information

Article

KVMod—A Novel Approach to Design Key-Value NoSQL Databases

Ahmed Dourhri, Mohamed Hanine * and Hassan Ouahmane

LTI Laboratory, National School of Applied Sciences, Chouaib Doukkali University, El Jadida 24000, Morocco; dourhriahmed@gmail.com (A.D.); hassan.ouahmane@yahoo.fr (H.O.)
* Correspondence: hanine.m@ucd.ac.ma

Abstract: The growth of structured, semi-structured, and unstructured data produced by the new applications is a result of the development and expansion of social networks, the Internet of Things, web technology, mobile devices, and other technologies. However, as traditional databases became less suitable to manage the rapidly growing quantity of data and variety of data structures, a new class of database management systems named NoSQL was required to satisfy the new requirements. Although NoSQL databases are generally *schema-less*, significant research has been conducted on their design. A literature review presented in this paper lets us claim the need to create modeling techniques to describe how to structure data in NoSQL databases. Key-value is one of the NoSQL families that has received too little attention, especially in terms of its design methodology. Most studies have focused on the other families, like column-oriented and document-oriented. This paper aims to present a design approach named *KVMod* (key-value modeling) specific to key-value databases. The purpose is to provide to the scientific community and engineers with a methodology for the design of key-value stores using the maximum automation and therefore the minimum human intervention, which equals the minimum number of errors. A software tool called *KVDesign* has been implemented to automate the proposed methodology and, thus, the most time-consuming database modeling tasks. The complexity is also discussed to assess the efficiency of our proposed algorithms.

Keywords: data modeling; database design; NoSQL; key-value; MDA

Citation: Dourhri, A.; Hanine, M.; Ouahmane, H. KVMod—A Novel Approach to Design Key-Value NoSQL Databases. *Information* **2023**, *14*, 563. https://doi.org/10.3390/info14100563

Academic Editor: Giuseppe Psaila

Received: 3 July 2023
Revised: 23 September 2023
Accepted: 9 October 2023
Published: 12 October 2023

1. Introduction

The volume and variety of data that are produced, modified, analyzed, and archived have rapidly increased as a result of the rise in user-driven content. Moreover, a large quantity of data is being produced by new sources, such as sensors, GPS, automatic trackers, and monitoring systems. These huge volumes of data, often named Big Data, are posing new challenges and opportunities for storage, analysis, and archiving [1,2].

Despite being useful for structured data, the classic relational approach to database design faced considerable difficulties as data requirements changed. Flexibility posed a significant issue. Because relational databases frequently used rigid schemas, it was difficult to deal with evolving data structures or new data types without making major modifications. This rigidity limited the capacity of companies to be agile and innovative as they dealt with a variety of dynamic data sources.The growing need for high availability presented another challenge. Relational databases have generally been single-server applications, rendering them exposed to hardware issues and service interruptions. Ensuring continuous access to data required expensive hardware investments and intricate failover mechanisms. In response to the need to solve these issues associated with the massive volumes and variety of data, a class of new types of systems identified as NoSQL have emerged [3].

However, NoSQL systems have some characteristics in common [4,5]:

1. They adopt flexible models of data, mostly schema-less;

2. Eventual consistency transactions are reached by relaxing the ACID (Atomicity, Consistency, Isolation, and Durability) properties to scale out while achieving high availability and low latency;
3. Query performance is achieved not only through data co-location but also through horizontal and elastic scalability;
4. Data can be easily replicated and horizontally partitioned over remote and local servers.

From another perspective, these systems are heterogeneous in terms of their characteristics, especially their data models. Because it is widely accepted by the scientific community, the family-based classification is adopted in this study [6].

- Key-value databases store data as a dictionary. Every item in the database is stored as a pair <k,v>, where k stands for key and represents an attribute name and v its value. Key-value databases ensure high performance in reading and writing operations. Redis and Riak KV are popular systems in this category;
- Document-oriented databases extend the key-value concepts by representing the value as a document encoded in conventional semi structured formats (like JSON). The advantage of this model is that it can retrieve a set of hierarchically structured information from a single key. MongoDB and CouchDB are document-oriented systems;
- Column-oriented databases: in a database of this family, data are grouped into column families whose schemas are flexible. A column family contains a set of columns. A column has a name, a timestamp, and a value with a complex or simple structure. Each column is stored in a separate location. Cassandra and HBase are examples of column-oriented systems;
- Graph databases represent a database as a graph structure. A graph is composed of a set of nodes (i.e., objects) and a set of edges describing the relationships between the nodes. These databases are efficient when data are strongly connected in which each of its nodes is reachable from the others. Neo4j is one of these systems.

These NoSQL systems initially appeared at the physical level, and, therefore, at the beginning, they lacked defined design approaches. Database design for relational databases, which is usually based on conceptual schemas such as Entity-Relationship or class diagrams, is not sufficient to design NoSQL databases. Database designers in the NoSQL context must capture not only the data to be stored but also how these data will be handled. Traditional data design approaches do not offer a suitable solution for these problems because they were created essentially to satisfy the ACID properties. In NoSQL, a database is considered "schemaless" because it does not require a predefined schema like relational databases; however, Atzeni [7] claims that the effects of data modeling can be beneficial and present two research lines. First, the diversity of systems and models could create difficulties for the database stakeholders. Therefore, modeling-based approaches get their legitimacy from the need to standardize access. Second, data models can serve as a basis for describing approaches at the logical and physical levels. Similarly, Rani and Kaur [8] argue that modeling NoSQL databases is still necessary in order to properly understand data storage. Chebotko et al. [9] also demonstrate a methodology for modeling a Cassandra database from a conceptual data model and a set of access queries. The authors used an example to demonstrate how structuring data in NoSQL stores can influence data size and query performance. Roy-Hubara et al. [10] proposed a method to create a graph database schema from an ERD (Entity Relationship Diagram) via a set of mapping rules.

The previously mentioned approaches require designers to manually apply a series of guidelines or heuristics to design a NoSQL database; this is why the MDA [11] can provide some automation and thus minimize human intervention. The MDA (Model-Driven Approach) relies on three layers to specify systems: the Computation-Independent Model (CIM), the Platform-Independent Model (PIM), and the Platform-Specific Model (PSM) [6]. The MDA is based on metamodels that are defined at each level; any model created is a metamodel instance. Then, this approach uses a series of model transformations to switch from one model to another [6]. De la Vega et al. present Mortadelo, a new methodology to design NoSQL stores using MDA [12]. Although the Mortadelo authors

provide consistent work, their proposal supported only column and document stores. Ait Brahim et al. [13] use an automatic MDA-based approach to transform UML conceptual schemas into NoSQL ones. The study has targeted column-, document-, and graph-oriented systems. Furthermore, Abdelhedi et al. [14] enrich the work of Ait Brahim et al. by dealing with OCL constraints.

NoSQL modeling works focused on column, document, and graph databases. The key-value databases also require works dealing with their particularities and treating them deeply. The existing works in NoSQL data modeling can be exploited to introduce a key-value data-modeling process since column, document, and graph databases can be seen as extended key-value ones [15,16]. To address this gap, this paper aims to present *KVMod*, a model-driven process for key-value database design. Regarding the literature search, hardly any studies, if any, have addressed this crucial topic. *KVMod*, to the best of our knowledge, is the first key-value data-modeling methodology based on MDA principles. This process claims to create a conceptual data model and an access query model to capture the data and functional requirements. Then, via a set of mapping rules, the process aims to automatically generate logical models and physical implementations for NoSQL key-value systems. In this paper, the physical level is meant to be the detailed data model provided by the specific DBMS. To visualize data models, a tool called *KVDesign* was developed to automate key-value data design according to the proposed methodology.

The rest of the article is organized as follows: Section 2 describes the running example used throughout the article and presents the key-value databases. Section 3 reviews the related works. Then, Section 4 details the different phases of the proposed data-modeling methodology. Section 5 assesses the modeling methodology and presents a software tool that implements it. Finally, Section 6 is dedicated to conclusions and future work.

2. Background

In order to make the paper self-contained, this section gives some background information on the technology key-value data stores. It presents the example used to illustrate the study concepts.

2.1. Running Example

This study relies on the *"airflights"* database, which represents an *airline flight management* platform around the world. A conceptual data model for the airline flight management platform is shown in Figure 1 using UML language. As it appears in the model, this platform manages *Aircraft, Flights, Airports, and Passengers* as principal entities. A *flight* is ensured by one and only one *aircraft*, while an *aircraft* can ensure several *flights*. A *flight* can be subject to several *localization* operations. *Passengers* can benefit from web service *access* during the flight using a free hotspot that logs visited sites.

The *air flight management* application, used to manage *air flights*, employs data from the class diagram of Figure 1. The following patterns are used to retrieve this data:
Q1 *Registration number* and *capacity* of *aircraft* whose *capacity* is within a given interval.

Q2 *Airports* of a *country (name, I.C.A.O code, and city)* sorted by ascending order of *cities*.
Q3 *Full name* and *passport number of passengers* on a specific *flight*.
Q4 *List of passengers* departing from a *country* on a specific date, sorted by ascending order of departure *cities*, then by ascending order of *departure time*, the destination *city* must be also displayed.
Q5 *Passengers* departing on a specific *date*, sorted by ascending order of the departed *country* and then ascending order of departed *cities*. The *flight code* and the *departure time* must be displayed
Q6 List of *websites* accessed by *passengers* on a given *flight*. The list is sorted in ascending order of *passenger ID*, then in descending chronological order of the *access time*.
Q7 *Aircraft* departed from an *airport* during a given period. The *departure time and date* must be displayed.
Q8 *Localization* data (geographical coordinates, crossing date, and time) of an *aircraft* used

in a *flight* in descending chronological order.
Q9 List of the *Flights* on a specific date, sorted in ascending order of *flight level*. Departure and arrival *cities* must be displayed.

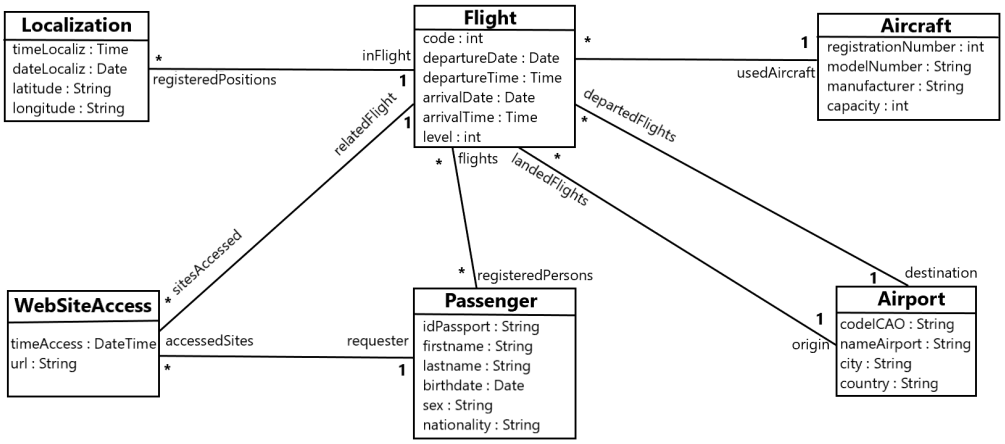

Figure 1. "Airflights database" running example schema.

2.2. Key-Value Family Stores

A key-value database is represented as a matrix of two columns: the first column contains the keys, and the second one contains the values related to the keys. In the context of programming languages, such a matrix is known as a *dictionary* or an *associative array*. The data can be easily accessed by looking up the key, which is a unique value in the set. These structures provide very efficient access; the data are accessible in an O(1) average time [17].

Relying on data persistence, key-value databases can be classified into three types [4]:

1. In-memory key-value systems, allowing particularly fast access to data using memory to store it, such as Memcached;
2. Persistence key-value systems, using disk to store data, such as Riak KV system;
3. Hybrid key-value systems, that put data in memory and save them if necessary, such as Redis.

The key-value systems offer several data types [18]: strings, lists, sets, hashes, and zsets (zset is a map of string members and numerical scores, ordered by scores). Furthermore, the new versions include additional types suitable for new use cases [19]. Redis and Riak KV are some popular members of this family. Because of this, Redis will be detailed later and used to illustrate how to model a key-value database.

To ignore the technical details at the physical level, a logical data model has been defined between the conceptual and physical models. The advantage would be the usability of this study to design databases for other key-value systems. At the physical level, a hash-like data structure is used. Hashes are used to store a map of attributes and their values against a key [20]. Application developers often use Redis hashes, named HSets, to represent their domain objects. In a logical model, the word *"associative array"* is used to designate these data types.

Key-value databases are frequently referred to as "schemaless", which seems to imply that creating a model before work begins is not required. These systems offer structures that are not always linked together in a relational sense. However, for example, an *air flight* database will include various business objects like *Airport, Flight, Aircraft, and Passenger* that need to be linked. One of the first things to decide is how to structure the data. This work represents a key-value database as a set of business objects and models it as a set of n *collections*, $\{C_1, C_2, \ldots, C_n\}$. Each *collection* C_i is itself a series of *associative arrays* that are accessible from the identifier $C_i{:}k_j$, where k_j is the key associated with the jth array in the ith *collection* C_i. The value associated can be seen as a mathematical matrix with two

columns: the first is the column of fields, and the second is the column of values associated with the fields. An example is illustrated in Figure 2 which represents an airport object with the fields: *name*, *city*, *country*, and *ICAO code*.

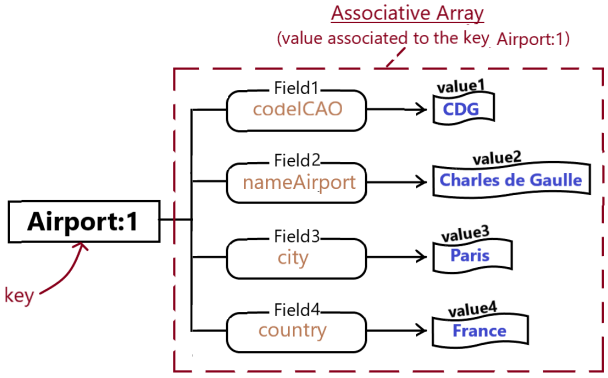

Figure 2. Example of an associative array in a key-value DBMS.

In this study, the set $\{k_1, k_2, \ldots, k_j, \ldots, k_m\}$ is considered a sequence of integers in arithmetic progression with an increment step of 1 (m represents the number of *associative arrays* within a *collection*). Representing a key of an *associative array*, the $C_i : k_j$ is automatically incremented by the DBMS to ensure the uniqueness of the identifier values of the keys. Figure 3 presents an example of a *collection* in the key-value context.

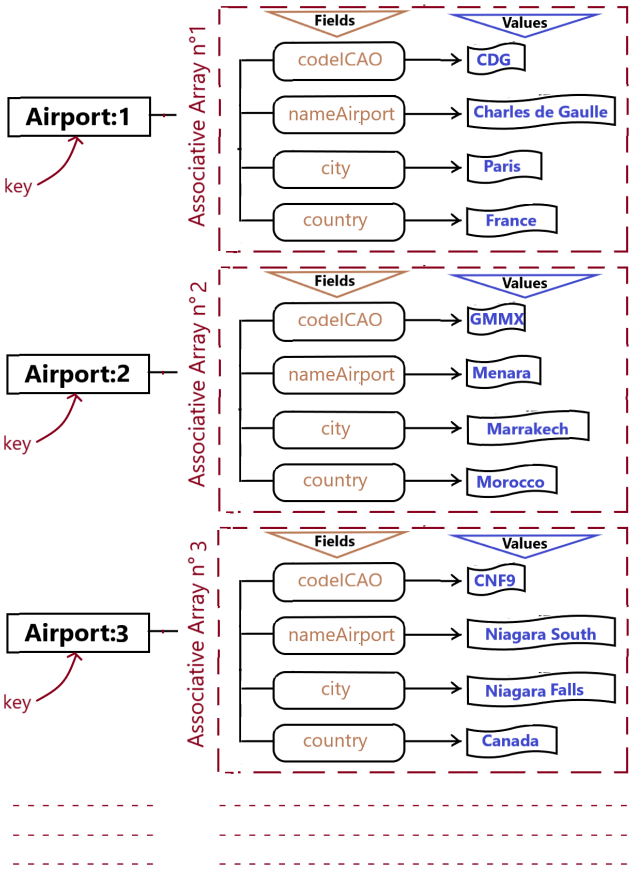

Figure 3. Associative arrays of a collection in key-value DBMS.

3. Related Works

With the arrival of NoSQL systems, various studies became interested in data modeling, proposing different approaches that have been developed to address this topic. Table 1 summarizes a list of these approaches.

Table 1. Approaches for designing NoSQL databases.

Study	Family [1]	Conceptual	Logical	Physical	Access Query Support	MDA Use
Chebotko et al. [9]	Col	ER	Chebotko diagram	Cassandra	Yes	No
De la Vega et al. [12]	-Doc -Col	ER	-Column metamodel- Document metamodel	Cassandra Mongo	Yes	Yes
Shoval et al. [10]	G	ER	ER	Neo4j	No	No
Gwendal et al. [21]	G	-ERD (for data) -OCL (for constraints)	-Graph metamodel (data) -Gremlin (constraints)	Neo4j OrientDB	No	Yes
Ait Brahim et al. [13]	-Doc -Col -G	ER	Generic Logical Metamodel	Mongo Cassandra Neo4j	No	Yes
Martinez-Mosquera et al. [22]	KV	ER	ER	Generic	No	Yes
Rossel et al. [23]	KV	ER	Rossel	Generic	No	No

[1] **KV**: key-Value; **Doc**: Document; **Col**: Column; **G**: Graph.

One of the earliest works on NoSQL database design was provided by Li [23]. His work introduced a set of techniques to transform traditional relational databases into *HBase*. To build an HBase database schema, first a relational schema would need to be created, which could take more time than the creation of a NoSQL schema directly from a conceptual data one.

Imam et al. proposed a series of document-oriented guidelines to design logical and physical models [24]. This work may serve as a learning tool for beginners to learn how industry experts use the guidelines and analyze the relationships between datasets.

In reference [25], Dos Santos Mello and De Lima present a design approach for converting an ERD (i.e., entity relationship diagram [26]) into a document-oriented schema and then into a *MongoDB* physical one. The authors highlight how access patterns are important when designing a NoSQL store because, based on this, some mapping strategies might be more appropriate than others. Due to this, authors have enriched Entity-Relationship diagrams with details about the estimated application workload, which is expressed using an XML-like technique [27].

Chebotko et al. [9] provided the basis for a query-driven methodology to model a column-oriented database. Data and access queries are captured, then, using a set of mapping rules, the methodology explains how to produce the logical column family model and the physical one under the *Apache Cassandra* system.

The authors of reference [13] propose an MDA-based approach that generates several physical data models from a conceptual schema. Unfortunately, the access queries were not considered despite their importance, mainly for accelerating data access.

Abdelhedi et al. enrich the previous work by also considering the constraints dimension with an MDA approach, despite the fact that access queries were not supported [14]. The result is a physical data model and an OCL code that describes the integrity constraints of the database.

The work presented in [12] is one of the few works that have covered the various aspects of NOSQL data modeling. It presents a query-driven methodology to model a document- or column-oriented database. Initially, the methodology captures the data and organizes it in a structural model, then captures the access queries as a query model. Afterwards, MDA concepts are used to generate the logical and physical models in a specified NOSQL family through a series of transformation rules. Finally, optimization techniques are used to eventually merge collections or column families and reduce data duplication and, therefore, the database size.

Shoval et al. proposed a data-modeling process for graph-oriented databases [10]. Firstly, an ERD would be created to represent the different data in a domain. In a second step, an adjusted version of the ERD should be developed to convert n-ary, inheritance, and aggregation relationships to ordinary binary relationships. In the last step, a graph database schema would be created with related DDL (i.e., data definition language) code on the targeted system.

Gwendal et al. [21], on the other hand, worked on graph data modeling using the MDA approach. Moreover, the paper supports database constraints. So, a physical data graph model is produced to organize domain data, and a *gremlin* code is also generated to represent data constraints.

Garcia-Molina et al. present a unified metamodel for relational and NoSQL paradigms, describing how each data model is integrated and mapped [28]. At the conceptual level, the study supports the different relationships between entities like aggregation, generalization, references, and edges. Their work presents the transformation rules to apply in order to obtain physical models. The authors argue that the work is useful for data modeling, whether forward or reverse engineering, but that the heterogeneity of systems and continuous innovation in NOSQL technology are barriers to applicability.

It should be noted that key-value modeling did not get enough attention in comparison with other NOSQL families. Behind this, key-value databases are often used without a predefined schema. For reference, some papers on key-value modeling are cited:

Martinez-Mosquera et al. [29] propose an approach for key-value data modeling using MDA concepts. The proposal is suitable for unstructured or semi-structured data, but it can be extended to structured data with some changes. Modeling activity has been divided into three phases expressed in UML: deployment diagram, class diagram, and key-value model. Initially, a UML deployment diagram is created to specify the physical resources at the conceptual level. Secondly, it should create a UML class diagram as an intermediate step before applying a set of transformation rules using the *QVT* standard, and finally, the key-value model has been produced.

Rossel et al. illustrate useful concepts for key-value big data design [22]. At first, a class diagram is created to capture and structure data, and then a series of rules should be applied to obtain the final schema of the datasets.

To the best of our knowledge, no work related to key-value database design has been able to propose a query-driven methodology using the MDA approach. To fill this gap, we have built *KVMod*, a key-value data-modeling process that automatically generates database implementations for key-value from a conceptual model.

4. Proposed Methodology

The general elements of the transformation process, as specified by *KVMod*, are described in the first subsection. More detail on these elements is provided in the next subsections.

4.1. General Overview

Figure 4 depicts the design process of a key-value database to follow when using *KVMod*. As introduced, *KVMod* uses the MDA approach. This means that *KVMod* relies on models that conform to a metamodel. *KVMod* begins by building a conceptual data model, which is transformed into a logical one data model before producing a physical implementation of the code generation for a selected key-value system.

As other authors [9,30] have previously indicated, it is not sufficient to create a conceptual data schema that specifies which entities compose the system and the relationships between them in order to design a NoSQL database. In NoSQL systems, it is important to know how these entities will be handled at runtime. This is why traditional data-modeling languages need to be improved with new tools for specifying how the data will be accessed. This implies handling two separate and related models; each one respects a metamodel. To ensure that the approach is integrative

- We proposed that the two metamodels be placed together in the *Platform-Independent Data Metamodel (PIDM)*, a conceptual-level metamodel in which both data and access queries are incorporated;
- The advantage is to avoid the complexity that can result from working on one model separately from the other. which can produce a work on a model that is far from the other;
- Due to its platform independence, many NoSQL paradigms (including the key-value one) can use this metamodel as input.

Afterward, at the logical level, we created *KVLM*, which stands for *Key-Value Logical Metamodel*, and it is an intermediate representation that contains information specific to the paradigm key value. *KVMod* begins with the definition of two metamodels: a *PIDM* and a *KVLM* (Key-Value Logical Metamodel). *KVLM* that is an intermediate representation that contains information specific to the paradigm key-value.

The process works according to the following steps:

1. An instance of the *PIDM* metamodel is provided as input to the transformation process.
2. The instance will be checked to see if it is error-free using the metamodel specifications.
3. Then, a M2M (i.e., model-to-model) transformation translates this instance into another one of *KVLM* by applying a set of transformation rules.
4. Finally, the third step of the process is a M2T (model-to-text) transformation that is performed to generate a physical implementation under the targeted technology (i.e., how data are structured on the machine key-value DBMS).

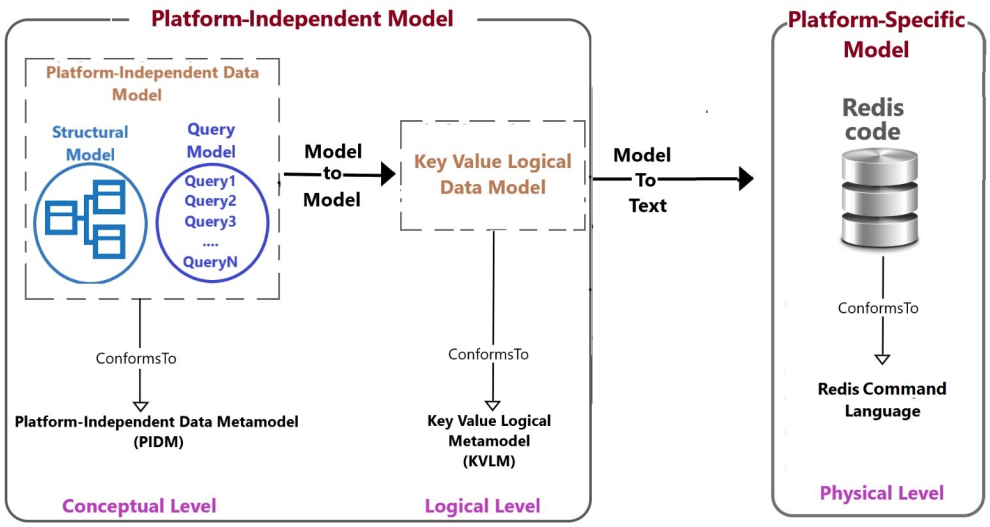

Figure 4. Transformation process of *KVMod* approach.

4.2. Platform-Independent Data Metamodel (PIDM)

As mentioned previously, instances of the Platform-Independent Data Metamodel *PIDM* are used as input for KVMod. Figure 5 depicts this metamodel, which contains the following:

- The ***structural model*** (Figure 5, left). It is defined in a UML-like syntax that is a widely used notation by developers and researchers in data-modeling. This notation is adequate to capture and structure the date requirements of domain data using the following rules:
  - The data are grouped into entities;
  - An entity contains attributes, which represent the data of its occurrences;
  - An entity may have references to other entities;
  - A reference can have constant cardinality, e.g., 0, 1, 2, 4, or unlimited cardinality, denoted as *.
- The ***query model*** (Figure 5, right) that represents the queries that will be sent to the database. Using an SQL-like syntax, these queries are defined in the *PIDM* metamodel over entities of the *structural model*. Navigation through these entities is performed by traversing their references. A query consists of the following clauses:
  - **FROM**. This clause specifies the *main entity* in which the query is executed;
  - **INCLUDE**. If the query needs other entities, references to the *main* entity can be added as *inclusion elements*. Inclusions can be recursively added while there are available references. This means that entities referenced in the inclusion clause can also be incorporated;
  - **SELECT**. It is used for the *projection* operation (i.e., the set of attributes to be retrieved by the query). The attributes to retrieve can come from the *main entity* or the *inclusion* entities;
  - **WHERE**. Using this key-word, a query can contain a boolean expression to filter occurrences that satisfy a given condition;
  - **ORDERBY**. As in SQL, it specifies the sorting attributes of a query result. The ordering can be in ascending or in descending direction. On another side, in multi-criteria ordering, the priority degrees are expressed using *weights* that are assigned to the sorting attributes. The weight is an integer number that provides information on the priority degree of the attribute. As an example, 1 for the most important sorting attribute, 2 for the second-most important attribute, etc.;
  - **AS**. It is used to give an easily identifiable name for an attribute in a query. It is useful to rename the references as well. An *alias* of an attribute can change from one query to another. Thus, a class association entitled *alias* is placed between the query and the attribute classes in the metamodel.

The set of attributes to be retrieved by the query is specified via the *projection* operation using the *Select* clause. An *alias* of an attribute can change from one query to another. Thus, a class association entitled *alias* is placed between the query and the attribute classes in the metamodel. The set of attributes to retrieve can come from the *main entity* or the *inclusion* entities. A query can contain a boolean expression to filter occurrences that satisfy a given condition. Finally, to order the result, an *ORDERBY* clause can be added. The model supports multi-criteria ordering in both directions (ascending and descending). In multi-criteria ordering, the priority degrees are expressed using *weights* that are assigned to the criteria (i.e., the sorting attributes). When the system sorts the query result, the *weight*, which is an integer number, of an arithmetic progression with a common difference of one provides information on the priority degree of the attribute, especially in the case of multi-criteria sorting. As an example, 1 for the most important sorting attribute, 2 for the second-most important attribute, etc.

The *weight* is the number of an arithmetic progression with a common difference of one, that provides information on the priority degree of an attribute when the system sorts the query result. The *weights* are integer numbers that start from one.

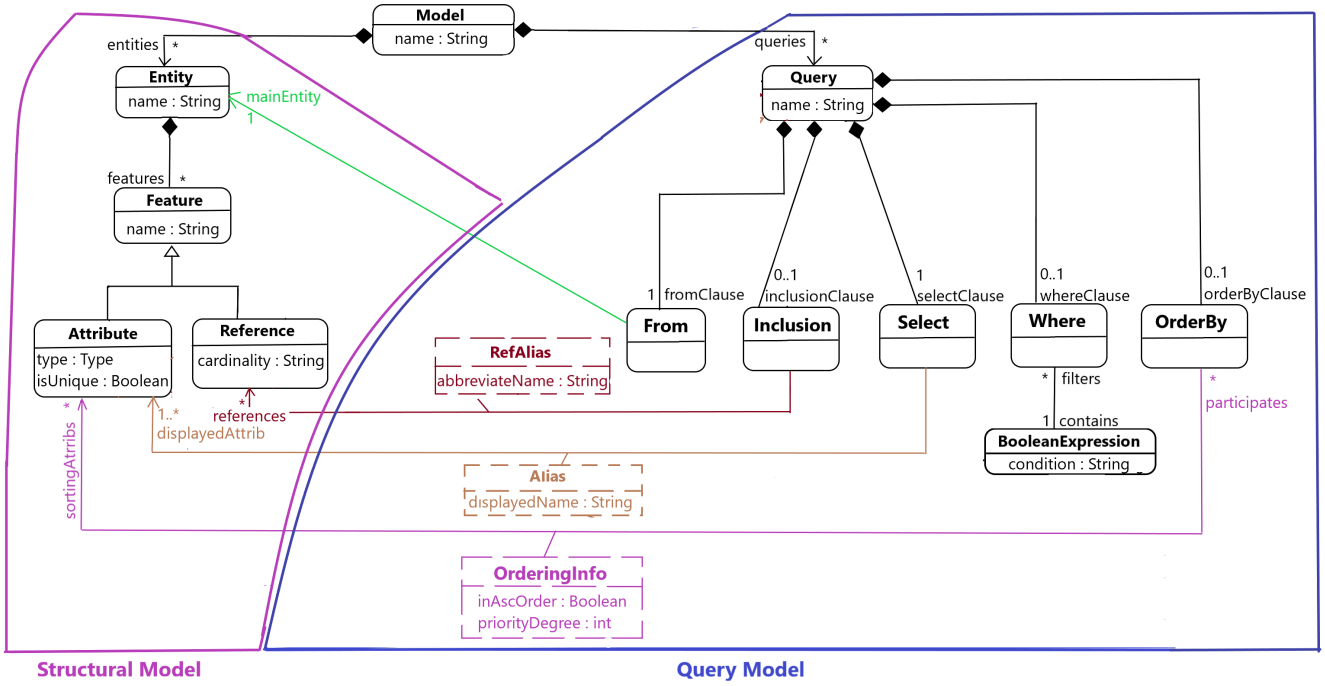

Figure 5. Components of the Platform-Independent Data Metamodel (PIDM): structural model and query model.

4.3. Running Example in the PIDM

This subsection shows how to represent the running example in the *PIDM* language. First, a textual syntax is created to instantiate models respecting the *PIDM*. Figure 6 shows how to express the entities *Passenger, Airport*, and *Flight* and the queries *Q1, Q2, Q3, Q4*, and *Q5* using this syntax. The definitions of the other entities and queries are consultable in an external repository [31].

In the example, the *entity* keyword is used to specify entities. Their attributes and references are written between braces. The *Flight* entity defines *code, departure Time and date, arrival Time* and *date, level* as attributes. A reference is specified with the *ref* keyword, followed by the referenced entity, the cardinality, and the reference name. For example, the statement *"ref Airport (1) origin"* in the *Flight* entity defines a reference named *origin* with a constant (1) cardinality, i.e., a *flight* has exactly one departure *airport*.

Queries in the *PIDM* are expressed in the textual notation using the list of clauses presented in Section 4.2, an SQL-like syntax. A query is specified by the *query* keyword followed by a *name*, which specifies the purpose of it. Then, a *SELECT* keyword is used to declare attributes to appear in the projection operation. After that, the *FROM* keyword indicates the *main entity*, and any other referenced entities are specified by the *INCLUDE* keyword.

Using the query *PassengersDepartingGivenCountry* as an example,

- The informations to display are: origin and destination city, departure time of flights, the data about their passengers (passport ID, first and last names, birthdate, sex, and nationality);
- The *main entity* is *Passenger*;
- The *Flight* and *Airport* entities are included;
- The query filters the results via a boolean expression based on *flight departure date* and *country* attributes. The expression contains two equality conditions combined by the *and* operator;
- Finally, using the *ORDERBY* clause, the query result will be sorted in ascending order of the attribute *departure city*, then in ascending order of the attribute *departure time flight*.

With this example, an instantiation of the *Platform-Independent Data Metamodel* is presented using a textual specification. The next subsections show the logical model for key-value databases and how *KVMod* can be used to build a physical model of a *Redis* database from a *PIDM* instance.

```
//entities
entity Passenger {
 id idPassport unique
 firstname text
 lastname text
 birthdate date
 sex text
 nationality text
 ref Flight[*] flights
 ref WebSiteAccess[*] accessedSites
}

entity Airport {
 id codeICAO unique
 nameAirport text
 city text
 country text
 ref Flight[*] landedflights
 ref Flight[*] departedflights
}

entity Flight {
 id code unique
 departureTime time
 arrivalTime time
 departureDate date
 arrivalDate date
 level int
 ref Localization[*] registeredPositions
 ref WebSiteAccess[*] sitesAccessed
 ref Passenger[*] registeredPersons
 ref Aircraft[1] usedAircraft
 ref Airport[1] origin
 ref Airport[1] destination
}
```

```
//queries
query Q1_aircraftsCapacityWithin:
SELECT registrationNumber,capacity
FROM Aircraft
WHERE capacity>="?" and capacity<="?"

query Q2_airportsGivenCountrySortedByCities:
SELECT nameAirport,CodeICAO,city
FROM Airport
WHERE country="?"
ORDER BY city ASC

query Q3_passengersOfGivenFlight:
SELECT firstName,lastName,idPassport
FROM Passenger
INCLUDE Passenger.flights as FL
WHERE FL.code="?"

query Q4_passengersDepartingGivenCountry:
SELECT Origin.city,Destination.city,FL.departureTime,
idPassport,firstName,lastName,birthdate,sex,nationality
FROM Passenger
INCLUDE Passenger.flights as FL,Passenger.flights.Origin as Origin,
Passenger.flights.Destination as Destination
WHERE FL.departureDate="?" and Origin.country="?"
ORDER BY Origin.city ASC,FL.departureTime ASC

query Q5_passengersDepartingGivenPeriod:
SELECT Origin.country,Origin.city,FL.departureTime,FL.code,
idPassport,firstName,lastName,birthdate,sex,nationality
FROM Passenger
INCLUDE Passenger.flights as FL,Passenger.flights.Origin as Origin
WHERE FL.departureDate="?"
ORDER BY Origin.country ASC,Origin.city ASC
```

Figure 6. PIDM textual notation that defines the Passenger, Airport, and Category entities and the queries Q1–Q5 of the running example of Section 2.1.

4.4. Key-Value Stores Metamodel

A logical metamodel has been defined to represent key-value family databases. It will be used to check the conformity of the generated logical model to the key-value paradigm. A logical model defines how the data should be implemented in a family, regardless of the DBMS. The proposed logical metamodel is shown in Figure 7 and is inspired by the model described in Section 2.2. As it appears,

- The key-value data model metaclass is the entry point to this metamodel;
- A *key-value data model* is considered a database schema that contains *collection* specifications;
- Each *collection* is a set of *associative arrays*;
- Identified by a unique key, an *associative array* is used to store a two-column matrix.
  - This matrix can be seen as a set of item pairs;
  - Each pair contains a field and a value.

By analogy to the relational world, a *collection* is a table, an *associative array* is a row table, a *field* is an attribute, and the *value* of a field is similar to the value of an attribute in RDBMS.

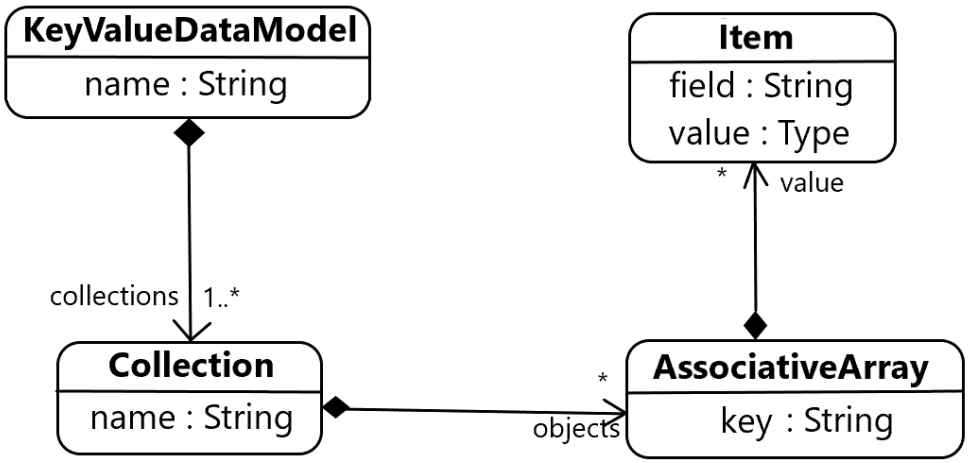

Figure 7. Metamodel for the logical modeling of key-value databases.

4.5. Transformations of a PIDM Instance to a Key-Value Logical Data Model

This section presents a series of mapping rules, based on the work of De La Vega et al. [12] and Ait Brahim et al. [13], for transforming a conceptual metamodel represented by a PIDM instance into a NoSQL key-value logical model via model-to-model transformations. The main idea behind these rules is the creation of a *collection* to support each access query. This idea respects the best practices and standards provided by key-value DBMS family systems. The strategy used to implement this idea is described in the following subsection.

4.5.1. Query to Collection Transformation

This transition is specified in Algorithm 1, working as follows: given an access query *AQ*, a new collection *C* would be created with the same name as *AQ* to support such a query. Initially, the fields of *C* are the projection attributes of *AQ* (i.e., the attributes that appear in the SELECT clause of *AQ*). Then, all the attributes involved in the query selection (i.e., the attributes that appear in the WHERE clause of *AQ*) should be extracted and added to the *collection* C. Moreover, if there are attributes included in the sorting criteria of the query (i.e., the attributes that appear in the ORDER BY clause of such a query), they will also appear in the fields of C. It will be necessary to ensure that each newly created field would have the same type as its related attribute in the *structural model*.

As an example, let us suppose we want to support the query *PassengersDeparting-GivenCountry* of the running example, presented in Figure 6. This query returns *passengers* departing from a *country* on a specific date, sorted by ascending order of departure *cities* and then by ascending order of *departure time*. *PassengersDepartingGivenCountry* is employed in the rest of the article to designate this query. For this query, a new *collection* named *PassengersDepartingGivenCountry* would be created.

- At the beginning, this collection would contain nine fields: the projection attributes *Origin.city, Destination.city, departureTime, idPassport, firstName, lastName, birthdate, sex*, and *nationality*;
- The selection attributes *FL.departureDate* and *Origin.country* must be added to the collection;
- The sorting criteria of the query (i.e., *Origin.city* and *FL.departureTime*) are also included at the beginning, and they must not appear twice (it is useless to put the same attribute more than once in a collection).

Finally, the key is a special field that provides functional dependency on the other fields of the *collection*. In key-value stores (or most of them) a key is not composite but it must be a single attribute. In this paper, the proposed solution is to always generate a key for each collection in order to identify the associative arrays of the collections in all possible cases. The collection *PassengersDepartingGivenCountry* would have a structure similar to Figure 8.

Algorithm 1: Query to collection transformation rule

```
/* Afterwards, D is a global variable that is accessible in all the
   algorithms.  D represents a data dictionary for a domain                */
GLOBAL VARIABLES D
Input: An access query AQ
Output: A collection C
C ← newCollection()
C.name ← AQ.name
C.id ← C.name + "_id"
/* The collection id is the concatenation of the collection name and the
   string ''_id''                                                          */
C.degree ← 1
/* AQ.projections is a dictionary that contains the names of displaying
   attributes (keys) and their aliases (values)                            */
foreach attr ∈ AQ.projections do
    f ← new Field()
    f.name ← attr.name
    f.type ← attr.type
    f.indexed ← False
    C.add(f)
    C.degree ← C.degree + 1
end
/* extractAttributes(AQ.conditions) is a function that browses the string
   parameter AQ.conditions, representing a selection clause of an access
   query.  The function returns a string array of the attributes used in
   the conditions                                                          */
/* C.fields is a string array of field objects in C. Each field is
   characterized by its name, type, and indexing                           */
foreach attr ∈ extractAttributes(AQ.conditions) do
    if attr ∉ C.fields then
        f ← new Field()
        f.name ← attr.name
        f.type ← attr.type
        f.indexed ← False
        C.add(f)
        C.degree ← C.degree + 1
    end
end
/* AQ.orderingAtrributes is a dictionary that contains the pairs (name of
   an attribute, sorting direction asc|desc)                               */
foreach attr ∈ AQ.orderingAttributes do
    if attr ∉ C.fields then
        f ← new Field()
        f.name ← attr.name
        f.type ← attr.type
        f.indexed ← True
        C.add(f)
        C.degree ← C.degree + 1
    else
        /* extractFieldsNames(C) is a function that constructs and returns a
           string array of fields from a collection C                      */
        F ← extractFieldsNames(C.fields)
        i ← F.index(attr)
        /* F.index(attr) returns the index of an attribute attr in the array
           F                                                               */
        ((C.fields)[i]).indexed ← True
    end
end
```

PassengersDepartingGivenCountry		
PassengersDepartingGivenCountry_id	numerical	**K**
Origin.city	text	↗
Destination.city	text	
FL.departureTime	time	↗
idPassport	text	
firstName	text	
lastName	text	
birthdate	date	
sex	text	
nationality	text	
FL.departureDate	date	
Origin.country	text	

Figure 8. The collection produced using Algorithm 1 to support the query *PassengersDepartingGivenCountry*.

The application of the previous algorithm is enough to generate *collections* that support the different queries. Nevertheless, using just this algorithm might result in non-optimized data designs due to redundant queries or excessive denormalization. The next subsection describes an optimization that helps solve this problem.

4.5.2. Query Merging

Let us say that we have two *collections* of a *key-value logical data model* that differ in a few fields but share many others. In this scenario, two separate *collections* might be constructed by using the first algorithm. It would be wise to merge them into a single *collection* whose fields are all the fields of both *collections*. Now suppose we have two *collections* that share the same fields but have different sorting criteria (i.e., indexed attributes). In this scenario, using the previous algorithm, two separate *collections* might be constructed with the same fields but with different indexed attributes. However, because the generated *collections* share the same fields and their indexed ones are different, they might be merged into one *collection* with the same fields; moreover, indexes would be created on all sorting criteria of either collection or both. The merging of two *collections* is performed using Algorithm 2, named the *collection merging algorithm*.

Figure 9 presents an example of the *collection merging*. As can be observed, the *collection* related to the query *PassengersDepartingGivenCountry* shows information on *passengers* departing from a given *country* on a given date. The *Collection* related to the query *passengersDepartingGivenPeriod* shows information on *passengers* departing from any *airport* in a given period. These *collections* share an important number of fields (90%+). Therefore, they can be merged into one *collection* named *PassengersDepartingGivenCountryPeriod*, which stores *passengers* leaving an *airport* with departure *city* and *country, destination city, flight code*, and the *flight date* and *time* indexed by the largest set of fields, i.e., *origin city* and *country, destination city*.

Algorithm 3, named as *collection schema optimization*, parses all the *collections* produced in the logical model, studies them, and, if necessary, reduces the number of *collections* based on Algorithm 2. The purpose is to avoid unnecessary data duplication in some fields, which will reduce the size of the database and slightly improve insertions, updates, and deletions.

Algorithm 2: Collection merging technique

Input: Two collections C_1,C_2
Output: A compacted collection C
$C \leftarrow new\ Collection()$
$C.name \leftarrow C_1.name + "\_" + C_2.name$
$C.id \leftarrow C.name + "\_id"$
foreach $f \in C_1.fields$ **do**
  $C.add(f)$
end
foreach $f \in C_2.fields$ **do**
  $Fnames \leftarrow extractFieldsNames(C)$
  if $f.name \notin Fnames$ **then**
    $C.add(f)$
    $C.degree \leftarrow C.degree + 1$
  else
    i=Fnames.index(f.name)
    if $f.indexed = True \wedge ((C.fields)[i]).indexed = False$ **then**
      $((C.fields)[i]).indexed \leftarrow True$
    end
  end
end

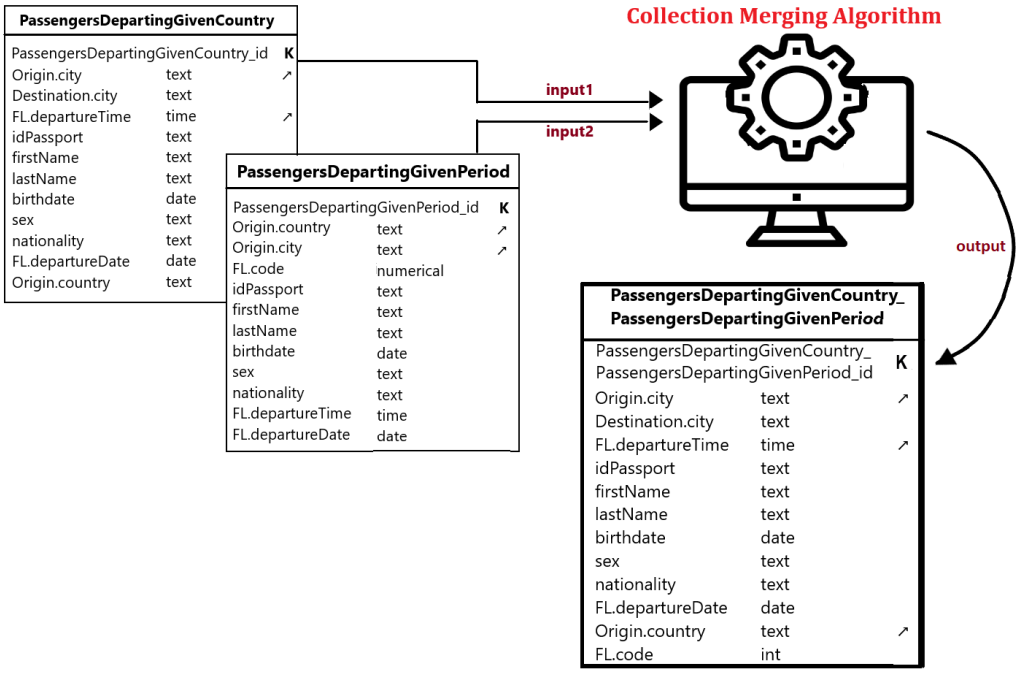

Figure 9. An example of collection merging using Algorithm 2.

Algorithm 3: Schema optimization method

Input: A set of collections CS
Output: A set of compacted collections CCS
$CCS \leftarrow CS$
foreach $C_i \in CS$ **do**
  if $C_i \notin CCS$ **then**
    // Collection previously compacted
    continue
  end
  foreach $C_j \in CCS - C_i$ **do**
    $C \leftarrow intersect(c_i, c_j)$
    /* The function intersect computes and returns a collection C which contains the common fields of two collections C_i and C_j. The function ensures that The identifiers of C_i and C_j don't do not appear in C. The name assigned to C is the name of C_i concatenated with the name of C_j. The id of C is its name concatenated with the string "_id" */
    $n_1, n_2 \leftarrow c_i.degree, c_j.degree$
    $n \leftarrow C.degree$
    if $n = 0$ **then**
      // ci and cj don't do not have common fields
      continue
    end
    if $\frac{n}{n_1} \geq 0.8 \wedge \frac{n}{n_2} \geq 0.8$ **then**
      // if the collections ci and cj share enough fields ($\geq$80%)
      $CCS \leftarrow CCS - C_j$
      $C_k \leftarrow merge(C_i, C_j)$
      /* From two given collections C_i and C_j, the function *merge* computes and returns a collection C_k which contains the common fields of C_i and C_j. The function ensures that The identifiers of C_i and C_j don't do not appear in C. The name assigned to C is the name of C_i concatenated with the name of C_j. The id of C is its name concatenated with the string "_id" */
      $CCS \leftarrow CCS \cup C_k$
    end
  end
end

4.6. Logical Model to Text Model Transformation

A model-to-text transformation can be used to convert the logical data model for key-value family databases into a physical database implementation. Key-value systems provide query languages for data definition and manipulation to define and manipulate data. In order to convert and obtain the code for the logical model into query languages, this model-to-text transformation must first translate each *collection* definition into its appropriate query counterpart corresponding language code. The definition of the *passengersDepartingGivenCountry* collection in *Redis* is shown in Figure 10.

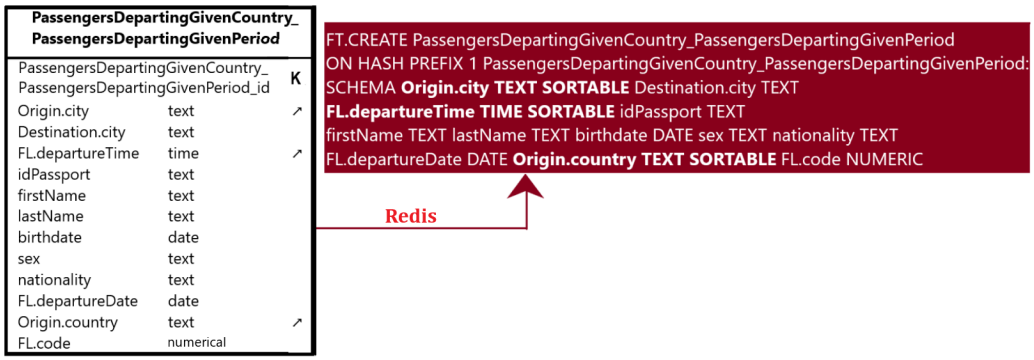

Figure 10. Logical to text transformation example in Redis.

A key-value system contains a key space able to store data as a set of pairs (key, value); it is not required to group such pairs into containers, but, in this study, we used HSet-based key-value databases, as shown in Figure 11.

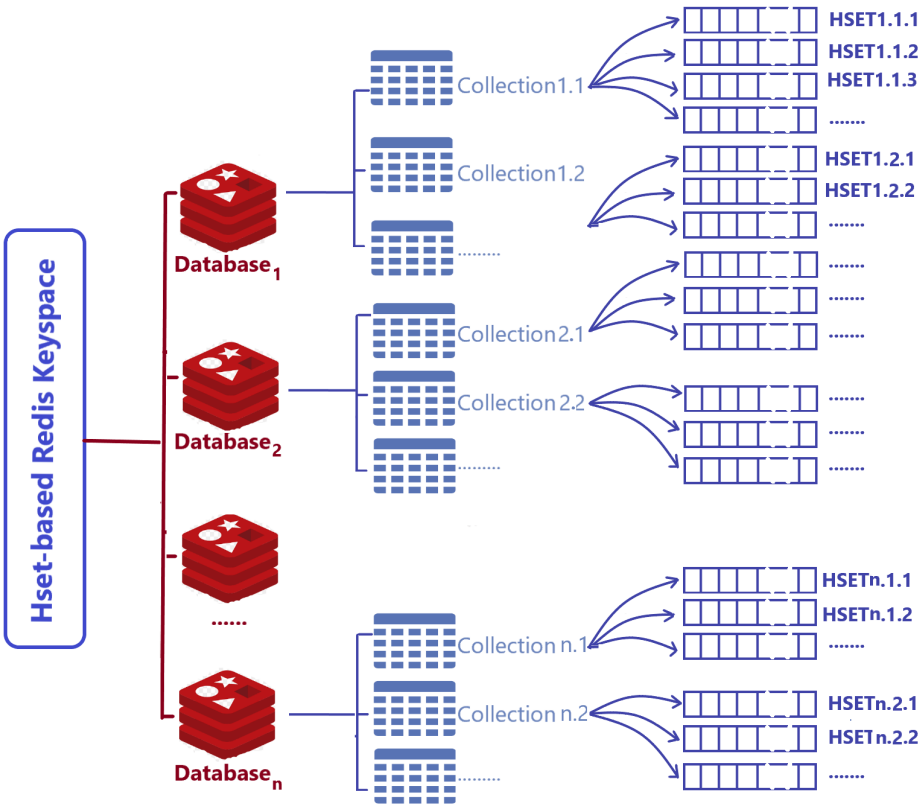

Figure 11. Redis concepts in HSet-based databases.

The proposal is to use some hierarchy of concepts:

- The key space of a key-value system can contain several databases;
- A database is composed of *collections*;
- A *collection* contains HSets representing the physical implementation of *associative arrays*;
- An HSet is a set of pairs <field,value> including a special field called named the HSet key identifying HSets within the system.

To create *collections* on a physical level, a *collection C* of a database *DB* can be named *DB_C*. This way would avoid the duplicates of *collection* names, for example, two *client*

collections, the first one belongs to *airflights* database and the second one belongs to *xbank* database. On the physical level of the targeted DBMS, the first collection is named *"airflights_client"*, the second one is named *"xbank_client"*. This situation rarely happens: for simplicity, it is considered in this paper that the name of a *C collection* at the logical level remains the same at the collection creation code on the physical level.

From another side, key uniqueness is essential. This rule is violated when a key has the same value across different *collections*. For example, if a key is used for *airflight* and *aircraft*, the value *2345* could be either the *airflight 2345* or *the aircraft 2345*. One way to avoid the clash of keys is to use a *prefix* that identifies the *collection* to which the key belongs. It is usual to use the *collection name* as a *prefix*. This way, the key *"airflight:2345"* would not conflict with the key *"aircraft:2345"*. Thus, an object that belongs to a *collection C* is named *C:k* if *k* represents the identifier key of the object.

In *Redis*, the schema predefinition of a *collection* makes the system able to perform complicated queries like multi-field queries and aggregation. A schema is created on *Redis* using the command *FT. CREATE*, which creates an index with the given specification [32]. *FT. CREATE* requires mostly the name of the index to create, the *prefix* used, which informs the engine about the keys it should index, the fields and their types, and likely their indexing as well.

Secondly, a *FT. SEARCH* command would be run [32] to retrieve the result of a query. It requires an index, which is invoked, and a predicate representing the search criteria in the query. Otherwise, a predicate is a boolean expression used to filter HSets, satisfying a condition within a *collection*.

Moreover, the command *HSET* is then used to insert data into *Redis* as an *associative array* [18]. In the running example, the Moroccan *airport n° 2* having with the code *"GMMX"*, the name *"Menara"*, and the city *"Marrakech"* are inserted using the command *HSET airflights_Airport:2 codeICAO "GMMX" nameAirport "Menara" city "Marrakech" country "Morocco"*.

In order to auto-generate keys when inserting new HSets, an idea is to first create an auto-increment integer key K_i for each C_i collection. The role of K_i is to store the key of the next new HSet to create. The transactions can provide a solution to ensure unique and auto-increment keys. Thus, whenever a new HSet of a collection C_i will be created, the following transaction should be invoked:

1. Obtain the value of the variable K_i;
2. Create a new HSet with the identifier key $C_i : K_i$;
3. Increment the variable K_i.

An example of a Redis transaction code to create a new HSet of a C_i collection is detailed below:

MULTI
x = GET K_i
HSET $C_i : K_i\ field_1\ val_1\ field_2\ val_2$ ***...***
x = x + 1
Set K_i ***= $x***
EXEC

5. KVMod's Implementation and Assessment

In this section, we show a software tool useful for key-value database design as presented in Section 4. We then evaluate the efficiency of the solution according to the proposed algorithms, and, afterward, we discuss their applicability to other systems in order to show how developers can deal with new key-value DBMS.

5.1. Implementation

A prototype of *KVMod* has been built to evaluate the described data-modeling methodology. This prototype is accessible for free in an external repository [33]. The next paragraphs depict the components of this repository.

The associated projects of the repository contain the metamodels described in Section 4 under the Ecore [34] format. The *PIDM* and *key-value metamodels* are included. Moreover, the projects include specifications for M2M and M2T transformations. Conventionally, languages such as ATL (Atlas Transformation Language) or ETL (Epsilon Transformation Language) are used to specify M2M transformations. These languages are suitable when each input element is transformed into one or more output elements. However, as explained in the transformation process, data structures and queries must be handled jointly when producing key-value models. For this, an imperative language was employed for the M2M transformation process. Xtend is selected, which is a Java-based language that provides advanced capabilities in model handling. In the case of M2T transformations, they are specified in EGL (Epsilon Generation Language) [35].

In order to manipulate *PIDM* instances, a textual Domain-Specific Language (DSL) [36] is also provided. This DSL language has been created with Xtext [37], which offers a configurable editor. Figure 12 shows a screenshot where the *airflights* case study is manipulated through the DSL editor. The left window shows the DSL syntax employed to define entities and queries over these entities. On the top right window, the related *PIDM* instance model of the processed *PIDM* file is shown. This instance will serve as an input for *KVMod's* transformation process. In the Properties view below, concrete element details from the model can be viewed, such as the *AttributeSelection* object selected in the figure. Finally, a project of examples is included, which contains *PIDM* specifications and NoSQL models associated, e.g., for the running example of this work.

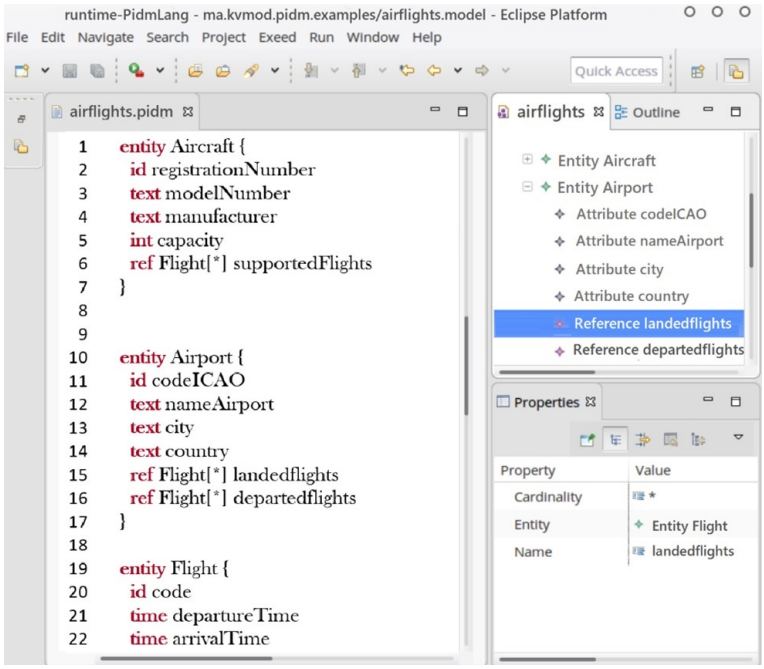

Figure 12. Editor of the PIDM textual DSL.

5.2. Assessment

To evaluate the efficiency, we took mainly the time complexity necessary to compute the design of a key-value database by varying each time the number of collections denoted m and the number of attributes denoted n.

5.2.1. Computational Resources

To design a key-value database, the modeler expresses entities of domain data and queries over entities in conformity with the *PIDM* metamodel. It is mainly a human task in which the machine is only used to input conceptual-level models. Then, the computer produces a key-value logical model from the conceptual one using the presented algorithms. Computation time is the most commonly used parameter to measure an algorithm's efficiency. This subsection examines the complexity of our algorithms to determine whether they can be run with a reasonable quantity of computation resources. For simplicity, big-O notation (i.e., asymptotic complexity) is used. O(1) for constant complexity, O(n) for linear complexity, etc.

Query to Collection Transformation Algorithm

Let T(n) be the time complexity to produce a *collection C* from a query *Q*, where *n* is the number of attributes in *Q* (*SELECT, WHERE* and *ORDERBY* attributes in *Q*). Operations outside loops are considered to have a constant cost. Loop bodies are also considered to be at a constant cost. The number of iterations of the algorithm must be estimated. This number varies between 3n (the worst case where any attribute in the query appears in all clauses, i.e., *SELECT, WHERE*, and *ORDERBY*) and *n* (best case where an attribute can not appear in more than one clause). Therefore, the query to *collection transformation algorithm* is O(n).

Collections Merging Algorithm

This algorithm is a function that receives two collections, C_1 and C_2, and merges them into a compact collection, *C*. Let n_1 be the number of fields in C_1, and n_2 be the number of fields in C_2. The first two instructions of the algorithm are considered to have a constant cost. The same holds true for the alternative structure (i.e., the if-else block), which forms the body of the inner loop. The number of loops iterations is in $O(n_1 \times n_2)$. If $n_1 \approx n_2$ the number of iterations is $O(n^2)$, where $n = \frac{n_1+n_2}{2}$. In this case, the complexity of the algorithm is in $O(n^2)$.

Schema Optimization Algorithm

Let m be the number of *collections*, and *N* be the total number of fields within the *collections*. Each *collection* contains, on average, n fields, where $n = \frac{N}{m}$. The algorithm iteration number is in $O(m^2)$, all the operations are in $O(1)$ except the operations $c < -intersect(c_i, c_j)$ and $c_k < -merge(c_i, c_j)$ which have an average complexity in $O(n^2)$. Consequently, the complexity of this algorithm is $O(n^2 \times m^2)$.

Conceptual Model to Key-Value Logical Model Transforming Process

For a conceptual schema of *N* attributes participating in *m* queries, the process starts by producing *m* collections in order to support access queries. This algorithm has a complexity of $O(m \times n)$, where *n* is the average number of fields in a collection ($n = \frac{N}{m}$). After that, Algorithm 3 is called to compact the generated *collections*. The complexity of this step is in $O(m^2 \times n^2)$. Finally, we can say that the global complexity of the process is in $O(m^2 \times n^2)$. Table 2 shows the process costs for different examples of *n* and *m*.

Table 2. Time complexity of conceptual to logical transformation process.

n \ m	**10**	**100**	**1000**
10	10^4	10^6	10^8
100	10^6	10^8	10^{10}
1000	10^8	10^{10}	10^{12}

To quantify the computational complexity described above, we estimate, for different database sizes, the generation times for the mapping rules from the conceptual to the logical model. A computer with a processor speed of 1 GHZ was used to illustrate the computation time. The Table 3 depicts the estimated results. The generation time is acceptable and may reach 17 min to complete for a big database that contains 1000 collections with 1000 fields on average in each collection.

Table 3. Computation time of conceptual to logical transformation process.

n \ m	10	100	1000
10	10 µs	1 ms	0.1 s
100	1 ms	0.1 s	10 s
1000	0.1 s	10 s	17 min

On the other side, a system with an Intel Core i7 processor and 8 GB of RAM running Windows 10 is used to experimentally verify the results. During experimentation, only system programs and the *KVDesign* tool are run on the system. For the data and the access queries, a program was developed to randomly generate a *PIDM* instance. Table 4 shows the observed values.

Table 4. Experimental results of a conceptual to logical transformation process.

n \ m	10	100	1000
10	180 µs	22 ms	3.7 s
100	35 ms	5.1 s	38 s
1000	19 s	95 s	26 min

5.2.2. Applicability to Other Systems

As the methodology is detailed for the *Redis* system, its structure facilitates its applicability to other key-value systems. By updating the code generator, new key-value products can be incorporated into *KVMod*. From a logical model, this generator would produce code for the new targeted system. For example, if we want to support the *Riak KV system* [38] as a target platform for *KVMod*, we will need to create a model-to-text transformation from the *key-value metamodel* of Figure 7 into code to define a database in conformity with the *Riak KV* features (e.g., data types and indexing methods). Moreover, if the targeted system had a set of specific characteristics, like the support or not of multi-field indexes, we might have to upgrade *KVMod* to support it.

6. Conclusions

In this article, *KVMod*, which is a rigorous key-value data modeling methodology, was presented. We established the fundamental key-value data modeling principles for *Redis* and defined mapping rules to switch from platform-independent conceptual models to Redis-specific model.

The current paper has powerful implications for practitioners and researchers, as presented below:

- The literature review shows that the design of the NoSQL databases can be useful to standardize access and understand its data storage;
- The combined MDA-based and query-driven methodology used in the current study holds several advantages for both researchers and practitioners. The use of MDA aids in automating the modeling process. The support of the access queries is in line with the best practices in database design in the NoSQL world;

- The proposal introduces a series of models at different levels in order to make the process enrichable, especially at the logical and physical levels;
- We described a robust data-modeling tool, named *KVDesign*, that automates some of the most time-consuming data-modeling tasks, including conceptual-to-logical transformations and code generation.

Despite the above-discussed contributions and advantages, the present work is not without its limitations:

- Firstly, the study supports only read queries, which are very important in a NoSQL context, but other operations like updates and insertions were not treated. For future research, we plan to extend our work to support all CRUD queries, including the aggregation operations.
- Secondly, key-value DBMS offers several data structures, including hashes, which are the only ones used in this work. We intend to study how to support other types, like sorted sets, to cover the maximum number of useful elements in the data design.
- On the other hand, a software KVMod-based tool was developed to design key-value databases. In the future, we plan to allow practitioners and designers to test it in different use cases and then collect user reviews in order to improve the design of key-value databases.
- Finally, due to the similarity of the DBMS of the same family, we plan to study database modeling in other key-value stores while benefiting especially from the conceptual and logical metamodels also introduced in our proposal.

Author Contributions: Conceptualization, M.H. and H.O.; Data curation, A.D.; Format analysis, H.O.; Investigation, M.H., A.D.; Methodology, M.H.; Project administration, H.O.; Resources, M.H. and A.D.; Software, M.H. and A.D.; Supervision, H.O.; Visualization, H.O.; Writing—original draft, A.D.; Writing—review and editing, H.O. and M.H. All authors have reviewed and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACID	Atomicity, Consistency, Isolation, and Durability
ATL	Atlas Transformation Language
CRUD	Create, Read, Update, and Delete
DBMS	Database Management System
DDL	Data Definition Language
DSL	Domain-Specific Language
EGL	Epsilon Generation Language
ERD	Entity Relationship Diagram
ETL	Epsilon Transformation Language
GPS	Global Positioning System
HSet	Hash Set
I.C.A.O	International Civil Aviation Organization
KVDesign	Design Tool for Key-Value Design
KVLM	Key-Value Logical Metamodel
KVMod	Key-Value Modeling
M2M	Model-to-model
M2T	Model-to-text
MDA	Model-Driven Architecture

NoSQL	Not Only SQL
OCL	Object Constraint Language
PIDM	Platform-Independent Data Metamodel
QVT	Query View Transform
RDBMS	Relational DBMS
SQL	Structured Query Language
UML	Unified Modeling Language
XML	Extensible Markup Language

References

1. Dourhri, A.; Hanine, M.; Ouahmane, H. A New Algorithm for Data Migration from a Relational to a NoSQL Oriented Column Database. In Proceedings of the International Conference on Smart City Applications (SCA21), Safranbolu, Turkey, 28 October 2021.
2. Akoka, J.; Comyn-Wattiau, I.; Laoufi, N. Research on Big Data—A systematic mapping study. *Comput. Stand. Interfaces* **2017**, *54*, 105–115. [CrossRef]
3. Corbellini, A.; Mateos, C.; Zunino, A.; Godoy, D.; Schiaffino, S. Persisting bigdata: The NoSQL landscape. *Inf. Syst.* **2017**, *63*, 1–23. [CrossRef]
4. Davoudian, A.; Chen, L.; Liu, L. A survey on NoSQL stores. *ACM Comput. Surv.* **2018**, *51*, 1–46. [CrossRef]
5. Diogo, M.; Cabral, B.; Bernardino, J. Consistency Models of NoSQL Databases. *Future Internet* **2019**, *11*, 43. [CrossRef]
6. Asadi, M.; Ramsin, R. MDA-Based Methodologies: An Analytical Survey. In Proceedings of the European Conference on Model Driven Architecture—Foundations and Applications, Berlin, Germany, 9–13 June 2008.
7. Atzeni, P. Data Modelling in the NoSQL world: A contradiction? In Proceedings of the 17th International Conference on Computer Systems and Technologies, Palermo, Italy, 23–24 June 2016.
8. Kaur, K.; Rani, R. Modeling and querying data in NoSQL databases. In Proceedings of the International Conference on Big Data (IEEE), Silicon Valley, CA, USA, 6–9 October 2013.
9. Chebotko, A.; Kashlev, A.; Lu, S. A Big Data Modeling Methodology for Apache Cassandra. In Proceedings of the IEEE International Congress on Big Data, Silicon Valley, CA, USA, 27 June–2 July 2015.
10. Roy-Hubara, N.; Rokach, L.; Shapira, B.; Shoval, P. Modeling Graph Database Schema. *IT Prof.* **2017**, *19*, 34–43. [CrossRef]
11. Hanine, M.; Lachgar, M.; Lachgar, S.; Elmahfoudi, O.; Boutkhoum, O. MDA Approach for Designing and Developing Data Warehouses: A Systematic Review & Proposal. *Int. J. Online Biomed. Eng.* **2021**, *17*, 99–110.
12. De la Vega, A.; García-Saiz, D.; Blanco, C.; Marta, Z.; Pablo, S. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Gener. Comput. Syst.* **2020**, *105*, 455–474. [CrossRef]
13. Abdelhedi, F.; Ait Brahim, A.; Atigui, F.; Zurfluh, G. MDA-Based Approach for NoSQL Databases Modelling. In Proceedings of the 19th International Conference on Big Data Analytics and Knowledge Discovery, Lyon, France, 28–31 August 2017.
14. Abdelhedi, F.; Ait Brahim, A.; Zurfluh, G. Applying a Model-Driven Approach for UML/OCL Constraints: Application to NoSQL Databases. In Proceedings of the Confederated International Conferences "On the Move to Meaningful Internet Systems", Rhodes, Greece, 21–25 October 2019.
15. Liu, S.; Rahman, M.R.; Skeirik, S.; Gupta, I.; Meseguer, J. Formal Modeling and Analysis of Cassandra in Maude. In *Formal Methods and Software Engineering. ICFEM 2014. Lecture Notes in Computer Science*; Merz, S., Pang, J., Eds.; Springer: Cham, Switzerland, 2014; Volume 8829. [CrossRef]
16. Neeru; Kaur, B. Cassandra vs. MySQL: Modelling and querying format. *IJCTA J.* **2016**, *9*, 5199–5206.
17. Shashank, T. *Professional NoSQL*, 1st ed.; Wrox: Birmingham, UK, 2011.
18. Carlson, J.L. *Redis in Action*, 1st ed.; Manning Publications: Greenwich, CT, USA, 2013.
19. Redis Official Documentation. Available online: https://redis.io/docs/manual (accessed on 28 April 2023).
20. Das, V. *Learning Redis*, 1st ed.; Packt Publishing: Birmingham, UK, 2015.
21. Gwendal, D.; Gerson, S.; Jordi, C.; Skeirik, S. UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases. In Proceedings of the 35th International Conference on Conceptual Modeling, Gifu, Japan, 14–17 November 2016.
22. Rossel, G.; Manna, A. A Modeling methodology for NoSQL Key-Value databases. *Database Syst. J.* **2017**, *8*, 12–18.
23. Li, C. Transforming relational database into HBase: A case study. In Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, Beijing, China, 16–18 July 2010.
24. Imam, A.A.; Basri, S.; Ahmad, R.; Watada, J.; Gonzlez-Aparicio, M.T.; Almomani, M.A. Data Modeling Guidelines for NoSQL Document-Store Databases. *Int. J. Adv. Comput. Sci. Appl.* **2018**, *9*, 544–555. [CrossRef]
25. de Lima, C.; dos Santos Mello, R. 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, New York, NY, USA, 11–13 December 2015.
26. Chen, P.P. The entity-relationship model—Toward a unified view of data. *ACM Trans. Database Syst.* **1976**, *9*, 9–36. [CrossRef]

27. Schroeder, R.; Duarte, D.; Mello, R.S. 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, Ho Chi Minh City, Vietnam, 5–7 December 2011.
28. Fernández Candel, C.; Sevilla, D.; García-Molina, J.; Chen, P.P. A unified metamodel for NoSQL and relational databases. *Inf. Syst.* **2021**, *104*, 101898. [CrossRef]
29. Martinez-Mosquer, D.; Lujan-Mora, S.; Navarrete, R.; Mayorga, T.C.; Herrera, H.; Rodrigo, V. An approach to Big Data Modeling for Key-Value NoSQL Databases. *RISTI—Rev. IbéRica Sist. E Tecnol. Informação* **2019**, *19*, 519–530.
30. Mior, M.; Salem, K.; Aboulnaga, A.; Liu, R. NoSE: Schema Design for NoSQL Applications. *IEEE Trans. Knowl. Data Eng.* **2017**, *29*, 2275–2289. [CrossRef]
31. Definition of the Running Example Queries and Entities. Available online: https://github.com/dourhriahmed/kvmod/blob/main/ma.kvmod.pidm.examples/airFlight.pidm (accessed on 14 July 2023).
32. Redis Search Module (Official Documentation). Available online: https://redis.io/docs/stack/search (accessed on 24 May 2022).
33. KVDesign Project. Available online: https://github.com/dourhriahmed/kvmod (accessed on 14 July 2023).
34. Steinberg, D.; Budinsky, F.; Paternostro, M.; Merks, E. *EMF: Eclipse Modeling Framework*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2009.
35. Rose, L.M.; Paige, R.F.; Kolovos, D.S.; Polack, F.A. The Epsilon Generation Language. In Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications, Berlin, Germany, 9–13 June 2008.
36. Kleppe, A. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2008.
37. Eysholdt, M.; Behrens, H. Xtext: Implement Your Language Faster than the Quick and DirtyWay. In Proceedings of the 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, Reno/Tahoe, NV, USA, 17–21 October 2010; pp. 307–309.
38. Riak Key-Value System Official Documentation. Available online: https://riak.com/products/riak-kv (accessed on 12 December 2022).