Computer Standards & Interfaces 83 (2023) 103642

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

MDICA: Maintenance of data integrity in column-oriented database applications

María José Suárez-Cabal [*], Pablo Suárez-Otero, Claudio de la Riva, Javier Tuya

Department of Computing, University of Oviedo, Spain

A R T I C L E I N F O

Keywords:
NoSQL
Column-oriented database
Cassandra
Database application
Data integrity

A B S T R A C T

Current information technologies generate large amounts of data for management or further analysis, storing it in NoSQL databases which provide horizontal scaling and high performance, supporting many read/write operations per second. NoSQL column-oriented databases, such as Cassandra and HBase, are usually modelled following a query-driven approach, resulting in denormalized databases where the same data can be repeated in several tables. Therefore, maintaining data integrity relies on client applications to ensure that, for data changes that occur, the affected tables will be appropriately updated. We devise a method called MDICA that, given a data insertion at a conceptual level, determines the required actions to maintain database integrity in column-oriented databases. This method is implemented for Cassandra database applications. MDICA is based on the definition of (1) rules to determine the tables that will be impacted by the insertion, (2) procedures to generate the statements to ensure data integrity and (3) messages to warn the user about errors or potential problems. This method helps developers in two ways: generating the statements needed to maintain data integrity and producing messages to avoid problems such as loss of information, redundant repeated data or gaps of information in tables.

1. Introduction

The development of new information technologies and their use in daily routines have allowed society to be completely interconnected. Information systems available to citizens, social networks and the Internet of Things, among others, generate large amounts of data, known as Big Data, which are stored for management or further analysis. On many occasions, this information is unstructured or distributed and comes from different data sources which has motivated the emergence of new paradigms of data storage and information management, different from traditional relational database systems.

The NoSQL database (Not only SQL) paradigm [1] addresses non-relational databases that do not use SQL for data manipulation. Unlike relational databases, they relax the ACID properties (Atomicity, Consistency, Isolation, Durability) as these are difficult to maintain, especially with distributed data. Nevertheless, NoSQL databases adhere to the BASE properties (Basically Available, Soft state, Eventually consistent) [2] providing horizontal scaling, which enables them to support a large number of simple read/write operations per second [3]. Moniruzzaman and Hossain [4] classify NoSQL databases in four basic categories: (1) key-value stores that associate identifiers and values, (2) document databases designed to manage and store documents with semi-structured data, (3) wide-column stores or column-oriented databases which present a distributed, tabular data structure that associates multiple attributes per key and (4) graph databases that use structured relational graphs of interconnected key-value pairings.

Many definitions of data integrity have been proposed [5], focusing on various aspects such as access control, data completeness, data consistency or correctness consisting of concurrency control, recovery mechanisms or semantic integrity [6]. If these aspects are not adequately managed, they can lead to different problems in information management. In Big Data and cloud environments, aspects of data integrity have been studied to prevent unauthorized modification of data from malicious attacks [7,8], ensure eventual consistency [9] when data is replicated throughout all the clusters [10,11] and, as NoSQL databases do not support transactions (with few exceptions), detect anomalies in data consistency [12] after a transaction developing transactional services [13].

Modeling NoSQL databases is usually query-driven [14] to optimize the query processing and, as a result, these databases are not normalized. In the case of key-value stores or column-oriented databases, which manage tabular data, each table is designed to satisfy a single query so

[*] Corresponding author.
E-mail address: cabal@uniovi.es (M.J. Suárez-Cabal).

https://doi.org/10.1016/j.csi.2022.103642
Received 27 October 2020; Received in revised form 21 February 2022; Accepted 1 April 2022
Available online 4 April 2022

all data that the query encompasses will be in this single table. Consequently, the same piece of data can be repeated in multiple tables and each table, and its data are isolated without references to another, as opposed to relational databases. These databases have some features that enforce semantic integrity like primary key or data type definitions, however, they do not have others to support referential integrity or guarantee the correctness when data are repeated. The scope of our work is focused on semantic integrity that cannot be enforced by using features of the NoSQL databases.

Consider, for example, an application which shows the list of book authors, the list of books of a given author published in a year and the list of books of a given author ordered by their title. Using a relational database there would be two tables: "Authors" with all the information of authors and "Books" for books and the reference of their authors. The data would be retrieved by three SQL queries with different FROM, WHERE and ORDER clauses. However, in column-oriented databases the usual design would be three tables, each of them to be requested by a query: "Authors", and "BooksByYear" and "BooksByTitle" with information of books and their authors but differently organized according to the search criteria (author and year of publication and author and title, respectively).

On the other hand, this approach to modeling, which does not require a conceptual data model, may also cause potential problems [15, 16] such as forgetting important domain concepts or their relationships, losing information, or even misunderstandings of the business rules that no stakeholder [17–24] notices because there is no representation of the data domain to be assessed. Researchers and companies which provide products and services for the commercial use of NoSQL databases [25, 26], have studied and made recommendations for the design of NoSQL databases considering conceptual data models besides queries. The goal is to avoid as far as possible inconsistencies or loss of information preserving features such as linear scalability and high availability without compromising performance.

Nevertheless, these proposals do not prevent that the same data are repeated in multiple tables due to denormalized models. If these data are not properly updated, data integrity could be endangered. Avoiding this issue relies on external mechanisms generally implemented in procedures of client programs that access data. These procedures should ensure that, independently of the number of times a piece of data is repeated in different tables, if a change is produced in any data, these data will be updated in each repetition. In the previous example, adding a book in a relational database involves inserting the book data in a single table, but in a column-oriented database, it must be added into two of the tables designed.

We address data integrity as the semantic integrity when data is repeated or referential integrity is not enforced. In this paper, we devise a method to support data integrity in column-oriented databases that we call *MDICA* (*M*aintenance of *D*ata *I*ntegrity in column-oriented database *A*pplications). Given a conceptual data model, a data insertion at a conceptual model level and a column-oriented database, MDICA determines the database statements that must be carried out against the database in order to preserve data integrity and advises the user about situations where it may be impended. Using MDICA, developers will get the necessary data definition and data management statements to include in their source code or, if a client program exists, they will be able to compare their procedures, and check whether data integrity will be preserved.

NoSQL databases are able to support a large number of read/write operations and they are optimised for obtaining and inserting data. Other typical operations (delete and update) in transactional systems are not efficiently supported by some of the NoSQL databases or are of little significance in terms of volume [27]. Thus, we will mainly focus on data insertions which are the most frequent in this paradigm designed for large volume data storage. Nevertheless, we will sketch out the approach for delete and update operations, leaving their detailed study for future work. We have implemented MDICA for Apache Cassandra™ [28] applications since Cassandra is the most popular wide-column store and one of the most used NoSQL databases [29].

We have previously addressed the initial idea about the maintenance of data integrity [30]. We now extend it with the following main contributions:

- To define the conditions that tables must satisfy to guarantee a good design according to a given conceptual data model.
- To provide an approach that determines for each insert operation in an entity or relation in a conceptual data model, the tables that are impacted by the operation in order to ensure data integrity.
- To determine the updates in the tables impacted by an insert operation maintaining data integrity.
- To provide error and warning messages for users about potential data integrity problems.
- An implementation of MDICA for Cassandra database applications, providing database statements in Cassandra Query Language (CQL) that will be generated to update the tables.
- A validation of MDICA through several case studies.

The remainder of the paper is organized as follows: Section 2 introduces the background and the terminology used. Section 3 includes an introductory example and the basic definitions of rules and procedures. Section 4 describes in detail rules and procedures to insert a tuple into an entity and Section 5 describes them in order to insert tuples into relations. In Section 6 the validation of the method is carried out in three case studies, including description of the experiments, analysis of results and threats to validity. Section 7 discusses MDICA extensions. The paper ends with conclusions and future work in Section 8.

2. Data models and notation

A data model [31] is a type of data abstraction that is used to represent the actual world of a system to be developed. It uses concepts that organize elements of data, their properties, and relationships between them. According to the abstraction level represented in data models, they can be categorized from a high-level or conceptual data model, which describes the domain or ideas close to the way final users perceive data, to a low-level or physical data model, which provides details of how the information is stored. Between these two extremes, we can find other models depending on the level of detail or what they represent, such as a logical data model which describes the semantics represented by a particular technology.

Here, we give some definitions and describe the basic notation that will be used in the remainder of the paper.

Conceptual data model.- A conceptual data model or conceptual model, denoted as M, which represents concepts of the system to be developed, is composed of entities, denoted as $e \in Ents(M)$, and relations between those entities, denoted as $R_{\{ei\}} \in Rels(M)$ where $e_i \in Ents(M)$. Entities and relations may be characterized by their properties, named attributes and denoted as $Attrs(I)$, where I is an item that hereafter refers to entity or relation. The primary key of an item I, denoted as $PK(I)$, is the set of attributes in I which uniquely identifies a concrete instance of the item. The rest of the attributes of I are non-key attributes. In a relation $R_{\{ei,\ ej\}}$, cardinality is the number of instances in the entity e_i related to the entity e_j, which can be 1:1, 1:n or n:m. Instances of an item (data at a conceptual model level) are represented by tuples. A tuple of an item I is defined as $tp(I) = <(a_1,v_1),\ (a_2,v_2),\ \ldots,\ (a_n,v_n)>$ where $a_i \in Attrs(I)$ and v_i is the value of a_i in the instance. We represent graphically a conceptual model as an Entity-Relationship model (ER model) [32].

Logical data model.- A logical data model or logical model, denoted as L, is composed of tables, denoted as $Tabs(L)$, which represent how data is stored in a column-oriented database. A table in a logical model L, denoted as $t \in Tabs(L)$, is a collection of ordered columns, denoted as $Cols(t)$. At the logical model level, data are represented by rows instead of tuples (used in the conceptual model). A row of a column *t* is defined as column-data pairs $r(t)=<(c_1, d_1), (c_2, d_2), \ldots, (c_n, d_n)>$ where $c_i \in Cols(t)$ and d_i is the data of c_i in the row. In Cassandra databases, the primary key of a table *t*, denoted as *Key(t)*, is the ordered list of key columns in *t*, composed of (1) partition key, *pKey(t)*: columns that identify the uniqueness of a particular row as well as the location or node where it is held, and (2) clustering key, *cKey(t)*: columns that determine the order of rows on a partition. The remaining columns of *t* are non-key columns.

Well-modelled table.- Well-modelled table denotes the table designed at a logical level following a given modeling process e.g. Chetboko et al. [20] or Mior et al. [21]. These processes state that a logical data model is obtained using the conceptual model and the queries of the application, which ensures a correct logical data model, not losing data represented by the conceptual model, to support query requirements allowing them to execute properly and to return data in the correct order.

Conceptual-Logical data model mapping.- A conceptual-logical data model mapping, denoted as *Map(M,L)*, is the association established between a conceptual model and a logical model. *Map(M,L)* provides information about:

(1) Associations between attributes of entities or relations, and columns of tables generated and vice versa. We say that an attribute generates a column when the association *attribute-column* exists where the attribute is mapped to the column,
(2) Tables generated from an item (entity or relation). We say that a table *t* is generated from item *I* when for each column of *t*, an association *attribute-column* exists with an attribute of *I*.

There are different types of *attribute-column* associations in mappings depending on if attributes are key (*ka*) or non-key (*na*), and if columns are key (*kc*) or non-key (*nc*):

- *ka-kc*: key attribute generates key column,
- *ka-nc*: key attribute generates non-key column,
- *na-kc*: non-key attribute generates key column, and
- *na-nc*: non-key attribute generates non-key column.

Fig. 1 depicts the mapping between an item *I* and a Cassandra table *t* generated from it. The item has two key attributes and three non-key attributes. The table has two key columns (a partition key I_pk_1 generated from the key attribute pk_1 and a clustering key I_a_1 generated from the non-key attribute a_1) and two non-key columns (I_pk_2 generated from the key attribute pk_2 and I_a_2 generated from the non-key attribute a_2). Note that attribute a_z does not generate any columns. The *attribute-column* associations are labelled according to each of the aforementioned types.

3. Data integrity based on conceptual and logical data models

This paper addresses the problem of data integrity maintenance in a column-oriented database when changes of data are produced by an insert operation of a tuple at a conceptual data model level. The data integrity maintenance process leverages the logical models generated from conceptual models and application queries through a set of mapping rules or patterns in the modeling process of column-oriented databases [20,21]. Fig. 2 depicts the integration of the modeling process (on the left) and the data integrity maintenance process, MDICA, devised in this work (on the right).

The inputs for the maintenance of data integrity will consist of (1) a tuple, which represents the data to insert, (2) a conceptual model and (3) a logical model. MDICA will generate the list of ordered data manipulation statements to execute against the database by applying two concepts defined in the following sections:

- Data manipulation rules (DMR) to determine which tables are impacted by the operation considering mappings between the conceptual and logical models,
- Data manipulation procedures (DMP) to determine which changes must be executed against the database to preserve data integrity (data manipulation statements).

Sometimes, the generated database statements will also retrieve data from tables in the database or even modify the logical data model (data definition statements). Moreover, MDICA will provide different types of messages to inform the users about potential data integrity problems.

3.1. Introductory example

In order to illustrate how data integrity has to be maintained, we use a simple example of a digital music store interacting with an information system in Cassandra, adapted from a Datastax tutorial [33], that we will also refer to throughout the remaining sections.

The conceptual model (Fig. 3 a) that represents users, playlists created by users, which are featured by tracks, tracks available in the system and artists who release tracks.

The logical model (Fig. 3 b), result of the modeling process, includes a table for each query in the application. In this example, required information is about playlists created by a user (Q1), artists whose name starts with a certain letter (Q2), tracks ordered by their title that have been released by a given artist (Q3) or that are from a specific genre (Q4) and tracks of a playlist (Q5). In each table, columns are labelled as primary key (K), partition key (C) with ascending (↑) or descending (↓) order, or non-key columns (without a label).

Table 1 displays the mapping, *Map(M,L)*, of items (entities and relations) in the conceptual model and tables generated from them in the logical data model with the associations between attributes and columns.

We consider below two situations in which there is an insert operation for the same relation between two entities but with different attributes in the tuples to insert. For each situation, we illustrate how data in a logical model should be updated, and we identify which problems may occur if data integrity is not maintained appropriately.

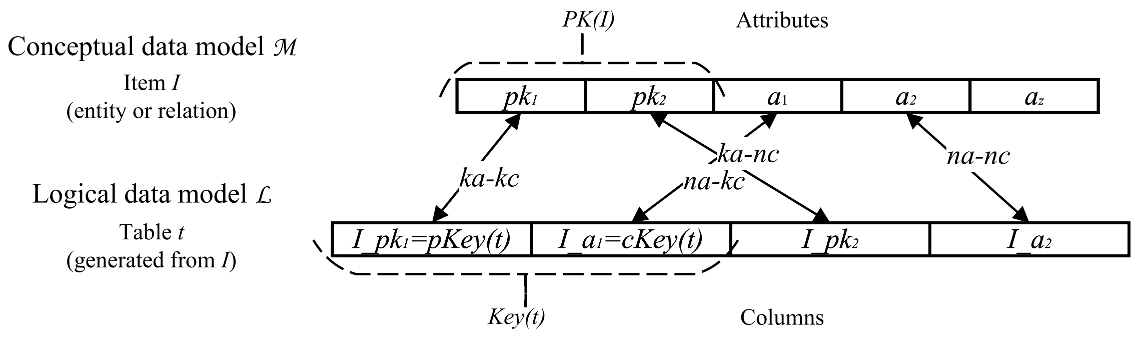


Fig. 1. Map(M,L) between an item I (entity or relation) and a table t generated from this item.

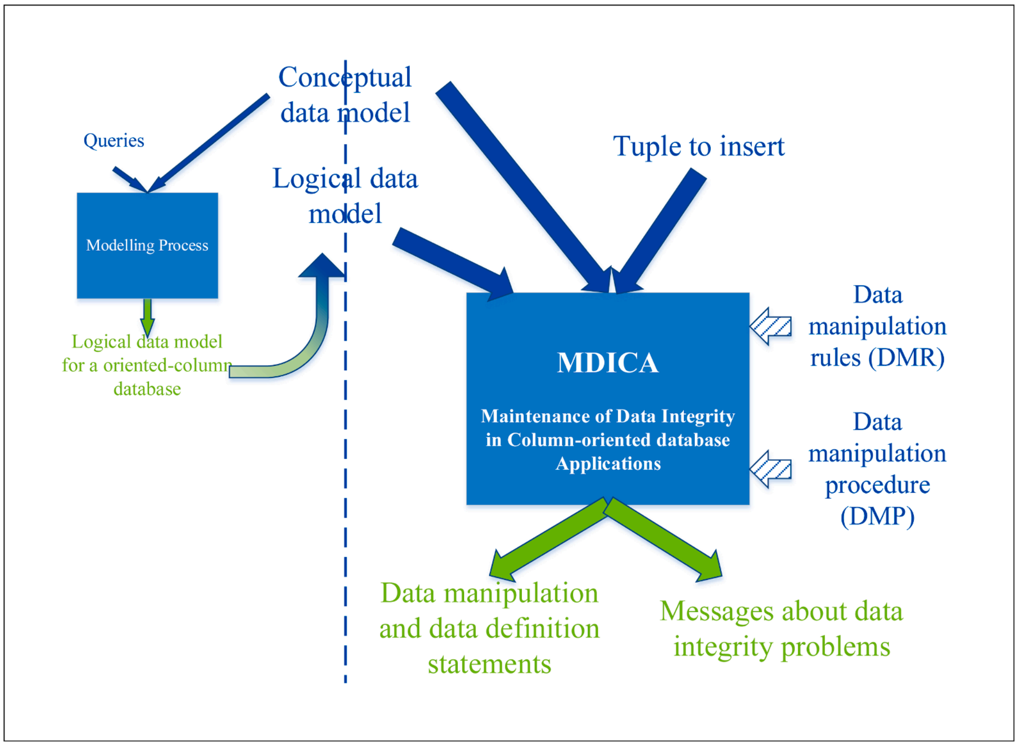

Queries
Conceptual data model
Logical data model
Modelling Process
Logical data model for a oriented-column database
Tuple to insert
MDICA
Maintenance of Data Integrity in Column-oriented database Applications
Data manipulation rules (DMR)
Data manipulation procedure (DMP)
Data manipulation and data definition statements
Messages about data integrity problems

Fig. 2. Integration of modeling and data integrity maintenance processes.

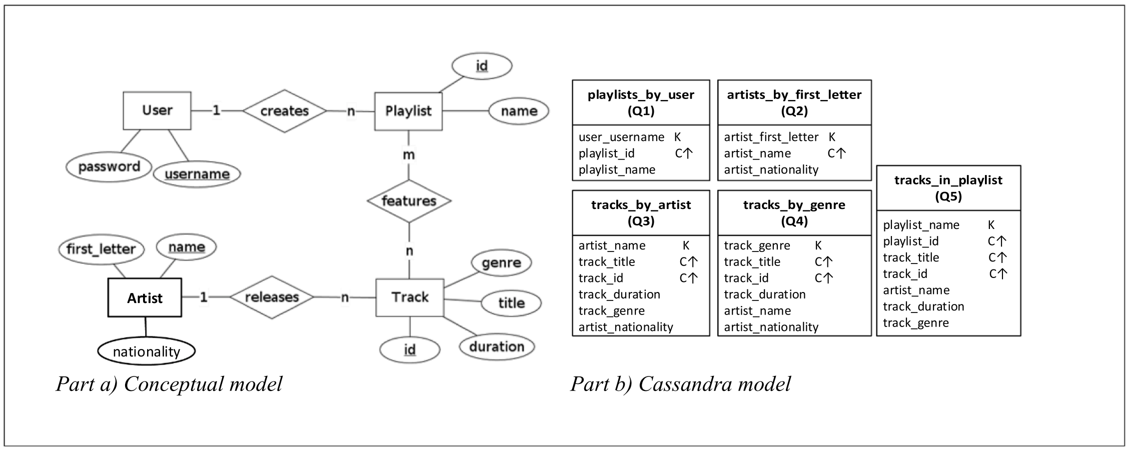


Fig. 3. Illustrative example: a digital music store.

Introductory Example, Part 1.- Consider a new track released by an artist. At the conceptual level, it implies inserting the artist (if it does not exist), the track and a new relation (releases) between these entities. According to the mapping (Table 1), in the logical model, tables to update are *artists_by_first_letter*, which stores data of the Artist, and *tracks_by_artist* and *tracks_by_genre*, which store data of "releases". So, in order to maintain data integrity in the database, it is necessary (1) to check whether the artist already exists in the table *artists_by_first_letter* and insert it if not, and (2) add new rows into *tracks_by_artist* and *tracks_by_genre*. The rest of the tables (*playlists_by_user* and *tracks_in_playlists*) are not impacted by the insertion.

Determining which tables must be updated is a difficult task if there are dozens of tables with data repeated and it is carried out manually. Omitting any of the tables will lead to potential integrity problems. For instance, if the table *tracks_by_genre* is forgotten, queries Q3 and Q4 will not retrieve the same tracks: the new track will be retrieved by Q3, which queries the table *tracks_by_artist*, but not by Q4, which queries the table *tracks_by_genre*.

Introductory Example, Part 2.-. Consider that another new track is released by the same artist, but now only the artist's name is known (neither first letter nor nationality are provided in the tuple). As the artist already exists in the database, tables to insert new data are *tracks_by_artist* and *tracks_by_genre*.

Cassandra only requires values for key columns in insert operations, the rest of the columns may not be provided. Therefore, inserting a new track without the artist's nationality in tables *tracks_by_artist* and *tracks_by_genre* is feasible although it would produce a situation of incompleteness of data: the nationality of that artist is known because it was previously inserted into *artists_by_first_letter* but now it will not be inserted for the new track. To avoid the incompleteness, it will be

Table 1
Map(M,L) for conceptual and logical data models.

	Entity Artist			Entity Track				Entity Playlist		Entity User	
	name	first_letter	nationality	id	title	genre	duration	id	name	username	password
Table playlist_by_user (Q1) from relation "creates" between Playlist and User											
user_username										ka-kc	
playlist_id								ka-kc			
playlist_name									na-nc		
Table artists_by_first_letter (Q2) from entity Artist											
artist_first_letter		na-kc									
artist_name	ka-kc										
artist_nationality			na-nc								
Table tracks_by_artist (Q3) from relation "releases" between Artist and Track											
artist_name	ka-kc										
track_title					na-kc						
track_id				ka-kc							
track_duration							na-nc				
tranck_genre						na-nc					
artist_nationality			na-nc								
Table tracks_by_genre (Q4) from relation "releases" between Artist and Track											
track_genre						na-kc					
track_title					na-kc						
track_id				ka-kc							
track_duration							na-nc				
artist_name	ka-nc										
artist_nationality			na-nc								
Table tracks_in_playlist (Q5) from relations "releases-features" between Artist, Track and Playlist											
playlist_name									na-kc		
playlist_id								ka-kc			
track_title					na-kc						
track_id				ka-kc							
artist_name	ka-nc										
track_duration							na-nc				
track_genre						na-nc					

necessary (1) to determine data for column *artist_first_letter* from the artist's name, (2) search the table *artists_by_first_letter* for the artist's nationality and (3) complete the data to be inserted in *tracks_by_artist* and *tracks_by_genre*.

However, it may be the case that the first letter cannot be determined and there is no table that retrieves the unknown information (first letter and nationality) for a given artist name. In this case, it will be necessary (1) to create a new table that relates artist names to first letters and nationalities, (2) populate it with data from *artists_by_first_letter*, and (3) query it to obtain the unknown information.

In this work, we will provide appropriate solutions to the maintenance of data integrity by inserting data in each table impacted by the change and/or creating and populating new tables to obtain the information required.

3.2. Data manipulation rules and procedures

MDICA is based on the definition and application of a set of rules and procedures to generate database statements and messages in order to maintain data integrity in a column-oriented database and identify potential threats.

In this section, we define these rules and procedures in general terms and, in subsequent sections, they will be particularized within the scope of inserting tuples: in an entity (Section 4), in a relation with cardinality 1:1, 1:n or n:m (Section 5.1) and in combinations of relations with a variety of cardinalities (Section 5.2).

The first step is to identify the tables in a logical model that must be updated when something in the real world, represented by a conceptual model, is inserted. To achieve this aim, we define the concept of Data Manipulation Rule (DMR):

Definition 1. *(Data Manipulation Rule, DMR)*.- Given a conceptual model M, an insert operation on an item *I* (entity or relation in M), a logical model L. A DMR determines:

(1) The *Map(M,L)* between M and L through the naming of the columns (by convention, an attribute of an item referenced as item. attr generates columns called item_attr),
(2) According to *Map(M,L)*, the set of target tables $TT \subseteq Tabs(L)$ which are impacted by the operation on the item *I*,
(3) The potential threats to the maintenance of data integrity if any target table is not well-modelled.

Depending on the mapping between *M* and *L*, risky situations may exist that will generate:

- Error messages "Absence of target tables to update" (ATT) which inform that it is not possible to execute the insert operation against the logical model because there is no target table.
- Warning messages "Absence of a key column generated from a key attribute" (TNW-K) and "Column not generated from any attribute" (TNW-C). These inform about a possibly misshapen logical model because a table is not well-modelled and may produce loss or unnecessary duplicity of data, or try to store data not supported by the conceptual model.

The second step is to generate the operations that must be executed against the database in order to properly update rows from values in a tuple. We define the Data Manipulation Procedure (DMP) to generate them:

Definition 2. *(Data Manipulation Procedure, DMP)*.- Given a tuple *tp(I)* to insert, the conceptual-logical data model mapping *Map(M,L)* between M and L, and the set *TT* of target tables determined by DMR. DMP determines:

(1) According to *Map(M,L)*, the suitability of *tp(I)* for the insert operation,
(2) For each column *c* of each target table $tt \in TT$, data taken from attribute *a* in *tp(I)* that generates *c* according to *Map(M,L)*, or retrieved from the database,

(3) For each table *tt*, the ordered list of manipulation operations (insert, update or select) to maintain data integrity in *TT*,
(4) Other additional messages, specific of the procedure, where applicable.

The algorithm DMP, included below, describes this procedure (Definition 2):

```
Algorithm DMP
Input: A tuple tp(I) to insert, the conceptual-logical data model mapping between M
  and L Map(M,L), and a set TT of target tables
Output: Database statements and messages
suitable = Analysis (tp(I), Map(M,L))
If (tp(I) is not suitable due to absence of value for any key attribute)
  generateMessage(Error, AKA)
  Abort
Else If (tp(I) is not suitable due to attribute does not correspond with any column)
  generateMessage(Warning,AWC)
End If
Foreach target table tt ∈TT
  Foreach c ∈ Cols(tt)
    data = FindData (c, tt, tp(I), Map(M,L))
    row = AddPair(c, data)
  End Foreach
  GenerateStatement (tt, row)
End Foreach
```

First, it analyses the tuple *tp(I)* (function *Analysis*) to determine its suitability:

- It contains an attribute-value pair for each key attribute of *I*. If this is not the case, DMP raises (in the function *GenerateMessage*) the error message “Absence of value for a key attribute” (AKA) because there is no value for primary keys at a conceptual model level; the insert operation cannot be executed, aborting the process and invalidating any previous operation on any table.
- Each attribute in *tp(I)* generated one or more columns in the database. Otherwise, the function *GenerateMessage* raises the warning message “Attribute does not correspond with any column” (AWC) to inform about a possible loss of information because values of those attributes will be not stored in the database.

Then, it processes each column *c* of each target table *tt*, assigning data to it through the function *FindData* and adding column-data pairs to the row to insert into *tt* (function *AddPair*). *FindData* will be defined within each scope depending on the item (an entity, a relation or multiple relations) and the content of the tuple. After all columns are processed, the function *GenerateStatement* generates the statements to be executed against the database.

4. Insert a tuple into an entity

The simplest case of insert operations at a conceptual model level is to insert a tuple into an entity. Next, DMR and DMP are defined specifically for this.

Definition 3. *(*Data Manipulation Rule for inserting a tuple into an entity, DMR-IE).- Definition 1 is applied where item *I* is an entity *e*∈*Ents (M)*. DMR-IE determines the set of target tables *TT*⊆*Tables(L)* generated from *e*. Each *tt*∈*TT* is well-modelled if ∀*pk*∈*PK(e)*, ∃*k*∈*Key(tt)* / an association *ka-kc* in *Map(M,L)* exists between *pk* and *k*.

Note: Each key attribute of *e* corresponds with a key column of *tt*, and non-key attributes of *e* could correspond with key or non-key columns, or not be in *tt*.

Definition 4. *(Data Manipulation Procedure for inserting a tuple into an entity, DMP-IE)*.- Definition 2 is applied where item *I* is an entity *e*∈*Ents (*M*)*. DMP-IE sets, for each column *c* of each target table *tt*∈*TT*, data taken exclusively from pairs *(*a_i*,*v_i*)* in *tp(e)*.

The general algorithm DMP (Section 3.2) is applied here but *FindData* is specialized for inserting a tuple in an entity:

```
Function FindData
Input: A column c, a target table tt, a tuple tp(e) to insert, the conceptual-logical data
  model mapping between M and L Map(M,L)
Output: data for c
If tp(e) has value v for attribute a corresponding to c      case 1
R   eturn v
Else If c∈Key(tt)                                            case 2
G   enerateMessage(Error,AKC)
A   bort
Else                                                         case 3
G   enerateMessage(Warning,ADC)
R   eturn null
End If
```

This function considers three situations:

- A column *c* is generated from attribute *a* that is in the tuple *tp(e)*(case 1): *FindData* returns as data the value *v* of the pair *(a,v)* in *tp(e)*.
- A key column *c* is generated from attribute *a* that is not in the tuple *tp (e)* (case 2): *GenerateMessage* raises the error message “Absence of data for a key column” (AKC) because it is not possible to insert rows in the table without data for any key column. The insert operation cannot be carried out and the process will abort and any previous operation on any table will be invalidated.
- A non-key column *c* is generated from attribute *a* that is not in the tuple *tp(e)* (case 3): *FindData* returns *null* because no data exists to be inserted for *c*. *GenerateMessage* raises the warning message “Absence of data for a non-key column” (ADC) to inform that the row will be inserted without this column.

Example 1.-. Consider the insertion of a tuple into entity Artist in the conceptual model in the introductory example (Section 3.1).

DMR-IE determines the mapping between conceptual and logical models (that can be seen in Table 1) and the target table *artists_by_first_letter* (generated from entity Artist). The key attribute in Artist (*name*) is mapped to a key column (*artist_name*) of *artists_by_first_letter*, so this table is well-modelled.

The following examples show different situations in which attribute-value pairs in the tuple change and what DMP-IE produces for each one.

Example 1.1.- Consider the tuple to insert has an attribute-value pair for each attribute of Artist:

<(artist.name, “author11”), (artist.first_letter, “a”), (artist.nationality, “nation11”)>

DMP-IE determines that the tuple is suitable and calls *FindData* for each column of table *artists_by_first_letter* that finds all the data from the tuple (case 1). The result is the row:

<(artist_first_letter, “a”),(artist_name, “author11”), (artist_nationality, “nation11”)>

Finally, DMP-IE generates the statement that inserts that row:

```
INSERT INTO artists_by_first_letter (artist_first_letter, artist_name, artist_nationality)
  VALUES (“a”, “author11”, “nation11”)
```

Example 1.2.- Consider the tuple to insert does not have an attribute-value pair for the non-key attribute *first_letter* of Artist that generated the key column *artist_first_letter*:

<(artist.name, “author12”), (artist.nationality, “nation12”)>

DMP-IE determines that the tuple is suitable but *FindData* is not able to set data to the key column *artist_first_letter* (case 2). The result is the next row that has a placeholder ‘$’ to represent the absence of data for this column:

<(artist_first_letter, $),(artist_name, “author12”), (artist_nationality, “nation12”)>

Although an artist can be inserted into a relational database without the first letter, the table *artists_by_first_letter* requires this data (because it is a key column), so it is not possible to carry out the insertion. DMP-IE generates an error message:

```
Error(AKC): Absence of data for key column artist_first_letter. No insertion is possible
```

Example1.3.- Consider the tuple to insert does not have an attribute-value pair for the non-key attribute *nationality* of Artist that generated the non-key column *artist_nationality*:

<(artist.name, “author13”), (artist.first_letter, “a”)>

Now, *FindData* is not able to obtain data for the non-key column *artist_nationality* (case 3). The result is the next row, with a placeholder ‘$’ for the data of this column:

<(artist_first_letter, “a”),(artist_name, “author13”), (artist_nationality, $)>

In this situation, the algorithm shows a warning message (absence of data for column *artist_nationality*) and generates an insert statement:

```
Warning(ADC): Absence of data for non-key column artist_nationality. Column is not
  inserted. Possible incomplete data stored in table artists_by_first_letter
INSERT INTO artists_by_first_letter (artist_first_letter, artist_name) VALUES (“a”,
  “author13”)
```

5. Insert a tuple into relations

Next, we will deal with inserting a tuple into relations at a conceptual model level, considering a binary relation (Section 5.1) and multiple relations (Section 5.2) that are illustrated with examples.

5.1. Insert a tuple into a binary relation

To define the specific DMR and DMP, we will consider different cardinalities of binary relations (1:1, 1:n and n:m).

Definition 5. (Data Manipulation Rule for inserting a tuple into a binary relation, DMR-IR).- Definition 1 is applied where item I is a relation between entities e_1 and e_2, $R_{\{e1,e2\}}\in$Rels(M). DMR-IR consists of two complementary rules to determine the set of target tables $TT=TT_{Ents}\cup TT_R\subseteq$Tables(L):

DMR-IR.1 determines the set of target tables $TT_{Ents} \subseteq Tables(L)$, generated from e_1 *and* e_2. DMR-IE (Definition 3) is applied to these entities.

DMR-IR.2 determines the set of target tables $TT_R \subseteq Tables(L)$, generated from $R_{\{e1,e2\}}$. Depending on the cardinality of $R_{\{e1,e2\}}$, each $tt_R\in TT_R$ is well-modelled, if:

1:1 relation: $\forall pk_{e1}\in PK(e_1) \vee \forall pk_{e2}\in PK(e_2)$, $\exists k\in Key(tt_R)$ / an association *ka-kc* in *Map(M,L)* exists between pk_{e1} and k or pk_{e2} and k.

1:n relation: $\forall pk_{e2}\in PK(e_2)$ $\exists k\in Key(tt_R)$ / an association *ka-kc* in *Map (M,L)* exists between pk_{e2} (key attribute of detail entity) and k.

n:m relation: $pk\in PK(e_1)\cup PK(e_2)$ $\exists k\in Key(tt_R)$ / an association *ka-kc* in *Map(M,L)* exists between *pk* and *k*.

Note: The rest of the attributes not included (from any entity or relation) may correspond with key or non-key columns, or not be in a $tt\in TT$.

If there is no target table determined by DMR-IE.1 for any of the entities, MDICA generates a warning message which informs about a possible loss of data: absence of target tables for some items in the tuple (ATA).

Definition 6. (Data Manipulation Procedure for inserting a tuple into a binary relation, DMP-IR).- Definition 2 is applied where item I is a relation between entities e_1 and e_2, $R_{\{e1,e2\}}\in$Rels(M). DMP-IR sets, for each column c of each target table tt$\in$TT, data taken from pairs (a_i,v_i) in $tp(R_{\{e1,e2\}})$ or retrieved from a table lookupTable$\in$Tabs(L).

The general algorithm DMP (Section 3.2) is now applied with the specialized *FindData* for inserting a tuple in a relation.

```
Function FindData
Input: a column c, a target table tt, a tuple $tp(R_{\{e1,e2\}})$ to insert, the conceptual-logical
  data model mapping between M and L Map(M,L)
Output: data for c
If $tp(R_{\{e1,e2\}})$ has value v for attribute a corresponding to c        case 1
  Return v
Else
  lookupQuery = CreateQuery (c, $tp(R_{\{e1,e2\}})$, Map(M,L))
  If (lookupQuery is executable)                                        case 4
    GenerateMessage(Information,ADC-S)
    Return data=lookupQuery
  Else
    lookupQuery = RecreateQuery (c, $tp(R_{\{e1,e2\}})$, Map(M,L))
    If (lookupQuery is executable)                                      case 5
      GenerateMessage(Information,ADC-C)
      Return data=lookupQuery
    Else
      If $c\in Key(tt)$                                                 case 2
        GenerateMessage(Error,AKC)
        Exit
      Else                                                              case 3
        GenerateMessage(Warning,ADC)
        Return null
      End If
    End If
  End If
End If
```

FindData will build a query named *LookupQuery*, defined below, which will retrieve data from a table for a column *c* when the data is not present in the tuple but already exists in the database (cases 4 and 5).

Definition 7. (LookupQuery).- Given a tuple $tp(R_{\{e1,e2\}})$ and a row of a target table tt $r(tt)=<(c_1,d_1),\ldots, (c_i,\$_i),\ldots(c_n,d_n)>$ where data for column c_i is unknown, represented by a placeholder $\$_i$. lookupQuery is an statement in the form SELECT c_i FROM lookupTable WHERE φ, where lookupTable$\in$Tabs(L) has the column c_i, and φ is a proposition, which holds for lookupTable, with columns and data retrieved from attribute-value pairs in $tp(R_{\{e1,e2\}})$ or from column-data pairs in r(tt).

This function *FindData* contemplates cases 1, 2 and 3 as inserting a tuple into an entity. Moreover, it considers two more situations when a column *c* is generated from attribute *a* that is not in $tp(R_{\{e1,e2\}})$, for which *lookupquery* is prepared to be executed against the database, obtains data for the column and replaces the placeholder in the row:

- Data for the column *c* can be retrieved from the database with *lookupQuery* (case 4). The function *CreateQuery*: (1) searches *L* and finds a *lookupTable* for which the proposition φ holds, and (2) prepares and returns *lookupQuery*. *GenerateMessage* raises the information message “Absence of data for a column, data might be retrieved from lookupTable executing lookupQuery” (ADC-S) to notify the need of a query to find unknown data, otherwise data integrity cannot be ensured because of the absence of data in some columns that already exists in others.
- Data for the column *c* can be retrieved from the database but *CreateQuery* is not able to prepare *lookupQuery* (case5). The function *RecreateQuery*: (1) searches *Q* looking for a table, named *sourceTable*, that stores data for c_i (column with unknown data), (2) generates a new table, named *remadeTable*, from *sourceTable*, with suitable keys so that the proposition φ holds, and (3) prepares and returns *lookupQuery* that retrieves data from *remadeTable*. In this case, *GenerateMessage* raises the information message “Absence of data for a column, an auxiliary table (remadeTable) might be created and populated from sourceTable, and data would be retrieved from remadeTable executing lookupQuery” (ADC-C) to notify the need to create, populate and query a new table to find unknown data.

In case 5, once *remadeTable* is created, it becomes part of *L*, so in subsequent insert operations, the process will be as in case 4.

Example 2.-. Consider the insertion of a tuple into the relation "releases" between entities Artist and Track in the conceptual model in the introductory example (Section 3.1).

DMR-IR determines a set of target tables considering two complementary rules:

- DMR-IR.1 implies the application of DMR-IE to both entities Artist and Track. No table is generated from entity Track. Therefore, table *artists_by_letter*, generated from entity Artist, is the only target table.
- DMR-IR.2 determines as target tables *tracks_by_artist* and *tracks_by_genre*, generated from the relation "releases". Both tables are well-modelled provided that the relation cardinality is 1:n and the primary key of entity Track (detail entity) is part of the key in both of them.

Different situations with a variety of attribute-value pairs in tuples to insert are shown below.

Example 2.1.- Consider the tuple to insert does not have an attribute-value pair for the non-key attribute *nationality* of Artist (which generated non-key columns *artist_nationality* in the target tables):

<(artist.name, "author21"), (artist.first_letter, "a"), (track.id, "id021"), (track.title, "title21"), (track.genre, "genre21"), (track.duration, 21)>

In this situation, *FindData* is not able to obtain data from the tuple for columns *artist_nationality*. If the artist has been previously inserted, it can retrieve the nationality from a table: *CreateQuery* generates a *lookupQuery* to retrieve data for the column *artist_nationality* from the table *artists_by_first_letter* (case 4). *lookupQuery* is "SELECT artist_nationality from artists_by_first_letter where artist_name="author21" and artist_first_letter="a"". When executing this *lookupQuery*, data retrieved will replace placeholders $ in rows:

artists_by_first_letter: <(artist_first_letter, "a"), (artist_name, "author21"),(artist_nationality,$)>

tracks_by_artist: <(artist_name, "author21"), (track_id, "id21"), (track_title, "title21"), (track_genre, "genre21"), (track_duration, 21), (artist_nationality, $)>

tracks_by_genre: <(track_genre, "genre21"), (track_id, "id21"), (track_title, "title21"), (track_duration, 21), (artist_name, "author21"), (artist_nationality, $)>

Finally, the algorithm shows a warning message due to the absence of tables generated from the entity Track and an information message indicating the need to retrieve data from the database, and it generates statements that ensure data integrity:

```
Warning(ATA): Absence of target tables for entity Track
Information(ADC-S): Absence of data for column artist_nationality.
    Select artist_nationality from table artists_by_first_letter
$ = SELECT artist_nationality FROM artists_by_first_letter WHERE artist_name=
  "author21" and artist_first_letter="a"
INSERT INTO artists_by_first_letter (first_letter, artist_name, artist_nationality)
  VALUES ("a", "author21",$)
INSERT INTO tracks_by_artist (artist_name, track_title, track_id, track_genre,
  track_duration, artist_nationality) VALUES ("author21", "title21", "id21",
  "genre21", 21, $)
INSERT INTO tracks_by_genre (track_genre, track_title, track_id, track_duration,
  artist_name, artist_nationality) VALUES ("genre21", "title21", "id21", 21,
  "author21", $)
```

Example 2.2.- Consider the tuple to insert has attribute-value pairs for all attributes of the entity Track but only one pair for the primary key (attribute *name*) of Artist:

<(artist.name, "author22"), (track.id, "id22"), (track.title, "title22"), (track.genre, "genre22"), (track.duration, 22)>

Now, *FindData* does not find data from the tuple for the key column *artist_first_letter* in table *artists_by_first_letter* or for non-key column *artist_nationality* in every target table. *CreateQuery* does not find any *lookupTable* from which the queries in the form "SELECT artist_first_letter/artist_nationality FROM lookuptable WHERE artist_name="author33"", were executable, although these columns exist in the table *artists_by_first_letter*. *RecreateQuery* creates and populates a new table, *rm_artists_by_first_letter*, that can retrieve the unknown values (case 5). The retrieved data will replace the placeholders $$_i$ in rows:

artists_by_first_letter: <(artist_first_letter, $1), (artist_name, "author22"), (artist_nationality, $2)>

tracks_by_artist: <(artist_name, "author22"), (track_id, "id22"), (track_title, "title22"), (track_genre, "genre22"), (track_duration, 22), (artist_nationality, $2)>

tracks_by_genre: <(track_genre, "genre22"), (track_id, "id22"), (track_title, "title22"), (track_duration, 22), (artist_name, "author22"), (artist_nationality, $2)>

For this situation, together with database statements, the algorithm shows a warning message due to the absence of target tables generated from Track and an information message to notify the need to create, populate and query a new table to maintain data integrity:

```
Warning(ATA): Absence of target tables for entity Track
Information(ADC-C): Absence of data for column artist_first_letter
    Create and populate table rm_artists_by_first_letter from artists_by_first_letter
    Select artist_first_letter from table rm_artists_by_first_letter
Information(ADC-S): Absence of data for column artist_nationality
    Select artist_nationality from table rm_artists_by_first_letter
CREATE TABLE rm_artists_by_first_letter (artist_name PRIMARY KEY,
  artist_first_letter, artist_nationality)
COPY rm_artists_by_first_letter (artist_name, artist_first_letter, artist_nationality)
  FROM artists_by_first_letter (artist_name, artist_first_letter, artist_nationality)
$1 = SELECT artist_first_letter FROM rm_artists_by_first_letter WHERE
  artist_name='author22'
$2 = SELECT artist_nationality FROM rm_artists_by_first_letter WHERE
  artist_name='author22'
INSERT INTO artists_by_first_letter (artist_first_letter, artist_name, artist_nationality)
  VALUES ($1, "author22", $2)
INSERT INTO tracks_by_artist (artist_name, track_title, track_id, track_genre,
  track_duration, artist_nationality) VALUES ("author22", "title22", "id22",
  "genre22", 22, $2)
INSERT INTO tracks_by_genre (track_genre, track_title, track_id, track_duration,
  artist_name, artist_nationality) VALUES ("genre22", "title22", "id22", 22,
  "author22", $2)
```

5.2. Insert a tuple including multiple relations

Tuples to insert at a conceptual model level can include attributes of entities related through more than one relationship. This section includes the definition of the specific DMR considering tuples whose attributes belong to entities related by multiple relations and an example.

Definition 8. (Data Manipulation Rule for inserting a tuple into two or more relations, DMR-IRR).- Definition 1 is applied where item I is a set of two or more relations RR⊆Rels(M) between a set of entities EE⊆Ents (M). DMR-IRR consists of two complementary rules to determine the set of target tables TT= $TT_{bR} \cup TT_{RR} \subseteq$Tables(L):

DMR-IRR.1 determines the set of target tables $TT_{bR} \subseteq Tables(L)$, generated from each binary relation $R_{\{ei,ej\}} \in RR$. DMR-IR (Definition 5) is applied to each $R_{\{ei,ej\}}$.

DMP-IRR.2 determines the set of target tables $TT_{RR} \subseteq Tables(L)$, generated from combinations of chained relations in *RR* with cardinality 1:1, 1:n and n:m. Depending on combinations of the cardinality of relations, each table $tt_{RR} \in TT_{RR}$ is well-modelled if:

Combination of 1:1 relations: $\exists e_i \in EE, \forall pk_{ei} \in PK(e_i)\ \exists k \in Key(tt_{RR})$ / an association *ka-kc* in *Map(M,L)* exists between pk_{ei} and *k*.

Combination of 1:n relations: $\exists e_n \in EE, \forall pk_{en} \in PK(e_n)\ \exists k \in Key(tt_{RR})$ / an association *ka-kc* in *Map(M,L)* exists between pk_{en} and *k*. Next, cases are distinguished depending on the position of the detail entity in the chained relations:

- Case 1:n - 1:n: the detail entity e_n is at the end of the chained relations.
- Case 1:n - n:1: the detail entity e_n is in the middle of the chained relations.
- Case n:1 - 1:n: two detail entities exist, at the beginning e_1 and at the end e_n of the chained relations and both must fulfill the proposition.

  Combination of n:m relations: $\forall e_i \in EE$, $\forall pk \in \cup PK(e_i)$ $\exists k \in Key(tt_{RR})$ / an association *ka-kc* in *Map(M,L)* exists between *pk* and *k*.
  Combination of 1:1, 1:n and n:m relations: an association *ka-kc* in *Map(M,L)* exists between a key column of tt_{RR} and every key attribute of: any entity in 1:1 relations, detail entities in 1:n relations and every entity in n:m relations.

Note: The rest of the attributes not included (from any entity or relation) may correspond with key or non-key columns, or not be in any target table $tt \in TT$.

Moreover, when inserting a tuple into a set of relations:

- MDICA generates warning messages informing about a possible loss of data if there is no target table determined by DMR-IRR.1 for any of the binary relations: absence of target tables for some items in the tuple (ATA).
- DMP-IR (Definition 6) is applied where item *I* is a set of two or more relations.

Example 3.-. Consider the insertion of a tuple into the relations ("releases" and "features") between entities Artist, Track and Playlist at the conceptual model level in the introductory example (Section 3.1).

DMR-IRR determines target tables considering two complementary rules:

- DMR-IRR.1 implies the application of DMR-IR to relations "release" and "features" that, recursively, implies the application of DMR-IE to entities Artist, Track and Playlist. No table is generated from entities Track or Playlist or from the relation "features". Table *artists_by_first_letter* (generated from entity Artist) and tables *tracks_by_artist* and *tracks_by_genre* (generated from relation "releases") are determined as target tables.
- DMR-IRR2 determines as a target table *tracks_in_playlist*, generated from the relations chained "releases" and "features", a combination of a 1:n relation (*Artist R Track*) and an n:m relation (*Track R Playlist*), respectively. The table is well-modelled because the key column *track_id* was generated from the primary key of Track (detail entity in the 1:n relation) and *playlist_id* was generated from the primary key of Playlist (Track and Playlist entities in the n:m relation).

6. Validation

In order to evaluate MDICA which, given a tuple to insert into a conceptual model, generates database statements and messages with the goal of maintaining data integrity in a column-oriented database, the following research questions are established:

RQ1: Given an insert operation at a conceptual model level, is it always possible to insert data at a logical model level? If not, what are the causes of this situation?

RQ2: What is the impact of an insert operation at a conceptual model level on the logical model in terms of the number of tables affected to maintain data integrity?

RQ3: How many database statements must be executed for each insert operation at the conceptual model level in order to maintain data integrity in the database?

RQ4: Is it always possible to ensure that data integrity is maintained? If this is not the case, what are the situations identified that can endanger it?

6.1. Experimental subjects

To answer the research questions, we have considered two options to select the experimental subjects: (1) standard benchmarks and (2) applications publicly available with a conceptual model.

Yahoo Cloud Serving Benchmark (YCSB) [34] has become the de-facto benchmark, designed by [35] to compare the performance of data stores and used for measuring performance, scalability, elastic speedup, throughput and latency [35–37] of different NoSQL databases. Since the logical model of YCSB only contains one table on which operations such as read or insert are executed, it is not suitable for the goals of the MDICA experimentation.

Therefore, we have searched for other case studies used in different works related to the design of Cassandra databases and with a variety of tables generated from items in the conceptual models (one entity, one or more relations, and relations with different cardinality). The selected case studies are:

- Digital Library Portal, used by Chebotko et al. [20] to illustrate the data modeling methodology. It is an application that features a collection of digital artifacts (papers, posters…) which appeared in various venues. Registered users can leave their feedback for venues and artifacts in the form of reviews, likes or ratings.
- Hotel reservations is used by Carpenter and Hewitt [15] to show how to design data models for Cassandra. It is a sample application that includes hotels, guests, the rates and availability of rooms, and reservations booked for guests. It also maintains a collection of "points of interest" near hotels.
- Digital music store (the introductory example in Section 3.1) is used as a tutorial intended for programmers interested in learning about Cassandra [33] and it covers the techniques used to create databases and tables. It is a Java web application that manages a collection of music files.

Each case study provides both the conceptual and the logical models. Table 2 displays information about the models:

- Conceptual models: items (entity or relation with its cardinality 1:n or n:m), their name and the number of key and non-key attributes (columns "#PK" and "#nPK").
- Logical models: tables, the items that generate them, their name (columns "From Item/s" and "From Name") and their number of key and non-key columns (columns "#Key" and "#nKey"). If a table is generated from more than one relation, column "From Item/s" is "multiple".

In short, the total number of items and tables are, respectively, 10 and 9 for Digital Library, 13 and 9 for Hotel Reservations, and 7 and 5 for Music Store.

6.2. Test cases design

For the evaluation of MDICA, we have generated for each case study a set of insert operations. Each operation will be a test case. The test cases have been systematically designed applying the classification-tree method [38]. We have regarded MDICA under two relevant aspects, named classifications: where it inserts (classification based on the item to insert) and what it inserts (classification based on the attribute-value pairs in the tuple to insert). For each classification, we have identified different classes:

- Where it inserts (item at a conceptual model level):
  - Entity: insertion in an entity.
  - Relation: insertion in a relation, which is subdivided into three classes depending on the cardinality: 1:1, 1:n and n:m.

Table 2
Conceptual and logical models used in the evaluation.

	Conceptual data model				Logical data model				
Case Study	Item	Name	# PK	# nPK	Table	From Item/s	From Name	# Key	# nKey
Digital Library	entity	artifact	1	3	artifacts	1:n	featuresDA	1	5
		review	1	3	artifacts_by_author	1:n	featuresDA	3	4
		user	1	3	artifacts_by_venue	1:n	featuresDA	3	3
		venue	2	3	ratings_by_artifact	1:n	featuresR	1	2
	1:n	featuresDA	3	6	experts_by_artifact	n:m	likes	3	3
		featuresR	2	6	users_by_artifact	n:m	likes	2	3
		posts	2	7	venues_by_user	n:m	likesV	3	3
	n:m	likes	2	6	artifacts_by_user	multiple	likes-featuresDA	3	3
		likesR	2	6	reviews_by_user	multiple	post-featuresR	3	5
		likesV	3	6					
Hotel Reservations	entity	amenity	1	1	guests	entity	guest	1	5
		guest	1	3	hotels	entity	hotel	1	3
		hotel	1	3	hotels_by_poi	n:m	is_near	2	3
		poi	1	2	pois_by_hotel	n:m	is_near	1	2
		reservation	1	3	amenities_by_room	multiple	has-offers	3	1
		room	1	1	available_rooms_by_hotel_date	multiple	has-is_available	3	1
		room_availability	1	1	reservations_by_confirmation	multiple	has-holds-is_for	2	4
	1:n	has	2	4	reservations_by_guest	multiple	has-holds-is_for	3	4
		holds	2	4	reservations_by_hotel_date	multiple	has-holds-is_for	3	3
		is_for	2	6					
	n:m	is_available	2	2					
		is_near	2	5					
		offers	2	2					
Music Store	entity	artist	1	2	artists_by_first_letter	entity	artist	2	1
		playlist	1	1	playlists_by_user	1:n	creates	2	1
		track	1	3	tracks_by_artist	1:n	releases	3	3
		user	1	1	tracks_by_genre	1:n	releases	3	3
	1:n	creates	2	2	tracks_in_playlist	multiple	releases-features	4	3
		releases	2	5					
	n:m	features	2	4					

  - Multiple relations: insertion of a tuple of two or more adjacent relations. In order to avoid a combinatorial explosion, there will only be one class for each group of relations which generated a table in the logical model.
- What it inserts (attribute-value pairs in the tuple):
  - *: Every attribute of the item to insert has a value in the tuple.
  - PK: Only key attributes have a value; the rest of the attributes are not in the tuple.
  - -attr: A non-key attribute of the item has no value in the tuple. There will be a class for each non-key attribute.
  - -PK: A key attribute of the item has no value. There will be a class for each key attribute.

We have combined each item in the conceptual model (in the first classification) with each class in the second classification, resulting in 289 test cases in total (118 for Digital Library, 118 for Hotel Reservations, and 53 for Music Store). For each case study and item, Table 3 displays the number of test cases for each of the combinations.

Once we have generated the test cases, we apply the rules and procedures defined in Sections 4 and 5 to each one. As previously described, after obtaining the mapping between conceptual and logical models, MDICA identifies the target tables (listed in Table 3, column “Target Tables”), generated from the items of the tuple to insert, and determines the database statements that should be executed against the database to maintain data integrity and the messages shown to users. The analysis of the results of these executions is detailed in the following sections.

6.3. Analysis of the insertion operations at a conceptual model level (RQ1)

To answer RQ1, we ran the test cases and inspected for each one if insertions were generated at a logical model level. We found that 45.0% of the test cases produced insertions into databases, and the remaining 55.0% did not.

Table 4 displays the number of test cases (289 in total) divided into those that inserted data into the database without generating error messages (130 test cases) and those that generated an error message without inserting rows into the databases (159 test cases).

The high number of the latter is due to the strategy for their design: in 86 test cases, the insert operation did not impact on any target tables (columns “ATT”, Absence of Target Tables); in 68 test cases, tuples did not have values for key attributes (columns “AKA”, Absence of value for a Key Attribute); and in 5 test cases, tuples did not have values for any key column and they could not be retrieved from the database either (columns “AKC”, Absence of data for a Key Column). For each of these test cases, MDICA generated the appropriate error message, described in Sections 3.2 and 4, depending on the reason why they did not insert any rows.

Answering RQ1, some situations do not enable data insertions into the conceptual model or the database due to a lack of data for key attributes or for key columns or an absence of tables where to insert. MDICA is a first help for developers since it can detect these situations and provide information (to add new tables or modify the tuple with additional attribute-value pairs) so that the insertion in both models is feasible.

6.4. Analysis of target tables impacted by an insertion (RQ2)

To answer RQ2, we analyze the target tables in each test case that did not generate an error message.

All tables in the logical models (listed in Table 2) are impacted by some test case as Table 3 displays. For those test cases that did not impact on any table, for which an ATT error message was generated, target tables were labelled as ‘-’. For relations, more than half of the insert operations impacted on more than one table. Moreover, the maximum number of target tables was reached when inserting a tuple of multiple relations (6 for Digital Library, 5 for Hotel Reservations and 4 for Music Store), accounting for more than 50% of the tables in each case study.

Answering RQ2, to insert a tuple at a conceptual model level impacts

Table 3
Test cases to evaluate MDICA and target tables impacted by each test case.

Case Study	Item	Name	Attribute-value Pairs					Target Tables
			*	PK	-attr	-PK	Total	
Digital Library	**entity**	artifact	1	1	3	1	**6**	-
		review	1	1	3	1	**6**	-
		user	1	1	3	1	**6**	-
		venue	1	1	3	2	**7**	-
	1:n	featuresDA	1	1	6	3	**11**	artifacts_by_venue, artifacts_by_author, artifacts
		featuresR	1	1	6	2	**10**	ratings_by_artifact
		posts	1	1	7	2	**11**	-
	n:m	likes	1	1	6	2	**10**	users_by_artifact, experts_by_artifact
		likesR	1	1	6	2	**10**	-
		likesV	1	1	6	3	**11**	venues_by_user
	multiple	likes-featuresDA	1	1	9	4	**15**	artifacts_by_user artifacts_by_venue, artifacts_by_author, artifacts, users_by_artifact, experts_by_artifact
		posts-featuresR	1	1	10	3	**15**	ratings_by_artifact, reviews_by_user
Total Digital Library			**12**	**12**	**68**	**26**	**118**	n/a
Hotel Reservations	**entity**	amenity	1	1	1	1	**4**	-
		guest	1	1	3	1	**6**	guests
		hotel	1	1	3	1	**6**	hotels
		poi	1	1	2	1	**5**	-
		reservation	1	1	3	1	**6**	-
		room	1	1	1	1	**4**	-
		room_availability	1	1	1	1	**4**	-
	1:n	has	1	1	4	2	**8**	hotels
		holds	1	1	4	2	**8**	-
		is_for	1	1	6	2	**10**	guests
	n:m	is_available	1	1	2	2	**6**	-
		is_near	1	1	5	2	**9**	hotels, hotels_by_poi, pois_by_hotel
		offers	1	1	2	2	**6**	-
	multiple	has-holds-is_for	1	1	10	4	**16**	guests, hotels, reservations_by_confirmation, reservations_by_guest, reservations_by_hotel_date
		has-is_available	1	1	5	3	**10**	available_rooms_by_hotel_date, hotels
		has-offers	1	1	5	3	**10**	amenities_by_room, hotels
Total Hotel Reservations			**16**	**16**	**57**	**29**	**118**	n/a
Music Store	**entity**	artist	1	1	2	1	**5**	artists_by_first_letter
		playlist	1	1	1	1	**4**	-
		track	1	1	3	1	**6**	-
		user	1	1	1	1	**4**	-
	1:n	creates	1	1	2	2	**6**	playlists_by_user
		releases	1	1	5	2	**9**	artists_by_first_letter, tracks_by_artist, tracks_by_genre
	n:m	features	1	1	4	2	**8**	-
	multiple	releases-features	1	1	6	3	**11**	artists_by_first_letter, tracks_by_artist, tracks_by_genre, tracks_in_playlist
Total Music Store			**8**	**8**	**24**	**13**	**53**	n/a
Total			**36**	**36**	**149**	**68**	**289**	n/a

Table 4
Test cases that produced insertions in databases and test cases that generated error messages.

	Insertions without error messages		Insertions with error messages						Total
			ATT		AKA		AKC		
Case Study	#	%	#	%	#	%	#	%	#
Digital Library	54	45.8	37	31.4	26	22.0	1	0.8	118
Hotel Reservations	55	46.6	32	27.1	29	24.6	2	1.7	118
Music Store	21	39.6	17	32.1	13	24.5	2	3.8	53
Total	**130**	**45.0**	**86**	**29.8**	**68**	**23.5**	**5**	**1.7**	**289**

on more or less tables, depending on those generated from items in the tuple. The more complex the tuple, in terms of the number of items it contains, the greater the number of target tables and the more error-prone, by forgetting to insert some of them. MDICA identifies the target tables regardless of the complexity of the tuple and decreases the probability of making mistakes in their selection.

6.5. Analysis of database statements (RQ3)

To answer RQ3, we analyze the database statements generated by DMPs to ensure data integrity in a database for test cases that did not generate error messages.

Table 5 displays information about test cases, the number of tables impacted and CQL statements generated for each test case. Database statements are split into INSERT, SELECT and CREATE&COPY and, for each one, columns "#", "%" and "Avg" display the number of statements, percentage of the total and average per test case, respectively.

For 130 test cases, MDICA generated 469 CQL statements in total (an average of 3.6 CQLs per test case) that included mostly INSERT statements (332, 70.8% of the total) but also SELECT (111, 23.7%) and CREATE&COPY (26, 5.5%).

An insert operation at a conceptual level may imply several INSERT statements at a logical level, as many as the number of target tables impacted, with an average of 2.6 necessary for each test case.

Inserting into an entity does not generate SELECT or CREATE&COPY because it implies the creation of a new instance that does not exist in the database. For inserting into relations, SELECT (average 0.9 per test case) and CREATE&COPY (average 0.2) statements were produced

Table 5
Database statements generated by DMPs.

Case Study	Item	Name	#Target Tables	#Test Cases	INSERT			SELECT			CREATE& COPY			Total	
					#	%	Avg	#	%	Avg	#	%	Avg	#	Avg
Digital Library	**1:n**	featuresDA	3	8	24	80.0	3.0	6	20.0	0.8	0	0.0	0.0	30	3.8
		featuresR	1	8	8	100.0	1.0	0	0.0	0.0	0	0.0	0.0	8	1.0
	n:m	likes	2	8	16	61.5	2.0	6	23.1	0.8	4	15.4	0.5	26	3.3
		likesV	1	8	8	44.4	1.0	6	33.3	0.8	4	22.2	0.5	18	2.3
	multiple	likes-featuresDA	6	11	66	76.7	6.0	16	18.6	1.5	4	4.7	0.4	86	7.8
		posts-featuresR	2	11	22	91.7	2.0	2	8.3	0.2	0	0.0	0.0	24	2.2
Total Digital Library				**54**	**144**	**75.0**	**2.7**	**36**	**18.8**	**0.7**	**12**	**6.3**	**0.2**	**192**	**3.6**
Hotel Reservations	**entity**	guest	1	5	5	100.0	1.0	0	0.0	0.0	0	0.0	0.0	5	1.0
		hotel	1	5	5	100.0	1.0	0	0.0	0.0	0	0.0	0.0	5	1.0
	1:n	has	1	6	6	50.0	1.0	6	50.0	1.0	0	0.0	0.0	12	2.0
		is_for	1	8	8	57.1	1.0	6	42.9	0.8	0	0.0	0.0	14	1.8
	n:m	is_near	3	7	21	63.6	3.0	10	30.3	1.4	2	6.1	0.3	33	4.7
	multiple	has-holds-is_for	5	12	60	76.9	5.0	18	23.1	1.5	0	0.0	0.0	78	6.5
		has-is_available	2	5	10	76.9	2.0	3	23.1	0.6	0	0.0	0.0	13	2.6
		has-offers	2	7	14	63.6	2.0	8	36.4	1.1	0	0.0	0.0	22	3.1
Total Hotel Reservations				**55**	**129**	**70.9**	**2.4**	**51**	**28.0**	**0.9**	**2**	**1.1**	**0.0**	**182**	**3.3**
Music Store	**entity**	artist	1	2	2	100.0	1.0	0	0.0	0.0	0	0.0	0.0	2	1.0
	1:n	creates	1	4	4	50.0	1.0	2	25.0	0.5	2	25.0	0.5	8	2.0
		releases	3	7	21	60.0	3.0	10	28.6	1.4	4	11.4	0.6	35	5.0
	multiple	releases-features	4	8	32	64.0	4.0	12	24.0	1.5	6	12.0	0.8	50	6.3
Total Music Store				**21**	**59**	**62.1**	**2.2**	**24**	**25.3**	**1.1**	**12**	**12.6**	**0.6**	**95**	**4.5**
Total				**130**	**332**	**70.8**	**2.6**	**111**	**23.7**	**0.9**	**26**	**5.5**	**0.2**	**469**	**3.6**

when it was necessary to retrieve data from tables to complete the rows to insert. In the evaluation, the same CREATE&COPY statement was generated for different test cases, however, in a real situation, once a new table is created, it becomes part of the database so it can be queried without repeating its creation. Therefore, the execution of CREATE&-COPY statements will be occasional, less frequent than in the case studies.

Answering RQ3, MDICA automatically generates the set of database statements that ensures data integrity in databases, which will require an INSERT statement for each target table generated from the items in the tuple, plus the appropriate SELECT statements for retrieving data of columns that the tuple does not have but that were inserted in other tables previously. Although less frequently than the other statements, there may be situations that will also require CREATE&COPY statements to add new tables to query data when it cannot be directly retrieved from the existent tables. Manually building the suitable set of statements for an insert operation may become tedious and error-prone for developers, therefore the use of MDICA is a considerable benefit in maintaining data integrity.

6.6. *Analysis of messages (RQ4)*

To answer RQ4, we analyze messages (error, warning and information) generated by MDICA for test cases.

To answer RQ1, we analysed test cases and identified those that made the insert operation impossible and for which error messages were generated. They indicated the need to create new tables to store the values or add other attribute-value pairs in the tuple to insert.

Table 6 displays the generated messages divided into information and warning messages. Columns "#" are the number of messages and "Avg" are the average number of messages for each test case.

Information messages (195 in total, average 1.5 messages per test case), described in Section 5.1, reported the absence of values in the tuple for some columns but data could be extracted from tables using SELECT statements (ADC-S: 169 messages, average 1.3) and CREATE&COPY statements (ADC-C: 26 messages, average 0.2).

The most important messages are warnings (832 in total, average 6.4), described in Sections 3.2, 4 and 5, because they give additional information about the insert operation that may endanger data integrity, although it can be carried out.

AWC (attribute does not correspond with any column) (363 messages, average 2.8) and ATA (absence of target tables for some items in the tuple), (334, average 2.6) warns the developer the inability of the logical model to store values of the tuple which could cause a potential loss of information. The developer should analyze the logical model and decide whether or not to add new tables or columns.

Other warning messages showed discrepancies between the expected and the actual design of databases: TNW-C (table not well-modelled because a column was not generated from any attribute) (59, average 0.5) and TNW-K (table not well-modelled because of the absence of a key column generated from a key attribute) (55, average 0.4). They could provoke that columns never store data, unnecessary repeated data or incorrect outputs of queries. The developer could avoid them changing columns or keys in tables.

ADC (absence of data for a non-key column) (21, average 0.2) alerted the user to the generation of gaps of data in columns. To avoid them, the developer should add attribute-value pairs to the tuple to store data in those columns.

Answering RQ4, in addition to error messages (treated in Section 6.3) and information messages, warnings provide particularly valuable knowledge for the maintenance of data integrity. MDICA identifies situations that can endanger it and generates the right messages. Using

Table 6
Information and warning messages generated by MDICA.

Case Study	#Test Cases	Information Messages						Warning Messages											
		ADC-S		ADC-C		Total		AWC		ATA		TNW-C		TNW-K		ADC		Total	
		#	Avg	#	Avg	#	Avg	#	Avg	#	Avg	#	Avg	#	Avg	#	Avg	#	Avg
Digital Library	54	60	1.1	12	0.2	72	1.3	306	5.7	141	2.6	6	0.1	0	0.0	7	0.1	460	8.5
Hotel Reservations	55	71	1.3	2	0.0	73	1.3	32	0.6	154	2.8	53	1.0	55	1.0	13	0.2	307	5.6
Music Store	21	38	1.8	12	0.6	50	2.4	25	1.2	39	1.9	0	0.0	0	0.0	1	0.0	65	3.1
Total	**130**	**169**	**1.3**	**26**	**0.2**	**195**	**1.5**	**363**	**2.8**	**334**	**2.6**	**59**	**0.5**	**55**	**0.4**	**21**	**0.2**	**832**	**6.4**

these messages, where appropriate, developers will be able to fix faults to ensure data integrity by including more values in the tuples, creating new tables, adding new columns, or making other changes in the databases.

6.7. *Threats to validity*

We evaluated MDICA with three case studies generating, systematically, a set of test cases for which it identified the impacted target tables, generated database statements to ensure data integrity and produced messages informing of those situations that could endanger the integrity. However, there are several threats to the validity of our experiments that may limit the ability to generalize the results. In this section we discuss threats to external, internal, construct and conclusion validity.

Threats to external validity.- The experimental subjects were drawn from a research paper, a tutorial and a book where the design of Cassandra databases was illustrated. These threats include the degree to which the subjects represent other case studies or real applications because they may appear rather limited. However, we consider them representative of partial or entire applications as they are examples in guidelines for many developers. Another threat is whether test cases are representative of real practice. They were systematically generated, based on the item to insert and the content of the tuple, and represent a large variety of insertions at a conceptual level that impacted on all the tables of each case study. These threats could be reduced by considering more experiments concerning other subjects and real insert operations, for which both the number of test cases that do not produce insertions and messages would probably be reduced. However, we consider the approach as appropriate for a complete evaluation of the method.

Threats to internal validity.- MDICA generates database statements to maintain data integrity in databases. We inspected them carefully and found that they maintain data integrity and are coherent in all case studies. As part of an ongoing research [39], an oracle is being developed to automatically determine if, starting from a consistent state of a database, the resultant state after executing database statements maintains data integrity.

Threats to construct validity.- In two of the three case studies, the logical models were designed following the modeling process that MDICA leverages. Therefore, a threat is that the experimental subjects contain the features that MDICA expects. To mitigate this threat, we included the third case study, "Hotel Reservations", designed according to a query-driven modeling process without considering the conceptual model.

Threats to conclusion validity.- We used as metrics the number of target tables, database statements and messages generated. Regarding the target tables, all tables in each case study were impacted by some test case. There is an INSERT statement for each target table and, when necessary, SELECT and CREATE&COPY statements. In our opinion, the messages generated seem to be sufficient, clear and appropriate to warn about potential data integrity faults although we do not have feedback from professional users. To mitigate this threat, messages should be validated by developers with different levels of experience.

7. MDICA extensions

In this section, we describe how to handle different issues of the approach not detailed in this work: how to adapt MDICA to other column-oriented databases and how to address delete and update operations.

7.1. *Handling other column-oriented databases*

The main difference between Cassandra and other logical models of other column-oriented databases is the design of the keys in the tables. In Cassandra, rows are identified by a compound primary key that may involve multiple columns, whereas in others such as Apache HBase [40] or Google Cloud Bigtable [41], rows are uniquely identified by their row key and sorted lexicographically by it.

Regarding MDICA, the mapping between conceptual and logical models will be the same except for the key composition in the tables. Due to the fact that HBase and Bigtable tables only have a row key, it will be necessary to map the attributes that generate the row key for each table. Thus, MDICA will be able to identify tables affected by the insertion of tuples.

7.2. *Delete and update operations*

To support delete and update operations, new data manipulation rules and procedures (DMR and DMP, respectively) must be defined considering the following situations, for which MDICA must provide statements or messages:

- Delete a tuple from an entity or a relation: statements to delete rows in tables generated from this entity or relation and, for each table generated from relations between this entity or relation and another:
  - to delete rows if there exists a key column generated from any attribute, otherwise,
  - to delete data of the columns generated from the attributes in the tuple.
- Delete values of attributes from an entity or relation: statements to delete data of the non-key columns generated from these attributes. If any attribute generated any key column, an error message should be produced because that column cannot have a null value.
- Update values of attributes from an entity or relation: statements to update data of the columns generated from these attributes.
- For all cases, MDICA must also consider the conditions included in these operations which specify the rows to change.

8. Related work

Most research oriented to improve the quality of databases or database applications focuses on relational databases and supposes that data integrity is ensured because relational database systems have their own mechanisms for the referential integrity control that guarantees it, avoiding data duplicity and a loss of information. Therefore, said research improves the quality of test databases by means of generating new databases [42,43] or reducing large ones [44], generating program-input for database applications [45] and other aspects such as determining the correctness of the schema [46] or the correctness of SQL statements considering the conceptual data model [47]. Moreover, mutation testing has been applied to validate different studies [48], even to NoSQL databases [49], and tools have been developed to support experiments evaluating testing techniques [50,51].

To achieve high availability, scalability and performance, NoSQL databases do not generally ensure strong consistency in all situations of data management and do not support transactions. Several transactional services have been developed [13] and tested [12] to detect anomalies in data consistency. Our approach is complementary to them: transactional services ensure consistency of data after a transaction and MDICA ensures data integrity after a change in all the tables where they are repeated.

Within the scope of data integrity, there is research about how malicious attacks can affect it in cloud environments [7,8]. Our objective is to ensure data integrity when inserting tuples whose data must be stored in a column-oriented database such as Cassandra rather than to avoid inserting information from external attacks. From the point of view of column-oriented databases, research on integrity is based on assuring the physical integrity when a row is replicated throughout all the clusters [10,11] or the completeness and correctness of data retrieved by queries [52]. However, MDICA is focused on data integrity, ensuring the integrity of repeated data in different tables (if any data changes in any column, that data will be updated in each repetition of the database) and the integrity of data between conceptual and logical models.

The official Cassandra developer team has proposed solving the problem of maintaining data integrity by the use of *materialized views* [53]. Materialized views are table-like structures that ensure data integrity automatically on the server side. Data is stored in a base table, but it can be retrieved from different views. Any insertion or change is executed against the base table and immediately updated for every view. The main drawbacks of materialized views are (1) they must have the same key columns as the base table and can only have a new key column from the rest of the columns and (2) they can only be created from a single base table rather than from a join of multiple tables as in relational databases, so data in materialized views can be accessed in limited ways. In our work, it does not matter if tables have a different or similar design: provided that the table is generated from an item of the tuple to insert, this table will be impacted by the insert operation and an INSERT statement will thus be generated for it.

Some problems related to the repeated data in several tables could be reduced if CQL select statements include join operators. In [54], joins are implemented by modifying the source code of Cassandra 2.0. However, it has not yet been included in Apache Cassandra.

One of the most important inputs in MDICA is the conceptual data model. However, it is a common practice to design NoSQL databases in general without an explicit conceptual model. To address this limitation, inferring normalized schemas that represent entities and relations for document databases is proposed in [55] and [56], although the research could be applied to other NoSQL databases. If we had to deal with situations where the conceptual model was not provided, we would include a previous task in which we inferred it from the logical model.

The conceptual model is not only an important element in ensuring data integrity but also in modeling a logical data model [57]. Starting from a conceptual model, it is automatically transformed into a NoSQL schema [22] that can serve the queries with minimal cost [18], it is mapped to heterogeneous datastores [23], and MongoDB [17] and HBase [19] databases are designed. In [24], a tool is designed to generate implementations for Cassandra and MongoDB from the same conceptual data model. Chebotko et al. [20] and Mior et al. [21] focus their approach on generating logical and physical Cassandra models from the application's conceptual data model and supported queries, which have been leveraged for this work.

9. Conclusions and future work

This paper presents a method, MDICA, that helps to maintain data integrity when data is inserted in column-oriented databases. It takes a tuple to insert at a conceptual model level and generates the database statements (both data management and data definition statements) needed to ensure the integrity in the database. Moreover, it produces messages (error, warning and information) which can guide developers in decisions about how to deal with data integrity in their applications. Although this method is implemented for Cassandra, it can be particularised for other column-oriented databases with small changes related to the map between the conceptual model and the logical model of these databases.

MDICA was validated by three case studies. Results showed that the method is automatically able to determine the tables impacted by an insert operation, generate the appropriate statements and warn about potential problems that could endanger it.

Developers can benefit from the use of this method, saving time and reducing mistakes. They can use the generated database statements to include them in their source code so that they do not forget to update any table, retrieve adequate data from other tables to complete rows to insert, and avoid making mistakes that endanger data integrity. For previously developed applications, they may compare statements with programmed procedures to facilitate the fault detection in the code and repair it when necessary. Considering the generated messages, developers will have an early warning against defects that can be prevented. They will be able to detect if values in the tuple are sufficient to insert successfully into the database, if new tables, columns or keys are necessary to store data satisfactorily or if there are columns that never store them.

Future work will be focused on different lines. One area of interest is to ensure data integrity when changes are produced at a logical model level, that is when a row in a table is inserted. We also plan to extend the approach to support delete and update operations, both at a conceptual level and logical level. Related to this, an important issue is to infer the conceptual model from a logical model to be able to apply MDICA when the conceptual model has not been considered previously.

CRediT authorship contribution statement

María José Suárez-Cabal: Conceptualization, Methodology, Formal analysis, Writing – original draft. **Pablo Suárez-Otero:** Software, Validation, Data curation. **Claudio de la Riva:** Visualization, Writing – review & editing, Project administration. **Javier Tuya:** Writing – review & editing, Supervision, Investigation, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by projects TIN2016–76956-C3–1-R funded by the Spanish Ministry of Economy and Competitiveness and PID2019–105455GB-C32 funded by MCIN/ AEI/10.13039/ 501100011033.

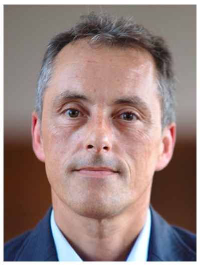
References

[1] N. Leavitt, Will NoSQL databases live up to their promise? Computer 43 (2010) 12–14, https://doi.org/10.1109/MC.2010.58 (Long. Beach. Calif).
[2] D. Pritchett, BASE: an acid alternative, Queue 6 (2008) 48–55, https://doi.org/10.1145/1394127.1394128.
[3] A. Makris, K. Tserpes, V. Andronikou, D. Anagnostopoulos, A classification of NoSQL data stores based on key design characteristics, Procedia Comput. Sci. 97 (2016) 94–103, https://doi.org/10.1016/j.procs.2016.08.284.
[4] A.B.M. Moniruzzaman, S. Hossain, NoSQL database: new era of databases for big data analytics - classification, characteristics and comparison, Int. J. Database Theor. Appl. 6 (2013).
[5] M. Zviran, C. Glezer, Towards generating a data integrity standard, Data Knowl. Eng. 32 (2000) 291–313, https://doi.org/10.1016/S0169-023X(99)00042-7.
[6] E.B. Fernandez, R.C. Summers, C. Wood, Database Security and Integrity, Addison-Wesley Longman Publishing Co., Inc., USA, 1981.
[7] P. Ghazizadeh, R. Mukkamala, S. Olariu, Data integrity evaluation in cloud database-as-a-service, in: Proceedings of the IEEE 9th World Congress Services, 2013, pp. 280–285, https://doi.org/10.1109/SERVICES.2013.40.
[8] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, V. Sassone, Blockchain-based database to ensure data integrity in cloud computing environments, in: Proceedings of the 1st Italian Conference on Cybersecurity, Venice, Italy, 2017.
[9] M. Diogo, B. Cabral, J. Bernardino, Consistency models of NoSQL databases, Futur. Internet 11 (2019) 43, https://doi.org/10.3390/fi11020043.
[10] A. Lakshman, P. Malik, Cassandra - A decentralized structured storage system, ACM SIGOPS Oper. Syst. Rev. 44 (2010) 35–40, https://doi.org/10.1145/1773912.1773922.
[11] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, B. Wong, Understanding the causes of consistency anomalies in apache Cassandra, Proc. VLDB Endow. 8 (2015) 810–813, https://doi.org/10.14778/2752939.2752949.
[12] M.T. González-Aparicio, M. Younas, J. Tuya, R. Casado, Testing of transactional services in NoSQL key-value databases, Futur. Gener. Comput. Syst. 80 (2018) 384–399, https://doi.org/10.1016/j.future.2017.07.004.
[13] V. Padhye, A. Tripathi, Scalable transaction management with snapshot isolation for NoSQL data storage systems, IEEE Trans. Serv. Comput. 8 (2015) 121–135, https://doi.org/10.1109/TSC.2013.47.
[14] J. Pokorny, B. Stantic, Challenges and opportunities in big data processing. Managing Big Data in Cloud Computing Environments, IGI Global, 2016, pp. 1–24, https://doi.org/10.4018/978-1-4666-9834-5.ch001.

[15] J. Carpenter, E. Hewitt, Cassandra: The Definitive Guide, 2nd ed., O'Reilly Media, Inc., 2016.
[16] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the NoSQL world, Comput. Stand. Interfaces 67 (2020), 103149, https://doi.org/10.1016/j.csi.2016.10.003.
[17] G. Zhao, W. Huang, S. Liang, Y. Tang, Modeling MongoDB with relational model, in: Proceedings of the 4th International Conference on Emerging Intelligent Data and Web Technologies, 2013, pp. 115–121.
[18] T. Vajk, L. Deák, K. Fekete, G. Mezei, Automatic NOSQL schema development: A case study, in: IASTED Multiconferences -, Proc. IASTED Int. Conf. Parallel Distrib. Comput. Networks, PDCN 2013 (2013) 656–663. https://doi.org/10.2316/P.2013.795-044.
[19] Y. Li, P. Gu, C. Zhang, Transforming UML class diagrams into HBase based on meta-model, in: Proceedings of the International Conference of Electrical and Electronics Engineering, 2014, pp. 720–724.
[20] A. Chebotko, A. Kashlev, S. Lu, A big data modeling methodology for apache Cassandra, in: Proceedings of the IEEE International Conference on Big Data, 2015, pp. 238–245, https://doi.org/10.1109/BigDataCongress.2015.41.
[21] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, NoSE: schema design for NoSQL applications, IEEE Trans. Knowl. Data Eng. 29 (2017) 2275–2289, https://doi.org/10.1109/TKDE.2017.2722412.
[22] F. Abdelhedi, A.A. Brahim, F. Atigui, G. Zurfluh, UMLtoNoSQL: automatic transformation of conceptual schema to NoSQL databases, in: Proceedings of the IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), 2018, pp. 272–279, https://doi.org/10.1109/AICCSA.2017.76.
[23] G. Daniel, A. Gómez, J. Cabot, UMLto[No]SQL: mapping conceptual schemas to heterogeneous datastores, in: Proceedings of the 13th International Conference on Research Challenges in Information Science, 2019, pp. 1–13.
[24] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla, P. Sánchez, Mortadelo: automatic generation of NoSQL stores from platform-independent data models, Futur. Gener. Comput. Syst. 105 (2020) 455–474, https://doi.org/10.1016/j.future.2019.11.032.
[25] CQL data modeling, CQL for DSE 6.8. https://docs.datastax.com/en/dse/6.8/cql/cql/ddl/dataModelingCQLTOC.html (accessed February 14, 2022).
[26] Guide to apache Cassandra data modelling - instaclustr. https://www.instaclustr.com/resource/6-step-guide-to-apache-cassandra-data-modelling-white-paper/ (accessed February 14, 2022).
[27] V.N. Gudivada, D. Rao, V.V. Raghavan, NoSQL Systems for Big Data Management, in: Institute of Electrical and Electronics, Engineers (IEEE, 2014, pp. 190–197. https://doi.org/10.1109/services.2014.42.
[28] Apache Cassandra, Apache Cassandra Documentation. https://cassandra.apache.org/_/index.html (accessed April 1, 2022).
[29] DB-engines ranking - popularity ranking of database management systems. https://db-engines.com/en/ranking (accessed February 14, 2022).
[30] P. Suárez-Otero, M.J. Suárez-Cabal, J. Tuya, Leveraging conceptual data models to ensure the integrity of Cassandra databases, J. Web Eng. 18 (2019) 257–286, https://doi.org/10.13052/jwe1540-9589.18461.
[31] R. Elmasri, S. Navathe, Fundamentals of Database Systems, 6th ed., Addison-Wesley Publishing Company, USA, 2010.
[32] P.P.S. Chen, The entity-relationship model – toward a unified view of data, ACM Trans. Database Syst. 1 (1976) 9–36, https://doi.org/10.1145/320434.320440.
[33] The Playlist tutorial. https://docs.datastax.com/en/archived/playlist/doc/java/playlistPreface.html (accessed February 14, 2022).
[34] V. Reniers, D. Van Landuyt, A. Rafique, W. Joosen, On the state of NoSQL benchmarks, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, 2017, pp. 107–112, https://doi.org/10.1145/3053600.3053622.
[35] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud computIng, 2010, pp. 143–154, https://doi.org/10.1145/1807128.1807152.
[36] P. Martins, P. Tomé, C. Wanzeller, F. Sá, M. Abbasi, NoSQL comparative performance study, in: Proceedings of the Advances in Intelligent Systems and Computing, 2021, pp. 428–438, https://doi.org/10.1007/978-3-030-72651-5_41, 1366 AISC.
[37] A. Hendawi, J. Gupta, L. Jiayi, A. Teredesai, R. Naveen, S. Mohak, M. Ali, Distributed NoSQL Data Stores: Performance Analysis and a Case Study, in: Proc. - 2018 IEEE Int. Conf. Big Data, Big Data 2018, Institute of Electrical and Electronics Engineers Inc., 2019: pp. 1937–1944. https://doi.org/10.1109/BigData.2018.8622544.
[38] M. Grochtmann, K. Grimm, Classification trees for partition testing, Softw. Testing, Verif. Reliab. 3 (1993) 63–82, https://doi.org/10.1002/stvr.4370030203.
[39] P. Suárez-Otero, M.J. Suárez-Cabal, J. Tuya, Verificación del mantenimiento de la consistencia lógica en bases de datos Cassandra |, in: Proceedings of the Jornadas de Ingeniería del Software y Bases de Datos, Seville, Spain, 2019.
[40] Apache HBase – apache HBase™ Home. https://hbase.apache.org/index.html (accessed February 14, 2022).
[41] Cloud bigtable: noSQL database service, Google cloud. https://cloud.google.com/bigtable/ (accessed February 7, 2022).
[42] M.J. Suárez-Cabal, C. de la Riva, J. Tuya, R. Blanco, Incremental test data generation for database queries, Autom. Softw. Eng. 24 (2017) 719–755, https://doi.org/10.1007/s10515-017-0212-7.
[43] S.A. Khalek, B. Elkarablieh, Y.O. Laleye, S. Khurshid, Query-aware test generation using a relational constraint solver, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, USA, 2008, pp. 238–247, https://doi.org/10.1109/ASE.2008.34. IEEE Computer Society.
[44] J. Tuya, C. de la Riva, M.J. Suárez-Cabal, R. Blanco, Coverage-aware test database reduction, IEEE Trans. Softw. Eng. 42 (2016) 941–959, https://doi.org/10.1109/TSE.2016.2519032.
[45] K. Pan, X. Wu, T. Xie, Program-input generation for testing database applications using existing database states, Autom. Softw. Eng. 22 (2015) 439–473, https://doi.org/10.1007/s10515-014-0158-y.
[46] P. McMinn, C.J. Wright, C.J. McCurdy, G.M. Kapfhammer, Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas, IEEE Trans. Softw. Eng. 45 (2019) 427–463, https://doi.org/10.1109/TSE.2017.2786286.
[47] W.K. Chan, S.C. Cheung, T.H. Tse, Fault-based testing of database application programs with conceptual data model, in: Proceedings of the 5th International Conference on Quality Software (QSIC'05), 2022, pp. 187–196, https://doi.org/10.1109/QSIC.2005.27.
[48] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, Mutating database queries, Inf. Softw. Technol. 49 (2007) 398–417, https://doi.org/10.1016/j.infsof.2006.06.009.
[49] H. Shahriar, S. Batchu, Towards mutation-based testing of column-oriented database queries, in: Proceedings of the ACM Southeast Regional Conference, 2014, pp. 1–6, https://doi.org/10.1145/2638404.2638470.
[50] C. Zhou, P. Frankl, JDAMA: java database application mutation analyser, Softw. Testing Verif. Reliab. 21 (2011) 241–263, https://doi.org/10.1002/stvr.462.
[51] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, SQLMutation: a tool to generate mutants of SQL database queries, in: Proceedings of the 2nd Workshop on Mutation Analysis (Mutation - ISSRE Workshops), 2006, https://doi.org/10.1109/MUTATION.2006.13, 1–1.
[52] G. Weintraub, E. Gudes, F. Kerschbaum, S. Paraboschi, Data integrity verification in column-oriented NoSQL databases. Data and Applications Security and Privacy XXXII, Springer International Publishing, Cham, 2018, pp. 165–181.
[53] Using materialized views, CQL for Cassandra 3.0. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useOverviewMV.html (accessed February 14, 2022).
[54] C. Peter, Supporting the Join Operation in a NoSQL System - Mastering the Internals of Cassandra, Norwegian University of Science and Technology, 2015.
[55] A. Ait Brahim, R. Tighilt Ferhat, G. Zurfluh, Extraction process of conceptual model from a document-oriented NoSQL database, Conf. Knowl. Syst. Eng. KSE 2019 (2019) 1–5. https://doi.org/10.1109/KSE.2019.8919400.
[56] D. Sevilla Ruiz, S.F. Morales, J.G. Molina, P. Johannesson, M.L. Lee, S.W. Liddle, A. L. Opdahl, Ó.P. López, Inferring versioned schemas from NoSQL databases and its applications. Conceptual Modeling, Springer International Publishing, Cham, 2015, pp. 467–480.
[57] D. Martinez-Mosquera, R. Navarrete, S. Lujan-Mora, Modeling and management big data in databases – a systematic literature review, Sustainability 12 (2020) 634, https://doi.org/10.3390/su12020634.

María José Suárez-Cabal is an assistant professor at the University of Oviedo, Spain, and is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). She obtained her PhD in Computing from the University of Oviedo in 2006. Her research has focused on software testing, and more specifically on testing SQL and database applications, being published in high impact international journals and conferences. Her current research interests also include testing NoSQL systems.

Pablo Suárez-Otero received his B.Sc. degree in Computer Engineering in 2015 and in M.Sc. in Computer Engineering in 2017 from the University of Oviedo. He is currently a PhD candidate in Computing at the University of Oviedo as well as an Assistant Professor at the University of Oviedo. He is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). His research interests include software testing, NoSQL databases and data modeling.



Claudio de la Riva received the Ph.D. degree in Computing from the University of Oviedo, Oviedo, Spain, in 2004.He is an Associate Professor with the University of Oviedo and member of the Software Engineering Research Group. His research interests include software verification and validation and software testing, mainly focused on testing database applications and massive data processing.

Javier Tuya received the Ph.D. degree in engineering from the University of Oviedo, Oviedo, Spain, in 1995. He is a Professor with the University of Oviedo, Oviedo, Spain, where he is the Research Leader of the Software Engineering Research Group. He is the Director of the Indra-Uniovi Chair, a member of the ISO/IEC JTC1/SC7/ WG26 Working Group for the recent ISO/IEC/IEEE 29,119 Software Testing Standard, and a Convener of the corresponding UNE National Body Working Group. His research interests in software engineering include verification, and validation and software testing for database applications and service transactions.