Schema Evolution

Zouhaier Brahmia
University of Sfax, Tunisia

Fabio Grandi
University of Bologna, Italy

Barbara Oliboni
University of Verona, Italy

Rafik Bouaziz
University of Sfax, Tunisia

INTRODUCTION

In information systems, not only do data change over time, also database schemata evolve frequently (Sjøberg, 1993). Changing database schema, which is a common but often troublesome task in database administration, occurs for many reasons such as changes in user requirements, compliance to new regulations, addition of new functionalities, or correction of deficiencies in the current schema. Usually, the database administrator changes a database schema through a schema definition language (e.g., SQL).

Two fundamental aspects are involved by schema change: (1) *semantics of change*, which deals with the effects of the change on the schema itself, in order to maintain schema consistency after schema changes, and (2) *change propagation*, which deals with the effects of the change on the underlying data, in order to guarantee data consistency with the modified schema.

For most database applications, changing the schema of the database without loss of existing data is a significant challenge: it is usually a time-consuming and error-prone task which must be done carefully. In the literature (Jensen et al., 1998), *schema evolution* has been defined as the modality for the management of schema changes which relieves database programmers and administrators from this burden, by automatically recovering extant data and possibly adapting them to the new schema. During the last two decades, a lot of theoretical work has been done on schema evolution in both temporal and conventional databases, within the relational and object-oriented settings and more recently for the XML environment, according to the growing adoption of XML as a data modeling language and storage format.

However, this issue remains so far almost unanswered at the practical level: existing commercial systems (i.e., DBMSs like Oracle and DB2, and schema management tools) provide a limited support for schema evolution. Thus, currently, each database administrator uses ad hoc techniques to manage the evolution of a database schema.

The main goal of this article is (1) to present the different research proposals that deal with schema evolution, and (2) to discuss the support of schema evolution in mainstream DBMSs. The rest of this article is organized as follows. The next section gives some background on our subject. In "Current Research in Schema Evolution," we present the different research proposals on schema evolution. "DBMS Support for Schema Evolution" surveys the support of schema evolution in the state of the art of database technology. Finally, future work directions and conclusion are provided.

BACKGROUND

In this section, we illustrate with a simple example the functioning of schema evolution, contrasting it with the lowest level of schema change support that can be embedded in a database, that is the modality of schema modification (Jensen et al., 1998). Assume that we have a relational database that contains only

DOI: 10.4018/978-1-4666-5888-2.ch753

an AUTHOR relation with the attributes ID (primary key), NAME, PHONE, and COUNTRY. The first state of this database is as shown in Figure 1.

The catalogues store information on the schema S1 of the AUTHOR relation. The table AUTHOR contains two tuples for two authors. Then consider the following schema changes:

```
ALTER TABLE AUTHOR
   DROP COLUMN PHONE;
ALTER TABLE AUTHOR
   ADD COLUMN EMAIL CHAR(30);
```

The schema modification technique allows users to effect changes to the database schema, but neither previous schema nor its underlying data are preserved: the old schema is replaced by the new schema, which is initially empty as data populating the old schema are discarded. The effects in our example would be as shown in Figure 2.

The database designer or administrator must restore information concerning authors Aicha and Cristiana in database state S2 by explicitly inserting them as new tuples, through the following SQL statements (executed within the same transaction containing the schema changes or later):

```
INSERT INTO AUTHOR
   VALUES(1, 'Aicha', 'Tunisia',
'aicha@author.tn');
INSERT INTO AUTHOR
   VALUES(2, 'Cristiana', 'Italy',
'cristiana@author.it');
```

Hence, this solution leads to loss of information and obsolescence of applications developed according to the original schema. Although this technique may seem unsuitable, since it is straightforward to implement, it is actually the most widely adopted technique for managing schema changes in existing information systems.

With schema evolution, after a schema change the old schema is replaced by the new one and old extant data, which are still conforming to the new schema, are automatically recovered. In our example, the effects would be as shown in Figure 3.

The new email information could then be introduced through the following SQL statements replacing Nulls with correct values:

```
UPDATE AUTHOR
   SET EMAIL='aicha@author.tn'
      WHERE ID = 1;
UPDATE AUTHOR
   SET EMAIL='cristiana@author.it'
      WHERE ID = 2;
```

The final state of the database is as shown in Figure 4.

Schema evolution can anyway lead to a partial loss of information. For instance, dropping columns means losing the related data. The change of the type of a column could lead to loss of data of this column if the new type is not compatible with the previous type. Furthermore, schema evolution technique does not guarantee continued functioning of existing applications, since programs that use dropped columns

Figure 1.

S1

AUTHOR (ID, NAME, PHONE, COUNTRY)

ID	NAME	PHONE	COUNTRY
1	Aicha	11223344	Tunisia
2	Cristiana	55667788	Italy

Figure 2.

S2

AUTHOR (ID, NAME, COUNTRY, EMAIL)

ID	NAME	COUNTRY	EMAIL
(empty)			