Design document for the Orion language

Alberto Hernández Chillón Diego Sevilla Ruiz Jesús García Molina

{alberto.hernandez1,dsevilla,jmolina}@um.es

Faculty of Computer Science
University of Murcia

Murcia, España, Enero 2020

Enero 2020

Index

Índice de contenido

Why are schema changes needed?

Two main reasons to evolve a database schema

- **Changes** in the functional and non-funcional **requirements**
- Improve the quality without affecting the semantics: **Database refactoring**

How are schema changes performed?

In **relational systems** an explicit schema exists:

- Schema must first be changed (**ALTER** operation)
- Data must be migrated to be compatible to the new schema –> This should be automated
- Some tools provide some kind of automation

How are schema changes performed?

In **NoSQL systems** (when schemas are implicit to data and code):

- No schema should be changed, but schema changes happen!
- Data should be migrated to be compatible to the new (implicit) schema –> This should be automated
- **How can schema evolution be automated if schemas are implicit?**

A schema change language

- Schema change operations can be defined at a **high level of abstraction**
- The language **engine automates the migration of data** to a new schema
- The **agile development** that promotes the use of NoSQL systems is facilitated
- The language **could be integrated** into tools that facilitate database migrations, e.g. Flyway

A strategy to define a schema change language

- **What languages** have been **previously** defined?
- Establish a **data model** for NoSQL schemas
- Define a **taxonomy** of schema changes for the data model
- State **requirements** for the language
- Define the **notation** and **semantics** of the language
- **Implementation** of the language
- **Evaluation** of the language

Changes on a graph

In a graph, changes can be applied on nodes and edges, and they can or not affect the graph structure

- To change properties of an existing node
- To change properties of an existing edge
- To add new nodes or remove existing nodes
- To add new edges or remove existing edges

Taxonomy of Schema changes in NoSQL systems: Considerations (I)

- A taxonomy should only include **Schema change operations**
- The language based on the taxonomy could also include **data migration operations** which could be defined by using operations in the taxonomy + conditions/actions
- **Schema change operations** vs **Migration operations**
  - Example: Add attribute to an entity vs Add attribute to instances satisfying a property
- Schema changes and migration operations can generate new variations of an entity
  - Add attribute -> Old instances could be removed if new and old applications must manage instances with the attribute added
  - Add attribute to instances satisfying a condition -> two variations of the entity should probably be managed

Taxonomy of Schema changes in NoSQL systems: Considerations (II)

- **Move** and **Copy** operations are defined in [1] to move or copy a property from an entity A to another entity B, when A has a reference to B
- Should these operations be part of a taxonomy?
- They seem rather a migration operation. Both involve an **Add attribute**, and **Move** also a **Delete attribute**. Additionally, an initialization of the attribute created is applied
  - **Move** an attribute **attr1** from A to B would be equivalent to **delete attr1** from A and **Add** it to B; and for each instance **ob: B, ob.attr1 = oa.attr1**, being **oa** the entity that references **ob**.
  - **Copy** would be equivalent to move but the attribute is not removed from the source entity

Taxonomy of Schema changes in NoSQL systems: Considerations (III)

In literature [x, y, z] schema changes are classified in several categories:

- **Additive vs Substractive**: In additive changes, new information is added without affecting existing information in the schema, e.g. add attribute; a substractive change involves loss of information, e.g. remove attribute. Some changes are additive and substractive, e.g. rename attribute.
- **Hard vs Soft**: Depending on a change requires or not that instances are moved to a different entity
- **Backwards** compatibility: A change is backwards compatible if a query on instances of the new schema can also be performed on old instances, e.g. remove attribute
- **Forwards** compatibility: A change is forward compatible if a query on instances of a schema can also be performed on instances of the new schema, e.g. add attribute

Taxonomy of Schema changes in NoSQL systems: Considerations (IV)

- **Capacity extensibility**: Following the same idea as in **Additive vs Substractive**, some operations do:
  - Extend capacity: By adding fields
  - Reduce capacity: By removing fields
  - Maintain capacity: By renaming fields
  - Change capacity: By casting fields
- Reference needed
- Operations can be classified in a graphic plot

Invariants

They must be satisfied after a change is performed:

- **Inv 1**: Name uniqueness for entities, attributes of a class and relationships. Usually specific databases do force this invariant
- **Inv 2**: A schema is a connected and directed graph. This is discussable, an example can be shown in which this invariant is not preserved
- **More rules?**

Taxonomy of NoSQL Schema changes

- **Entity**:
  - Add entity, Remove entity, Rename entity, Split entity, Merge entity
- **Attribute**:
  - Add attribute, Remove attribute, Rename attribute, Change the type, Nest attributes
- **Reference**:
  - Add reference, Modify cardinality, Morph reference
  - Change the type of the attribute used: This is not a basic operation, since it is database dependant. In some systems it does not make sense...
- **Aggregation**:
  - Add aggregate, Modify cardinality, Morph aggregation, Unnest aggregation

Some issues to be addressed

- Should **completeness** and **correctness** of the taxonomy be considered?
- Consideration: Completeness can not be achieved, since we cannot think on covering each and every operation defined for each and every database
- Should all the operations be atomic?
- Correctness is achieved on a script if each operation is correct
- A database can be inconsistent during the execution of a script, but it will be consistent by the end of it
- If using evolution blocks, a database should be consistent in between the evolution blocks containing operations

Requirements

- The language should be **platform-independent**
- It should be based on a **data model** that includes relationships: References and aggregations, and structured primitive data types
- It should allow to express schema changes and migration operations
- A migration operation could include a condition
- A script can integrate one or more operations that are applied in atomic way
- Operations scripts are **models** to take advantage of MDE technologies
- **To be completed**

Definition of the language

- For each operation we should define:
  - Syntax concrete or notation
  - Semantics
  - Analyze how the change affects to existing instances
- Semantics should be defined at schema level and database level. Pre and postconditions not too much formal
- It should be used a running example and figures to illustrate the change if is needed

Add entity

Add an entity to the schema

- Two previous invariants should be satisfied
- It is a not atomic schema change operation
- It is additive, soft and forwards compatible
- No database changes are involved

...Syntax example...

Split an entity

Split an entity E1 in two entities E2 and E3 by partitioning its set of properties in two subsets. A frequent split aims to separate a new entity from one existing

- At database level: E1 instances are not removed if split is aimed to separate a new entity.
    - Properties of an entity can be optional
    - It is a not atomic operation

...Syntax example...

Add attribute

Add an attribute to an entity

- Types and default value are optional
- It is a schema change operation –> atomic operation
- It is additive, soft and forwards compatible

...Syntax example...

Move

Moves an attribute (and also relationships?)

- At schema level:
  - **Precondition**: Let mr:MovieRating and m:Movie be, then mr.movie = m.title
  - **Postcondition**: Value is removed from mr and it is added to m as rating
- At database level:
  - **Effect**: For each mr:MovieRating instance mr.value is removed and added to m:Movie such that mr.movie=m.title
- New name is optional
- It is a migration operation –> it is not atomic
- it is additive/substractive and soft

...Syntax example...

Implementation

- Level of automation –> to measure gain of productivity
- Models could be used to check correctness of a migration script

A migration tool for NoSQL databases

- Possible integration of the language in a tool similar to Flyway
- History of changes is stored -> List of migration scripts -> they are models

Research contributions (I)

Regarding previous approaches, our contributions could be as follows:

- A more rich data model which include: entities and properties which can be attributes, references and aggregations
- To define a taxonomy of schema changes
- To build a schema change operation language including all the primitive change included in the taxonomy, and some non-primitive operations which are very frequently used in NoSQL database evolution as move, copy and denormalization. With these operations, migration scripts can be written to express schema change requiring a combination of operations
- To provide an implementation of the language operations, but we only address the automation of the changes required on data

Research contributions (II)

Regarding previous approaches, our contributions could be as follows:

- To propose an agile approach to evolve NoSQL schemas by using our language. Developers work on a development database and they can visualize schema and apply changes on it as textual scritps that include one or more operations. These scripts shoud be moved to the IC server and then they can be accepted or rejected. If they are accepted the production database must be automatically updated
- Scripts are models and we can take advantage of this to build some technologies that facilitate some tasks: use model transformations to export scripts to other NoSQL schema evolution languages, check validity of a script
- Our solution could be part of one tool aimed to automate the NoSQL schema change, which automates the changes on the data and also identifies pieces of code to be modified and suggest changes. In this paper, we will not address the problem of code analysis

Índice de contenido

Use cases and integration

Use cases

- Athena_Schema_1 + Operations in Orion = Athena_Schema_2
- Athena_Schema_2 - Athena_Schema_1 = Operations in Orion

Integration and transformations

- Darwin –> uNoSQLSchema
- uNoSQLSchema <–> Athena
- Code generation for a certain system

Imperative language: For now

Índice de contenido

Darwin command operations

EntityTypes:

- ENTITY ADD *name* (*version*)? (*features*)?
- ENTITY DEL *entityName*
- ENTITY RENAME *entityName* TO *entityName*
- ENTITY SPLIT *entityName* INTO *name1*(*attr1...attrN*), *name2* (*attr1...attrN*)
- ENTITY MERGE *entityName1*, *entityName2* INTO *name* WITH *condition*

Common feature operations:

- ATTR/REF/AGGR DEL *entity((versions)?)* :: *name* : (*type*)? (*version*)?
- ATTR/REF/AGGR RENAME *entity((versions)?)* :: *name* TO *newName*
- ATTR/REF/AGGR COPY *sEntity((versions)?)* :: *name1* TO *tEntity((versions)?)* (:: *name2*)? WITH *condition*
- ATTR/REF/AGGR MOVE *sEntity((versions)?)* :: *name1* TO *tEntity((versions)?)* (:: *name2*)? WITH *condition*

Índice de contenido

Operations in a document-based context

Attributes:

- ATTR ADD *entity((versions)?)* :: *name* : (*type* (*defValue*)?)?
- ATTR CAST *entity((versions)?)* :: *name* : *newType*

References:

- REF ADD (EntityType).(Attribute) TO (EntityType) version?
- REF CARD (NewCardinality)
- REF TO AGGR (Aggregate)

Aggregates:

- AGGR ADD (EntityType) STRUCT
- AGGR CARDINALITY (NewCardinality)
- AGGR DEL (EntityType) version?
- AGGR CHANGE (Ref)

More operations

Variations:

- VAR ADD (Int)?   STRUCT
- VAR DEL (EntityType).(Int)
- VAR RENAME (EntityType).(Int) (NewInt)
- VAR MOVE (EntityType).(Int) (NewEntityType)
- VAR COPY (EntityType).(Int) (NewEntityType)(.Int)?

Reduce functions:

- REDUCE (field1, ...fieldN) => (op.)
- MERGE (field1, ...fieldN) => (op.)
- MAP (field1, ...fieldN) => (op.)

Índice de contenido

Remaining elements for which define operations:

RelationshipTypes:

- Same operations defined for EntityType
- Binding a reference to a specific RelationshipType

Keys:

- Add a Key to a specific attribute
- Drop a Key
- Rename a Key
- Change the binded attribute

Composed references:

- ???