retrieved by the query is specified as a list of *AttributeSelection* elements. This list can contain attributes coming from the *From* or the *Inclusion* entities. The *condition* of a query is captured with a *BooleanExpression*, which allows to declare any desired restrictions. The notation for boolean expressions is not shown in this article for the sake of simplicity and brevity, as this syntax is probably known by the reader. Finally, ordering can be specified through a set of *AttributeSelections*, again coming from the entities selected by the *From* and *Inclusion* elements.

Some elements from the GDM metamodel inherit from the *AnnotatableElement* class, which means that these elements can be *annotated*. Annotations are text indications that can be used to provide extra information to Mortadelo that may be useful when performing the transformation of a GDM instance into a NoSQL logical data model. For instance, *Entities* of the GDM can include the *@highlyUpdated* annotation. This annotation tells Mortadelo that the annotated entity receives a lot of transactional operations, e.g., inserts or updates. This detail might be of importance for some of the NoSQL paradigms, so it can be taken into account by the transformation rules when generating a NoSQL design. Apart from entities, *Queries* and *Features* can also be annotated.

As it can be seen, GDM specifications do not contain NoSQL details, which allows employing them as input for any transformation to a concrete technology.

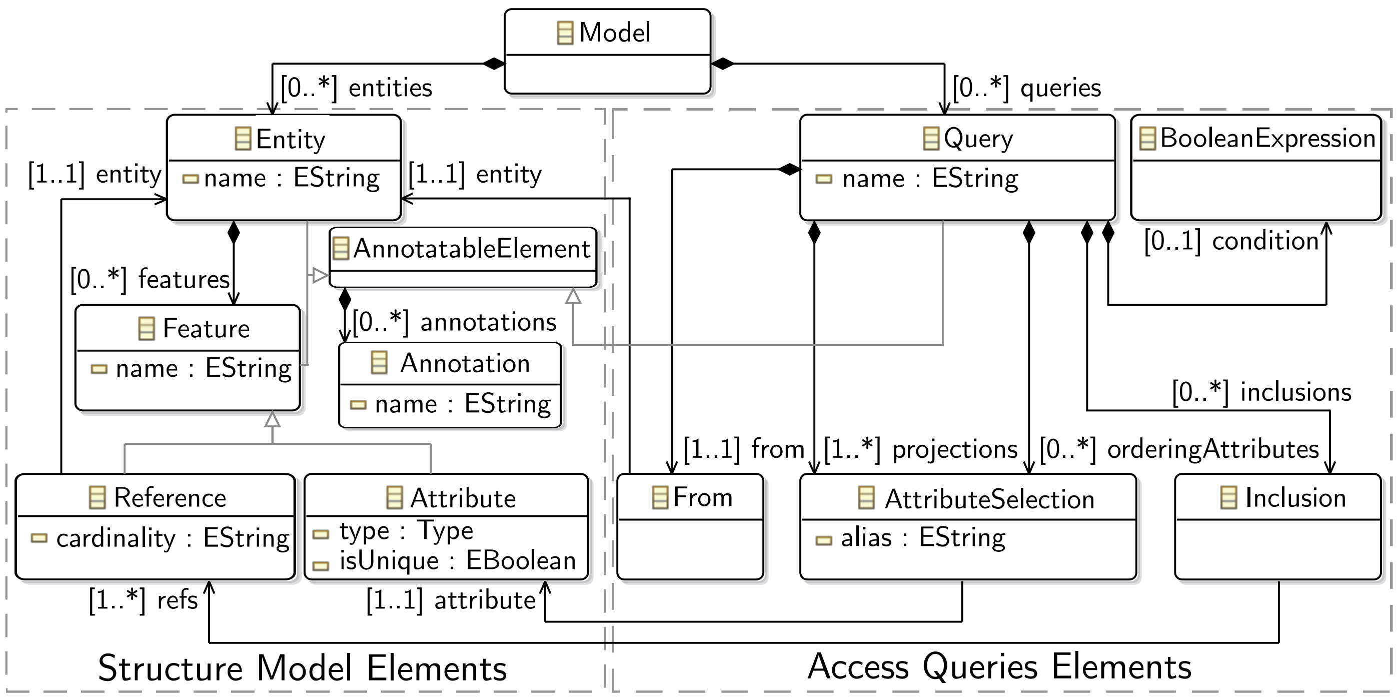


Figure 7: Main components of the *Generic Data Metamodel (GDM)*: Structure Model and Access Queries.

```
// Entities
entity Product {
  id productId
  text name
  text description
  number price
  ref Category[*]
        categories
  ref Provider[1]
        provider
}

entity Category {
  id categoryId
  text name
  text description
}

entity Provider {
  id providerId
  text name
}
```

```
// Queries
query Q1_productsById:
  select prod.productId,
         prod.name, prod.price,
         prod.description
  from Product as prod
  where prod.productId = "?"

query Q2_productsAndCategoryByName:
  select prod.name, prod.price,
         prod.description,
         cat.name
  from Product as prod
  including prod.categories as cat
  where prod.name = "?"

query Q3_productsByCategory:
  select prod.name, prod.price,
         prod.description,
         cat.name
  from Product as prod
  including prod.categories as cat
  where cat.name = "?"
  order by prod.price
```

Figure 8: GDM textual notation that defines the *Purchase* and *Category* entities and the queries *Q1*-*Q3* of the running example of Section 2.1.

4.3. Running Example in the GDM

We now show how the running example presented in Section 2.1 can be expressed in the GDM language. We created a textual syntax to instantiate models conforming to the GDM. Figure 8 shows how the *Product*, *Category* and *Provider* entities and queries *Q1*, *Q2* and *Q3* can be represented using this syntax. A complete specification showing how the rest of the entities and queries are defined can be consulted in an external repository[3]. The entities represented in this textual syntax follow the same structure as the one shown in the conceptual model of Figure 1.

In the example, entities are specified through the *entity* keyword, and their attributes and references are expressed between braces. For instance, the *Product* entity defines *productId*, *name*, *description* and *price* as attributes. Reference specifications start with the *ref* keyword, followed by an entity type, a cardinality, and a reference name. As an example, the statement *ref Category[*] categories* in the *Product* entity defines a reference called *categories*, with type *Category*, and with an unbounded (*) cardinality, i.e., a product can have an unlimited number of categories.

Queries in the GDM textual notation start by the *query* keyword followed by a name and an SQL-like syntax that specifies the objective of the query. The selected *projection attributes* appear after a *select* keyword. Then, the main entity is specified following the *from* keyword, and any extra referenced entities are indicated with an *including* clause. For all three depicted queries, the main entity is *Product*, and in *Q2* and *Q3* the *Categories* entity is included. The *where* clause gathers the conditions of the projection attributes of the query. *Q1* and *Q2* have an equality condition with the *productId* and *name* of a *Product*, respectively, while *Q3* uses the *name* of a *Category*. Lastly, an optional *order by* clause captures a sorting criteria for the results of the query, such as *order by prod.price* in *Q3*.

In this section, we have seen an example of how input databases can be textually specified by the instantiation of the structure data model and the access queries of the Generic Data Metamodel. Next sections show the logical models for column family and document-oriented data stores, and how Mortadelo can be used to generate a physical implementation of a Cassandra or MongoDB database from a GDM instance.

[3]https://github.com/alfonsodelavega/mortadelo/blob/master/es.unican.istr.mortadelo.gdm.examples/eCommerce.gdm

4.4. Column Family Stores Metamodel

Figure 9 shows the logical metamodel we have created for representing column family databases. This metamodel is inspired by the work of Chebotko et al. [21], who proposed a notation for column family stores known as *Chebotko diagrams*. As it can be seen, the *DataModel* metaclass is the entry point to this metamodel. A *DataModel* can be considered as a database schema that contains several *Table* specifications.

Each table contains a set of *columns*. Each column can have an associated *Type*, which can be a *SimpleType*, i.e., a built-in primitive type, such as a *Integer*; a *Tuple* of values; a *Collection* of values; or a *UserDefinedType*. *Tuples* are made up of elements of different types, whereas *Collections* store zero or more elements of the same type. *User-defined types (UDTs)* are a mechanism to give names to tuples and their fields, making easier the management of these tuples. For example, to represent the remaining time to complete a task, we might use a tuple *(int, int, int)*, which represents the remaining hours, minutes and seconds respectively. On the other hand, we might create a UDT, called *TimeDuration = (hours : int, minutes : int, seconds : int)*. The second option might be preferable when these durations are used in several places, or when we want to clarify the meaning of each tuple field. UDTs can also be composed, this is, we can use UDT types inside the fields of another UDT.

Columns can also belong to a column family, in the BigTable sense, and a table might contain several column families. In the case of Cassandra, each table contains just a column family, so the terms column family and table will be used indistinctly in the rest of this paper.

In addition to its columns, each table must define a *primary key*, which is an ordered group of columns. As commented in Section 2.2, the primary key can be decomposed into two separated subsets: the *partition key*, which is used to divide rows of a column family in different subsets or partitions, and the *clustering key*, which is used to sort column family partitions so that some ranges of rows inside a partition can be more easily retrieved. So, each column in a primary key would be either a *PartitionKey* or a *ClusteringKey*. As stated in Section 2.2, a table requires at least one partition key column, while clustering keys are not mandatory.

4.5. Document Databases Metamodel

Figure 10 shows the logical metamodel for document databases. As introduced in Section 2.3, a document data model is composed of *Collections*, which have a *name* that identifies them. Each collection is used to store documents that, in most cases, share the same structure. This structure is captured in a *DocumentType* element. A *DocumentType* defines this structure through a set of *Fields*, which can be *Primitive* elements, *Arrays* of elements, or even nested *DocumentTypes* embedded inside the *main* one. All collections have a main *DocumentType*, which in the metamodel (and in the rest of this paper) is denoted as *root*.

Document stores allow the creation of heterogeneous collections, i.e., collections where their documents conform to very different schemata. However, for most situations, the recommended practice is to split any heterogeneous collection into several homogeneous ones, where each collection conforms to a single schema. So, as collections of heterogeneous documents are actually helpful in very rare cases, and to avoid the extra complexity they add, we decided to limit collections to a single document type.

4.6. Transformations for Column Family Stores

This section describes a set of rules, inspired by the work of Chebotko et al. [21] and Mior et al. [22], for transforming a GDM model into a column family model by means of model-to-model transformations. These rules are driven by a main strategy: creating a column family to support each application query. This strategy is aligned with the guidelines and best practices provided by column family vendors, such as *Cassandra*, and they are also the base idea on which the transformations specified by Chebotko et al. [21] and Mior et al. [22] are grounded. Next subsection describes this strategy.

4.6.1. Query to Column Family Transformation

This main strategy is precisely defined in Algorithm 1 of AppendixA, and it works as follows: Given an access query *aq*, we create a new column family *cf*, with the same name as *aq*, for supporting such a query. The column family *cf* initially has as many columns as *aq* has projection attributes. For instance, let us suppose we want to transform the query *Q4* of our running example, depicted in Figure 11. This query retrieves clients' purchases happening in the last months of a given year, so we refer to this query as *clientPurchasesNearChristmas* in the remaining of the article. In this case, a new column family with the same name would be created. This column family would have 11 columns, one per each projection attribute, from *client.clientId* to *address.postalCode*. Each newly created column would have the same type as its projection attribute in the structural model. For example, the *client.name* column would have the same type as the *name* attribute of the *Client* class.

Once the base structure for the column family is calculated, we need to establish a partition key and, if required, a clustering key. For this purpose, we first extract all attributes that are involved in the query condition. Then, we process each attribute *attr* included in this set of attributes. If there is an attribute *attr* included in a comparison by equality, its associated column is added to the partition key. If *attr* is present in a comparison by inequality (e.g. greater or less than operation), its column is added to the clustering key. For example, in the case of query *clientPurchasesNearChristmas*, *client.clientId* and *purchases.year* would be included in the partition key, whereas *purchases.month* would be part of the clustering key. Attributes involved in inequalities are added to the clustering key in the same order they appear in the query.

Finally, the query sorting criteria is processed to take advantage of the clustering key as a sorting structure. To do this, we check first that the *ordering clause* can be computed by the target database engine. To be supported, the ordering criteria must be compatible with the ordering imposed by the elements

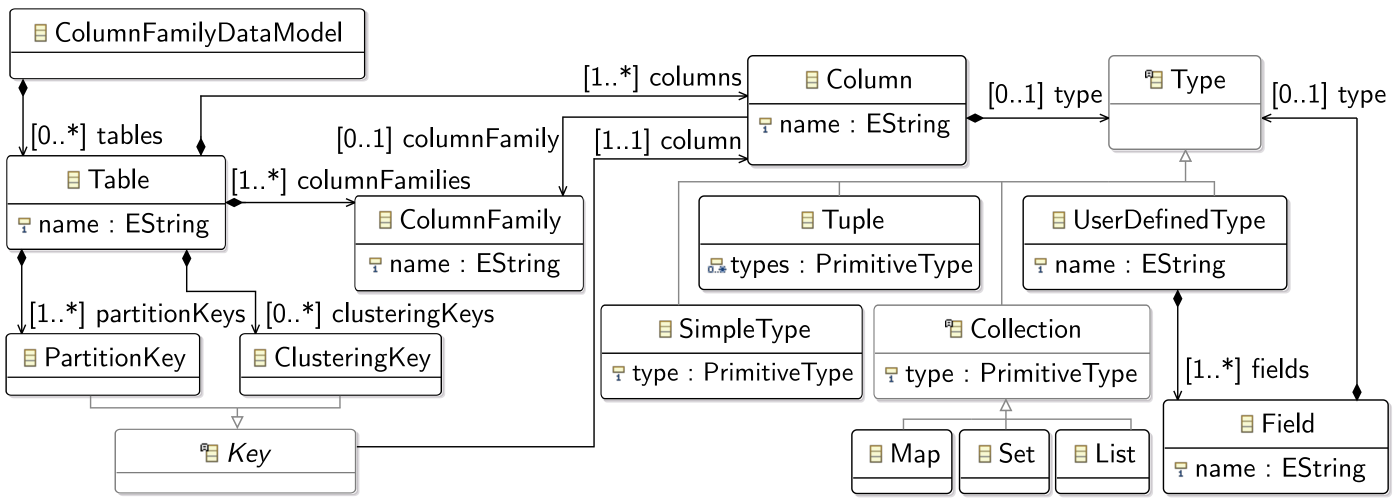


Figure 9: Metamodel for the logical modeling of column family databases.

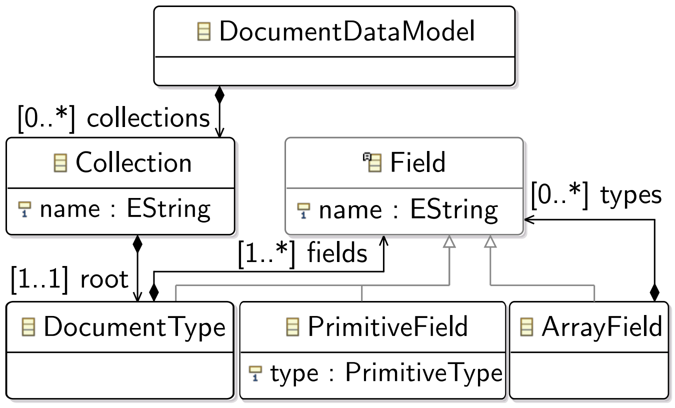


Figure 10: Metamodel for the logical modeling of document-based stores.

```
query Q4_clientPurchasesNearChristmas:
  select client.clientId, client.name,
         client.nationality,
         purchases.purchaseId, purchases.year,
         purchases.month, purchases.day,
         lines.quantity, lines.unitPrice,
         product.name, address.postalCode
  from Client as client
  including client.purchases as purchases,
            client.purchases.lines as lines,
            client.purchases.lines.product as product,
            client.purchases.shipAddress as address
  where client.clientId = "?1" and purchases.year = "?2"
    and purchases.month >= 10
  order by purchases.month, product.price
```

Figure 11: Access query retrieving purchases of a client before Christmas.

already included in the clustering key, i.e., those resulting from processing the inequalities. For instance, in the case of query *clientPurchasesNearChristmas*, if the sorting criteria were just *product.price*, it would be incompatible with the clustering key, since the clustering key is already sorting by month. In this case, attributes in the sorting criteria are ignored by the transformation process, and this sorting should be performed at the application level.

For a sorting criteria to be compatible with an existing clustering key, the columns of the clustering key must appear as the first elements of the ordering criteria, and in the same order as in the clustering key. If that is the case, those extra attributes of the sorting criteria that are not present in the clustering key are included, in the same order as they appear in the ordering clause. In the query *clientPurchasesNearChristmas*, its sorting attributes (*purchases.month* and *product.price*) are compatible with the clustering key generated by the conditions, so we would add *price* to the end of the clustering key.

Finally, we must ensure *uniqueness* of each column family row, which might not be a trivial process. When a query involves a single entity from the GDM, *row uniqueness* is ensured if the entity contains a unique attribute and this attribute is part of the primary key, which is usually the case (e.g. *Purchases* have a *purchaseId*). In a query involving several entities, one entity plays the role of *main* entity and the other entities are *secondary* ones. The main entity is the one specified in the *from* clause, while the secondary entities are provided in the *including* section. *Row uniqueness* can then be determined by the upper bounds of the references' cardinalities between the main entity and the secondary entities. If this upper bound is 1, this means that the secondary entity can be also identified by the identifier of the main entity. For instance, this happens between *Purchase* and the *shipAddress* for that purchase, as this particular address can be identified by the *purchaseId*. Therefore, a column family containing just data from *Purchase* and *Address* would have *row uniqueness* ensured just by including *purchaseId* as primary key.

On the other hand, when the upper bound of a relationship is greater than one, the identifier of the main entity is not enough to provide *row uniqueness*. This would be the case of a query with *Client* as main entity, and involving *Purchase* as a secondary entity. In this case, *client.id* would not be unique for each combination of *client* and *purchase* instances. This problem can be solved adding *purchase.purchaseId* to the primary key of the column family involving *Client* and *Purchase*.

With these premises in mind, we devised an algorithm for ensuring *row uniqueness* (See Algorithm 2 of AppendixA). This algorithm works as follows: first of all, all paths from a main entity to its secondary entities are calculated. Then, we remove those paths that are a subpath of another one. Next, if we can ensure uniqueness at the end of each path, the problem would be solved. Therefore, if we can calculate an identifier for each entity at the end of each path, the problem would be solved. We would only need to check whether this identifier is already contained in the primary key, and if it is not, we would add it at the end of the clustering key.

To calculate such an identifier, there are two cases. The first and most direct one is when the entity itself has an attribute marked as *isUnique*. We would simply select that attribute as identifier. This might be the case of *Purchase* as secondary entity of *Client*. The second case happens when the entity has no attribute marked as *isUnique*. In this case, we might borrow one from another entity. To find an entity from which we can borrow an identifier for a path end, we search for a preceding entity in that path that fulfills the following two conditions: (1) it has an attribute *attr* marked as *isUnique*; and, (2) just one instance of this entity can be related to the path end. If an entity like this is found, we use its attribute *attr* as identifier for the path end. Otherwise, uniqueness would be ensured by using an artificial identifier, i.e., a *dummy* column that is unique for each row of the column family.

In the case of query *clientPurchasesNearChristmas*, two paths are identified: (1) *client.purchases.shipAddress*, and, (2) *client.purchases.lines.product*. In the first path, the ending entity is *Address*, and it does not have any attribute marked as *isUnique*. However, it can take *purchaseId* from *Purchase*. The attribute *purchaseId* was not included in the primary key of the column family being generated, so it is added at the end of the clustering key. For the second path, the entity at the end is *Product*. This entity has the *productId* attribute marked as *isUnique*. This attribute is also not included in the primary key of the generated column family and, consequently, it is added at the end of the clustering key. In the case of these attributes to ensure uniqueness, the order in which they are added to the clustering key is not relevant, since they are used neither for retrieving ranges of values nor for sorting purposes; they are just included to ensure uniqueness. So, it does not matter which path end is processed first.

The application of this *query-to-column-family* transformation rule is enough to generate a set of column families that support a set of queries. Nevertheless, applying just this rule might lead to non-optimized designs because of redundant queries, or to an excessive degree of denormalization. Next subsections describe two optimizations that help alleviate this problem.

4.6.2. *Query Merging*

Let us suppose we have two queries with the same equality conditions in the *where* clause, no inequalities, no sorting criteria, but with different projection attributes. Using just the basic transformation rule described in the previous section, two individual column families would be generated for this case. Nevertheless, since the resulting column families share the same partition key, have no clustering key, and they differ just in the number of columns, they might be merged into a single column family, with the same partition key and the combination of columns from their projections. For this merging to be possible, the set of columns from the primary key that grant row uniqueness, according to the method described in the previous section, must also be the same for both individual column families. If this condition also holds, queries are compatible and can be merged, avoiding unnecessary data duplication of those columns that are repeated in both query projections, and thus reducing database size and slightly improving insertions, deletions and updates.

Therefore, when two queries satisfying these conditions are found in the set of access queries, both queries are merged to create a single query that synthesizes both and that, once processed, generates a single column family. To merge two queries q_1 and q_2, we create a new compacted query q' that has, as projection attributes the union of the attributes of q_1 and q_2; as inclusions, the union of the inclusions of q_1 and q_2; and, as *where* clause, the *where* of either q_1 or q_2, as both clauses must be equal.

Moreover, this can be generalized to also cover the following additional situations:

1. Both queries have identical equality conditions, inequality conditions, and there is no sorting criteria in both queries. In this case, the merging process works as previously described.
2. Conditions of case 1 holds and, in addition, only one query has a sorting criteria. In this case, the merging process works as previously described, and the only existing sorting criteria is added to the merged query.
3. Conditions of case 1 holds and, in addition, the sorting criteria of one query is a subset of the sorting criteria of the other query. In this case, the merging process works as previously described, and the largest sorting criteria is added to the merged query.

Figure 12 shows an example of the application of the *Query Merging* process. As can be observed, queries *Q5* and *Q6* show purchases information in a year along with *billing* or *postal code* information, respectively. These queries share the same equality condition (i.e. comparison by year), and the sorting criteria of *Q5* is a subset of the one of *Q6*: {*pur.month*}, and {*pur.month,addr.postalCode*}. Therefore, they can be merged in a single query named *Q5_and_Q6*, which retrieves purchases with billing and address data in a year, ordered by the largest sorting criteria, i.e., the one of the merged *Q6*. The query merging process described in this section is detailed in Algorithm 3 of AppendixA.

4.6.3. *Partition Key Softening*

We introduced an annotation over entities (see Section 4.2), labeled as *@highlyUpdated (HU)*, to specify that one entity of the GDM will receive a high number of insertions and updates as compared to their reads. The rationale behind this annotation is to warn that denormalization or duplication of these entities should be controlled whenever possible, since replicating their

```
query Q5_purchasesWithBillsByYear:
  select pur.purchaseId, pur.year,
         pur.month, pur.totalAmount,
         bill.billId, bill.billingData
  from Purchase as pur
  including pur.bill as bill
  where pur.year = "?"
  order by pur.month

query Q6_purchasesWithPostalCodeByYear:
  select pur.purchaseId, pur.year, pur.month,
         pur.totalAmount, addr.postalCode
  from Purchase as pur
  including pur.shipAddress as addr
  where pur.year = "?"
  order by pur.month, addr.postalCode
```

merged into

```
query Q5_and_Q6:
  select pur.purchaseId, pur.year, pur.month,
         pur.totalAmount, bill.billId,
         bill.billingData, addr.postalCode
  from Purchase as pur
  including pur.bill as bill,
            pur.shipAddress as addr
  where pur.year = "?"
  order by pur.month, addr.postalCode
```

Figure 12: Query Merging example.

data might have a non-negligible impact on insertions and updates. In Section 2.1, we mentioned that *Purchases* is an entity that is expected to receive a high number of writing operations, so it makes sense to annotate this entity as HU.

Now, let us suppose we want to perform the query of Figure 13, *Q7*, where information about the purchases of a client in a concrete month are retrieved. This query has three equalities: *client.clientId*, *purchases.year* and *purchases.month*. The query *clientPurchasesNearChristmas*, which we presented in Section 4.6.1, contains the projection attributes that are gathered in *Q7*, and almost the same operations in the condition clause: the first two equalities are the same (*clientId* and *year*), but the *purchases.month* participates in an inequality instead of an equality, to obtain the purchases of the months before Christmas.

These queries would not be merged using the algorithm previously described, as their *where* clause conditions do not match. However, if we created a column family having as partition key the *client.clientId* and *purchases.year* attributes, and *purchases.month* as clustering key, this column family would support both *clientPurchasesNearChristmas* and *Q7* queries. From the read operations' perspective, using just one column family would be slightly worse, as *Q7* would include an equality comparison against a clustering key column, which is slower than comparing with a partition key one. However, from a insert, update and delete perspective, having one column family is better, since these operations would be performed against one column family instead of two.

```
query Q7_clientPurchasesInAMonth:
  select client.clientId, client.name,
         purchases.purchaseId,
         purchases.year, product.name,
         lines.quantity, lines.unitPrice
  from Client as client
  including client.purchases as purchases,
            purchases.lines as lines,
            lines.product as product
  where client.clientId = "?1"
    and purchases.year = "?2"
    and purchases.month = "?3"
```

Figure 13: Partition key softening example. If the marked *purchases.month* = *?3* equality is changed with an inequality (e.g. ≥ or <), then this query and the one of Figure 11 could be answered with the same column family.

Therefore, in our transformation process, we look for all query pairs (q_1 and q_2) where the previous condition holds, i.e., queries are compatible, and *softening* some equalities in q_1 allows answering both with the same column family. In these cases, if both queries involve an entity annotated as *Highly Updated (HU)*, the queries are merged as explained in the previous subsection, replacing the equality condition of q_1 with an inequality. Thus, we ensure that only one column family would be generated, avoiding data duplication of the entity with a high numbers of insertions, updates or deletions.

4.6.4. General Transformation Algorithm

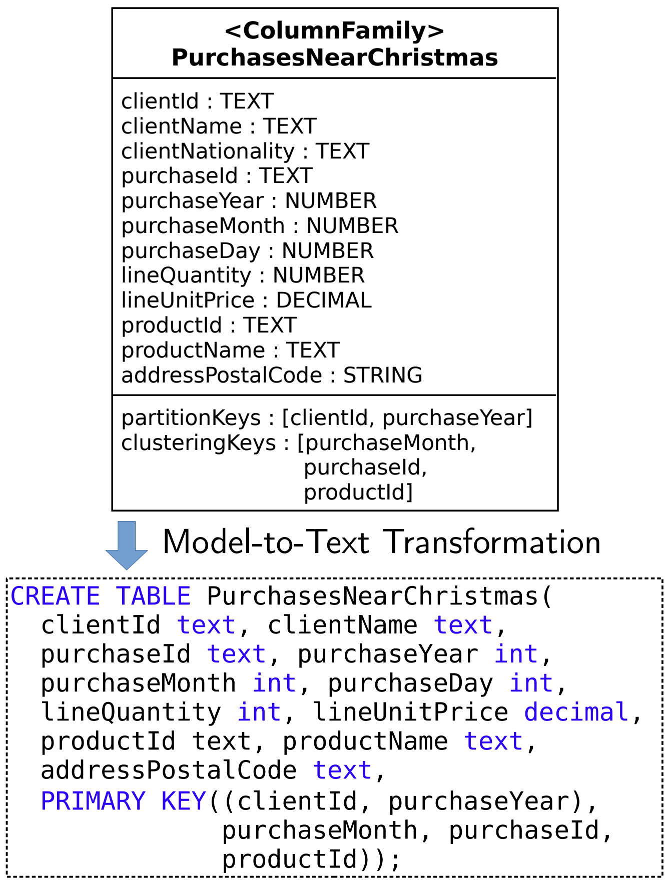
Using these three rules, the process for transforming a GDM into a logical column store model is as follows:

1. Queries are merged, producing a more compact set of queries that is free of redundant ones.
2. Partition key constraints are softened, to merge queries in the presence of highly updated entities.
3. A column family per query is generated. This query-to-column-family transformation process calculates partition keys, clusterings keys, tries to perform sorting at the database level and ensures row uniqueness.

As an example of the generated column families, Figure 14, top shows the column family that would be generated to allow the *clientPurchasesNearChristmas* query.

4.6.5. Logical model to physical model transformation

Once the logical data model for column family databases has been generated, it can be transformed into a concrete database implementation by means of a model-to-text transformation. Cassandra offers an SQL-like language for database management, called *Cassandra Query Language (CQL)*. So, this model-to-text transformation must convert a logical model into CQL code, which is pretty straightforward: each column family definition is transformed into its corresponding CQL counterpart. Figure 14, bottom shows the CQL counterpart that would generate the column family depicted in Figure 14, top.



```
CREATE TABLE PurchasesNearChristmas(
  clientId text, clientName text,
  purchaseId text, purchaseYear int,
  purchaseMonth int, purchaseDay int,
  lineQuantity int, lineUnitPrice decimal,
  productId text, productName text,
  addressPostalCode text,
  PRIMARY KEY((clientId, purchaseYear),
              purchaseMonth, purchaseId,
              productId));
```

Figure 14: Top: column family generated to answer the *clientPurchasesNearChristmas* query of Figure 11. Bottom: resulting CQL code to instantiate the column family in a Cassandra Database.

4.7. *Transformations for Document Databases*

This section describes the process for automatically transforming a GDM conceptual data model into a document database. This transformation process is described in detail by Algorithms 4 and 5 of AppendixA, which are inspired by the work of de Lima and dos Santos Mello [30]. Nevertheless, unlike [30], we allow denormalizations when generating a document database.

4.7.1. *GDM to Document Logical Metamodel*

As it was mentioned in Section 2.3, document databases aim to improve performance by applying data nesting. When a query requests a hierarchy of nested objects, this hierarchy can be returned accessing to a single well-defined location. This way, having to retrieve information from different places of the database and having to combine this collected information to produce the hierarchy of objects is avoided. Therefore, when designing a document-oriented database, we need to make two main decisions: (1) identify the number of hierarchies we need to store in the database; and, (2) determine how many nested elements each hierarchy should contain.

In our case, to calculate how many hierarchies would be stored in the generated database, we identify what are the entry points to the structural model of a GDM instance. These entry points are determined by the *from* clauses of the GDM access queries, i.e., the *main* entities. So, the first step of our transformation process is to select all entities that appear at least once in these *from* clauses. Then, for each collected entity, we create a *Collection* in the document logical model (see Figure 10). Each one of these collections has a *root* document type to host its corresponding main entity.

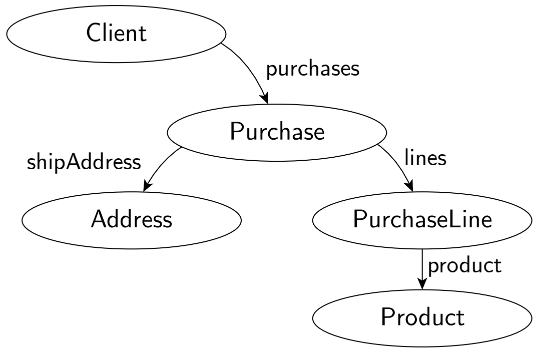


Figure 15: *Client* access tree calculated using only the query of Figure 11.

Secondly, we generate the contents of the root documents for each collection. For that purpose, we need to know which information from other entities might be accessed from each main entity. To find this information, we process all queries where each main entity appears in the *from* clause. For these queries, we collect all references that appear in their *including* clauses. With these references, we build an *access tree*. An *access tree* specifies which entities are accessed from a certain entity and through which references are accessed. For example, Figure 15 shows an access tree for the inclusions of query *clientPurchasesNearChristmas*, which was illustrated in Figure 11. For example, this access tree specifies, among other details, that the *PurchaseLine* entity is accessed from *Client* through the *purchases.lines* reference.

Using the main entity's *access tree*, we generate the contents of its associated root document type as follows. First of all, we add all attributes of the root entity to the document, independently of whether these attributes are used by any query or not. Adding simple attributes to a document does not increase document size noticeably, and this addition might make database evolution easier, as they can be used to support new queries or modifications of the existing ones. As an example, for the *Client* entity, we would add to the root document schema fields for the *clientId*, *name* and *nationality* attributes.

As a second step, we process the references of the root entity. For each reference, a new field is created in the document whose contents are being generated. To determine the type for that new field, there are two issues that need to be considered separately: (1) the reference's cardinality; and (2) the reference's target entity. If the target entity is contained inside the access tree, the data of that entity might be required when processing a query over the root document. Therefore, the data of the target entity must be included in the document being generated as a sub-document. This means that the new field will be a document, or a collection of documents, depending on the reference cardinality.

On the other hand, if the target entity is not included in the access tree, we might even discard this reference. Nevertheless, as before, we include it to make evolution easier. This reference would contain the identifier value of the referenced entity, when the cardinality of this reference is one, and an array of identifier values when this cardinality is greater than one.

When a reference is transformed as a subdocument, we execute the content generation process recursively to generate its contents. In the recursive call, the subdocument to be populated is used as root document, and for the access tree we use the subtree with root in the entity accessed through the reference.

In our previous example, the reference *purchases* of the *Client* entity is in the access tree, so it is nested inside the root document of the *Client* collection. Since this reference is unbounded, *purchases* is transformed as an array of subdocuments. To generate this subdocument, the content generation process is invoked again, using the subdocument for the *Purchase* as main document, and the subtree with root in the *Purchase* node as access tree. For this subdocument, the attributes *purchaseId*, *month* and *year* would be added as primitive fields. Then, the *bill*, *shipAddress*, and *lines* would be processed. The *bill* reference is not included in the access tree, so we generate a primitive field with *id* as type to hold a bill identifier value[4]. The *shipAddress* and *lines* references are in the access tree, so a subdocument and an array of subdocuments would be created for them, respectively. This subdocuments would be populated using the same process, until reaching the access tree leaves.

The described strategy embeds everything that is required in order to improve query performance. Nevertheless, this strategy might lead to a high level of denormalizations, which might not be adequate in some cases. For instance, since *Purchase* is a root entity, its data would be replicated in the *Purchase* collection and in the *Client* collection.

If we wanted to reduce the degree of denormalization, we might execute the previous transformation process with two extra optimizations. The first optimization aims to decrease denormalization in general. To do it, once the access trees are computed, we remove from these trees all nodes that corresponds to entities that are roots of other access trees. So, when these trees are generated, these root entities will not be embedded, and identifier values will be used for establishing the references instead.

The second optimization aims to reduce denormalization for those entities where data replication is discouraged. These entities should be annotated as *Highly Updated (HU)*. If this optimization is active, the transformation process checks whether an HU entity appears in more than one access tree. If so, it means their data would be duplicated. To avoid it, it should only appear in one access tree. If there is a tree in which the *Highly Updated* entity acts as root, then this entity would remain in that tree, and it is removed from all other trees. If there is not such a tree, it is computed in which tree the *Highly Updated* is accessed more frequently. The entity will remain in such a tree and it will be removed from all other trees.

[4]Bill is not included in the access tree because we computed it for just one query (see Figure 11), as we did not want to overwhelm the reader with a large and complex access tree. Nevertheless, if all queries were considered, this entity would end inside the access tree, and thus also in the resulting collection.

4.7.2. Document Logical Metamodel to MongoDB

As in the column family stores case, once the logical model for a document database has been obtained, it can be transformed into code for generating a concrete implementation in a specific document store technology. This process, as before, is performed by means of model-to-text transformations that basically map the concepts of the logical metamodel to constructs of the language of the target document database. Currently, we have implemented this code generation process for MongoDB, and new document stores will be targeted in the future.

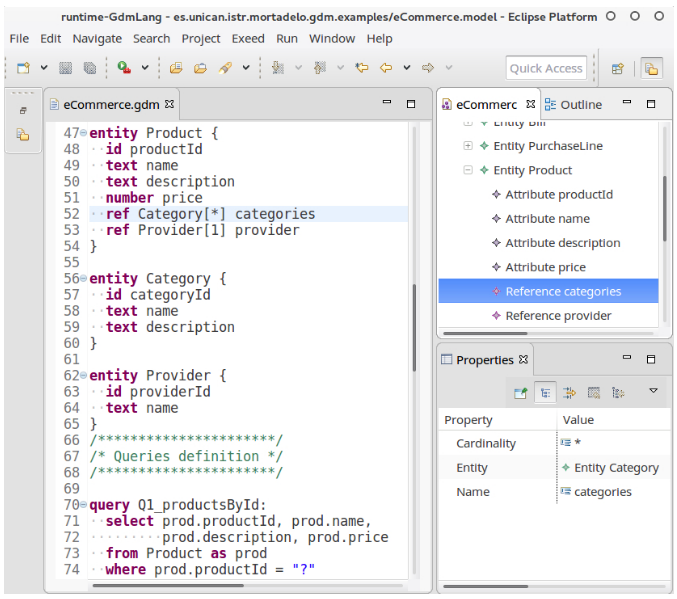
4.8. Implementation

We have implemented a prototype of Mortadelo to assess the transformation process presented in the previous section. This implementation is available under a free licence in an external repository[5]. Next paragraphs summarize the main components of this repository.

The metamodels presented in Section 4 can be found in the corresponding projects of the repository in *Ecore* [38] format. Precisely, the GDM, column family, and document metamodels are included. In addition, the projects also contain the model-to-model and model-to-text specifications that conform the transformation process. Conventionally, M2M transformations are specified through model-to-model languages such as ETL or ATL. These languages are useful when each input element of a certain type is transformed into one or more output elements. However, as we have seen in the transformation sections, data structure and access queries have to be treated all at once when generating column family or document logical models, instead of in a one-by-one basis. For this reason, we decided to employ an imperative language for the M2M transformation process. We selected *Xtend*[6], which is a Java-based language that offers advanced model manipulation capabilities. In the case of M2T transformations, they are specified in EGL (Epsilon Generation Language) [39].

For the GDM metamodel, a textual Domain-Specific Language (DSL) [40] for the manipulation of GDM instances is also provided. This DSL has been implemented with Xtext [41], which provides a full-featured and easily configurable editor. Figure 16 shows a screenshot, where the online shop case study is manipulated through the DSL editor. The left window shows the syntax of the DSL, which allows to define and validate entities and queries over these entities. On the top right window, the corresponding GDM instance model of the processed GDM file is shown. This instance would be the input of Mortadelo's transformation process. Below, in the Properties view, individual details of concrete elements from the model can be consulted, such as the *AttributeSelection* object selected in the figure.

[5]https://github.com/alfonsodelavega/mortadelo
[6]https://www.eclipse.org/xtend/


runtime-GdmLang - es.unican.istr.mortadelo.gdm.examples/eCommerce.model - Eclipse Platform
File Edit Navigate Search Project Exeed Run Window Help
Quick Access
eCommerce.gdm
eCommerc
Outline
Properties

```
entity Product {
  id productId
  text name
  text description
  number price
  ref Category[*] categories
  ref Provider[1] provider
}

entity Category {
  id categoryId
  text name
  text description
}

entity Provider {
  id providerId
  text name
}
/*********************/
/* Queries definition */
/*********************/

query Q1_productsById:
  select prod.productId, prod.name,
         prod.description, prod.price
  from Product as prod
  where prod.productId = "?"
```

Entity Bill
Entity PurchaseLine
Entity Product
Attribute productId
Attribute name
Attribute description
Attribute price
Reference categories
Reference provider

Property	Value
Cardinality	*
Entity	Entity Category
Name	categories

Figure 16: Editor of the provided GDM textual DSL.

Finally, an examples project is included, which contains GDM specifications and resulting NoSQL schemas, e.g., for the running example of this paper.

5. Evaluation and Discussion

As it was stated in the introduction, the main goal of Mortadelo was to automatically generate databases for different NoSQL paradigms from the same high-level data model. The previous section has shown how this goal is satisfied for the running example. This section analyzes how Mortadelo works in other case studies, providing evidences about its applicability. To evaluate whether our work can be used in different settings, we analyzed it from the following four perspectives: (EI-1) expressiveness of the GDM metamodel; (EI-2) performance of the generated databases; (EI-3) required computational resources for executing the model transformations; and (EI-4) adaptability of our approach to changes in the NoSQL paradigms. Moreover, for each one of these items, comparisons with related work will be provided as required.

5.1. EI-1: GDM expressiveness

The very first issue we analyzed to assess general applicability of our approach was GDM expressivity. This is, we studied whether the conceptual modeling language we created is expressive enough to model any database. Since GDM mimics the *Entity-Relationship* notation, GDM's expressiveness should be as good as ER's. ER has been widely used for decades without big issues, so GDM should have a good expressiveness. To verify this hypothesis, we modeled a set of external case studies in GDM. These case studies were extracted from the NoSQL literature, and are often used as testbeds in NoSQL research. More specifically, we used four case studies: (1) a *hotel management* system, extracted from Carpenter and Hewitt [25]; (2)

Case Study	Hotel	EAC	Store	Venues	Windows
#Classes	9	4	11	4	8
#Relations	9	3	8	6	9
#Att/Class	3.2	3.7	2.1	5.25	4.75
#Queries	9	5	6	9	7

Table 2: NoSQL case studies statistics.

the *EasyAntiCheat (EAC)* system[7], used as case study by Mior et al. [22]; (3) a *store* database, taken from de Lima and dos Santos Mello [30]; and (4) a *digital venues* example, used by Chebotko et al. [21]. Additionally, we included a fifth case study from a cooperation with a local software company. This case study involves industrial data related to a cutting machine for bars that were used in window manufacturing.

Table 2 shows some basic statistics about these case studies to provide a general overview of their size and structure. These case studies might seem small, but this is a typical case in NoSQL systems, where we want to store thousands of instances of just a few entities, and then perform a set of well-defined queries while maintaining a good performance.

Just a minor issue was detected when modeling these case studies in GDM. GMD does not directly support inheritance between *entities*, but the *Store* case study uses inheritance to model the relationships between some entities. We opted for not including inheritance in GDM initially for the following reasons. First of all, inheritance added a lot of picky details that made the description of the transformation rules far more complex. Secondly, some authors consider inheritance as a harmful mechanism and they recommend the use of other alternatives, such as composition [42]. Thirdly, inheritance can be represented by means of other mechanisms, such as, for instance, the single-table pattern [43], among others. Therefore, to keep this work affordable, we opted for skipping inheritance initially, and representing it by means of other mechanisms. As part of our future work, we will explore how to deal with inheritance in NoSQL systems.

In addition, the experienced reader might reasonably argue that the @*highlyUpdated* annotation is not sufficient and that, to specify workloads more precisely, we should quantify the rate of reads and updates over each entity. This way, considering the cost of each operation, our model transformations should be able to find the solution that provides a better performance for that specific workload, such as Mior et al. [22] do. There is a rationale behind not doing it, as we explain in the following.

The measurement and characterization of these workloads in real settings might get really complex. Modern applications have several cache levels, which make the cost of read operations variable. Therefore, something more complex than a single value may be required to represent operation cost. Moreover, operation rates might not be fixed, as they could vary depending on each scenario. For example, applications that sell tickets for sport events might have a peak of updates over a

[7] https://www.easyanticheat.net/

purchase entity just after starting the sales period for very demanded events, like the Super Bowl final, but more stable update operations during the rest of the year. Also, applications with variable operations rates might be interested in optimizing a few frequent scenarios. For instance, the Super Bowl final ticket selling event might be considered a critical scenario, and so the application should offer a very good performance, despite happening only once a year. Finding effective mechanisms to specify all these issues appropriately is a big challenge that can hardly be addressed in the context of a single work. Consequently, approaches that aim to find optimal solutions, like the one of Mior et al. [22], must make some assumptions to simplify things and make the problem affordable. For instance, Mior et al. [22] neglect the effect of caches, and only work with applications whose operation rate is fixed.

Therefore, any solution we had used to quantify workloads would have been unsatisfactory in real settings. Thus, since the focus of our work was not how to specify workloads precisely, we opted for using a simple solution that: (1) it did not obscure the focus of our work, which was the generation of databases for heterogeneous NoSQL paradigms, and (2) it was also satisfactory in a wide range of situations. In our case, it is the database designer who must decide, after analyzing different design drivers, whether issues such as denormalization might be problematic. If it is considered so, she should act accordingly, e.g., using the @*highlyUpdated* annotation.

Finally, some readers might miss some information about database replication and sharding in these database designs. This information, although key in NoSQL, is often considered orthogonal to the database schema design, and therefore, specified separately. For instance, Kolovos et al. [44] use different DSLs to specify schema design and deployment issues. Therefore, as part of out future work, we plan to study how to complement Mortadelo with additional languages that cover sharding and replication.

5.2. EI-2: Quality of the Generated Databases

This section evaluates whether the performance of the databases generated by Mortadelo is good enough to use them in real settings. To fulfill this task, we carried out two different actions: (1) we compared performance of the databases generated by Mortadelo and with databases generated using other state-of-the art approaches; and (2) we checked with industrial practitioners whether the databases generated by Mortadelo have enough quality to be deployed in real settings. Both actions are described in the following. But before doing it, it is worth to remember that improving performance of state-of-the-art approaches was not our main goal. The main contribution of Mortadelo is to generate databases for different NoSQL paradigms from the same high-level data model. Therefore, we are mainly interested in checking that performance has not been sacrificed to provide heterogeneity.

5.2.1. Performance Comparison

To analyse performance of databases generated by Mortadelo with state-of-the-art approaches, we gathered all approaches that were able to generate databases for column-family stores or document stores. We exclude from this selection those that do not provide automation [20, 35, 32, 33] or that target NoSQL paradigms currently not addressed by Mortadelo [24]. Thus, the selected approaches were Chebotko et al. [21], Mior et al. [22] and de Lima and dos Santos Mello [30]. Then, for each gathered approach, we tried to generate the target databases for the case studies we have modelled in previous section (see Table 2). However, this task was not feasible because some of these state-of-the-art approaches could not be easily replicated. For instance, de Lima and dos Santos Mello [30] do not provide a precise specification of their transformation rules, as they are just informally described. Therefore, we were not able to unambiguously interpret them.

To solve this issue, we used for the comparison the same case study appearing in the paper that describes each approach. For instance, Chebotko et al. [21] use a case study about *venues* to illustrate how their approach works. For this case study, they provide an ER model as well as the Cassandra implementation generated by their approach. So, we used these artifacts to perform the comparisons against Mortadelo. This way, it was ensured that the generated databases were correct, and we have not misinterpreted anything during the database generation process. The same strategy was applied to the other selected approaches.

As commented in Sections 4.6 and 4.7, our transformation rules are heavily grounded in those available in the literature [21, 30, 22]. Consequently, databases generated by Mortadelo and by the selected state-of-the-art approaches have a very similar structure, being even completely identical in some cases. The differences, when found, are due to some optimizations we added to the transformation rules, such as the *query merging mechanism* (see Section 4.6.2). When these differences appear, they affect to a small number of target elements.

Therefore, as most parts of the database designs of Mortadelo and the state-of-the-art approaches are identical, both databases are expected to have a similar performance. Some queries are solved using the same database structures, so they will have identical response times. Therefore, we have left these queries with identical design responses out from this analysis, and we have focused on those queries whose targeted design elements differ between Mortadelo and the other approaches. We present and discuss the results of this comparison in the following.[8]

Mortadelo and Chebotko et al. [21]. To compare Mortadelo with Chebotko's solution, we used the *venues* case study, which contains 9 queries. We modeled that case study in GDM to automatically generate a Cassandra implementation using Mortadelo. 7 out of the 9 queries were answered using exactly the same tables in both solutions. Only two queries, identified as *Q5* and *Q9*, were answered using different structures.

In this case, the differences are due to the query merging mechanism of Mortadelo. In Chebotko et al. [21], for each

[8]A detailed description of the case studies, the artifacts associated to each one of them and of how response times have been measured can be consulted in the supplementary material that accompanies this paper.

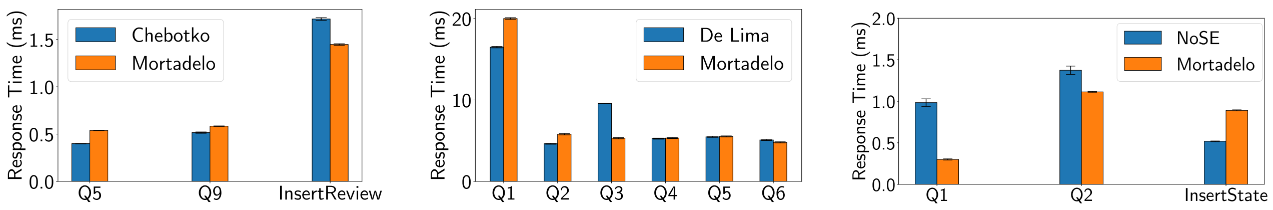


Figure 17: Average response times of Mortadelo vs. Chebotko et al. (left), De Lima and Mello (middle), and Mior et al.'s *NoSE* (right).

query, a table is always generated to support it. Therefore, *Q5* and *Q9* lead to the generation of two different tables. On the other hand, Mortadelo is able to detect that these queries are compatible, by means of checking elements such as query conditions and sorting criteria (see Section 4.6.2 for details). So, these queries could be merged into a single one, generating just one table, instead of two.

To asses the effect of the query merging mechanism, we measured the response times of these queries. Moreover, it should be taken into account that the query merging mechanism was not only designed to improve read performance, but also to reduce redundancy, which should help to improve insert performance and to reduce database size. Therefore, we also measured the cost of adding a new row of data in both cases and the effect of this merging on the database size. Figure 17, left, shows the response times of *Q5* and *Q9* as well as the cost in inserting new data in the generated table.

As it can be seen, concerning the read operations, *Q5* and *Q9* are slightly faster in Chebotko's design. This is because row size of Chebotko's solution is smaller than Mortadelo's, as the Mortadelo table is fragmented in two pieces in Chebotko. Therefore, Cassandra needs to process less data in Chebotko, which improves performance by 23.5% on average. On the other hand, write operations need to access just one table in Mortadelo, whereas two tables are required in Chebotko, which makes Mortadelo 19% faster.

Regarding database size, a noticeable reduction is not finally achieved for this concrete case study. The tables associated to *Q5* and *Q9* in Chebotko share the same key, but the rest of table columns are disjoint. Therefore, we are only avoiding redundancies in the table key. The impact of this redundancy in the database size is negligible.

Mortadelo and de Lima and dos Santos Mello [30]. To compare with de Lima's method, we generated a MongoDB database using Mortadelo for the *Store* case study, which is the same case study that authors of [30] use to illustrate their approach. In this case, the differences between Mortadelo and de Lima are due to the denormalization introduced by Mortadelo. As it was commented in Section 4.7, while de Lima's solution avoids denormalization, Mortadelo nests some data into different collections to improve those operations that need to access several domain entities.

The effect of this denormalization on database performance is illustrated in Figure 17, middle. Queries *Q1*, *Q2*, *Q4*, and *Q5* are read operations that only involve a single collection in both cases, whereas *Q3* and *Q6* accesses several collections.

In the single collection case, as it can be seen, Mortadelo performs pratically equal than de Lima for *Q4*, and *Q5*, and it is a 2.7% and 3.6% slower for *Q1* and *Q2*. In these latter cases, the documents retrieved by Mortadelo have a bigger size, due to the denormalization, which have a negative impact in performance because MongoDB needs to process more data.

Nevertheless, when accessing multiple collections, Mortadelo offers better performance than de Lima. Responses times can be considered equal for *Q6*, but in the case of *Q3*, Mortadelo is a 26% faster. This is because to compute *Q3*, a single collection is accessed in Mortadelo, whereas that two collections must be joined in the De Lima's solution.

Summarising, Mortadelo might perform better than de Lima and dos Santos Mello [30] depending on the characteristics of each case study. In any case, the degree of denormalization in Mortadelo can be controlled by using the *HighlyUpdated* annotation. Therefore, Mortadelo can generate the databases without denormalization when required.

Mortadelo and NoSE [22]. As commented in Section 3, NoSE [22] firstly generates the database designs that might implement the conceptual data model provided as input. Then, NoSE formulates and solves a binary integer problem to find, among all these candidate designs, the one offering the best performance. The database design generated by Mortadelo is always included in this set of candidate designs. So, if Mortadelo's design is not selected, it is because NoSE has found an alternative design with a better performance. This means that we can never improve NoSE performance, just equal it.

Nevertheless, we run some rough experiments to asses how performance might degrade in our approach as compared to NoSE. For these experiments, we used one of the case studies that Mior et al. employed to evaluate their approach, which was the *Easy Anti Cheat* case study. We compared the Cassandra design provided in [22] with the one generated by Mortadelo. In the case of NoSE, the database was generated assuming a workload with 80% of writings and 20% of reads.

For 3 out of the 5 queries for the case study, the target structures of the databases generated by Mortadelo and NoSE are identical, differing only for the two remaining ones, identified as *Q1* and *Q2*. For these queries, Mortadelo creates two different tables, whereas NoSE avoids to denormalize data, creating just a single table. To do it, in this case, NoSE moves part of the logic to compute the query to the application level. This way, read performance decreases, but writing performance improves, as to add new data we have to access just one table, whereas in Mortadelo it would be two. Considering it was specified that there are more insertions than reads, global performance is improved in NoSE. Moreover, database size of the structures associated to these queries is around twice in Mortadelo.

To quantify how much better NoSE performs, we have measured the effect of this denormalization. Figure 17 (right) shows the results. As expected, query performance is better in Mortadelo, particularly in *Q1*. On the other hand, for insertions, NoSE is faster than Mortadelo. Since most of the operations are insertions, NoSE would be faster than Mortadelo by ∼30% in general terms. The obtained performance loss shows that allowing a database designer to move part of the logic of certain operations to the application level, just as NoSE does, could be a good inclusion to Mortadelo's features. This addition could be performed by defining new annotations. It should also be highlighted than this performance penalty limits to *Q1* and *Q2*. For the other queries, there is no performance penalty.

It might be argued that Mortadelo is more efficient than NoSE for reads, whereas that NoSe performs better for insertions. So, it might be thought that, in general, Mortadelo should be used for read-intensive applications and NoSE for write-intensive applications. Nevertheless, it should be noticed that this database design was generated by NoSE assuming an 80% of insertions. If this rate decreased, the generated database design would be closer to Mortadelo's, putting more attention to read operations performance.

5.2.2. *Expert Evaluation*

To complement these academic experiments, we tried to check with industrial practitioners whether the databases generated by Mortadelo have an acceptable quality to be deployed in real settings. For this purpose, we contacted a local company that develops software applications for Industry 4.0 and whose engineers had been using MongoDB and Cassandra in production during several years. They provided us a case study related to a cutting machine for bars that were used in window manufacturing. Using Mortadelo, we modeled this case study, and we generated database implementations for Cassandra and MongoDB. All these artifacts where reviewed for a senior engineer of this company, who only suggested changing the name of some attributes, confirming the generated designs have quality enough to be deployed in production.

5.3. *EI-3: Computational Resources*

This section analyses the computational complexity of our transformation process to verify if it can be executed with a reasonable amount of computational resources.

The transformation process for column stores starts by comparing pairs of queries in order to compact them. So, the complexity of this step would be $O(n^2)$, where n is the number of access queries in the GDM model. After that, a column family is generated for each query throughout a set of basic calculations, except for the *row uniqueness* algorithm. This algorithm builds a kind of access tree and processes its branches. The algorithm iterates over the tree branches, and for each branch, over their nodes, in order to determine whether a branch is identified. This process is repeated each time a new branch is identified, so the complexity, in the worst case, would be $O(p^2q)$, where p is the number of tree branches and q is the maximum path length. In normal scenarios, these numbers are expected to be low, since queries traversing more than 10 entities, i.e., $q = 10$, and following more than 5 different paths, i.e., $p = 5$ are very extreme cases. In any case, the complexity of this transformation process would be polynomial.

In the case of document databases, we iterate over the number of queries to detect the main entities, and to build the access trees of each one of these entities. The complexity of this process is $O(n)$, where n is the number of queries. Then, each access tree is traversed to generate document types. This process would be $O(mr)$, where m is the number of main entities, and r the maximum number of nodes in an access tree. As before, both values are expected to be low. So, the complexity of this transformation process is also polynomial.

To complement and quantify the computational complexity analysis described above, me measured database generation times of the different case studies for the rules for column family and document databases. All generations took only a few seconds to complete, also including any IO operations for managing script files and models. These times are similar to the ones obtained by other rule-based generation approaches, such as Chebotko et al. [21] and de Lima and dos Santos Mello [30].

5.4. *EI-4: Extension Capabilities of Mortadelo*

The structure of our framework facilitates its extensibility. In the following subsections we discuss the required steps to extend the support of Mortadelo for new NoSQL features, technologies and paradigms.

5.4.1. *EI-4.1: Incorporation of New NoSQL Features to the Transformation Process*

NoSQL technologies are still novel, which makes frequent the apparition of new features in their database management systems. If we wanted to take advantage of these new features, we would need to extend or modify one or more elements of our transformation process. We analyze how our platform deals with these changes by extending our column families approach to support a Cassandra feature that was not used in the original transformation process: *user-defined types (UDTs)*.

As explained in Section 4.4, UDTs allow defining structures composed of several fields of different type. In Cassandra, UDTs can be helpful to reduce the number of rows that must be stored in a column family. We illustrate this by using the example of Figure 18. The query *Q2_productsByName* of our running example (see Section 2.1), includes the *categories* data of the *products*. As each *product* might have several *categories*, the column family generated for this query would have a row per combination of a *product* and each one of its *categories*, as shown in Figure 18, left. This is, if a product belongs to three categories, the corresponding column family would store three rows, and data of this product would be replicated in these rows. This can be avoided by creating a user-defined type for *categories*, named *categoryUDT*, as show in Figure 18, right. Now, *categories* can be stored in a single column as a list of

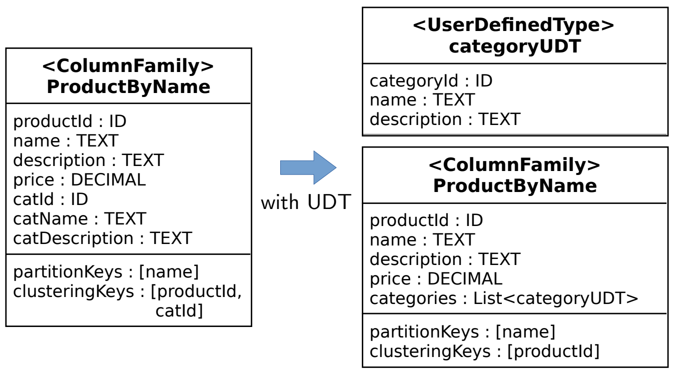


Figure 18: Left: column family where product data has to be replicated for each category; right: the use of a category user-defined type avoids this replication.

this new type. This way, we do not need to replicate products by each category, which reduces the size of the column family.

To incorporate this new feature into Mortadelo, we would need to update the column family logical model. This change was already incorporated to Figure 9, which includes UDTs as one of the possible types of a column. Apart from changes in this metamodel, an update of the code generators that transform elements to and from this metamodel is also required. After that, we would need to devise a rule for deciding when to use a user-defined type, program and incorporate it to our model transformation process. A possible new rule could be as follows: to use a UDT for an entity inside a column family, none of the fields of that entity can belong to the *primary key* of such a column family. Moreover, to make the process of creating a user-defined type affordable, the entity must be easily serializable. So, we consider the creation of UDTs only for those entities that have no further relationships with other entities inside a query. When these conditions are met, a new UDT is created. Once designed, this new rule could be incorporated to the transformation process described in Section 4.6.

5.4.2. EI-4.2: Incorporation of new NoSQL Technologies

New NoSQL systems belonging to any of the supported NoSQL paradigms can be incorporated into Mortadelo by implementing a new code generator. This code generator would transform logical models of the corresponding paradigm into code for the new NoSQL technology. For instance, if we wanted to support the column-family store *ScyllaDB* [27] as a target platform for Mortadelo, we would need to create a *model-to-text transformation*, or code generator, from the column family metamodel of Figure 9 into code to define a database according to the ScyllaDB characteristics. Moreover, if the newly incorporated NoSQL system had a set of specific features, we might have to modify Mortadelo to support them, as we did for user-defined types.

5.4.3. EI-4.3: Incorporation of New NoSQL Paradigms

Currently, Mortadelo supports two NoSQL paradigms: column family and document-based stores. Nevertheless, there are other paradigms that could also benefit from having a design generation process. For instance, *key-value* stores are very popular and extended database systems where data bundles are managed through some sort of identifier or key. So, the data structure of this kind of systems could be understood as a hash table. Redis [45] and Dynamo [46] are some well-known examples of key-value stores.

If we wanted to include support for a new NoSQL paradigm, e.g., the mentioned key-value stores, into Mortadelo, we would need to: (1) provide a logical metamodel for that new paradigm; (2) define and implement the model transformations for converting GDM conceptual models into logical models for the new paradigm; and (3) create a code generator for a target technology of this new paradigm.

As NoSQL paradigms are very heterogeneous, including a new one that differs a lot from the ones already supported might require taking into account certain aspects that did not manifest for the previous paradigms during the design generation process. For instance, most key-value stores do not impose any schema restrictions on the stored data: values could be stored as plain text without schema, XML or JSON documents, or concrete data structures supported by the technologies, such as lists, sets or maps. This schema freedom was not present in column families or document stores, where data is stored in tabular or document-based formats, respectively. As a result, some extra relevant aspects might be required for the purposes of guiding the transformation from the GDM to the new logical model, which might be captured in new GDM annotations.

6. Summary and Future Work

This work has presented *Mortadelo*, a model-driven design process for the generation of NoSQL databases. The main contribution of Mortadelo, when compared with other state-of-the-art approaches, is that, from the same conceptual data model, Mortadelo is able to generate implementations for different NoSQL technologies, such as column family or document-based ones. To the best of our knowledge, this is the first NoSQL database design methodology with these characteristics. To generate these NoSQL databases, Mortadelo first applies a set of predefined rules to transform an instance of a conceptual model into a logical model of a concrete NoSQL paradigm, e.g., column family stores. Then, code generation templates are used to obtain the implementation of a concrete NoSQL technology, such as Cassandra. Some aspects of this transformation process can be customized by providing extra information in the conceptual model through *annotations*. For instance, we can specify that an entity has a high updates ratio, and the transformation process would treat it accordingly.

In general, Mortadelo improves state of the art when compared to Li [23], de Lima and dos Santos Mello [30], Chebotko et al. [21] and Mior et al. [22] by targeting more than one NoSQL paradigm; and when compared to Herrero et al. [33] and Atzeni et al. [32] by automating the database design process. Moreover, oppositely to Chebotko et al. [21] and de Lima and dos Santos Mello [30], the database generation rules can be modified by means of high-level annotations, which allow handling the same transformation scenario differently depending on the particularities of each case study.

To develop Mortadelo, the following elements were created:

- A metamodel, called *Generic Data Metamodel (GDM)*, for the technology-agnostic modeling of NoSQL databases. GDM, to the best of our knowledge, is the first metamodel that integrates the two views that are required for NoSQL database modeling: (i) a view with the specification of *entities* with their *attributes* and *relationships*; and (ii) a view specifying the *data access patterns* that would be used to retrieve and update data.
- *Annotations* for the GDM metamodel that can be used to provide hints to the model transformation process about how some entities should be handled.
- Metamodels for supporting logical models of two different NoSQL paradigms: *column family stores* and *document databases*.
- A set of rules and algorithms were implemented to perform an automatic Model-to-Model transformation of the database modeled by the user with the GDM to the logical models of both supported NoSQL paradigms.
- Another set of rules to generate concrete code implementations for Cassandra (*column family store*) and MongoDB (*document database*) from the logical models, following a Model-to-Text transformation process.

We evaluated Mortadelo's capabilities by using several database case studies from the NoSQL literature. As a result of this evaluation, we concluded that (i) the GDM is expressive enough to model different kind of databases; (ii) databases generated for Cassandra and MongoDB are valid and usable; (iii) the process has a low computational cost; and (iv) Mortadelo can be extended in order to incorporate new NoSQL features to the transformation process. We also described how our proposal can be expanded to support other NoSQL paradigms and technologies.

As future work, we plan to extend Mortadelo to offer a more fine-grained control of the transformation process, so that different implementations of the same database can be created and compared. For example, users can choose to generate a more normalized or denormalized implementation in document databases, or to indicate some performance improving parameters such as how to partition the data in systems where this partitioning is optional, like MongoDB. Moreover, we will study how sharding and replication issues might be orthogonally specified to schema designs, and whether we might be able of automatically deploying completely configured NoSQL systems in, for instance, a cloud platform. Finally, an interesting future research line is, obviously, the extension of Mortadelo to support other NoSQL paradigms, such as key-value stores.

Acknowledgements

Authors would like to express their gratitude to the software engineers from Soincon (http://soincon.es/) for their collaboration during the evaluation of Mortadelo. This work has been partially funded by the Government of Cantabria (Spain) under the doctoral studentship program from the University of Cantabria, and by the Spanish Government under grant TIN2017-86520-C3-3 R.

AppendixA. GDM to Logical Model Transformations

Here we show, in pseudo-code format, the algorithms that transform a GDM instance into another instance conforming to one of the logical models described throughout the article, namely, column family or document-based logical models.

```
Input: An access query $aq$
Output: A column family $cf$
$cf \leftarrow newColumnFamily()$;
$cf.name \leftarrow aq.name$;
foreach $attr \in aq.projections$ do
    $newColumn \leftarrow new\ Column()$;
    $newColumn.name \leftarrow attr.name$;
    $newColumn.type \leftarrow attr.type$;
    $cf.add(newColumn)$;
end
/* extractAttributes : get the attributes from
   the where clause in appearing order          */
foreach $attr \in extractAttributes(aq.condition)$ do
    if $isInEquality(attr, aq.condition)$ then
        $pk \leftarrow new\ PartitionKey()$;
        $pk.column = cf.columns.find(c \mid c.name = at.name)$;
        $cf.partitionKey.add(pk)$;
    end
    if $isInInequality(attr, aq.condition)$ then
        $ck \leftarrow new\ ClusteringKey()$;
        $ck.column = cf.columns.find(c|c.name = at.name)$;
        $cf.clusteringKey.add(ck)$;
    end
end
/* sorting criteria is only included if
   compatible with current clustering key       */
if $compatibleOrdering(cf.clusteringKey, aq.orderingAttributes)$ then
    foreach $attr \in aq.orderingAttributes$ do
        if $attr \notin cf.clusteringKey$ then
            $ck \leftarrow new\ ClusteringKey()$;
            $ck.column = cf.columns.find(c|c.name = attr.name)$;
            $cf.clusteringKey.add(attr)$;
        end
    end
end
$cf \leftarrow ensureUniqueness(cf, aq)$;
```

Algorithm 1: Query to Column Family Transformation Rule.

Algorithm 1 shows how to generate an appropriate column family to answer a given access query. This generated column family would be included in the final design suggested by Mortadelo. As this algorithm does not ensure row uniqueness by itself, we can apply Algorithm 2 using as input the generated column family and the original access query to add the necessary modifications to obtain this uniqueness. A complete description of how these algorithms work can be found in Section 4.6.1.

```
Input: A column family $cf$
Input: An access query $aq$
Output: A column family $cf'$
$cf' \leftarrow cf$;
$paths \leftarrow computeAllPathsFromRoot(aq)$;
$paths \leftarrow removeIdentifiedPaths(paths)$;
while $paths \neq \emptyset$ do
    $p \leftarrow paths.getOne()$;
    // search an id in the path entities
    $id \leftarrow findIndentifier(p)$;
    if $exists(id)$ then
        // Column creation from id attr omitted
        if $column(id) \notin cf'.columns$ then
            $cf'.columns.add(column(id))$;
        end
        $cf'.clusteringKey.add(column(id))$;
        /* This addition might identify more
           than one path                          */
        $paths \leftarrow removeIdentifiedPaths(paths)$;
    else
        /* An artificial id is required to
           identify this path.  This id makes
           the row (and all paths) unique         */
        $dummyId \leftarrow createDummyIdColumn()$;
        $cf'.columns.add(dummyId)$;
        $cf'.clusteringKey.add(dummyId)$;
        $paths \leftarrow \emptyset$ ; // end of the process
    end
end
```

Algorithm 2: Ensure uniqueness of column family rows.

During this work, we have also described some optimizations that can be applied to reduce the number of column families that are generated when proposing a new design. Algorithm 3 defines a method that reduces the number of queries to process by Algorithms 1 and 2, based on the possibility to use the same column family to answer several access queries. Therefore, the application of this method can contribute to avoid unnecessary data redundancy. See Section 4.6.2 for more details.

Lastly, Algorithms 4 and 5 are used to transform a GDM instance into a document data model. First, Algorithm 4 is used to obtain the different collections that will be present in the output data model. During this obtention, the root document type of each collection is populated through Algorithm 5. This process is defined in Section 4.7.

```
Input: A set of queries $Qs$
Output: A set of compacted queries $CQs$
$CQs \leftarrow Qs$;
foreach $q_i \in Qs$ do
    if $q_i \notin CQs$ then
        // Query previously compacted
        continue;
    end
    foreach $q_j \in CQs - \{q_i\}$ do
        if $\neg compatibleUniqueness(q_i, q_j)$ then
            // Different uniqueness sets
            continue;
        end
        if $equalities(q_i) = equalities(q_j)$
         $\wedge\ inequalities(q_i) = inequalities(q_j)$ then
            if $(q_i.orderingAttrs = q_j.orderingAttrs)$
              $\vee\ (hasOrdering(q_i)\ \wedge \neg hasOrdering(q_j))$
              $\vee\ (q_j.orderingAttrs \subset q_i.orderingAttrs)$ then
                $q_i.projections \leftarrow$
                  $q_i.projections \cup q_j.projections$;
                $CQs \leftarrow CQs - \{q_j\}$;
            else if $(\neg hasOrdering(q_i)\ \wedge hasOrdering(q_j))$
              $\vee\ (q_i.orderingAttrs \subset q_j.orderingAttrs)$ then
                $q_j.projections \leftarrow$
                  $q_j.projections \cup q_i.projections$;
                $CQs \leftarrow CQs - \{q_i\}$;
            end
        end
    end
end
```

Algorithm 3: Query merging optimization.

```
Input: A GDM instance model $gdm$
Output: A document data model $ddm$
$mainEntities \leftarrow gdm.queries.collect((q) \mid q.from)$;
foreach $me \in mainEntities$ do
    $col \leftarrow new\ Collection()$;
    $col.name \leftarrow me.name$;
    $accessTree \leftarrow allQueryPaths(me, gdm.queries)$;
    $col \leftarrow populateDocumentType(col.root, accessTree)$;
    $ddm.collections.add(col)$;
end
```

Algorithm 4: GDM to document data model transformation.

```
Input: A document type $dt$
Input: A $node$ of an access tree
Output: A constructed document type $dt$
$nodeAttributes \leftarrow$
  $node.entity.features.select(f|f.isTypeOf(Attribute))$;
$nodeReferences \leftarrow$
  $node.entity.features.select(f|f.isTypeOf(Reference))$;
foreach $attr \in nodeAttributes$ do
    $pf \leftarrow new\ PrimitiveField()$;
    $pf.name \leftarrow attr.name$;
    $pf.type \leftarrow attr.type$;
    $dt.fields.add(pf)$;
end
foreach $ref \in References$ do
    $targetNode \leftarrow node.arcs.find(a|a.name =$
      $ref.name).target$;
    if $exists(targetNode)$ then
        $baseType \leftarrow new\ DocumentType()$;
        $populateDocumentType(baseType, targetNode)$;
    else
        $baseType \leftarrow new\ PrimitiveField()$;
        $baseType.type \leftarrow findIdType(ref.entity)$;
    end
    $baseType.name \leftarrow ref.name$;
    if $ref.cardinality = 1$ then
        $dt.fields.add(baseType)$;
    else
        $arrayField \leftarrow new\ ArrayField()$;
        $arrayField.type \leftarrow baseType$;
        $dt.fields.add(arrayField)$;
    end
end
```

Algorithm 5: Populate a *DocumentType* given an access tree.

References

[1] L. Ang, K. P. Seng, A. M. Zungeru, G. K. Ijemaru, Big sensor data systems for smart cities, IEEE Internet of Things Journal 4 (2017) 1259–1271. doi:10.1109/JIOT.2017.2695535.

[2] C. Costa, M. Y. Santos, Reinventing the energy bill in smart cities with nosql technologies, in: S.-i. Ao, G.-C. Yang, L. Gelman (Eds.), Transactions on Engineering Technologies, Springer Singapore, Singapore, 2016, pp. 383–396. doi:10.1007/978-981-10-1088-0_29.

[3] Y. Lu, Industry 4.0: A survey on technologies, applications and open research issues, Journal of Industrial Information Integration 6 (2017) 1 – 10. doi:10.1016/j.jii.2017.04.005.

[4] M. Y. Santos, J. O. e Sá, C. Andrade, F. V. Lima, E. Costa, C. Costa, B. Martinho, J. Galvão, A big data system supporting bosch braga industry 4.0 strategy, International Journal of Information Management 37 (2017) 750 – 760. doi:10.1016/j.ijinfomgt.2017.07.012.

[5] E. Barbierato, M. Gribaudo, M. Iacono, Performance evaluation of nosql big-data applications using multi-formalism models, Future Generation Computer Systems 37 (2014) 345 – 353. doi:10.1016/j.future.2013.12.036.

[6] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, S. Schiaffino, Persisting big-data: The nosql landscape, Information Systems 63 (2017) 1 – 23. doi:10.1016/j.is.2016.07.009.

[7] R. Hecht, S. Jablonski, NoSQL Evaluation: A Use Case Oriented Survey, in: Int. Conf. on Cloud and Service Computing (CSC), IEEE, 2011, pp. 336–341. doi:10.1109/CSC.2011.6138544.

[8] F. Gessert, et al., NoSQL Database Systems: A Survey and Decision Guidance, Computer Science - Research and Development 32 (2017) 353–365. doi:10.1007/s00450-016-0334-3.

[9] K. Chodorow, MongoDB: The Definitive Guide: Powerful and Scalable Data Storage, O'Reilly Media, 2013.

[10] J. C. Anderson, J. Lehnardt, N. Slater, CouchDB: The Definitive Guide: Time to Relax, O'Reilly Media, 2010.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, ACM, New York, NY, USA, 2007, pp. 205–220. doi:10.1145/1294261.1294281.

[12] J. L. Carlson, Redis in action, Manning Publications Co., 2013.

[13] E. Hewitt, Cassandra: the definitive guide, O'Reilly Media, 2010.

[14] L. George, HBase: the definitive guide: random access to your planet-size data, O'Reilly Media, 2011.

[15] T. Vajk, P. Fehér, et al., Denormalizing Data into Schema-Free Databases, in: 4th Int. Conf. on Cognitive Infocommunications, IEEE, 2013, pp. 747–752.

[16] T. Haerder, A. Reuter, Principles of Transaction-Oriented Database Recovery, ACM Computer Surveys (1983) 287–317. doi:10.1145/289.291.

[17] D. G. Chandra, Base analysis of nosql database, Future Generation Computer Systems 52 (2015) 13 – 21. doi:10.1016/j.future.2015.05.003.

[18] E. F. Codd, The Relational Model for Database Management: Version 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[19] P. P.-S. Chen, The Entity-relationship Model–Toward a Unified View of Data, ACM Trans. Database Syst. 1 (1976) 9–36. doi:10.1145/320434.320440.

[20] L. Li, X. Zhao, UML Specification and Relational Database., Journal of Object Technology 2 (2003) 87–100. doi:10.5381/jot.2003.2.5.a1.

[21] A. Chebotko, A. Kashlev, S. Lu, A Big Data Modeling Methodology for Apache Cassandra, in: International Congress on Big Data, IEEE, 2015, pp. 238–245. doi:10.1109/BigDataCongress.2015.41.

[22] M. J. Mior, K. Salem, et al., NoSE: Schema Design for NoSQL Applications, IEEE Transactions on Knowledge and Data Engineering 29 (2017) 2275–2289. doi:10.1109/TKDE.2017.2722412.

[23] C. Li, Transforming Relational Database into HBase: A Case Study, in: IEEE Int. Conf. on Software Engineering and Service Sciences, 2010, pp. 683–687. doi:10.1109/ICSESS.2010.5552465.

[24] G. Daniel, G. Sunyé, J. Cabot, UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases, in: Conceptual Modeling, Springer, 2016, pp. 430–444. doi:10.1007/978-3-319-46397-1_33.

[25] J. Carpenter, E. Hewitt, Cassandra: The Definitive Guide: Distributed Data at Web Scale, O'Reilly, 2016.
[26] R. Cattell, Scalable SQL and NoSQL Data Stores, SIGMOD Records 39 (2011) 12–27. doi:10.1145/1978915.1978919.
[27] ScyllaDB, https://www.scylladb.com/, since 2015.
[28] HBase, https://hbase.apache.org/, since 2008.
[29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2008) 4:1–4:26. doi:10.1145/1365815.1365816.
[30] C. de Lima, R. dos Santos Mello, A workload-driven logical design approach for NoSQL document databases, in: Proceedings of the 17th International Conference on Information Integration and Web-based Applications &Services (iiWAS), ACM Digital Library, 2015, pp. 73:1–73:10. doi:10.1145/2837185.2837218.
[31] R. Schroeder, D. Duarte, R. S. Mello, A workload-aware approach for optimizing the xml schema design trade-off, in: Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, iiWAS '11, ACM, New York, NY, USA, 2011, pp. 12–19. doi:10.1145/2095536.2095542.
[32] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the nosql world, Computer Standards & Interfaces (2016). doi:10.1016/j.csi.2016.10.003.
[33] V. Herrero, A. Abelló, O. Romero, NoSQL Design for Analytical Workloads: Variability Matters, in: Conceptual Modeling (ER), 2016, pp. 50–64. doi:10.1007/978-3-319-46397-1_4.
[34] M. Lawley, R. Topor, A query language for eer schemas, in: ADC'94 Proceedings of the 5 th Australian Database Conference, Global Publications Service, 1994, pp. 292–304.
[35] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to NoSQL systems, Information Systems 43 (2014) 117 – 133. doi:10.1016/j.is.2013.05.002.
[36] A. Kleppe, The Object Constraint Language, 2 ed., Addison-Wesley, 2003.
[37] P. Atzeni, P. Cappellari, et al., Model-independent Schema Translation, The VLDB Journal 17 (2008) 1347–1370. doi:10.1007/s00778-008-0105-2.
[38] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd ed., Addison-Wesley Professional, 2009.
[39] L. M. Rose, R. F. Paige, D. S. Kolovos, F. A. Polack, The Epsilon Generation Language, in: Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications, Springer-Verlag, 2008, pp. 1–16. doi:10.1007/978-3-540-69100-6_1.
[40] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, Addison-Wesley Professional, 2008.
[41] M. Eysholdt, H. Behrens, Xtext: Implement Your Language Faster than the Quick and Dirty Way, in: 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2010, pp. 307–309. doi:10.1145/1869542.1869625.
[42] J. Hunt, Inheritance Considered Harmful!, Springer Professional Computing, Springer, 2003, pp. 363–381.
[43] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2002.
[44] D. S. Kolovos, F. Medhat, R. F. Paige, D. D. Ruscio, T. van der Storm, S. Scholze, A. Zolotas, Domain-specific languages for the design, deployment and manipulation of heterogeneous databases, in: Proceedings of the 11th International Workshop on Modelling in Software Engineerings, MiSE@ICSE 2019, Montreal, QC, Canada, 2019, pp. 89–92. URL: https://dl.acm.org/citation.cfm?id=3340728.
[45] Redis, https://redis.io/, since 2009.
[46] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (2007) 205–220. doi:10.1145/1323293.1294281.