Relational Schema Evolution for Program Independency*

Young-Gook Ra

Department of Electrical and Computer Engineering, University of Seoul, Korea
ygra@uos.ac.kr

Abstract. The database schema is assumed to be stable enough to remain valid even as the modeled environment changes. However, in practice, data models are not nearly as stable as commonly assumed by the database designers. Even though a rich set of schema change operations is provided in current database systems, the users suffer from the problem that schema change usually impacts existing application programs that have been written against the schema. In this paper, we are exploring the possible solutions to overcome this problem of impacts on the application programs. We believe that for continued support of the existing programs on the old schema, the old schema should continue to allow updates and queries, as before. Furthermore, its associated data has to be kept up-to-date. We call this the program independency property of schema change tools. For this property, we devise so-called program independency schema evolution (PISE) methodology. For each of the set of schema change operations in the relational schemas, the overview of the additional code blocks due to the PISE compliance is presented in order to prove the comprehensiveness and soundness of our PISE methodology.

Keywords: schema evolution, program independency, database view, relational database, data model, schema version, database schema, capacity-augmenting schema change, type mismatch, shared database

1 Introduction

Database designers construct a schema with the goal of accurately reflecting the environment modeled by the database system. The resulting schema is assumed to be stable enough to remain valid even as the modeled environment changes. However, in practice, data models are not nearly as stable as commonly assumed by the database designers. Schema changes may include the simple addition/deletion of relation or of an attribute, or the decomposition of a complex relation into several simpler relations, to just name a few. It would relieve much of a database administrator' work if these changes were automatically handled by a dedicated software tool, especially if this tool's functionality would include the verification of integrity constraints and the translation of the stored database structured accordingly to the old schema into the structure of the new schema. Indeed many such tools and methodologies have been proposed [1, 2].

* This work was supported by the University of Seoul, in 2003.

G. Das and V.P. Gulati (Eds.): CIT 2004, LNCS 3356, pp. 273–281, 2004.

Even though a rich set of schema change operations is provided in current database systems, there still remains the more serious problem that schema change usually impacts existing application programs that have been written against the schema. Hence, in current DBMSs, schema updates are more restricted by their impact on existing programs rather than by the power of the supported schema change language. This problem could be partially solved using some schema version systems [3], because they maintain old schema versions and thus allow existing programs to still run against the old versions while new programs can be developed on the desired new schema version. However, in schema version systems, old schema versions are only accessible in consultation mode. This means that old programs would access obsolete data through their schema.

We believe that for continued support of the existing programs on the old schema, the old schema should continue to allow updates and queries, as before. Furthermore, its associated data has to be kept up-to-date. We call this the program independency property of schema change tools. Precisely speaking, an update on tuple values of a schema version (VS2) is visible to another schema version (VS1) if the update values are key values of VS1. If the values are non-key values of VS1, then the update is visible to VS1 as long as the update causes different non-key values of which key values are the same within a version or among versions. If the non-key value update is not propagated to VS1, the values are set to null in VS1.

There are research works that recognized the importance of the program independency property in schema updates in order to continually support existing application programs and they provide solutions [4–6]. Put simply, they basically use procedures to resolve the type mismatch of different versions: they require that such procedures are provided by the users for each occurrence of a schema change. The view approach also has been advocated by several researchers [7, 8] as an alternative solution approach for the resolution of the type match problem. The idea is that database views are to provide multiple perspectives of instances, each of which corresponds to a specific schema version. The view approach has advantages over the procedural one that the views are to provide the difference perspectives, update on derived perspectives are well investigated, and the performance of generating derived perspectives can easily be optimized.

However, Tresch and Sholl [8] have observed that the view based solutions can't support the full range of schema changes. literature [1]. In particular, view mechanisms, being defined to correspond to a stored named query [9], cannot simulate capacity-augmenting schema changes. Bertino [7] and our earlier work [10] have proposed to extend views that can add more schema components in order to assure that the capacity-augmenting changes can be simulated using views. However, our experiments have shown that this is a hard to achieve in a truly efficient manner. We are proposing a more practical solution in this paper. Our approach is neither confined to non-capacity-augmenting changes nor requires a capacity-augmenting view mechanism. This alternative solution, which is called the program independency Schema Evolution (PISE) methodology, is based on the following simple idea. For a non-capacity-augmenting schema

change operation we simply derive a view schema intended by the operation and for a capacity-augmenting schema change operation we propose that (1) schema is directly modified for the additional capacity required by the operation, (2) the original schema is reconstructed as a view based on the modified schema, and (3) the target schema is generated as a view also based on the modified schema.

The next section explains overall of our approach. In the next section, we discuss related works. In the next PISE section, we present schema evolution operators with the illustration of algorithms that achieves the program independency of the operators. In the final section, our work is concluded with a future work.

2 Overview of Our Approach

The PISE methodology assumes that all schemas, through which either interactive human users or application programs interact, are actually views derived from one underlying base schema B as shown in Figure 1 (a). Suppose the existing program *Prog1* runs against the *VS1* schema and the program *Prog2* is being developed and requires new database services. For the new database services, the database administrator initiates the schema transformation T on *VS1* with the expectation that *VS1* would be transformed into the target schema *VS2*. Instead, the desired target schema would be created as a view *VS2* with a newer version number as shown in Figure 1 (a). The PISE methodology that generates this desired schema as a view for both capacity-augmenting and non-capacity-augmenting cases consists of three steps. Each of these steps is detailed in the following.

We first augment the underlying base schema B by the amount of capacity necessary to generate the desired target schema. In case T is non-capacity-augmenting, this step is skipped. This is called the base-schema-change step. In Figure 1 (b), the base schema B is in-place changed into B' by this step, i.e., B disappears. Then we generate the desired target schema (*VS2*) using a view derivation over the enhanced base schema B' as shown in Figure 1 (c). This step is referred to as the target-view-derivation step. Note that the user schema VS1 may become invalid because its source schema B no longer exists, i.e., it has been modified by the previous step. Lastly, we reconstruct the original base schema B as a view defined over the modified base schema B' as shown in Figure 1 (d). This preserves the source schema of the old user schema *VS1*, namely, the original base schema B. Hence, *VS1* now becomes valid again. This is called the original-base-schema-restoration step. This step is again skipped in case T is non-capacity-augmenting, because the base schema has not been changed to begin with for this case. In Figure 1 (d), we can see that the new program *Prog2* can now be developed on its desired new schema version *VS2*, while the old program *Prog1* still continues to run against the same schema VS1 despite the schema transformation T.

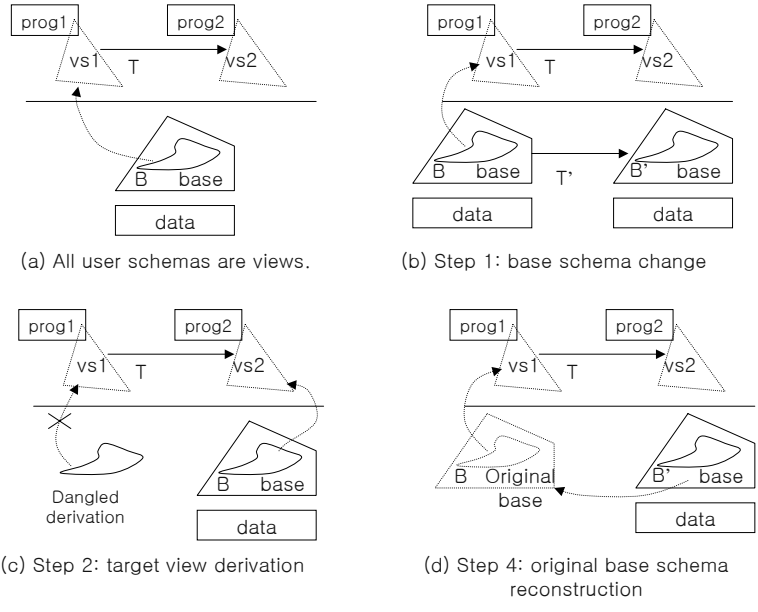

Fig. 1. Overview of our approach.

3 Related Work

The schema change approaches are classified into four categories. The first category corresponds to in-place schema change systems that support the conventional notion of schema change, i.e., in-place changing the shared base schema and propagating the schema change to the instance level. The second category corresponds to schema version systems that support the conventional notion of versioning, i.e., the old schema is stored with its snapshot of the database only for consultation purpose. The third category is named procedural program independent schema change systems that support the notion of program independency, i.e., the application programs on the old schema are continuously supported despite the schema changes, but resolve any type of mismatch between the instances and a schema version by procedures written by a programming language. The fourth category is named view-based schema change systems that also support the program independency notion, but the distinction is that the type mismatch is resolved by implementing each schema version as a view of the underlying base schema while the mismatch is resolved by user/pre-defined procedures in the third category systems. Because our interest is on the program independency of the schema change, we focus on the third and fourth categories of the systems.

The procedural program independent systems include the encore system [4], CLOSQL [6], and Clamen [12]. The encore system allows instances of different type version to be accessed by providing exception handlers for the properties that the types of the instances do not contain. The CLOSQL proposes that

update/backdate functions are provided (either by the user or predefined for some operations) for each attribute which convert the instances from the format in which the instance is stored to the format that an application program expects. In Clamen's scheme, an instance is physically represented as a disjoint union of all versions, while in Zdonik's, an instance is physically represented as an interface type, which is a minimal cover of all versions.

View-based program independent schema change systems include Tresch and Scholl [8], Bertino [7], and Thomas and Shneiderman [13]. Tresch and Scholl advocate views as a suitable mechanism for simulating schema evolution and also state that schema evolution can be simulated using views if the evolution is not capacity-augmenting. Bertino presents a view mechanism and indicates that it can be utilized to simulate schema evolution. Thomas and Shneiderman propose that an old schema is implemented as a network sub-schema. However, the old schema can no longer be changed because this system only supports in-place schema change operators and the old schema is not a base schema but a derived one.

4 Schema Change Operators Using PISE

Our PISE approach covers a comprehensive set of schema change operations that have been proposed in [1]. The set we have investigated includes change-name, add-attribute, delete-attribute, promote-key, demote-key, decompose, compose, partition, merge, export, import, extract, and import-dependency. Out of the operators, the add-attribute, promote-key, decompose, export, and extract operators are capacity-augmenting and the rest are non-capacity-augmenting. As for an representative example, the extract operation is explained. The syntax and semantics of the extract operator are presented and the algorithm that changes the source base schema, derive the target view, and reconstruct the original base schema is illustrated. Besides, for the extract operator, we show that the generated schema has the same behavior as that of a base schema for update operations and the PISE implementation of the extract operation satisfies the program independency principle.

The syntax of the extract operation is *"extract(rel, R(a_1, a_2, ..., a_m, b_1, b_2, ..., b_n)"*. The semantics is to extract non-key attributes a_1, a_2, ..., a_m, b_1, b_2, ..., b_n from the relation *rel* and forms a new relation *R*. The relation *rel* is decreased by the attributes b_1, b_2, ..., b_n.

Theorem 1. The extract operation is capacity-augmenting.

Proof: This operation divide the relation *rel* into the two relations *newRel* and *R* and there could be instances that belong to either of the two relations but not to the relation *rel*. Those could be instances of the relation *R* that do not share the a_1, a_2, ..., a_m attribute values with the instances of the *newRel* or the instances of the relation *newRel* that do not share the a_1, a_2, ..., a_m attribute values with the instances of the *R*.

PISE Implementation. By the PISE methodology, all the relations are defined from the source relations as views. For example, as seen in Figure 2, the operand

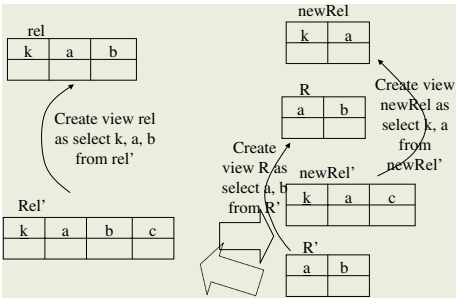

Fig. 2. The PISE implementation of the extract operation.

relation *rel* is initially defined as a view that just projects the *k, a, b* attributes from the relation *rel'*. In the first step, the source relation is physically changed to acquire the augmented schema element. For example, the source relation *rel'* as divided into the two relations *newRel* and *R'* as seen in the lower part of Figure 2. In the second step, target relations are derived from source relations. For example, the target relations *newRel* and *R* are defined as "select k, a from newRel'" and "select a, b from R'" as shown in the right part of Figure 2. In the third step, the source relations before the first step is recovered using view derivations. In the lower part of Figure 2, the dotted block arrow represents the view derivation that accomplishes the third step in this extract operation. In this example, the reconstruction is achieved by joining the *rel'* and *R'* relations based on the attribute *a*.

Theorem 2. The PISE implementation defined above in Figure 2 generates the target relations as expected by the operation semantics. In addition, the expected change also arises in the instance level.

Proof: The new relation *R(a, b)* is extracted from the relation *rel(k, a, b)* and the relation *rel* is changed into the relation *newRel* which is decremented by the attribute *b*. By this, the expected target relations have been generated in the schema level. An instance *inst* of the operand relation *rel* has values v_k, v_a, v_b for the attributes *k, a, b*, respectively. The *inst* instance is derived from the instance *inst'* of the source relation *rel'* by projecting the *k, a, b* attributes. The real instance *inst'* has values of v_k, v_a, v_b, v_c for the attributes *k, a, b, c*, respectively, and is divided into the two tuples (v_k, v_a, v_c) and (v_a, v_b) by the above PISE 1 step and these are transformed into (v_k, v_a) and (v_a, v_b) by the target schema derivation. As a result, the operand instance $inst = (v_k, v_a, v_b)$ is divided into the tuples (v_k, v_a) and (v_a, v_b) and this is the expected instance level change by this operation.

Theorem 3. The target relations generated by the above PISE implementation (Figure 2) shows the same behavior as the base relations for the insert, update, delete operations.

Proof: The insert operation into the relation *newRel* is transformed to the insert operation into the relation *newRel'*. Reversely, this appears as the instance has been inserted into the relation *newRel* because the relation *newRel* is the project

of the relation *newRel'*. The same phenomena holds for the relations R and *R'*. The update and delete operations on the target relations are also propagated to the source relations and this reversely shows the same effect as if the operations are directly performed on the target relations.

Theorem 4. The above PISE implementation of the extract operation (Figure 2) recovers their original relation *rel* both in the schema level and in the instance level. In addition, the recovered schema exhibits the same behavior as the base schema for the insert, update, delete operations.

Proof: The proof omitted because it is similar to those of theorem 3 and 4.

Theorem 5. The above PISE implementation of the extract operation (Figure 2) assures the program independency property.

Proof: In Figure 2, let the schema before the change *VS1* and that after the change *VS2*. First we examine the scope of *VS2* tuples that are accessed from *VS1*. All tuples of the relation *newRel* of *VS2* become those of the relation *rel* of *VS1*. The tuples of the relation R are not instances of the relation *rel* and thus, all the tuples do not have to be accessed from the relation *rel* of *VS1*. Only the tuples that share the a values with the tuples of the *newRel* relation are accessed from the relation *rel*. The b values of such tuples of the relation R are determined by the k values and thus there could be no inconsistencies of b values against k values. Thus, these b values are accessed from the *rel* relation of *VS1*. In reverse, we examine the scope of *VS1* tuples that accessed from *VS2*. The tuples of the *rel* relation of *VS1* all become the instances of the *newRel* relation of *VS2*. Thus, the key attribute k values of the *rel* tuples are all accessed from the *newRel* relation of *VS2*. The a attribute values of the *rel* relation are all accessed as the values of a attribute of the R relation of *VS2*. This is true because the values of the a attribute of the relation *rel* are all the shared values between the a attribute of *newRel* and the a attribute of R. The b values of the *rel* relation of *VS1* can be non-unique against the key-attribute a values of the relation R of *VS2* and they are set to null. Thus, we have showed that the data values of *VS1* (*VS2*) are accessed from *VS2* (*VS1*) under the program independency principle.

The k value creation, deletion or the modification in the relation *rel* of *VS1* is transformed into the same operation in the source relation *newRel'* and in turn the k values for the *newRel* relation of *VS2* is created, deleted or modified. The k value creation, deletion or modification of the *newRel* relation of *VS2* is similarly propagated to *VS1*. When the a value in the *rel* relation of *VS1* is created, deleted or modified, it is transformed into the creation, deletion or the modification of the a values of both source relations *newRel'* and *R'* and results that both a values of the *newRel* and the R relations of *VS2* are create, deleted or modified in *VS2*. Thus, the key k value deletion or modification in *VS2* is observed from *VS1* and in reverse, the key k and a value deletion or modification in *VS1* is observed from *VS2*.

The a value creation, deletion or update in the relation R of *VS2* is transformed into the same operation in both source relations *R'*. This may cause that the created, deleted or the modified *R'* tuple do not share the a values with tuples of *newRel'* any longer and it may result that the a value change is hidden

from the *a* attribute of the *rel* relation of *VS1*. The creation, deletion or modification of the *b* value in the *rel* relation of *VS1* is transformed into the same operation of the *b* value in the source relation*R'*. Then, the *b* change is seen from the *b* attribute of the relation *R* of *VS2*. The *b* value change of the *R* relation of *VS2* is similarly propagated to the corresponding *b* value of the relation *rel* of *VS1*. In summary, the nonkey value change of *VS1* (*VS2*)is propagated to *VS2* (*VS1*) as long as the integrity constraint is not violated. Thus, we have showed that the updates on *VS1* (*VS2*) are visible to *VS2* (*VS1*) under the program independency principle.

5 Conclusion and Future Work

In this paper, we present a solution to the problem of schema evolution affecting existing programs. We believe that other schemas should not be affected by a change on a given schema, even though they share a common base schema and data repository. In addition, old schemas should be preserved and kept to up to date to continue to support for existing programs. We call schema change operations that satisfy the above conditions program independent.

To demonstrate the usefulness and practicality of our PISE approach, we have chosen a comprehensive set of schema change operations, This set covers most of the schema change operations that we found in the literature [1]. Concerning the performance of the PISE system, it is closely tied to the performance of the supporting view system, because the user's schema in the PISE system is actually a view schema. Thus, further investigation is necessary to see how the conventional query optimization techniques can be used to cope with the problem of the performance degradation due to the long derivation chains that could result when the PISE system runs over time.

References

1. Ben Shneiderman and Glenn Thomas, "An architecture for automatic relational database system conversion," *ACM transactions on Database Systems*, Vol. 7, No. 2, pp. 235–257, 1982.
2. Peter McBreien and Alexandra Poulovassilis, "Schema evolution in heterogeneous database architecture, a schema transformation approach," *Conference on Advanced Information Systems Engineering*, pp. 484–499, 2002.
3. Edelweiss and Clesio Saraiva dos Santos, "Dynamic schema evolution managementusing version in temporal object-oriented databases," *International Workshop on Database and Expert Systems Application*, pp. 524–533, 2002.
4. A. H. Skarra and S. B. Zdonik, "The management of changingtypes in object-oriented databases", *Proc.* 1^{st} *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 483–494, 1986.
5. A. Mehta, D. L. Spooner and M. Hardwick, "Resolution of type mismatches in an engineering persistent object system," *Tech. Report, Computer Science Dept., Rensselaer Polytechnic Institute*, 1993.
6. S. Monk and I. Sommerville, "Schema evolution in oodbs using class versioning," *SIGMOD RECORD*, Vol. 22, No. 3, 1993.

7. E. Bertino, "A view mechanism for object-oriented databases," 3^{rd} *International Conference on Extending Database Technology*, pp. 136–151, 1992.
8. M. Tresch and M. H. Scholl, "Schema transformation without database reorganization," *SIGMOD RECORD*, pp. 21–27, 1992.
9. J. Ullman, "Principle of database systems and knowledge-based systems," *Computer Science Press*, Vol. 1, 1988.
10. Y. G. Ra and E. A. Rundensteiner, "A transparent schema evolution system based on object-oriented view technology," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, 1997.
11. W. Kim and H. Chou, "Versions of schema for OODB," *Proc.* 14^{th} *Very large Databases Conference*, pp.149–159, 1988.
12. S. M. Clamen, "Type evolution and instance adaptation," *Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science*, 1992.
13. G. Thomas and B. Shneiderman, "Automatic database system conversion: a transformation language approach to sub-schema implementation," *IEEE Computer Software and Applications Conference*, pp. 80–88, 1980.