Leveraging conceptual data models to ensure the integrity of Cassandra databases

Pablo Suárez-Otero, María José Suárez-Cabal, Javier Tuya

Computer Science Department, University of Oviedo, Campus de Viesques, Gijón

Email {suarezgpablo, cabal, tuya}@uniovi.es

Abstract

The use of NoSQL databases for cloud environments has been increasing due to their performance advantages when working with big data. One of the most popular NoSQL databases used for cloud services is Cassandra, in which each table is created to satisfy one query. This means that as the same data could be retrieved by several queries, these data may be repeated in several different tables. The integrity of these data must be maintained in the application that works with the database, instead of in the database itself as in relational databases. In this paper, we propose a method to ensure the data integrity when there is a modification of data by using a conceptual model that is directly connected to the logical model that represents the Cassandra tables. This method identifies which tables are affected by the modification of the data and also proposes how the data integrity of the database may be ensured. We detail the process of this method along with two examples where we apply it in two insertions of tuples in a conceptual model. We also apply this method to a case study where we insert several tuples in the conceptual model, and then we discuss the results. We have observed how in most cases several insertions are needed to ensure the data integrity as well as needing to look for values in the database in order to do it.

Keywords. NoSQL, Cloud, Conceptual Model, Logical Model, Cassandra, Logical Data Integrity

1. INTRODUCTION

The importance of NoSQL databases has been increasing due to the advantages they provide in the processing of big data [1]. These databases were created to have a better performance than relational databases [2] in operations such as reading and writing [3] when managing large amounts of data. This improved performance has been attributed to the abandonment of ACID constraints [4]. NoSQL databases have been classified in four types depending on how they store the information: [5]: those based on key-values like Dynamo where the items are stored as an attribute name (key) and its value; those based on documents like MongoDB where each item is a pair of a key and a document; those based on graphs like Neo4J that store information about networks, and those based on columns like Cassandra that store data as columns.

Internet companies make extensive use of these databases due to benefits such as horizontal scaling and having more control over availability [6]. Companies such as Amazon, Google or Facebook use the web as a large, distributed data repository that is managed with NoSQL databases [7]. These databases solve the problem of scaling the systems by implementing them in a distributed system, which is difficult using relational databases. Examples of companies that are using these NoSQL for their web services are Telefonica, Facebook or EA in the case of MongoDB [8] and Netflix, eBay or Sony in the case of Cassandra [9].

Cassandra is a distributed database developed by the Apache Software Foundation [10]. Its characteristics are [11]: 1) a very flexible scheme where it is very convenient to add or delete fields; 2) high scalability, so the failure of a single element of the cluster does not affect the whole cluster; 3) a query-driven approach in which the data is organized based on queries. This last characteristic means that, in general, each Cassandra table is designed to satisfy a single query [12]. If a single datum is retrieved by more than one query, the tables that satisfy these queries will store this same datum. Therefore, the Cassandra data model is a denormalized model, unlike in relational databases where it is usually normalized. The integrity of the information repeated among several tables of the database is called logical data integrity.

Cassandra does not have mechanisms to ensure the logical data integrity in the database, unlike relational databases, so it needs to be maintained in the client application that works with the database [13]. This is prone to mistakes that could incur in the creation of inconsistencies of the data. Traditionally, cloud-based systems have used normalized relational databases in order to avoid situations that can lead to anomalies of the data in the system [18]. However, the performance problems of these relational databases when working with big data have made them unfit in these situations, so NoSQL systems are used although they face another problem, that of ensuring the logical data integrity [6].

To illustrate this problem, consider a Cassandra database that stores data relating to authors and their books. This database has two tables, one created to satisfy the query "books that a given author has written" (Books_by_author) and another created to satisfy the query "find information of a book giving its identifier" (Books). Note that the information pertaining to a specific book is repeated in both tables. Suppose that during the development of a function to insert books in the database, the developer forgets to introduce a database statement to insert the information of the new book in table "Books_by_author". This produces an inconsistency of the data, as the inserted book would only be in table "Books" and not in table "Books_by_author", although both tables store the Id of books. This example is illustrated in Figure 1. Partition key columns are labelled 'K' and the clustering key columns are labelled 'C' [24]. These columns compound the primary key of a Cassandra table:

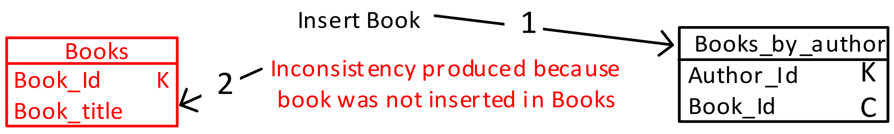


Figure 1 Logical integrity broken

We have identified two types of modifications that may produce an inconsistency:

- *Modifications of the logical model*: when there is a modification regarding the tables, such as the creation of a new table or the addition of columns to an existing table. Data integrity can be broken as the new columns could store data that may already be stored in other tables of the database. These data must be inserted in the new columns in order to maintain the logical integrity of the data.

- *Modification of data:* we define a modification of data as the change of the values (insertion, update or deletion) stored in a row of the logical model or the change of the values assigned to attributes in a tuple from the conceptual model. After a modification of data in a table, an inconsistency is produced if the modified data has functional dependencies with other data stored in other tables and they are not updated. This type of modification is shown in Figure 1.

As the number of tables with repeated data in a database increases, so too does the difficulty of maintaining the data integrity. In this work we introduce an approach for the maintenance of the data integrity when there are modifications of data. This article is an extension of earlier work [14] incorporating more detail of the top-down use case, a new casuistic for this case where it is necessary to extract values from the database and a detailed description of the experimentation carried out. The contributions of this paper are the following:

1. A method that automatically identifies the tables that need maintenance of the data integrity.
2. A method to automatically generate a set of Cassandra Query Language (CQL) statements [15] to ensure the data integrity in the identified tables.
3. An evaluation in a case study of the proposed method inserting tuples in the conceptual model.

This paper is organized as follows. In Section 2, we review the current state of the art. In Section 3, we describe our method to ensure the logical integrity of the data and detail two examples where this method is applied. In Section 4, we evaluate our method inserting tuples and analyse the results of these insertions. The article finishes in Section 5 with the conclusions and the proposed future work.

2. Related work

Most works that study the integrity of the data are focused on the physical integrity of the data [19]. This integrity is related to the consistency of a row replicated throughout all of the replicas in a Cassandra cluster. However, in this work we will study the maintenance of the logical integrity of the data, which is related to the integrity of the data repeated among several tables.

Logical data integrity in cloud systems has been studied regarding its importance in security [16] [17]. In these studies, research is carried out into how malicious attacks can affect the data integrity. As in our work, the main objective is to ensure the logical integrity, although we approach it from modifications of data implemented in the application that works with the database rather than from external attacks.

The problem of the maintenance of the logical data integrity has been researched by the official team of Cassandra, partially solving it by developing the feature “Materialized views” [20]. These “Materialized views” are table-like structures where the data integrity is ensured automatically on the server side. Usually, in Cassandra data modelling, a table is created to satisfy one specified query. However, with this feature the data stored in the created tables (named base tables) can be queried in several ways through Materialized Views, which are query-only tables (data cannot be inserted in them). Whenever there is a modification of data in a base table, it is immediately reflected in the materialized views. Each materialized view is synchronized with only one base table, not being possible to display information from more tables, unlike what happens in the materialized views of the relational databases. To implement a table as a materialized view it must include all the primary keys of the base table. Scenarios like queries that retrieve data from more than one base table cannot be achieved by using Material Views, requiring the creation of a normal Cassandra table. In this work we approach a solution for the scenarios that cannot be obtained using these Materialized Views.

Related to the aforementioned problem is the absence of Join operations in Cassandra. There has been research [21] about the possibility of adding the Join operation in Cassandra. This work achieves its objective of implementing the join by modifying the source code of Cassandra 2.0. However, it still has room for improvement with regard to its performance.

The use of a conceptual model for the data modelling of Cassandra databases has also been researched [22], proposing a new methodology for Cassandra data modelling. In this methodology the Cassandra tables are created based also on a conceptual model, in addition to the queries. This is achieved by the definition of a set of data modelling principles, mapping rules, and mappings. This research [22] introduces an interesting concept: using a conceptual model that is directly related to the Cassandra tables, an idea that we use for our approach.

The conceptual model is the core of the previous research [22]. However, it is unusual to have such a model in NoSQL databases. To address this problem, there have been studies that propose the generation of a conceptual model based on the database tables. One of these works [23] presents an approach for inferring schemas for document databases, although it is claimed that the research could be used for other types of NoSQL databases. These schemas are obtained through a process that, starting from the original database, generates a set of entities, each one representing the information stored in the database. The final product is a normalized schema that represents the different entities and relationships.

In this work we propose an approach for maintaining data integrity in Cassandra database. This approach differs from the related works of [22] and [23] in that they are focused on the generation of database models while in our approach we are focused on the data stored in the database. Our approach maintains data integrity in all kinds of tables, contrasting with the limited scenarios where Materialized Views [20] can be applied. Our approach does not modify the nature of Cassandra implementing new functionalities as [21], it only provides statements to execute in Cassandra databases.

3. ENSURE LOGICAL DATA INTEGRITY

Cassandra databases usually have a denormalized model where the same information could be stored in more than one table in order to increase the performance when executing queries, as the data is extracted from only one table. This denormalized model implies that the modification of a single datum that is repeated among several tables must be carried out in each one of these tables to maintain the data integrity. In order to identify these tables, we use a conceptual model that has a connection with the logical model (model of the Cassandra tables). This connection [22] provides us with a mapping where each column of the logical model is mapped to one attribute of the conceptual model and one attribute is mapped from none to several columns. We use this attribute-column mapping for our work to determine in which tables there are columns mapped to the same attribute.

Our approach has the goal of ensuring the data integrity in the Cassandra databases by providing the CQL statements needed for it. We have identified two use cases for our approach: the top-down and the bottom-up:

- Top-down use case: This use case is applied when the conceptual model is the reference model to define modifications of data. In this use case given a modification of data in the conceptual model (insertion, update or deletion of a tuple), our approach maps the attributes from the conceptual model to the columns of the logical model. After that, the insertions, updates and deletions of rows that must be carried out in order to ensure the data integrity are determined. Finally, our approach creates the CQL statements to apply these modifications of data.
- Bottom-up use case: This use case is applied when the logical model is the reference model to define modifications of data. In this use case, given a modification of data in the logical model (insertion, update or deletion of a row), our approach identifies through the use of the attribute-column mapping the attributes mapped to the columns of the row. Then, our approach determines the modifications of data in the conceptual model (insertion, update or deletion of tuples) equivalent to the given modification of data in the logical model. If there is no conceptual model, it should be obtained using inferring approaches like [23].

Note that the output of the bottom-up is the same as the input of the top-down. Therefore, we can combine these two use cases to systematically ensure the data integrity after a modification of data in the logical model. Note that these last modifications already ensure the logical integrity so the top-down use case does not trigger the bottom-up use case, avoiding the production of an infinite loop. The combination between these processes is illustrated in Figure 2:

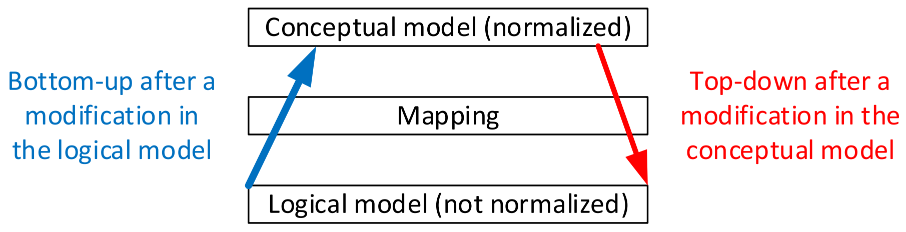

Conceptual model (normalized)
Bottom-up after a modification in the logical model
Mapping
Top-down after a modification in the conceptual model
Logical model (not normalized)

Figure 2 Top-down and bottom-up use cases combined

The scope of this work is to provide a solution for the top-down use case through a method that is detailed in the following subsection. Then, in Subsections 3.2 and 3.3 we detail two examples where we apply this method. As Cassandra excels in its performance when reading and writing data (insertions) [3], in this work we focus on the insertions of data.

3.1. Ensure logical data integrity given an insertion of a tuple

In order to provide a solution for the top-down use case, we have developed a method that identifies which tables of the database are affected by the insertion of the tuple in the conceptual model and also determines the CQL statements needed to ensure the logical data integrity. The input of this method is a tuple with assigned values to attributes of entities and relationships. Depending on where it is inserted, it contains the following values:

- Entity: values assigned to attributes of an entity. The primary key of the entity must have an assigned value.
- Relationship: values assigned to attributes of both entities and attributes of the relationship. The primary keys of both entities must have assigned values.

Our method is composed of the following steps:

1. Identify in the logical model the columns mapped to the attributes with assigned values in the tuple by means of the attribute-column mapping.
2. Collect the tables that need insertions of values in the **insert-list (**list of tables to insert the tuple**)**. Each table of the logical model is analyzed and, depending on where the tuple is inserted, the table is collected if it meets the following criteria:
    - Inserted in an entity: the primary key of the table must only contain columns mapped to attributes of the entity.
    - Inserted in a relationship: the table must contain columns mapped to attributes of at least one of the related entities. The primary key must be compound of columns mapped to attributes of these related entities.
3. For each table in the **insert-list**, generate an INSERT statement with a placeholder for the value of each column. This placeholder will be replaced by a value extracted either from the tuple to be inserted or from the database. First, through the attribute-column mapping, each column of the table is checked in order to assign a value from the tuple, whenever it is possible. If no value can be assigned from the tuple, the column is added to the **extract-list** (list of columns whose value to insert must be obtained from the database). For each column contained in the **extract-list**, the following subprocess is undertaken:

    3.1. Define the **lookup-query** to extract the value to assign to the column. The criterion of this query must be a column that uniquely identifies the value to extract. Depending on the attribute that is mapped to the column in the **extract-list** the criterion is:

    - Mapped to non-key attribute: the criterion must be a column mapped to the primary key of the entity and the value assigned to this primary key in the tuple.
    - Mapped to key attribute: the criteria must be the columns mapped to attributes of the entity with assigned values from the tuple. More than one value can be extracted by the **lookup-query**.

3.2. Find in the logical model a table where the **lookup-query** can be executed. This table must have as primary key the columns that compose the criteria of the **lookup-query** as well as the column that should store the value to be extracted. We follow a first-fit algorithm in this search, so the first table that fits the **lookup-query** is used to execute it.

3.3. Execute the **lookup-query** against the database. The placeholder for this column is replaced by the value obtained in this execution.

4. When all the INSERT statements are completed (all the columns have an assigned value), execute them.

The time complexity of our method is ***O(n)*** as it only depends on the number of tables and the statements to execute in each table. Figure 3 depicts graphically this method.

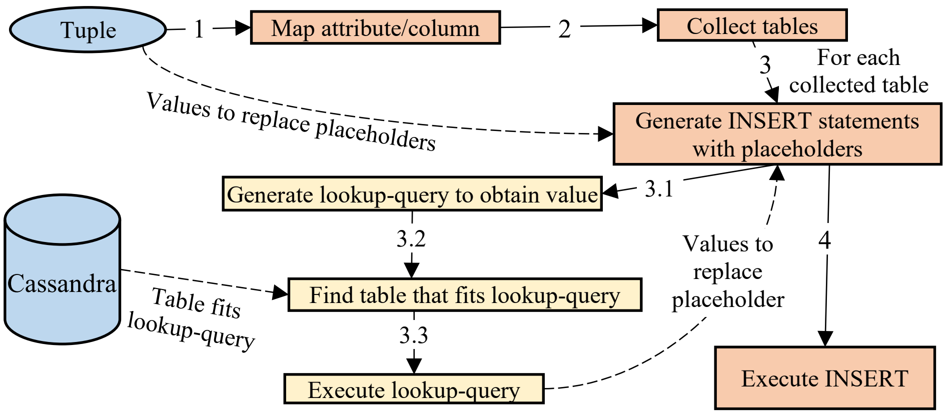

Tuple
1
Map attribute/column
2
Collect tables
3
For each collected table
Values to replace placeholders
Generate INSERT statements with placeholders
Generate lookup-query to obtain value
3.1
3.2
Cassandra
Table fits lookup-query
Find table that fits lookup-query
3.3
Execute lookup-query
Values to replace placeholder
4
Execute INSERT

Figure 3 Process of the method to maintain data integrity

3.2. Example of the insertion of a tuple in the conceptual model

In this section we detail an example where we apply our method to the insertion of a tuple in a conceptual model. The conceptual model of this example, displayed in Figure **4**, is composed of the entities "Author" and "Book", with a relationship one to many between them. Primary key attributes are labelled 'PK'. The logical model is that displayed in the introduction of this work in Figure 1. In this example we insert a tuple in the relationship 'Writes' containing the values assigned to the attributes "Id" and "Title" of a Book and the "Id" of the Author who wrote it.

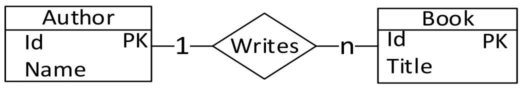
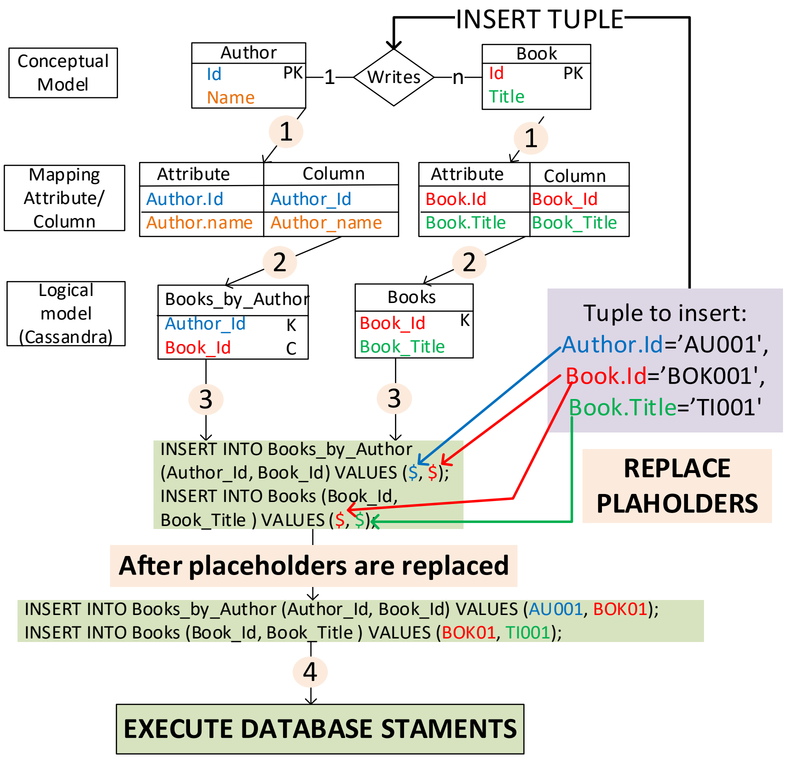


Figure 4 Conceptual Model used for the examples

First (Step 1), we map the attributes with assigned values from the tuple (attributes Id of Author and Id and Title of Book) to their columns of the logical model (columns Author_Id, Book_Id and Book_name). Then (step 2), we collect the tables "Books_by_Author" and "Books" as they contain these mapped columns. For each collected table (step 3), one

INSERT statement is generated with a placeholder ($) per column. Then, the tuple is checked, through the attribute-column mapping, in order to replace the placeholders with values from the tuple. In this example, all the placeholders are replaced with values from the tuple so these CQL statements are finally executed (step 4). This process is illustrated in Figure 5.



```
INSERT INTO Books_by_Author
(Author_Id, Book_Id) VALUES ($, $);
INSERT INTO Books (Book_Id,
Book_Title ) VALUES ($, $);
```

```
INSERT INTO Books_by_Author (Author_Id, Book_Id) VALUES (AU001, BOK01);
INSERT INTO Books (Book_Id, Book_Title ) VALUES (BOK01, TI001);
```

Figure 5 Process of ensuring the logical integrity of data given an insertion of a relationship Writes between a book and an author

3.3. *Example of the insertion of a tuple requiring lookup queries*

In this example we detail an insertion of a tuple where lookup-queries are required in order to ensure the data integrity. The conceptual model and the tuple to be inserted are the same as in the previous example. The logical model has two more tables: one created to search for authors by their id (“Author”) and another created to search for the books that an author has written by their name (“Books_by_author_name”). This model is illustrated in Figure 6.

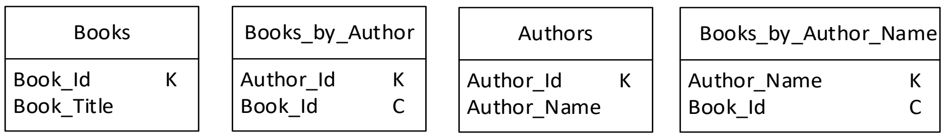
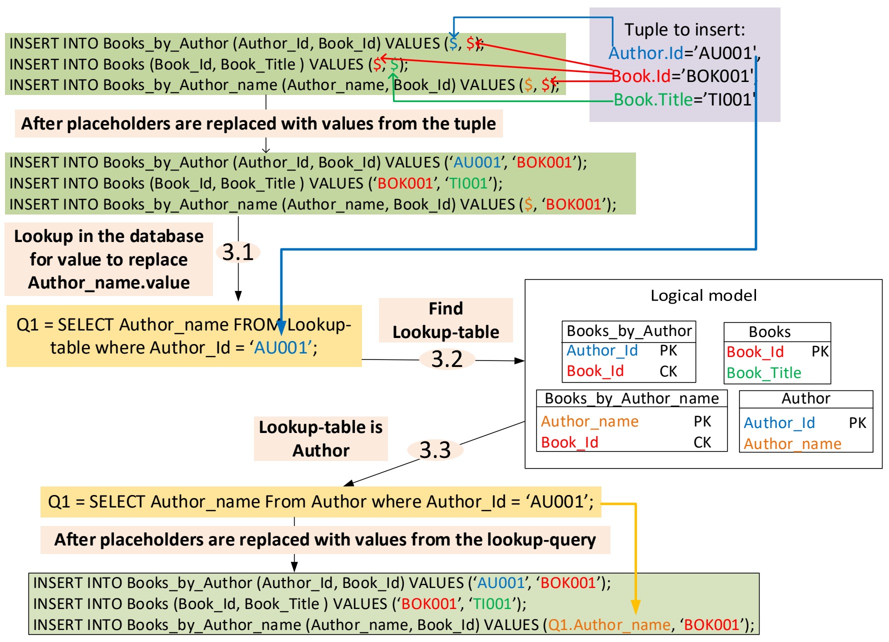


Figure 6 Logical model of example from Section 3.3

Step 1 is the same as in the example from the previous subsection but in Step 2 table 'Books_by_Author_Name" is collected as it contains attributes mapped to both entities Author and Book. In Step 3, the INSERT statements generated for both tables 'Books' and 'Books_by_Author' are also the same as in the previous example. In the case of the new collected table 'Books_by_Author_Name", the placeholder for column 'Author_Name' cannot be replaced with a value from the tuple as there is no value assigned to attribute 'Name' of 'Author' in it. Therefore, the placeholder of this column must be replaced through a lookup-query with a value extracted from the database.

In the first sub-step (3.1) the **lookup-query** is defined. As column 'Author_name' is mapped to the non-key attribute 'Name' from entity 'Author', the criterion of this query must be a column mapped to the primary key of this entity which is column 'Author_Id'. Then, a table to execute this query is searched for (Step 3.2), so it must meet the following requirements: its primary key must be column 'Author_Id' and it must also store column 'Author_name'. The table that fulfils these requirements is "Authors". In the next sub-step (3.3), the **lookup-query** is executed (Q1 in Figure 7) against the database (Step 3.3) to extract the value that replaces the placeholder in the INSERT statement. These steps are illustrated in Figure 7.



```
INSERT INTO Books_by_Author (Author_Id, Book_Id) VALUES ($, $);
INSERT INTO Books (Book_Id, Book_Title ) VALUES ($, $);
INSERT INTO Books_by_Author_name (Author_name, Book_Id) VALUES ($, $);
```

```
INSERT INTO Books_by_Author (Author_Id, Book_Id) VALUES ('AU001', 'BOK001');
INSERT INTO Books (Book_Id, Book_Title ) VALUES ('BOK001', 'TI001');
INSERT INTO Books_by_Author_name (Author_name, Book_Id) VALUES ($, 'BOK001');
```

```
Q1 = SELECT Author_name FROM Lookup-table where Author_Id = 'AU001';
```

```
Q1 = SELECT Author_name From Author where Author_Id = 'AU001';
```

```
INSERT INTO Books_by_Author (Author_Id, Book_Id) VALUES ('AU001', 'BOK001');
INSERT INTO Books (Book_Id, Book_Title ) VALUES ('BOK001', 'TI001');
INSERT INTO Books_by_Author_name (Author_name, Book_Id) VALUES (Q1.Author_name, 'BOK001');
```

Figure 7 Process of querying information required to maintain the logical integrity

4. Evaluation

In this section we detail and explain the results of applying our method to ensure the data integrity of the data for multiple insertion of tuples in entities and relationships of a case study [22]. This case study is about a data library portal with a conceptual model, illustrated in Figure 8, that contains 4 entities and 5 relationships. Its logical model is composed of 9 tables and it is illustrated in Figure 9. Counter columns are labelled as ++. In the following subsections we detail how we have systematically created the tuples to insert, the analysis of the results for the tuples inserted in entities and relationship and an overall discussion of the results.

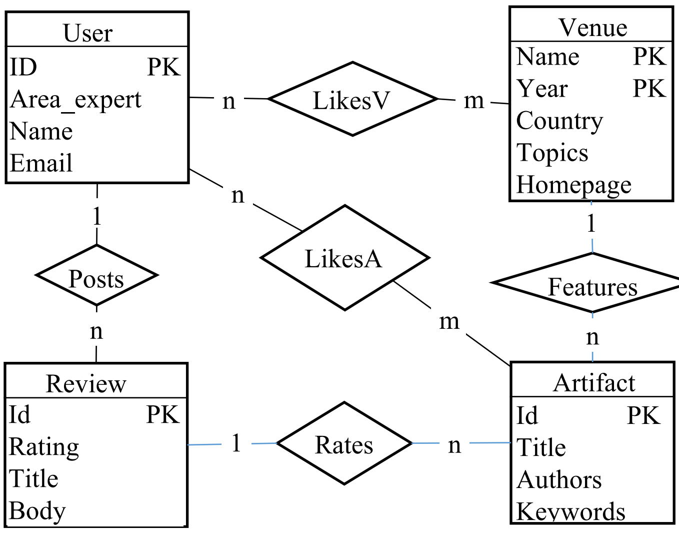
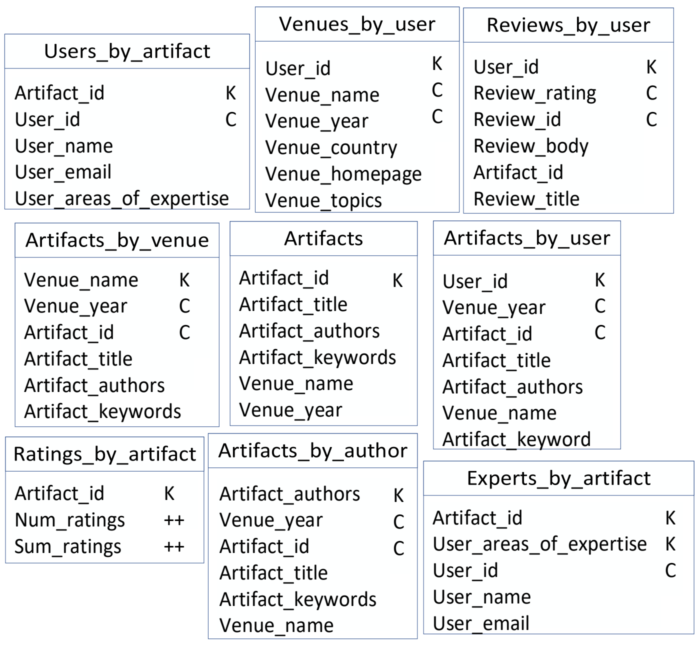


Figure 8 Conceptual model of the case study



Users_by_artifact	
Artifact_id	K
User_id	C
User_name	
User_email	
User_areas_of_expertise	

Venues_by_user	
User_id	K
Venue_name	C
Venue_year	C
Venue_country	
Venue_homepage	
Venue_topics	

Reviews_by_user	
User_id	K
Review_rating	C
Review_id	C
Review_body	
Artifact_id	
Review_title	

Artifacts_by_venue	
Venue_name	K
Venue_year	C
Artifact_id	C
Artifact_title	
Artifact_authors	
Artifact_keywords	

Artifacts	
Artifact_id	K
Artifact_title	
Artifact_authors	
Artifact_keywords	
Venue_name	
Venue_year	

Artifacts_by_user	
User_id	K
Venue_year	C
Artifact_id	C
Artifact_title	
Artifact_authors	
Venue_name	
Artifact_keyword	

Ratings_by_artifact	
Artifact_id	K
Num_ratings	++
Sum_ratings	++

Artifacts_by_author	
Artifact_authors	K
Venue_year	C
Artifact_id	C
Artifact_title	
Artifact_keywords	
Venue_name	

Experts_by_artifact	
Artifact_id	K
User_areas_of_expertise	K
User_id	C
User_name	
User_email	

Figure 9 Logical model of the case study

4.1. Selection of tuples to insert

In this section we describe the systematic selection of tuples to be inserted in the entities and relationships of the conceptual model of this case study. Depending on the number of attributes of an entity with assigned value in a tuple we have made the following classification:

- Complete (C): every attribute has an assigned value.
- Partial (P1 or P2): the primary key and some of the non-key attributes have assigned values. The number concatenated to the letter 'P' represents the number of non-key attributes with assigned value. There are only Partial 1 and Partial 2 tuples because every entity of this case study has 3 non-key attributes.
- Incomplete (I): only the primary key has an assigned value.

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 a similar 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).

4.2. Insertions in entities

Table 1 displays the results of applying our method to determine the CQL statements that are needed to insert the values in the database while maintaining the logical integrity of data over 32 insertions of tuples in entities.

In the column **Entity** we display both in which entity the tuple is inserted and a tag to display the number of attributes with assigned values: Complete (C), Partial 1 (P1), Partial 2 (P2), Incomplete (I) or ALL. These tags also indicate the number of insertions that each row represents:

- C and I: these rows display the information of an insertion of a tuple (complete or incomplete)
- P1 and P2: these rows display the information of three insertions of tuples (three Partial 1 or three Partial 2). We have comprised the results of all insertions of Partial 1 tuples in a single row as all of them returned the same results. This also happens with the insertions of Partial 2 tuples.
- ALL: These rows display the information of eight tuple insertions. These are the cases where the 8 insertions of tuples in an entity return the same result.

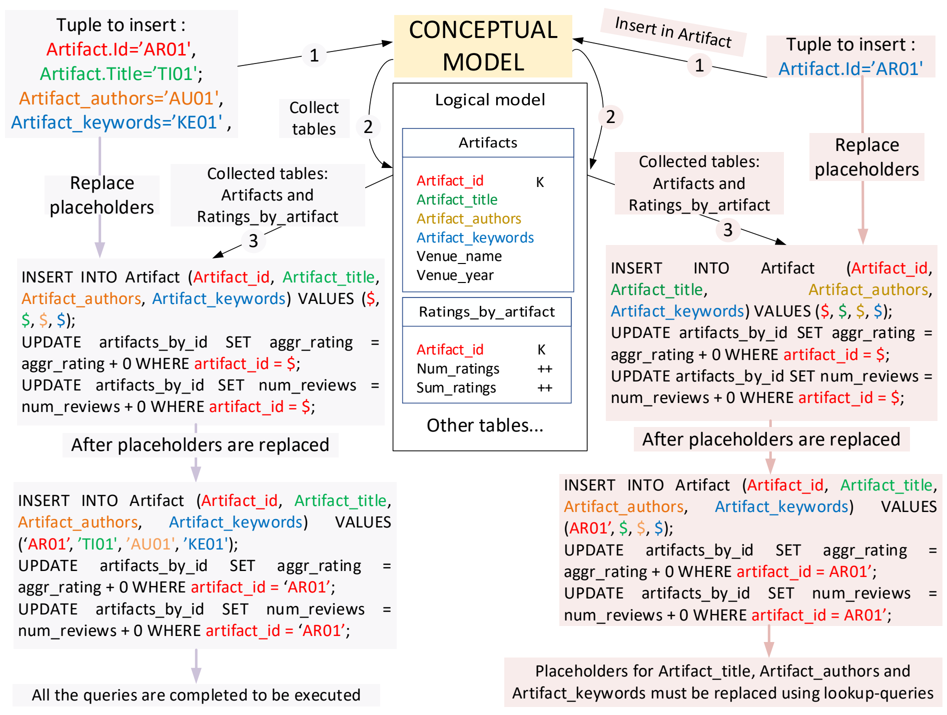
The number of insertions of tuples that the row represent are displayed in column **Insertions represented**. The outputs are displayed in the columns **INSERT, UPDATE** and **SELECT,** with the number of statements for each of these operations and in column **Total** with the sum of all of these operations.

Table 1 Evaluated Insertions in Entities

Entity	Insertions represented	INSERT	UPDATE	SELECT	Total
Venue (ALL)	8	0	0	0	0
User (ALL)	8	0	0	0	0
Review (ALL)	8	0	0	0	0
Artifact (C)	1	1	2	0	3
Artifact (P2)	3	1	2	1	4
Artifact (P1)	3	1	2	2	5
Artifact (I)	1	1	2	3	6

In these insertions, we observe how only the tuples inserted in Artifact have CQL statements in their input, as these values can be inserted in the tables "Artifacts' and "Ratings_by_Artifacts". On the other hand, the tuples that are inserted in the entities Venue, User or Review have an empty output (0 CQL statements) as they cannot be inserted in any table. This is because none of the tables of the logical model has as primary key columns mapped to attributes of these entities (Step 2 in our method to ensure the data integrity). If a developer wants to specifically insert data of just these entities the logical model should be modified by adding tables that contain information on only these entities. With the current state of the logical model, the data related to these entities is not queried alone, only when they are related with data from other entities. Therefore, there are no tables where the data pertaining to only one of these entities can be inserted.

We also observe an inverse relation between the number of attributes with assigned value and the lookup-queries created (SELECT statements). The more attributes with assigned values the tuple has, the less lookup-queries are needed. This is because in Step 3 of our method, the more attributes with assigned value the tuple has, the more placeholders can be replaced with these values. For example, for tuples inserted in Artifact, when the tuple is complete there is no need for lookup-queries (0 SELECT statements) but for incomplete tuples 3 lookup-queries were needed (3 SELECT statements). This is illustrated in Figure 10.



```
INSERT INTO Artifact (Artifact_id, Artifact_title, Artifact_authors, Artifact_keywords) VALUES ($, $, $, $);
UPDATE artifacts_by_id SET aggr_rating = aggr_rating + 0 WHERE artifact_id = $;
UPDATE artifacts_by_id SET num_reviews = num_reviews + 0 WHERE artifact_id = $;
```

```
INSERT INTO Artifact (Artifact_id, Artifact_title, Artifact_authors, Artifact_keywords) VALUES ('AR01', 'TI01', 'AU01', 'KE01');
UPDATE artifacts_by_id SET aggr_rating = aggr_rating + 0 WHERE artifact_id = 'AR01';
UPDATE artifacts_by_id SET num_reviews = num_reviews + 0 WHERE artifact_id = 'AR01';
```

```
INSERT INTO Artifact (Artifact_id, Artifact_title, Artifact_authors, Artifact_keywords) VALUES ($, $, $, $);
UPDATE artifacts_by_id SET aggr_rating = aggr_rating + 0 WHERE artifact_id = $;
UPDATE artifacts_by_id SET num_reviews = num_reviews + 0 WHERE artifact_id = $;
```

```
INSERT INTO Artifact (Artifact_id, Artifact_title, Artifact_authors, Artifact_keywords) VALUES (AR01', $, $, $);
UPDATE artifacts_by_id SET aggr_rating = aggr_rating + 0 WHERE artifact_id = AR01';
UPDATE artifacts_by_id SET num_reviews = num_reviews + 0 WHERE artifact_id = AR01';
```

Figure 10 Comparison between complete tuple and incomplete tuple of Artifact

4.3. Insertions in relationships

Table 2 displays the results of applying our method to determine the CQL statements needed to maintain the data integrity over 320 insertions of tuples in relationships.

The inputs for the relationships are displayed in the following columns:

- **Relationship**: relationship where the tuple is inserted.
- **Entity I and Entity II**: entities related with a tag to display the number of attributes with assigned values for that entity in the tuple (C, P2, P1, I and ALL). Rows with tags P1 or P2 display the output of any combination that compounds a partial tuple of their type, similarly as in the insertions in entity where they return the same results regardless of which attributes have assigned values. In the rows where the tag is ALL it means that it displays the output for all the combinations of tuples inserted, as it is the same output regardless of the number of attributes with an assigned value (C, P1, P2, I).
- **Relationship Master**: 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 master entities. These relationships are displayed in this column.

The outputs are displayed in the columns **INSERT, UPDATE** and **SELECT** with the number of statements for each of them and in the column **Total** with the sum of all of them.

The rows represent the different number of insertions depending on the tags in columns **Entity I** and **Entity II**. As in these insertions there are attributes of two entities, the number of insertions that a row represents is the multiplication of the different possible combinations from the two entities. For example, if the tag in Entity I is 'P1' (there are 3 'Partial 1' combinations) and the tag for Entity II is 'ALL' (all 8 combinations for an entity) then the row represents 24 rows. These numbers are displayed in column **Insertions represented**.

Table 2 Insertion of tuples in relationships

Relationship	Entity I	Entity II	Relationship Masters	Insertions represented	INSERT	UPDATE	SELECT	Total
Features	Venue(ALL)	Artifact (C)	-	8	3	2	0	5
Features	Venue(ALL)	Artifact (P2)	-	24	3	2	3	8
Features	Venue(ALL)	Artifact (P1)	-	24	3	2	6	11
Features	Venue(ALL)	Artifact (I)	-	8	3	2	9	14
Posts	Review (C)	User (ALL)	Rates	8	1	2	0	3
Posts	Review (P2)	User (ALL)	Rates	24	1	2	1	4
Posts	Review (P1)	User (ALL)	Rates	24	1	2	2	5
Posts	Review (I)	User (ALL)	Rates	8	1	2	3	6
Rates	Review (C)	Artifact (C)	Posts & Features	1	4	2	0	6
Rates	Review (C)	Artifact (P2)	Posts & Features	3	4	2	3	9
Rates	Review (C)	Artifact (P1)	Posts & Features	3	4	2	6	12
Rates	Review (C)	Artifact (I)	Posts & Features	1	4	2	9	15
Rates	Review (P2)	Artifact (C)	Posts & Features	3	4	2	1	7
Rates	Review (P2)	Artifact (P2)	Posts & Features	9	4	2	4	10
Rates	Review (P2)	Artifact (P1)	Posts & Features	9	4	2	7	13
Rates	Review (P2)	Artifact (I)	Posts & Features	3	4	2	10	16
Rates	Review (P1)	Artifact (C)	Posts & Features	3	4	2	2	8
Rates	Review (P1)	Artifact (P2)	Posts & Features	9	4	2	5	11
Rates	Review (P1)	Artifact (P1)	Posts & Features	9	4	2	8	14
Rates	Review (P1)	Artifact (I)	Posts & Features	3	4	2	11	17
Rates	Review (I)	Artifact (C)	Posts & Features	1	4	2	3	9
Rates	Review (I)	Artifact (P2)	Posts & Features	3	4	2	6	12
Rates	Review (I)	Artifact (P1)	Posts & Features	3	4	2	9	15
Rates	Review (I)	Artifact (I)	Posts & Features	1	4	2	12	18
LikesV	User (ALL)	Venue (C)	-	8	1	0	0	1
LikesV	User (ALL	Venue (P2)	-	24	1	0	1	2
LikesV	User (ALL)	Venue (P1)	-	24	1	0	2	3
LikesV	User (ALL)	Venue (I)	-	8	1	0	3	4
LikesA	Artifact (C)	User (C)	Features	1	6	2	0	8
LikesA	Artifact (C)	User (P2)	Features	3	6	2	2	10
LikesA	Artifact (C)	User (P1)	Features	3	6	2	4	12
LikesA	Artifact (C)	User (I)	Features	1	6	2	6	14
LikesA	Artifact (P2)	User (C)	Features	3	6	2	8	16
LikesA	Artifact (P2)	User (P2)	Features	9	6	2	10	18
LikesA	Artifact (P2)	User (P1)	Features	9	6	2	12	20
LikesA	Artifact (P2)	User (I)	Features	3	6	2	14	22
LikesA	Artifact (P1)	User (C)	Features	3	6	2	16	24
LikesA	Artifact (P1)	User (P2)	Features	9	6	2	10	18
LikesA	Artifact (P1)	User (P1)	Features	9	6	2	12	20
LikesA	Artifact (P1)	User (I)	Features	3	6	2	14	22
LikesA	Artifact (I)	User (C)	Features	1	6	2	16	24
LikesA	Artifact (I)	User (P2)	Features	3	6	2	18	26
LikesA	Artifact (I)	User (P1)	Features	3	6	2	20	28
LikesA	Artifact (I)	User (I)	Features	1	6	2	22	30

These results show again the inverse relation between the number of attributes with assigned value and the creation of lookup-queries. In all insertions of tuples that do not have the information of both entities complete (all attributes with assigned values), lookup-queries are needed. This shows how it is common to look up values in the database in order to ensure the data integrity.

In the previous section, it was shown how it was not possible to insert values of attributes of different entities in the database, such as those of the entity Review. However, in this section we have observed how these values are inserted in the database when they are contained in the tuple along with the values to establish the relationships Post & Rates. This is observed in tuples inserted in the relationship Posts (Rows 5 to 8 of Table 2) where there is 1 INSERT statement and 2 UPDATE statements in each one of them. We compare these insertions in the entity Review and in the relationship Post in Figure 11. In this illustration. both tuples contain the complete information of a Review and the tuple inserted in Posts also contains values assigned to the primary keys of User and Artifact in order to establish the relationships Post and Features (Review is detail of Artifact). For the tuple inserted in Review (Step 1 on the left), no table is collected (Step 2 on the left) because no table has as primary key columns mapped to attributes from only Review. However, in this same step 2 for the tuple inserted in Post (on the right side), two tables are collected: Reviews_by_User and Ratings_by_artifact. Although Reviews_by_User contains two columns mapped to attributes from the entity Review,it also contains another mapped to an attribute of User, explaining why it was not collected for the tuple inserted in Review.

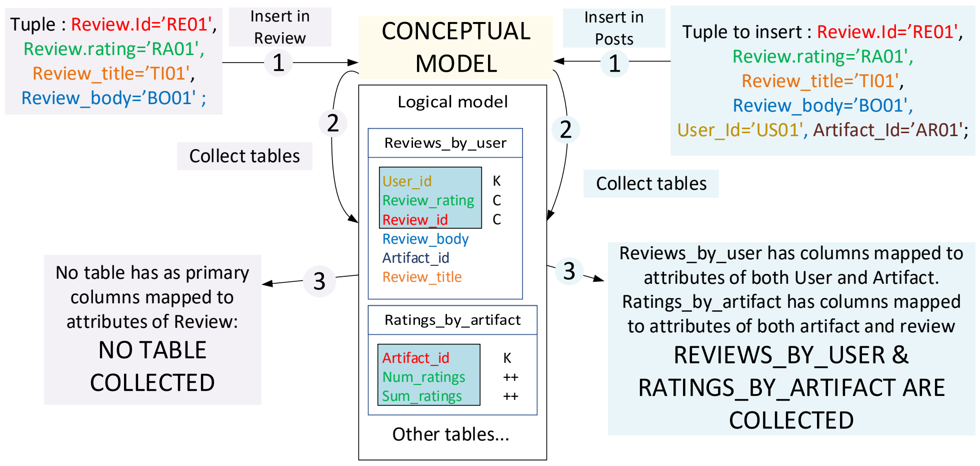

Tuple : Review.Id='RE01', Review.rating='RA01', Review_title='TI01', Review_body='BO01' ;
Insert in Review
1
CONCEPTUAL MODEL
Insert in Posts
1
Tuple to insert : Review.Id='RE01', Review.rating='RA01', Review_title='TI01', Review_body='BO01', User_Id='US01', Artifact_Id='AR01';
2
Collect tables
Logical model
Reviews_by_user
User_id K
Review_rating C
Review_id C
Review_body
Artifact_id
Review_title
2
Collect tables
No table has as primary columns mapped to attributes of Review:
NO TABLE COLLECTED
3
3
Reviews_by_user has columns mapped to attributes of both User and Artifact. Ratings_by_artifact has columns mapped to attributes of both artifact and review
REVIEWS_BY_USER & RATINGS_BY_ARTIFACT ARE COLLECTED
Ratings_by_artifact
Artifact_id K
Num_ratings ++
Sum_ratings ++
Other tables...

Figure 11 Difference of tables collected depending on attributes with assigned value in the tuple

4.4. Overall discussion of the results

We have seen in both types of insertions that usually several statements are needed to insert the tuple while ensuring the data integrity due to the denormalized model. A summary of these insertions is displayed in **Table 3**: the number of tuples inserted, and **Total**, **Average** and **Maximum** number of operations INSERT, UPDATE and SELECT operations needed to ensure the data integrity.

Table 3 Summary of the results for ensuring the data integrity for the inserted tuples

		Operation INSERT			Operation UPDATE			Operation SELECT		
Entity/ Relationships	Number of inserted tuples	Total	Average	Maximum	Total	Average	Maximum	Total	Average	Maximum
Artifact	8	8	1	1	16	2	2	24	4.5	9
Review	8	0	0	0	0	0	0	0	0	0
User	8	0	0	0	0	0	0	0	0	0
Venue	8	0	0	0	0	0	0	0	0	0
Features	64	192	3	3	128	2	2	285	4.45	9
LikesA	64	384	6	6	128	2	2	728	11.37	22
LikesV	64	64	1	1	0	0	0	96	1.5	3
Posts	64	64	1	1	128	2	2	96	1.5	3
Rates	64	256	4	4	128	2	2	384	6	12
Total	352	968	2.75	6	528	1.5	2	1623	4.61	22

The results displayed in Table 3 show that, in general, a denormalized logical model requires several database statements to ensure the logical integrity of the data in order to insert the values of a tuple in the Cassandra tables. For 352 insertions in the conceptual model, we needed 968 INSERT statements, 528 UPDATE statements and 1623 SELECT statements to ensure the logical integrity in the Cassandra database. As previously explained, there is an empty output (no database statements) in the particular cases of the insertions of tuples that only contain values assigned to attributes of entities Venue, Review or User. This is because no table has as primary key, a column mapped to only attributes of these entities. The information of a Venue, Review or a User must be inserted alongside the information of relationships such as LikesV, Posts or LikesA, respectively.

In 75% of the insertions carried out, data needed to be inserted in more than one table. This shows how a denormalized model such as the logical model contrasts with a normalized model like the conceptual model. An insertion of a single tuple in the conceptual model can mean several insertions in different tables of the logical model. The SELECT statements (lookup-queries) are also quite common in order to ensure the data integrity, there being at least one in 93.45% of the insertions.

We have also detected an inverse relation between the number of SELECT statements and the number of attributes with an assigned value in the tuple. The tuples inserted in entities can contain up to 3 non-key attributes with assigned values while those inserted in relationships contain up to 6 non-key attributes with assigned values (the combination of the 3 attributes of each entity of the relationship). This inverse relationship is shown in Figure 12 where each bar represents the average of SELECT operations needed for the number of attributes with an assigned value in the tuple. We observe how the average of SELECT operations decreases as the number of attributes with assigned value increases.

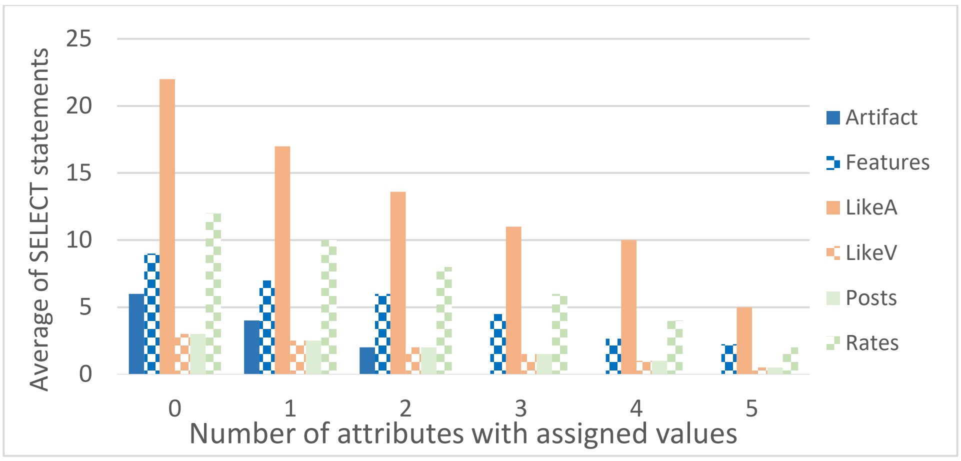


Figure 12 Inverse Relationship between SELECT operations and the number of attributes with assigned value

4.5. Threats to validity

The main threats to validity to this work are related to the optimization of our algorithm and the confirmation that the CQL statements determined by it ensure data integrity. For the first threat, currently our method always obtains a single value when executing a query in step 3.2. This process can be quite inefficient as multiple queries with the same criteria can be executed against the same table. To optimize this process, we want in the future to modify this process in step 3.2. The method will be designed to minimize the number of queries by maximizing the number of columns in each query.

Regarding the second threat, we have inspected very carefully the statements that our method generates in order to ensure that they maintain the data integrity in Cassandra. However, in an ongoing work we are developing an oracle that is able to automatically determine that the database statements generated by our method to insert a certain tuple maintain the data integrity.

5. CONCLUSIONS

Nowadays, the use of NoSQL databases for web systems like cloud environments is increasing due to the performance advantages they provide processing big data. Despite the improved performance, there are further problems such as how to ensure the data integrity in these databases. In this work we have proposed a method that given an insertion in conceptual model it detects the tables that are affected by this insertion and the CQL statements needed to ensure the data integrity of the database. Without a method like this, developers need to manually determine these statements very carefully in order to not implement statements that incur in the production of inconsistencies of the data. We have also evaluated our method in a case study where we inserted several tuples in both entities and relationships, successfully ensuring the data integrity. We have observed that in most cases it was necessary to insert data in more than one table due to the denormalization of the data in several tables. This denormalization means that an insertion of a tuple in a normalized model implies several insertions in the denormalized model. Another observation was that it is very common to need to query data from the database through the execution of queries in these insertions of tuples. Both the insertions and the querying of data show how complex it can be to ensure the data integrity as several statements are required in order for it to be achieved. This complexity also increases when more tables with the same repeated information are in the logical model.

We conclude that our method helps developers to ensure data integrity in client applications as web services that may work with databases composed of dozens or even hundreds of tables. Using the proposed method, data integrity is always ensured regardless of the number of tables that need maintenance. This saves time and money as the developer does not need to manually determine these statements. This method is also able to ensure data integrity in a Cassandra database regardless of what tables compose the database. This is an improvement from other approaches like the Materialized Views which need specific restrictions to be met in order to use them. However, we consider that it also possible to combine our method with the Materialized Views by creating tables as Materialized Views whenever it is possible and using our method for the remaining tables.

As future work we want to delve deeper into the bottom-up use case by proposing a method for integrating it with the method proposed in this work for the top-down use case in order to provide a full solution when there is a modification of data in the logical model. Regarding the optimization of our method we want to reduce the number of queries as we have detailed in the threats to validation. Another future research line is how to create conceptual models based solely on the logical model so that the systems that were not created with a conceptual model can also use our method. Finally, the whole approach may leverage the Model-driven engineering paradigm. As the inputs of the top-down approach are a conceptual model and the queries issued against it, the CQL query generation could be integrated in an MDE framework as an extension of its code generation capabilities.

ACKNOWLEDGMENTS

This work was supported by the TESTEAMOS project (MINECO-17-TIN2016-76956-C3-1-R) and the PERTEST project (MINECO-13-TIN2013-46928-C3-1-R) of the Ministry of Economy and Competitiveness, Spain. It has also been supported by the project GRUPIN14-007 of the Principality of Asturias and by the ERDF.

REFERENCES

[1] Moniruzzaman, A. B. M, Hossain and Syed Akhter (2013). *Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv preprint arXiv:1307.0191.*

[2] Leavitt, Neal. (2010). Will NoSQL databases live up to their promise? Computer, Vol 43, No 2, pp 12-14.

[3] Li, Yishan, and Manoharan, Sathiamoorthy. (2013). *A performance comparison of SQL and NoSQL databases*. In Communications, computers and signal processing, pp 15-19

[4] Cattell, Rick. (2011). Scalable SQL and NoSQL data stores. Acm Sigmod Record, Vol 39, No 4, pp 12-27

[5] Tauro, Clarence. JM, Aravindh, Shreeharsha and Shreeharsha, A. B. (2012). *Comparative study of the new generation, agile, scalable, high performance NOSQL databases*. International Journal of Computer Applications, Vol 48, No 20, pp. 1-4.

[6] Bhogal, Jagdev and Choksi, Imran (2015). Handling big data using NoSQL. In *IEEE 29th International Conference on Advanced Information Networking and Applications Workshops (WAINA),* pp. 393-398

[7] Pokorny, Jaroslav (2013). NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, Vol *9 No* 1, pp 69-82.

[8] MongoDB Inc (2019). Who uses MongoDB https://www.mongodb.com/who-uses-mongodb Accesed: 2019-03-13

[9] Datastax (2019). Case Studies, https://www.datastax.com/resources/casestudies Accessed: 2019-03-13

[10] Apache Software Foundation. (2016). *Apache Cassandra,* http://cassandra.apache.org/ Accessed: 2019-03-13

[11] Han, Jing et al (2011). *Survey on NoSQL database*. In 6th international conference on Pervasive computing and applications (ICPCA), 2011 pp. 363-366

[12] Datastax (2015). *Basic Rules of Cassandra Data Modeling,* https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling Accessed 2019-03-13

[13] Rajanarayanan Thottuvaikkatumana. (2015). *Cassandra Design Patterns, second edition*, ed. Packt Publishing Ltd

[14] Suárez-Otero, Pablo, Suárez Cabal, María José and Tuya, Javier (2018). Leveraging Conceptual Data Models for Keeping Cassandra Database Integrity. In WEBIST 2018, pp 398-403

[15] Apache Software Foundation (2016). The Cassandra Query Language (CQL) http://cassandra.apache.org/doc/latest/cql/ Accessed 2019-03-13

[16] Ghazizadeh, Puya, Mukkamala, Ravi and Olariu, Stephan (2013). Data Integrity Evaluation in CloudDatabase-as-a-Service. In *IEEE Ninth World Congress on Services* pp 280-285

[17] Aniello, Leonard et al (2017). Blockchain-based Database to Ensure Data Integrityin Cloud Computing Environments. In *13th European Dependable Computing Conference (EDCC)*, pp 151-154

[18] Olmsted, Aspen and Santhanakrishnan, Gayathri (2016). Cloud Data Denormalization of Anonymous Transactions. In Cloud Computing Seventh International Conference on Cloud Computing, GRIDs, and Virtualization, pp 42-46.

[19] Datastax. (2017). How are consistent read and write operations handled? : https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html Accessed 2019-03-13

[20] Datastax (2015). *New in Cassandra: Materialized Views:* https://www.datastax.com/dev/blog/new-in-cassandra-3-0-materialized-views Accessed 2019-03-13

[21] Christian Peter. (2015). *Supporting the Join Operation in a NoSQL System. Master's thesis*. Norwegian university of Science and Technology, Norway

[22] Chebotko, Artem; Kashlev, Andrey and Lu, Shiyong (2015). *A Big Data Modeling Methodology for Apache Cassandra*. In IEEE International Congress on Big Data (BigData'15), pp. 238-245

[23] Sevilla Ruiz, Diego, Morales Feliciano, Severino and García Molina, Jesús (2015). *Inferring versioned schemas from NoSQL databases and its applications.* In International Conference on Conceptual Modeling (ER 2015), pp. 467-480

[24] Datastax (2019). Creating a table: https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/useCreateTable.html Accessed 2019-05-20

Biographies

Pablo Suárez-Otero received his B.Sc. degree in Computer Engineering in 2015 and in M.Sc. in Computer Engineering in 2017 from the University of Oviedo. He is currently a PhD candidate at the University of Oviedo. He is also an Assistant Professor at the University of Oviedo. He is a member of the Software Engineering Research Group. His research interests include software testing, NoSQL databases and data modelling.

María José Suárez-Cabal is an assistant professor at the University of Oviedo, Spain, and is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). She obtained her PhD in Computing from the University of Oviedo in 2006. Her research focusses on software testing, and more specifically on testing database applications.

Javier Tuya is Professor in the Computing Department at the University of Oviedo, Spain. His current research interests in the field of Software Testing include database driven applications, data engineering, testing techniques and automation. He has been the manager in many research and technology transfer projects and published in different international conferences and journals. He held the position of CIO of the University of Oviedo and currently he is Director of the Indra-Uniovi Chair, member of the ISO working group that works in the development of the new software testing standard ISO/IEC/IEEE 29119, and convenor of the UNE national body workgroup on software testing.