Abstract

Evolutionary development needs to deal with constant changes in requirements, including new or complement and change existent ones. Most of time, to accommodate requirements, changes in database schema are mandatory, in addition to software code modification. This work’s proposal helps to conduct these changes with minor impact in software dependability, providing a structured and trustworthy method to conduct the evolutionary process. A specification for an integration layer, supported by the modularized schema metadata is proposed reducing the need for physical changes to the database, which in turn prevents query loss, thus increasing availability and maintainability.

1. Introduction

Software systems are in constant evolution and most of them manipulate data through a persistence layer supported by a database. Between the evolution cycles new requirements may arise, demanding changes in the database schema. These changes cannot compromise the software’s dependability.

The key process in database design is its definition, which is made in the form of a schema that represents the database metadata. Assessing the impacts of the required changes in all application levels is essential.

An issue in database evolution according to Cleve et al [1] is that the loss of already working queries, defined as query loss, can reach up to 70% per schema version, especially due to the technology impedance mismatch between the application and the database [2].

Our proposal towards schema evolution is to adapt the database modularization technique described in [3] reducing the need for database schema changes and to provide a structured and trustworthy method to conduct the evolution process. We also provide the specification for an integration layer, supported by the modularized schema metadata. Therefore, changes are applied at this layer reducing the need for physical changes, which in turn prevents query loss, thus increasing availability. In this way the maintainability process can be conducted in an easier and more safety way, which contributes to the system integrity.

Nowadays, database evolution is an increasingly important area due to the arising of new incremental software development methods, such as the Agile ones.

2. Database Modularization and Evolution

Database modularization decomposes the database schema into database modules that contain a conceptual view of a subschema along with its related functionalities. The resulting database modules are loosely coupled and autonomous subschemas that are defined by grouping transactions performed by the application in subsystems, thus increasing cohesion and positively impacting further maintainability.

The technique provides a way to conduct the database design in an evolutionary scenario. It narrows down the scope when assessing the required changes and impacts caused by the software evolution. Later the modules interoperability is provided by the integration object layer.

Our work includes the definition of a repository that is populated along with the process and it can be queried to support the evolution process at each cycle also providing an external view to the application of the database modules. Figure 1 shows the classical database design process including the two database modularization phases: “Collection and Analysis of Modularization Requirements” and “Modularization Design”.

The “Collection and Analysis of Modularization Requirements” is responsible for grouping the software systems functionalities in subsystems, according to the transactions performed by the application.

The “Modularization Design” is conducted within the Conceptual Design and is divided into four stages. In the first stage a subschema is define for each subsystem in respect to the elements that it accesses. An overlap between these subschemas can occur, when a database element belongs to more than one subsystem’s subschema. Therefore, the next stage treats the information sharing, where each element is classified considering read or write access performed by each subsystem. Next the elements are grouped according to the subsystems responsible to perform the
writing operations. Later, the intersection operations between these groups define the database modules. An empty intersection originates a database module with the participating elements; whereas a non-empty intersection results in a database module with the intersecting elements or an union among all elements from the groups. Finally the databases modules are encapsulated with the writing (private procedure) and reading transactions (public procedure) and the logical and physical database schemas for each module are created. A graphical representation of a database module is presented in Figure 2.

The integration of the database modules is performed by the integration object layer, which is responsible for several types of conversions, such as: data, data structures, semantic value of data, data validation rules, procedures, interfaces and data managers. In our work it consists of a metadata repository of the database modules that can be queried to check if the already existing database modules support the new requirements. It is also updated to reflect database schema changes.

As shown in Figure 3, the integration object approach in [4] was extended to provide a transparent view of the database modules through an execution engine that obtains information from the metadata repository. The integration object layer functionalities can be automated, by either extending the database management system metadata or through a framework.

3. Conclusion and Future Work

Preserving software integrity through an evolutionary process demands a great deal of effort, especially because they involve two different paradigms: code and database designs. Database modularization is a standardized way to evolutionary database schema design.

A database module holds both the subschema and the subset of related application transactions. Ergo, it improves traceability and contributes in the maintainability process, performed at module level.

Database changes can occur at the integration object metadata level, which increase availability, since less physical schemas changes are required and the already implemented queries (already in use in the operational environment) can be preserved.

Future work includes the development of a framework to implement the integration object layer.

References