Universidad de Oviedo

Maintenance of the logical consistency in Cassandra

Ph.D. Thesis Dissertation in Computer Engineering

Programa de Doctorado en Informática

Author

Pablo Suárez-Otero González

Supervisor

Dr. Pablo Javier Tuya González

December 2022

Universidad de Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español/Otro Idioma: Mantenimiento de la consistencia lógica en Cassandra	Inglés: Maintenance of the logical consistency in Cassandra

2.- Autor	
Nombre: Pablo Suárez-Otero González	DNI:
Programa de Doctorado: Programa de Doctorado en Informática	
Órgano responsable: Centro Internacional de Postgrado	

RESUMEN (en español)

Tradicionalmente, los proyectos de software han utilizado exclusivamente BDs (bases de datos) relacionales. Sin embargo, recientemente la cantidad de datos a gestionar han aumentado exponencialmente, afectando negativamente al rendimiento de las BDs. Debido a esto, han surgido diferentes tecnologías especializadas para gestionar Big Data, siendo una de éstas las BDs NoSQL. Un tipo de estas BDs son las BDs orientadas a columnas, cuyo esquema contiene tablas sin relaciones entre sí y en las que cada tabla se diseña para que una consulta que se requiere ejecutar desde la aplicación pueda ser ejecutado, utilizando un modelo conceptual durante este diseño. Esto implica un esquema desnormalizado que fomenta la duplicación de datos, al poder consultarse el mismo dato en dos consultas diferentes. Esto conlleva problemas en cuanto a la consistencia lógica de los datos, que asegura que los mismos datos que se almacenan en dos tablas diferentes son realmente los mismos.

En esta tesis abordamos dos problemas de las bases de datos orientadas a columnas: 1) el mantenimiento de la consistencia lógica o integridad de los datos ante cambios en los datos y 2) el mantenimiento de la consistencia entre modelo conceptual y esquema cuando hay una evolución de la BD. El primer problema se abordará desde dos enfoques diferentes: uno preventivo denominado MDICA que evita la creación de inconsistencias al modificar los datos en la BD y otro reactivo denominado CONCODA que detecta si se han creado inconsistencias después de modificar los datos en la BD. El segundo problema se abordará a través de un enfoque MDE (Model-driven Engineering) denominado CoDEvo que determinará cómo evolucionar el esquema después de los cambios del modelo conceptual para mantener la consistencia entre éste y el modelo conceptual, lo que denominamos consistencia inter-modelo.

MDICA determina las sentencias de base de datos necesarias para ejecutar contra la base de datos una determinada modificación de datos, que consiste en una tupla para insertar, borrar o actualizar a nivel conceptual, manteniendo a su vez la consistencia lógica. MDICA ayuda a los desarrolladores de dos maneras: generando las sentencias de BD necesarias para mantener la consistencia lógica y generando mensajes para evitar problemas como la pérdida de información, los datos repetidos redundantes o tablas vacías. Hemos realizado una

experimentación para validar MDICA, comprobando exhaustivamente las operaciones de BD que MDICA determinó para un conjunto de cambios en el nivel conceptual para su evaluación.

El objetivo de CONCODA es determinar si una BD orientada a columnas mantiene la consistencia lógica después de que se hayan realizado una o varias modificaciones de datos contra la BD. Hemos ideado un oráculo utilizando una BD relacional que implementa el modelo conceptual debido a que las BDs relacionales conservan la consistencia lógica. Una vez realizadas las modificaciones de datos, tanto en la BD relacional como en la base de datos orientada a columnas, se ejecutarán consultas equivalentes en ambas BD. Si estas consultas devuelven los mismos datos, se garantiza la coherencia lógica. CONCODA también se ha usado en la experimentación para validar las sentencias de BDs determinadas por MDICA.

El objetivo de CoDEvo es el de obtener los cambios específicos a aplicar sobre el esquema ante cambios en el modelo conceptual para mantener la consistencia entre ellos. Hemos analizado varios proyectos de código abierto para obtener patrones de cómo evoluciona el esquema para automatizar dicho proceso. Esta parte de la tesis se integrará en un framework que abordará otros problemas relacionados con la evolución de la base de datos, incluyendo el mantenimiento de la consistencia lógica.

RESUMEN (en Inglés)

Traditionally, software projects have exclusively used relational DBs (databases). However, in recent times the amount of data that these projects manage have exponentially increased, negatively affecting the DB performance. Due to this, different technologies specialized to manage big data have emerged, being one of these technologies the NoSQL DBs. One type of these DBs are the column-oriented DBs, whose schema contains tables without relationships between each other. Each one of these tables is designed specifically for satisfying the requirements of a query that will be executed by the client application, using a conceptual model during this design. This design strategy implies a denormalized schema that encourages data duplication, as the same data can be queried several times, thus storing the data in each of the tables that satisfy the queries. This brings problems regarding the logical consistency of the data, which ensures that the same data that are stored in two different tables are actually the same data.

In this thesis we address two problems from column-oriented databases: 1) the maintenance of the logical consistency or data integrity when there are changes in the data and 2) the maintenance of the consistency between conceptual model and schema when there is an evolution of the DB. The first problem will be addressed from two different approaches: a preventive one named MDICA that prevents the creation of inconsistencies when modifying data in the DB and a reactive one named CONCODA that detects if inconsistencies have been created after modifying data in the DB. The second problem will be addressed through a MDE (model-driven engineering) approach named CoDEvo that will determine how to evolve the schema after the conceptual model changes in order to maintain the consistency between it and the conceptual model, which we name as inter-model consistency.

Universidad de Oviedo

MDICA determines the database statements required to perform against the database a given modification of data, which consists on a tuple to insert, delete, or update at the conceptual level, while maintaining the logical consistency. MDICA 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. We have performed experimentation to validate MDICA, exhaustively checking the database operations that MDICA determined for a set of changes at the conceptual level for evaluation.

CONCODA determines if a column-oriented DB maintains the logical consistency after one or more modifications of data have been performed against the DB. We have devised an oracle using a relational DB that implements the conceptual model, due to the fact that relational databases ensure logical consistency through the normalization of the schema and the implementation of integrity constraints. After the modifications of data are performed against both the relational DB and the column-oriented database, equivalent queries will be executed against both DBs. If these queries return the same data, then the logical consistency is ensured. CONCODA also serves as validation for MDICA to ensure that the database statements determined by MDICA actually maintain the logical consistency.

The objective of CoDEvo is to obtain the changes that need to be applied to the schema when the conceptual model evolves in order to maintain consistency between the conceptual model and the schema. We have analysed several open-source projects to obtain patterns of how the schema evolves to automate this process. This part of the thesis will be integrated into a framework that will address other problems related to database evolution, including the maintenance of logical consistency.

SR. PRESIDENTE DE LA COMISIÓN ACADÉMICA DEL PROGRAMA DE DOCTORADO EN INFORMÁTICA

CONTENTS

In memoriam of my father

ACKNOWLEDGEMENTS

This work has been performed under the research projects PID2019-105455GB-C32 funded by MCIN/ AEI/10.13039/501100011033, TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Economy and Competitiveness; GRUPIN14-007, funded by the Principality of Asturias (Spain); TIN2016-76956-C3-1-R, funded by the Spanish Ministry of Science and Technology; the Severo Ochoa pre-doctoral grant PA-21-PF-BP20-184; and ERDF Funds.

AGRADECIMIENTOS

Quiero agradecer en primer lugar el apoyo que he tenido durante estos años de tesis por parte de mi director Javier Tuya, que siempre ha estado disponible para ayudarme en todo momento y corregir mis fallos en este periodo. También quiero agradecer el apoyo recibido por parte del resto del Grupo de Investigación en Ingeniería del Software, especialmente a María José Suárez-Cabal que también me ha apoyado de forma constante durante estos años en la investigación cuyo resultado se muestra en esta tesis.

También quiero acordarme del profesor Michael J. Mior de la Rochester Institute of Technology, cuya colaboración ha sido un gran aporte a esta investigación. Quiero agradecer a su vez la gran experiencia personal y académica que fue la estancia realizada en Rochester bajo su supervisión.

No puedo olvidarme de mi familia, la cual ha sido un gran apoyo en momentos muy difíciles. Especial agradecimiento a mi madre, que durante la mayor parte de la realización de esta tesis ha tenido que estar soportando momentos muy difíciles y duros por la situación causada por la enfermedad de mi padre. Sé que no ha sido fácil ejercer de cuidadora y a su vez seguir mostrándome el apoyo incondicional que he recibido.

ABSTRACT

Traditionally, software projects have exclusively used relational DBs (databases). However, in recent times the amount of data that these projects manage have exponentially increased, negatively affecting the DB performance. Due to this, different technologies specialized to manage big data have emerged, being one of these technologies the NoSQL DBs. One type of these DBs are the column-oriented DBs, whose schema contains tables without relationships between each other. Each one of these tables is designed specifically for satisfying the requirements of a query that will be executed by the client application, using a conceptual model during this design. This design strategy implies a denormalized schema that encourages data duplication, as the same data can be queried several times, thus storing the data in each of the tables that satisfy the queries. This brings problems regarding the logical consistency of the data, which ensures that the same data that are stored in two different tables are actually the same data.

In this thesis we address two problems from column-oriented databases: 1) the maintenance of the logical consistency or data integrity when there are changes in the data and 2) the maintenance of the consistency between conceptual model and schema when there is an evolution of the DB. The first problem will be addressed from two different approaches: a preventive one named MDICA that prevents the creation of inconsistencies when modifying data in the DB and a reactive one named CONCODA that detects if inconsistencies have been created after modifying data in the DB. The second problem will be addressed through a MDE (model-driven engineering) approach named CoDEvo that will determine how to evolve the schema after the conceptual model changes in order to maintain the consistency between it and the conceptual model, which we name as inter-model consistency.

MDICA determines the database statements required to perform against the database a given modification of data, which consists on a tuple to insert, delete, or update at the conceptual level, while maintaining the logical consistency. MDICA 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. We have performed experimentation to validate MDICA, exhaustively checking the database operations that MDICA determined for a set of changes at the conceptual level for evaluation.

CONCODA determines if a column-oriented DB maintains the logical consistency after one or more modifications of data have been performed against the DB. We have devised an oracle using a relational DB that implements the conceptual model, due to the fact that relational databases ensure logical consistency through the normalization of the schema and the implementation of integrity constraints. After the modifications of data are performed against both the relational DB and the column-oriented database, equivalent queries will be executed against both DBs. If these queries return the same data, then the logical consistency is ensured. CONCODA also serves as validation for MDICA to ensure that the database statements determined by MDICA actually maintain the logical consistency.

The objective of CoDEvo is to obtain the changes that need to be applied to the schema when the conceptual model evolves in order to maintain consistency between the conceptual model and the schema. We have analysed several open-source projects to obtain patterns of how the schema evolves to automate this process. This part of the thesis will be integrated into a

framework that will address other problems related to database evolution, including the maintenance of logical consistency.

RESUMEN

Tradicionalmente, los proyectos de software han utilizado exclusivamente BDs (bases de datos) relacionales. Sin embargo, recientemente la cantidad de datos a gestionar han aumentado exponencialmente, afectando negativamente al rendimiento de las BDs. Debido a esto, han surgido diferentes tecnologías especializadas para gestionar Big Data, siendo una de éstas las BDs NoSQL. Un tipo de estas BDs son las BDs orientadas a columnas, cuyo esquema contiene tablas sin relaciones entre sí y en las que cada tabla se diseña para que una consulta que se requiere ejecutar desde la aplicación pueda ser ejecutado, utilizando un modelo conceptual durante este diseño. Esto implica un esquema desnormalizado que fomenta la duplicación de datos, al poder consultarse el mismo dato en dos consultas diferentes. Esto conlleva problemas en cuanto a la consistencia lógica de los datos, que asegura que los mismos datos que se almacenan en dos tablas diferentes son realmente los mismos.

En esta tesis abordamos dos problemas de las bases de datos orientadas a columnas: 1) el mantenimiento de la consistencia lógica o integridad de los datos ante cambios en los datos y 2) el mantenimiento de la consistencia entre modelo conceptual y esquema cuando hay una evolución de la BD. El primer problema se abordará desde dos enfoques diferentes: uno preventivo denominado MDICA que evita la creación de inconsistencias al modificar los datos en la BD y otro reactivo denominado CONCODA que detecta si se han creado inconsistencias después de modificar los datos en la BD. El segundo problema se abordará a través de un enfoque MDE (Model-driven Engineering) denominado CoDEvo que determinará cómo evolucionar el esquema después de los cambios del modelo conceptual para mantener la consistencia entre éste y el modelo conceptual, lo que denominamos consistencia inter-modelo.

MDICA determina las sentencias de base de datos necesarias para ejecutar contra la base de datos una determinada modificación de datos, que consiste en una tupla para insertar, borrar o actualizar a nivel conceptual, manteniendo a su vez la consistencia lógica. MDICA ayuda a los desarrolladores de dos maneras: generando las sentencias de BD necesarias para mantener la consistencia lógica y generando mensajes para evitar problemas como la pérdida de información, los datos repetidos redundantes o tablas vacías. Hemos realizado una experimentación para validar MDICA, comprobando exhaustivamente las operaciones de BD que MDICA determinó para un conjunto de cambios en el nivel conceptual para su evaluación.

El objetivo de CONCODA es determinar si una BD orientada a columnas mantiene la consistencia lógica después de que se hayan realizado una o varias modificaciones de datos contra la BD. Hemos ideado un oráculo utilizando una BD relacional que implementa el modelo conceptual debido a que las BDs relacionales conservan la consistencia lógica. Una vez realizadas las modificaciones de datos, tanto en la BD relacional como en la base de datos orientada a columnas, se ejecutarán consultas equivalentes en ambas BD. Si estas consultas devuelven los mismos datos, se garantiza la coherencia lógica. CONCODA también se ha usado en la experimentación para validar las sentencias de BDs determinadas por MDICA.

El objetivo de CoDEvo es el de obtener los cambios específicos a aplicar sobre el esquema ante cambios en el modelo conceptual para mantener la consistencia entre ellos. Hemos analizado varios proyectos de código abierto para obtener patrones de cómo evoluciona el esquema para automatizar dicho proceso. Esta parte de la tesis se integrará en un framework que abordará otros problemas relacionados con la evolución de la base de datos, incluyendo el mantenimiento de la consistencia lógica.

I INTRODUCTION

In this chapter we introduce the main information of the thesis, focusing on the main goals of thesis, how we achieved them, the contributions obtained and the publications that the doctoral candidate authored related to the thesis.

In section I.1 we detail the research context of the thesis. In section I.2 we describe the research hypothesis. In section I.3 we describe the research goals of the thesis, whose contribution are detail in section I.4 and the methodologies that we used during the thesis are described in section I.5. The publications related to the thesis research are detailed in section I.6 alongside the research management. The chapter finishes by detailing the thesis manuscript organization in section I.7.

I.1 RESEARCH CONTEXT

Since the beginnings of software development, databases have been used to store and manage the data of software applications [1]. Since the 1970s and until now the databases that have been used the most in software projects are the relational databases [2]. In these databases, data are stored in tables with relationships between these tables, which allows the data of different tables to be joined between each other, allowing a normalized model where data is not repeated among the tables. However, as more data is stored in the database, the database operations performed against it, such as queries and data manipulation operations (insertions, deletions, and updates), become more costly [3]. This was not a major problem until the beginning of the current century, when the amount of data to manage started to increase exponentially, slowing the execution of the database operations [4].

In order to improve the performance of the database, the database can be distributed in several physical nodes that together compose a cluster, each one storing a portion of the data, as well as replicating these data in more than one node. There have been new proposals regarding relational databases to distribute the data, although the scalability that they provide is still limited [5] due to the difficulties in distributing the data while assuring the ACID properties (atomicity, consistency, isolation, and durability).

Due to the performance limitations of relational databases when managing big data, corporations such as Google and Facebook started in the 2000s their research on alternatives to relational databases when working with big data. One of these alternatives were a wide set of new databases, each one with a different approach, which were grouped together with the denomination NoSQL (“Not only SQL")[6]–[10].

NoSQL databases are specifically used when high performance to both insert and query large amounts of data is needed, and relational databases cannot achieve this performance [11]. This performance is achieved in part by allowing a distributed system as well as by replicating data, which is difficult to achieve in traditional relational databases. Note that distributing the data across physical nodes of a cluster makes more difficult to guarantee the ACID properties, which implies several issues such as:

- The data integrity (also referred as **logical consistency** in the rest of this thesis [12]) is not guaranteed in the database, requiring it to be maintained in the application side [13].

- It is more difficult to validate the transactions performed against the database [14].

Despite not guaranteeing the ACID properties, NoSQL databases guarantee the BASE properties [15]:

- Basic Availability: the database will at least answer even where there is a failure in part of the distributed system.
- Soft state: the data stored can change without external intervention.
- Eventual consistency: the nodes that store the same replicated data will eventually store the same values.

Depending on their data model [6], NoSQL databases are classified in:

- Key-value-oriented such as Redis, Riak or Dynamo.
- Column-oriented such as Cassandra, HBase or BigTable.
- Document-oriented such as MongoDB, Couchbase or CouchDB.
- Graph-oriented such as Neo4J, JanusGraph or TigerGraph.

In this thesis we focus on column-oriented databases, which are composed of tables with no relationships between each other, in contrast to relational databases. The contributions of this thesis aim to be useful for column-oriented databases in general, but we have particularized during the research in the most used NoSQL column-oriented databases Apache Cassandra, developed by the Apache Software Foundation [16]. There are significant differences between the data model of a Cassandra database and a relational database such as the aforementioned lack of relationships in Cassandra. Another difference is regarding the primary key, as the primary key from a Cassandra table is composed of two types:

- **Partition key:** determines the physical node where the data is going to be stored.
- **Clustering key**: determines how the data is ordered inside a physical node.

By default, the results of a Cassandra query can only be filtered through a WHERE clause using only values from the primary key, being the partition key mandatory and the clustering key optional values (see section II.1.7). Secondary indexes, although available, are not recommended in Cassandra due to the deficient performance that they generate when they are used in a query [13]. Cassandra also allows a wider range of data to be stored in the database than relational databases, such as collections which allows to store more than one value in a table cell.

Another significant difference is the creation of the database schema. In both relational databases and column-oriented databases it is recommended to design a *conceptual model* that represents a normalized version of the components of the system. In the case of a relational database, the *logical model* or *schema* is created based on this conceptual model, maintaining the normalization of the model. However, in column-oriented databases such as Cassandra the creation of the schema or logical model is different, as the tables are created following a query-driven approach to retrieve data faster, in which each table satisfies a query to be executed by the client application [17]–[19]. This approach implies data duplication, as the same data can be queried in more than one way, thus creating more than one table to satisfy each query. This duplication of data and the absence of internal mechanisms such as integrity constraints makes more difficult to maintain of the logical consistency.

This thesis addresses two research lines focused on problems related to column-oriented databases:

- The **logical consistency** maintenance for changes in the data by client applications (chapters III and IV of this thesis).
- **Database evolution,** focusing on maintaining the consistency between conceptual model and the schema (chapter V of this thesis).

Usually, the data of a database is modified through the execution of database statements embedded in a client application to insert, delete, and update data. The implementation of these modifications is usually organized following a normalized model such as the conceptual model by implementing a function for each modification. For instance, in a relational database with a normalized schema each modification of data involving a conceptual model entity is usually implemented in a function that performs the modification in only one table. Furthermore, when some data is repeated such as in foreign keys, integrity constraints are defined internally in the database to ensure the logical consistency. However, in column-oriented databases the denormalization and lack of integrity constrains makes difficult to ensure this consistency. When a modification of data is performed against the database, several tables are affected by the modification, creating inconsistencies if one of them is forgotten during the implementation of the database statements. For this purpose, we propose **two approaches** for the **maintenance of the logical consistency**, one preventive for new applications named MDICA and a reactive one for existing application named CONCODA**,** that will determine the following:

- **MDICA:** the database statements required to perform a **modification of data** (insertion, deletion or update of a tuple at the conceptual level) in the column-oriented database while maintaining the logical consistency.
- **CONCODA**: if the database maintains the logical consistency after one or more modifications of data have been performed against the column-oriented database.

The other research line is related to changes in the structure of the schema of a column-oriented database. Both the conceptual model and the schema can change during the lifetime of a project, as new requirements can trigger their evolution. As data model design methodologies use an explicit conceptual model to design the schema, an **inter-model consistency** exists that ensures that the schema fulfils the conceptual restrictions specified in the conceptual model (relationships, primary keys…). A new requirement that changes either the conceptual model or the schema may jeopardize this inter-model consistency. For instance, if a conceptual model evolves by adding a new conceptual restriction and the schema is not properly updated, the schema might allow to store new data that contradicts this new restriction. This problem is even more critical in NoSQL column-oriented databases where repetition of data is common. We approach this problem in this thesis by proposing a **MDE (model-driven engineering) approach** named **CoDEvo** that determines how the schema must evolve when there is an evolution of the database that changes the conceptual model to maintain the inter-model consistency.

I.2 Research Hypothesis

This thesis addresses problems related to column-oriented databases regarding the logical consistency and database evolution. The techniques developed for problems related to these topics in databases have been mostly focused on relational databases. However, for NoSQL column-oriented databases, these techniques cannot be applied due to the differences that exists with relational databases such as in the data model, where repetition of data is common. Therefore, innovative approaches focused on column-oriented databases are required to solve these specific problems. This general hypothesis is divided in the following three:

H1. When modifying data in a column-oriented database, the logical consistency is usually maintained in the client applications by implementing the appropriate database statements and related code to insert, update or delete data in the required tables. This might lead to mistakes during this implementation produced by developers that incur in the loss of this consistency. An approach that automatically provides these database statements or another one that determines if the database statements executed maintain the logical consistency, will help developers to avoid the production of inconsistencies.

H2. The column-oriented database Apache Cassandra allows the use of the "materialized views" feature, which automatically maintains the logical consistency for modifications of data by creating a table-like structure based on a primary table. However, their use is limited to a set of possible table designs, as it is detailed in section II.2.1. An approach to maintain the logical consistency that is not limited will be a general solution that can be applied in all possible scenarios.

H3. The data models of a database, such as the conceptual model and the schema, are designed after an initial set of project requirements. When these requirements change, the data models may be updated to keep fulfilling the project requirements. In column-oriented databases where the schema is usually denormalized and techniques focused on relational databases cannot be applied to it, it is more difficult to maintain the consistency between the conceptual model and the schema. An approach that guides how to evolve the database when there are requirement changes will help developers to maintain this consistency between the conceptual model and the schema, avoiding mistakes in the definition of the schema.

I.3 Research Goals

The main goal of this thesis is to maintain both the logical consistency and the inter-model consistency when there is an evolution of the database, which is divided in two research goals:

1. Preserve the logical consistency of the data that is stored in a column-oriented database. We have approached this research goal in two ways: a preventive one to avoid the creation of inconsistencies and a reactive one that detects if inconsistencies were created after the data is modified by a client application. We defined the following subgoals for each of these approaches:
   - 1.1. Prevent the creation of inconsistencies when insertions, deletions, and updates at the conceptual level (tuples of data), that we name **modifications of data**, are performed against the column-oriented database through the execution of database statements. We divide this subgoal in the following ones:
     - 1.1.1. Determine the actions required to maintain logical consistency when a modification of data is performed against a column-oriented database.
     - 1.1.2. Automate the maintenance of the logical consistency to prevent the creation of inconsistencies by determining the database statements required to perform a modification of data against the database.
     - 1.1.3. Evaluate and validate the research goal 1.1.2 through case studies.
   - 1.2. Detect the inconsistencies that exist in a column-oriented database after performing one or more modifications of data. When inconsistencies are detected, identify the tables and the data stored in them that are inconsistent.

2. Maintain the inter-model consistency that exists between a conceptual model and a column-oriented database schema when there is an evolution of the database due to new requirements. This sub-goal is further divided in the following specific sub-goals:
   2.1. Determine the type of changes in the data models that occur in real open-source projects by analyzing them, focusing on how they affect the database schema regarding the consistency between the conceptual model and the schema.
   2.2. Determine the actions to perform in the column-oriented database schema to maintain the inter-model consistency for specific changes in the conceptual model in order to maintain the consistency between conceptual model and the schema.
   2.3. Evaluate the efficacy and efficiency of the approach established from sub-goal 2.2 using real projects that use column-oriented databases.

Figure 1 displays the research goals hierarchically.

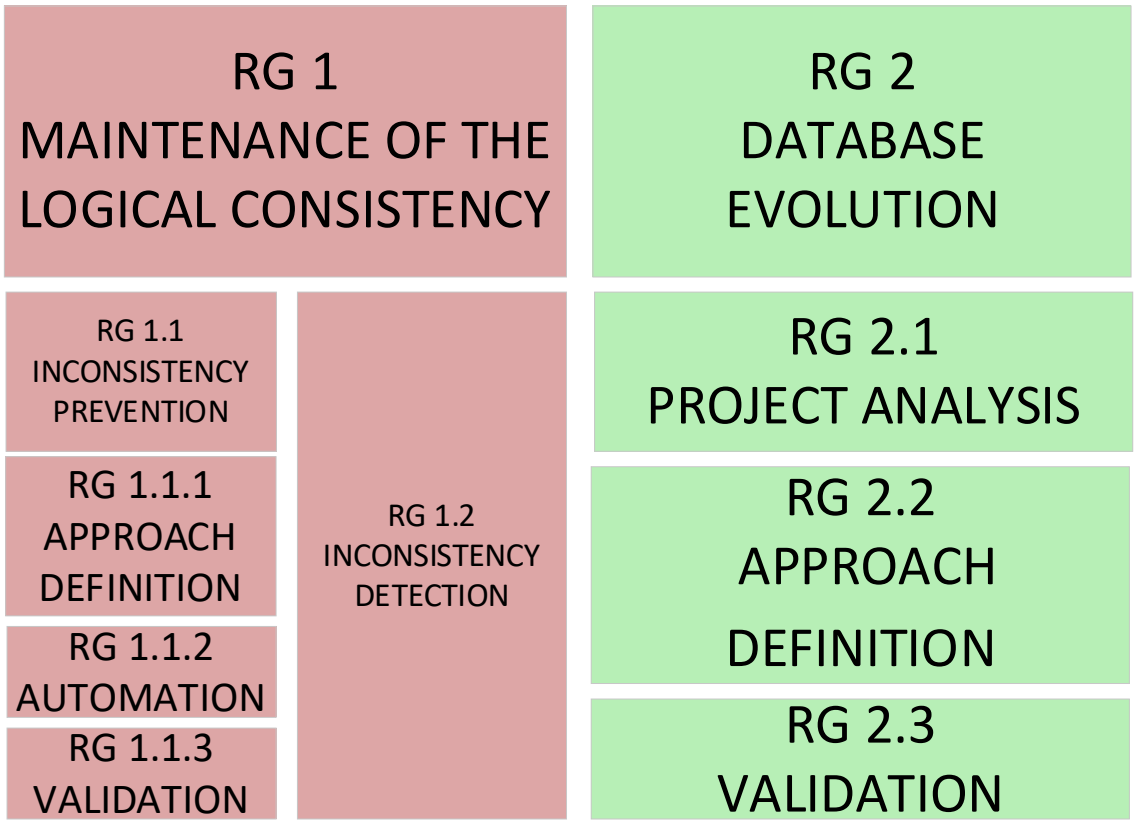


Figure 1 Thesis Research Goals

I.4 CONTRIBUTIONS

The main contribution of the thesis is to maintain both the logical consistency and the inter-model consistency of a column-oriented database, particularizing on Cassandra databases. Considering the thesis research goals defined in section I.3, we have made the following contributions for each research goal:

I.4.1 Research goal 1

C1. The definition of the conditions that tables must satisfy to guarantee a good design in a column-oriented databases according to a given conceptual data model.

C2. The determination of the database statements that are required to perform against the databases in order to perform a modification of data while maintaining the logical consistency through a novel approach named **MDICA**.

C3. The generation of advises when there are issues during the modification of data that cannot be automatically solved.

C4. Automation of MDICA in a tool named **CONSISTE**.

C5. The evaluation of MDICA through its application to a set of case of studies obtained from the literature where each one contains a conceptual model and a schema.

C6. The determination of whether a column-oriented database maintains the logical consistency after modifications of data have been performed against the database through a novel approach named **CONCODA**.

C7. The determination of the tables where data inconsistencies have been detected regarding the logical consistency in a column-oriented database.

C8. Am empirical validation of MDICA using CONCODA.

I.4.2 Research goal 2

C9. Establishment of a framework for the evolution of a column-oriented database when there are changes in the requirements.

C10. An analysis of the conceptual model and schema changes that occur in open-source projects due to new requirements.

C11. Definition of an MDE approach named **CoDEvo** for schema evolution to maintain the inter-model consistency when there is a change in the requirements that directly change the conceptual model.

C12. Determination of the specific actions required to evolve a schema when there is a change that modified the conceptual model that was used to design the schema.

C13. Automation of CoDEvo in a tool.

C14. Validation of CoDEvo by comparing the schemas determined by it against the schemas determined by the developers of a set of open-source projects where there were requirement changes.

I.5 Research Methodology

In this section we first describe the methodologies that have been used in this thesis, and then map for each research goal the methodologies used to achieve them:

- Literature survey: we analysed research work that address schema design, schema evolution and logical consistency maintenance. We also considered how these issues were solved in open-source projects.
- Action research [20]: Class of methods that combine the theory and practice by considering the needs of real projects.
- Empirical validation: we validated the results of the research using real case studies from open-source projects. This allowed us to check the validity of the approaches by comparing them against a real system.

- Incremental development: we used incremental processes based on agile development methodologies such as Scrum. The creation of tools using these processes allow to incrementally add the results of the research.

These methodologies were applied in the corresponding research goals (RGs):

- RG 1.1.1: We performed a *literature survey* to obtain information about the diverse types of modifications of data from both the literature and from open-source projects that use column-oriented databases.
- RG 1.1.2: We applied the *action research* methodology using the knowledge obtained in RG 1.1.2 to develop MDICA to maintain the logical consistency. Then, using the *incremental development*, we implemented this approach in a tool.
- RG 1.1.3: We validated MDICA using the *empirical validation* methodology through its application in several case studies composed of a conceptual model and a database schema obtained from different projects from the literature.
- RG 1.2: We applied the *action research* methodology to develop CONCODA, which checks if a set of database statements against a column-oriented database maintains the logical consistency. We validated CONCODA through the *empirical validation* methodology by applying it to the case studies used in RG 1.1.3.
- RG 2.1: We performed a *literature survey* from research works and open-source projects, obtaining information about the different changes that could modify the conceptual model and would require further actions in the schema to maintain the inter-model consistency.
- RG 2.2: We applied the *action research* methodology to develop CoDEvo to maintain the inter-model consistency after a change in the conceptual model. This approach was implemented in a tool using the *incremental development* methodology.
- RG 2.3: We validated CoDEvo using the *empirical validation* methodology through the application of the developed approach in several open-source projects with a schema evolution history.

I.6 PUBLICATIONS AND RESEARCH MANAGEMENT

In this section we enumerate and describe the different publications that we have authored during the thesis, mapping each publication with one or more of the contributions of this thesis that were enumerated in the previous section. Figure 2 displays a summary of these publications organized by research goals (RG 1.1, RG 1.2, and RG 2) and an additional line for publications that were a general dissemination of the research. Regarding the RG 1.1 and RG 1.2 we have published several works in national and international conferences as well as two JCR journals. Regarding the RG 2, we have also published several works in national and international conferences, and we have also submitted a work to the JCR journal “Journal of Systems and Software” whose detail is presented in the chapter V of this thesis. The Research Goal 2 was done in collaboration with the Rochester Institute of Technology (RIT) from New York, USA. This collaboration started in 2020, conducting a research visit in RIT of 3 months from January 2022 to April 2022.

In addition to the works published in journals and conferences, the doctoral candidate has also attended to several summer schools related to the topics of the thesis, as well as presented the advances of the thesis in several locally and internationally events.

In the next subsections we detail the publications for each research line as well as general divulgation publications, specifying which contribution is addressed in each work.

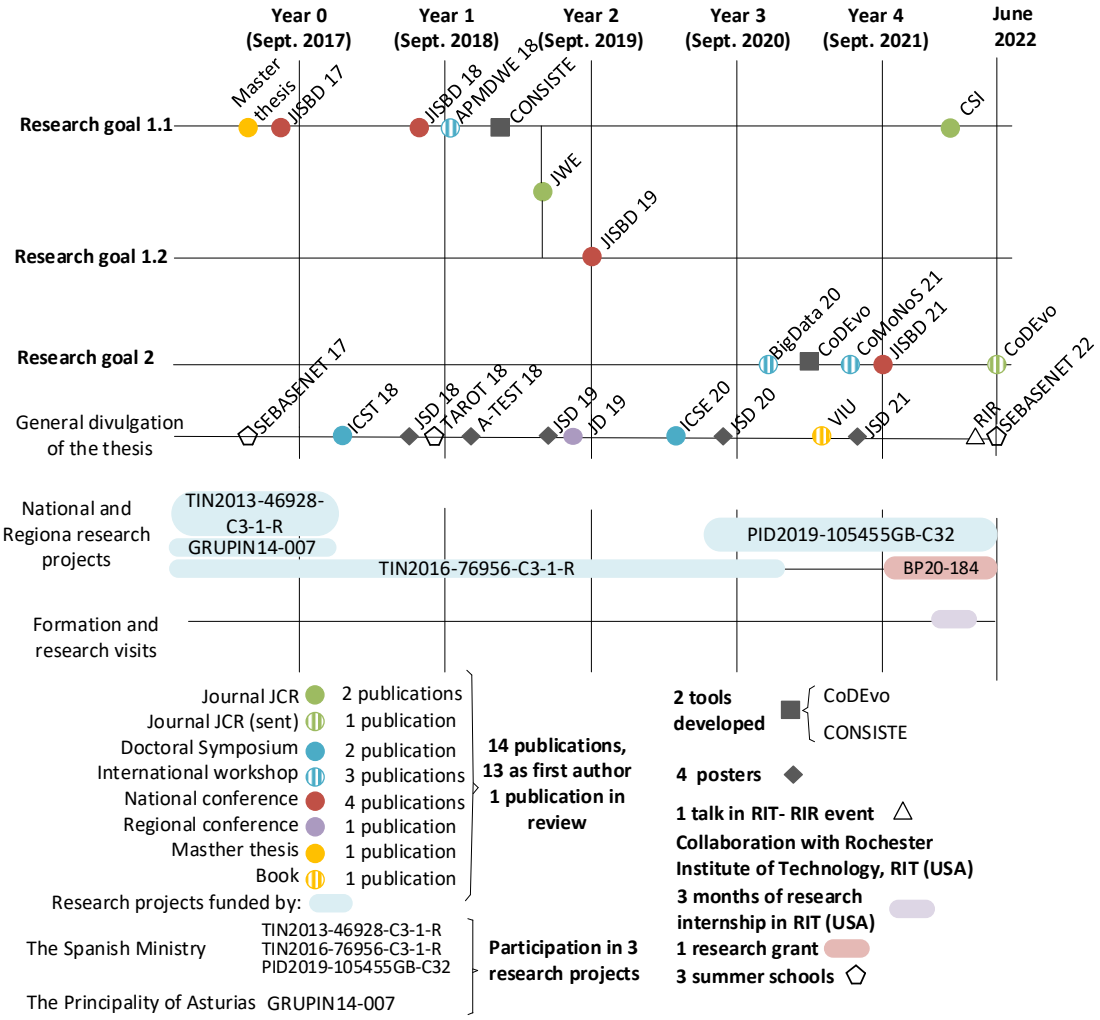


Figure 2 Summary of the thesis research and divulgations

I.6.1 Research goal 1.1: Maintenance of the logical consistency-Preventive approach

The RG 1.1 addresses the maintenance of the logical consistency from a preventive point of view by providing the database statements required to perform a desired modification of data. For this research goal we published the following 6 publications, indicating at the end of each reference the contributions provided in it:

[21] Pablo Suárez-Otero. Advisor: Javier Tuya. Mantenimiento de la consistencia lógica en bases de datos NoSQL Cassandra ante modificaciones de datos a nivel de modelo conceptual de datos. 2017. (**Master thesis**)-**C2, C3**

[22] Pablo Suárez-Otero, Javier Gutiérrez, Claudio de la Riva and Javier Tuya. Mantenimiento de la Consistencia Lógica en Cassandra. Jornadas XXII Jornadas de Ingeniería del Software y Bases de Datos. 2017 (**JISBD 2017**)-**C2**

[23] Pablo Suárez-Otero, María José Suárez-Cabal and Javier Tuya. Evaluación del mantenimiento de la consistencia lógica en Cassandra. Jornadas XXIII Jornadas de Ingeniería del Software y Bases de Datos. 2018 (**JISBD 2018**)-**C2**

[24] Pablo Suárez-Otero, María José Suárez-Cabal and Javier Tuya. Leveraging conceptual data models for keeping Cassandra database integrity. WEBIST 2018-Proceedings of the 14th International Conference on Web Information Systems and Technology. V1. 2018 (**APMDWE 18**)-**C2**

[25] Pablo Suárez-Otero, María José Suárez-Cabal and Javier Tuya. Journal of Web Engineering. 2019 (**JWE**)-**C2, C3**

[26] María José Suárez-Cabal, Pablo Suárez-Otero, Claudio de la Riva and Javier Tuya. MDICA: Maintenance of data integrity in column-oriented database applications. Computer Standards & Interfaces. 2023 (**CSI**)-**C1, C2, C3, C5**

We started addressing this research goal before the thesis began in a Master thesis by proposing an approach to maintain the logical consistency for insertions and deletions of data, although with room for improvement. This master thesis received a “Alan Turing” award as second-best master thesis focused on software engineering by the “Colegio Oficial de Ingenieros Informáticos del Principado de Asturias” in 2019.

These first advances were also published among three conference articles [22]–[24]. First, we proposed an initial approach for maintaining the logical consistency [22], publishing its evaluation in a later work [23].A more refined approach, along with its evaluation, was published later in an international workshop [24].

In [25] we published our first work in a JCR journal, where we defined a preliminary approach to address the maintenance of the logical consistency, as well as our first empirical validation. This was completed in another JCR journal, formalizing it, and naming it as MDICA [26]. The detail of these works is presented in this thesis in Chapter III.

As a result of this research, a tool named CONSISTE was developed:

CONSISTE: A tool that automatically provides the database statements to be executed given a tuple to insert, delete or update. It additionally allows the automation of these database statements by executing them against the database. **C4**

I.6.2 Research Goal 1.2: Maintenance of the logical consistency-Reactive approach

The research goal 1.2 focuses on maintaining the logical consistency from a reactive point of view, by checking if the database currently has logical consistency. We have published 2 works regarding this research goal:

[27] Pablo Suárez-Otero, María José Suárez-Cabal and Javier Tuya. Verificación del mantenimiento de la consistencia lógica en bases de datos Cassandra. XXIV Jornadas de Ingeniería del Software y Bases de Datos. 2019 (**JISBD 2019**)-C6, C7, C8

[25] Pablo Suárez-Otero, María José Suárez-Cabal y Javier Tuya. Journal of Web Engineering. 2019 (**JWE**)-C8

We first published the definition of the approach named "CONCODA" that addresses this research goal in [27]. In this work we also presented its application to validate MDICA in a preliminary evaluation. The complete evaluation of this approach was published in [25]. In Chapter IV we include part of these works, extending them in order to provide more detail to the process of CONCODA.

I.6.3 Research Goal 2: Database evolution: Schema evolution

The research goal 2 focuses on maintaining the consistency between the conceptual model and the schema of a column-oriented database such as Cassandra. This research goal was addressed in a collaboration with the assistant professor Michael J. Mior from the Rochester Institute of Technology. This research line has 3 publications and an article submitted to a JCR journal:

[28] Pablo Suárez-Otero, Michael J. Mior, María José Suárez-Cabal and Javier Tuya. Maintaining NoSQL database quality during conceptual model evolution. 2020 IEEE International Conference on Big Data. 2020 (**BigData 20**)-C9, C10

[29] Pablo Suárez-Otero, Michael J. Mior, María José Suárez-Cabal and Javier Tuya. An Integrated Approach for Column-Oriented Database Application Evolution Using Conceptual Models. International Conference on Conceptual Modeling. 2021. (**CoMoNoS 21**)-C9

[30] Pablo Suárez-Otero, Michael J. Mior, María José Suárez-Cabal and Javier Tuya. Evolución en sistemas de bases de datos orientadas a columnas ante cambios conceptuales. XXV Jornadas de Ingeniería del Software y Bases de Datos. 2021 (**JISBD 21**)-C8

[31] Pablo Suárez-Otero, Michael J. Mior, María José Suárez-Cabal and Javier Tuya. CoDEvo: Column family database evolution using model transformations. Sent to JCR Journal "Journal of Systems and Software", under review. 2022. (**CoDEvo**)-C10, C11, C12, C14

We first published a proposal of a framework for the evolution of the database when there is a change in the requirements that evolve the conceptual model and the schema in [28] and in [30]. In these works, we also analysed several open-source projects to obtain knowledge about the changes in the conceptual model that evolve the schema due to changes in the requirements that initially changed the conceptual model.

The previous framework was extended in [29] by also considering direct changes to the schema as well as specifying how each part of the framework (inter-model consistency, logical consistency, and client application update) is going to be addressed. This framework description is included in Chapter V of this thesis.

In [31] we approached in detail the evolution of the schema to maintain the consistency between it and the conceptual model. We created a MDE approach where, given a change in the conceptual model, it provides the changes to perform in the schema to maintain this consistency. Most of this work is included in Chapter V of the thesis. This approach was implemented in a tool also named CoDEvo that automates the process:

CoDEvo: A tool based on MDE that automatically provides the changes that are required to evolve the schema after a change in the conceptual model. This information is provided using model. C13

I.6.4 General dissemination related to the Thesis

We have published 3 articles in regional and international conferences where we presented the main objectives of this thesis.

[32] Pablo Suárez-Otero. Analysis of the Logical Consistency in Cassandra. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. 2018. (**ICST 18**).

[33] Pablo Suárez-Otero. Mantenimiento de la consistencia lógica en Cassandra. VIII Jornadas doctorales de la Universidad de Oviedo. 2019. (**JD 19**)

[34] Pablo Suárez-Otero. Towards data integrity in Cassandra database applications using conceptual models. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings. 2020. (**ICSE 20**)

First, we presented a doctoral symposium at the top conference on software testing, ICST [32]. In this work we briefly described our objectives and research timeline. In [33] we did a similar work, presenting it in a regional conference where doctoral candidates presented their advances in their PhD. Lastly, we published a thesis divulgation in the doctoral symposium track of the most important conference on software engineering, ICSE [34], where we obtained useful feedback to our work from a panel of experts in software engineering.

We also published several posters where we explained the advances of the thesis. Most of these posters were published during the doctoral symposiums organized by the doctoral committee of the University of Oviedo (**JSD 18, JSD 19, JSD 20, JSD 21**) and one was presented during the workshop A-TEST in 2018 (**A-TEST 18**).

During this thesis, a book intended to be used by students of a Big Data Master in the Valencian International University was published in 2021 [35]. In this book, parts of the advances in maintenance of the logical consistency were included.

[35] Pablo Suárez-Otero. “Sistemas de almacenamiento y gestión Big Data”. Valencia International University, 2021 (**VIU**)

I.6.5 Projects and Research Management

This thesis has been done under the supervision of the Software Engineering Research Group (GIIS) of the University of Oviedo, specially by Javier Tuya and María José Suárez-Cabal. The first two research lines were supervised entirely by them. As for the third research line, we started a collaboration with assistant professor Michael J. Mior from the Rochester Institute of Technology (RIT).

During the collaboration with the assistant professor Mior, the doctoral candidate visited the Rochester Institute of Technology once from the 24th of January of 2022 to the 23rd of April of 2022. During this stay, the work related to the third research line was finished.

This thesis was funded by several national and regional projects, which were leaded by Javier Tuya and Claudio de la Riva:

PERTEST – Pruebas de la persistencia de datos y perspectiva de usuario bajo nuevos paradigmas - TIN2013-46928-C3-1-R: Research project funded by the Spanish ministry about software testing in Big Data. Main researcher: Javier Tuya

TestEAMoS - Testing emergent technology applications: massive data processing and NoSQL databases - TIN2016-76956-C3-1-R: Research project funded by the Spanish ministry about software testing in both Big Data and mobile. Main researcher Javier Tuya

GRUPIN - GRUPIN14-007: Research project funded by the Principality of Asturias about software testing. Main researcher Claudio de la Riva.

TestBUS - Testing Beyond Unit and SQL -PID2019-105455GB-C32: Research project funded by MCIN/ AEI/10.13039/501100011033 about software testing in Big Data and gamification. Main researchers: Javier Tuya and Claudio de la Riva.

During the thesis, the doctoral candidate was granted a pre-doctoral grant “Severo Ochoa” by the Principality of Asturias, starting in October 2021. The code of this grant is **PA-21-PF-BP20-184**.

I.7 THESIS ORGANIZATION

The lines of research and organization of the chapters, are summarized as follows:

Chapter I. **Introduction**. Details the research context, hypothesis, goals, methodologies, and contributions of the thesis.

Chapter II. **Background and related work**. Contains the background and related works to our research, indicating the similarities and differences of our work with them.

Chapter III. **Logical consistency maintenance: MDICA**. Details how the logical consistency is maintained when performing a modification of data (insert, update or deletion) at the conceptual level against a column-oriented database. It determines the database statements required to perform that modification of data at the database level while maintaining the logical consistency.

Chapter IV. **Logical consistency maintenance: CONCODA.** Details how to determine if a column-oriented database maintains the logical consistency after one or more modifications of data have been performed against the database. It contains an empirical validation of MDICA.

Chapter V. **Database evolution: Schema evolution: CoDEvo**. Details how the database schema must evolve after a change in the requirements that affects the conceptual model. It provides the actions to perform in the schema by means of a MDE approach, as well as an analysis of the most common changes in the conceptual model detected in open-source projects.

Chapter VI. **Final Remarks**. Summarizes the conclusions of the thesis and the future work by means of new research lines focused on continuing working on the

framework for database evolution as well as new scenarios to cover in both the logical consistency maintenance and the evolution of the database.

Appendices. Contains the appendices of the thesis

II BACKGROUND AND RELATED WORK

This chapter describes the background and the related work of this thesis. In the background subsection we describe different concepts and technologies from databases and model driven engineering. In the related work subsection, we detail research work related to our work.

The background will focus on databases and model-driven concepts. Regarding the databases, we first will describe general terminology applied to all types of databases to then focus on NoSQL databases, particularly in column-oriented ones, finishing with a description of concepts and technologies from Apache Cassandra, the most used column-oriented database. Regarding model-driven concepts, we will focus on different concepts, from the more general ones related to models to the more specific ones related to the model-driven engineering methodology.

The related work will be focused on research works that strictly related to our research, either by similarity or due to them being used in our research.

II.1 BACKGROUND

In this section we focus on detailing concepts related to databases and model-driven engineering. We first introduce the history and description of the currently used databases in subsection II.1.1. In subsections II.1.2 and II.1.3 we detail several general concepts related to all types of databases. Starting from subsection II.1.4 we focus on NoSQL databases, particularizing in column-oriented databases in subsections II.1.6 and II.1.7. In subsection II.1.8 we detail the differences between the physical consistency and the logical consistency. In subsection II.1.9 we describe what model-driven engineering means, providing more specific detail in subsubsections II.1.9.1 and II.1.9.2 to model transformations and model abstraction levels.

II.1.1 Databases: History and introduction

Databases have been used alongside applications since the 60s, where the first types of databases were designed by Charles Bachman [36]. These databases followed a hierarchical model similar to a tree: starting from a root record, each piece of data in the database had a parent record. These databases had several problems such as being difficult to use, not having any theoretical foundation and there were difficulties differentiating the logical model and the physical model.

Due to the aforementioned problems, these databases were replaced in the 70s by the relational databases [37]. These databases store the data in structured tables through rows and columns. Each column is assigned one data type such as text, integer or Boolean which determines the type of data that can be stored in that column. The main characteristic of a relational database is that the tables that compose the logical and physical model of the database have relationships between each other, which allows a normalized schema where there are no data duplications besides primary key values. These relationships are determined using foreign keys, which store in a second table the value of the primary key stored in another one. In order to maintain data integrity, these databases allow the implementation of integrity constraints in order to ensure that these foreign keys are storing values that are also stored in the table that they are referencing. The most popular DBMSs use the SQL language, existing differences between the operations supported in each DBMS.

The relational model was proposed in 1970 to model the data of a relational database [38]. This model allows to represent the data in terms of tables and relationships which could be implemented in a database. In 1976, a higher-level model was proposed: the entity-relationship model (ER model) [39]. In the entity-relationship model it is easier to know the relationships between entities as they are explicitly defined along with their cardinality. The relational model is mainly used for helping in the implementation of relational databases. On the other hand, the entity-relationship model is more widely used, being also used non-relational databases during the design of their data models as a way of representing a conceptual view of the system.

During the 90s, IBM released the architecture of data warehouse [40]. Data warehouses were created due to the problems that users experienced when they needed to obtain useful information for business decisions from their data sources such as:

- Databases designed for short and predictable-update transactions.
- Legacy data that was obtained using several types of applications.
- Heterogeneous data.
- Complex queries against normalized databases.
- New requirements to analyse more complex data.

Data warehouses were able to offer users the required information to make decisions based on their needs, which we refer as data engineering [41]. Data warehouses offer several features such as organizing the data in specific subjects, integrating data from all databases of the organization, historical views of the data and no volatility, as they show operation data at a given moment.

In the 2000s, relational databases experienced performance issues as they were not able to properly manage the exponential growth of data that occurred during this decade [11]. One of the alternatives that raised were new types of databases called **NoSQL databases**, which in addition to higher performance when managing this growth of data they allow horizontal scalability, improving their performance and assuring reliability. Compared to relational databases they have simpler data models and simpler query languages or APIs to query data. However, they lack mechanisms to manage data consistency of repeated data among the database as well as integrity constraints of the data.

In the next subsections we describe general terminologies related to databases, focusing after that on NoSQL database, particularizing on column-oriented databases which are the main focus on this thesis.

II.1.2 Data engineering

The data that are stored in the database are usually analysed to obtain useful knowledge for the owners of the data. Data engineering is a concept that summarizes the methods that perform the actions required to provide data for business intelligence, data science analysis and machine learning algorithms.

A systematic overview of several achievements reached by several data engineering methods is presented in [42]. These achievements are divided in four generations: two past ones, the current one and a future one. It is important to note that a generation does not replace the previous one, but instead adds more value to them. In the following list we briefly describe each generation:

1. In the first generation the methods were focused on increasing the data quality or transforming the data into something that was required. Some of the tasks approached

in this generations were *data exploration*, *schema extraction*, *elimination of duplication*, *data integration* and *datatype transformation*.

2. In the second generation, several methods were created to define data engineering pipelines that are continuously being executed. These pipelines allow the users to defined processes that combine different algorithms for pre-processing of the data.
3. In the third generation, which is currently on-going, the objective is to choose the most suitable method for a task from the ones that already exist from the previous generations. Currently, it is up to the user to choose these methods, although there is on-going to research to help users in this regard.
4. Regarding the future fourth generation, several problems are described to be solved in the future such as some tasks requiring the knowledge of domain experts to be solved. In [42] it is proposed that in the future, part of the tasks from previous generations, such as data curation, are done automatically through an intelligent toolset.

II.1.3 Database properties and CAP theorem

In this subsection, we describe different properties that exist for databases as well as the CAP theorem.

II.1.3.1 ACID properties

Relational databases ensure the ACID properties which are: Atomicity, Consistency, Isolation and Durability [43]. These properties ensure different concepts related to the transactions that are executed against a relational database:

- Atomicity: When performing a transaction, either every data change from the transaction is executed against the database, or no data change is executed.
- Consistency: The transaction always leaves the database in a consistent state, which is when it conforms to the integrity constraints that are implemented in the database.
- Isolation: A transaction is not affected by other transactions that are executed at the same time.
- Durability: A transaction that has been successfully executed against the database will persist even in the future there are failures in the database.

These properties are not ensured in distributed systems such as NoSQL column-oriented databases, which, on the other hand, ensure the BASE properties.

II.1.3.2 BASE properties

Most NoSQL databases have difficulties assuring the ACID database due to the distribution of the database. On the other hand, they ensure the BASE properties [15] which are more flexible than the ACID properties:

- Basic Availability: The database is going to answer even when there is an event of failure.
- Soft State: The data of the database can change without intervention from a client application due to eventual consistency.
- Eventual consistency: It ensures that eventually all nodes will store the same data. Note that this consistency is different from the one of ACID. The consistency from BASE refers to data stored across nodes of a physical cluster, while the consistency from ACID refers to the consistency of the data inside the database considering integrity constraints.

II.1.3.3 CAP theorem

The CAP theorem [44] denotes that there is a trade-off between the consistency, availability, and partition tolerance. It claims that that a database can only ensure two of these three properties at once. These three properties ensure the following:

- Consistency: It ensures that when requesting data to the database, the database will always return the correct answer.
- Availability: It ensures that when requesting data, the database will always answer.
- Partition tolerance: It ensures that the database can be partitioned in different nodes.

Most NoSQL assure the partition tolerance property to allow distribution of the data. This means that each NoSQL database must choose between the consistency and the availability. For instance, in Cassandra the availability is guaranteed, while in MongoDB the consistency is guaranteed. On the other hand, relational databases do not usually ensure partition tolerance, guaranteeing both the availability and the consistency.

II.1.4 NoSQL databases

As described in the introduction, NoSQL databases are a group of databases that were created as alternatives to relational databases for scenarios where relational databases have performance problems [45]. It is important to note that NoSQL databases do not replace relational database, in fact they can work together in hybrid database systems [46].

Depending on the data model, they are classified in key-value oriented, document-oriented, graph-oriented, and column-oriented [3]. In the following paragraphs we briefly describe each of them:

- Key-value oriented databases are schema-less databases where the data are stored in key-value associations. The simplicity of this database makes it a great option for retrievals of data required by applications working like a cache [45]. Examples of key-value-oriented databases are Redis, Voldemort, Riak, or Dynamo.
- Document-oriented databases use structured data files (XML, JSON or YAML) to store the data. The documents are schema free so each document can store properties that are not in other documents. Examples of document-oriented databases are MongoDB, Couchbase or CouchDB.
- Graph-oriented databases are composed of nodes with relationships between each other. They have some similarities to relational databases as they store relationships. However, they improve the performance of relational databases as they find the most optimal path when querying data of relationships. They also lack a schema, as each node can have different properties from other nodes of the same set. Examples of graph-oriented databases are Neo4J, JanusGraph or TigerGraph.
- Column-oriented databases are composed of tables with columns and rows, like relational databases but without relationships between tables. As this type of database will be the main focus of the thesis, they will be described in detail in the following sections.

Table 1 displays a summary of the most relevant properties of each database regarding the problems addressed in this thesis.

Database type	Data stored	Normalization	ACID	Distributed System	Examples
Relational database	Tables with columns, rows and relationships	Yes	Yes	No	MySQL, PostgreSQL, Microsoft SQL Server
NoSQL Column-oriented	Tables with columns and rows	No	No	Yes	Cassandra, HBase, BigTable
NoSQL Document-oriented	Documents (JSON, XML or YAML)	Possible	No	Yes	MongoDB, Couchbase, CouchDB
NoSQL Graph-oriented	Nodes and edges	Possible	Yes [47]	Partially [48]	Neo4J, JanusGraph, TigerGraph
NoSQL Key-value oriented	Hash tables	No	No	Yes	Redis, Voldemort, Riak, Dynamo

Table 1 Database comparison summary

II.1.5 Query languages for databases

Database systems provides languages to query and manipulate data stored in them. The most known of these languages is SQL [49], the query language used for most relational databases. The manipulation operations of SQL are mainly *INSERT* to insert rows, *UPDATE* to change data from a row and *DELETE* to delete rows. However, the most relevant feature in SQL is the wide range of queries that can be executed against the tables of the database through the statement *SELECT*. Using the JOIN operator, it can retrieve data from several related tables in a single *SELECT* statement. The operator *GROUP BY* allows to group data in order to apply them an aggregate function such as the average value.

NoSQL databases do not share a common query language, even between databases of the same type. For this reason, we will only briefly describe the languages of the most used databases of each type [2]: MongoDB for document-oriented, Neo4J for graph-oriented, Redis for key-value-oriented and Apache Cassandra for column-oriented:

- In MongoDB, the most used document-oriented database, the queries, and manipulation operations are usually executed using applications through an API, existing different libraries for several program languages such as JAVA or C#. Nevertheless, they provide a shell language called MongoDB shell that allows to interact with a MongoDB database. Like SQL, it allows to insert, delete, and update data through several methods, being all operations atomic for a single document.
- Neo4J provides the query language Cypher [50]. Although the syntax is vastly different from SQL, the developers of Cypher claim that it is remarkably similar in functionality as both databases are mainly composed of table and nodes with relationships. They use the keyword MATCH to query data, CREATE to insert nodes and relationships, SET to update them, and DELETE to delete them.
- Redis provides the module RediSearch [51] where instead of operators like SELECT in SQL or MATCH in Neo4J, they employ expressions for complex queries. They also provide commands like GET which returns the value associated to a key. They use the command SET to insert and update data and GETDEL to delete data.
- Apache Cassandra uses the query language CQL (Cassandra Query Language) whose syntax is remarkably similar to SQL in order to make easier the adaptation of developers from SQL to Cassandra. It shares with SQL the SELECT, INSERT, UPDATE and DELETE operations, although it does not contain operators such as JOIN and GROUP BY. To

execute a query (SELECT operation) against a Cassandra database, the query must meet certain requirements in its structure. The search criteria of the query (WHERE) must be exclusively composed of values assigned to key columns of the table. Specifically, if a search criterion is used in a query, all the columns that belong to the partition key of the table must contain a value in the search criteria. Additionally, the columns that are clustering key can also be part of the search criteria, although it is not required. On the other hand, non-key columns cannot be part of search criteria.

II.1.6 NoSQL Column-oriented databases

Column-oriented databases are composed of tables composed of columns and rows, being at least one column part of the primary key. In contrast to relational databases, in column-oriented databases there are no relationships. As well as other NoSQL databases, column-oriented databases allow a distributed system in which the data is distributed among several physical nodes that compound a cluster. Each of these nodes will store a part of the database, following the specifications of the developers. This distribution of the database also allows horizontal scaling, where new nodes can be added to the cluster at any time.

The internal management of the data when executing database operations such as queries also differs to relational databases. Every time a query is executed against a relational database, all rows and columns are scanned regardless of the columns needed in the query. In column-oriented databases such as Cassandra, this is avoided to improve performance through two means:

- Columns: Only the columns that participate in the query will be searched [52].
- Rows: As the data is distributed among several physical nodes through the values of the primary key, only the nodes that contain the data to be queried are requested to return the data.

In column-oriented databases, the queries can only be filtered through the values to the primary key. For this reason, each table must be specifically designed to satisfy queries that will be executed in the client application, following a query-driven approach. This is also different from relational databases, where all the columns can be used for filtering in a query. Therefore, the schema of a relational database is modelled after the specifications of the data that they are going to store, while the schema of a column-oriented databases is modelled after the requirements of the application that is going to query data from the database. This query-driven approach also implies that there are going to be repeated data among the database tables, as these data can be queried in several ways from the application. This repetition of data further implies a denormalized model, unlike the usually normalize relational data model.

The most used databases are Apache Cassandra and HBase. Although both are column-oriented database, there are also several differences between them [53]. The most relevant ones are that Cassandra does not ensure consistency among the nodes, while HBase ensures it. On the other hand, HBase has a SPOF, as a primary node exists in the cluster. Regarding the primary key, in Cassandra multiple columns can be part of it, while in HBase only one column can be part of the primary key.[50]

II.1.7 Apache Cassandra

Apache Cassandra is a NoSQL column-oriented database that was initially released in 2008 by Avinash Lakshman and Prashant Malik [54]. Like other NoSQL databases it allows to distribute the data among several nodes, which guarantees both high availability and horizontal scalability.

The architecture of Cassandra is composed of a cluster that contains several physical nodes where the data is going to be stored [55]. Cassandra has a masterless architecture; therefore, the system will keep being available when a node fails. The nodes are interconnected through a ring that allows the data to be transmitted among them. The data stored in the system will be distributed along the ring of nodes, trying to achieve the most balanced share of data possible. In addition to the distribution of the data among the nodes, there is also replication of the same data. Even if a node does not have the data for which it is queried, it will be able to respond to the query and access the required information by requesting the data to the appropriate nodes. The developer chooses the number of servers on which the data will be replicated. These data will be automatically replicated among the different database nodes. If a node fails, the data stored on that node will be replicated to other nodes in the cluster to ensure availability.

Cassandra does not ensure the consistency of the data among the nodes of the cluster. Nevertheless, Cassandra ensures that eventually all nodes will be consistent as it fulfils the BASE properties. Cassandra additionally extends this property by providing levels of consistency for its reads and writes through tunable consistency [56]. This tunable consistency allows to configure the numbers of nodes that shall be asked before returning the result to the end-user. This, however, comes at a cost, as higher the consistency level, higher response times.

The tables of a Cassandra database are located in keyspaces [57], where different options such as the replication strategy can be applied to all tables. Each table has a primary key that is composed of the **partition key** and the **clustering key**. The partition key values determine in which node the data are stored. The clustering key values determine how data are ordered in the nodes. A table must have at least a column as partition key, being the clustering key optional. The rest of columns are non-key. Each column can store only data of a type specified at the creation of the column.

As we described in section II.1.5 the search criteria of a query in Cassandra must be exclusively composed of values assigned to key columns of the table. Every column from the partition key must be in the search criteria, while the clustering key columns are optional. The operator for the partition key column can only be ‘=’ and ‘IN’. The clustering key columns additionally allow the operators ‘<’, ‘>’, ‘=<’ and ‘>=’.

II.1.8 Physical consistency vs logical consistency

When referring to consistency in a NoSQL database such as Cassandra, the most common interpretation is that this consistency refers to the consistency of the data that is stored among the nodes that compound the physical database [58], which is the one referred in the CAP theorem. Regarding the CAP theorem, Apache Cassandra ensures partition tolerance (distribution of the system) and availability. This means that the consistency of the data among the nodes of a Cassandra database is not ensured, meaning that in a specific moment the same data could be different in different nodes. On the other hand, as we described in the previous section, Cassandra ensured the eventual consistency from the BASE properties. In this thesis we refer to this consistency as **physical consistency**.

In this thesis we do not address the problems that can occur of not assuring the physical consistency. Instead, we focus on the consistency of the data that is repeated among the tables of the database, usually referred as **logical consistency** or data integrity. In Cassandra there are no mechanisms that always ensure data integrity, [42] passing the responsibility of assuring the consistency of the data in the application side [59]. In addition, as denormalization is needed in the schema of Cassandra and other column-oriented databases due to the lack of relationships

between tables, this problem becomes more difficult to solve than in a database where normalization is the preferable choice as in relational databases. We will address the problems regarding the logical consistency maintenance on chapters III and IV of this thesis.

II.1.9 Model-driven engineering

Model-driven engineering (MDE) is a software development approach that proposes the use of models to improve the quality of software products [60]. MDE uses metamodels that define the different components of software architecture. The instances of these metamodels are models, which represents a possible scenario of this architecture. MDE technologies aim to combine the following:

- Domain-specific modelling languages (DSML): They are used by developers to create models based on a metamodel. A DSML is defined using these metamodels.
- Model transformations: They are used to create another model or textual information from one or more inputs, which can be models or textual information.

In the next subsubsections we describe in detail the model-transformations as well as the abstraction level in which a model can be classified.

II.1.9.1 Model transformations

Model-transformation techniques are able to obtain target models or textual artifacts using as input other models or textual artifacts, although at least the input or the output must be a model [61]. Model-transformation therefore allows the transformations model-to text (M2T), text-to-model (T2M) and model-to-model (M2M). In M2T transformations one or more source models are used to obtain target textual artifacts such as a set of database statements to execute a schema change. In T2M transformations, a target model can be created using textual artifacts as a source. In M2M transformations, a set of source models are used to create target models of another type.

Different definitions for model transformation terms are specified in [62]. How each input component that is in the inputs can be transformed into an output component is defined through the **transformation rules**. A set of transformation rules can together define how to transform one input models to one target model. This set is called a **transformation definition**. However, this last definition is extended in [63] by proposing that multiple source and target models can participate in the transformations.

Transformations can also be classified by the abstraction level of the models that participate:

- Vertical transformations are those where the source models and the target models are from a different abstraction level.
- Horizontal transformations are those where the source and target models are from the same abstraction level.

II.1.9.2 Model abstraction levels

According to [64] there are three abstraction level that you can classify a model:

- Computation Independent Model (CIM): The models that are completely unconnected to a particular technology. For instance, a conceptual model of a database is classified in this level.
- Platform independent model (PIM): The models that are related to a technology, but they are not specific of a platform. For instance, a logical model of a database is classified in this level.

- Platform Specific model (PSM): The models that are specific of a platform from a specific technology. For instance, a physical model of a database is classified in this level.

II.2 RELATED WORK

In this subsection we describe related research and technologies that address or are related to the problems approached in this thesis: maintenance of the logical consistency and the maintenance of the inter-model consistency.

II.2.1 Materialized views

As we describe in section II.1.7, the search criteria of Cassandra queries can only contain columns from the primary key of the table. Due to these restrictions, if the same information needs to be queried using different search criteria, multiple tables that store the same data may be created, implying duplication of data. This brings the aforementioned problems that are addressed in this thesis about the logical consistency. One alternative developed by the Cassandra developers to avoid the repetition of data is the **materialized view** feature [65].

Materialized views are table-like structures that can manage the maintenance of the logical consistency from the database-side, allowing to reduce the denormalization in the schema. A primary table is used to create a materialized view, which allows to query the data stored in this primary table in more ways than what the primary table directly does. Materialized views have the following restrictions [66]:

- All the primary keys from the primary table must be part of the primary key of the materialized view.
- Only one non-key column from the primary table can be added to the primary key of the materialized view.
- If the primary table contains rows with null values in the non-key column that was added to the primary key of the materialized view, these rows are not included in the materialized view.
- There can only be one primary table for each materialized view, thereby not allowing table joins.

Whenever there is a new insertion of data in the primary table, Cassandra automatically updates the materialized view with the new data. This insertion of data is performed asynchronously, meaning that the actual insertion of data in the materialized view is delayed. After the insertion in the primary table is performed, Cassandra executes a read-repair to the materialized view to regain consistency of data between the materialized view and the primary table. The same process is executed when data is deleted in the primary table. On the other hand, there cannot be direct modifications of data in the materialized views.

Regarding performance, materialized views are not appropriate to use when low cardinality data (too many repeated values) are inserted in the database. In these scenarios, even creating secondary indexes display a better performance.

Materialized views are a good option regarding data model design when there are existing tables that contain in the primary key the columns that will be used in the search criteria of queries to be executed against the database. However, in most cases the tables of databases from real projects do not share enough columns [28], limiting the use of materialized views. The limitation

of only allowing a single primary table also prevents a normalized model of primary tables similar to a relational database. In this thesis we propose an approach that is not limited by these restrictions, allowing the maintenance of the logical consistency for all possible schema designs.

II.2.2 Implementing the join operation in Cassandra

One of the main reasons of why a normalized model cannot be used in Cassandra is because of the absence of join operators. Cassandra does not have foreign keys and relationships, making impossible to join the data of two different tables in the same query. However, there has been a work where this operation is implemented in Cassandra[67]. In this work the operation `SELECTJOIN` is implemented, which allows to relate information stored in different tables.

The `SELECTJOIN` operation requires that all columns that participate in the JOIN are part of the primary key of any table. Alternatively, the columns can be indexed as secondary indexes. A `SELECTJOIN` statement has the following syntax:

```
SELECTJOIN columns FROM nameTable1, nametable2 JOINON nametable1.pk =
                                nametable2.fk;
```

The rows returned to the user must match the criteria specified in the JOINON clause (`nametable1.pk = nametable2.fk`).

In order to implement this operation, the code of Cassandra 2.0 was modified as much as required, which according to the author was mostly performed in the package cql3. This implementation consists of transforming the SELECTJOIN operation introduced by the user in several normal SELECT operations that extract the data from the Cassandra database. Then, internally, a Java application executes a Cartesian operation through a recursive method.

The performance results of this implementation leave room for improvement, as the performance is worse than using joins in a standard MySQL database, being the Cassandra implementation twenty times worse than MySQL when querying one million rows.

Although a potential interesting solution that could add an extremely useful operation in Cassandra, the current implementation results are not good enough to consider it a valid solution for the problem of the logical consistency in Cassandra.

II.2.3 Schema design

There are multiple schema design methodologies for Cassandra [18], [19], [68]. The general schema design recommendations advise to create a table for each query that is going to be executed against the database [59]. However, only using the queries for the design of the schema may lead to incorrect designs of the schema. For instance, the design of a table must ensure that a row cannot be overwritten by mistake in future insertions of data. This can happen if the primary key of the table is compound of values that can be repeated among the rows of the tables. Let use an example of a system that stores information about products that are sold in a shop:

Suppose that there is a query requirement for obtaining information of the products by asking through the price of the product. If only this requirement is considered when designing the table, a possible design would be to create a table whose primary key is composed by only the price of the product, and the rest of information (id, name…) are stored as non-key columns. However, there could be more than one product that has the same price. Therefore, if two products are inserted with the same price, the data of the product that was inserted first would be

overwritten with the information of the second product. The solution would be to also include in the primary key of the table a column that uniquely identifies a product. However, this information is not provided by the schema of Cassandra, requiring additional information such as a conceptual model.

In order to avoid the aforementioned scenario, the schema design methodologies referenced in this section recommend using an explicit conceptual model in addition to the queries. The use of this explicit model is especially important to avoid faulty designs such as the example described in this section, as when developers only have an implicit model in their minds, they are prone to committing mistakes [29].

In the following subsections we describe in detail the following methodologies ordered by date of publication, highlighting their differences:

1. KDM: Kashlev Data Modeller [18]: Determines a schema given a conceptual model and queries.
2. NoSE: Schema design for NoSQL Applications [68]: Improves the previous methodology by optimizing the generation of the schema for a specific target application.
3. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models [19]: MDE approach that, in addition to column-oriented databases, it also provides a schema for document-oriented databases.

II.2.3.1 KDM: Kashlev Data Modeller

The first methodology was proposed in [18] which resulted in the tool KDM, which is currently unavailable to use. Their proposal is based on data modelling principles, mapping rules and mapping patterns.

They specify four data modelling principles that serve as foundation for the mapping of the conceptual model to the schema (logical model):

- DMP1 (Know Your Data): It consists of defining a normalized conceptual model.
- DMP2 (Know Your Queries): It consists of identifying the queries that must be executed.
- DMP3 (Data nesting): It consists of organizing multiple entities together in accordance with a certain criterion.
- DMP4 (Data duplication): It consists on encouraging the duplication of the data, contrary to relational databases.

The mapping rules are used to guide the design of the schema:

- MR1 (Entities and Relationships): The entities and relationships design the tables, whose instances will be the table rows. The attributes are mapped to columns.
- MR2 (Equality Search Attributes): The equality search criteria are used to compound the partition key and, optionally, part of the clustering keys. The fulfilment of this rule fulfils the query requirements.
- MR3 (Inequality Search Attributes): The inequality search criteria will be part of the clustering key.
- MR4 (Ordering attributes): They are used to define the order (ascending or descending) of the clustering keys.
- MR5 (Key attributes): The key attributes of the entities are mapped to the primary keys. These are used to guarantee row uniqueness in the table in order to avoid scenarios like the one described at the beginning of subsection II.2.3.

The mapping patterns are used for the automation of the schema design. Given a query and a conceptual model, the mapping pattern is able to define a table design.

They additionally propose a visualization technique called Chebotko Diagram to present the design of a database schema, combining the tables and the query-driven application workflow transitions.

II.2.3.2 NoSE: Schema design for NoSQL Applications

The second methodology was proposed in [68]. This work proposes an approach for schema design that recommends a schema that is optimized for a specific target application named NoSE. NoSE uses a cost-based approach that estimates the performance of the schema candidates in order to choose the one with a better result.

The outputs of NoSE are 1) a recommend schema and 2) a set of plans. These set of plans recommend how the application should use the schema when implementing queries or data manipulation operations.

NoSE employs a workload that contains the frequency of execution of the queries and updates to be executed against the database. With this workload, NoSE is able to provide high-level optimizations. The four steps for the advice of the schema are the following:

1. Candidate Enumeration: A set of possible schemas are generated. These schemas must be able to satisfy the query requirements.
2. Query Planning: Generation of a space of implementation plans for the queries that can be executed.
3. Schema optimization: A BIP (binary integer program) is generated from the possible schemas and plan spaces. In this step several possible plans that minimize the cost of the execution of the queries are chosen.
4. Plan recommendation: It chooses a single plan from the previous step set of plans chosen.

II.2.3.3 Mortadelo: Automatic generation of NoSQL stores from platform-independent data models

The third schema design methodology was proposed in [19]. This methodology, named Mortadelo, expands its scope to also consider document-oriented databases such as MongoDB. The main concept of Mortadelo is that through a MDE approach, it is able to provide with the same inputs a possible schema for both NoSQL document databases and NoSQL column databases.

The inputs are similar to the previous two approaches: a conceptual model and a set of queries. The main difference is that these inputs are inserted as models and the output are also models that conform to metamodels for both types of NoSQL databases.

They also made performance comparisons against the previous two methodologies, obtaining that the schemas designed by Mortadelo had better performance for readings than NoSE but worse than KDM. On the other hand, for write operation the schema of Mortadelo obtained better performance than KDM but worse than NoSE.

II.2.4 Schema Inference

The use of an explicit conceptual model when designing the schema of a NoSQL column-oriented database is strongly recommended, although not strictly required. Because of this, there are

developers who only think of a conceptual model without explicitly defining it. This can be a problem when either implementing new operations in the client application that modify data of the database or when evolving the schema, as the developer can forget the specific details of the implicit conceptual model. Additionally, if this implementation or evolution of the schema is done by other developers different from the ones that designed the database, it also makes more possible the commitment of mistakes. For this reason, there have been approaches that propose the creation of a normalized model from a denormalized schema of a column-oriented database.

One of these approaches [69], based on MDE, proposed to infer a conceptual model from the schema of a NoSQL database, approaching column-oriented, key-value and document databases. The architecture of this approach is based in three stages:

1. A Map-reduce operation that is used to obtain a version of an entity, creating a collection.
2. The obtained collection is inserted into a model that conforms to a JSON metamodel.
3. The schema is obtained using reverse-engineering and a model-to-model transformation that uses as input the JSON model and returns an output of a model that conforms to a NoSQL-Schema metamodel defined by the authors.

The most notable contribution in this work and most important part of the approach is the reverse-engineering process that obtains the detail of the schema. This process is composed of three parts:

1. **Building the raw schema of an entity**: The obtention of a JSON object that accomplishes the following rules:
    1. It has the same structure as the entity including all attributes, nested entities, and arrays
    2. Each primitive value is substituted by its equivalent JSON type
2. **Obtaining the Version Collection**: For each entity, only one version will be used, which is obtained using a Map-Reduce operation. The result is the Version Collection.
3. **Schema obtention**: The different components of the schema are obtained:
    1. Entities
    2. Attributes of the entities
    3. Relationships

Another approach [17] was created with a similar objective but focused on column-oriented databases. This approach, like in the previous one, obtains a renormalized model using a three-step process:

1. Generate a physical schema.
2. Identify the dependencies that exist between the attributes of the physical schema. These dependencies can be either provided by a user with knowledge of the system or automatically.
3. Renormalization of the physical schema using the depending.

II.2.5 Schema Evolution

Schema evolution work has traditionally focused on relational databases with approaches that address issues such as the evolution of the integrity constraints [70] and foreign keys [71], or

guidelines to evolve relational schemas [72]. Another topic addressed is how the database schema and the data of a relational database must evolve after an ontology change [73], similar to what we address in chapter V for column-oriented databases. However, these works are focused on relational databases, making their use for other systems such as NoSQL databases more difficult. For instance, column-oriented databases do not have relationships between tables, making the works about the evolution of the integrity constrains and foreign keys unfit for these kinds of databases. Another work focused on relational databases defined an ontology that details how the schema of a relational database must evolve from changes in the conceptual model [74].

In the following subsections we describe in more detail works related to database evolution and the schema which are focused on NoSQL, and more specifically column-oriented NoSQL databases.

II.2.5.1 Migcast, Data migration strategies and Evobench: Database evolution on NoSQL databases

Multiple works have been published in the last years focused on database evolution by the same research group [75]–[79]. They have covered a wide range of topics, focusing on NoSQL databases, especially column-oriented databases, and document-oriented databases. In this subsection we briefly describe some of their works that cover topics that are related with our work.

They developed the tool Migcast [78] which advises the developers in the process of database evolution by recommending different data migration strategies focusing on the financial cost. These strategies can be determined as they are able to predict the financial cost, depending on the cloud provided that is chosen. Migcast is able to maintain an internal cost model by considering different characteristic of the data instance, the expected workload, evolution of the schema and the prices of the cloud providers.

The specific data migration strategies were identified in another work [77],where they compare their benefits and disadvantages:

- Eager migration: The migration is performed at the moment of the data change. It is the migration strategy with more financial cost as it even migrates data that is not going to be used in the future.
- Lazy migration: The data is only migrated when there is a requirement to access these data. This strategy allows to have no migration cost when the schema evolves, although at the cost of never having the data up to date in the database.
- Incremental migration: Similar to lazy migration, but it also migrates data at certain moments of the day, ideally when data accesses are less frequent.
- Predictive migration strategy: It identifies the data that was accessed in the past and calculates how likely are certain data to be required again in the future. These data will be always kept up to date, performing as soon as possible the required data migrations.

These migrations strategies are then recommended depending on the scenario presented by the developers.

EvoBench is a conceptual framework to measure the performance of both schema evolution and data migration in document-oriented NoSQL databases [76]. This benchmark consists of the following components: 1) data model, 2) testing data, 3) schema modification to evolve it, 4) queries to execute for lazy migrations and 5) the metrics. EvoBench has three stages:

1. Benchmark Preflight: The user introduces the components described before.
2. Benchmark Execution: The benchmark is executed, measuring the executing times.
3. Benchmark post-execution: The results are analysed.

II.2.5.2 U-Schema: A unified metamodel for NoSQL and relational databases

A recent work [80] has addressed the need of using a metamodel to represent the schemas of NoSQL databases. Although column-oriented databases have an explicit schema, other NoSQL databases are schema-less, where the schema is implied in the data. To address this, they have defined the metamodel "U-Schema", which is the first unique metamodel that can be used to define the model of all NoSQL database types. The main contributions of this work are the following:

- The definition of a unified metamodel to represent database schemas for relational systems and NoSQL systems. This metamodel is mainly composed of collection of types, which are either entity types or relationship types. These types include structural variations, which can be eighter logical features (references and keys) or structural features (attributes and aggregations). The attributes can be of diverse types as well: primitive types, lists, tuples, sets or maps.
- The identification of two types of mapping: *forward mapping* which maps an element from a NoSQL or relational model to U-Schema and *reverse mapping* which is a mapping in the opposite direction. Reverse mappings are required when the U-Schema has elements that are not unique for a particular data model. Through the definition of these two types of mappings, they have established the term *canonical mapping*, which ensures that each element of a data model corresponds to an element of U-Schema, assuring forward mappings. This also ensures that a U-schema generated from a data model can reproduce this original database, assuring bidirectionality. On the other hand, reverse mappings are not always ensured, requiring in some scenarios to define reverse mapping for specific data models, not being this part of the scope of the work.
- A strategy to extract unified schemas from databases. They improve past works about schema extraction by also considering account scalability and performance. This is not applied to column-oriented databases, as they already have a defined physical schema.
- Insights on how U-Schema can be used for several processes such as implementation of database utilities in environments with multiple databases:
    - Definition of a generic query language.
    - Database migrations.
    - Definition of a generic schema language.
    - Generation of datasets for testing purposes.
    - Schema visualization.

This work provides a remarkably interesting contribution which we could have used in several parts of our work, especially regarding the schema evolution part. However, this work was published too recently, after we had already defined the approaches exposed in this thesis, not being to use their contributions.

II.2.5.3 The Orion language: Towards a taxonomy of schema changes for NoSQL databases:

A work addressed schema evolution for NoSQL databases [81], focusing on a general approach for both MongoDB and Apache Cassandra. They provide a taxonomy of several generic changes in the schema which can then be translated to the schema of the appropriate database. They use the advances of their previous work "U-schema" which was described in the previous

subsection to define these changes. The main purpose of this work is to support schema changes in a platform-independent way. This would allow developers to work in a multi-database environment, not requiring them to be specialised in a specific database.

The main differences with our approach are that we focus on only column-oriented databases and the source of the change in the schema is a change of the requirements of the system that modify the conceptual model. We have also specifically addressed changes detected in open-source projects, providing information about the recurrence of each change. We have also based our solutions on how the developers of these projects have addressed the evolution of the schema in order to provide a schema design similar to what a human developer would create.

III LOGICAL CONSISTENCY MAINTENANCE: MDICA

This chapter details the preventive approach developed for the maintenance of the logical consistency in column-oriented databases when a modification of data is performed against the database that we named MDICA. MDICA was first introduced in a work published in Journal of Web Engineering [25] which was extended in another work published in Computer Standards & Interfaces [26], which is the main source of the content detailed in this chapter.

Section III.1 introduces the problems regarding the maintenance of the logical consistency in column-oriented databases. Section III.2 describes several definitions that are going to be used in the rest of this chapter. Section III.3 introduces the general approach to maintain the logical consistency given a modification of data (insertion, deletion, or update). Section III.4 introduces the case study used for the examples of the application of MDICA for each type of modification of data. Section III.5, III.6 and III.7 detail how to maintain the logical consistency for each type of modification of data: inserting, deleting and updating data, respectively. Section III.8 contains the experimentation using MDICA.

III.1 INTRODUCTION

As we described in chapter I of this thesis, the design of a column-oriented database schema such as Cassandra follows a query-driven approach in which the data is organized based on queries. This means that, in general, each table from a column-oriented database is designed to satisfy a single query [59]. If a single datum is retrieved by more than one query, the tables that satisfy these queries must store this same datum. This implies a denormalized model, which contrasts with relational databases where the model is normalized. Due to this, any modification of data (insert, delete or update) requires the execution of several database statements in the tables where the data are stored in order to maintain the data integrity, which we refer in this thesis as **logical consistency** of the data. It is important to note that the logical consistency is different from the physical consistency, which is usually the “consistency” most referred in NoSQL column-oriented databases (see II.1.8). In this chapter and the following one, we only address problems regarding the logical consistency.

NoSQL column-oriented databases do not have mechanisms to ensure the logical consistency in the database, unlike relational databases, so it needs to be maintained in the client application that works with the database [82]. This is prone to mistakes that could incur in the creation of inconsistencies of the data. As the number of tables with repeated data in a database increases, so too does the difficulty of maintaining the logical consistency. In this chapter we introduce an approach named MDICA for the maintenance of the logical consistency for an insertion, deletion, or update of a tuple at the conceptual level which is consequently performed at the database level through the execution of database statements. MDICA determines these database statements along with other necessary procedures to maintain the logical consistency. Both for the definition of the tuple to insert, delete or update at the conceptual level and for the identification of the tables, we use an explicit conceptual model that has a connection with the logical model or schema (model of the Cassandra tables) [18].

In the next subsections we define the specifications of MDICA for each type of modification of data: insertion, deletion, and update of tuples. Section III.2 contains the definition and description of the basic notation used in the chapter. Section III.3 contains the general specification of MDICA that is shared among the three types. Sections III.4, III.5, III.6 contain the

detail of how MDICA addresses the maintenance of the logical consistency for insertions, deletions, and updates of data. Section III.7 contains an evaluation of MDICA through its application to three case studies.

III.2 DATA MODELS AND NOTATION

A data model [1] 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 this chapter.

Conceptual data model. - A conceptual data model or conceptual model, denoted as $\mathcal{M}$, which represents concepts of the system to be developed, is composed of entities, denoted as $e \in Ents(\mathcal{M})$, and relationships between those entities, denoted as $R_{\{ei\}} \in Rels(\mathcal{M})$ where $e_i \in Ents(\mathcal{M})$. Entities and relationships 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 relationship $R_{\{ei, ej\}}$, 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) [39].

Schema. - A schema, logical data model or logical model, denoted as $\mathcal{L}$, is composed of tables, denoted as *Tabs(S)*, which represent how data is stored in a column-oriented database. A table in a schema $\mathcal{S}$, denoted as $t \in Tabs(S)$, is a collection of ordered columns, denoted as *Cols(t)*. At the schema 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. In the rest of column-oriented databases, the primary key columns are of the same type, all denoted as *Key(t)*. 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 modelling process e.g. Chebotko et al. [18] or Mior et al. [68] 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 $\mathcal{Map}(\mathcal{M},\mathcal{S})$, is the association established between a conceptual model and a schema. $\mathcal{Map}(\mathcal{M},\mathcal{S})$ provides information about:

(1) associations between attributes of entities or relationships, 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 several 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.

Figure 3 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.

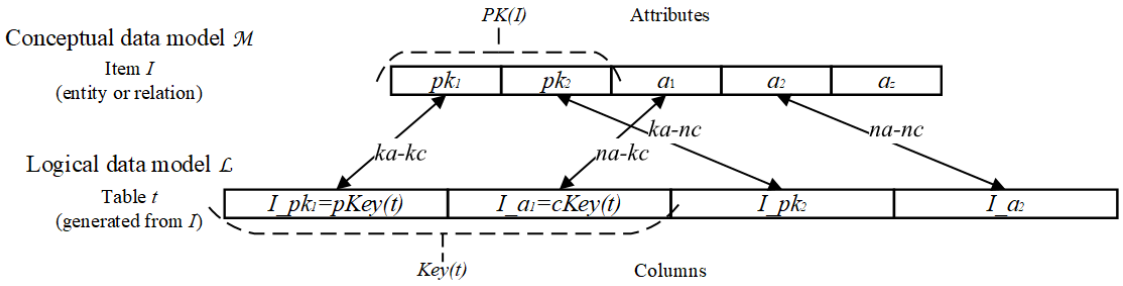


Figure 3 $\mathcal{Map}(\mathcal{M},\mathcal{S})$ between an item I (entity or relation) and a table t generated from this item

Modification of data. - A modification of data is an insertion, update, or deletion of tuples at the conceptual model level, which are consequently performed at the database level to target tables *t* through database statements. Therefore, from a tuple *tp(I)* to be inserted, deleted or updated, several column-data pairs *r(t)* to be inserted, deleted or updated in different target tables *t* are obtained from MDICA The generalization of this is detailed in section III.3, while the formalization of each type of modification of data is presented in sections III.5, III.6 and III.7.

III.3 LOGICAL CONSISTENCY BASED ON CONCEPTUAL AND LOGICAL DATA MODELS

This section addresses the problem of the logical consistency maintenance in a column-oriented database for modifications of data. In each of the following sections we will focus solely on one type of modification of data: insertions in section III.5, deletions in section III.6 and updates in section III.7. The logical consistency maintenance process leverages the schema s generated

from conceptual models and application queries through a set of mapping rules or patterns in the modelling process of column-oriented databases [18], [19], [68]. Figure 4 depicts the integration of the modelling process (on the left) and the data integrity maintenance process, MDICA, devised in this chapter (on the right).

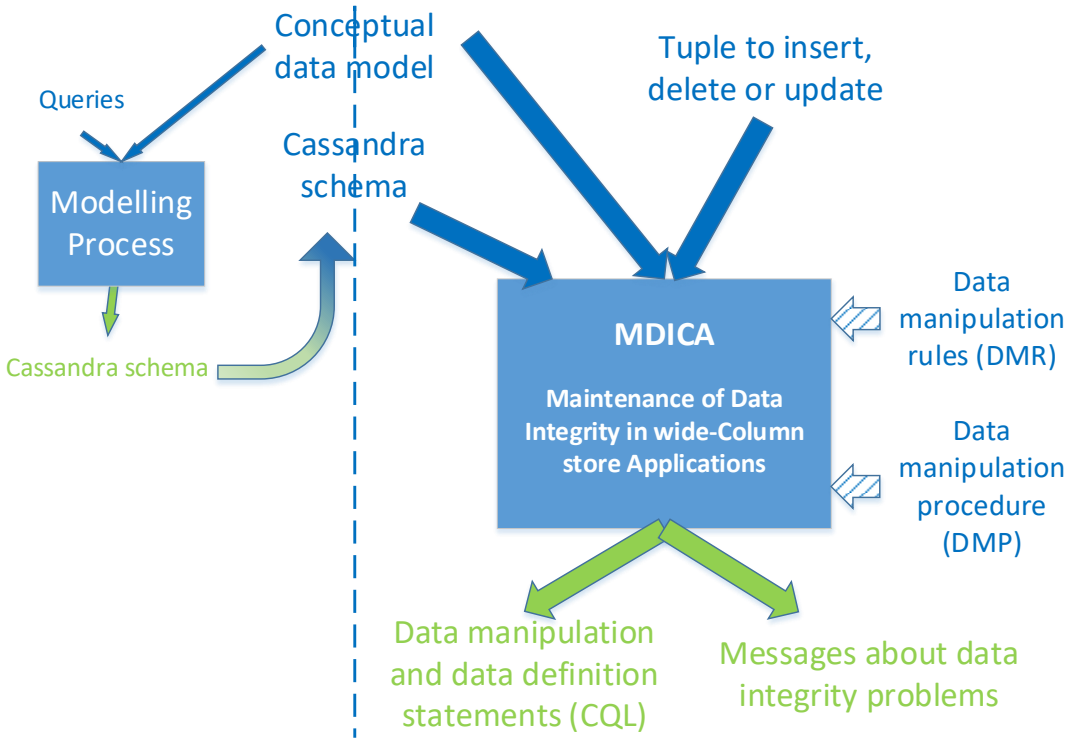


Figure 4 Integration of Modelling and Data Integrity Maintenance processes

In all types of modifications of data, the maintenance of the logical consistency will consist of (1) a tuple, which represents the data to insert, delete or update, (2) a conceptual model and (3) a schema. In addition, there could be more inputs specific of each type that are detailed in their specific section. MDICA will generate the list of ordered data manipulation statements to execute against the database by applying two concepts that will be defined in the following sections:

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

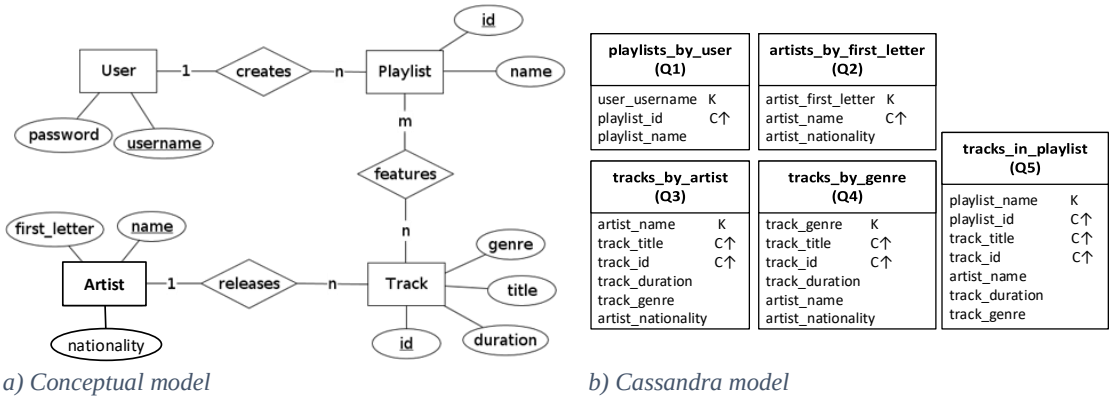
Sometimes, the generated database statements will also retrieve data from tables in the database or even modify the logical data model (data definition statements). Moreover, MDICA will provide several types of messages to inform the users about potential data integrity problems.

III.4 Introductory case study

In order to illustrate how data integrity has to be maintained, we use a case study of a digital music store interacting with an information system in Cassandra, adapted from a Datastax tutorial [83], that we will also refer to throughout the remaining sections of this chapter.

The conceptual model (Figure 5.a) that represents users, playlists created by users, which are featured by tracks, tracks available in the system and artists who release tracks.

The schema, (Figure 5.b), result of the modelling process, includes a table for each query in the application. In this case study, 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).



a) Conceptual model *b) Cassandra model*

Figure 5 Illustrative case study: a digital music store

Table 2 displays the mapping, $\mathcal{M}ap(\mathcal{M},\mathcal{S})$, 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.

	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 relationship "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											
45rtista_first_letter		na-kc									
45rtista_name	ka-kc										
45rtista_nationality			na-nc								
Table tracks_by_artist (Q3) from relationship "releases" between Artist and Track											
45rtista_name	ka-kc										
track_title					na-kc						
track_id				ka-kc							
track_duration							na-nc				
tranck_genre						na-nc					
45rtista_nationality			na-nc								
Table tracks_by_genre (Q4) from relationship "releases" between Artist and Track											
track_genre						na-kc					
track_title					na-kc						
track_id				ka-kc							
track_duration							na-nc				
45rtista_name	ka-nc										
45rtista_nationality			na-nc								
Table tracks_in_playlist (Q5) from relationships "releases-features" between Artist, Track and Playlist											
playlist_name									na-kc		
playlist_id								ka-kc			
track_title					na-kc						
track_id				ka-kc							
45rtista_name	ka-nc										
track_duration							na-nc				
track_genre						na-nc					

Table 2 $Map(\mathcal{M},S)$ for conceptual and logical data models

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

Situation 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 relationship (releases) between these entities. According to the mapping (Table 2), in the schema, 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 challenging task if there are dozens of tables with data repeated and it is conducted 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*.

Situation 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 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, updating, or deleting data in each table impacted by the change and/or creating and populating new tables to obtain the information required.

III.5 MAINTENANCE OF THE LOGICAL CONSISTENCY FOR DATA INSERTIONS

In this section, we define the DMR (Data Manipulation Rules) and DMP (Data Manipulation Procedures) that correspond to all types of insertions. In subsequent subsections, these rules and procedures will be particularized within the scope of inserting tuples: in an entity (III.5.1), in a relationship with cardinality 1:1, 1:n or n:m (III.5.2) and in combinations of relationships with a variety of cardinalities (III.5.3).

The first step is to identify the tables in a schema 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 for Insertions (DMR-I):

Definition 1 (Data Manipulation Rule for Insertions, DMR-I).- Given a conceptual model M, an insert operation on an item I (entity or relationship in M), a schema S. A DMR determines:

1) the *Map(M,S)* between M and S 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,S)*, the set of target tables $TT \in Tabs(S)$ which are impacted by the operation on the item I,
3) the potential threats to the maintenance of the logical consistency if any target table is not well-modelled.

Depending on the mapping between M and S, 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 schema 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 schema 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 for Insertions, DMP-I).- Given a tuple tp (I) to insert, the conceptual-logical data model mapping Map(M,S) between M and S, and the set TT of target tables determined by DMR-I. DMP-I determines:

1) according to *Map(M,S)*, 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,S)*, or retrieved from the database,
3) for each table $tt \in TT$, the ordered list of manipulation operations (insert, update or select) to maintain the logical consistency in *TT*,
4) other additional messages, specific of the procedure, where applicable.

Algorithm 1 describes this procedure (Definition 2):

Algorithm DMP-I
Input: a tuple *tp(I)* to insert, the conceptual-logical data model mapping between M and S $\mathcal{Map(M,S)}$, and a set *TT* of target tables
Output: database statements and messages

suitable = Analysis (*tp(I)*, $\mathcal{Map(M,S)}$)
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 $tt \in TT$
  ForEach $c \in Cols(tt)$
    data = FindData (*c*, *tt*, *tp(I)*, $\mathcal{Map(M,S)}$)
    row = AddPair(*c*, data)
  End ForEach
  GenerateStatement (*tt*, row)
End ForEach

Algorithm 1 Data Manipulation Procedure for Insertions (DMP-I)

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-I 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 relationship, 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.

III.5.1 Insertion of 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 $e \in Ents(\mathcal{M})$. DMR-IE determines the set of target tables $TT \subseteq Tables(S)$ generated from *e*. Each $tt \in TT$ is well-modelled if $\forall pk \in PK(e), \exists k \in Key(tt)$ / an association *ka-kc* in $\mathcal{Map}(\mathcal{M},\mathcal{S})$ 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 $e \in Ents(\mathcal{M})$. DMP-IE sets, for each column *c* of each target table $tt \in TT$, data taken exclusively from pairs (a_i,v_i) in *tp(e)*.

The general algorithm DMP (Section III.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 S Map(M,S)
Output: data for c

If tp(e) has value v for attribute a corresponding to c        case 1
  Return v
Else If c ∈Key(tt)                                            case 2
        GenerateMessage (Error,AKC)
        Abort
      Else                                                    case 3
        GenerateMessage (Warning,ADC)
        Return null
    End If
End If
```

Algorithm 2 FindData for insertions of an entity

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 conducted, 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 III.4).

DMR-IE determines the mapping between conceptual and schemas (that can be seen in Table 2) 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 conduct 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")

III.5.2 Insertion of a tuple into a relation

Next, we will deal with inserting a tuple into a relationship at a conceptual model level. In definitions of specific DMR and DMP, we will consider different cardinalities of binary relationships (1:1, 1:n and n:m) and illustrate them with an example.

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

- *DMR-IR.1* determines the set of target tables $TT_{Ents} \subseteq Tables(S)$, 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 $TT_R \subseteq Tables(S)$, 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 $\mathcal{M}ap(\mathcal{M},S)$ 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 $\mathcal{M}ap(\mathcal{M},S)$ 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 $\mathcal{M}ap(\mathcal{M},S)$ 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 relationship, DMP-IR).- Definition 2 is applied where item *I* is a relationship between entities e_1 and e_2, $R_{\{e1,e2\}} \in Rels(\mathcal{M})$. 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 $lookupTable \in Tabs(S)$.

The general algorithm DMP (Section III.5) 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 S Map(M,S)
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,S))
  If (lookupQuery is executable)                                  case 4
    GenerateMessage (Information,ADC-S)
    Return data=lookupQuery
  Else
    lookupQuery = RecreateQuery (c, tp(R{e1,e2}), Map(M,S))
    If (lookupQuery is executable)                                case 5
      GenerateMessage (Information,ADC-C)
      Return data=lookupQuery
    Else
      If c∈Key(tt)                                                case 2
        GenerateMessage (Error,AKC)
        Exit
      Else                                                        case 3
        GenerateMessage (Warning,ADC)
        Return null
      End If
    End If
  End If
End If
```

Algorithm 3 FindData for insertions of a relationships

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 $lookupTable \in Tabs(S)$ 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 $\mathcal{L}$ 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 the logical consistency 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 $\mathcal{L}$, so in subsequent insert operations, the process will be as in case 4.

Example 2.- Consider the insertion of a tuple into the relationship "releases" between entities Artist and Track in the conceptual model in the introductory example (Section III.4).

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 relationship "releases". Both tables are well-modelled provided that the relationship 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 the logical consistency:

Warning(ATA): Absence of target tables for entity Track
Information(ADC-S): Absence of data for column artist_*nationality*.
Select artist_nationality from table *artist*s_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,* which 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 the logical consistency:

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)

III.5.3 Insertion of a tuple into 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 relationships and an example.

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

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

DMP-IRR.2 determines the set of target tables $TT_{RR} \subseteq Tables(S)$, generated from combinations of chained relationships in *RR* with cardinality 1:1, 1:n and n:m. Depending on combinations of the cardinality of relationships, 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 $Map(\mathcal{M},S)$ 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 $Map(\mathcal{M},S)$ 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 relationships and both must fulfil 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 $Map(\mathcal{M},S)$ exists between *pk* and *k*.

Combination of 1:1, 1:n and n:m relations: an association *ka-kc* in $\mathcal{Map(M,S)}$ exists between a key column of tt_{RR} and every key attribute of: any entity in 1:1 relationships, detail entities in 1:n relationships 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 relationships ("releases" and "features") between entities Artist, Track and Playlist at the conceptual model level in the introductory example (Section III.4).

DMR-IRR determines target tables considering two complementary rules:

- DMR-IRR.1 implies the application of DMR-IR to relationships "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 relationship "features". Table *artists_by_first_letter* (generated from entity Artist) and tables *tracks_by_artist* and *tracks_by_genre* (generated from relationship "releases") are determined as target tables.
- DMR-IRR2 determines as a target table *tracks_in_playlist*, generated from the relationships chained "releases" and "features", a combination of a 1:n relationship (*Artist R Track*) and an n:m relationship (*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).

III.6 MAINTENANCE OF THE LOGICAL CONSISTENCY DATA DELETIONS

The inputs for the maintenance of the logical consistency when deleting data will consist on (1) the entity or relationship whose instances are to be deleted, (2) a conceptual model, (3) a schema and (4) the criteria used to determine whether the instance of an entity or relationship is deleted from the database or not, which we name **deletion criteria**. MDICA will generate the list of ordered data manipulation statements to execute against the database by applying data manipulation rules for deletion of tuples (DMR-D) and data manipulation procedures for deletion of tuples (DMP-D) that we define later in this section.

Like in the insertions of data, the generated databases statements may retrieve data from the database or modify the schema in order to preserve the logical consistency and execute the intended deletion of data.

In this section, we define the rules and procedures in general terms and in a subsequent section, they will be particularized within the scope of deleting data that either belongs to an entity (Section III.6.1) or to a relationship with cardinality 1:1, 1:n or n:m (Section III.6.2).

The first task to perform is to identify the tables where data must be deleted in accordance with the deletion criteria. As like for the insertions, we define the concept of Data Deletion Rule (DDR) for this:

Definition 9 (Data Manipulation Rule for Deleting a tuple, DMR-D). – Given a conceptual model *M*, a delete operation on an item *del* (entity or relationship in *M*), a schema *S*. A DMR-D determines:

1) the *Map(M,S)* between M and S 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,S)*, the set of target tables *TT* $\in$*Tabs(S)* which are impacted by the deletion on the item I,
3) the potential threats to the maintenance of the logical consistency if any target table is not well-modelled.

Depending on the mapping between M and S, 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 delete operation against the schema 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 schema 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 delete the rows that fulfil the deletion criteria. We define the Data Manipulation Procedure for Deletions (DMP-D) to generate them:

Definition 10 (Data Manipulation Procedure for Deletions, DMP-D).- Given a deletion criteria *w*, the entity or relationship whose instances need to be deleted *del*, the conceptual-logical data model mapping *Map(M,S)* between M and S, and the set *TT* of target tables determined by DMR-D. DMP-D determines:

1) for each target table *tt* $\in$*TT*, the necessary data taken from the deletion criteria or retrieved from the database to execute the correct DELETE operation in the table,
2) for each table *tt*, the ordered list of manipulation operations (SELECT or DELETE) to maintain the logical consistency in TT,
3) other additional messages, specific of the procedure, where applicable.

The algorithm DMP-D, included below, describes this procedure (Definition 10):

Algorithm DMP-D
Input: an entity or relationship del with a where W clause, the conceptual-logical data model mapping between M and S Map(M,S), and a set TT of target tables
Output: database statements and messages

ForEach target table tt ∈ TT
 ForEach c ∈ Key(tt)
  value = findData (c, W, Map (M, L))
  tableWhere = AddPair(c, value)
 End ForEach
 ListPairValues = keyValues (tableWhere)
 Statements = {}
 ForEach pair (a, v) ∈ **ListPairValues**
  Statements <- GenerateAllStatements (pair (a,v) , ListPairValues, tableWhere, tt)
  RemovePair (pair(a,v), ListPairValues)
 End ForEach
 Return Statements

Algorithm 4 Data Manipulation Procedure for Deletions (DMP-D)

For each table that belongs to the target tables ($tt \in TT$), DMP-D analyses each key column *c* and creates a pair of the column and either a single value or a set of pair of values (**AddPair**):

- When it is a single value, it means that for all statements that will be executed against the target table, this value will always be assigned to *c*.
- When it is a set of pair of values, the pair is composed of 1) a value to assign to c and 2) a pair composed of a key column and a value. This key column is mapped to a key attribute of an entity from the conceptual model, which is either the item *del* (entity or relationship) or related to it through relationships in the conceptual model. When generating the required delete statements (**GenerateAllStatements**), the value 1) will only be assigned to statements where the value in the pair 2) is assigned to the key column. Previously, all the 2) pairs are inserted in a list (**keyValues**).

The created pairs for each column are stored in the list **tableWhere**. Right after **tableWhere** contains all pair values for *tt*, function **keyValues** inserts in the list ListPairValue unique pairs of a key column and a value. Then, for each pair all the possible statements that will contain that pair generated (**GenerateAllStatements**) using the appropriate values for the rest of the columns that are obtained from ListPairValues and tableWhere. For instance, if there are two attribute keys a and b from two different entities, with each one having two different values assigned to them, there would be 4 statements in order to cover all possible combinations.

III.6.1 Deletion of a tuple from an entity

For the deletion of instances of entities, deletion procedures must be executed in each table that contains information of the entity ($\exists c \in tt, c_{Entity}$). That is that a table is assigned as a target table if just one column of it is mapped to an attribute of the entity.

The general algorithm DMP-D is applied but FindData is specialized for deleting an entity. Inside this function there are calls to several sub-functions that do the following:

- **correspondingAttribute (c: Column, Map (Cm, Lm))**: Obtains the attribute associated to the column *c*.
- **entity (a: Attribute)**: Obtains the entity associated to the attribute *a*.
- **CreateQuery (c: Column, w: Tuple, Map (Cm, Lm))**: Creates a query to execute against the database to obtain the value required for *c*, using as criteria the values inside tuple *w*.

- **CreateQuery (a: Attribute, w: Tuple, Map (Cm, Lm))**: Creates a query to execute against the database to obtain the value required for *a,* using as criteria the values inside tuple *w*.
- **RecreateQuery (c: Column, a: Attribute, w: Value, Map (Cm, Lm))**: Creates an appropriate table to execute a query against the database to obtain the value required for *c,* using as criteria the value *w* applied to the attribute *a*. Then, it executes the query against the created table.
- **AssignKeyPair (key: Value, pkValue: Pair)**: Creates an association of a value key to be assigned to the column c of the function FindData. The *pkValue* pair consists of an association between a primary key of an entity and a value of that key, which we name as ***keyValue***. This will be used in the DMP algorithm to associate the value *key* in the DELETE statement when the value ***keyValue*** is also used.
- **AssignValues (v : Value)**: Assigns the value v to a list that calls this function.

```
Function FindData
Input: a column c, a target table tt, a deletion criteria W that contains values v associated to attributes from entity e, the conceptual-logical data model mapping between M and S Map(M,S)
Output: value for c (cases, 1, 2) or set of values (cases 3 and 4)
a = correspondingAttribute (c, Map (M,L))
ec = entity (a)
keyEc = key (ac)
If W has value v for attribute a corresponding to c                                    case D1
        Return v

Else If w has values v for attributes key of ec                                        case D2
     lookupQuery = CreateQuery (c, w, Map(M,S))
        If (lookupQuery is executable)                                                 case D2-1
                GenerateMessage(Information,ADC-S)
                Return data=lookupQuery
        Else
                lookupQuery = RecreateQuery (key(ec), w, Map(M,S))
                If (lookupQuery is executable)                                          case D2-2
                        GenerateMessage(Information,ADC-C)
                        Return data=lookupQuery
         Else                                                                          case D5
                GenerateMessage(Error,AKC)
                Exit
        End If
Else if w has not value v for attribute key of ec and ec == e                          case D3
         lookupQueryKey = CreateQuery (key(e), w, Map(M,S))
         If (lookupQueryKey is not executable)
                lookupQueryKey = RecreateQuery (key(e), w, Map(M,S))
                If (lookupQuery is not executable)                                     case D5
                        GenerateMessage(Error,AKC)
                        Exit
                End If
        End If
        listValues = {}
         ForEach value in lookupQueryKey
                lookupQuery = CreateQuery (c, value, Map(M,S))
                If (lookupQuery is executable)                                         case D3-1
                        GenerateMessage(Information,ADC-S)
                        listValues <- AssignKeyPair (lookupQuery, {key(ec), value})
                Else
                        lookupQuery = RecreateQuery (c, value, Map(M,S))
                        If (lookupQuery is executable)                                 case D3-2
                                GenerateMessage(Information,ADC-C)
                                listValues <- AssignKeyPair (lookupQuery, {key(ec), value})
                        Else
                                GenerateMessage(Error,AKC)
                                Exit
                        End If
```

```
                End If
    End ForEach
    Return {data} = listValues

Else                                                          case D4
        If w  has not value v for attribute key of e:
                lookupQueryKey = CreateQuery (key(e), w, Map(M,S))
                If (lookupQueryKey is not executable)
                        lookupQuery = RecreateQuery (key(e), w, Map(M,S))
                        If (lookupQuery is not executable)          case D5
                                GenerateMessage(Error,AKC)
                        Exit
                        End If
                End If
                listValuesKey <- AssignValues (lookupQueryKey)
        Else
                listValuesKey <- AssignValues (w.getValue (e))
                ForEach value in listValuesKey
                        lookupQuery = CreateQuery (key(ec), value, Map(M,S))
                        listValues = {}
                        If (lookupQuery is executable)              case D4-1
                                ForEach valueFK in lookupQuery
                                {Values} = FindData (c, valueFK, Map (M, L))
                                ForEach valueColumn in Values
                                        listValues <- AssignKeyPair (valueColumn, {key(e), value})
                                End  ForEach
                        Else
                                lookupQuery = RecreateQuery (key(ec), value, Map(M,S))
                                If (lookupQuery is executable)      case D4-2
                                        ForEach valueFK in lookupQuery
                                        {Values} = FindData (c, valueFK, Map (M, L))
                                        ForEach valueColumn in Values
                                                listValues <- AssignKeyPair (valueColumn, {key(e),
value})
                                        End  ForEach
                                Else                                case D5
                                        GenerateMessage(Error,AKC)
                                        Exit
                                End If
                        End If
        End If
        Return {data} = listValues

End If
```

Algorithm 5 FindData for deletions of tuples from an entity

FindData considers four main situations:

- The clause where ***w*** contains a value for column ***c*** (**case D1**): FindData returns as data the value v of the pair (***a, v***) in ***w.***
- The clause where ***w*** does not contain a value for column ***c*** (**cases D2, D3 and D4**). There are several situations for this:
    - ***w*** contains a value for the key of the entity ***ec, a (*****case D2***).* In this case, a query *lookupquery* is prepared by function *CreateQuery* to execute against the database data to obtain the required value for column *c*. This *lookupquery* requires that the schema contains a table where the value can be retrieved using the proposition φ (key value of ***ec*****).** There are two ways of achieving this:
        - If the *lookupquery* can be executed directly over a *lookupTable* for which the proposition φ holds, the value is retrieved and returned as data (case D2-1). *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 the logical consistency cannot be ensured because of the impossibility of executing the deletion of data.

- Data for the column c can be retrieved from the database but *CreateQuery* is not able to fully prepare *lookupQuery* (case D4-2) as there is no existing table where proposition φ holds. The function *RecreateQuery*: (1) searches *Q* looking for a table, named *sourceTable*, that stores data for c_i (column with unknown data) and the key value of *ec*, (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*.

- ***w*** does not contain a value for the key of entity ***ec*** but attribute ***a*** belongs to the entity whose instance is to be deleted (**case D3**). In this case a query *lookupQueryKey* is prepared to execute against the data in order to retrieve the key values associated to rows that store the same values that are contained in ***w.***
  - If the *lookupQueryKey* can be executed directly over a *lookupTable* for which the proposition φ holds, the key values are retrieved (case D3-1). Then, for each key value retrieved**, the same process as for** cases 4 and 5 is applied, returning *FindData* a list of pairs, where the first value is the value to associate with ***c*** in a deletion statement and the second value is a pair that associates the key with the according value.
  - Values for the key of ***ec*** can be retrieved from the database but *CreateQuery* is not able to prepare *lookupQuery* (case DE3-2). The function *RecreateQuery*: (1) searches *Q* looking for a table, named *sourceTable*, that stores data for k_i (key attribute with unknown data), (2) generates a new table, named *remadeTable*, from *sourceTable*, with suitable keys so that the proposition φ holds, and (3) continues the process described in case 7, executing the *lookupQuery* over the *remadeTable*.
- Attribute ***a*** belongs to an entity ***ec*** that is different from entity ***e*** (**cases D4-1 and D4-2**). In this scenario, the objective is to obtain the instances of *ec* that are associated to the instances of e that fulfil the deletion criteria specified in w. These values are obtained through the *lookupQuery*. After obtaining the primary key values of the instances of *ec* associated to e, the function *FindData* is executed recursively to obtain the necessary values required to be associated to c.

- If no value could be obtained, then the algorithm returns an AKC error (case D5)

In order to further explain this algorithm, let consider the following examples:

Example 3:

Consider the deletion of the instances of the Entity **user** that accomplish the deletion criteria **'username' = 'juan'**:

DMR-D determines the mapping between conceptual model and schemas (that can be seen in Table 2) and the target table *playlist_by_user*. The table contains columns mapped to attributes from user, therefore it is detected as a target table.

Then, DMP-D calls *FindData* to build the appropriate DELETE statement. As the partition key of the table is mapped to the attribute 'username' from 'User', FindsData returns the value 'juan' obtained from the deletion criteria (case D1).

Finally, DMP-D generates the statement that deletes the row:

```
DELETE FROM playlist_by_user WHERE username = 'juan';
```

Example 4:

Consider the deletion of the instances of the Entity **playlist** that accomplish the deletion criteria **'name = 'pl1'**:

DMR-D determines the mapping between conceptual model and schemas (that can be seen in Table 2) and the following target tables where columns mapped to attributes from playlist are detected: *playlist_by_name* and *playlist_by_user*:

Example 4.1: Table playlist_by_name

DMP-D calls *FindData* to build the appropriate DELETE statement. As the partition key of the table is mapped to the attribute 'name from 'playlist', FindsData returns the value 'pl1' obtained from the deletion criteria (case D1).

Finally, DMP-D generates the statement that deletes the row:

```
DELETE FROM playlist_by_name WHERE playlist_name = 'p1';
```

Example 4.2: Table playlist_by_user:

DMP-D calls *FindData* to build the appropriate DELETE statement. In this situation, *FindData* is not able to obtain the required values from the deletion criteria for the columns user_usersame and playlist_id.

To obtain the correct values, *CreateQuery* generates a *lookupQuery* to retrieve data for the column *user_usersame* (case D4), which is mapped to the key attribute of the entity 'user'. As the deletion criteria does not contain a value for the key of Entity playlist in order to obtain the instances of 'user' related to the playlists that fulfil the deletion criteria, first the key values of these instances must be retrieved. For this, *CreateQuery* generates a *lookupQuery* to obtain the values using as the selection criteria the name of the playlist from the table tracks_by_playlist. *LookupQuery* is "SELECT playlist_id from tracks_by_playlist where playlist_name="pl21".

After obtaining the key values of playlist, a lookupQuery is created to obtain the instances of user associated to these key values of playlist. However, there is no lookupTable where the lookupQuery "SELECT user_username from lookupTable where playlist_id="pl21", although these columns exist in the table playlist_by_user. *RecreateQuery* creates and populates a new table, *rm*_playlist_by_user, which can retrieve the unknown values.

The retrieved data for both playlist_id and user_username will replace the placeholders $\$_i$ in rows:

```
DELETE FROM playlist_by_user WHERE user_username = $2 AND playlist_id = $1
```

Note that the value that replaces $2 has a functional dependency with the value that replaces $1, as the values for column “user_username” where obtained searching for a specific value of “playlist_id”.

III.6.2 Deletion of a tuple from a relationship

Next, we will address deleting relationships between entities. In definitions of specific DMR-D and DMP-D for relationships, we will consider different cardinalities of binary relationships (1:1, 1:n and n:m) and illustrate them with an example. The determination of the target tables is the same as for the insertions of tuples in relationships.

Definition 11 Data Manipulation Rule for deleting a relationship between instances of two entities, DMR-DR).- Definition 9 is applied where item *del* is a relationship between entities e_1 and e_2, $R_{\{e1,e2\}} \in Rels(\mathcal{M})$. DMR-DR *determines the set of target tables* $TT_R \subseteq$ Tables(S)*, 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 $Map(\mathcal{M},S)$ 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 $Map(\mathcal{M},S)$ exists between pk_{e2} (key attribute of detail entity) and *k*. Additionally, it must also accomplish that there is at least one attribute of *e1* mapped to a column of tt_R.

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

Definition 12 (Data Manipulation Procedure for deleting instances of a binary relationship, DMP-DR).- Definition 10 is applied where item *del* is a relationship between entities e_1 and e_2, $R_{\{e1,e2\}} \in Rels(\mathcal{M})$. DMP-DR sets, for each column *c* of each target table $tt \in TT$, data taken from deletion criteria or retrieved from a table $lookupTable \in Tabs(S)$.

The general algorithm DMP-D (Section III.6) is now applied with the specialized FindData for deleting an instance of a relationship

```
Function FindData
Input: a column c, a target table tt, a deletion criteria W that contains values v associated to attributes from entities e1 and e2, the conceptual-logical data model mapping between M and S Map(M,S)
Output: data: value for c (cases, DR1, DR2) or set of values (DR3, DR4)
If W has value v for attribute a corresponding to c                    case DR1
  Return v
Else
a = correspondingAttribute (c, Map (M,L))
ec = entity (a)
if w has values v for attributes that belong to ec
        If w has value v for attributes key of ec                      case DR2
                lookupQuery = CreateQuery (c, w, Map(M,S))
                If (lookupQuery is executable)                         case DR2.1
                        GenerateMessage(Information,ADC-S)
                        Return data=lookupQuery
        Else
                 lookupQuery = RecreateQuery (c, w, Map(M,S))
                 If (lookupQuery is executable)                        case DR2.2
                        GenerateMessage(Information,ADC-C)
                        Return data=lookupQuery
                Else                                                   case DR5
                        GenerateMessage(Error,AKC)
                        Exit
                End If
        End If
```

```
Else if w has not value v for attribute key of ec, a is key of ec and w contains values for any attribute of ec
case DR3
        lookupQueryKey = CreateQuery (key(ec), w, Map(M,S))
        If (lookupQueryKey is not executable)
                lookupQueryKey = RecreateQuery (key(e), w, Map(M,S))
        End If
        If (lookupQuery is not executable)                                  case DR5
                GenerateMessage(Error,AKC)
                Exit
        End If

        listValues = {}
        ForEach value in lookupQueryKey
                lookupQuery = CreateQuery (c, value, Map(M,S))
                If (lookupQuery is executable)                              case DR3.1
                        GenerateMessage(Information,ADC-S)
                        listValues <- AssignKeyPair (lookupQuery, {key(e), value})
                Else
                        lookupQuery = RecreateQuery (c, w, Map(M,S))
                        If (lookupQuery is executable)                      case DR3.2
                                GenerateMessage(Information,ADC-C)
                                listValues <- AssignKeyPair (lookupQuery, {key(e), value})
                        Else
                                GenerateMessage(Error,AKC)                  case DR5
                                Exit
                        EndIf
                End If
        End ForEach
        Return {data} = listValues
Else if w does not have values for ec but it does for attributes of the other entity of the relationship
                                                                            case DR4
        e2 = otherEntity (r, ec)
        lookupQueryKey = CreateQuery (key(ec), w, Map(M,S))
         If (lookupQueryKey is not executable)
                lookupQueryKey = RecreateQuery (key(ec), w, Map(M,S))       case DR4.2
        End If
         If (lookupQueryKey is not executable)                              case DR5
                GenerateMessage(Error,AKC)
                Exit

        Else                                                                case DR4.1
                listValues = {}
                ForEach value in lookupQueryKey
                        Values <- FindData (c, value, Map (M, L))
                        ForEach valueFK in values
                                listValues <- AssignKeyPair (valueFK, {key(e), value})
                End ForEach
                Return {data} = listValues
        End If

Else                                                                        case DR5
        GenerateMessage(Error,AKC)
        Exit
    End If
  End If
 Return {data} = listValues

End If
```

Algorithm 6 FindData for deletions of tuples from a relationship

Similar to previous functions, *FindData* will build a query named *LookupQuery (see Definition 7),* 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).

This function *FindData* contemplates cases DR1, DR2, DR3 and DR5 as the same as deleting an instance of an entity (D1, D2, D3 and D5). Moreover, it considers one more situation, DR4, when a column *c* is mapped to an attribute *a* that belongs to an attribute of one of the related entities, e1, but *W* only contains values associated to attributes of the other entity of the relationship, e2. In this case, all instances between *e2* that meet the deletion criteria and their related instances of *e1* must be deleted. This situation is divided in two subscenarios:

- Scenario D4.1: Function *CreateQuery* searches for the key values of instances of entity *ec* that are related to *e2 (1).* After getting these key values, each of these values is used iteratively to invoke recursively *FindData,* where *W* is replaced with the key value (2). Inside the invoked *FindData,* scenarios *DR1* or *DR2* are executed if a value exists or *DR5* if there is no value. If a value is returned by *FindData,* the pair of this value and the key value used for invoking *FindData* is added to the list *listValues* (3). After all these pairs are obtained, *FindData* returns *listValues* (4).
- Scenario D4.2: Function *CreateQuery* searches for the key values of instances of entity *ec* that are related to *e,* but it cannot be executed. The function *RecreateQuery* (1) searches *Q* looking for a table, named *sourceTable*, that stores data for *c* (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.* After this point, it continues from step 2 of Scenario D4.1.

Example 5:

Consider the deletion of the instances of the relationship **features** that accomplish the deletion criteria **'playlist.id = '1' and 'track.id = 1'**:

DMR-D determines the mapping between conceptual model and schema (that can be seen in Table 2) and the target table tracks_in_playlist. The table contains key columns mapped to attributes from both playlist and track, therefore it is detected as a target table.

DMP-D calls *FindData* to build the appropriate DELETE statement. In this situation, *FindData* is not able to obtain the required values from the deletion criteria for the columns playlist_name and track_title.

To obtain the correct values *CreateQuery* generates a *lookupQuery* to retrieve data for the column playlist_name (case DR2). A *lookupQuery* is created to obtain the value of playlist_name associated to the key value of playlist.id '1'. However, there is no lookupTable where the lookupQuery *"SELECT playlist_name FROM lookupTable WHERE playlist_id="1*", although these columns exist in the table tracks_in_playlist. *RecreateQuery* creates and populates a new table, *rm*_playlist_by_id, which can retrieve the unknown values.

The same occurs for track_title (case DR2)., needing to create first a lookupQuery to get the track_title associated to the key value of track.id '1', and then a RecreateQuery *rm*_tracks_by_id that can retrieve the unknown values.

The retrieved data for both playlist_name and track_title will replace the placeholders $\$_i$ in rows:

DELETE FROM tracks_in_playlist WHERE playlist_name= $1 AND playlist_id = '1' AND track_title = $2 AND track_id = '1'

Example 6:

Consider the deletion of the instances of the relationship **features** that accomplish the deletion criteria **'track.id = 1'**:

Similar to the previous example, track_title is obtained in the same way.

However, for column playlist_id, all the playlist_id associated to track_id '1' must be obtained. A lookupQuery is created to obtain the value of playlist_id associated to the key value of track.id '1'. However, there is no lookupTable where the lookupQuery "SELECT playlist_id from lookupTable where track_id="1", although these columns exist in the table tracks_in_playlist. *RecreateQuery* creates and populates a new table, *rm*_playlist_by_id, which can retrieve the unknown values.

Then, for each value in playlist_id, the associated playlist_name is searched as in Example 1.

The retrieved data for columns playlist_id, playlist_name and track_title will replace the placeholders $\$_i$ in rows:

```
DELETE FROM tracks_in_playlist WHERE playlist_name= $3 AND playlist_id = $2 AND
track_title = $1 AND track_id = '1'
```

Note that there is a functional dependency between $2 and $3, so each value of $3 must be used in the same statement as the value of $2 used to obtain it.

III.7 MAINTENANCE OF THE LOGICAL CONSISTENCY FOR DATA UPDATES

The inputs for the maintenance of the logical consistency for updating tuples will consist on (1) a tuple that contains the values that need to be updated, (2) a conceptual model, (3) a schema and (4) a selection criterion that will determine which instances of an entity need to be updated. Like for the insertions and deletions, MDICA will generate the list of ordered data manipulation statements to execute against the database. Similar to previous operations, retrievals of data from tables of the database may be required to maintain the logical consistency.

In this section, we define the rules and procedures that will generate the database statements and messages required to maintain the logical consistency for the update of data of a tuple. As updates can only be associated to attributes of entities or relationships, the algorithm will be the same for both cases. Regarding attributes from entities, we will require to obtain the value of the primary key of the entity in order to properly specify what instances of the entity need to be updated. Regarding attributes from relationships, we will require to obtain the value of the primary key of the entities related in order to properly specify what instances of the relationship need to be updated.

The first step is to identify the target tables where updates of data need to be performed. These target tables will be those where there are columns mapped to the attributes whose values are going to be updated. Similar to the insertions error "Absence of target tables to update" (ATT) can be displayed, as well as the warnings "Absence of a key column generated from a key attribute" (TNW-K) and "Column not generated from any attribute" (TNW-C), which are related to the design of the schema.

The second step is to generate the database statements that must be executed against the database to update the rows. We define the Data Manipulation Procedure (DMP-U) to generate them:

Definition 13 (Data Manipulation Procedure for Updates-DMP-U): Given a tuple *tp (U)* with values to update, the conceptual-logical data model mapping *Map(M,S)* between M and S, the criteria W that determines the rows to update, an item I (entity or relationship in M) corresponding to the conceptual structure of *tp (U)*, and the set *TT* of target tables determined by DMR. DMP-U determines:

1) for each column *c* of each target table *tt* $\in$ *TT*, data taken *W* that generate the following depending on the database operation:
    a. UPDATE statement: an appropriate "WHERE" criteria for the table or retrieved from the database
    b. DELETE+INSERT: for the DELETE, an appropriate "WHERE" criteria to delete the row to be updated. For the INSERT, data to be inserted in each column of the table. This data must be the one deleted with the DELETE with the value updated.
2) for each table *tt*, the ordered list of manipulation operations (INSERT, UPDATE or SELECT) to maintain the logical consistency in *tt*,
3) other additional messages, specific of the procedure, where applicable.

The algorithm DMP-U detailed in Algorithm 7, describes this procedure (Definition 13):

```
Algorithm DMP-U
Input: a tuple tp(U) to update, the conceptual-logical data model mapping between M and S Map(M,S), a W
criteria of instances to update, an item I (entity or relationship in M) corresponding to the conceptual structure
of tp (U) and a set TT of target tables
Output: database statements and messages

suitable = Analysis (tp(U), W, Map(M,S))
If (W is not suitable due to absence of values)
  generateMessage(Error, AC)
  Abort
Else If (tp(U) is not suitable due to attribute does not correspond with any column)
        generateMessage(Warning,AWC)
End If

ForEach target table tt ∈ TT
   If (tp(U) ∈ key(tt)) then
          dataWhere = FindDataWhere (tt, W, tp, I, Map(M,S))
                ForEach where ∈ dataWhere
                      dataUpdate = ExtractData (tt, where)
                      GenerateDelete (where, tt)
                      GenerateInsert (dataUpdate, tt)
  Else then
          dataWhere = FindDataWhere (tt,w,  tp (U), Map(M,S))
                ForEach where ∈ dataWhere
                      GenerateUpdate (dataWhere, tt)
                End ForEach
  End If
End ForEach
```

Algorithm 7 Data Manipulation Procedure for Updates (DMP-U)

First, DMP-U analyses the tuple *tp(U)* and *W* (function **Analysis**) to determine their suitability:

- It contains values assigned to at least one attribute of the entity or to the primary keys of the relationship. If this is not the case, DMP-U raises (in the function

GenerateMessage) the error message “Absence of value for criteria” (AC) because there is no value to filter the rows to.

- Each attribute in *tp(U)* 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, DMP-U processes each target table *tt* to determine the database operations required to perform in them to update the data. There are two possible variants depending on if the columns from *tp (U)* to be updated are part of the primary key of *tt* or not:

- Part of the PK: the values cannot be updated directly with an UPDATE statement. Instead, the row to be updated needs first to be deleted and then inserted again with the value updated.
- Non-key: the values are updated with an UPDATE statement.

In both cases a certain WHERE criteria for each *tt* needs to be determined using the function *FindDataWhere*. Note that there could be several rows that need to be updated. In that case, there would be several WHERE criteria returned by *FindDataWhere*. For each WHERE criteria, an UPDATE statement (*GenerateUpdate* function) or a DELETE-INSERT (*GenerateDelete* and *GenerateInsert* functions) statement combination is generated by the appropriate functions.

The following algorithm contains the specification of the function *FindDataWhere*, which returns the data to be used in the criteria of the modification of data statements that are required. Most of the functions used in *FindDataWhere* were already explained in the description of *FindData* for the deletions of data. The ones that were not used before have the following purpose:

AddPair (Column: c, v: Value): Assigns to the column *c* the value *v* to use in the deletion criteria (WHERE) of the DELETE statement.

keyValues (List <Pair {v: Value, pair {k: Attribute, kv: Value}}: Creates a list of unique pair of key attributes *k* of an entity assigned to *kv* values. The parameter *v* is not used in the function.

GenerateAllStatements (pair (k: Attribute, kv: Value) , ListPairValues, tt): Generates a list of all possible criteria for key attribute *k* with the assigned value *kv*.

RemovePair (p: Pair(a: Attribute, av: Value), l: List <Pair {v: Value, pair {k: Attribute, kv: Value}): Removes from the list *l* all the entries that contain for the parameter “pair {k: Attribute, kv: Value}” the same pair as *p*.

Function FindDataWhere
Input: a target table *tt*, an item i (entity or relationship) to be updated, a criteria W that contains the criteria that a row must meet to be updated, the conceptual-logical data model mapping between M and S *Map(M,S)*
Output: a list *whereValues* that contains several sets of associations column-value for the key columns

```
a = correspondingAttribute (c, Map (M,L))
keyEc = key (i)
attributeValueKey = {}
ForEach c ∈ key (tt)
If W has value v for attribute a corresponding to c                    case U1
  attributeValueKey <- AssignKeyPair ("all", {c, v})
Else

If w has value v for attributes key of ec                              case U2
      lookupQuery = CreateQuery (c, key(i), w, Map(M,S))
      If (lookupQuery is executable)                                   case U2-1
          GenerateMessage(Information,ADC-S)
          attributeValueKey <- AssignKeyPair (lookupQuery, {key(ec), v })
      Else
      lookupQuery = RecreateQuery (c, key(i), w, Map(M,S))
      If (lookupQuery is executable)                                   case U2-2
          GenerateMessage(Information,ADC-C)
          attributeValueKey <- AssignKeyPair (lookupQuery, {key(ec), v })
     Else                                                              case U5
          GenerateMessage(Error,AKC)
          Exit
     End If
Else if w has not value v for attribute key of i and a ∈ i             case U3
      lookupQueryKey = CreateQuery (key(i), w, Map(M,S))
      If (lookupQueryKey is not executable)                            case U3-2
          lookupQueryKey = RecreateQuery (c, key(i), w, Map(M,S))
          If (lookupQueryKey is not executable)                        case U3-1
                GenerateMessage(Error,AKC)
                Exit
          End If
     End If
     ForEach value in lookupQueryKey
          lookupQuery = CreateQuery (c, {lookupQueryKey}, w, Map(M,S))
          If (lookupQuery is executable)                               case U3-3
                GenerateMessage(Information,ADC-S)
                attributeValueKey <- AssignKeyPair (lookupQuery, {key(i), value})

          Else
          lookupQuery = RecreateQuery (c, {lookupQueryKey}, w, Map(M,S))
          If (lookupQuery is executable)                               case U3-4
                GenerateMessage(Information,ADC-C)
                attributeValueKey <- AssignKeyPair (lookupQuery, {key(i), value})
          Else                                                         case U5
                GenerateMessage(Error,AKC)
                Exit
          End If
      End ForEach
      Return {data} = attributeValueKey

Else                                                                   case U4
          e2 = entity (c)
          lookupQueryKey = CreateQuery (key(i), w, Map(M,S))
          If (lookupQueryKey is not executable)
                lookupQueryKey = RecreateQuery (c, key(i), w, Map(M,S))
                If (lookupQuery is not executable)                     case U4-1
                      GenerateMessage(Error,AKC)
                      Exit
                End If
          End If
          ForEach keyValue ∈ lookupQueryKey
                lookupQueryKeyE2 = CreateQuery (key(i), keyValue, Map(M,S))
                If (lookupQueryKeyE2 is not executable)
```

```
                    lookupQueryKeyE2 = RecreateQuery (c, key(i), w, Map(M,S))
                    If (lookupQueryKeyE2 is not executable)                    case U5
                        GenerateMessage(Error,AKC)
                        Exit
                    End If
                End If
                ForEach keyValueE2 ∈ lookupQueryKeyE2
                    listValues = FindDataWhere (c, keyValueE2, Map (M, L))
                    ForEach value in listValues
                        attributeValueKey <- AssignKeyPair (value, {key(i), keyValue })
                    End ForEach
            End ForEach

            ListPairValues = keyValues (attributeValueKey)
            Statements = {}
            ForEach pair (a, v) ∈ ListPairValues
                Statements <- GenerateAllStatements (pair (a,v) , ListPairValues, tt)
                RemovePair (pair(a,v), ListPairValues)
            End ForEach

            Return Statements

        End If
    End ForEach
```

Algorithm 8 FindDataWhere for updates of data

FindDataWhere considers four main situations:

- The clause where ***w*** contains a value for column ***c*** (case U1): *FindDataWhere* returns as data the value ***v***, indicating that for all UPDATE statements that are going to be executed, v will always be the value assigned for c.
- The clause where ***w*** does not contain a value for column ***c*** (cases U2, U3 and U4). There are several situations for this:
    - ***w*** contains a value ***v*** for the key of the entity ***ec*** of *a (*case U2*).* In this case, a query *lookupquery* is prepared by function *CreateQuery* to execute against the database data to obtain the required value for column *c*. This *lookupquery* requires that the schema contains a table where the value can be retrieved using the proposition φ *(*key value of ***ec).*** There are two ways of achieving this:
        - If the *lookupquery* can be executed directly over a *lookupTable* for which the proposition φ holds, the value is retrieved and returned as a pair of data (case U2-2). This pair indicates that the value to assign to c is always used in the UPDATE statements alongside the value ***v***. *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 the logical consistency cannot be ensured because of the impossibility of executing the deletion of data.
        - Data for the column c can be retrieved from the database but *CreateQuery* is not able to fully prepare *lookupQuery* (case U2-1) as there is no existing table where proposition φ holds. The function *RecreateQuery*: (1) searches *Q* looking for a table, named *sourceTable*, that stores data for c_i (column with unknown data) and the key value of *ec*, (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*. The next steps are the ones from the case U2-1.

- ***w*** does not contain a value for the key of entity ***ec*** but attribute ***a*** belongs to the entity whose instance are to be updated (case U3). In this case a query *lookupQueryKey* is prepared to execute against the data in order to retrieve the key values associated to rows that store the same values that are contained in ***w.***
  - If the *lookupQueryKey* can be executed directly over a *lookupTable* for which the proposition φ holds, the key values are retrieved (case U3-1). Then, for each key value retrieved***,*** the same process as for cases in U-2 are applied, assigning to the returned pair of values the association that each value for a must be in the same statement as the value of the primary key that was used in the *lookupQuery*.
  - Values for the key of ***ec*** can be retrieved from the database but *CreateQuery* is not able to prepare *lookupQuery* (case U3-2). The function *RecreateQuery*: (1) searches *Q* looking for a table, named *sourceTable*, that stores data for k_i (key attribute with unknown data), (2) generates a new table, named *remadeTable*, from *sourceTable*, with suitable keys so that the proposition φ holds, and (3) continues the process described in case 7, executing the *lookupQuery* over the *remadeTable*.
- Attribute ***a*** belongs to an entity ***ec*** that is not the same as entity ***e*** (case U4). In this scenario, the objective is to obtain the instances of *ec* that are associated to the instances of e that fulfil the deletion criteria specified in w. These values are obtained through the *lookupQuery*. After obtaining the primary key values of the instances of *ec* associated to *e*, the function *FindDataWhere* is executed recursively to obtain the necessary values required to be associated to c. These values are always associated to the value of the primary key of e that was used to obtain the values of the primary key of *ec*.

- If no value could be obtained, then the algorithm returns an AKC error (case U5)

After all possible criteria values are obtained, the algorithm finishes with the generation of a list where each element is a list of values assigned to attributes to be used in a single statement in DMP-U.

In order to further explain these algorithms, let consider the following examples:

Example 7:

Consider the update of the **nationality** of an **“Artist”** to **“French”** according to the following criteria **‘artist.name = ‘John’**.

DMR determines the mapping between conceptual model and schema, determining that the update operation must be applied against the target table tracks_by_artist, artists_by_first_letter and tracks_inplayst (see Figure 5). These tables contain the name of the artist, therefore are detected for the update of data. We describe the approach for each table in different subexamples:

Example 7.1: Table tracks_by_artist

DMP-U calls *FindData* to build the appropriate UPDATE statement. In this case the table contains the partition key artist_name, therefore it is able to build the appropriate UPDATE statement following case U1.

UPDATE tracks_by_artist SET nationality = “French” FROM WHERE artist_name= ‘John’

Example 7.2: Table artists_by_first_letter

DMP-U calls *FindData* to build the appropriate UPDATE statement. In this situation, *FindData* is not able to obtain the required values from the deletion criteria for the column artist_first_letter.

To obtain the correct values *CreateQuery* generates a *lookupQuery* to retrieve data for the column artist_first_letter (case U2). A *lookupQuery* is created to obtain the value of artist_first_letter associated to the key value of artist.name ‘John’. However, there is no *lookupTable* where the *lookupQuery* “SELECT artist_first_letter from *lookupTable* where artist_name="1", although these columns exist in the table artist_first_letter. *RecreateQuery* creates and populates a new table, *rm*_ artist_first_letter, that can retrieve the unknown values (case U2-1). These values are used to replace the placeholders $\$_i$ in rows:

UPDATE artist_by_first_letter SET artist_nationality = “French” FROM WHERE artist_first_letter = $\$_i$ AND artist_name= ‘John’

Example 7.3: Table tracks_by_genre

DMP-U calls *FindData* to build the appropriate UPDATE statement. In this situation, *FindData* is not able to obtain the required values from the deletion criteria for the columns track_genre, track_title and track_id.

To obtain the correct values all the tracks associated to the author with the name ‘John’ must be obtained. A *lookupQuery* is created to obtain the value of track_id associated to the key value of artist.name ‘John’. However, there is no *lookupTable* where the *lookupQuery* “SELECT track_id from lookupTable where artist_name="John", although these columns exist in the table tracks_by_genre. *RecreateQuery* creates and populates a new table, *rm*_tracks_by_genre, which can retrieve the unknown values (case U-4).

In the case of the values for the columns track_genre and track_title, a *lookupQuery* is created to obtain the value of them associated to the key values of track.id that were obtained in the previous step. However, there is no *lookupTable* where the *lookupQuery* “SELECT track_id from lookupTable where artist_name="John", although these columns exist in the table tracks_by_genre. *RecreateQuery* creates and populates a new table, *rm*_tracks_by_genre2, that can retrieve the unknown values (case U-4).

The retrieved data for the three columns will replace the placeholders $\$_i$ in rows

UPDATE tracks_by_genre SET artist_nationality = “French” FROM WHERE track_genre = $\$_2$ track_title= $\$_3$ AND track_id= $\$_1$

Note that there is a functional dependency between $2 and $3 with $\$_1$, so each value of $3 and $2 must be used in the same statement as the value of $1 used to obtain it.

III.8 EVALUATION

In this section we evaluate MDICA. In this evaluation we have focused on insertions of data as other typical operations (delete and update) in transactional systems, although considered in MDICA, 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. We have established the following research questions for this evaluation:

- RQ1: Given an insert operation at a conceptual model level, is it always possible to insert data at schema 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 schema in terms of the number of tables affected to maintain the logical consistency?
- RQ3: How many database statements must be executed for each insert operation at the conceptual model level in order to maintain the logical consistency in the database?
- RQ4: Is it always possible to ensure that the logical consistency is maintained? If this is not the case, what are the situations identified that can endanger it?
- RQ5: Do the database statements determined by MDICA maintain the logical consistency?

The research questions RQ1, RQ2, RQ3 and RQ4 are answered in the following subsections. RQ5 is answered in the section IV.4 from chapter IV, where an approach for checking the maintenance of the logical consistency in a column-oriented database is defined.

III.8.1 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) [84] has become the de-facto benchmark, designed by [85] to compare the performance of data stores and used for measuring performance, scalability, elastic speedup, throughput and latency [85]–[87] of different NoSQL databases. Since the schema 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 relationships, and relationships with different cardinality). The selected case studies are:

- Digital Library Portal, used by Chebotko et al. [18] to illustrate the data modelling 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 [13] 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 case study in subsection III.4) is used as a tutorial intended for programmers interested in learning about Cassandra [83] 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 model and the schema. Table 3 displays information about the models:

- Conceptual models: items (entity or relationship with its cardinality 1:n or n:m), their name and the number of key and non-key attributes (columns "#PK" and "#nPK").
- Schemas: 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 relationship, 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.

	Conceptual data model				Logical data model				
Case Study	Item	Name	# PK	# nPK	Table	From Item/s	From Name	# Key	# nKey
DigitalLibrary	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					
HotelReservation	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					
MusicStore	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					

Table 3 Conceptual and logical models used in the evaluation

III.8.2 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 [88]. 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):
  - Relation: insertion in a relationship, which is subdivided into three classes depending on the cardinality: 1:1, 1:n and n:m.

    - 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 relationships which generated a table in the schema.
- 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 4 displays the number of test cases for each of the combinations.

Case Study	Item	Name	Attribute-value Pairs *	PK	-attr	-PK	Total	Target Tables
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 to evaluate MDICA and target tables impacted by each test case

Once we have generated the test cases, we apply the rules and procedures defined in Section III.5 to each one. As previously described, after obtaining the mapping between conceptual model and schema, MDICA identifies the target tables (listed in Table 4, 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 the logical consistency and the messages

shown to users. The analysis of the results of these executions is detailed in the following sections.

III.8.3 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 schema level. We found that 45.0% of the test cases produced insertions into databases, and the remaining 55.0% did not.

Table 5 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 III.5, depending on the reason they did not insert any rows.

	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**

Table 5 Test cases that produced insertions in databases and test cases that generated error messages

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.

III.8.4 Analysis of target tables impacted by an insertion (RQ2)

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

All tables in the schemas (listed in Table 3) are impacted by some test case as Table 4 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 relationships, 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 relationships (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 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.

III.8.5 Analysis of database statements (RQ3)

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

Table 6 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.

Case Study	Item	Name	#Target Tables	#Test Cases	INSERT #	INSERT %	INSERT Avg	SELECT #	SELECT %	SELECT Avg	CREATE& COPY #	CREATE& COPY %	CREATE& COPY Avg	Total #	Total 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 Reservation	**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 Reservation				**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**

Table 6 Database statements generated by DMPs

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 relationships, SELECT (average 0.9 per test case) and CREATE&COPY (average 0.2) statements were produced 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 the logical consistency 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 the logical consistency.

III.8.6 Analysis of messages (RQ4)

To answer RQ4, we analyse 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 7 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.

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 Reservation	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**

Table 7 Information and Warning Messages generated by MDICA

Information messages (195 in total, average 1.5 messages per test case), 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), because they give additional information about the insert operation that may endanger the logical consistency, although it can be conducted.

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 schema to store values of the tuple which could cause a potential loss of information. The developer should analyse the schema 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 and information messages, warnings provide particularly valuable knowledge for the maintenance of the logical consistency. MDICA identifies situations that can endanger it and generates the right messages. Using these messages, where appropriate, developers will be able to fix faults to ensure the logical consistency by including more values in the tuples, creating new tables, adding new columns, or making other changes in the databases.

III.8.7 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 the logical consistency and produced messages informing of those situations that could endanger the consistency. 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 the logical consistency in databases. We inspected them carefully and found that they maintain the logical consistency 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 the logical consistency.

Threats to construct validity.- In two of the three case studies, the schemas were designed following the modelling 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 modelling 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 the logical consistency faults although we do not have feedback from professional users. To mitigate this threat, messages should be validated by developers with diverse levels of experience.

IV MAINTAINANCE OF THE LOGICAL CONSISTENCY: CONCODA

In this chapter we detail how to check if a column-oriented database maintains the logical consistency after a set of database statements to perform a modification of data haven been executed against the database. This is done through an approach that we named CONCODA, which was first presented in [27]. We also used CONCODA in two experimentations, one published in [27] where multiple initial states of the database were checked and another one published in [25] where all possible combinations of possible tuples for a case study were inserted in a database with the same initial state. Section IV.1 introduces the problem of determining if the database maintain the logical consistency after a modification of data. Section IV.2 contains the description of the case study used in the rest of the section, both for explaining the proposition and for the experimentation. Section IV.3 contains the detail of the process of CONCODA. Section IV.4 details the experimentation performed using CONCODA.

IV.1 INTRODUCTION

In the previous chapter of the thesis, we proposed a preventive approach named MDICA that provided the database statements required to perform a modification of data (insertion, deletion or update of a tuple) in a column-oriented database. MDICA is intended to help developers to avoid the commitment of mistakes during the implementation of database statements and other related code procedures required to perform a modification of data. However, for existing applications that already have database statements and code procedures implemented, MDICA is not as helpful, being more helpful a reactive approach.

In order to address the problems related to the maintenance of the logical consistency in existing projects, we propose a reactive approach named **CONCODA**. CONCODA aims to determine if the database statements that are executed to perform a modification of data incur in the creation of inconsistencies by checking if the column-oriented database maintains the logical consistency after the modification is performed. If the consistency is not maintained, it identifies the tables where there are inconsistencies, allowing the identification of the database statements that incurred in the creation of inconsistencies. This will be helpful for developers so they can detect defects in their applications that are producing inconsistencies in the data during the execution of the application.

In the next sections of this chapter, we describe in section IV.2 the case study used during the description of CONCODA and in the experimentation. Section IV.3 contains the description of CONCODA, specifying and detailing its phases. Section IV.4 contains the experimentation using CONCODA.

IV.2 CASE STUDY

In this section we detail and explain the case study that will be used both during the explanation of CONCODA through examples and for the experimentation. The case study is named “Data Library Portal” [18] which is composed of a conceptual model and a schema designed after this conceptual model. The conceptual model is illustrated in Figure **6**, containing 4 entities and 5 relationships. Its column-oriented database schema is composed of 9 tables, and it is illustrated

in Figure 7. Partition keys are labelled as **K**, clustering keys are labelled as **C** and Counter columns are labelled as **++**.

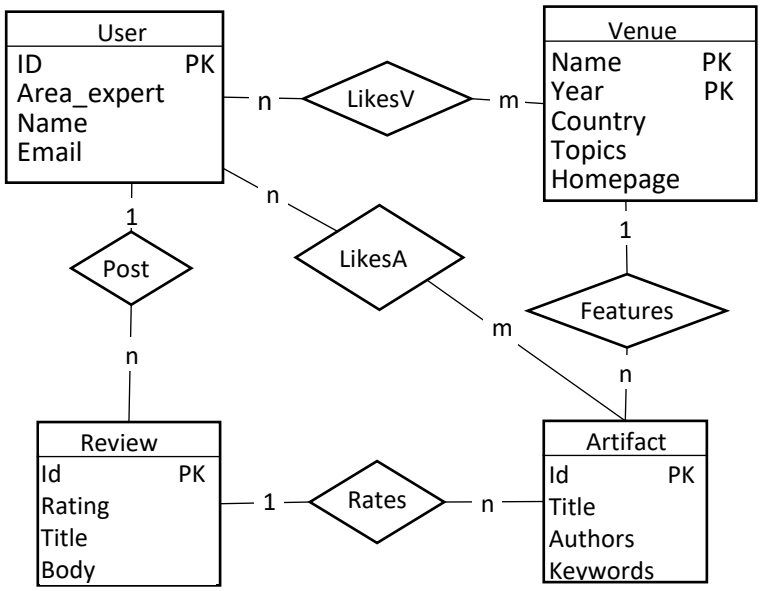


*Figure **6** Conceptual model of Data Library Portal case study*

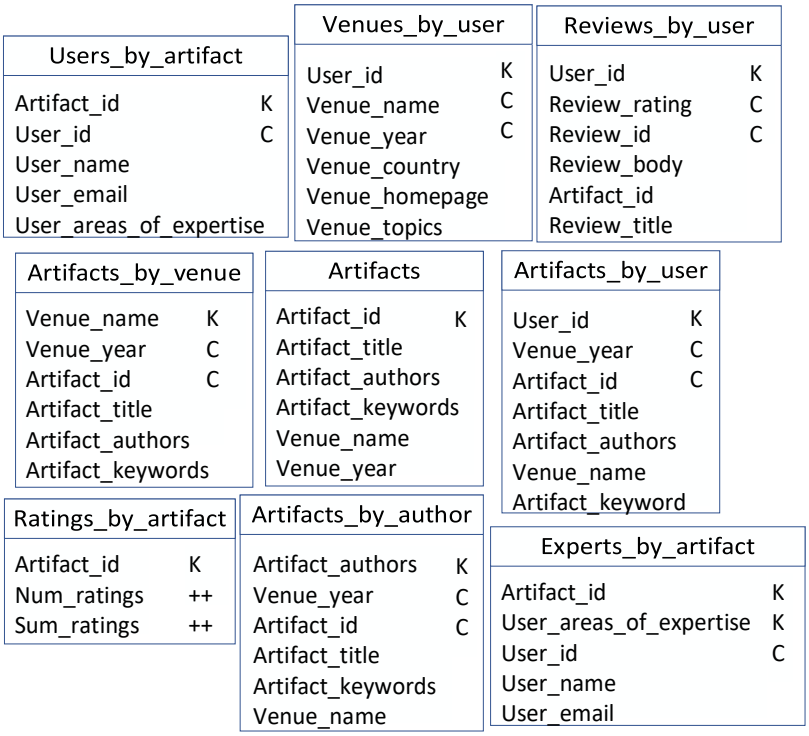


Figure 7 Schema of Data Library Portal case study

IV.3 VERIFICATION OF THE LOGICAL CONSISTENCY IN A COLUMN-ORIENTED DATABASE

In this section we detail CONCODA, a reactive approach to check if a column-oriented database maintains the logical consistency after one or more modifications of data have been performed against the database. This will help to detect defects in the database statements that are executed to perform the modification of data.

In order to check if the database maintains the logical consistency, we will use a relational database that ensures logical consistency, which serves as an **oracle**. This relational database will implement in its schema a normalized model of the column-oriented schema, which is obtained from the conceptual model. The entities of the conceptual model are implemented in this schema as table and the relationships through the appropriate foreign keys or additional tables.

The basis of using a relational database as an oracle is that relational databases ensure the logical consistency through model normalization and the definition of integrity constraints. Therefore, when the relational database and the column-oriented database initially store equivalent data, if the same modification of data is performed in both databases, they should still store equivalent data after the modification is performed. In order to perform in the relational database the same modification of data, CONCODA will determine the SQL database statements required for it.

In the rest of this section, we will refer to the relational database as **OracleDB** and to the column-oriented database as **TargetDB**.

CONCODA consists of two phases:

1. **Database statements determination and execution**: The database statements required to perform a modification of data in both **TargetDB** and **OracleDB** are first determined and then executed.
2. **Logical consistency check**: CONCODA created and execute the queries to check if the logical consistency is maintained in the column-oriented database.

In the following subsection we detail each of these phases:

IV.3.1 Database statements determination and execution.

In this phase of CONCODA, the modifications of data are performed against both TargetDB and OracleDB. In Algorithm 9 the process of performing the modification of data in each database is detailed.

Inputs: TargetDB ***tdb***, OracleDB ***odb,*** modification of data ***md***, conceptual model ***cm***	
TargetDB	OracleDB
applicationToTest (md, db)	SQLStatements <- convertSQL (md, cm) For each statement in SQLStatement: ExecuteSQL (odb, md)

Algorithm 9 Determination and execution of database statements for a modification of data

Both TargetDB and OracleDB must start with equivalent data. Otherwise, CONCODA would not be able to determine if the logical consistency is maintained at the end of the process. In TargetDB the modification is performed through the application whose database statements are to be tested (*applicationToTest*). On the other hand, in OracleDB, CONCODA must first determine the SQL statements required to perform the modification of data (*convertSQL*). Note that a modification of data is an insertion, update, or deletion of a tuple. As the model of the OracleDB is normalized, CONCODA determines a single SQL statement for each entity and many-to-many relationship that participates in the modification of data. For instance, in an insertion of a tuple that contains data of an entity, one SQL INSERT statement is created to insert the data in the table that stores the data of the entity. After the SQL statements are determined, they are executed against OracleDB (*ExecuteSQL*).

Let consider an example to illustrate this phase by inserting a relationship between the entities *User* and *Artifact*.

The tuple that contains the data that relates one *User,* and one *Artifact* is inserted in both the TargetDB and the OracleDB. To simplify the example, the tuple contain data for all attributes of the User and the Artifact. This means that in OracleDB, three SQL statements will be created and executed to insert the required data into the tables that store data of the entities (tables "User" and "Artifact") and the relationship (table "User_Artifact"). Likewise, in the TargetDB the required data will be inserted in "Users_by_artifact" and "Experts_by_artifact" through database statements determined by the application to verify. This insertion is illustrated in Figure 8.

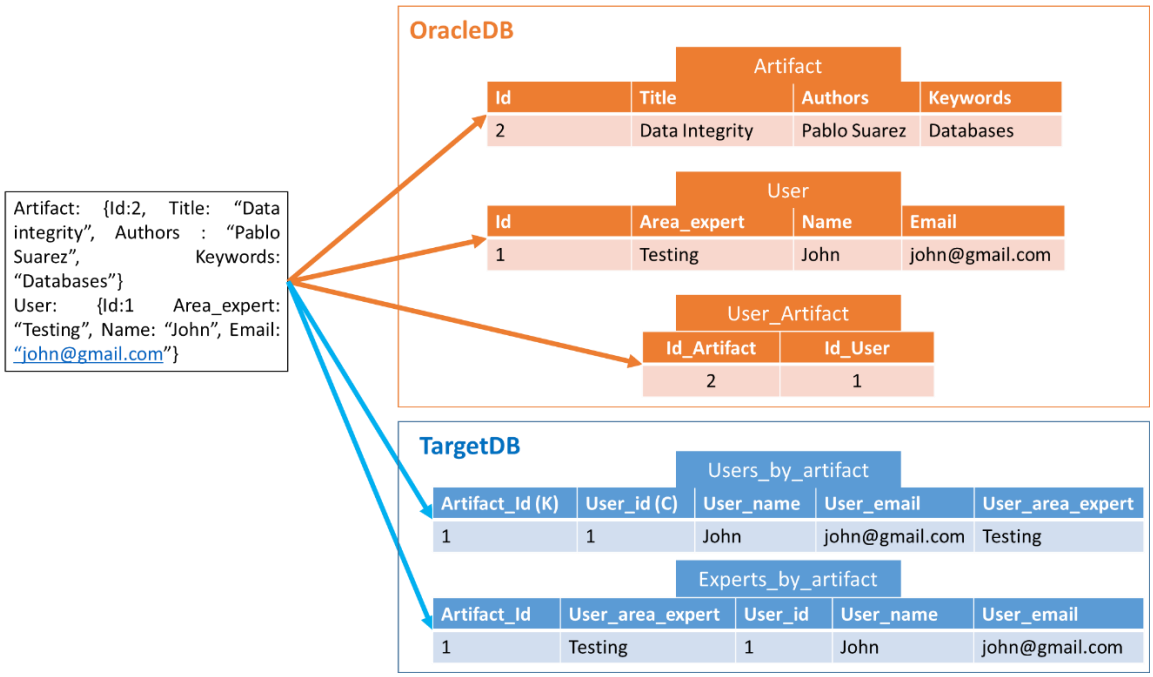


Figure 8 Insertion of tuple in the relational database (OracleDB) and the column-oriented database (TargetDB)

IV.3.2 Check if the database maintains the logical consistency

In this phase of CONCODA, the databases are compared to check if the logical consistency was maintained in TargetDB. Algorithm 10 describes the process of checking this consistency is performed.

Inputs: TargetDB ***tdb***, OracleDB ***odb,*** schema of TargetDB***, sdb,*** conceptual model ***cm*** Output: InconsistenciesDetected
InconsistenciesDetected <- [] For each table in sdb: dataTDB <-extractAllData (table, tdb) dataODB <-extractEquivalentData (table, odb, cm) For each rowTDB in dataTDB and rowODB in dataODB: If (rowTDB != rowODB): InconsistenciesDetected<-Inconsistency (table, rowTDB) Return InconsistenciesDetected

Algorithm 10 Determination and execution of database statements for a modification of data

CONCODA will create and execute for each table of TargetDB one query against both databases:

- **TargetDB**: a query (*extractAllData*) that retrieves all the data stored in the table (dataTDB)

- **OracleDB:** an equivalent SQL query (*extractEquivalentData*) to the table from the TargetDB to extract the same data (dataODB)

Then, it will compare the results obtained of both queries, row by row. If the results are not the same (*rowTDB* and *rowODB*), the inconsistency is noted (*InconsistenciesDetected*). After all tables are checked, the developer is given the information of the inconsistencies detected. If no inconsistency was detected, it means that the logical consistency was maintained when performing the modification of data. Let use an example to illustrate this phase by continuing the example from subsection IV.3.1, where, to simplify the example, only the tables "Users_by_artifact" and "Experts_by_artifact" are analysed. In this example we first detail the creation of the queries and then the comparison of their execution result.

1. **Creation of queries for both the target DB and the oracle**. For TargetDB, CONCODA creates the queries that retrieve all the data from these two tables. Likewise, CONCODA creates the equivalent SQL queries are the same data stored in those tables from OracleDB, requiring two INNER JOIN operators in both queries to relate the data between Users and Artifacts. This is illustrated in Figure 9.

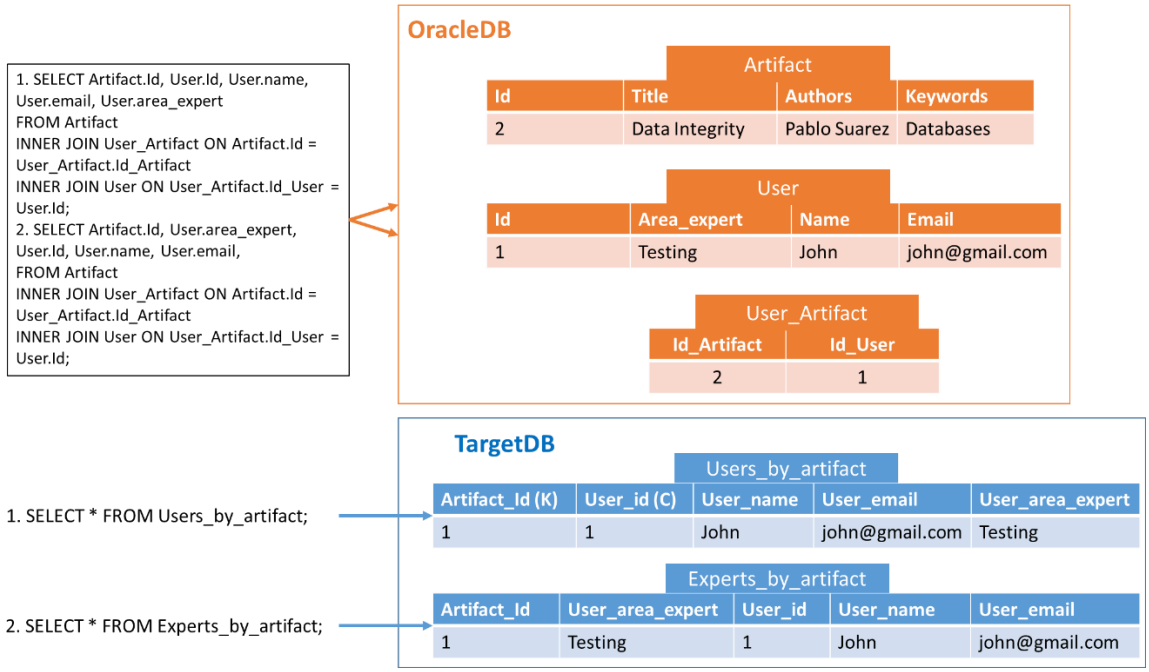


Figure 9 Comparison of queries against a Relational DB (OracleDB) and the Column-oriented DBs (TargetDB)

2. **Queries execution and result comparison**. The data stored in each column-oriented table is compared with the data returned by its equivalent query in the relational database. If they return the same data, the logical consistency is maintained. Otherwise, the data operation in step 1 inquires the creation of inconsistencies in that particular table. This is illustrated in Figure 10.

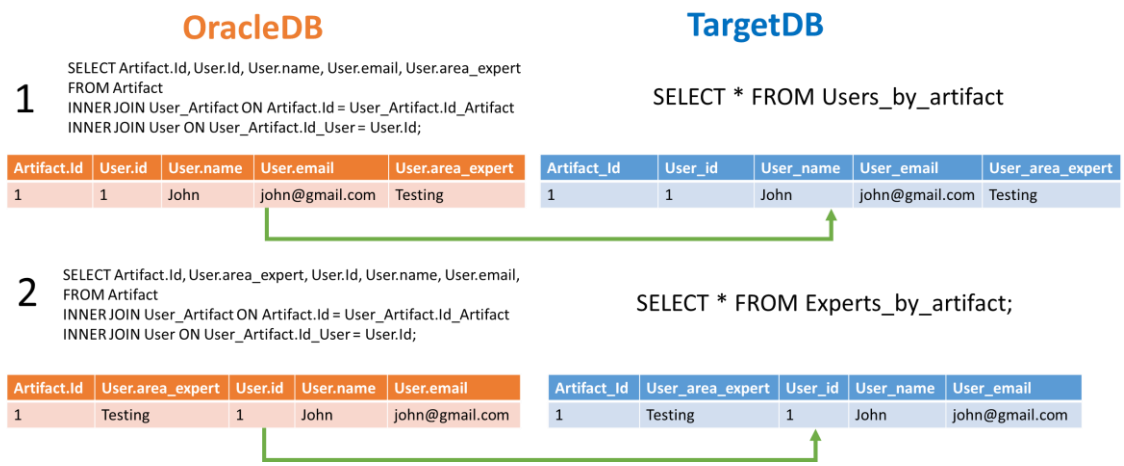


Figure 10 Check of the logical consistency maintenance

IV.4 EXPERIMENTATION

In this section we detail the experimentation performed using CONCODA to validate MDICA, which answers the research question RQ5 “Do the database statements determined by MDICA maintain the logical consistency?” that was established in section III.8 from chapter III of this thesis.

We detail two experimentations that use the case study from Section IV.2: one in Section IV.4.1 that was first published in [27] where a set of tuples are inserted in a database with different start data and another one in Section IV.4.2 that was first published in [25] where all the possible tuples of the case study are inserted in a database that starts empty.

IV.4.1 First experimentation: different database initial state

In this subsection we describe the first experimentation that we performed to verify if MDICA maintains the logical consistency, which was first published on [27]. We designed several test cases in which in each one we perform a modification of data through MDICA that **inserts a tuple** in a database whose schema is detailed in section IV.2. In each test case we consider both the tuple to insert and the initial state of the database. Each **tuple** is a collection of attribute-value pairs from entities or relationships which were obtained through a systematic combination detailed in the next subsubsection.

IV.4.1.1 Test case design

The tuples to be inserted have been systematically obtained by combining all possible attribute-value pairs of the entities or relationships (attributes of the two related entities or more if there are several relationships) that can be determined of the following tuple types:

1. Complete: One tuple that that contains an attribute-value pair for each attribute of the entity or related entities.
2. Partial: Several tuples (as much as non-key attributes of the entity or related entities) in which in each one of them there is an attribute without a value assigned to it.
3. Incomplete: One tuple with only values assigned to the key values of the entity or related entities.

4. IncompleteRel: Only when the tuple contains relationships: several tuples (as much as entities related) in which, in each one of them, there is an entity from the relationships that only has attribute-value pairs for its key attributes.

In the case of the relationships, if any of the related entities is a secondary entity (cardinality "many" in a "one to many" relationship), the relationships between this secondary entity and the primary entity (cardinality "many" in a "one to many" relationship) are included as well by assigning values to the keys of the primary entity.

For example, in the case of the relationship between Venue and User we used the following tuples:

Complete Type tuple:

{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1, Area_expert: "Testing", Name: "John", Email: "john@gmail.com"}}

Partial Type tuples:

{Venue: {Name: "JISBD", Year: 2022, Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1 Area_expert: "Testing", Name: "John", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1 Area_expert: "Testing", Name: "John", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases"}
User: {Id:1 Area_expert: "Testing", Name: "John", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1, Name: "John", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1, Area_expert: "Testing", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1 Area_expert: "Testing", Name: "John"}}

Incomplete Type tuple:

{Venue: {Name: "JISBD", Year: 2022}
User: {Id:1}}

IncompleteRel Type tuples:

{Venue: {Name: "JISBD", Year: 2022}
User: {Id:1, Area_expert: "Testing", Name: "John", Email: "john@gmail.com"}}
{Venue: {Name: "JISBD", Year: 2022, Country: "Spain", Topics: "Software Engineering, Testing, Databases", Homepage: www.sistedes.es}
User: {Id:1}}

Each of the determined tuples is inserted by two test cases, each one with a different initial state of the database:

1. The DB starts empty.
2. The DB starts with data that will be queried in order to maintain the logical consistency (SELECT). These data have been determined by us by analysing the SELECT statements determined by MDICA in the test case 1 where the DB starts empty.

When the tuple is of the type "Complete" the only test case executed is the first one, as no SELECT statements were determined by MDICA in the test cases where the DB started empty.

IV.4.1.2 Test cases results

Table 8 displays the test cases that were designed and executed along with a summary of the execution result. The information of what is displayed in each column is the following:

- Entity/Relationship: indicates the entities or relationships whose attributes contain associated values in the tuple.
- Test cases: total of test cases where a tuple with information of that entity or relationships were inserted.
- Success: total of successful test cases.
- Blocked: total of blocked test cases. These blocked test cases are those where MDICA detected that it is impossible to insert the tuple in a required table due to not having values to insert in key columns.
- INSERT: Number of INSERT statements required in the test cases.
- SELECT: Number of SELECT statements required in the test cases.

For each of these columns there is a division depending on the state of the database for the test case executed (Empty or with Data).

Entity/Relationship	Test cases		Success		Blocked		INSERT		SELECT	
	Empty	Data	Empty	Data	Empty	Data	Empty	Data	Empty	Data
Review	5	4	5	4	0	0	0	0	0	0
User	5	4	5	4	0	0	0	0	0	0
Venue	5	4	5	4	0	0	0	0	0	0
Artifact	5	4	5	4	0	0	5	4	6	6
Features	10	9	7	9	3	0	21	27	6	27
Posts-Rates	14	13	9	13	5	0	36	37	9	9
LikesV	10	9	10	9	0	0	10	9	6	5
LikesA-Features	14	13	9	13	5	0	63	63	26	45
Total	68	60	57	60	11	0	135	140	53	92

Table 8 Summary of first experimentation

In total, we executed 128 test cases which all were successful for our objective of validating MDICA. In 117 of them, the test case passed, verifying that the logical consistency was maintained by MDICA. In the remaining 11 test cases, the test result was blocked, as MDICA issued an AKC error as an absence of values for a key column was detected. All these blocked test cases occurred when the database started empty. This was because during the process, the tuple did not contain values for a key column and when MDICA executed SELECT statements to obtain those missing values, the return was empty.

Although these blocked test cases cannot be considered as passes test cases as the tuple was not inserted, they were still considered successful results for validating MDICA, as the logical consistency was preserved, and it was the expected outcome from MDICA.

IV.4.2 Second experimentation: exhaustive insertion of tuples

In this subsection we detail a more exhaustive experimentation than the one detailed in the previous section regarding the tuples to insert, although only considering the empty initial state of the database. In this second experimentation, we executed several test cases where all possible types of tuples of entities and relationships that could be obtained in the case study were inserted. In the following subsubsections, we first describe how the test cases were designed and the classification of the tuples to insert. After this, we describe the results of the test cases, splitting the explanations of the results of the test for tuples of entities and tuples of relationship.

IV.4.2.1 Test cases design

In order to test all possible combination of tuples for each entity and relationship, we define the following classification to indicate how many attributes of an entity have an assigned value in the tuple to insert in the test case:

- Complete (C): every attribute of the entity has an assigned value.
- Partial (P1 or P2): the primary key and two (P2) or one (P1) non-key attributes have assigned values.
- Incomplete (I): only the primary key has an assigned value.

Note that this classification is different from the one in the subsection IV.4.1.1. The tuple type Partial has been divided in P1 and P2 in order to be more specific when displaying the results. The tuple type IncompleteRel is not defined as these tuples are already considered when a tuple of a relationship contains values for each non-key attribute of an entity (C) and no values for the non-key attributes of the other entity (I).

Using this classification, we have made an exhaustive combination of tuples to be inserted in each entity, generating a total of 8 tuples for each: 1 complete tuple, 1 incomplete tuple, 3 partial tuples with 2 attributes with assigned values and 3 partial tuples with one attribute with an assigned value.

In the case of the relationship, we have followed the same approach, combining the different combinations of the two related entities. As the number of possible tuples for an entity is 8, the number of tuples we have inserted per relationship is 64 (8 multiplied by 8).

In this experimentation, the database always started empty.

IV.4.2.2 Test cases results: Entities

Table 9 displays the results of executing the test cases that inserted tuples of entities through MDICA.32 test cases were executed, 8 for each entity from.

Entity	Test cases represented	Blocked	INSERT	UPDATE	SELECT	Total
Venue (C)	1	0	0	0	0	0
Venue (P2)	3	0	0	0	0	0
Venue (P1)	3	0	0	0	0	0
Venue (I)	1	0	0	0	0	0
User (C)	1	0	0	0	0	0
User (P2)	3	0	0	0	0	0
User (P1)	3	0	0	0	0	0
User (I)	1	0	0	0	0	0
Review (C)	1	0	0	0	0	0
Review (P2)	3	0	0	0	0	0
Review (P1)	3	0	0	0	0	0
Review (C)	1	0	0	0	0	0
Artifact (C)	1	0	1	2	0	3
Artifact (P2)	3	0	1	2	1	4
Artifact (P1)	3	0	1	2	2	5
Artifact (I)	1	0	1	2	3	6

Table 9 Summary of second experimentation: Entities

Column **Entity** displays in which entity the tuple is inserted and a tag that indicates the type of tuple inserted: Complete (C), Partial 1 (P1), Partial 2 (P2), Incomplete (I). The number of test cases that each row represents is displayed in column **Test cases represented**. **Blocked** displays the test cases whose results was blocked, similar to the first experimentation. The database statements required by average in the test cases are displayed in columns **INSERT, UPDATE** and **SELECT,** with the number of statements for each of these operations and in column **Total** with the sum of all of these operations by average.

Similar to what was observed in the first experimentation, none of the tuples that contain values for attributes of the entities Venue, User and Review were inserted in any table as no table met the requirements to do so. In the rest of insertions, we were able to check that the logical consistency was maintained by MDICA, and no test case was blocked.

IV.4.2.3 Test cases results: Relationships

Table 10 displays the results of applying MDICA to determine the CQL statements needed to maintain the logical consistency over 320 insertions of tuples of relationships through MDICA.

The information of these tuples is displayed in different columns:

- Relationship: relationship of the entities whose attributes have assigned values in the tuple.
- Entity I and Entity II: entities related along the tag that indicates how many attributes of the entity have assigned values (C, P2, P1, I).
- Primary Relationship: if any related entities are a detail of other entities whose attributes were not initially in the tuple (many to one relationship), we include these relationships in the tuple by assigning values to the primary keys of the primary entities and indicate these relationships in this column.

The columns **Test cases represented**, **Blocked, INSERT, UPDATE** and **SELECT** display the same information as in the previous section for test cases where tuples of entities were inserted.

Relationship	Entity I	Entity II	Primary Relationships	Insertions represented	INSERT	UPDATE	SELECT	Blocked	Total
Features	Venue(C)	Artifact (C)	-	1	3	2	0	0	5
Features	Venue(P2)	Artifact (C)	-	3	3	2	0	0	5
Features	Venue(P1)	Artifact (C)	-	3	3	2	0	0	5
Features	Venue(I)	Artifact (C)	-	1	3	2	0	0	5
Features	Venue(C)	Artifact (P2)	-	3	3	2	3	1	8
Features	Venue(P2)	Artifact (P2)	-	9	3	2	3	1	8
Features	Venue(P1)	Artifact (P2)	-	9	3	2	3	1	8
Features	Venue(I)	Artifact (P2)	-	3	3	2	3	1	8
Features	Venue(C)	Artifact (P1)	-	3	3	2	6	2	11
Features	Venue(P2)	Artifact (P1)	-	9	3	2	6	6	11
Features	Venue(P1)	Artifact (P1)	-	9	3	2	6	6	11
Features	Venue(I)	Artifact (P1)	-	3	3	2	6	2	11
Features	Venue(C)	Artifact (I)	-	1	3	2	9	1	14
Features	Venue(P2)	Artifact (I)	-	3	3	2	9	3	14
Features	Venue(P1)	Artifact (I)	-	3	3	2	9	3	14
Features	Venue(I)	Artifact (I)	-	1	3	2	9	1	14
Posts	Review (C)	User (C)	Rates	1	1	2	0	0	3
Posts	Review (C)	User (P2)	Rates	3	1	2	0	0	3
Posts	Review (C)	User (P1)	Rates	3	1	2	0	0	3
Posts	Review (C)	User (I)	Rates	1	1	2	0	0	3
Posts	Review (P2)	User (C)	Rates	3	1	2	1	1	4
Posts	Review (P2)	User (P2)	Rates	9	1	2	1	1	4
Posts	Review (P2)	User (P1)	Rates	9	1	2	1	1	4
Posts	Review (P2)	User (I)	Rates	3	1	2	1	1	4
Posts	Review (P1)	User (C)	Rates	3	1	2	2	2	5
Posts	Review (P1)	User (P2)	Rates	9	1	2	2	6	5
Posts	Review (P1)	User (P1)	Rates	9	1	2	2	6	5
Posts	Review (P1)	User (I)	Rates	3	1	2	2	2	5
Posts	Review (I)	User (C)	Rates	1	1	2	3	1	6
Posts	Review (I)	User (P2)	Rates	3	1	2	3	3	6
Posts	Review (I)	User (P1)	Rates	3	1	2	3	3	6
Posts	Review (I)	User (I)	Rates	1	1	2	3	1	6
Rates	Review (C)	Artifact (C)	Posts & Features	1	4	2	0	0	6
Rates	Review (C)	Artifact (P2)	Posts & Features	3	4	2	3	0	9
Rates	Review (C)	Artifact (P1)	Posts & Features	3	4	2	6	0	12
Rates	Review (C)	Artifact (I)	Posts & Features	1	4	2	9	0	15
Rates	Review (P2)	Artifact (C)	Posts & Features	3	4	2	1	1	7
Rates	Review (P2)	Artifact (P2)	Posts & Features	9	4	2	4	1	10
Rates	Review (P2)	Artifact (P1)	Posts & Features	9	4	2	7	1	13
Rates	Review (P2)	Artifact (I)	Posts & Features	3	4	2	10	1	16
Rates	Review (P1)	Artifact (C)	Posts & Features	3	4	2	2	2	8
Rates	Review (P1)	Artifact (P2)	Posts & Features	9	4	2	5	6	11
Rates	Review (P1)	Artifact (P1)	Posts & Features	9	4	2	8	6	14
Rates	Review (P1)	Artifact (I)	Posts & Features	3	4	2	11	2	17
Rates	Review (I)	Artifact (C)	Posts & Features	1	4	2	3	1	9
Rates	Review (I)	Artifact (P2)	Posts & Features	3	4	2	6	3	12
Rates	Review (I)	Artifact (P1)	Posts & Features	3	4	2	9	3	15
Rates	Review (I)	Artifact (I)	Posts & Features	1	4	2	12	1	18
LikesV	User (C)	Venue (C)	-	1	1	0	0	0	1
LikesV	User (P2)	Venue (C)	-	3	1	0	0	0	1
LikesV	User (P1)	Venue (C)	-	3	1	0	0	0	1
LikesV	User (I)	Venue (C)	-	1	1	0	0	0	1
LikesV	User (C)	Venue (P2)	-	3	1	0	1	0	2
LikesV	User (P2)	Venue (P2)	-	9	1	0	1	0	2
LikesV	User (P1)	Venue (P2)	-	9	1	0	1	0	2
LikesV	User (I)	Venue (P2)	-	3	1	0	1	0	2
LikesV	User (C)	Venue (P1)	-	3	1	0	2	0	3
LikesV	User (P2)	Venue (P1)	-	9	1	0	2	0	3
LikesV	User (P1)	Venue (P1)	-	9	1	0	2	0	3
LikesV	User (I)	Venue (P1)	-	3	1	0	2	0	3
LikesV	User (C)	Venue (I)	-	1	1	0	3	0	4

Relationship	Entity I	Entity II	Primary Relationships	Insertions represented	INSERT	UPDATE	SELECT	Blocked	Total
LikesV	User (P2)	Venue (I)	-	3	1	0	3	0	4
LikesV	User (P1)	Venue (I)	-	3	1	0	3	0	4
LikesV	User (I)	Venue (I)	-	1	1	0	3	0	4
LikesA	Artifact (C)	User (C)	Features	1	6	2	0	0	8
LikesA	Artifact (C)	User (P2)	Features	3	6	2	2	0	10
LikesA	Artifact (C)	User (P1)	Features	3	6	2	4	0	12
LikesA	Artifact (C)	User (I)	Features	1	6	2	6	0	14
LikesA	Artifact (P2)	User (C)	Features	3	6	2	8	1	16
LikesA	Artifact (P2)	User (P2)	Features	9	6	2	10	1	18
LikesA	Artifact (P2)	User (P1)	Features	9	6	2	12	1	20
LikesA	Artifact (P2)	User (I)	Features	3	6	2	14	1	22
LikesA	Artifact (P1)	User (C)	Features	3	6	2	16	2	24
LikesA	Artifact (P1)	User (P2)	Features	9	6	2	10	6	18
LikesA	Artifact (P1)	User (P1)	Features	9	6	2	12	6	20
LikesA	Artifact (P1)	User (I)	Features	3	6	2	14	2	22
LikesA	Artifact (I)	User (C)	Features	1	6	2	16	1	24
LikesA	Artifact (I)	User (P2)	Features	3	6	2	18	3	26
LikesA	Artifact (I)	User (P1)	Features	3	6	2	20	3	28
LikesA	Artifact (I)	User (I)	Features	1	6	2	22	1	30
TOTAL	-	-	-	320	240	128	400	112	768

Table 10 Summary of second experimentation: Relationships

The results show that 208 test cases passed, verifying that MDICA maintains the logical consistency. However, in the remaining 112 test cases, the result was blocked the insertion was blocked by MDICA for the same reasons as the first experimentation. Whenever an attribute that has an associated key column in a target table lacked a value in the tuple, MDICA issued an AKC error. However, as in the first experimentation, we considered the blocked test cases as successful results as they were expected results.

V DATABASE EVOLUTION: SCHEMA EVOLUTION: CODEVO

This chapter details our approach for schema evolution in a column-oriented database when there are changes in the project requirements that change the conceptual model, which we named CoDEvo. The idea of a framework for database evolution has been presented in an international conference [29], focusing in this chapter on the schema evolution part from this framework. The detail of CoDEvo that is presented in this section is to be sent to the JCR journal "Journal of Systems and Software" after having received a first review from the reviewers.

Section V.1 introduces the problem of database evolution, particularizing in schema evolution. Section V.2 describes the general framework for database evolution. Section V.3 contains the description of CoDEvo, which focuses on the schema evolution part of the framework. Section V.4 details the experimentation performed using CoDEvo, comparing its results against the schema of open-source projects in order to validate it.

V.1 INTRODUCTION

In the previous two chapters of this thesis, we have addressed the problem of the maintenance of the logical consistency when the data stored in the database is modified through modifications of data at the conceptual level. In this chapter, we address the problems related to database evolution in column-oriented databases, particularizing in schema evolution. When the database schema is designed after a conceptual model (see section II.2.3) we say that there is a consistency between them that we name inter-model consistency, whose maintenance we approach in this chapter.

Typical evolution of a database comes from requirements change that directly changes the schema or the conceptual model. If either the conceptual model or the schema evolves without considering the other, inconsistencies can be created between the conceptual model and the schema. Several problems can happen because of these inconsistencies, such as a schema that allows future storage of data that contradicts the conceptual model constraints [29]. This last problem is significantly more difficult to approach in NoSQL column-oriented databases, as the same data is usually duplicated in several tables (see section II.1.6).

In addition to the maintenance of the inter-model consistency, the logical consistency must be maintained as well. The application queries are also affected by the evolution of the schema, as they must consider the latest version of the schema. Inconsistencies also increase the probability of mistakes in the schema design, such as allowing future storage of data that contradicts constraints specified in the conceptual model [29].

The main objective of this chapter is to determine how the schema must evolve after a change of requirements that evolved the conceptual model in order to maintain the inter-model consistency. This will be achieved through a MDE approach named CoDEvo that will determine these actions using model transformation. The contributions of this chapter are:

1. The definition of a framework for database evolution that covers the maintenance of the inter-model consistency, the logical consistency, and the client application update.
2. The determination of how the database schema must evolve to maintain the inter-model consistency with the conceptual model through model transformations. We

additionally determine a model that contains the detail of the evolved schema by applying the aforementioned transformations to the source schema.

3. A set of experiments to validate CoDEvo, where we apply CoDEvo to several changes in the conceptual models of real projects. We then compare the schemas that are in the repository of the projects with the schema generated by CoDEvo to determine if this schema satisfy the project requirements.

The remainder of this chapter is structured as follows. Section V.2 contains the description of the proposed framework for evolution of the database. Section V.3 details the design of the approach CoDEvo. Section V.4 contains the experimentation details and the threats to validity.

V.2 COLUMN-ORIENTED DATABASE EVOLUTION FRAMEWORK

We propose a framework to evolve the schema that starts with an evolution of the database structures of the schema due to a change in the project requirements. The framework requires models of the current conceptual model, the current schema, a mapping between conceptual model and schema and the change in the schema that triggers this procedure. This framework is composed of four stages that are illustrated in Figure 11 along with their interactions with each other:

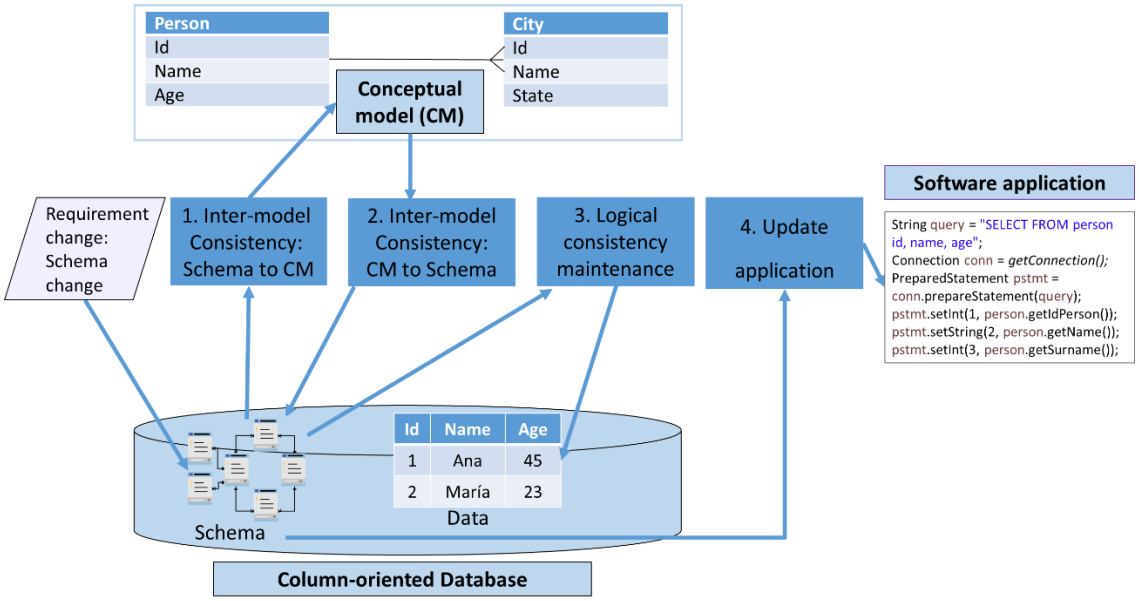


Figure 11 Full framework for column-oriented database evolution, from a requirement change to the update of the application

In the following paragraphs we describe each stage, detailing their objective and the how we intend to approach them.

Stage 1: Inter-model Consistency: Schema to CM: aims to maintain inter-model consistency after a change to the schema by performing the required updates to the conceptual model to reflect the change in the schema. We propose to use a MDE (model-driven engineering) approach that provides the actions to perform in the conceptual model.

Stage 2 Inter-model Consistency: CM to Schema: aims to maintain inter-model consistency by applying in the required changes to schema in order to adapt it to the conceptual model generated in stage 1. For this stage we have developed the MDE approach CoDEvo that

determines how the schema must evolve. We detail this stage in the following sections of this chapter.

Stage 3: Logical consistency maintenance: aims to maintain the logical consistency of the database after the schema evolves. To address this problem, we propose to use a MDE approach to determine the data migrations that are required to maintain the logical consistency. In order to determine these migrations, we will employ in future work the knowledge obtained in our previous works related to logical consistency maintenance that are detailed in the chapters III and IV of this thesis.

Stage 4: Update application: aims to update the client application by adapting it to the new schema. Although there is no existing work that studies this specific problem, program repair approaches are an interesting option [89]. These approaches aim to fix a bug or solve an inconsistency in software. We propose using a similar approach that updates the application in both the database statements that are embedded in the application and the application code that prepares the database statements and process the result of the execution of these database statements.

In the rest of this chapter, we focus on Stage 2. The remaining stages will be addressed in future work following this thesis.

V.3 CoDEvo description

In this chapter of the thesis, we focus on the stage 2 of the framework for database evolution described in the previous section by proposing **CoDEvo**, a model-driven engineering approach that addresses the evolution of the database schema using model transformations [90]. It uses higher-order transformations (HOTs) for this [91], managing the changes in the conceptual model, the database schema, and the data as models, which are named transformation models. In addition to these transformation models, CoDEvo provides output models with the transformations applied.

The section is divided in several subsections. Subsection V.3.1 defines several terms used during the chapter. Subsection V.3.2 contains the description of the metamodels and the general process of CoDEvo. Subsections V.3.3 and V.3.4 contain the detail of each model that is used in CoDEvo. Subsection V.3.5 contains the detail of how, given a conceptual model change, CoDEvo determines the actions to evolve the schema in order to maintain the inter-model consistency. Subsection V.3.6 describe how the database code to perform the required changes in the schema is obtained.

V.3.1 Related Terms definitions

In this section, we define several terms related to model transformation and NoSQL databases that we will use during the description of CoDEvo.

Inter-model consistency: A guarantee that the database schema conforms to the constraints defined in the conceptual model. Any evolution of either the conceptual model or the database schema may affect the other.

Metaclass: A class of a metamodel. The instances of these metaclasses are named **elements**. These elements are contained in the models that conform to the metamodels.

Evolution: One or more changes in either the conceptual model or the database schema that are required to maintain the inter-model consistency. A **change** is a single insertion, removal, or update of a single element from either the conceptual model or the schema.

Transformation: Generation of a target model or textual artifacts given one or more source models [63]. A transformation is executed based on a trigger condition and performs a set of actions to generate a specific target model, which together comprise a **transformation rule**. A **transformation definition** is composed of several transformation rules that together describe how the target models or texts are generated.

Transformation model: A model that contains the changes that are going to be applied in a *transformation*. Higher-order model transformations (HOTs) use this type of model, with several types depending on the amount of source models and target models [91]. In this work we will use the (de)composition transformation type, which requires having at least one transformation as an input model and one transformation as an output model.

V.3.2 General process of CoDEvo and metamodels used during the process

The metamodels, models that conform to them, database procedures and transformation definitions that compose CoDEvo are illustrated in Figure 12 with the following notation: Metamodels as rounded corner rectangles, models as blue (input) or green (output) rectangles, database procedures as magenta rectangles and transformation definitions as ellipses. The models conform to metamodels with the same name, except for the Source schema and Target schema, that both conform to the metamodel DB schema. This is displayed with dashed arrows that connect the model with the metamodel that they conform to. The models are connected through properties that are defined in the following subsection (e.g., a column is associated with the attribute mapped to it through the identifier of the attribute). These metamodels, models, database procedures are organized [64] in three layers:

1. **Computationally Independent Models** (CIM): Models independent of the data model of the database schema and the technology: **Conceptual Model**, **Queries** and **Conceptual Model Evolution**.
2. **Platform Independent Models** (PIM): Models that depend on the data model of the database schema, but not on a specific technology: **Source schema**, **Schema evolution**, and **Target schema.** These models will be either inputs or outputs in the following M2M (Model to Model) transformation definitions:
   a. **EvolveSchema:** Receives as inputs all the models from CIM and S*ource schema* and generates the model *Schema evolution* that specifies all the changes of the *DB schema* required to reflect the conceptual model change.
   b. **ApplyEvo**: Receives as inputs the *Source schema* and the *Schema evolution* and generates the *Target schema* which contains the schema structure detail after the changes defined in *Schema evolution* are applied to the schema.
3. **Platform Specific Models** (PSM): Models that depend on the database technology. This layer contains the M2T (Model to Text) transformation definition **SchemaGen,** that generate database procedures specific to a particular database technology (*Schema statements***)**. These database procedures contain the database statements and instructions required to perform the changes specified in the models *Schema evolution*.

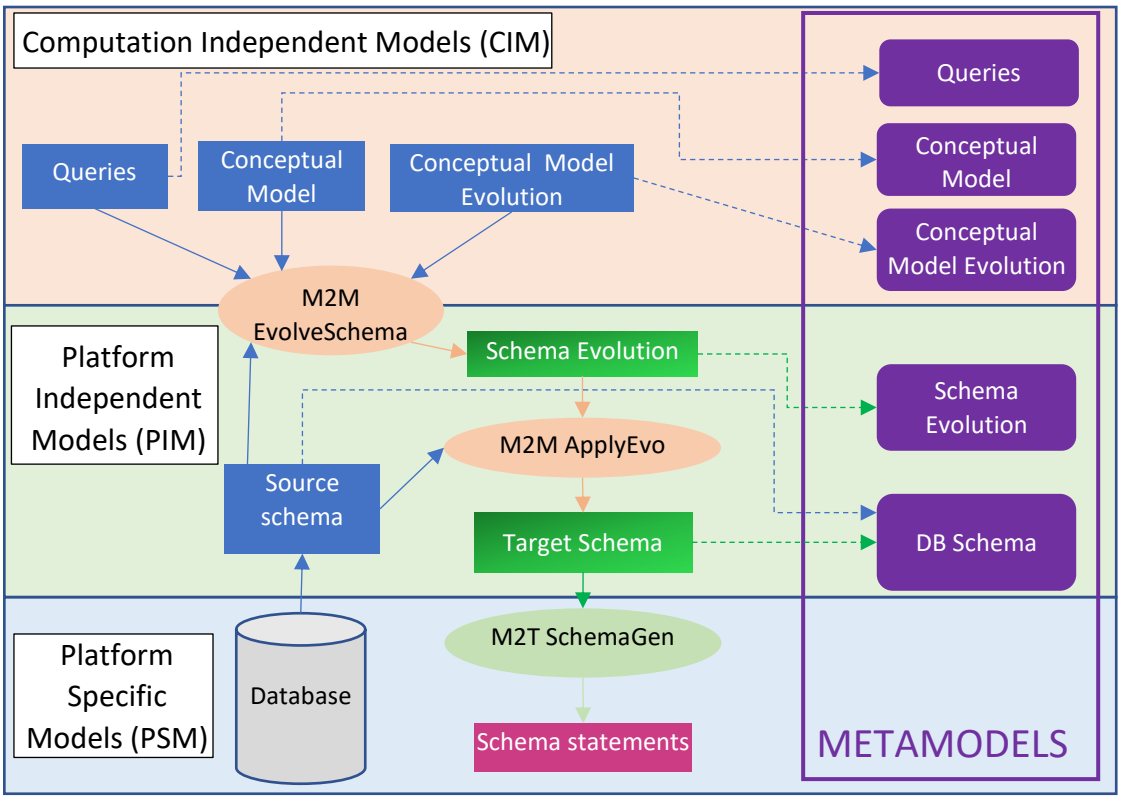


Figure 12 General process of CoDEvo. Solid arrows display the inputs and outputs of Model-To-Model and Model-To-Text transformations. Doted arrows display the metamodel that a model conforms to.

In the following subsections of this section, we describe these metamodels and transformation definitions. Each metamodel will be explained and illustrated separately in following subsections, although the relationships of a class with a class of another metamodel are illustrated through reference objects. All these metamodels are joined together in a full metamodel where the relationships between classes of the different individual metamodels are established. This full metamodel is illustrated in the appendix in Figure 22.

V.3.3 Input models

The input models of CoDEvo are the *conceptual model*, *queries*, *source schema* and *conceptual model evolution*. The metamodels of these four models are based on the metamodels defined by de la Vega et al. [19] which they based on the work of Chebotko et al. [18]. In each of the following subsubsection, we detail the components of each model:

V.3.3.1 Conceptual model metamodel

The *conceptual model* metamodel is composed of the metaclasses **Entity**, **Weak entity**, **Attribute** and **Relationship**. The entities and weak entities are associated with one or more attributes**.** The property 'isUnique' from an attribute specifies if it is part of the primary key. A relationship associates two entities (can be weak) with a specific cardinality: 1:1 (one to one), 1:n (one to many) or n:m (many to many), established by the attributes cardinality1 and cardinality2. This metamodel is illustrated in Figure 13.

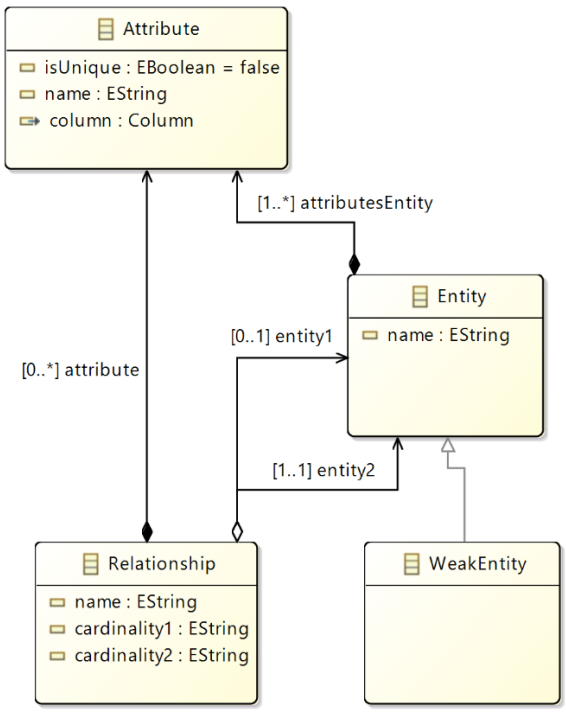


Figure 13 Conceptual Model metamodel

V.3.3.2 Queries metamodel

The metaclasses of the *queries* metamodel are **Requirement Query**, **Selection** and **Filter**. Note that these requirement queries are the conceptual queries used for designing a table of the database and they do not have a direct relationship with the application queries (SELECT, INSERT, DELETE, UPDATE). Thus, a requirement query element is associated to a table element from the model *source schema*. A requirement query element is also associated through composition to a selection element and a filter element. Each selection element is associated with several attributes from the model *conceptual model*, which specifies the attributes that are requested by the query. Each filter element is also associated to several attributes, which are used to define restrictions in a query, similar to the WHERE clause of a SQL query. This metamodel is illustrated in Figure 14.

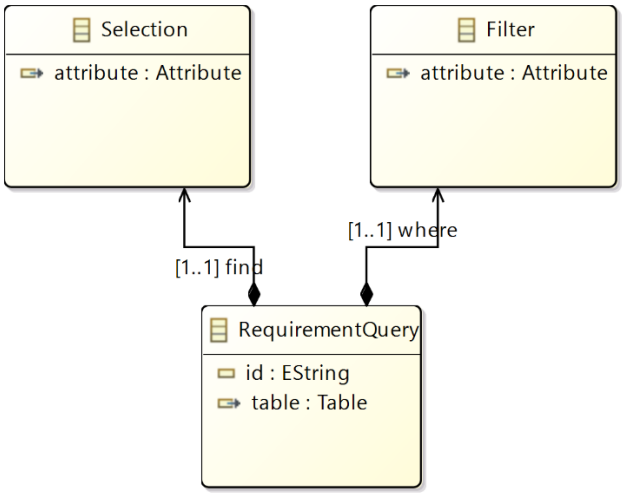


Figure 14 Queries metamodel

V.3.3.3 Schema metamodel

Source schema conforms to the metamodel *DB schema*, which contains the design details of a NoSQL column-oriented database schema with the metaclasses **TableFamily**, **Table**, **Type** (also named custom type) and **Column**. All the tables compound a TableFamily. A table or a custom type of element is associated to one or more column elements. Additionally, a column contains a property names 'isKey' to indicate if it is part of the primary key as well as an association with an attribute from the *conceptual model***,** which we refer as mapping. A table element contains the reference "query" that associates it to the queries that it satisfies from the *queries* model. A custom type can be associated with multiple columns from a table. This metamodel is illustrated in Figure 15.

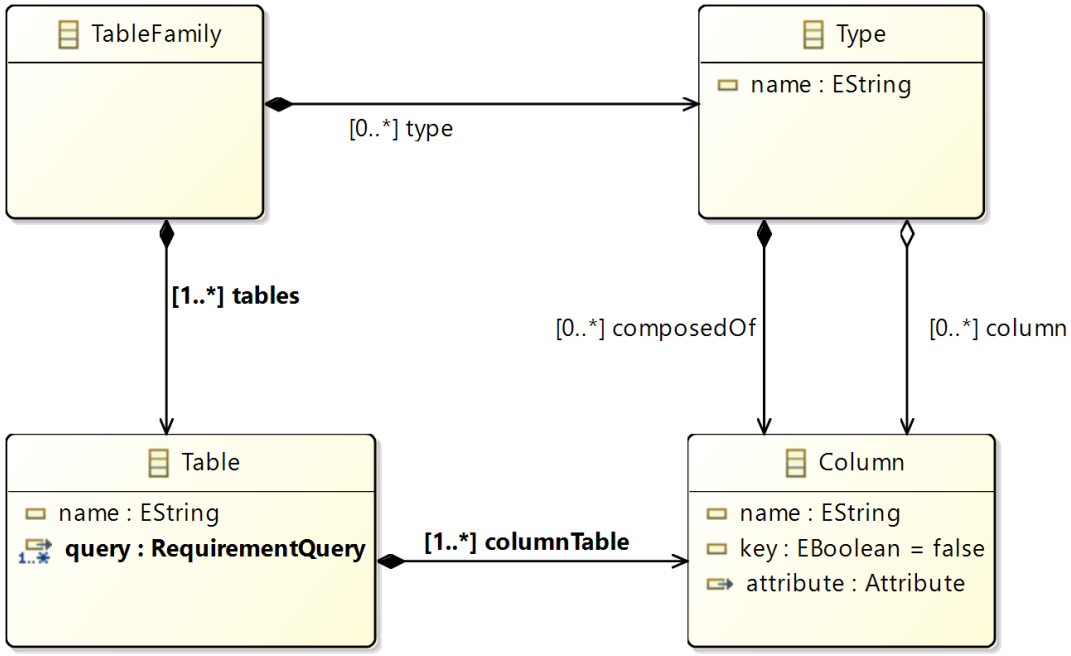


Figure 15 Column family database schema metamodel

V.3.3.4 Conceptual model evolution metamodel

The last input is the *conceptual model evolution* that contains the detail of changes performed in the *conceptual model*. The scenarios that we consider for changes in the conceptual model are obtained from two types of sources: from research works that address the evolution of different databases and empirically from detection of cases in real case projects, which are detailed in section V.4. In Table 11, we classify the conceptual model changes by the conceptual model structure directly affected by the change. We also specify if the conceptual model change was detected in a project (P) or/and in a research work (RW):

Structure	Change	Description	P	RW
Entity	AddEntity	Creation of a new entity along with the attributes associated to it.	X	X
Entity	AddWeakEntity	Creation of a new weak entity and its relationship with the owner entity.	X	X
Entity	DeleteEntity	Removal of an entity, including all relationships the entity had.	X	X
Entity	MergeEntity	Merge of two exiting entities into a resulting one		X
Entity	SplitEntity	Split one existing entity into one.		X
Entity	AddPK	Change of a non-key attribute of an entity into a key attribute.	X	X
Entity	RemovePK	Change of a key attribute of an entity into a non-key attribute.	X	X
Attribute	AddAttribute	Creation of a new attribute and association to an existing entity.	X	X
Attribute	RemoveAttribute	Removal of an attribute from an entity	X	X
Attribute	SplitAttribute	Split an existing attribute from an entity into two or more attributes.	X	
Relalationship	AddRel	Establishment of a new relationship between two entities. It also defines the cardinality of the relationship (1:1, 1:n or n:m).	X	X
Relalationship	UpdateCardinality	Change of the cardinality of an existing relationship.	X	
Relalationship	RemoveRelationship	Removal of an existing relationship		X

Table 11 Conceptual Model changes detected in projects and research work

The metamodel is illustrated in Figure 16.

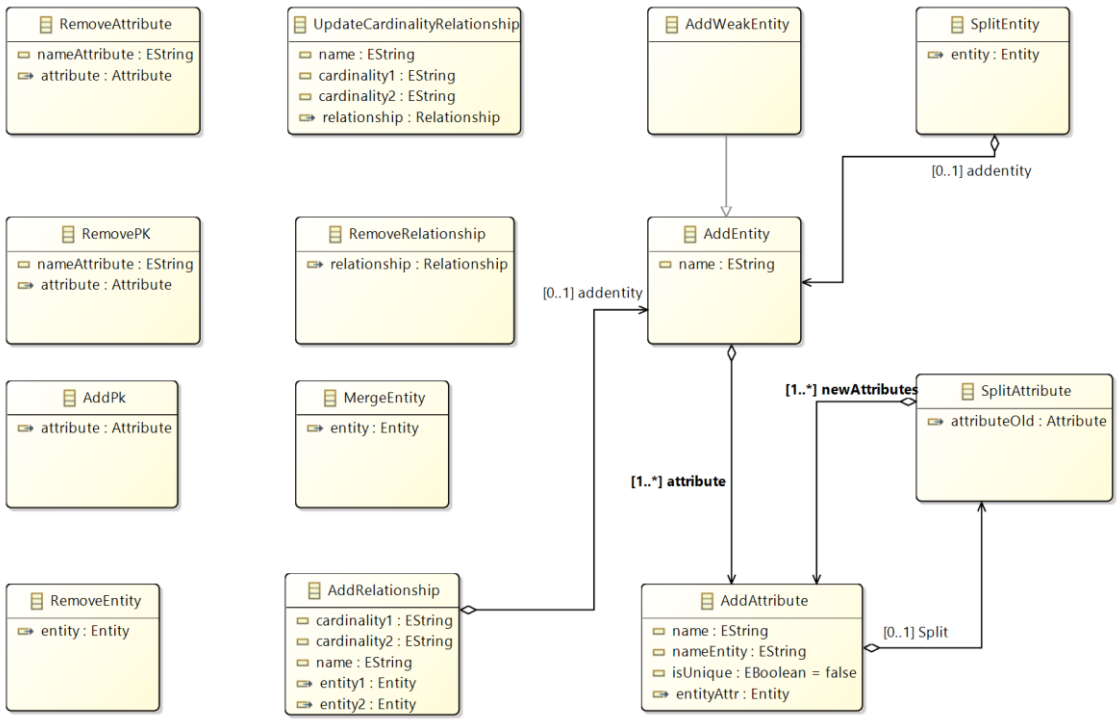


Figure 16 Conceptual Model Evolution metamodel

V.3.4 Output models

The M2M transformation definition EvolveSchema uses the inputs described in the previous subsection to obtain the output model SchemaEvolution. SchemaEvolution contains classes that describe how the source schema must evolve through associations with elements from other metamodels:

- **Add** which details the tables, types and columns that need to be added to the schema.
- **Remove** that details the tables and columns that need to be removed from the schema.
- **AddPK** that details the columns that need to be added to the primary key of a table.
- **RemovePK** that details the columns that need to be removed from the primary key of the table.

The M2M transformation definition ApplyEvol conform to the metamodel “Schema” which was detailed in the previous subsection. These classes are illustrated in Figure 17.

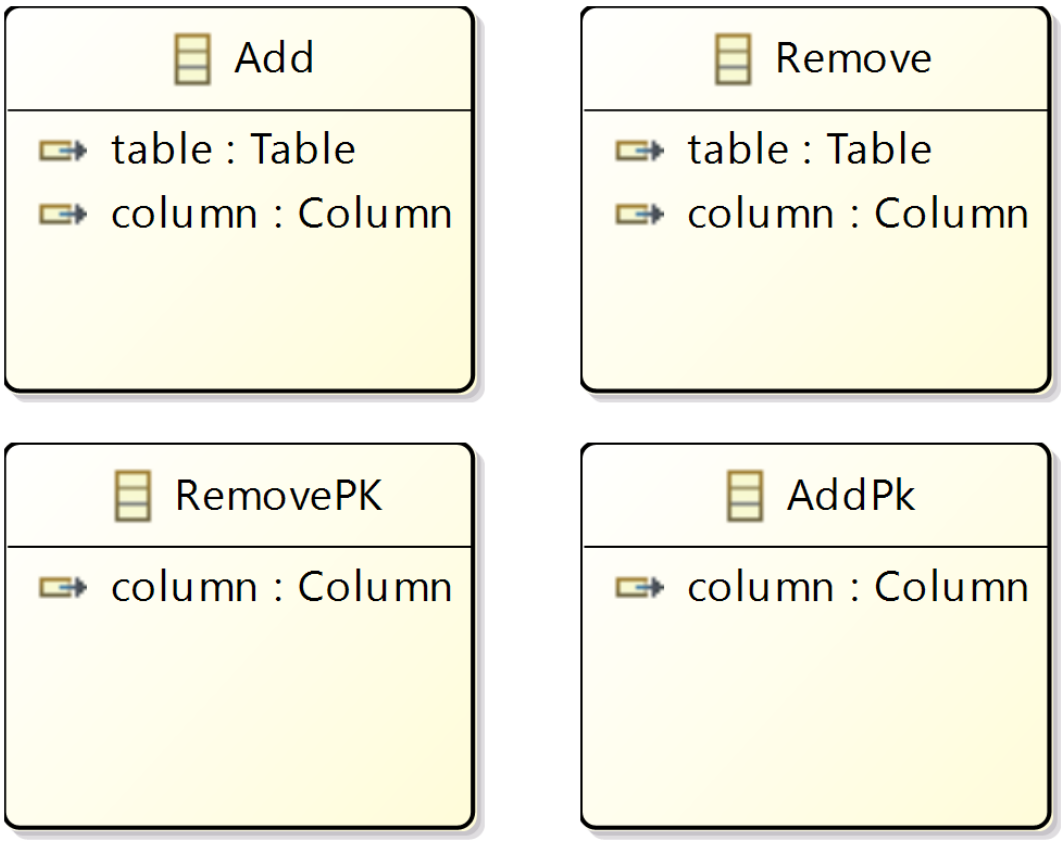


Figure 17 Schema Evolution metamodel

V.3.5 M2M EvolveSchema transformation definition and ApplyEvo description

EvolveSchema contains the transformation rules that specify the transformations of the schema required to maintain the inter-model consistency after a specific change in the conceptual model. These transformations are displayed in the output model *schema evolution*. The inputs of these transformation are the four models described in the previous subsection.

Each transformation rule is composed of several predicates, functions, and atomic transformations. A predicate is an expression that provides a Boolean value given specific properties of one or more elements. A function receives one or two elements as inputs and returns either a property of this element (e.g., name, cardinality) or one or more elements associated to the input elements. An atomic transformation is an operation that, given one or more elements, generates or removes a single element from *source schema* or creates an association between two elements of *source schema*. To differentiate between functions, predicates and transformations, the names of functions and predicates start with lowercase and the names of transformations start with uppercase.

We denote each element (entity, attribute, table…) as follows:

- Conceptual Model:
  - e: An **entity**.
  - a: An **attribute**. It is associated with one entity.
  - r: A **relationship**. It is associated with two entities, establishing a certain cardinality: 1:1, 1:n or n:m.
- Schema:
  - t: A **table**. When in uppercase ('T'), it represents all the tables that are in the model.
  - ct: A **type** or **custom type**. It can be associated with several tables in an aggregation.
  - c: **A column**. It is associated with either a table or a custom type in a composition. Each column is also associated with an attribute (mapping). At least one of the columns associated to a particular table must be part of the primary key.
- Query:
  - rq: A **requirement query**. It is associated with one table that fits it.
  - f: A **filter**. It is associated with a query and a set of attributes.
  - s: A **selection**. It is associated with a query and a set of attributes.

V.3.5.1 Predicates and functions

The predicates used in the definition of the transformations are as follows:

- **isIn (a, q):** if attribute *a* appears anywhere (select or filter) in the requirement query *q*.
- **isKey (c):** if *c* is part of the primary key of the table where it belongs.
- **isKey (a):** if attribute *a* is part of the primary key of the entity that is associated.
- **isKey (a, t):** if the attribute *a* is mapped to a column that is part of the primary key in entity *e.*
- **isMapped (a, c)**: if the attribute *a* is mapped to the column *c.*
- **mapped (e, t):** if entity *e* has at least one attribute mapped with one column associated with table *t*.
- **mappedPK (e, t):** if the attributes that are primary key of entity *e* are mapped to columns associated with table *t* that are key.
- **mapped (a, t):** if the attribute *a* is mapped with any column associated to table *t*.

The functions used in the transformations, which either return information about the association of an element with another or properties of an element, are the following:

- **attribute (c):** returns the attribute that the column *c* is mapped to.

- **cardinality (e1, e2):** returns the cardinality of the relationship between the entities e1 and e2: '1:1', '1:n' or 'n:m'.
- **column (a, t):** returns the column from *t* that is mapped to *a.*
- **top (r):** in a relationship *r* with cardinality "1:n", returns the entity on the side of the cardinality value "1".
- **bottom (r):** in a relationship *r* with cardinality "1:n", returns the entity on the side of the cardinality value "n".
- **query (t):** returns the set of queries that are associated to the table *t*.
- **left (r):** in a relationship *r* with cardinality "1:1" or "n:m", returns the entity on the left side.
- **right (r):** in a relationship *r* with cardinality "1:1" or "n:m", returns the entity on the right side.

V.3.5.2 Transformations

The actions required to evolve the schema while maintaining the inter-model consistency are obtained using transformations of models. These actions are ultimately used in transformations, which we classify in **atomic transformations** and **transformations rules**. Atomic transformations perform at most two actions that can either be creation of elements or association of existing elements. In the case of transformation rules, each one is triggered by a specific conceptual model change and their objective is to maintain inter-model consistency to evolve the schema considering the conceptual model change. We formally define the transformation rules, using atomic transformations, predicates, and functions.

V.3.5.2.1 Atomic Transformations

The atomic transformations that we use in the transformation rules are the following:

- **Associate (t, c)**: associates the column *c* with the table *t*.
- **Associate (t, ct):** associates the custom type *ct* with the table *t.*
- **CopyAddPK (t, c)**: creates a copy of table *t* with the column *c* added to the primary key.
- **CopyRemPK (t, c)**: creates a copy of table *t* that does not contain *c* in the primary key. In order to perform this atomic transformation, there must be other columns that are part of the primary key.
- **CreateCol (a)**: creates a column based on attribute *a*, naming this new column with the same name as *a*.
- **CreateColPK (a)**: creates a column based on attribute *a*, naming this new column with the same name of *a* and sets this column as key.
- **CreateTable (e, {c1, c2...cN})**: creates a table with the name of the entity *e,* and associates to this new table the set of columns *c1, c2...cN*. Note that the symbols '{}' denote a set of elements.
- **CreateTable (r, {c1, c2...cN})**: creates a table with the name of the relationship *r,* and associates to this new table the set of columns *c1, c2...cN*.
- **CreateType (e, {c1, c2...cN})**: creates a custom type with name *e* and associates to it the set of columns *c1, c2...cN*.
- **Remove (c)**: removes column *c* and its associations (tables and attributes). If c is part of the primary key of a table, there must be other columns that are part of the primary key in the associated table to perform this operation.
- **Remove (t)**: removes table *t* from the model as well as its associated columns.

V.3.5.2.2 Transformation rules

For each of the transformation rules, there is a formal definition of how the database schema must evolve using the functions, predicates, and atomic transformation previously defined. In certain transformation rules we use additional predicates that are only used in these rules. For each of these additional predicates there are both an informal explanation of its purpose as well as a formal definition.

In the formal definition of some rules, we use the symbol '|', which in a notation such as "action **|** condition", represents that the action is performed for each time that the condition is met.

The transformation rules that we have defined are the following:

Rule 1. **AddEntity (e)**: Add an entity *e*, composed of several attributes *a,* to the conceptual model:

AddEntity(e) := CreateTable (e, {CreateCol(a) | a ∈ e)})

Rule 2. **AddWeakEntity (we)**: Add a weak entity *we,* composed of attributes *a* to the conceptual model. This weak entity also contains a relationship with a primary entity *pe*:

AddWeakEntity (we) := let ct = CreateType (we, {CreateCol(a) | a ∈ we})
∀ t ∈ T, mapped (we, t): Associate(t, ct)

The first step of the transformation is the creation of the custom type, which is assigned to a variable ct. This custom type is then associated to tables that store the primary entity of the relationship. This association is only required in one of the selected tables, being the developer the responsible of choosing in which of the selected tables the association is performed.

Rule 3. **AddPK (a, e)**: Add the attribute *a* to the primary key of an entity *e*. Let *P (t, a)* be the predicate that checks if there is a key column mapped to the attribute *a* in a given table *t*.

P(t, a):=∃c∈t, (attribute(c) == a ∧ isKey (c)):
AddPK (a, e) :=∀ t∈T, mappedPK (e, t) ∧ ¬P(t, a):CopyAddPK(t, Create (a)),Remove (t)

The new columns associated to *a* are inserted as PK in every table that contains e to guarantee that each row inserted in the table is not overwritten by mistake.

Rule 4. **RemovePK (a, e)**: Remove the attribute *a* from the primary key of an entity *e*. Let PQ (t, a, c) be the predicate that checks if a column *c* is mapped to an attribute *a* and this attribute *a* is not in any of the queries that table *t* is associated to.

PQ (t, a, c) := attribute(c) == a ∧ ¬isIn (a, query (t))
RemovePK (a, e) := ∀ t ∈ T, mapped (a, t) ∧ ∃c∈t, PQ (t, a, c): CopyRemPK (t, c), Remove (t)

Rule 5. **AddAttribute (a, e)**: Add a new attribute *a* to *e*:

AddAttribute (a, e) := ∀ t∈T, mapped (e, t): Associate(t, CreateCol(a))

From all the tables *t* that satisfy the selection criterion, it is only required to associate the new column to one of them. This decision is up to the developer.

Rule 6. **RemoveAttribute (a, e) (Case 1)**: Remove an attribute *a* from *e.* The target tables affected by the transformation **must not contain** any column that is to be removed in the primary key . Let *P (a, c)* be a predicate that checks if the column *c* is non-key as well as mapped to *a*.

$P(a,c)=$ attribute (c) == a ∧¬isKey (c):
RemoveAttribute (a, e):= ∀ t ∈ T, mapped (e, t) ∧ ∃c∈t, P (c, a): Remove (c)

Rule 7. **RemoveAttribute (a, e) (Case 2):** Remove an attribute *a* from *e.* The target tables affected by the transformation **must contain** any column that is to be removed in the primary key. Let *P(a, c)* be a predicate that checks if there is a column *c* mapped to an attribute *a* and that *c* is part of the primary key.

P(a, c) := attribute(c)== a ∧ isKey (c)
RemoveAttribute (a, e) := ∀ t ∈ T, mapped (e, t) ∧ ∃c∈t, P (c): CopyRemPK(t, c), Remove (t)

Rule 8. **SplitAttribute (a, {a1, a2... an})**: Split an attribute *a* to several attributes *a1, a2, … an.*

SplitAttribute (a, {a1, a2... an}) := ∀t∈T, mapped (a, t):
{Associate(t, CreateCol (a1, a2... an))} , Remove(column(a, t))

Rule 9. **AddRelationship (r) (Case 1):** Add a new relationship *r* between two entities with a cardinality of either 1:1 or n:m.

AddRelationship (r) := let e1 = left(r), e2 = right(r)
CreateTable(r, {CreateColPK(a) | a ∈ e1 ∨ e2, isKey(a)} ∪ {CreateCol (a) | a ∈ e1 ∨ e2, ¬ IsKey(a)})

Rule 10: **AddRelationship (r) (Case 2):** Add a new relationship *r* between two entities with a cardinality of 1:n.

AddRelationship (r) := let e1 = top(r), e2 = bottom(r)
CreateTable(r, {CreateColPK(a) | a ∈ e2, IsKey(a)} ∪ {CreateCol (a) | a ∈ e1 ∨ e2, ¬IsKey(a) ∨ a ∉ e2)})

Rule 11: **UpdateCardinality (r) (Case 1):** Update the cardinality of a relationship *r* with a new cardinality of 1:n. Let P (t, e) be a predicate that checks if all the key attributes of an entity *e* have columns mapped to the primary key of a table *t*.

P (t, e) :=∀a∈ e, isKey(a) ∧ isKey(a, t)

UpdateCardinality (r) := let e1 = top (r), e2 = bottom (r)

∀t∈T, mapped (e1, t) ∧ mapped (e2,t) ∧ ¬P (t, e2): CopyAddPK(t, CreateCol(pk(e2))), Remove(t)

Rule 12. **UpdateCardinality (r) (Case 2):** Update the cardinality of a relationship *r* to a new cardinality n:m. Let P be a predicate that checks if a table *t* contains key columns for all attributes that are part of the primary key of the entities *e1* and *e2*.

P (t, e1, e2) := ∀a∈ e1 ∨ e2 , isKey(a) ∧ isKey(a, t)

UpdateCardinality (r) := let e1 = left (r), e2 = right (r)

∀t∈T, mapped (e1, t) ∧ mapped (e2,t) ∧ P (t, e1, e2):

CopyAddPK(t, CreateCol(pk(e2)) ∪ CreateCol(pk(e1))), Remove(t)

Rule 13. RemoveEntity (e): Remove an entity *e* from the conceptual model**.** We consider three scenarios (1, 2.a and 2.b), depending on the characteristics of the table from the schema where transformations need to be performed.

RemoveEntity (e) :=

∀t∈T, mapped (e, t) ∧ (∀ c ∈ pk(t), attribute(c)): RemoveTable(t)

∀t∈T, mapped (e, t) ∧ ¬(∀ c∈t, attribute(c)∈e):

∀c∈t attribute(c)∈e ∧ ¬isKey (c): Remove(c)

∀c∈t attribute(c)∈e ∧ isKey (c): CopyRemPK(t, c) , Remove (t)

In scenario 2.b, the creation of a copy table without the columns selected is only done once

Rule 14. MergeEntity (e1, e2): Merge two entities *e1* and *e2* into one entity *e*, We consider two scenarios (1 and 2) depending on the cardinality of the relationship between e1 and e1.

MergeEntity (e1, e2) :=

let a1pk = pk (e1), a2pk = pk(e2), car =cardinality (e1, e2)

1. ∀t∈T, mapped (e1, t) ⊻ mapped (e2, t):

   CopyAddPK (t, {CreateCol (a1pk) ∪ CreateCol (a2pk)}, Remove (t)

2. ∀t∈T, mapped (e1, t) ∧ mapped (e2, t) ∧ car ≠"n:m":

   CopyAddPK (t, {CreateCol (a1pk) ∪ CreateCol (a2pk)}, Remove (t)

Rule 15. RemoveRelationship (r) Remove a relationship *r* from the conceptual model. We consider two scenarios depending on whether the table contains only information of the relationship (scenario 1), or it also contains information from other entities (scenario 2).

RemoveRelationship (r):=
let e1 = left(r), e2 = right(r)
1. ∀t∈T, mapped (r, t) ∧ (∀c∈t, attribute(c)∈e1 ∨ attribute(c)∈e2):RemoveTable(t)
2. ∀t∈T, mapped (r, t) ∧ (∀c∈t, attribute(c)∈e1 ∨ ¬attribute(c)∈e2): SplitTable (t, e1, e2)

Case for the conceptual model change "Split Entity": There are no changes in the schema. Therefore, no transformation rule is created for this conceptual model change.

After the transformations are executed, the model *schema evolution* is generated, containing the changes that need to be performed. After this, the transformation definition **ApplyEvo** begins, which generates the final schema with the changes determined in *schema evolution* applied to it.

V.3.6 M2T SchemaGen and M2T DataModifier descriptions

SchemaGen and **DataGen** generate the database statements and actions required to perform in an Apache Cassandra database, the operations specified in models S*chema evolution* and *Data modification*. SchemaGen generates the database statements that evolve the database schema in its correct sequence. DataGen generates a code-like text that details how to migrate the data.

V.4 EXPERIMENTATION AND VALIDATION

In this section, we validate CoDEvo through its application to nine case studies of projects stored in public repositories that use the column-oriented database Apache Cassandra. These repositories contain the information of the changes in the schema that have been performed since the creation of the projects, which are recorded through commits in the repository. We considered that each commit where the schema is changed generates a new **version** of the schema, which may contain one or more conceptual model changes. During the experimentation, for each version, we compared the schema of the repository with the schema generated by CoDEvo. In this section, we refer to these schemas as '**repository schema**' and '**CoDEvo schema**', respectively.

CoDEvo has been automated by implementing the transformation rules defined in section V.3.5 in an Eclipse project using ATL. CoDEvo receives as inputs the models that contain necessary information for the evolution of the schema and generates as outputs the models that define how the schema must be evolved. The detail of these inputs and outputs and how they are obtained is displayed in the following table.

Component	How it was obtained
Input: **Conceptual model** ATL file	**Manually** through reverse engineering analyzing the database schema (tables, columns, primary key of the tables).
Input: **Source schema** ATL file	**Automatically** by reading the CQL script of the repository schema from a particular version. It also contains a **semi-automatically** obtained mapping between attributes of the conceptual model and columns of the schema.
Input: **Queries** ATL file	**Automatically** by using the information provided in the mapping. Each one of the queries contains an association of attributes from the conceptual model that define how each table of the schema is defined.
Input: **Conceptual model evolution** ATL file	**Manually** by defining the conceptual model changes, which are obtained by analyzing the differences that exist between the conceptual model of this version and the previous one.
Output: **Schema evolution** ATL file (CoDEvo schema)	**Automatically** executing M2M transformation **EvolveSchema.**
Output: Database **schema statements** to execute the changes to perform in the schema	**Automatically** executing M2T transformation **SchemaGen**.
Output: **Target Schema** after the changes are applied	**Automatically** executing the M2M transformation **ApplyEvo**.

Table 12 Components of evaluation of how they were obtained

The projects that we use for this validation are the following:

- **Minds** (https://gitlab.com/minds), a social network
- **PowSyBl** (https://github.com/powsybl), a framework for the implementation of power system simulations and analysis software
- **Thingsboard** (https://github.com/thingsboard/thingsboard): IoT platform for data collection
- **Wireapp** (https://github.com/wireapp/wire-server): Encrypted communication app
- **Blobkeeper** (https://github.com/sherman/blobkeeper): Distributed file-storage service
- **Reviews-service** (https://github.com/bon-app-etit/reviews-service): Module for restaurants
- **Sunbirds** (https://github.com/ekstep-sp/sunbird-devops): A project for learning and human development
- **Doudouchat** (https://github.com/doudouchat): No description provided
- **Sop** (https://github.com/SharedCode/sop): A database engine within a code library.

All conceptual model changes detected in these projects are displayed in Table 13.

Conceptual Model Changes	Minds	PowSyBl	Thingsboard	wireapp	Blobkeper	Review-service	Sunbirds	Doudochat	Sop	All projects
AddEntity	15	6	0	8	2	0	9	2	2	44
SplitAttribute	1	0	0	0	0	0	0	0	0	1
AddAttribute	21	35	4	30	2	0	27	11	4	134
AddRelationship	17	11	0	7	0	0	6	0	0	41
RemovePK	2	0	0	0	0	0	0	2	0	4
RemoveAttribute	2	5	0	2	0	0	11	1	1	22
AddPK	2	0	1	0	0	0	0	2	0	5
UpdateCardinality	0	0	0	0	0	1	0	0	0	1
AddWeakEntity	0	14	0	0	0	0	0	0	0	14
RemoveEntity	0	0	0	1	0	0	0	1	0	2
TOTAL	60	71	5	48	4	1	53	19	7	268

Table 13. Conceptual Model changes detected

V.4.1 Research questions

In order to validate CoDEvo, we have defined the following research questions:

RQ6. How much difference is there between the CoDEvo schemas and the repository schemas?

RQ7. Are the CoDEvo schemas valid considering the project requirements?

To answer RQ6 and RQ7, we used the conceptual model changes introduced in each project version as inputs for CoDEvo along with the schema from the previous version. Then, we compared the schema from the repository with the schema generated by CoDEvo for each version to obtain their differences.

In the following subsections we answer each RQ separately and the threats to validity.

V.4.1.1 RQ6: How much difference is there between the CoDEvo schemas and the repository schemas?

To respond this research question, we compared the schema in the project repository with the schema generated by CoDEvo to obtain the differences between them for each project version. We categorized these differences in one of three types based on the following:

1. **Type I, Same results**: When both the repository schema and the CoDEvo schema are the same.
2. **Type II, More database structures**: When the schema generated by CoDEvo contains more database structures than the ones in the schema from the project repository. These differences were detected when adding both a new entity (conceptual model change "AddEntity") and a new relationship of this entity with another one (conceptual model change "AddRelationship) in the same version.
3. **Type III, Different database structures**: When new database structures in the repository schema and the CoDEvo schema are different. There are three possible Type III subtypes depending on the conceptual model change that triggers the difference:
   a. **Type III-A: New non-boolean attribute (**conceptual model change "AddAttribute")
   b. **Type III-B: New boolean attribute (**conceptual model change "AddAttribute")

c. **Type III-C: New relationship** (conceptual model change “AddRelationship”).

Table 14 displays the number of repository versions in accordance with the aforementioned types.

Project	I	II	III			TOTAL
			III-A	III-B	III-C	
Minds	10	9	4	0	2	25
PowSybl	14	0	0	0	2	16
Thingsboard	4	0	0	0	0	4
Wireapp	8	5	3	1	0	17
Blobkeeper	4	0	0	0	0	4
Review-service	1	0	0	0	0	1
Sunbirds	13	1	0	0	0	14
Doudouchat	8	1	0	0	0	9
Sop	3	0	0	0	0	3
TOTAL	65	16	7	1	4	93
Percentage	69.9	17.20	7.5	1.1	4.3	100

Table 14 Types of versions per project

In the following subsections we analyse the differences detected between the CoDEvo schema and the repository schema of each version, excluding Type I where both schemas are the same (69.9% of the total).

V.4.1.1.1 Analysis of Type II schema differences

Type II results were detected when the CoDEvo schema contained more database structures than the repository schema. These results were only detected when both an entity and a relationship of this entity with another one are added in the same version. Both the CoDEvo schema and the repository schema contained a table for the relationship (Rules 9 and 10). However, CoDEvo additionally created a table to store only information of the entity (Rule 1).

Figure 18 depicts the insertion of a new entity ‘Hashtags’ and its relationship with entity ‘User’, highlighting the differences between the schemas from the repository and CoDEvo. Both the repository schema and the CoDEvo schema contain the table “User_Hashtags” for the new relationship. However, as described before, CoDEvo additionally added a table “Hashtags” for the new entity.

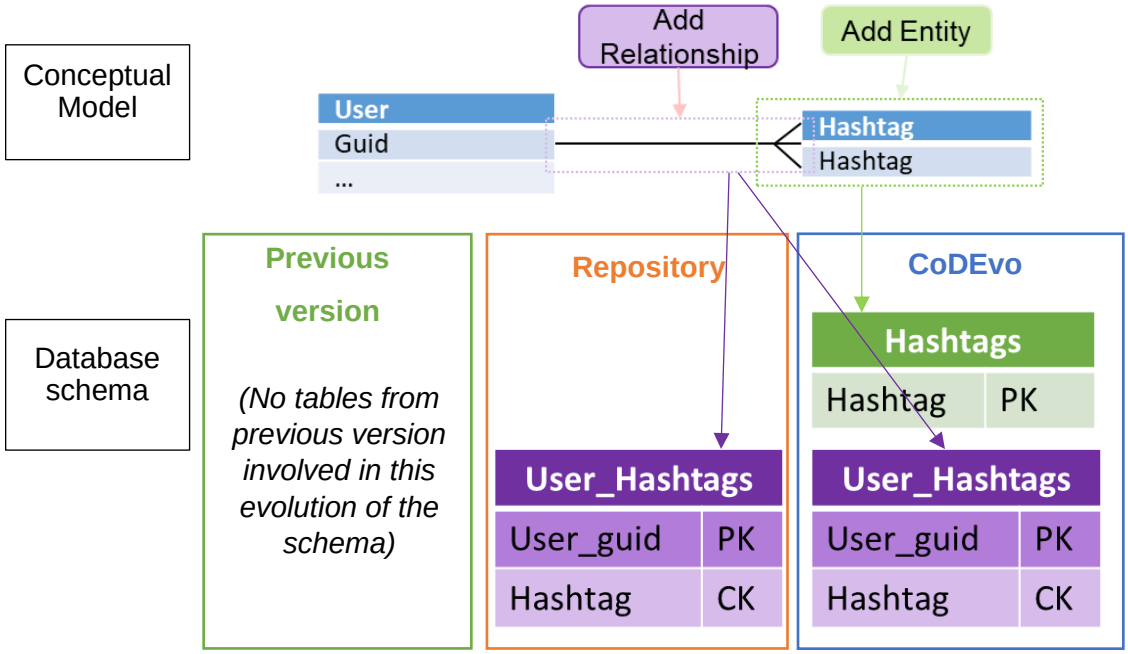


Figure 18 CoDEvo schema vs repository schema version for the insertion of an entity and a relationship

V.4.1.1.2 Analysis of Type III schema differences

There are three variants of Type III results: two for the addition of a new attribute that depends on the data type of the attribute: a non-Boolean attribute (Type III-A) or a Boolean attribute (Type III-B). The third variant is the addition of a new relationship (Type III-C). Table 15 displays the comparison of the number of these conceptual model changes that were detected in Type III results against Type I and Type II results.

Conceptual Model Changes	Type III	Type I or Type II	All
AddAttribute	9	95	104
AddRelationship	3	38	41
TOTAL Changes	12	133	145
Percentage	8.28	91.72	100

Table 15. Type III vs Type I or Type II for the same conceptual model change

It is important to note that in most additions of attributes or relationships the CoDEvo schema either was the same as the repository schema or contained more database structures. Only in **8.28%** of these changes in the conceptual model both schemas were different. In the following subsections we describe these differences depending on the Type III variant.

V.4.1.1.3 Analysis of Type III-A and Type III-B schema differences

Type III-A and III-B results were detected when adding a new attribute. In them, CoDEvo added a column mapped to the new attributes to the tables that store information about the entity of the attribute. However, the repository schema evolved by creating a new table to store this new attribute. There are two variants of this new table depending on the data type of the attribute: a non-Boolean attribute (Type III-A) or a Boolean attribute (Type III-B).

V.4.1.1.3.1 Type III-A schema differences: New non-Boolean attribute

Type III-A results were detected when, for the addition of a non-Boolean attribute, the repository schema contained a table with the following columns: 1) key columns mapped to the primary key of the new attribute's entity and 2) a non-key column mapped to this attribute. Figure 19 depicts the differences of the schemas of CoDEvo and the repository from a version

of the project 'Minds' where the attributes "last_retry" and "retries" were added to the entity "Entities". To address this addition, the project developers added to the repository schema the table "search_dispatcher_queue", which contained column mapped to the primary key of entity "Entities" as well as two columns for the new attributes. On the other hand, in the CoDEvo schema two columns with the same as the new attributes were added to the table "entities".

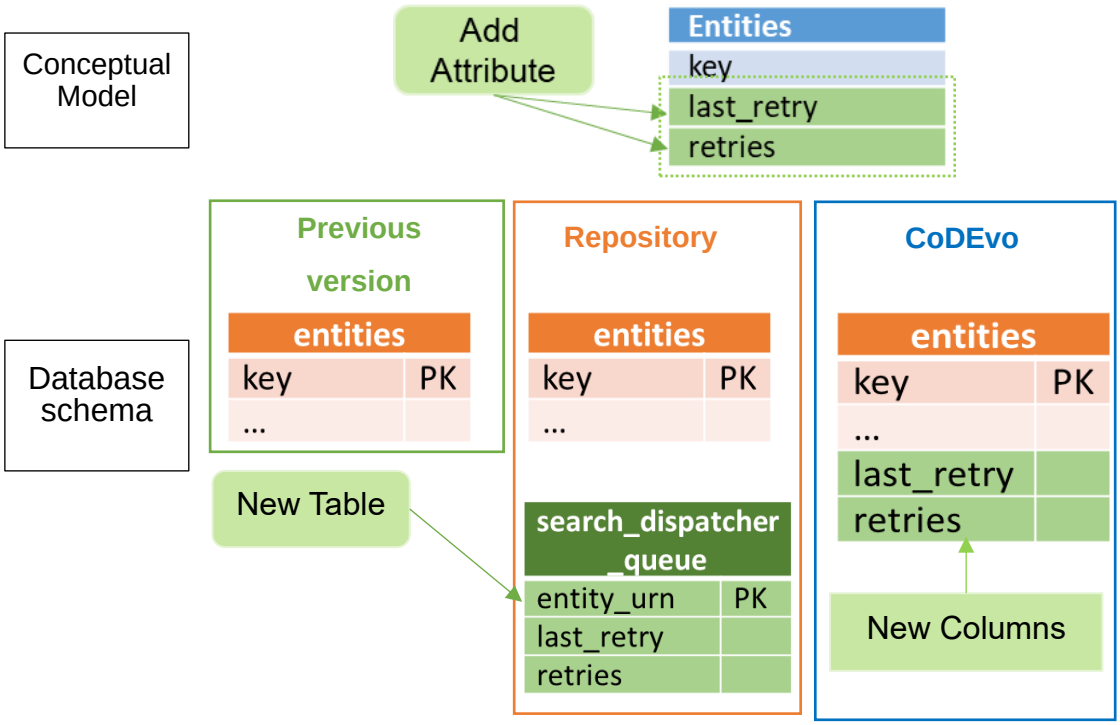


Figure 19 Comparison between the CoDEvo schema and the repository schema for the addition of two attributes "last_retry" and "retries" to entity 'Entities'

V.4.1.1.3.2 Type III-B schema differences: New Boolean attribute

Only one Type III-B result was detected, specifically in a version of the project WireApp, where the Boolean attribute "whitelist" was added to the entity 'Team'. Figure 20 depicts the differences between the schemas of CoDEvo and the repository in this version. In the repository schema, a table that only contains a column mapped to the primary key of entity 'Team' was added. If the key value of a 'Team' is added to the table, it means that 'whitelist' is True for that particular 'Team'. On the other hand, CoDEvo added a column named 'Whitelist' to the table 'team', which stores information about the entity of the same name, in order to store the value of 'Whitelist' alongside the rest of the information of a particular 'team'.

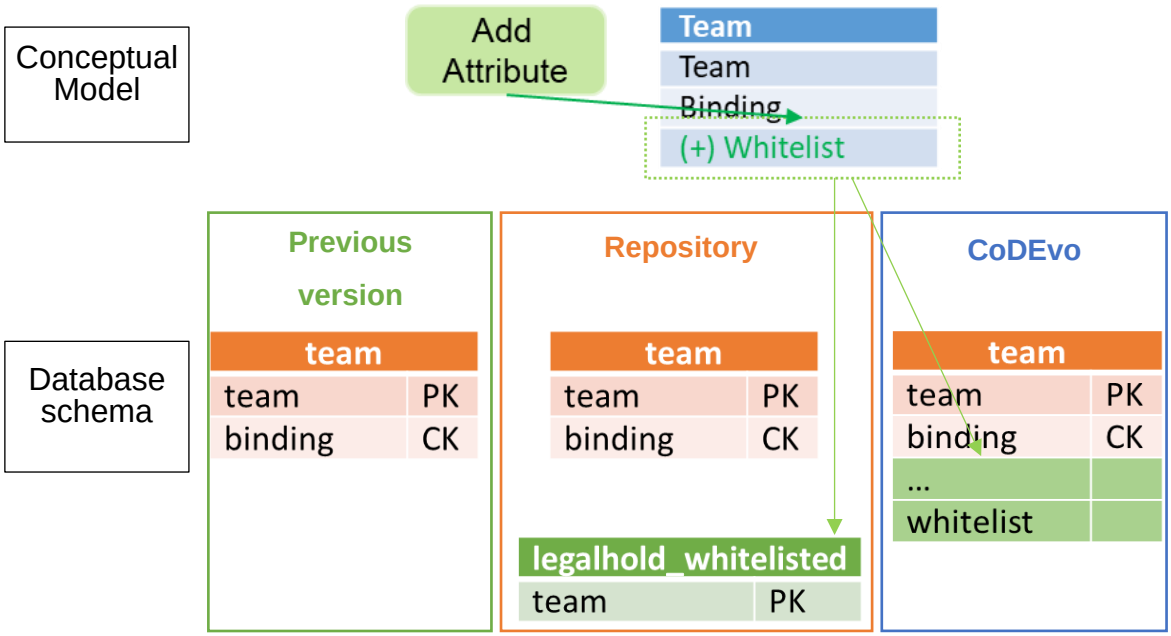


Figure 20 Comparison between the CoDEvo schema and repository schema from the project Wireapp for the addition of a Boolean attribute "whitelist" to entity Team

V.4.1.1.4 Analysis of Type III-C schema differences: 'New relationship'

For the addition of a new relationship, CoDEvo proposed the creation of a new table to store the information of the relationship. However, in two versions from project 'PowSybl' and one from project 'Minds' the repository schema evolved by altering a table that initially stored information of one of the entities. After it was altered, the table stored information of the relationship. This alteration consisted of additions to the table columns which were mapped to the primary key of the other entity of the relationship.

Figure 21 displays the differences between the schemas of CoDEvo and the repository from a version PowSybl for the addition of a relationship "one to many" between entities 'BusBreaker' and 'IccConverterStation'. CoDEvo created a new table ('Bus_IccConverterStation') to store information of this relationship. On the other hand, in the version the schema evolved by adding the columns "Bus" and "ConnectableBus" to the table "IccConverterStation".

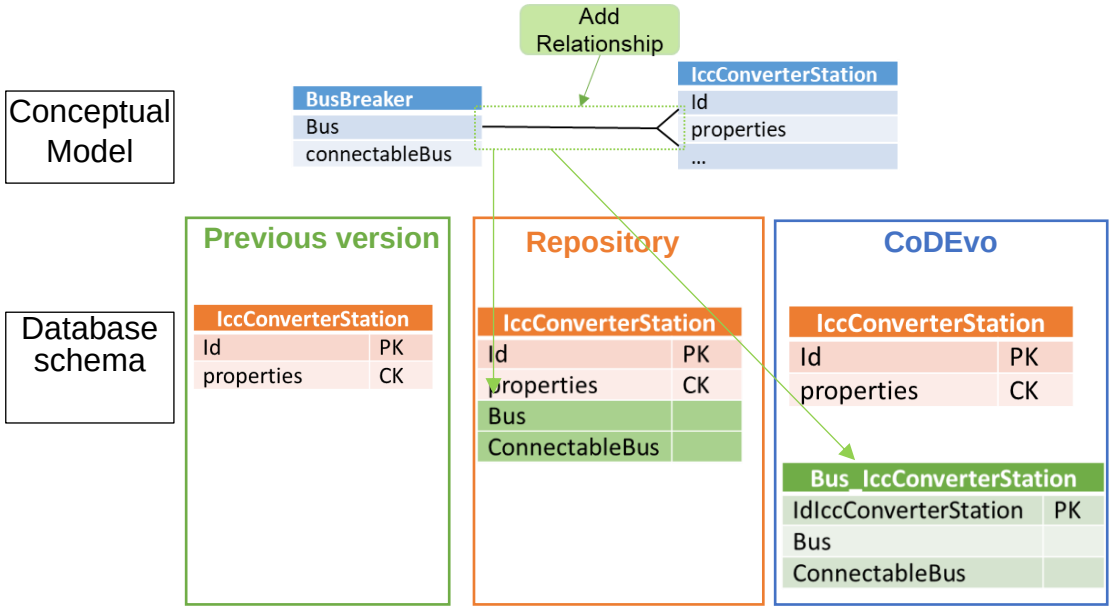


Figure 21 CoDEvo schema vs repository schema of Type III version from project PowSyBl for inserting a new relationship between entities BusBreaker and IccConverterStation

V.4.1.1.5 Conclusion for RQ6

From the results obtained by comparing the schemas of the repository and CoDEvo for each version (see Table 14), we were able to determine that 69.9% were the same schema (Type I). Another 17.2% of the total were Type II results, where CoDEvo added more database structures in addition to the ones that were in the repository schema. Altogether, Type I and Type II results were 87.1% of the total of versions. This shows how CoDEvo was able to generate for these versions the same database structures as the human developers.

The remaining results where the modifications of the schema were different (12.9%) were classified as Type III. In RQ7, we analyse if the schemas of CoDEvo that differ from the repository one are still valid regarding the project requirements, and, in that case, whether they are better or worse than the repository schema.

V.4.1.2 RQ7: Are the CoDEvo schemas valid considering the project requirements?

In this section we discuss at what extent the differences of database structures between the schema of CoDEvo and the repository affect the following:

1. Fulfilling of the project query requirements: We determine if the CoDEvo schemas fulfil the projects requirements of a version when the CoDEvo schema allows to query same data that can be queried from the repository schema. This is done by determining the possible ways of querying data from the repository schema and checking if the CoDEvo schema allows them
2. Maintenance of Logical Consistency: We compare the database operations (INSERT, DELETE, UPDATE) that developers must implement in the client application to maintain the logical consistency of the database when the data stored in the database is modified.

V.4.1.2.1 Differences regarding requirements of the queries

In this section we determine if the CoDEvo schema satisfies query requirements of a project version when it differs from the repository schema. In the comparison performed for RQ1, Type

I results were obtained when both schemas were the same, not requiring to be analysed. Type II results meant that the database structures from the repository were also in the CoDEvo schema, therefore CoDEvo was able to satisfy the requirements as well and they do not require to be analysed as well. However, Type III results showed that the schemas of CoDEvo and the repository contained different database structures. In the following paragraphs we analyse if in each of the Type III subtypes the schema of CoDEvo still satisfies the project requirements.

Type III-A and Type III-B results are obtained when adding a new attribute. In both subtypes, CoDEvo added a new column to a table that stored information of the attribute's entity, while in the repository schema a new table was created. A new column for storing data of the attribute fulfils the requirements of considering the new attribute, as it associates this data to the instance of the entity that it belongs. On the other hand, if a table is designed by the developers with a different primary key in order to query data in a different way, the CoDEvo schema would not allow the same queries as the repository schema. However, this scenario was not detected in any of the researched projects.

Type III-C results were obtained for some of the versions that added a new relationship. CoDEvo added a new table to store information of the new relationship. Note that this option was also chosen by the repository schema in 38 out of the 41 new relationships inserted in the projects. In the other 3 versions where the schema from CoDEvo and the repository are different, the schema provided by CoDEvo did not have all the possible queries that could be executed against the repository schema. For instance, in the repository schema you could filter the data using values for columns "Id" and "properties", while in the CoDEvo schema only values for "Id" could be used, as "properties" was not in the new table.

In conclusion, CoDEvo schemas were able to satisfy the requirements regarding storage of the new attribute or relationship. The only scenario where CoDEvo might provide an unsatisfactory schema is when there is a specific requirement of how data must be queried, which CoDEvo does not consider at this moment. This can be fixed by allowing developers to incorporate to the CoDEvo process the requirements regarding querying of data. It is also noteworthy that in most versions the CoDEvo schema and the repository schema are the same. Of the remaining versions, only a minority show the CoDEvo schema as less suitable for possible queries than the repository schema.

V.4.1.2.2 Differences regarding data logical consistency maintenance

When there is an update of data implemented in a client application, it may require additional database statements to maintain the logical consistency in the database. These statements depend on the characteristics of the schema, such as the duplication of data among the table. In this section we describe the differences between the schemas of CoDEvo and the repository regarding the database statements that must be implemented in the client applications for updates of data. As Type I results show that both schemas are the same, we only analyse the differences when there was a Type II or Type III result.

V.4.1.2.2.1 Versions with Type II results

Type II results were detected for the addition of both an entity and a relationship in the same version. In these versions, the repository schema contained a table to store the relationship, while, in addition to this table, the CoDEvo schema also contained a table to only store data of the new entity. This additional table increases the development cost, as a new table must be considered when implementing changes in the data, making the CoDEvo schema more costly than the repository one. On the other hand, an additional table does not negatively affect the

execution time of data update operations, as only a few database statements are required for updating data in this additional table.

This means that the only cost increase of a new table for storing data of the entity is during the development of the application. On the other hand, this addition table has the advantage of expanding the ways of querying data in the database. As future work, for users who do not want this additional table, we could incorporate interactivity by asking if users prefer a table to store information of the entity or a table that stores the relationship.

V.4.1.2.2.2 Versions with Type III-A and Type III-B results

Type III results for adding an attribute showed that, while CoDEvo altered an existing table by adding a column for the new attribute, in the repository schema a new table was added. If we compare them regarding development cost, performance, and query possibilities we obtain the following:

Development cost: The CoDEvo schema is less costly than the ones from the repositories as it avoids the consideration of a new table. The only changes required in the client application for the CoDEvo schema are those of adding the new attributes in the existing queries of the new tables. On the other hand, for the repository schema all the code related to the new table must be created from scratch.

Performance: Both schemas are similar, although the CoDEvo schema has the better performance. The CoDEvo schema should be similar in performance to the previous version schema, as it only requires the consideration of new columns in data operations that already existed in the client application. On the other hand, the repository schema is marginally more time consuming due to requiring execution of new database statements for the new table

Regarding logical consistency maintenance the CoDEvo schema is better in both development cost and performance than the repository schemas. Developers would require less time to update the client application and this application would also have better performance when executing operations against the database.

V.4.1.2.2.3 Versions with Type III-C results

Type III-C results were detected when adding a new relationship, generating CoDEvo a new table, while the version altered an existing table. If we compare the development cost, performance, and query possibilities we obtain the following:

Development cost: The repository schema avoids the creation of a new table, which might make it less costly than the CoDEvo schema. However, in this case we also need to consider that in repository schema the purpose of the altered table is modified from querying information of one entity to querying information of a relationship. This means that all the code related to that table must be modified in order to include information of the other entity of the relationship. Therefore, the development costs for the repository schema might be higher than for the CoDEvo schema depending on the amount of code that needs to be modified.

Performance: The repository schema would have better performance due to not requiring a new table to consider when executing data operations. It is the opposite scenarios that was described in section V.4.1.2.2.2 where the CoDEvo schema had best performance for the same reasons.

V.4.2 Threats to validity

We evaluated CoDEvo with several case studies, comparing the database structures proposed by CoDEvo with the database structures that are actually implemented in the repository schemas. However, there are several threats to validity of this experimentation. In this subsection we discuss threats to external and internal validity.

Threats to external validity: Some of the case studies that we used for the experimentation were also used to define the transformation rules of CoDEvo. These projects were named in prior work where we proposed the main idea of CoDEvo [28]. This is a threat to the validity of the experimentation, as the CoDEvo schemas might always be the same as the repository schemas, artificially increasing the effectiveness of CoDEvo generating appropriate schemas artificially. To mitigate this threat, we added more projects to be used in the analysis: Sunbirds, Doudouchat and Sop. The results that we obtained in these three projects were successful, although we are searching for more projects to analyse in order to further mitigate this threat.

Threats to internal validity: The projects that we used for the experimentation do not provide an explicit conceptual model. We needed to infer the conceptual model from these projects by analyzing the database structures of the repository schema. The same was required to obtain the conceptual model changes, which were obtained by analyzing the evolution of the schema in each eversion. This manual determination is a threat as we could have defined these conceptual models and conceptual model changes to perfectly fit into the transformation rules defined in CoDEvo. In order to avoid this, we carefully analysed each version to obtain the equivalent conceptual model. Then, we compared one version with the previous one to obtain what was changed, identifying the conceptual model changes applied in the version. There are also techniques that can be used to obtain the conceptual model from a column-oriented database schema [17], [92]. Using these external techniques, we could also ensure that the conceptual models that we use in the experimentation are not biased towards the transformations defined in CoDEvo. Note that these techniques can obtain different conceptual models as they have different ways of obtaining them.

VI FINAL REMARKS

This thesis has two main research goals that address the following problems in column-oriented databases: maintain the logical consistency for modifications of data and maintain the inter-model consistency for the evolution of the database. The first research goal is approached from two different point of views: a preventive approach named MDICA, and a reactive approach named CONCODA. The second research goal is addressed through a MDE approach named CoDEvo.

In the following subsections we first detail the conclusions for each research goal along with the contributions achieved in each one and then we detail the general conclusion of the thesis. We finish with the description of the research lines to address in future work.

VI.1 CONCLUSIONS FOR EACH RESEARCH GOAL

In this section we describe the conclusions of the two research goals of the thesis. As each research goal is further divided in subgoals, we will detail the conclusion for all the bottom-level subgoal.

Research goal 1: Maintenance of the logical consistency

The first research goal was approached with two different approaches: a preventive one named MDICA to determine the database statements required to perform modification of data and a reactive one named CONCODA to determine if the execution of a set of database statements to perform a modification of data maintained the consistency. For each of these approaches we defined the consequent research goals whose conclusion we detail in the following paragraphs.

Research goal 1.1: MDICA

As mentioned before, we devised MDICA to prevent the creation of inconsistencies when performing modifications of data, which included insertions, deletions, and updates of tuples. MDICA. We were able to successfully achieve this research goal with MDICA, providing not only the database statements and data procedures but also advising of issues during the process, such as lack of values for an insertion in a key column, and about the validity of the schema tables and the tuples of the modification. This subgoal was further divided in three subgoals:

Research goal 1.1.1: Determine actions for the modification of data

For each type of modification of data (insertion, deletion, and update) we defined a general algorithm that determined and generated the database statements needed to perform a given modification of data. In addition to the general algorithm, we addressed particular scenarios in each type of modification of data to differentiate the modifications involving only an entity and the ones involving more than one entity through relationships. In this determination, we provided the contributions C1, C2 and C3.

Research goal 1.1.2: Automation of MDICA

MDICA was also automated in a tool named CONSISTE, addressing the research goal 1.2.2. With this tool we were able to automate the experimentation that was detailed in chapter III which evaluated MDICA. This tool provided the contribution C4.

Research goal 1.1.3: Evaluation of MDICA

For the evaluation of MDICA, we defined four research questions that were addressed in chapter III. The conclusions that we obtained answering these research questions were the following:

RQ1: We detected that it was not always possible to insert data due to two factors: the tuple to insert did not have values to key attributes or the tuple lacked values for key columns of tables where the data must be inserted. MDICA can help to detect these situations and provide the information required to fix the insertions so they can be performed against the database.

RQ2: The number of tables depends on the complexity of the tuple to insert. As more entities and relationships participate in the insertion, more tables are detected as target tables. Nevertheless, MDICA identified all target tables regardless of the complexity of the tuple.

RQ3: We detected that in addition to the expected INSERT statements to insert the data, it was also common to require SELECT statements for retrieving data required for the maintenance of the logical consistency. In addition to this, we also detected less frequent scenarios where we required to CREATE and populate new tables in order to properly query these data. Although this creation of table is only performed the first execution, as in following executions the table will already exist.

RQ4: We detected scenarios where although logical consistency is not ensured to be lost, it is in danger such a value of a tuple not being inserted in any table. To address these scenarios, MDICA identified them and provided developer warning messages that can fix these scenarios.

The evaluation provided the contribution C5.

Research goal 1.2: CONCODA

The objective of the research goal 1.2 was to detect if a column-oriented database maintains the logical consistency after one or more modifications of data have been performed against the database. To address this research goal, we devised a reactive approach named CONCODA, which used a relational database that implements the conceptual model as an oracle to determine the logical consistency maintenance.

CONCODA also provides the information of the table where inconsistencies were detected, in order to help developer to fix the mistakes in the database statements that modify data in these tables. Then, if inconsistencies are detected, MDICA can be used to fix them, which was also evaluated using this reactive approach, obtaining successful results.

CONCODA was also used to answer the following research question first defined in chapter III where MDICA was devised:

RQ5: We performed two experimentations using CONCODA by executing test cases where tuples were inserted using MDICA, verifying through CONCODA that MDICA provided database statements that maintained the logical consistency. There were test cases in which the insertion was blocked due to MDICA not being able to guarantee the logical consistency, which further demonstrated how MDICA prioritizes maintaining the logical consistency over inserting data.

CONCODA provided the contributions C6 and C7, while its application provided the contribution C8.

Research goal 2: Database evolution: Schema evolution

The second research goal aims to maintain the inter-model consistency when the database evolves. For this purpose, we devised the MDE approach CoDEvo that determines the action to perform in the schema when the conceptual model evolves to maintain the inter-model consistency (consistency between schema and conceptual model). We divided this research goals in the following subgoals:

Research goal 2.1: Schema evolution analysis in open-source projects

We researched in several open-source projects in order to address the research goal 2.1. We identified several changes in the conceptual model in these projects as well as how the schema evolved in accordance with these conceptual model changes. We noted that the most common changes were those regarding creations of entities, attributes, or relationships. Updates or deletions of other structures, although detected, were less common to happen.

The analysis of the open-source projects provided the contribution C10.

Research goal 2.2: Schema evolution to maintain inter-model consistency

Using the information obtained in the aforementioned analysis, we devised CoDEvo, a MDE approach that, through transformations, provide a model that contains the information of how the schema must evolve. In addition to models, we were also able to transform the actions specific in the model into the database statements required to execute these actions.

CoDEvo provided the contributions C11 and C12, while its automation in a tool provided the contribution C13. CoDEvo is also part of the framework for database evolution described in section V.2, whose definition provided contribution C9.

Research goal 2.3: CoDEvo experimentation and validation

The research goal 2.3 was addressed through an experimentation, using open-source projects, that was based on two research questions:

RQ6: 69.9% of the schemas obtained by CoDEvo were the same as the ones determined by the developers of the projects used in the experimentation. 17.2% of schema had additional database structures (tables or columns) than the schema from the developers. These additional structures were created by the approach to widen the possible queries that can be executed against the database. Their absence in the schema from the developers was caused to not being a requirement to execute these queries. In the remaining 12.9% schemas the database structures were different, which we analysed in the following research question.

RQ7: We studied if the schemas from CoDEvo were still valid to obtain the same data as in the schema from the developers. We were able to verify this in all scenarios from the experimentation, although we acknowledged potential scenarios related to particular requirements regarding queries where CoDEvo might provide a schema that is not adapted to these particular requirements.

The evaluation of CoDEvo provided the contribution C14.

VI.2 General Conclusions

The main objective of this thesis was to address problems that were specific to column-oriented databases and where the techniques or approaches for relational databases could not be applied. Two of these problems were the maintenance of the logical consistency for modifications of data and the maintenance of the inter-model consistency for schema evolution. In this thesis we proposed approaches for them, two for the maintenance of the logical consistency and one for the database evolution. We were able to accomplish the hypothesis of the thesis by defining these novel approaches that specifically address these problems for column-oriented databases, validating them with real open-source projects.

These approaches can help developers to avoid mistakes when evolving the database or when implementing the database statements and procedures for modifications of data. For instance, MDICA can be used to generate the database statements to include in the application source code that are required for performing a modification of data. This helps developers to not forget target table, retrieve the adequate data from other tables to complete the database statements, and avoid making mistakes that endanger the logical consistency. For previously developed applications, CONCODA may be applied to determine if any client application maintains the logical consistency when performing modifications of data. It is important to note, that both MDICA and CONCODA.

Regarding the evolution of the database CoDEvo helps developers to avoid mistakes when evolving the database as the requirements change. It guarantees the inter-model consistency, which assures that they database will not allow to store data inconsistent with the conceptual view of the system. We have obtained through experimentation that CoDEvo is able to obtain the same schema as human developers in 87% of the schema evolution cases that were part of the experimentation performed for the validation. In the remaining 13% schemas, although different, the CoDEvo schemas was still valid to satisfy the requirements of the projects used in the experimentation.

Nevertheless, there are still other problems that have been not addressed in this thesis, which we intent to approach in the future and are described in the next subsection.

VI.3 Future Work

In this subsection we briefly describe the different problems that we want to address in the future, starting from the advances obtained in this thesis.

VI.3.1 Database evolution: Maintenance of the logical consistency after the evolution of the schema

In section V.2 we described that the logical consistency may be affected when the schema evolves. We have so far identified two causes:

1. Creation of new database structures (tables or columns) that must store data that is already stored in other databases.
2. The conceptual model now specifies constraints that contradict some of the data that was previously stored in the database, such as the change of the cardinality of a relationship.

We plan to approach this problem in the following works by continuing to use a MDE approach that will create models that specify the data to migrate, identifying the source tables and the target tables. As the schema evolution problem addressed in this thesis also uses a MDE approach, we plan to join both approaches together to achieve an automation of the process that covers both the evolution of the schema addressed in this thesis with the maintenance of the logical consistency.

VI.3.2 Database evolution: Program update after the evolution of the schema

The client applications contain code and embedded database statements for a specific version of a database schema. If the database schema changes, the client application must be changed as well as we described in section V.2. We propose to automatically update the application using the information provided by the models obtained through CoDEvo. We plan to do this through a program-repair approach that does the following:

1. Identify the code related to the schema, both embedded statements and source code that depends on the schema such as classes for the conceptual model entities.
2. Update the identified code considering the information provided by the models obtained from CoDEvo.

After we finish this program-repair approach, we will have developed all the necessary approaches for database evolution when there is a change of the requirements that change the conceptual model (components or constraints).

VI.3.3 Database evolution: Changes directly applied to the schema

In the research line 3 we have started from new requirements that change the conceptual model. However, there can also be new requirements that directly change the schema as we described in [29] and in section V.2 for the general framework for database evolution. We plan to propose a MDE approach similar to CoDEvo, which given a change in the schema, determines how the conceptual model must evolve in order to guarantee the inter-model consistency.

Afterwards, CoDEvo will be applied in order to determine the rest of changes to apply in the schema if they were needed to maintain the inter-model consistency.

VI.3.4 Modifications of data directly to the database side

Regarding the problem of maintaining the logical consistency when there are modifications of data which was addressed in the research lines 1 and 2, we plan to extend it to also cover modifications of data that are directly executed to the database.

Given a modification of data in the schema (insertion, update or deletion of a row), we plan to identify determine the modifications of data in the conceptual model (insertion, update or deletion of tuples) equivalent to the given modification of data in the schema. Afterwards, we will apply MDICA in order to maintain the logical consistency by performing the modification of data in the rest of the database.

CONCLUSIONES

El principal objetivo de esta tesis era abordar problemas específicos de las bases de datos orientadas a columnas en los que no se podían aplicar las técnicas o enfoques destinados a las bases de datos relacionales. Dos de estos problemas eran el mantenimiento de la consistencia lógica para las modificaciones de los datos y el mantenimiento de la consistencia inter-modelo durante la evolución de la base de datos ante un cambio de los requisitos. En esta tesis hemos propuesto enfoques para abordar ambos problemas, MDICA y CONCOSA para abordar el mantenimiento de la consistencia lógica y CoDEvo para la evolución de la base de datos. Hemos podido cumplir la hipótesis de la tesis definiendo a través de estos enfoques específicamente diseñados para las bases de datos orientadas a columnas. A su vez, han sido validados utilizando proyectos de software obtenidos de artículos de investigación y repositorios públicos.

Estos enfoques pueden ayudar a los desarrolladores a evitar errores tanto en la evolución de la base de datos como en la implementan de las operaciones y procedimientos de la base de datos necesarios para ejecutar modificaciones de datos. Específicamente, MDICA puede utilizarse para generar las sentencias de la base de datos que deben incluirse en el código fuente de la aplicación y que son necesarias para realizar una modificación de los datos. Esto ayuda a los desarrolladores a evitar errores comunes en modelos de datos desnormalizados como olvidar tablas sobre las que se deben ejecutar las modificaciones de datos y otros errores comunes que pongan en peligro la consistencia lógica de los datos. En el caso de las aplicaciones desarrolladas previamente, CONCODA sirve para determinar si la aplicación cliente mantiene la consistencia lógica al realizar una determinada modificación de datos.

En cuanto a la evolución de la base de datos, CoDEvo ayuda a los desarrolladores a evitar errores al evolucionar la base de datos a medida que cambian los requisitos. Al garantizar la consistencia inter-modelo, se asegura que la base de datos no permitirá almacenar datos inconsistentes con la visión conceptual del sistema. A través de la experimentación con proyectos software de open-source en los que ha existido una evolución del esquema, hemos obtenido que CoDEvo es capaz de obtener el mismo esquema que los desarrolladores en el 87% de los casos de evolución del esquema de estos proyectos. En el 13% de casos restante, los esquemas proporcionados por CoDEvo, aunque diferentes, seguían siendo válidos para satisfacer los requisitos de los proyectos utilizados en la experimentación.

APPENDICES

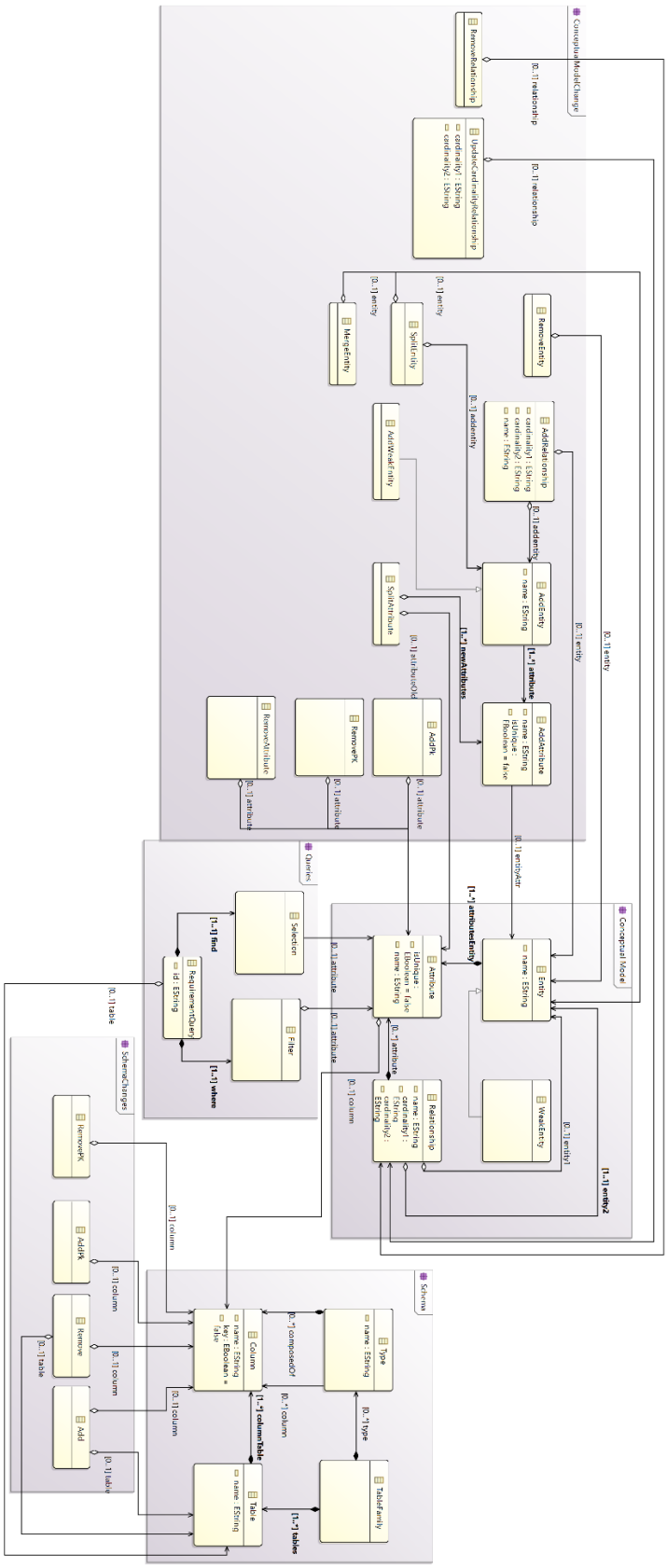


Figure 22 Metamodel union for CoDEvo

REFERENCES

[1] R. Elmasri, S. B. Navathe, R. Elmasri, and S. B. Navathe, *Fundamentals of Database Systems*. Springer, 2000.

[2] DB Engines, "DB-Engines Ranking," *https://db-engines.com/en/ranking*, 2022. https://db-engines.com/en/ranking (accessed Jun. 17, 2022).

[3] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *2011 6th international conference on pervasive computing and applications*, 2011, pp. 363–366.

[4] Y. Zhang, J. Ren, J. Liu, C. Xu, H. Guo, and Y. Liu, "A survey on emerging computing paradigms for big data," *Chinese Journal of Electronics*, vol. 26, no. 1, pp. 1–12, 2017.

[5] T. N. Khasawneh, M. H. AL-Sahlee, and A. A. Safia, "SQL, NewSQL, and NoSQL databases: A comparative survey," in *2020 11th International Conference on Information and Communication Systems (ICICS)*, 2020, pp. 13–21.

[6] R. Cattell, "Scalable SQL and NoSQL data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[7] V. N. Gudivada, D. Rao, and V. v. Raghavan, "NoSQL Systems for Big Data Management," in *2014 IEEE World Congress on Services*, Jun. 2014, pp. 190–197. doi: 10.1109/SERVICES.2014.42.

[8] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?," *Computer (Long Beach Calif)*, vol. 43, no. 2, pp. 12–14, Feb. 2010, doi: 10.1109/MC.2010.58.

[9] A. Makris, K. Tserpes, V. Andronikou, and D. Anagnostopoulos, "A Classification of NoSQL Data Stores Based on Key Design Characteristics," *Procedia Comput Sci*, vol. 97, pp. 94–103, 2016, doi: 10.1016/j.procs.2016.08.284.

[10] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[11] C. J. M. Tauro, S. Aravindh, and A. B. Shreeharsha, "Comparative study of the new generation, agile, scalable, high performance NoSQL databases," *Int J Comput Appl*, vol. 48, no. 20, pp. 1–4, 2012.

[12] Instaclustr, "Guide to Apache Cassandra Data Modelling," Feb. 14, 2022. https://www.instaclustr.com/resource/6-step-guide-to-apache-cassandra-data-modelling-white-paper (accessed Jul. 17, 2022).

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

[14] M. T. González-Aparicio, M. Younas, J. Tuya, and R. Casado, "Testing of transactional services in NoSQL key-value databases," *Future Generation Computer Systems*, vol. 80, pp. 384–399, Mar. 2018, doi: 10.1016/j.future.2017.07.004.

[15] D. Pritchett, “BASE: An Acid Alternative,” *Queue*, vol. 6, no. 3, pp. 48–55, May 2008, doi: 10.1145/1394127.1394128.

[16] A. Foundation, “Apache Cassandra.” 2008.

[17] M. J. Mior and K. Salem, “Renormalization of NoSQL database schemas,” in *International Conference on Conceptual Modeling*, 2018, pp. 479–487.

[18] A. Chebotko, A. Kashlev, and S. Lu, “A big data modeling methodology for Apache Cassandra,” in *2015 IEEE International Congress on Big Data*, 2015, pp. 238–245.

[19] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla, and P. Sánchez, “Mortadelo: Automatic generation of NoSQL stores from platform-independent data models,” *Future Generation Computer Systems*, vol. 105, pp. 455–474, 2020.

[20] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, “Action research,” *Commun. ACM*, vol. 42, no. 1, pp. 94–97, 1999, doi: 10.1145/291469.291479.

[21] P. Suárez-Otero, “Mantenimiento de la consistencia lógica en bases de datos NoSQL Casssandra ante modificaciones de datos a nivel de modelo conceptual de datos,” University of Oviedo, Gijón, 2017.

[22] P. Suárez-Otero, J. Gutierrez, C. de la Riva, and J. Tuya, “Mantenimiento de la Consistencia Lógica en Cassandra,” 2017.

[23] P. Suárez-Otero González, M. J. Suárez Cabal, P. J. Tuya González, “Evaluación del mantenimiento de la consistencia lógica en Cassandra,” *Actas de las XXIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2018)*, 2018.

[24] P. Suárez-Otero González, M. J. Suárez Cabal, P. J. Tuya González, “Leveraging conceptual data models for keeping cassandra database integrity,” 2018.

[25] P. Suárez-Otero, M. J. Suárez-Cabal, and J. Tuya, “Leveraging conceptual data models to ensure the integrity of Cassandra databases,” *Journal of Web Engineering*, 2019.

[26] M. J. Suárez-Cabal, P. Suárez-Otero, C. de la Riva, and J. Tuya, “MDICA: Maintenance of data integrity in column-oriented database applications,” *Comput Stand Interfaces*, vol. 83, p. 103642, 2023.

[27] P. Suárez-Otero González, M. J. Suárez Cabal, P. J. Tuya González, “Verificación del mantenimiento de la consistencia lógica en bases de datos Cassandra,” *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)(24ª. 2018. Cáceres)*, 2019.

[28] P. Suárez-Otero, M. J. Mior, M. J. Suárez-Cabal, and J. Tuya, “Maintaining NoSQL database quality during conceptual model evolution,” in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2043–2048.

[29] P. Suárez-Otero, M. J. Mior, M. J. Suárez-Cabal, and J. Tuya, “An Integrated Approach for Column-Oriented Database Application Evolution Using Conceptual Models,” in *International Conference on Conceptual Modeling*, 2021, pp. 26–32.

[30] P. Suárez-Otero González, M. J. Mior, M. J. Suárez Cabal, P. J. Tuya González “Evolución en sistemas de bases de datos orientadas a columnas ante cambios

conceptuales," *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*, 2021.

[31] P. Suárez-Otero, M. J. Mior, M. J. Suárez-Cabal, and J. Tuya, "CoDEvo: Column family database evolution using model transformations,. Sent to Journal of Systems and Software, 2022.

[32] P. Suárez-Otero, "Analysis of the Logical Consistency in Cassandra," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 430–431.

[33] P. Suárez-Otero, "Mantenimiento de la consistencia lógica en Cassandra," 2019.

[34] P. Suárez-Otero, "Towards data integrity in Cassandra database applications using conceptual models," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 246–249.

[35] P. Suárez-Otero, *Sistemas de almacenamiento y gestión Big Data*. Valencian International University (VIU), 2021.

[36] https://www.thinkautomation.com/histories/the-history-of-databases/, "The History of databases." (accessed Jul. 17, 2022).

[37] D. Maier, *The theory of relational databases*, vol. 11. Computer science press Rockville, 1983.

[38] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks CACM, 13 (6)." June, 1970.

[39] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976, doi: 10.1145/320434.320440.

[40] C. Bontempo and G. Zagelow, "The IBM Data Warehouse Architecture," *Commun. ACM*, vol. 41, no. 9, pp. 38–48, Sep. 1998, doi: 10.1145/285070.285078.

[41] M. Velicanu, G. Matei, and others, "Database Vs Data Warehouse," *Revista Informatica Economică*, vol. 3, no. 2007, p. 43, 2007.

[42] M. Klettke and U. Störl, "Four Generations in Data Engineering for Data Science," *Datenbank-Spektrum*, vol. 22, no. 1, pp. 59–66, 2022, doi: 10.1007/s13222-021-00399-3.

[43] B. Medjahed, M. Ouzzani, and A. Elmagarmid, "Generalization of ACID properties," 2009.

[44] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer (Long Beach Calif)*, vol. 45, no. 2, pp. 30–36, 2012, doi: 10.1109/MC.2011.389.

[45] J. Bhogal and I. Choksi, "Handling big data using NoSQL," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, 2015, pp. 393–398.

[46] S. Bjeladinovic, Z. Marjanovic, and S. Babarogic, “A proposal of architecture for integration and uniform use of hybrid SQL/NoSQL database components,” *Journal of Systems and Software*, vol. 168, p. 110633, 2020, doi: https://doi.org/10.1016/j.jss.2020.110633.

[47] A. Woodie, “Neo4j Going Distributed with Graph Database,” 2020. https://www.datanami.com/2020/02/04/neo4j-going-distributed-with-graph-database/ (accessed Aug. 30, 2022).

[48] Neo4J, “Neo4j Graph Database,” 2022. https://neo4j.com/product/neo4j-graph-database/ (accessed Aug. 30, 2022).

[49] C. J. Date, *A Guide to the SQL Standard (2nd Ed.)*. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[50] Neo4J, “Cypher Query Language,” 2022. https://neo4j.com/developer/cypher/ (accessed Jul. 19, 2022).

[51] Redis, “RediSearch.” https://redis.io/docs/stack/search/ (accessed Jul. 19, 2022).

[52] Stitchdata, “Columnar database: a smart choice for data warehouses,” *Columnar database: a smart choice for data warehouses*. https://www.stitchdata.com/columnardatabase/ (accessed Jul. 03, 2022).

[53] Bekker; Alex, “ Cassandra vs. HBase: twins or just strangers with similar looks?,” Jun. 19, 2018. https://www.scnsoft.com/blog/cassandra-vs-hbase (accessed Jul. 19, 2022).

[54] A. Lakshman and P. Malik, “Cassandra,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010, doi: 10.1145/1773912.1773922.

[55] Apache, “Cassandra Basics.” https://cassandra.apache.org/_/cassandra-basics.html (accessed Jul. 19, 2022).

[56] L. Westoby, “How Apache Cassandra™ Balances Consistency, Availability, and Performance,” 2019. https://www.datastax.com/blog/how-apache-cassandratm-balances-consistency-availability-and-performance (accessed Jul. 03, 2022).

[57] Apache, “Data Definition.” https://cassandra.apache.org/doc/latest/cassandra/cql/ddl.html (accessed Jul. 19, 2022).

[58] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong, “Understanding the causes of consistency anomalies in Apache Cassandra,” *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 810–813, Feb. 2015, doi: 10.14778/2752939.2752949.

[59] T. Hobbs, “Basic rules of Cassandra data modeling,” *URL https://www. datastax. com/dev/blog/basic-rules-of-cassandra-data-modeling*, 2016.

[60] D. C. Schmidt, “Guest Editor’s Introduction: Model-Driven Engineering,” *Computer (Long Beach Calif)*, vol. 39, no. 2, pp. 25–31, 2006, doi: 10.1109/MC.2006.58.

[61] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, “Survey and classification of model transformation tools,” *Softw Syst Model*, vol. 18, no. 4, pp. 2361–2397, 2019.

[62] A. Kleppe, J. Warmer, W. Bast, and E. MDA, “The model driven architecture: practice and promise.” Addison-Wesley Boston, 2003.

[63] T. Mens and P. van Gorp, “A taxonomy of model transformation,” *Electron Notes Theor Comput Sci*, vol. 152, pp. 125–142, 2006.

[64] O. M. G. MDA, “Object Management Group Model Driven Architecture.” 2008.

[65] Datastax, “Using materialized views,” *https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useOverviewMV.html*, Apr. 18, 2022.

[66] Datastax, “Use Materialized Views,” 2015. ref https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useCreateMV.html]: (accessed May 25, 2022).

[67] C. Peter, “Supporting the Join Operation in a NoSQL System-Mastering the internals of Cassandra,” NTNU, 2015.

[68] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu, “NoSE: Schema design for NoSQL applications,” *IEEE Trans Knowl Data Eng*, vol. 29, no. 10, pp. 2275–2289, 2017.

[69] D. Sevilla Ruiz, S. F. Morales, and J. García Molina, “Inferring versioned schemas from NoSQL databases and its applications,” in *International Conference on Conceptual Modeling*, 2015, pp. 467–480.

[70] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, “Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++,” *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 117–128, 2010.

[71] P. Vassiliadis, M.-R. Kolozoff, M. Zerva, and A. v Zarras, “Schema evolution and foreign keys: a study on usage, heartbeat of change and relationship of foreign keys to table activity,” *Computing*, vol. 101, no. 10, pp. 1431–1456, 2019.

[72] J. Delplanque, A. Etien, N. Anquetil, and S. Ducasse, “Recommendations for evolving relational databases,” in *International Conference on Advanced Information Systems Engineering*, 2020, pp. 498–514.

[73] V. Sugumaran and V. C. Storey, “The role of domain ontologies in database design: An ontology management and conceptual modeling environment,” *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 3, pp. 1064–1094, 2006.

[74] N. F. Noy and M. Klein, “Ontology evolution: Not the same as schema evolution,” *Knowl Inf Syst*, vol. 6, no. 4, pp. 428–440, 2004.

[75] S. Scherzinger, M. Klettke, and U. Störl, “Managing schema evolution in NoSQL data stores,” *arXiv preprint arXiv:1308.0514*, 2013.

[76] M. L. Möller, M. Klettke, and U. Störl, “EvoBench–A Framework for Benchmarking Schema Evolution in NoSQL,” in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 1974–1984.

[77] A. Hillenbrand, U. Störl, S. Nabiyev, and M. Klettke, “Self-adapting data migration in the context of schema evolution in NoSQL databases,” *Distrib Parallel Databases*, pp. 1–21, 2021.

[78] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, and M. Klettke, “MigCast: putting a price tag on data model evolution in NoSQL data stores,” in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1925–1928.

[79] U. Störl *et al.*, “Curating variational data in application development,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1605–1608.

[80] C. J. F. Candel, D. S. Ruiz, and J. J. García-Molina, “A unified metamodel for nosql and relational databases,” *Inf Syst*, vol. 104, p. 101898, 2022.

[81] A. H. Chillón, D. S. Ruiz, and J. G. Molina, “Towards a taxonomy of schema changes for NoSQL databases: the Orion language,” in *International Conference on Conceptual Modeling*, 2021, pp. 176–185.

[82] R. Thottuvaikkatumana, *Cassandra Design Patterns*. Packt Publishing Ltd, 2015.

[83] Datastax, “The Playlist tutorial,” *Datastax*, Apr. 18, 2022.

[84] V. Reniers, D. van Landuyt, A. Rafique, and W. Joosen, “On the State of NoSQL Benchmarks,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, Apr. 2017, pp. 107–112. doi: 10.1145/3053600.3053622.

[85] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*, 2010, p. 143. doi: 10.1145/1807128.1807152.

[86] A. Hendawi *et al.*, “Distributed NoSQL Data Stores: Performance Analysis and a Case Study,” in *2018 IEEE International Conference on Big Data (Big Data)*, Dec. 2018, pp. 1937–1944. doi: 10.1109/BigData.2018.8622544.

[87] P. Martins, P. Tomé, C. Wanzeller, F. Sá, and M. Abbasi, “NoSQL Comparative Performance Study,” 2021, pp. 428–438. doi: 10.1007/978-3-030-72651-5_41.

[88] M. Grochtmann and K. Grimm, “Classification trees for partition testing,” *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, Jun. 1993, doi: 10.1002/stvr.4370030203.

[89] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, “Data-guided repair of selection statements,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 243–253.

[90] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, “Model transformations? transformation models!,” in *International Conference on Model Driven Engineering Languages and Systems*, 2006, pp. 440–453.

[91] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, “On the use of higher-order model transformations,” in *European Conference on Model Driven Architecture-Foundations and Applications*, 2009, pp. 18–33.

[92] D. Sevilla Ruiz, S. F. Morales, and J. García Molina, “Inferring versioned schemas from NoSQL databases and its applications,” in *International Conference on Conceptual Modeling*, 2015, pp. 467–480.