Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 10, No. 02, June 2024.

A MODEL DRIVEN FRAMEWORK FOR COLLABORATIVE AND DYNAMIC DESIGN AND IMPLEMENTATION OF NOSQL-ORIENTED DATA WAREHOUSES

Khadija Letrache and Mohammed Ramdani

(Received: 10-Feb.-2024, Revised: 12-Apr.-2024, Accepted: 27-Apr.-2024)

ABSTRACT

Nowadays, modernizing the data warehouse ecosystem is a key challenge in decision-support systems. This modernization is crucial for ensuring scalability and meeting evolving business requirements, especially with the advent of big data. A promising solution involves implementing data warehouses with contemporary data stores, such as NoSQL. In this context, we introduce in this paper a framework that leverages Model-driven Architecture (MDA) to design and implement modern data warehouses across NoSQL data stores. Our MDA approach aims to offer a collaborative, dynamic and reusable process for developing NoSQL-oriented data warehouses tailored to specific project requirements. It facilitates the automatic and dynamic generation of a hybrid data-warehouse model from its conceptual model, which encompasses structural, domain and access parameters. Moreover, our framework includes the generation of implementation code for the data warehouse, along with a set of files to validate, document and illustrate the data-warehouse schema on a target platform. Finally, we present a detailed case study to highlight the effectiveness of our MDA framework.

KEYWORDS

Data warehouse, Model driven architecture, Metamodel, Dynamic transformation rule, NoSQL, Document, Key-value, Column-Family, Graph.

1. INTRODUCTION

A few years ago, traditional data warehouses were implemented on relational systems and utilized for analyzing operational data derived from relational databases [1]. However, with the advent of big data, numerous voices proclaimed the end of the data-warehousing era, asserting that such systems had become obsolete [2]-[3]. Meanwhile, according to a survey conducted by TDWI [4], it was found that a significant majority of enterprises, approximately 75%, continue to utilize on-premises or cloud-based data warehouses. The survey also revealed that more than a half of these enterprises have transitioned from analyzing only traditional structured data to exploring new types of data. This shift introduced a new generation of data-warehousing systems, deploying a new data stack that covers the entire decision-support ecosystem, from data storage to data visualization. This process is commonly known as data-warehouse modernization or data-warehouse augmentation [5].

Consequently, the focus for Business Intelligence (BI) developers often shifts towards mastering various technologies rather than addressing the analysis of actual requirements. In this context, we introduce a collaborative framework in this paper that enables the automatic generation of data-warehouse models across different NoSQL data stores. Our approach leverages the power of the Model-driven Architecture (MDA) paradigm [6] to automate the process of modeling and implementing data warehouses on selected platforms. The objective of the proposed framework is to guide BI developers in constructing their data warehouses on any desired platform, even with limited expertise in that particular platform. Furthermore, automation allows developers to seamlessly incorporate best practices from previous designs into future projects, enabling them to leverage valuable feedback and accumulate essential knowledge.

The proposed framework employs dynamic transformation rules to generate a data -mart model that is driven by the project's requirements. Traditionally, in classical MDA approaches, users define static transformation rules to automatically generate a specific model, either star or snowflake schema and its implementation code. However, when applied to NoSQL-oriented data warehouses, the obtained code provides limited metadata about the generated model. In contrast, our approach maximizes the

K. Letrache and M. Ramdani are with Informatics Department, LIM Laboratory, Faculty of Science and Techniques of Mohammedia, University Hassan II, Casablanca, Morocco Emails: khadija.letrache@fstm.ac.ma and mohammed.ramdani@fstm.ac.ma

utility of MDA by utilizing it as a channel for conveying design and tuning practices through predefined transformation rules. Consequently, the model generated by our framework is recommended based on the project's specific requirements, ensuring a more tailored and efficient design process.

To achieve this objective, this paper introduces four metamodels designed to effectively represent the target data stores: document, column-family, key-value and graph stores. Furthermore, we propose an extended conceptual metamodel that encompasses all the essential aspects required during the data warehouse-design process. Additionally, we introduce dynamic transformation rules that automatically derive destination models from the conceptual model.

Leveraging MDA model-to-text transformation rules, we then automatically generate the DW implementation code and documentation. These outputs are presented in three distinct files: The first contains the DW implementation code for a representative platform, which, in the case of NoSQL stores, offers limited metadata due to their schema-less nature. The second file is a data template, generated in JSON format, to provide guidance on data organization and storage. Lastly, a validation file is included to offer additional metadata and aid in verifying and validating the integrity of the DW data during the loading phase.

The remainder of this paper is organized as follows: Section 2 discusses the most relevant works related to NoSQL-oriented data warehouses. Section 3 outlines our approach and introduces the proposed conceptual metamodel, providing a background and context for our proposal. In Section 4, we delve into the metamodel and transformation rules for document-oriented data warehouses, while Section 5 focuses on key-value oriented DWs. The discussion on column-family DWs is presented in Section 6 and graph- oriented DWs are examined in Section 7. A case study that illustrates our approach is detailed in Section 8. In Section 9, we describe the generated code and documentation files and the used transformation rules. Finally, Section 10 concludes the paper, summarizing our findings and proposing directions for future research and perspectives.

2. RELATED WORKS

The primary objective of a data warehouse is to effectively store enterprise data, enabling data analysis and decision-making. Traditionally, DWs have been implemented using relational database management systems (RDBMS) as the foundational layer for data storage [1]. However, with the emergence of NoSQL databases, there has been a growing suggestion to utilize these data stores for the data-warehousing systems. In [7], the authors Chevalier et al. have studied the transition from a data warehouse conceptual model to NoSQL logical model; namely, column-family (CF) and document oriented stores. The authors provide the outline rules to model a data warehouse on document or column-family oriented stores. The same authors have proposed in [8] the implementation of OLAP cuboids in a document oriented data store designed as flat and shattered models and using materialized views. Boussahoua et al. [9] conducted a comprehensive study on implementing data warehouses in column-family oriented stores. The authors employed a k-means clustering method to effectively identify the necessary column families to group attributes. Other studies have investigated the implementation of data warehouses in graph- oriented databases, including the approaches proposed by Sellami [10] and Vaisman [11]. Additionally, Benhissen et al. [12] employed a shattered model, utilizing a distinct node for each attribute.

On the other hand, there has been research dealing with the development and automation of NoSQL-oriented data warehouses through model-driven approaches. In [13], the authors proposed an MDA approach for generating data warehouses in the four NoSQL data stores. The authors proposed a single metamodel that encompasses the basic concepts of document, column-family, key-value and graph rather than data-warehousing concepts. The proposed metamodel contains a "Value" class in all stores which is not a concept related to metamodeling. The authors also proposed four metamodels for specific database systems in each store type. These latter are generated from the generic PSM. The authors illustrated and provided transformation rules to obtain a column-family PSM from a relational PSM. In another related work focusing on Cassandra, the author proposed in [14] an approach for generating a data warehouse model within Cassandra, based on an information model that represents the DW conceptual model, but does not use data-warehousing concepts. Additionally, Yangui et al. [15] put forward an approach for mapping data from a multidimensional model of document and column-family

oriented databases as a flat model in both cases. More recently, Oukhouya et al. [16] proposed an MDA approach for DW design using a generic PIM (Platform Independent Model) metamodel for both NoSQL and relational platforms. The proposed metamodels present some drawbacks, such as the association between measures and dimensions, the cardinality of the primary key in the relational model or the generalization between identifiers, atomic fields and documents in the MongoDB metamodel. In a recent work, [17] deals with key-value oriented DWs using an MDA approach. Finally, another work proposed by Abdelhedi et al. [18] aims to create document-oriented data warehouses from relational data lakes using an MDA approach. However, the proposed transformation rules are related to models rather than metamodels, transforming data records into documents, for instance.

By analyzing the aforementioned works and their contributions to the modernization of the data warehouse ecosystem, we notice that all approaches are based on static transformation rules, where the user must independently choose a specific target model driven solely by the description of multidimensional concepts. Thus, our approach aims to guide users in designing a DW model tailored to each specific project, based on dynamic transformation rules. These rules are implemented through a collaborative process, driven by environmental parameters and developers' feedback. The objective of our approach is to leverage the model-driven paradigm to create a collaborative framework that facilitates the sharing and capitalization of developers' knowledge. Table 1 summarizes the state-of-the-art and outlines our contribution according to the following criteria: whether it is an MDA approach, the type of proposed transformation rules (static *vs.* dynamic, manual *vs.* automatic), the source model of the transformations, the target NoSQL store and model (flat using a single "table", star, snowflake, hybrid which is a combination of different models) and finally, the nature of the code generated by the proposed approach.

Table 1. Comparative analysis of related works on NoSQL-oriented data warehouses.

Paper	MDA	Trans. Rules	Source Model	Target NoSQL Store	Target Model	Generated code
[7]-[8]	Yes	Static	Multidimensional	CF/Document	Flat model/Star Model	-
[9]	No	Dynamic - Automatic (k-means)	-	CF	Flat	-
[10]	No	Static - Manual	Multidimensional	Graph	Snowflake	-
[12]	No	Static - Manual	Multidimensional	Graph	Shattered	-
[11]	No	-	Multidimensional	Graph	Star-Snowflake	-
[13]	Yes	Static - Automatic	Relational PSM	Document - CF - Key value- Graph	Generic NoSQL	-
[14]	Yes	Static	Information Model	CF	Flat	Impl. code
[15]	Yes	Static	Multidimensional	Document - CF	Flat	-
[16]	Yes	Static - Automatic	Generic	Relational - Document - CF	Star (re lational, document) - Flat (CF)	Impl. code
[17]	Yes	Static - Automatic	Relational	Key value	Flat	-
[18]	Yes	Static	Relational	Document	-	Data file
Our Approach	Yes	Dynamic - Automatic	Extended Multidimensional	Document - CF - Key value- Graph	Hybrid (with vertical partitioning)	Impl. code- Validation file - Data template

3. OUR MDA APPROACH

3.1 Approach Description

As a reminder, the MDA is a modeling and automation approach based on metamodels. The MDA architecture distinguishes between three abstraction levels in any software application:

1. Computation-independent Model (CIM): This level captures the user requirements and high-level specifications of the application.
2. Platform-independent Model (PIM): This level represents the conceptual model of the application, independently of any specific target platform.
3. Platform-specific Model (PSM): This level represents the application's design tailored to a specific target platform.

The transition between these levels is performed automatically through the use of model-to-model transformation rules expressed using a transformation language, such as Atlas Transformation Language (ATL) [19] and Query/View/Transformation (QVT) [6]. Moreover, the MDA allows the generation of the implementation code from the PSM model through model-to-text transformation rules. The MDA offers numerous advantages, such as automation, communication and interoperability, while reducing development time and costs [20].

In this work, we employed the MDA paradigm to design a framework for data-warehouse modeling and implementation across diverse NoSQL stores, as illustrated in Figure 1. The objective of our approach is not only to automatically derive the data warehouse PSM model from the PIM model, but also to generate a model with a recommended design and configuration defined regarding developers' feedback and data stores' recommendations. Therefore, instead of choosing a predefined target model, flat, star or snowflake, our collaborative approach aims to provide the user with a suitable hybrid schema based on the defined project parameters. In this context, the transformation rules are implemented with a focus on the following aspects:

- Depending on the target platform, determining when to use normalization, denormalization and/or vertical partitioning.
- Identifying which project parameters should guide the DW modeling to obtain a tailored and efficient model.
- Determining which configuration parameters are crucial for the model's performance and that must be communicated through the transformation rules.

In fact, besides the DW structural model, our transformation rules address three main aspects: sharding, distribution and vertical partitioning. Indeed, in NoSQL systems, defining the appropriate sharding and distribution configuration is a crucial design consideration. Vertical partitioning is also an important design aspect in NoSQL, where splitting a "table" can provide better performance compared to using a single table. It's important to note that replication parameters are not included in our approach, as these are determined at the DW level, that is, for the database as a whole rather than for specific data marts. Thereby, we extended the PIM metamodel described in [21] to include essential project parameters needed to generate a tailored model of a data mart. This dynamic design enables us to formulate a transformation rule as follows:

$$S_p + \text{Dynamic Transformation rule } (p) = D_p \qquad (1)$$

where S is the source model (PIM), p is a set of project's parameters and D_p is a fitted destination model (PSM) in terms of p.

However, to ensure coherent resulting models without missing or redundant elements, dynamic transformation rules must carry out the following constraints:

- Completeness: During the execution of transformation rules, all the necessary DW components and attributes are generated to avoid any omissions.
- Disjoint: During the execution of transformation rules, no DW components or attributes are generated more than once, thus avoiding redundancy.

It is essential to note that the objective of this paper is not to define specific design patterns, as these should be the outcome of numerous experiments tailored to each project's specific architecture and data characteristics.

Finally, from the obtained PSM, we generate three DW files using model-to-text transformation rules. The first one is the implementation code which in case of NoSQL databases because of their schemaless aspect [22] does not include all DW metadata. Therefore, our framework generates two additional files: data template and a schema validation file, the purpose of which is to enhance developers' comprehension of the derived model and to provide them with directives for the loading phase.

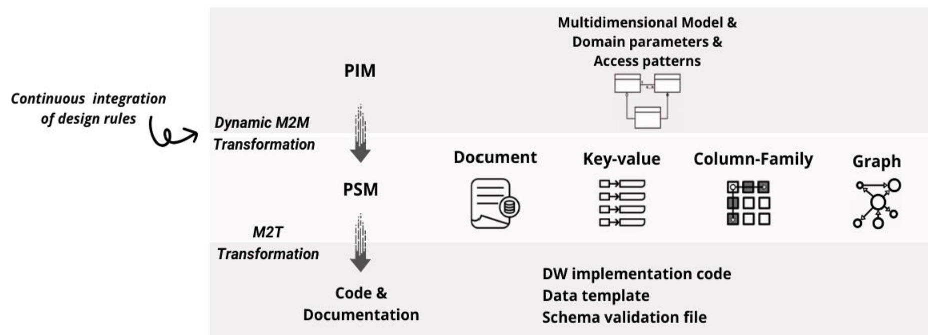

Figure 1. Our collaborative model-driven framework for NoSQL-oriented data warehouses, generated from the PIM model using collaborative dynamic transformation rules that are continuously enhanced with new design practices.

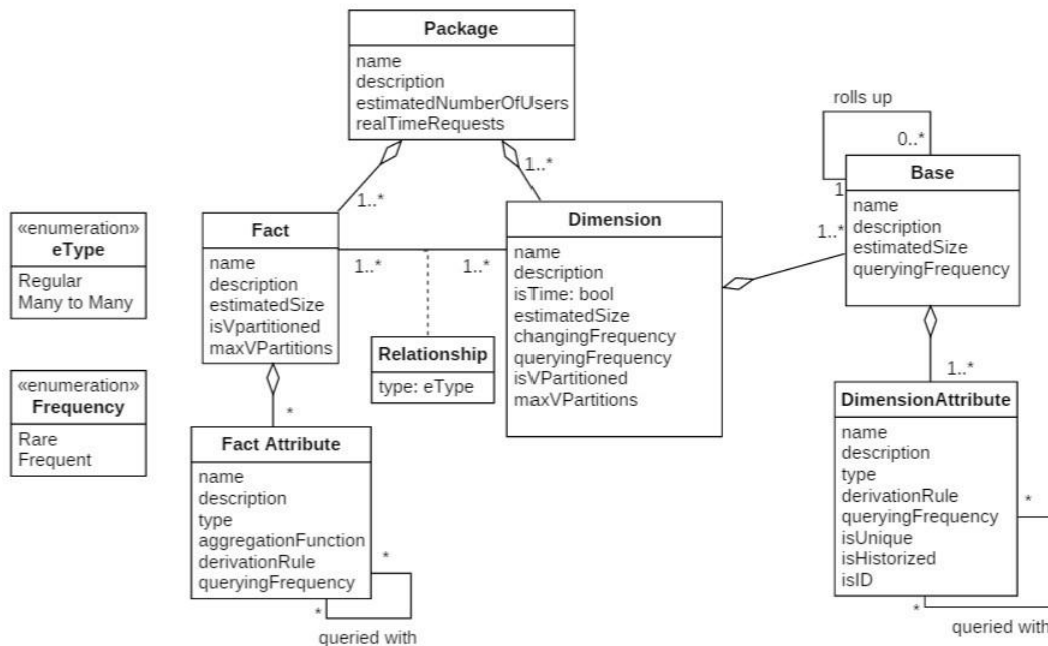

Figure 2. Our extended multidimensional metamodel (PIM) comprising domain and access parameters.

3.2 PIM Metamodel

To represent the data-warehouse conceptual model (PIM), we propose an extended multidimensional metamodel, as illustrated in Figure 2. This metamodel describes the data-warehouse structure and all aspects of the environment necessary for its design. Within this metamodel, the data warehouse is depicted as a package that includes facts and dimensions. Facts contain attributes (measures) characterized by a name, type and aggregation function. Furthermore, facts are linked to their related dimensions through relationships (either one-to-many or many-to-many relatioships). Additionally, dimensions are composed of base elements that represent hierarchy levels.

Besides the multidimensional aspects, our proposed metamodel is further extended to encompass domain parameters and access patterns, which play a crucial role in guiding data-warehouse design, especially within NoSQL platforms. This adaptation of the metamodel enables the definition of dynamic transformation rules based on these parameters. Consequently, our conceptual metamodel incorporates domain parameters related to data-size estimation, the number of users and a realTimeRequests boolean. For facts, in addition to estimated data size, attributes, such as isVpartitioned and maxPartitions are included. These allow users to specify whether facts could be vertically partitioned and to define the maximum number of partitions allowed. Furthermore, we have enhanced fact attributes by adding querying frequency attributes and the "queried with" association to capture access patterns among fact attributes. In regard to dimensions, similar to facts and fact attributes, we have incorporated attributes for estimated data size, is Vpartitioned, maxPartitions and the querying frequency of the dimension. Additionally, we introduced the changing frequency attribute for dimensions, which influences decisions related to dimension design. At the dimension attributes' level, we introduced is Historized to indicate decisions that must be made during the conception phase, impacting the attribute physical design. Attributes is Unique and is ID were also added to further define dimension characteristics. To capture access patterns between dimension attributes, the association "queried with" has been added. It is important to highlight that the environment parameters can be personalized and enriched by BI developers according to their specific needs.

4. DOCUMENT-ORIENTED DATA WAREHOUSE

4.1 Document-oriented Databases Metamodel

A document-oriented database consists of a collection of records stored in formats such as JSON, BSON, XML or YAML. Each record, referred to as a document, comprises key-value pairs Key:Value. The schema-less nature of these databases allows records within the same collection to possess different attributes, prompting us to represent a collection by a FieldSet (Figure 3). A FieldSet is defined as a group of documents that share the same fields. Each field, also known as a property, is characterized by a type (e.g. string, integer, array, list, …etc.), a description, a derivation rule and an isHistorized attribute. Additionally, a FieldSet may include other FieldSets, known as embedded documents. Collections in some document databases may have an optional identifier. These collections can also contain reference keys, similar to foreign keys in a relational model, which reference other collections using their URIs. It is possible to designate certain fields within a collection as required, ensuring their presence. Moreover, to improve model performance, parameters relating to sharding and distribution can be specified at the collection level. In our metamodel, these aspects are represented as dictionaries, allowing for the inclusion of numerous parameters as needed.

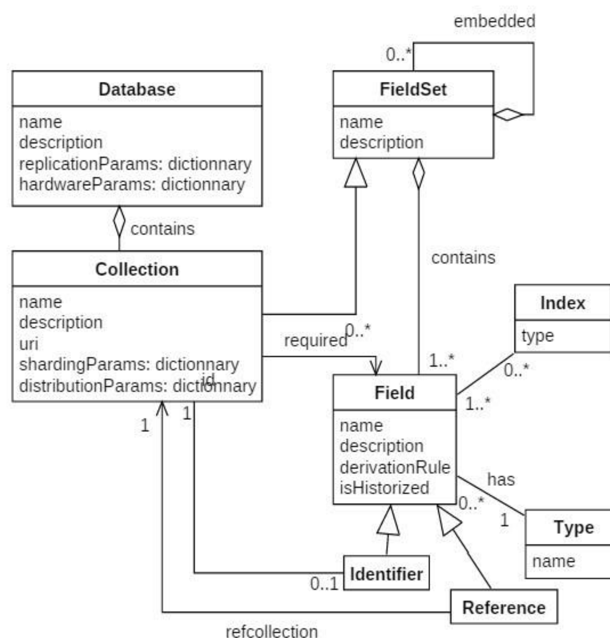

Figure 3. Our proposed document-oriented metamodel.

4.2 From PIM to Document-oriented PSM

In a document-oriented data warehouse, the transformation rules aim to determine the collections to be created for facts and dimensions within the data warehouse. They also dictate how data is distributed across these collections and define which attributes should be embedded or referenced. The configuration of these parameters is a critical task. It must be based on a combination of factors related to domain-specific requirements and access patterns, in addition to the experience and feedback of business-intelligence developers.

Basically, three fundamental models serve as a foundation for document-oriented data warehouses:

- Flat Model: In this model, a single collection is dedicated for each fact with its associated dimensions. Within this collection, all attributes of the fact are represented as fields, while the dimensions are incorporated as embedded documents.
- Star Model: In this model, an individual collection is created for each fact and for each dimension. Each dimension is then referenced within the corresponding fact collection. Hierarchies are embedded into the root or terminal dimension.
- Snowflake Model: This model employs separate collections to store facts, dimensions and hierarchy levels, joined through references.

In our approach, in addition to these classical models, we propose a hybrid model that combines embedded, referenced and vertically partitioned documents. This model is driven by project parameters defined in the PIM model through automatic and dynamic transformation rules. Below, we present possible mappings for each DW element.

- Fact F: Each fact F, defined by a set of fact attributes, can be mapped to a single collection CF with all the fact attributes represented as fields. Alternatively, it can be mapped to a set of collections, each containing a sub-set of the fact attributes and sharing the same dimensions. (Figure 4) illustrates an example of a dynamic rule for mapping a fact table to either a single collection or two collections. This decision is based on the fact table's estimated size, the querying frequency of its attributes and whether vertical partitioning of the facts is permitted by the user. It should be noted that the division of a fact into multiple collections can be executed based on more complex formulae. For example, using the "queried with" attribute allows for grouping fact attributes that are commonly queried together into the same collection.
- Regular Dimension: Each dimension *D*, defined by a set of dimension attributes, can be mapped to an embedded document within the fact collection, to a single collection C_d or to a set of collections.

```
rule Fact2Collection1{
    from f:PIM!Fact (f.estimatedSize < thisModule.FactSizeThreshold or f.isVpartitioned=false)
    to c:Doc!DocCollection (
    Name<- f.Name,
    Identifier <- thisModule.Fact2ID(f),
    shardingParams <- '(sharding,disabled)',
    contains <- c.Identifier,
    contains <- f.Attributes-> asSequence() -> collect(e|thisModule.FA2Field(e)),
    required <- c.contains,
    contains<-Doc!Reference.allInstances(),
    Embedded <-Doc!FieldSet.allInstances()->select (e|e.Description='Embedded Dimension')
    )
}
rule Fact2Collection2{
    from f:PIM!Fact (f.estimatedSize >= thisModule.FactSizeThreshold and f.isVpartitioned=true)
    to c1:Doc!DocCollection (
    Name<- f.Name,
    Identifier <- thisModule.Fact2ID(f),
    shardingParams <- '(sharding,enabled), (shardingKey, date)',
    contains <- c1.Identifier,
    contains <- f.Attributes-> asSequence() ->select (fa|fa.queryingFrequency='frequent') -> collect(e|thisModule.FA2Field(e)),
    required <- c1.contains,
    contains<-Doc!Reference.allInstances(),
    Embedded <-Doc!FieldSet.allInstances()->select (e|e.Description='Embedded Dimension')
    ),
    c2:Doc!DocCollection (
    Name<- f.Name,
    Identifier <- thisModule.Fact2ID(f),
    shardingParams <- '(sharding,enabled), (shardingKey, date)',
    contains <- c2.Identifier,
    contains <- f.Attributes-> asSequence() ->select (fa|fa.queryingFrequency='rare') -> collect(e|thisModule.FA2Field(e)),
    required <- c2.contains,
    contains<-Doc!Reference.allInstances(),
    Embedded <-Doc!FieldSet.allInstances()->select (e|e.Description='Embedded Dimension')
    )
}
```

Figure 4. ATL dynamic transformation rule for mapping a fact to collections based on its estimated size and the querying frequency of its attributes.

- Hierarchy: Each hierarchy level *L* within dimension *D* is mapped according to the mapping of *D*. If *D* is mapped as embedded, then L is also mapped as embedded. Otherwise, *L* can be mapped to an embedded document E_h within collection C_d or to a separate collection C_h.
- Many-to-many dimension: In traditional data warehouses, this kind of dimension is linked to the fact using a degenerate fact [21]. However, in the context of NoSQL data stores, which support a wide range of data types, the many-to-many dimensions D m2m are mapped to a field in the fact collection. This field utilizes collection data types, such as sets or lists, to store the dimension if it has only a single attribute. In cases where the dimension possesses multiple attributes, a distinct collection Cm2m is created to accommodate the dimension's attributes and a collection field is added to the fact to store references to this dimension's collection.

5. KEY VALUE-ORIENTED DATA WAREHOUSE

5.1 Key-value-oriented Databases Metamodel

Key-value stores, akin to hash tables [23]-[24], function by mapping values to specific keys. These values can range from primitive data types like integers and strings to more complex ones like lists, blobs and JSON documents. A distinguishing characteristic of key-value databases is their schema-less nature, which allows users to dynamically add or remove values (fields) at any time. This results in records that may possess different attributes within the same table.

To represent this flexible data model, we propose the following metamodel (Figure 5). The principal component in this metamodel is the table, which has a primary key, also known as a partition key and a set of attributes. Additionally, most key-value data stores offer the ability to define multiple indices on attributes, known as secondary, sort or local keys. These indices play a crucial role in enabling efficient data querying. Due to the flexibility of the key-value model, we utilize KeyValueSets to group key-value pairs that share the same attributes. Although the KeyValueSet concept is not employed during the implementation phase, it helps developers understand how data can conceptually be organized.

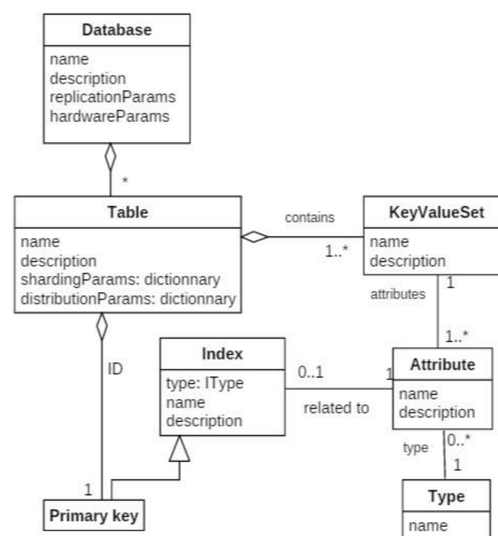

Figure 5. Our proposed key-value oriented metamodel.

5.2 From PIM to Key-value Oriented PSM

In the context of key-value oriented data warehouse and considering the lack of joins in most key-value stores, we adopt in this paper a flat model using a single table. Under this model, each fact is associated to a single table with all its corresponding dimensions. The table has a primary key formed by the concatenation of all dimensions keys, to allow and facilitate data querying. All attributes of the fact and dimensions are mapped to attributes within that single table. The same solution is also applicable for the many-to-many dimensions, but using collection data type. Additionally, an index is created for each dimension attribute required for querying data as illustrated by the (Figure 6). It is important to note that

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 10, No. 02, June 2024.

defining indices is a design consideration requiring knowledge and expertise in the chosen database system and environment. Such expertise can be communicated through dynamic transformation rules.

```
rule Dim2KV{
    from d:PIM!Dimension
    to k:KV!KeyValueSet(
    Name<-d.Name,
    Description <- d.Description,
    composed <- d.hierarchy.contains_Att -> asSequence() -> collect(e|thisModule.DA2Attribute(e))
    )
}
lazy rule DA2Attribute{
    from da:PIM!DimensionAttribute, desc:PIM!Descriptor, oid:PIM!OID
    to f:KV!Attribute(
    Name<- da.Name,
    Type <- da.Type,
    Description <-  da.Description,
    Index <- if da.queryingFrequency='frequent' then da.thisModule.DA2Index(da) else '' endif
    )
}
lazy rule DA2Index{
    from a:PIM!DimensionAttribute
    to I: KV!Index (
    Name<-'I_'+a.Name,
    Type <-'Secondary'
    )
}
```

Figure 6. ATL dynamic transformation rule for mapping a dimension to a KeyValueSet and dimension attributes to attributes and generating an index for each frequently accessed attribute.

6. COLUMN-FAMILY ORIENTED DATA WAREHOUSE

6.1 Column-family Oriented Database Metamodel

NoSQL column or column-family stores, also referred as wide-column stores [25], closely resemble key-value stores [23], with data stored as keys mapped to values and rows or records having varying attributes. However, in column-family stores, data can be grouped into column families [24], where each column family represents a specific map of data [25]. Thus, to represent the column-family oriented data model, we propose the metamodel depicted in (Figure 7). In this metamodel, a database (also referred to as a keyspace in some data stores) is composed of tables. Each table is made up of columns that can be organized into column families translated by the 0,1 multiplicity. It is important to note that in some NoSQL column-family stores, like Cassandra, a column family is synonymous with a table. Our proposed metamodel accommodates all scenarios. Moreover, in some column-family stores, column families can be further grouped by super columns, which we represent in our metamodel with a reflexive association between column families. Additionally, in most column-family stores, each column is assigned a timestamp, which the DBMS uses to manage consistency conflicts. Each table has a row key, serving as its primary key. It is noteworthy that, unlike relational databases, NoSQL column-family stores do not support joins. Finally, tables in our metamodel include two dictionaries for configuration parameters related to sharding and distribution, which are key design considerations for such types of databases. Meanwhile, hardware and replication parameters are defined at the database level. These configuration parameters are generally determined based on the platform used, hardware capacity and expert feedback.

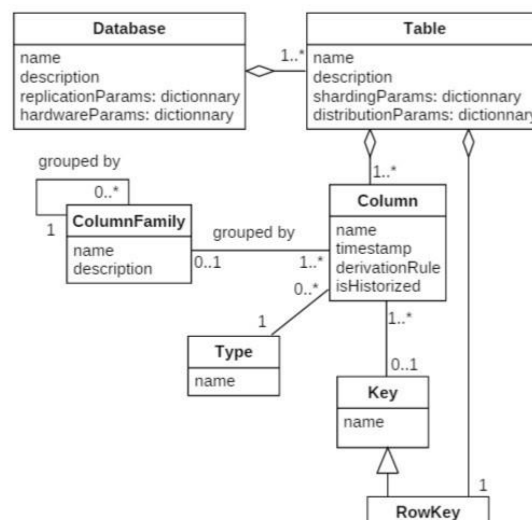

Figure 7. Our proposed column-family oriented metamodel.

6.2 From PIM to Column-Family oriented PSM

To design a column-family oriented data warehouse, similar to a key-value oriented data warehouse, the flat model is the only viable approach. This model utilizes a single table to manage both facts and dimensions, including both regular and many-to-many dimensions, with columns organized into column families.

Indeed, one crucial design consideration in column-family stores is how columns are grouped, which implicitly defines how data is stored and accessed. Therefore, the organization of columns must be carefully planned based on access patterns, as defined at the PIM level. The division of columns into column families can be guided by a straightforward transformation rule, such as using access frequency, or it can involve more complex criteria, like clustering columns into sub-groups according to the "queried with" association.

Figure 8 shows an example of a dynamic transformation rule: The rule generates a table containing all attributes of facts and dimensions, grouped into two column families, one for frequently queried attributes and the other for rarely accessed attributes. Additionally, the rule recommends two configuration parameters: the compaction strategy and the clustering key.

```
rule Fact2Table{
    from f: PIM!Fact
    to T: CL!Table (
    Name <- f.Name,
    distributionParams <- 'compaction, LeveledCompactionStrategy ',
    shardingParams <- 'clusteringkey, date',
    composed <- f.Attributes -> asSequence() -> collect(e|thisModule.FA2Column(e)),
    composed <- CL!Column.allInstances(),
    ID <- thisModule.Fact2RowKey(f),
    composed <- T.ID.attributes
    ),
    cf1:CL!ColumnFamily(
    Name <- 'FrequentlyAccessed'
),
    cf2:CL!ColumnFamily(
    Name <- 'RarelyAccessed'
)}
rule DA2Column{
    from da:PIM!DimensionAttribute
    to c:CL!Column(
    Name <- da.Name,
    Type <- da.Type,
    DerivationRule<- 'Derivation rule: ' + da.DerivationRule,
    IsHistorized <- da.isHistorized ,
    groupedBy <- if da.queryingFrequency='frequent' then CL!ColumnFamily.allInstances()-> select(cf|cf.Name='FrequentlyAccessed')->first()
                else
                CL!ColumnFamily.allInstances()->select(cf | cf.Name = 'RarelyAccessed')->first()
                endif
)}
rule FA2Column{
    from fa:PIM!Fact_Attribute
    to f:CL!Column(
    Name <- fa.Name,
    Type <- fa.Type,
    Formula<- 'Derivation rule: ' + fa.DerivationRule+ ' and Aggregation Function: ' + fa.Aggregation_Function,
    groupedBy <- if fa.queryingFrequency='frequent' then CL!ColumnFamily.allInstances()-> select(cf|cf.Name='FrequentlyAccessed')->first()
                else
                CL!ColumnFamily.allInstances()->select(cf | cf.Name = 'RarelyAccessed')->first()
                endif
)}
```

Figure 8. ATL dynamic transformation rule for converting a fact into a table with two column-families; one for frequently accessed attributes and one for rarely accessed attributes. The rule specifies configuration parameters essential for data mart table creation.

7. Graph-oriented Data Warehouse

7.1 Graph-oriented Metamodel

Graph databases have many distinct characteristics when compared to other NoSQL databases and these characteristics can also vary when comparing different graph database platforms. In this paper, we introduce a metamodel designed to capture the essential concepts necessary for the implementation of a data warehouse in a graph-database environment. The primary components in a graph-oriented database are nodes, which store data and edges which represent relationships between nodes, as depicted in Figure 9. Each edge has two endpoints: a start and an end node. Furthermore, like nodes, edges can also possess attributes. They may be directed or undirected, a distinction represented in our metamodel by the "isDirected" attribute. The "type" attribute categorizes edges based on other characteristics, such as whether they are weighted, reflexive or composite. Additionally, attributes may include an "isUnique" property, allowing for the specification of unique identifiers at the node level, functioning like primary keys. Each attribute can also be assigned one or many indices of different types. Finally, unlike other

types of NoSQL stores, in graph-oriented databases, most configuration parameters are set at the database level rather than at the node level, as represented in our metamodel. However, we have retained a configuration dictionary at the node level for specific cases.

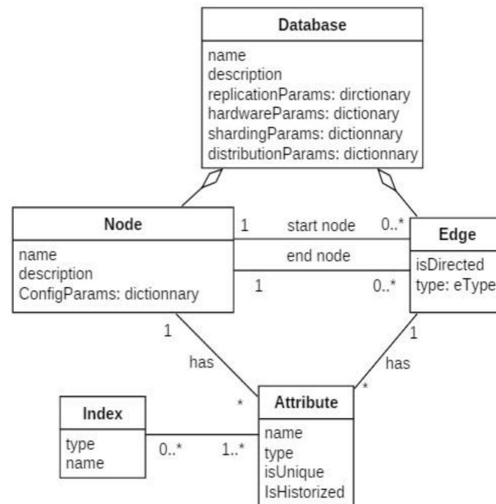

Figure 9. Graph-oriented metamodel (PSM).

```
helper  def: FactFAThreshold : Integer = 30;
rule Fact2SingleNode{
    from f: PIM!Fact (f.Attributes.count()<=thisModule.FactFAThreshold or f.isVpartitioned=false)
    to N: Graph!Node (
    Name <- f.Name,
    Description <- 'Fact Node',
    Attributes <- f.Attributes -> asSequence() -> collect(a|thisModule.FA2Attribute(a))
)
}
rule Fact2MultiNode{
    from f: PIM!Fact (f.Attributes.count()>thisModule.FactFAThreshold and f.isVpartitioned=true)
    to nodes : Sequence(Graph!Node) (
    nodes <- thisModule.SplitFA(f.Attributes)
    )
}
```

Figure 10. ATL dynamic transformation rule for mapping a fact into one or multiple nodes based on the number of fact attributes.

7.2 From PIM to Graph-oriented PSM

In designing a graph-oriented data warehouse, two primary models are typically considered; star and snowflake, while the flat model with one single node is not recommended in graph stores as it does not leverage the inherent advantages of graph databases.

- Star model: this model employs separate nodes to store facts and dimensions, joined through edges. Hierarchies are placed in the related dimension's node.
- Snowflake model: this model employs separate nodes to store facts, dimensions and hierarchies, joined through edges.

Building on these two fundamental models, our approach aims to dynamically generate a hybrid model that satisfies the DW requirements, rather than being a predetermined choice. Below, we present possible mappings for each DW component:

- Fact F: Each fact is mapped to a single node N containing all the fact attributes, which is then connected to its dimension nodes through edges. Alternatively, a fact can be mapped to several nodes that contain groups of fact attributes. This approach is used in scenarios such as a high number of fact attributes or a large size of the fact. Figure 10 illustrates an example of a dynamic transformation rule for mapping a fact based on the number of fact attributes.
- Regular Dimension *D*: A dimension can be either normalized or denormalized, depending on the transformation rule parameters. In the normalized case, each dimension is mapped to a node N_d, which is joined to the fact node *N* and contains all the dimension's attributes and hierarchies. In the denormalized case, the dimension and its hierarchies are mapped to distinct nodes. It is

important to note that, in both scenarios, the dimension nodes can be further vertically split into multiple nodes under certain conditions, such as an excessive number of attributes or if some attributes are rarely queried. Furthermore, to leverage the attributes of edges, we add two attributes to each edge to manage historical data: is Last and Modification Date. Figure 11 illustrates an example of a dynamic transformation rule for dimension mapping, driven by the changing frequency of the dimension.

- Many-to-many dimensions: These dimensions can be implemented similarly to regular dimensions, using a dedicated node connected to the fact node through edges.

```
rule Dim2Node{
    from d:PIM!Dimension (d.changingFrequency='rare' or d.changingFrequency='')
    to N: Graph!Node(
    Name<- d.Name,
    Description <- d.Description,
    Attributes <- d.Attributes -> asSequence() -> collect(a|thisModule.DA2Attribute(a))
    ),
    e:Graph!Edge (
    startNode <- Graph!Node.allInstances() ->select (n|n.Description='Fact Node'),
    endNode <- N
    )
}
rule Base2Node{
    from b:PIM!Base (b.Dimension.changingFrequency='frequent')
    to N: Graph!Node(
    Name<- b.Name,
    Description <- b.Description,
    Attributes <- b.Attributes -> asSequence() -> collect(a|thisModule.DA2Attribute(a))
    ),
    e:Graph!Edge (
    startNode <- Graph!Node.allInstances() ->select (n|n.Name=b.rolls_up_to.Name),
    endNode <- N
    )
}
lazy rule DA2Attribute{
    from da:PIM!DimensionAttribute
    to a:Graph!Attribute(
    Name<- da.Name,
    Type <- da.Type ,
    IsUnique<- da.isUnique,
    IsHistorized <- da.isHistorized
    )
}
```

Figure 11. ATL dynamic transformation rule for converting a dimension; slowly changing dimensions and their hierarchies are mapped to a single node. In contrast, frequently changing dimensions are mapped with a separate node for each hierarchy level.

8. FROM NOSQL PSM TO CODE

In addition to model-to-model transformations, the MDA offers the capability of generating text through model-to-text transformations. This feature is particularly useful when generating the implementation code for a specific platform or generating text in any desired format. In our approach, as previously mentioned, three different files are generated for each data-warehouse project:

- The data warehouse or data-mart implementation code for the target platform.
- Data-template file.
- Schema-validation file.

To ensure flexibility and clarity for the framework's users, the data-template and schema-validation files are generated in JSON format.

8.1 From PSM to Implementation Code

In NoSQL stores, due to the schemaless aspect of these databases, a predefined schema is not required. In document databases, the schema is dynamically derived from the documents inserted into the database. For other types of NoSQL stores, creating a data warehouse necessitates defining only the table names while the attributes are deduced at the data-loading time. For this reason, the implementation code generated by our framework is coupled with other files to provide developers with all metadata available in the PSM model.

In case of key-value databases, the table can be created with an initial schema. This can be performed through a JSON file in some platforms or using a programming language or a platform-specific script. To illustrate that, we generate in this work the implementation code for the DynamoDB [26] platform. In DynamoDB, only the primary key attribute is mandatory and an optional sort key can be specified.

The remaining attributes are defined during the data-loading process. Below, we present a conceptual template for creating a table in DynamoDB:

```
aws dynamodb create-table \
–table-name T-name \
–attribute-definitions \
AttributeName=PKA-name,
AttributeType=PKA-type \
AttributeName=SKA-name,
AttributeType=SKA-type \
–key-schema \
AttributeName=PKA-name,KeyType=HASH \
AttributeName=SKA-name,KeyType=RANGE \
```

In the context of a column-family data warehouses, various specific languages are available for generating the implementation code, with most of them being specific to each database system. In our framework, we generate the code to implement the data warehouse in Hbase. In this latter, only the table name and the names of its column-families are required. Below, we present a conceptual template for creating a table in Hbase:

```
CREATE HBASE TABLE T   name,CF1   name,CF2   name, ..., CFn   name
```

Figure 12 illustrates the transformation rule used to generate the data-mart table for our case study.

```
query Hbase_File=Column!Database.allInstances()->
collect(d | d.Code_Generator().writeTo('/PIM2PSM/Hbase.sql'));

helper context Column!Database def : Code_Generator(): String=
 Column!Table.allInstances() -> select(d|not d.oclIsUndefined()) -> iterate (t;code:String=''|
code
+ 'CREATE '+'"'+ t.Name+'"'
+    Column!ColumnFamily.allInstances() -> select(c|not c.oclIsUndefined()) -> iterate (cf;columnsF:String=''|columnsF
 + ' , ' + '"'+ cf.Name +'"'
 ))
;
```

```
CREATE "StoreSales" , "FrequentlyAccessed" , "RarelyAccessed"
```

Figure 12. M2T transformation rule and the obtained implementation code to create the data-mart table in HBase for our case study.

In graph-oriented databases, many specific query languages exist, depending on the platform used, such as Gremlin, SPARQL and Cypher. In our framework, we generate a code for Neo4j, which utilizes the Cypher language. In this scenario, only the names of the nodes and the edges joining them are specified:

```
CREATE (f:N-name) CREATE (di : Ndi -name)// foreach dimension
MATCH (f:N-name), (di : Ndi   name)
CREATE (f)-[r:Ed-name]   > (di)
//foreach edge
```

8.2 From PSM to Schema-validation File

In data-warehousing systems, each field plays a crucial role in the analysis and visualization phases, making precise control over the DW's data essential. Such control ensures accurate calculations and prevents the occurrence of empty data in reports. However, in NoSQL data stores, being schemaless, the responsibility for managing the database schema and business rules is assigned to the application layer rather than to the database itself. Consequently, in NoSQL-oriented data-warehousing systems, schema control typically occurs during the loading phase.

In our MDA approach, the schema-validation file provides the user with all available metadata and outlines on how data should be organized, considering that the implementation code does not offer this information. This file can also be used in data-loading batches to control data structure. It serves to outline the required fields and to provide a detailed description of each field. In case of a document-oriented data warehouse, a schema-validation file is generated for each collection encompassing all its fields or embedded documents. In the case of a key-value data warehouse, all table attributes are directly listed in a single file. For a column-family data warehouse, a single schema-validation file is generated for the data mart, with column-family attributes grouped in embedded documents. In case of graph-oriented DW, a validation file is generated for each node. Figure 13 shows the schema-validation file

generated for our case study.

```
-- command to write in JSON file
query Schema_validation=Doc!DocCollection.allInstances()->
collect(d | d.Code_Generator().writeTo('/PIM2PSM/SchemaValidation.json'));

helper context Doc!DocCollection def : Code_Generator(): String=
 '{\n "$id":"/'+self.Name+' .schema.json",//schema URI \n' +
    '"title":"'+self.Name+'",\n'+
    '"description":"'+self.Description+'",\n'+
    '"type":"object",\n'+
    '"required":['+ self.required -> select(c|not c.oclIsUndefined()) ->
                iterate (fa;required:String=''|
    required + fa.Name+
    if self.required.last()<>fa then ','
    else '],\n'
    endif
    )
+
'"Properties":{ \n'+
'//Measures Fields \n'+
self.contains -> select(c|not c.oclIsUndefined()) -> iterate (fa;contains:String=''|
    contains+
'"'+ fa.FName+'":{ \n'+
'"type":"' + fa.FType+ '", \n'+
'"description":"' + fa.FDescription +'"} \n'
    )
+self.Embedded -> select(c|not c.oclIsUndefined()) -> iterate (em;contains:String=''|
    contains+
    '//'+ em.Name+' Embedded Dimension \n'+
'"'+ em.Name +'":{'+
'type:"object",\n'+
'properties:{\n'+
em.contains -> select(c|not c.oclIsUndefined()) -> iterate (e;field:String=''|
    field+
'"'+ e.FName+'":{ \n'+
'"type":"' + e.FType+ '", \n'+
'"description":"' + e.FDescription +'"} \n'
    )
)
+' }'
;
```

```
1{
2 "$id":"/Store .schema.json",//schema URI
3"title":"Store",
4"description":"Sales measures",
5"type":"object",
6"required":[WholeSalesCost,ExTax,NetPaid,NetProfit,],
7Properties":{
8//Measures Fields
9"WholeSalesCost":{
10"type":"Float",
11"description":"Calculates the total amount of sales"
12} ,
13"ExTax":{
14"type":"Float",
15"description":"Calculates the amount of taxes"
16} ,
17"NetPaid":{
18"type":"Float",
19"description":"Calculates the net paid by customers"
20} ,
21"NetProfit":{
22"type":"Float",
23"description":"Calculates the net profit by deducing the amount of ta
24} ,
25//Item Embedded Dimension
26"Item":{
27type:"object",
28properties:{
29"ItemDescription": {
30type:"String",
31description:"Represents the product description"},
32"ProductName": {
33type:"String",
34description:""},
35},
36//Date Embedded Dimension
37"Date":{
38type:"object",
39properties:{
40"Date": {
```

Figure 13. M2T transformation rule to generate a schema-validation file.

8.3 From PSM to Data Template

To enhance developers' understanding of the data-warehouse model, we generate a data template for each destination model. This generated template is a JSON file that includes examples of automatically generated data, along with descriptions of each attribute as detailed in the PIM model and transferred to the target PSM model. Figure 14 illustrates the definition of the model-to-text transformation rule and the resulting data template in the case of a key-value PSM.

```
-- @path KV=/PIM2PSM/KeyValue_PSM.ecore
-- command to write in JSON file
query Template_File=KV!Table.allInstances()->
collect(d | d.Code_Generator().writeTo('/PIM2PSM/CompactDataTemplate.json'));
helper context KV!Table def : Code_Generator(): String=
 '{\n ' + self.PK.Name + ':"'+self.contains -> select(c|not c.oclIsUndefined()) -> iterate (k;pk:String=''|
    pk
    +'#'+ k.Name+':1'
)
+'",\n'
+self.contains -> select(c|not c.oclIsUndefined() and c.Name=self.Name) -> iterate (fact;f:String=''|
    f
    +fact.composed -> select(a|not a.oclIsUndefined()) -> iterate (fa;a:String=''|
    a+ fa.Name+':"'+fa.Type.TypeToData() +'"//'+fa.Description+
    if fact.composed.last()<>fa then ',\n'
    else '\n'
    endif
    )
)+
self.contains -> select(c|not c.oclIsUndefined() and c.Name<>self.Name) -> iterate (k;edge:String=''|
    edge
    +'ID_'+ k.Name+':"'+k.Name+'1" //'+  k.Description+
    if self.contains.last()<>k then ',\n'
    else '\n'
    endif
)
+' }\n '
+self.contains -> select(c|not c.oclIsUndefined()and c.Name<>self.Name) -> iterate (dim;d:String=''|
    d+ '{\n'+ self.PK.Name+ ':"'+dim.Name+'1",\n'
    +dim.composed -> select(a|not a.oclIsUndefined()) -> iterate (da;a:String=''|
    a+ da.Name+':"'+ dim.Name+'1 '+ da.Name+'"//'+ da.Description+
    if dim.composed.last()<>da then ',\n'
    else ' \n }\n'
    endif
    ));
helper context String def : TypeToData(): String=
if self = 'Integer' or self='Float' then '100'
    else if self='String' then 'A'
            else if self='Date' then '01/01/2000'
                else self
                endif
            endif
```

```
1{
2 PK:"#Item:1#Customer:1#Date:1#Store:1#StoreSales:1",
3WholeSalesCost:"100"//Calculates the total amount of sales,
4ExtTax:"100"//Calculates the amount of taxes,
5NetPaid:"100"//Calculates the net paid by customers,
6NetProfit:"100"//Calculates the net profit by deducing the amo
7ID_Item:"Item1" //Represent the product description,
8ID_Customer:"Customer1" //Represents the customer description,
9ID_Date:"Date1",
10ID_Store:"Store1" //Represents the store description,
11 }
12 {
13PK:"Item1",
14ItemDescription:"Item1 ItemDescription",
15CurrentPrice:"Item1 CurrentPrice",
16Category:"Item1 Category",
17Color:"Item1 Color",
18ProductName:"Item1 ProductName"
19 }
20{
21PK:"Customer1",
22FirstName:"Customer1 FirstName",
23LastName:"Customer1 LastName",
24Login:"Customer1 Login",
25EmailAdresse:"Customer1 EmailAdresse"
26 }
27{
28PK:"Date1",
29Date:"Date1 Date",
30Month:"Date1 Month",
31Year:"Date1 Year"
32 }
33{
34PK:"Store1",
35Name:"Store1 Name",
36NbEmployees:"Store1 NbEmployees",
37FloorSpace:"Store1 FloorSpace"
38 }
```

Figure 14. M2T transformation rule to generate a JSON data-template file.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 10, No. 02, June 2024.

9. CASE STUDY

To illustrate and validate our approach, we conducted experiments utilizing the benchmark database TPC- DS [27], which comprises a fact table named StoreSales and four dimensions; namely, Item, Customer, Date and Store. We used the Eclipse Modeling Framework (EMF) to create the PIM metamodel, along with our four proposed PSM metamodels; namely, document, key-value, column-family and graph. We then implemented the model-to-model and model-to-text transformation rules using the ATL language [19]. Figure 15 displays the metamodel instance and the properties of the Customer Login attribute, with the "querying frequency" set to "rare." This attribute serves as a parameter for the dynamic transformation rule related to dimensions and their attributes, as illustrated in Figure 4. The same applies to the attributes First Name, Last Name, Email Address, Current Price, Color, NbEmployees and FloorSpace. This is the reason why they have been placed in a separate column family, as explained by the rule displayed in Figure 8. The Customer dimension, defined as rarely accessed, has been placed in a separate collection in the document store, while the Date and Store dimensions were embedded within the fact collection due to their frequent access. In contrast, due to its estimated size, the Item dimension was divided: frequently queried attributes were placed in an embedded document and the remaining attributes were stored in a separate collection. In the graph scenario, the Item dimension, identified as frequently changing, was normalized by creating a dedicated node for the hierarchy-level category, as defined by the transformation rule displayed in Figure 11. In the key-value scenario, a single table was created to hold all facts and dimension attributes. These were grouped by KeyValueSet to show data organization, even though this concept is not directly applied during the implementation phase. Similarly, we have implemented model-to-text transformation rules, facilitated by the ATL writeTo function, as previously demonstrated.

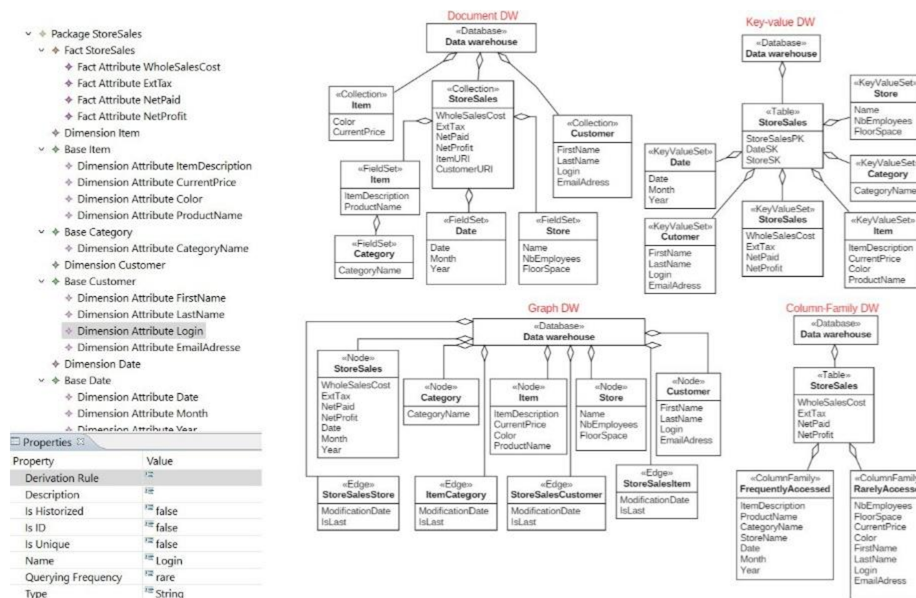

Figure 15. On the left, the PIM of our case study, showing the 'Login' attribute defined as rarely queried. On the right, the obtained physical models for the four NoSQL stores, with attributes organized according to the transformation rules previously presented.

10. CONCLUSION

In this paper, we introduced an MDA-based framework for the design and implementation of NoSQL-oriented data warehouses. We proposed a conceptual model that captures all the essential concepts needed to design a data warehouse and facilitates the transition to specific models. Additionally, we presented four metamodels to represent the logical model of a data warehouse related to document, column-family, key-value and graph data stores.

Furthermore, we proposed possible designs for a data warehouse in each type of data store. These designs are supported by dynamic transformation rules, enabling the automatic and dynamic derivation of target models. These models are tailored based on the metadata provided at the conceptual-model level, involving not just structural concepts, but also domain and access parameters. By employing

model-to-text transformations, our framework generates three critical files to implement and document the obtained model. To validate our approach, we have presented a case study demonstrating the practical implementation of dynamic transformation rules in the ATL language, showcasing the resulting models and files. Our proposal is driven by our conviction that MDA serves not only to model and automate data-warehouse creation, but also to foster a collaborative environment. This environment enables developers to consolidate their design patterns and feedback through the definition of transformation rules.

In our future work, we plan to delve deeper into each individual model, defining specialized design patterns for each targeted platform and comparing their performances. Additionally, we aim to incorporate more data platforms into our MDA framework, thereby expanding the scope and applicability of our approach.

REFERENCES

[1] W. H. Inmon and D. Linstedt, Data Architecture:A Primer for the Data Scientist, Elsevier Kaufman, 2014.

[2] C. Costa and M. Y. Santos, "Evaluating Several Design Patterns and Trends in Big Data Warehousing Systems," Proc. of the Int. Conf. on Advanced Information Systems Engineering, pp. 459-473, Springer, Cham, June 2018.

[3] F. Halper, "Modernizing the Organization to Support Data and Analytics," TDWI, Best Practices Report, [Online], Available: https://tdwi.org/research/2022/06/ppm-all-best-practices-report-modernizing-the-organization-support-data-analytics.aspx?tc=assetpg, 2022.

[4] D. Stodder, "Modernizing Data and Information Integration for Business Innovation," TDWI, [Online], Available: https://f.hubspotusercontent30.net/hubfs/6618383/Report%20-%20TDWI%20Best%20Practices%20-%20Q4-2021.pdf, Q4 2021.

[5] S. Chowdhury, [Online], Available: https://www.ibm.com/developerworks/analytics/library/baaugment-data-warehouse4/ba-augment-data-warehouse4-pdf.pdf.

[6] OMG, "MDA Guide Rev. 2.0," Object Management Group Model Driven Architecture (MDA), OMG Document ormsc/2014-06-01, [Online], Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01, 2014.

[7] M. Chevalier, M. E. Malki, A. Kopliku, O. Teste and R. Tournier, "How Can We Implement a Multi-dimensional Data Warehouse Using NoSQL?," Proc. of the Int. Conf. on Enterprise Information Systems, LNBIP, vol. 241, pp. 108-130, Springer, Cham, April 2015.

[8] M. Chevalier, M. El Malki, A. Kopliku, O. Teste and R. Tournier, "Document-oriented Data Warehouses: Models and Extended Cuboids," Proc. of the 2016 IEEE 10th Int. Conf. on Research Challenges in Information Science (RCIS), DOI: 10.1109/RCIS.2016.7549351, Grenoble, France, 2016.

[9] M. Boussahoua, O. Boussaid and F. Bentayeb, "Logical Schema for Data Warehouse on Column-oriented NoSQL Databases," Proc. of the Int. Conf. on Database and Expert Systems Applications, LNISA, vol. 10439, pp. 247-256, Springer, Cham, August 2017.

[10] A. Sellami, A. Nabli and F. Gargouri, "Transformation of data warehouse schema to NoSQL graph data base," Proc. of the 18th Int. Conf. on Intelligent Systems Design and Applications (ISDA 2018), vol. 2, pp. 410-420, Vellore, India, December 6-8, 2018, Springer International Publishing, 2020.

[11] A. Vaisman, F. Besteiro and M. Valverde, "Modeling and Querying Star and Snowflake Warehouses Using Graph Databases," Proc. of New Trends in Databases and Information Systems: ADBIS 2019 Short Papers, Workshops BBIGAP, QAUCA, SemBDM, SIMPDA, M2P, MADEISD and Doctoral Consortium, Proceedings 23, pp. 144-152, Bled, Slovenia, Springer International Publishing, September 8–11, 2019.

[12] R. Benhissen, F. Bentayeb and O. Boussaid, "GAMM: Graph-based Agile Multidimensional Model," CEUR, [Online], Available: https://ceur-ws.org/Vol-3369/paper2.pdf, 2023.

[13] F. Kalna, A. Belangour, M. Banane and A. Erraissi, "MDA Transformation Process of a PIM Logical Decision-making from NoSQL Database to Big Data NoSQL PSM," Int. J. of Engineering and Advanced Technology, vol. 9, no. 1, pp. 4208-4215, 2019.

[14] D. Prakash, "NOSOLAP: Moving from Data Warehouse Requirements to NoSQL Databases," Proc. of the 14th Int. Conf. on Evaluation of Novel Approaches to Software Engineering, vol. 1: ENASE, pp. 452-458, DOI: 10.5220/0007748304520458, May 2019.

[15] R. Yangui, A. Nabli and F. Gargouri, "Automatic Transformation of Data Warehouse Schema to NoSQL Data Base: Comparative Study," Procedia Computer Science, vol. 96, pp. 255-264, 2016.

[16] L. Oukhouya, A. El Haddadi, B. Er-Raha and A. Sbai, "Automating Data Warehouse Design With MDA Approach Using NoSQL and Relational Systems," J. of Theoretical and Applied Information Technology, vol. 101, no. 23, pp. 7941-7957, 2023.

[17] A. Srai and F. Guerouate, "MDA Approach for Generating the PSM Model for the NoSQL Key-value Database, Application on Redis," Proc. of the 2023 3rd Int. Conf. on Innovative Research in Applied

Science, Engineering and Technology (IRASET), pp. 1-5, Mohammedia, Morocco, 2023.
[18] F. Abdelhedi, R. Jemmali and G. Zurfluh, "Relational Databases Ingestion into a NoSQL Data Warehouse," arXiv preprint, arXiv: 2203.06949, 2022.
[19] Eclipse, "ATL Documentation," [Online], Available: https://www.eclipse.org/atl/documentation.
[20] OMG, "MDA - The Architecture of Choice for a Changing World," [Online], Available: https://www.omg.org/mda/
[21] K. Letrache, O. El Beggar and M. Ramdani, "The Automatic Creation of OLAP Cube Using an MDA Approach," Software: Practice and Experience, vol. 47, no 12, pp. 1887-1903, 2017.
[22] W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy and B. Luo, "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments: A Systematic Literature Review," Big Data and Cognitive Computing, vol. 7, no. 2, Atricle no. 97, DOI: 10.3390/bdcc7020097, 2023.
[23] P. J. Sadalage and M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, 1st Edition, ISBN-10: 0321826620, Pearson Education, 2013.
[24] A. Meier and M. Kaufmann, SQL & NoSQL Databases, ISBN-10: 3658245484, Springer Fachmedien Wiesbaden, 2019.
[25] A. Vaisman and E. Zimányi, Data Warehouse Systems: Design and Implementation, 2nd Edition, ISBN-10: 3642546544, 2022.
[26] G. DeCandia et al., "Dynamo: Amazon’s Highly Available Key-value Store," ACM SIGOPS Operating Systems Review, vol. 41, no. 6, pp. 205-220, 2007.
[27] TPC BENCHMARK, StandardSpecification, Version 3.2.0, pp. 1-141, [Online], Available: http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf, June 2021.

ملخص البحث:

في هذه الأيّام، يعدّ تحديث البيئات الخاصّة بمستودعات البيانات من التّحدّيات الأساسية في أنظمة دعْم القرارات. وهذا التّحديث ضروري لضمان قابلية الأنظمة للتّوسيع وتلبية متطلّبات عملها، لا سيّما بعد ظهور "البيانات الضّخمة". وهناك حلّ واعد لهذه المسألة يتمثّل في تنفيذ مستودعات بياناتٍ مع مخازن للبيانات على غرار (NoSQL).

في هذه الورقة، نقدّم إطارَ عملٍ معزَّزاً بنموذجٍ لتصميم وتنفيذ مستودعات حديثة للبيانات. وتهدف المنهجية المتّبعة إلى تقديم طريقةٍ تعاونيةٍ وديناميكيةٍ وقابلةٍ لإعادة الاستخدامٍ لتطوير مستودعات بياناتٍ مخصَّصة لمشاريع ذات متطلّبات معينة. وتسهّل الطّريقة المقترحة التّوليد الأوتوماتيكي والدّيناميكي لنموذج مستودع بياناتٍ هجينٍ من نموذجه المفاهيمي، يشمل المتغيّرات البنيوية، ومتغيّرات الحقول، ومتغيّرات الوصول.

كذلك يتضمّن إطار العمل المقترح توليد الشّيفرة الخاصّة بالتّنفيذ لمستودع البيانات المقترح، إلى جانب مجموعةٍ من الملفّات للتحقُّق من النّموذج المقترح وتوثيقه ورسمٍ مخطط مستودع البيانات على منصّة هَدَف. من جانب آخر، نقدّم دراسةَ حالةٍ مفصّلةً لتسليطِ الضّوء على فاعلية إطار العمل المقترح.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).