2020 IEEE International Conference on Big Data (Big Data)

Maintaining NoSQL Database Quality During Conceptual Model Evolution

Pablo Suárez-Otero
Computer Science Department
University of Oviedo
Gijón, Spain
suarezgpablo@uniovi.es

Michael J. Mior
Department of Computer Science
Rochester Institute of Technology
Rochester, New York, USA
mmior@cs.rit.edu

Maria José Suárez-Cabal
Computer Science Department
University of Oviedo
Gijón, Spain
cabal@uniovi.es

Javier Tuya
Computer Science Department
University of Oviedo
Gijón, Spain
tuya@uniovi.es

***Abstract*—Database schemas evolve over time to satisfy changing application requirements. If this evolution is not performed correctly, some quality attributes are at risk such as data integrity, functional correctness, or maintainability. To help developer teams in the design of database schemas, several design methodologies for NoSQL databases have proposed to use conceptual models during this process. The use of an explicit conceptual model can also help developers in the tasks of schema evolution. In this work-in-progress paper, we propose a framework that, given a change in the conceptual model, identifies what must be modified in a NoSQL database schema and the underlying data. We researched several open source projects that use Apache Cassandra to study the benefits of using a conceptual model during the schema evolution process as well as to understand how these models evolve. In this first work, we have focused on studying seven types of conceptual model changes identified in these projects. For each change we describe the transformation required in the database schema to maintain the consistency between the schema and the model as well as the migration of data required to the new schema version.**

Keywords—quality, database evolution, NoSQL, column-oriented databases, conceptual model

I. INTRODUCTION

Data models such as conceptual models and database schemas are designed to satisfy the requirements of software applications. These requirements may change during the lifetime of an application, also forcing changes to the schema or the conceptual model that imply further changes in the data that may jeopardize the integrity of the existing data if not done correctly. These changes to the schema are referred as schema evolution, which has been studied for both relational databases [1][3] and NoSQL databases [6][10]. In the case of NoSQL databases, the data model is different in each database; some are considered schema-less in that they have a flexible schema such as Neo4J and some others such as Cassandra [21] have a stronger column-oriented schema (similar to relational databases). Schema evolution in NoSQL databases with strictly defined schemas presents problems which are similar to those suffered by relational databases but also influenced by the unique characteristics of NoSQL databases. For instance, in the case of Cassandra, schema design follows a query-driven approach [21]. This means that if a single datum is queried using multiple criteria, this datum may need to be stored in more than one table. Any evolution of the schema that modifies one of these tables must also consider the other tables where the datum is stored to maintain data integrity.

In order to help developers design NoSQL databases, several design methodologies recommend using a conceptual model to create the database schema. For instance, Chebotkto et al. [11] and Mior et al. [12] propose to obtain a schema for the database Cassandra using both an explicit conceptual model and the queries required by the client application. Even without these methodologies, developers usually have in mind an implicit conceptual model to design the schema. Without such a conceptual model, it would be difficult to perform schema management tasks such as its evolution. Changes to the conceptual model imply a required evolution of the database schema in order to maintain consistency with the database schema and ensure its quality. Another problem is that migration of data may be needed when changing the conceptual model. Note that some of the data that must be migrated to the new schema version may contradict new constraints in the conceptual model such as a change of the cardinality of a relationship. This last problem is not presented in the aforementioned data model design methodologies, as their objective was to create a new schema that did not require any migration of data.

In this work, we address the aforementioned problems regarding schema evolution when the conceptual model changes. We define a framework that provides the modifications required in the database schema to reflect the conceptual model change and to maintain the consistency between this schema and this model. Additionally, we approach the migrations of data and change of the application queries required after the schema change. In this first work, we focus on column-oriented databases using Cassandra as the main database for the case study selections. The main contributions of this work are:

1. A study of open source projects that use Cassandra. We analyze the evolution of the conceptual model in these projects, how a conceptual model aids evolution, and the identification of conceptual model changes during the evolution process.
2. A framework that determines what needs to be modified in the database schema to reflect a conceptual model change as well as how to migrate the data to the new schema version. It also suggests possible changes to application queries in order to adapt them to the new schema.

The remaining of this paper is structured as follows. Section 2 details the related work. In Section 3, we study several projects that use Cassandra and their evolution of the schema. Section 4 presents our schema evolution framework. Section 5 finishes with conclusions and future work.

II. Related work

Schema evolution research focused on relational databases has approached several topics specific to these databases such as how integrity constraints evolve [1], maintenance during schema changes [2], how foreign key changes affect the database [3], or recommendations to properly evolve the schema [4]. Other works are focused on studying how the schema must be modified after a change in the ontology considering also the data stored in the database in this modification [5]. Because these works are focused on relational databases, it is difficult to use them for other systems such as NoSQL databases.

Scherzinger et al. have analyzed different topics related to schema evolution on NoSQL databases [6], [7], [8], [9], [10]. They propose a first approach to manage schema evolution [6] that is then used to define an evolution schema framework named ControVol [7]. They also defined the middleware Darwin [8] that proposes mappings between versions of the same schema, also calculating the monetary cost of these migrations [9]. Finally, they present a self-adapting methodology to choose the best strategy for migrating data between different versions of the same NoSQL database [10]. These works have addressed several issues in both relational and NoSQL databases but only consider direct changes to the database schema.

Regarding schema design, there are several works that have given a great importance to the conceptual model. To obtain a database schema Chebotko et al. [11] proposed using a conceptual model in addition to the queries by defining the KDM (Kashlev Data Modeler) tool. Each query required by the client application is transformed into a table of the logical model (similar to the database but without data types) and then to a table of the database schema using a set of transformation rules. With the same objective, and also using a conceptual model, Mior et al. [12] incorporate statistical information about query frequency and expected data volume developing the NoSE (NoSQL Schema Evaluator) tool. Following the same research line, De la Vega et al. [13] use KDM and NoSE as a starting point to devise the Mortadelo tool which generalizes the generation of database schemas by also addressing the schema design of document-oriented databases such as MongoDB.

III. Project Analysis

In this section, we study the schema of several open source projects using the Cassandra database and the evolution of these schemas to determine what kind of changes happened during their evolution and to understand how an explicit conceptual model can help during this process. We detail this information in the answers to the following questions:

1. Q1: Why and how can an explicit conceptual model help when defining and evolving database schemas?
2. Q2: How do conceptual models evolve and how is this evolution reflected in the schema?

In the next subsections, we briefly describe the case study selection and answer both questions.

A. Case study selection

We found seven projects with updates in their schema throughout successive versions in public repositories such as GitHub and GitLab that we will use to answer Q1 and Q2:

- Thingsboard [14]: IoT platform for data collection.
- Minds[15]: Open source social network.
- Powsybl[16]: Framework to simplify development of software for power system simulations and analysis.
- Wireapp[17]: Encrypted communication app.
- COVID-19[18]: Manages COVID-19 cases from Italy.
- Reviews-service[19]: Module for restaurant.
- Blobkeeper[20]: Distributed file-storage service.

TABLE I. displays a summary of these projects showing the number of Cassandra tables of each project and how many times each type of conceptual model change happened.

B. Q1: Why and how can an explicit conceptual model help when defining and evolving database schemas?

None of the selected projects provide in their repositories a conceptual model, but they could have used a conceptual model during the application development. As mentioned in the introduction, developers may have an implicit conceptual model in mind when modeling the database schema. In these projects we have found evidence of the use of such a conceptual model like in the terminology used to name the tables and the columns. All the projects follow the naming conventions of the columns, where it is compound of the name of the associated attribute and, in some cases, the entity of this attribute as well. For instance, in table "comment" from the project Minds we observe several of these columns such as "access_id", "container_guid" or "owner_guid".

Another evidence that we have found regarding the use of at least an implicit conceptual model is the consistency found when implementing relationships. To ensure row uniqueness every table that stores a relationship contains the columns that are associated to the primary keys of the relationship entities. For instance, we observe how particular columns are constantly duplicated in several tables establishing several relationships of a particular entity with others. The most recurrent case of this are seem in the projects Minds and Powsybl with the columns "user_guid" duplicated in 25 tables and "networkUuid" duplicated in 16 tables.

These pieces of evidence show that an implicit conceptual model has been used to create the schema. However, only using an implicit model may make the evolution of the schema more difficult, especially when it is done by a team. Note that a Cassandra database may contain tens of tables such as the Minds schema with 65 tables and it may be maintained by more than one developer. Using an explicit conceptual model facilitates the consistency of the model as well as its data quality, especially when working in a team. It also helps the automation of processes such as the determination of the tables where the same modification of data must be performed at the same to maintain the integrity of the data [24].

An explicit conceptual model also helps to avoid faulty implementations of tables that store data related to relationships such as omitting the columns associated to the primary key of an entity if those attributes are not explicitly queried. For example, suppose that a table is created to satisfy a query that retrieves information of attributes that are not keys. If when implementing the table only the query is considered, developers can easily forget to implement additional columns to ensure row uniqueness in the table. If an explicit conceptual model is available, it is less probable that the developer team will make these kinds of mistakes.

TABLE I. INFORMATION SUMMARY OF THE STUDIED PROJECTS

Conceptual model change type	**Case studies**							
	Thingsboard	***Minds***	***Powsybl***	***wireapp***	***COVID-19***	***Blobkeeper***	***Reviews-Service***	***TOTAL***
NUMBER OF TABLES	**30**	**65**	**17**	**72**	**5**	**5**	**6**	**200**
Entity	5	14	7	2	7	0	0	35
Add Entity	4	14	7	2	0	0	0	27
Update primary key of an entity*	1	0	0	0	7	0	0	8
Attribute	3	6	29	6	6	9	2	61
Add non-key attribute	3	5	26	6	6	5	2	53
Remove non-key attribute	0	0	3	0	0	4	0	7
Split non-key attribute*	0	1	0	0	0	0	0	1
Relationships	0	11	21	1	0	0	7	40
Add relationship	0	11	21	1	0	0	6	39
Update cardinality of relationship*	0	0	0	0	0	0	1	1
Total Changes	8	31	57	9	13	9	9	136

C. *Q2: How do conceptual models evolve and how is this evolution reflected in the schema?*

To answer this question, we have studied every change of the schema registered in each commit of the aforementioned projects. As they only contain the database schema, we have manually inferred the conceptual model and determined the equivalent conceptual model change from each schema change, which are displayed TABLE I. We have detected 7 types of these conceptual model changes and some of them are very specific of NoSQL databases and were not detected in the ontology of Noy et al. [5]. These last type of changes are indicated with the symbol '*' in TABLE I.

The most frequently detected changes are additions. In particular, for the change "*Add Entity*" we have detected two possible approaches: 1) the usual creation of a table with a column associated to each conceptual attribute and 2) the more novel definition of a type. This last approach was observed in the project Powsybl for entities "vertex" or "terminalRef", which were implemented as custom types. On the other hand, in the case of the change "*Add Relationship*", the only approach that we have detected is the creation of a new table that stores the instances of the entities that have a relationship. In the case of the changes "*Add Attribute*" the only detected approach is the addition of a column associated to the attribute in at least one table. In the case of the change "*Remove attribute*" the detected approach is the removal of the columns associated to the attribute.

When there is a change of type "*Update primary key of an entity*", we have identified that it implies further changes to the primary key of tables that store information of this entity. For instance, in the project "Thingsboard" we detected that after the addition of the attribute "type" in the entity "entity_view", its associated column was added to the primary key. A similar approach is used for the type of change "*Update cardinality of a relationship*". In what was probably a mistake from the first conceptual model version in the project "Reviews-Service", the relationship "one review for several restaurants" changed to "several reviews for one restaurant". The detected approach was the addition of a column associated to the primary key of 'Review' to the table primary key.

IV. SCHEMA EVOLUTION FRAMEWORK FOR CONCEPTUAL MODEL CHANGES

In this section we describe our framework that determines what actions a developer should perform in the database after a change of the conceptual model. These actions include the required changes in the schema, the data and the application queries in order to maintain consistency between the model, the schema, and the underlying data. In this work we focus on describing the actions to perform after each of the detected changes in the researched projects (see TABLE I.).

This framework takes as inputs: 1) the original conceptual model; 2) the conceptual model change; 3) the original database schema. The outputs are the required changes in the database schemas and, if necessary, the modification of data and changes of queries to be adapted to the new schema.

In each of the following subsections we discuss the conceptual model changes that can be done against each conceptual structure (entity, attribute and relationship). For each change we describe the transformations required in the database schema to maintain its consistency with the new model. Additionally, when required, we describe the modifications in both the data and the database queries of the client application.

A. *Schema transformation on changes to entities*

Entities are the main conceptual structure representing an independent object that can be differentiated to others. Both the attributes and relationships exist to represent characteristics of an entity and their interaction with other entities. In a relational database, entities are usually transformed into a new table, however in NoSQL databases such as Cassandra, such transformations would only happen if there is a query that retrieves all the data about an entity.

*1) **Add entity***

Depending on client application queries, developers will choose between creating a table or creating a custom type in order to reflect the conceptual model change:

- **Table**: Developers will create a table when there is a query that only retrieves the data of the new entity. For each attribute of the new entity, the new table will contain an associated column. The primary key of the table will contain the columns associated to the primary key of the entity.
- **Custom type**: Developers can opt to create a custom type when there is no query that only retrieves the data of the entity. This type will contain a non-key column for each attribute of the entity. Note that a custom type cannot be part of the primary key of a Cassandra table. Therefore this option can only be used if the entity is not part of any relationship that requires its primary key to be part of the primary key of a table that stores this relationship (see section IV.C.1))

2) *Update primary key: Remove attribute from the PK*

Modify the tables where the attribute is part of the primary key. If these tables do not explicitly contain this attribute, then the column associated with the attribute is removed, forcing the creation of a new table.

Data from the original table must be migrated to the new table.

Database operations that retrieve or modify data from the transformed table should be modified to not include the removes column.

3) *Update primary key: Add attribute to the PK*

The required transformations will depend on the stored information, which can be the information of only the entity or a relationship where the entity is involved.

a) Table stores a 1:1 relationship or a 1:n relationship where the entity is the upper bound (cardinality 1) of the relationship:

No change is required in the schema, data or queries.

b) Only the entity, a 1:n relationship where the entity is the lower bound of the relationship (cardinality n) or a n:m relationship

The primary key of the table will contain the column associated to the added attribute. If the primary key of the table does not originally contain this column as primary key, a new table needs to be created with this column plus the original ones.

If a new table is required, the data from the original table must be migrated to the new table.

Database operations that retrieve or modify data from the transformed table must also be modified to include the new column.

B. Schema transformation on changes to attributes

An attribute represents a piece of information that belongs to an entity or a relationship. In the case of the entities, the attributes that are key are used to uniquely identify an instance of an entity.

1) *Add a non-key attribute*

The developer will transform at least one table that stores the primary key of the attribute's entity by adding to this table a column associated to the new attribute.

Database operations that retrieve or modify data from the table should be modified to consider the new column.

2) *Remove a non-key attribute*

The developer will remove in every table the columns associated to the removed attribute. If this column is part of the primary key, a new table shall be created. If the partition key of the original table is compound of only the removed column, then at least one column from the original clustering key will be partition key in the new table.

If a table is created, the data from the original table will be migrated to the new table.

Database operations that retrieve or modify data from the transformed table must remove the reference to the removed columns.

3) *Split non-key attribute in several new ones*

The table will be transformed by replacing the column associated to the original attribute with columns associated to the new attributes.

Developers will migrate data from the original column to the new ones. Because the data is migrated from one column to two or more columns, the developer team must decide which of the resulting columns data is migrated to.

Database operations that modify data of the transformed tables should change the references to the original attribute referencing the attributes created from the split.

C. Schema transformation on changes to relationships

Relationships are the connections between entities. One of the most important components of a relationship is its cardinality (*1:1, 1:n* or *n:m*). A developer must consider cardinalities when implementing a table in order to ensure row uniqueness in the table.

1) *Addition of a relationship*

Developers will create a new table to store the relationship information. The columns that will compose the primary key depend on the cardinality of the relationship:

- 1:1 relationship: Columns associated to the primary key of at least one of the related entities.
- 1:*n* relationship: Columns associated to the primary key of the entity of the lower-bound of the relationship (cardinality n).
- *n:m* relationship: Columns associated to the primary key of both entities.

The table will contain a non-key column for each non-key attribute of the related entities that the developer wants to store (none of them are required). Exceptionally, if one of the entities has been implemented as a custom type, the table will contain the following non-key columns depending on the cardinality relationship:

- 1:1 relationship and 1:*n* relationship where the entity is the upper bound of the relationship (cardinality *1*). The table will contain a non-key column with the data type of the custom type.
- 1:*n* relationship, the entity is a weak entity. The resulting table will contain a collection column of the custom type (set, list or map) [22]

Note that a custom type implementation for an entity cannot be used for creating a table that implements a relationship that requires the primary key of both entities to be part of the table primary key (e.g n:m relationships).

2) *Update cardinality*

a) 1:n to 1:1, n:m to 1:1, n:m to 1:n

No change is required in the schema. Regarding the data, developers should check the data stored in the database as it may contradict the new cardinality. A possible solution is to create a copy of the original table to perform the future operation in it, while conserving the original table with the data contained before the change.

b) 1:1 to 1:n

To ensure row uniqueness, the tables that store the relationship shall have in the primary key the columns associated to the entity from the lower bound (entity with bound n). If the primary key of the table does not originally contain this column as PK, the developer must create a new table adding this column to the PK.

If a new table is required, the data from the original table must be migrated to the new table.

Database operations where the tables is involved should be modified to include the new attribute in the primary key.

c) 1:1 to n:m, 1:n to n:m

To ensure row uniqueness, the table that stores the relationship shall include in its primary key the columns associated with the primary key of both entities. If the primary key of the table is transformed, developers will need to create a new table.

If a new table is created, the data from the original table must be migrated to the new table. Additionally, the database operations where the table is involved should be modified to include the new attribute in the primary key.

V. Conclusions and Future Work

In this work we proposed a framework that determines the modifications in the database schema required to maintain consistency given a change in a conceptual model. This consistency ensures database quality during evolution by assuring that the database schema evolves considering the application requirements. Any loss of consistency may provoke situations where the database loses properties such as row uniqueness in tables due to an incorrect schema definition. Additionally, we have also approached maintaining data quality by describing data migrations that are required to maintain integrity after a schema change.

As a result of having studied several real projects as the basis for our framework, we ensured that the transformations proposed for the database schema are real transformations and not purely theoretical. This framework will help developer teams to address the evolution of the conceptual model, easing tasks related to schema evolution and ensuring that database quality is maintained.

As future work, we want to extend our framework by approaching conceptual model changes that were not detected in the researched projects but were defined in the taxonomy of Noy et al. [5]. We also plan to formalize our framework using models and then automate it using these models in a model transformation language such as ATL [23]. These models will establish relationships between the conceptual model components, the changes performed in them and the target schema components to obtain the modifications to perform in this schema.

Another line of research that we will approach in more detail regards data integrity during schema evolution. When the database schema changes, either after a direct modification or after a change in the conceptual model, the data contained in the database may lose its integrity. We plan to extend our framework so that it automatically manages data to avoid the loss of data integrity when there is a conceptual model change.

As we need a conceptual model for our framework, we also want to address consistency between the conceptual model and the database schema when this schema changes. This would be the opposite direction from what we have studied in this work.

Acknowledgment

This work was supported by the TestBUS project (PID2019-105455GB-C32) and the TESTEAMOS project (MINECO-17-TIN2016-76956-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] C.A., Curino, H.J. Moon, A. Deutsch, A and C. Zaniolo. "Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++.", in *Proceedings of the VLDB Endowment*, 2010, vol. 4, no. 2, pp. 117-128.

[2] M. de Jong, A. van Deursen and A. Cleve. "Zero-downtime SQL database schema evolution for continuous deployment." in *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp 143-152

[3] 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

[4] J. Delplanque, A. Etien, N. Anquetil, S. Ducasse. "Recommendations for Evolving Relational Databases." in *CAiSE 2020-32nd International Conference on Advanced Information Systems Engineering*, 2020, pp 498-514.

[5] N. F. Noy, and M. Klein. "Ontology evolution: Not the same as schema evolution." *Knowledge and information systems*, vol 6, no 4, pp 428-440, 2004

[6] S. Scherzinger, M. Klettke and U. Störl. "Managing schema evolution in NoSQL data stores." 2013. [Online]. Available: *arXiv:1308.0514*

[7] S. Scherzinger, T. Cerqueus and E. Cunha de Almeida. "Controvol: A framework for controlled schema evolution in NoSQL application development." In *2015 IEEE 31st International Conference on Data Engineering*, pp 1464-1467 2015.

[8] U. Störl, D- Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke and S. Scherzinger.. "Curating variational data in application development" in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. pp. 1605-1608, 2018.

[9] 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*.pp-1925-1928, 2019.

[10] A. Hillenbrand, U. Störl, M. Levchenko, S.Nabiyev and M. Klettke. "Towards Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases." in *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. Pp 133-138

[11] A. Chebotko, A. Kashlev, L. Andrey; S. Lu. "A big data modeling methodology for Apache Cassandra" in *2015 IEEE International Congress on Big Data*. pp 238-245, 2015

[12] M. J. Mior, K. Salem, A. Aboulnaga and R. Liu. "NoSE: Schema design for NoSQL applications". *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no 10, pp. 2275-2289, 2017

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

[14] "ThingsBoard". GitHub. https://github.com/thingsboard/thingsboard/blob/416c3fd10e0b58755fcdcc297e3ebd9e4b04c7f1/dao/src/main/resources/cassandra/schema-entities.cql (accesed Sep. 30, 2020)

[15] "Social Network Minds". GitLab. https://gitlab.com/minds/engine/-/blob/master/Core/Provisioner/Provisioners/cassandra-provision.cql (accesed Sep. 30, 2020)

[16] "Powsybl". GitHub. https://github.com/powsybl/powsybl-network-store/blob/master/network-store-server/src/main/resources/iidm.cql (accesed Sep. 30, 2020)

[17] "WireApp". GitHub. https://github.com/wireapp/wire-server/blob/develop/docs/reference/cassandra-schema.cql (accesed Sep. 30, 2020)

[18] "COVID19". GitHub. https://github.com/dilettalagom/COVID19sabd (accesed Sep. 30, 2020)

[19] “Bon-app-etit”. GitHub. https://github.com/bon-app-etit/reviews-service/tree/master (accesed Sep. 30, 2020)

[20] “Blobkeeper”. Github. https://github.com/sherman/blobkeeper/ (accesed Sep. 30, 2020)

[21] DataStax. “Data Modeling Concepts”. https://docs.datastax.com/en/dse/6.8/cql/cql/ddl/dataModelingApproach.html (accesed Sep. 30, 2020)

[22] DataStax. “Creating the set type”. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useSet.html (accesed Sep. 30, 2020)

[23] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev. “ATL: A model transformation tool”. *Science of computer programming*, vol. 72, no 1-2, pp. 31-39, 2008

[24] 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*, vol. 18, no. 6, pp. 257-286