Lossless Conditional Schema Evolution

Ole G. Jensen[1] and Michael H. Böhlen[2]

[1] Department of Computer Science, Aalborg University,
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark
guttorm@cs.aau.dk
[2] Faculty of Computer Science, Free University of Bozen-Bolzano,
Dominikanerplatz 3, I-39100 Bolzano, Italy
boehlen@inf.unibz.it

Abstract. Conditional schema changes change the schema of the tuples that satisfy the change condition. When the schema of a relation changes some tuples may no longer fit the current schema. Handling the mismatch between the intended schema of tuples and the recorded schema of tuples is at the core of a DBMS that supports schema evolution. We propose to keep track of schema mismatches at the level of individual tuples, and prove that evolving schemas with conditional schema changes, in contrast to database systems relying on data migration, are lossless when the schema evolves. The lossless property is a precondition for a flexible semantics that allows to correctly answer general queries over evolving schemas. The key challenge is to handle attribute mismatches between the intended and recorded schema in a consistent way. We provide a parametric approach to resolve mismatches according to the needs of the application. We introduce the mismatch extended completed schema (MECS) which records attributes along with their mismatches, and we prove that relations with MECS are lossless.

1 Introduction

Schema evolution occurs when the schema of a populated database is changed. After the schema of a relation has evolved some tuples no longer fit the schema. The mismatch between the intended schema of a tuple and the recorded schema of the tuple, i.e., the schema used to record the tuple in the database, is inherent to evolving schemas. Handling this mismatch is at the very core of a DBMS that supports schema evolution.

The paper considers conditional schema evolution. A *conditional schema change* is an operation that changes the schema of the tuples that satisfy the condition. Conditional schema changes properly subsume regular (i.e., unconditional) schema changes, since it is always possible to have the condition select the entire extent of a relation. The main difference is that conditional schema evolution results in several current and equally important schemas.

As a first step towards a foundation for conditionally evolving database schemata, we present a theoretical framework for conditional evolution at the level of the relational model. The framework is based on evolving schemas consisting of a set of schema segments (akin to versions) where each segment defines the intended schema of a subset of tuples. We show that in contrast to current commercial database systems evolving schemas are lossless models. The lossless property ensures that schema changes can be

P. Atzeni et al. (Eds.): ER 2004, LNCS 3288, pp. 610–623, 2004.

rolled back and that tuple updates and schema changes are orthogonal operations, i.e., we never have to resort to data migration to deal with schema mismatches.

After the schema of a relation has evolved the intended and recorded schema of some tuples are out of sync. The attribute mismatches between the intended and recorded schemas of tuples have to be resolved systematically to get sensible answers to queries. We suggest a parametric approach that resolves attribute mismatches according to the needs of the application.

We propose the *mismatch extended completed schema* (MECS) which records both attributes and their corresponding attribute mismatches. We prove that relations with MECS are lossless evolution models. One of the salient features of such relations is that schema changes can be dealt with as standard tuple updates. We introduce parametric mismatch resolution of relations with MECS and establish the upper bound on its time complexity to be proportional to the size of the relation.

2 Preliminaries

2.1 Evolving Schemas

An *evolving schema*, $E = \{S_1, \ldots, S_n\}$, generalizes a relation schema and is defined as a set of schema segments. A *schema segment*, $S = (\mathcal{A}, P)$, consists of a schema $\mathcal{A}$ and a qualifier P. Throughout, we write $\mathcal{A}_S$ and P_S to directly refer to the schema and qualifier of segment S, respectively. As usual, a *schema*, $\mathcal{A} = \{A_1, \ldots, A_n\}$, is defined as a set of attributes. For the purpose of this paper, no distinction is made between schemas and sets of attributes. A *qualifier* P is either TRUE, FALSE, or a conjunction/disjunction of attribute constraints. An *attribute constraint* is a predicate of the form $A\theta c$ or $\neg(A\theta c)$, where A is an attribute, $\theta \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison predicate, and c is a constant. An evolving schema may have segments with different schemas. Consequently, some tuples may be missing attributes that appear in other segments. In order to evaluate attribute constraints on such tuples, $A\theta c$ is an abbreviation for $\exists v(A/v \in t \wedge v\theta c)$ where t is a tuple. Likewise, $\neg(A\theta c)$ is an abbreviation for $\neg\exists(A/v \in t \wedge v\theta c)$. Note that this implies that the constraints $\neg(A = c)$ and $A \neq c$ are not equivalent.

A tuple t is a set of attribute values where each attribute value is an attribute/value pair: $\{A_1/v_1, \ldots, A_n/v_n\}$. The value must be an element of the domain of the attribute, i.e., if $dom(A)$ denotes the domain of attribute A, then $\forall A, v, t(A/v \in t \Rightarrow v \in dom(A))$. A tuple t *qualifies* for a segment S, $qual(t, S)$, iff t satisfies the qualifier P_S. A tuple satisfies a qualifier, $P(t)$, iff the qualifier is TRUE or the tuple makes the qualifier true under the standard interpretation. If a tuple t qualifies for a segment S in an evolving schema E, then $\mathcal{A}_S$ is the *intended schema* of t, i.e., $\forall t, S, E(S \in E \wedge qual(t, S)) \Rightarrow is(t, E) = \mathcal{A}_S)$. A tuple t *matches* a segment S iff the schema of S and t are identical: $match(t, S)$ iff $\forall A(A \in \mathcal{A}_S \Leftrightarrow \exists v(A/v \in t))$. If a tuple t matches a segment S in the evolving schema E, then $\mathcal{A}_S$ is the *recorded schema* of t, i.e., $\forall t, S, E(S \in E \wedge match(t, S) \Rightarrow rs(t, E) = \mathcal{A}_S)$.

2.2 Conditional Schema Changes

A *conditional schema change* is an operation that changes the set of segments of an evolving schema. The condition determines the tuples that are affected by the schema

change. A *condition* C is either TRUE, FALSE, an attribute constraint, or a conjunction of attribute constraints. For the purpose of this paper we consider two conditional schema changes: adding an attribute, $\alpha(A, E, C)$, and deleting an attribute, $\beta(A, E, C)$. An extended set of schema changes that includes mappings between attributes and a discussion of their completeness can be found elsewhere [9].

$\alpha(A, E, C)$: An attribute A is added to the schemas of all segments that do not already include the attribute. For each such segment two new segments are generated: a segment with a schema that does not include the new attribute and a segment with a schema that includes the new attribute. Segments with a schema that already includes A are not changed.

$\beta(A, E, C)$: The attribute A is deleted from the schemas of all segments that include the attribute. For each such segment two new segments are generated: a segment with a schema that still includes the attribute and a segment with a schema that does not include the attribute. Segments with a schema that does not include A are not changed.

The precise formal definitions of conditional attribute additions and deletions are given in Figure 1.

$$\alpha(A, \emptyset, C) = \emptyset$$
$$\alpha(A, \{(\mathcal{A}, P)\} \cup E, C) = \begin{cases} \{(\mathcal{A}, P)\} \cup \alpha_A(E, C) & \text{iff } A \in \mathcal{A} \\ \{(\mathcal{A} \cup \{A\}, P \wedge C), (\mathcal{A}, P \wedge \neg C)\} \cup \alpha_A(E, C) & \text{iff } A \notin \mathcal{A} \end{cases}$$

$$\beta(A, \emptyset, C) = \emptyset$$
$$\beta(A, \{(\mathcal{A}, P)\} \cup E, C) = \begin{cases} \{(\mathcal{A}, P)\} \cup \beta_A(E, C) & \text{iff } A \notin \mathcal{A} \\ \{(\mathcal{A} \setminus \{A\}, P \wedge C), (\mathcal{A}, P \wedge \neg C)\} \cup \beta_A(E, C) & \text{iff } A \in \mathcal{A} \end{cases}$$

Fig. 1. Adding ($\alpha(A, E, C)$) and Deleting ($\beta(A, E, C)$) Attribute A on Condition C

2.3 Running Example

Assume a schema that models students: $S_{\text{tudent}} = (N_{\text{ame}}, M_{\text{ajor}}, L_{\text{evel}}, G_{\text{rade}})$. The schema requires a name, major, level, and grade for each student. We consider two conditional schema changes.

$$\alpha({}_{\text{s}}U_{\text{pervisor}}, S_{\text{tudent}}, L_{\text{evel}} = grad) \quad (1)$$
$$\beta(G_{\text{rade}}, S_{\text{tudent}}, M_{\text{ajor}} = bio),\ \alpha(C_{\text{redits}}, S_{\text{tudent}}, M_{\text{ajor}} = bio) \quad (2)$$

The first conditional schema change assigns a supervisor to graduate students. Therefore, a ${}_{\text{s}}U_{\text{pervisor}}$ attribute is added to the schema of graduate students: (N_{ame}, M_{ajor}, L_{evel}, G_{rade}, ${}_{\text{s}}U_{\text{pervisor}}$). The schema ($N_{\text{ame}}$, M_{ajor}, L_{evel}, G_{rade}) remains valid for undergraduate students. Note that we are left with *two* current and equally important schemas.

The second conditional schema change requires that a new credit system is used for students with a major in biology. The credit system replaces the old grading system. Thus, biology students need to be recorded with a C_{redits} attribute instead of a G_{rade} attribute. Obviously, biology students may be enrolled as undergraduate or graduate students. Therefore, the schema change applies to the schema of graduate and undergraduate students. This yields an evolving schema with a total of four segments:

$$S_1 = (\{N, M, L, G\}, \neg(L = grad) \wedge \neg(M = bio))$$
$$S_2 = (\{N, M, L, G, U\}, L = grad \wedge \neg(M = bio))$$
$$S_3 = (\{N, M, L, C\}, \neg(L = grad) \wedge M = bio)$$
$$S_4 = (\{N, M, L, C, U\}, L = grad \wedge M = bio)$$

An example instance of the evolving S_{tudent} schema is illustrated in Figure 2. Each tuple is shown as specified by the user, i.e., with values for the attributes of the recorded schema. The intended schema (is) is shown to the right. Note that only tuples t_3 and t_5 match on their recorded and intended schema. For the other three tuples the recorded and intended schema do not match. For example, tuple t_1 has the recorded schema $rs = (N, M, L, G)$ and the intended schema $is = (N, M, L, C)$.

t1 = (N/john,M/bio,L/ugrad, G/9.2) is = (N,M,L,C)
t2 = (N/anne,M/math,L/grad, G/8.7) is = (N,M,L,G,U)
t3 = (N/tom,M/math,L/ugrad, G/5.9) is = (N,M,L,G)
t4 = (N/kim,M/bio,L/grad, G/7.1,U/rick) is = (N,M,L,C,U)
t5 = (N/rita,M/bio,L/grad, C/31,U/mike) is = (N,M,L,C,U)

Fig. 2. An Instance of the S_{tudent} Schema

3 Lossless Schema Evolution

This section develops a framework that can be used to decide whether an evolving database model is *lossless*. Intuitively, a model is lossless iff at each point it can be used to determine the intended schema of tuples. This also means that schema changes can be rolled back. We use the evolving schema from Section 2.1 as a yardstick. Essentially, a lossless model must be able to determine the intended schema with at least the same degree of precision as the evolving schema.

A key issue with evolving schemas is that after several schema changes the schema may no longer permit to correctly determine the intended schema. To see this assume a model that preserves deleted attributes so it is possible to roll back to previous states (this is a common technique that is also used by commercial database systems). In order to replace grades with credits we drop the grade attribute and add a credit attribute. With dropped attributes being preserved we end up with the schema (N, M, L, G, C). This is clearly not the intended schema, and without extra information it is impossible to figure out that the credit attribute is supposed to replace the grade attribute. Thus, the model is lossy.

Evolving schema preserves qualification of tuples. Thus, tuples with an intended schema are guaranteed to also have an intended schema after the schema has evolved. Moreover, this intended schema is unique. The proofs have been omitted due to space consideration, but can be found in full detail in [10].

Lemma 1. *Let E be an evolving schema, t be a tuple, and $\gamma(A, E, C)$ be a conditional schema change where $\gamma \in \{\alpha, \beta\}$. If t qualifies for a segment in E, then t also qualifies for a segment in $\gamma(A, E, C)$, i.e., $\forall E, S, t, \gamma(S \in E \wedge qual(t, S) \Rightarrow \exists S'(S' \in \gamma(A, E, C) \wedge qual(t, S')))$.*

Lemma 2. *Let E and E' be evolving schemas, and $\gamma \in \{\alpha, \beta\}$ be a conditional schema change such that $\gamma(A, E, C) = E'$. Let t be a tuple that qualifies for a single segment in E, then the qualifying segment of t in E' is* unique.

Lemma 1 and 2 guarantee that the evolving schema always uniquely determines the intended schema of each tuple in a relation. This is required to accurately answer general queries over evolving relations (cf. Section 5). Moreover, for each tuple the evolving schema provides an intended schema that is consistent with the schema changes applied, i.e., a conditional schema change only changes the intended schema of a tuple if the tuple satisfies the corresponding condition.

We characterize an evolution model $M = (D, \Gamma, is)$ by a database schema D, a set of schema change operations Γ, and a function $is : t \times D \rightarrow \mathcal{A}$ that given a tuple t and a database schema D determines the intended schema of t. Each schema change operation $\gamma \in \Gamma$ is a function $\gamma : D \times C \rightarrow D$ that given a database schema and a condition applies the conditional schema change to produce a new database schema. An evolution model that associates the same intended schemas with tuples as the evolving schema is *lossless* iff it continues to do so after a conditional schema change has been applied.

Definition 1. (lossless) *Let E be an evolving schema and M be an evolution model. Let t be a tuple and $\gamma(A, E, C)$ be a conditional schema change. M is* lossless *iff*

$$\begin{aligned}&\forall t, \gamma, C, E, M(\\ &\quad M = (D, \Gamma, is') \wedge \gamma \in \Gamma \wedge\\ &\quad is(t, E) = is'(t, D) \Rightarrow is(t, \gamma(A, E, C)) = is'(t, \gamma(A, D, C))\end{aligned}$$

Example 1. Consider the evolution model $M = (D, \Gamma, is)$ based on the completed schema. The completed schema $D = \mathcal{A}$ contains all attributes $\mathcal{A}$ ever introduced, i.e., only attribute additions change the schema. The schema change operations Γ on the completed schema are therefore defined as follows: $\alpha(A, \mathcal{A}, C) = \mathcal{A} \cup \{A\}$ and $\beta(A, \mathcal{A}, C) = \mathcal{A}$. A property of the completed schema is that the intended and actual schema of tuples are synchronized, i.e., $is(t, \mathcal{A}) = \mathcal{A}$. Clearly, the completed schema is not lossless, since attribute deletions do not change the intended schema of tuples as required by the evolving schema.

4 Attribute Mismatches

This section investigates the mismatch between the recorded and intended schema of tuples. We illustrate the four type of mismatches that may occur at the level of individual attributes, and establish the relationship between conditional schema changes and attribute mismatches.

A *history* $H = [\gamma_1(A_1, E, C_1), \ldots, \gamma_n(A_n, E, C_n)]$ is a sequence of conditional schema changes where $\gamma_i \in \{\alpha, \beta\}$. Any schema can be constructed by adding each attribute unconditionally. E.g. $\alpha(G, \alpha(L, \alpha(M, \alpha(N, E_0, \text{TRUE}), \text{TRUE}), \text{TRUE}), \text{TRUE})$ constructs the initial student schema where $E_0 = \{(\{\}, \text{TRUE})\}$ is an evolving schema with a single empty segment. We assume that segments with FALSE qualifiers are removed which will yield an evolving schema with a single segment having the intended

schema. It follows that any evolving schema can be constructed from a history by first constructing the initial schema and then adding in sequence the same conditional schema changes applied to the evolving schema. We write E_H to denote the evolving schema defined by H.

4.1 Mismatches Types

Let H be a history and $\mathcal{A}_H$ be the schema containing all attributes added by schema changes in H. Let t be a tuple with recorded schema $\mathcal{A}_r$ and intended schema $\mathcal{A}_i = is(t, H)$. Only attributes in $\mathcal{A}_H$ can be queried. Since $\mathcal{A}_r \subseteq \mathcal{A}_H$ and $\mathcal{A}_i \subseteq \mathcal{A}_H$ (otherwise t would not be a valid tupel for H) an attribute $A \in \mathcal{A}_H$ causes one of four possible types of mismatches depending on its membership of $\mathcal{A}_r$ and $\mathcal{A}_i$, respectively:

- **No mismatch** ($M_1 : A \in \mathcal{A}_r \wedge A \in \mathcal{A}_i$) The attribute appears in both the recorded and intended schema of a tuple. For example, for Kim (tuple t_4 in Figure 2) there is no mismatch for attributes N, M, E, and U.
- **Not recorded** ($M_2 : A \notin \mathcal{A}_r \wedge A \in \mathcal{A}_i$) The attribute appears only in the intended schema of the tuple. These mismatches occur when schema changes add new attributes. For example, John (tuple t_1 in Figure 2) was not recorded with a value for the C_{redits} attribute, which was added after the tuple was inserted into the database.
- **Not available** ($M_3 : A \in \mathcal{A}_r \wedge A \notin \mathcal{A}_i$) A tuple is recorded with the attribute, but the attribute does not appear in the intended schema of the tuple. Mismatches of this kind are the result of attribute deletions. For example, John (tuple t_1 in Figure 2) is recorded with a grade of 9.2, but according to the intended schema is not supposed to have a grade attribute.
- **Not applicable** ($M_4 : A \notin \mathcal{A}_r \wedge A \notin \mathcal{A}_i$) The attribute neither appears in the recorded nor the intended schema of the tuple. Such mismatches occur e.g. for tuples that do not satisfy the condition for an attribute addition. For example, the C_{redits} and $_sU_{\text{pervisor}}$ attributes are not available to Tom (tuple t_3).

Table 1 shows the attribute mismatches and attribute values (where available) for each tuple in Figure 2. Note that the recorded and intented schema of a tuple can be determined directly from the attribute mismatches of the tuple. The mismatch type of each attribute determines whether that attribute appears in both schemas (M_1 type mismatches), only in the intented schema (M_2 type mismatches), only in the recorded schema (M_3 type mismatches), or in neither schema (M_4 type mismatches).

Table 1. Attribute Mismatches and Attribute Values for the Evolving Student Schema

N_{ame}	M_{ajor}	L_{evel}	G_{rade}	C_{redits}	$_sU_{\text{pervisor}}$
$M_1/john$	M_1/bio	$M_1/ugrad$	$M_3/9.2$	M_2	M_4
$M_1/anne$	$M_1/math$	$M_1/grad$	$M_1/8.7$	M_4	M_2
M_1/tom	$M_1/math$	$M_1/ugrad$	$M_1/5.9$	M_4	M_4
M_1/kim	$M_1/math$	$M_1/grad$	$M_3/7.1$	M_2	$M_1/rick$
$M_1/rita$	M_1/bio	$M_1/grad$	M_4	$M_1/31$	$M_1/mike$

4.2 Mismatches and Conditional Schema Changes

Conditional schema changes change the set of attributes in the intended schema of tuples. Because the intented schema can be determined from the set of attribute mismatches, conditional schema changes also change the attribute mismatches of tuples.

The following two lemmas establish this relationship between attribute mismatches and conditional schema changes.

Let A be an attribute, t be a tuple, H be a history, and M_i be an attribute mismatch with $i \in \{1, \ldots, 4\}$. If the attribute mismatch between t and $is(t, E_H)$ on attribute A is M_i, then $m(A, t, E_H) = M_i$. Note that $m(A, t, \emptyset)$ is M_3 iff $\exists v(A/v \in t)$ otherwise M_4, since the intended schema of t is empty.

Lemma 3. *Let H be a history and t be a valid tuple of E_H with $m(A, t, E_H) = M_i$. Let $\alpha(A, E_H, C)$ be a conditional attribute addition, then*

$$m(A, t, \alpha(A, E_H, C)) = \begin{cases} M_{i-2} & \textit{iff } i \in \{3, 4\} \wedge C(t) \\ M_i & \textit{otherwise} \end{cases}$$

Lemma 4. *Let H be a history and t be a valid tuple of E_H with $m(A, t, E_H) = M_i$. Let $\beta(A, E_H, C)$ be a conditional attribute deletion, then*

$$m(A, t, \beta(A, E_H, C)) = \begin{cases} M_{i+2} & \textit{iff } i \in \{1, 2\} \wedge C(t) \\ M_i & \textit{otherwise} \end{cases}$$

5 Mismatch Resolution

When querying an evolving database, the DBMS has to systematically resolve attribute mismatches. We discuss three sensible and intuitive policies to resolve attribute mismatches at the level of individual attributes (it would be easy to add other policies).

- **Projection:** Resolves the mismatch by using the recorded attribute value. Clearly, this is only possible if the attribute appears in the recorded schema of the tuple. Therefore, projection can only be used to resolve M_1 and M_3 attribute mismatches.
- **Replacement:** Resolves the mismatch by replacing the (missing) attribute value with a specified value.
- **Exclusion:** Resolves the mismatch by excluding the tuple entirely for the purpose of the query.

To illustrate the resolution policies we provide a series of examples.

Example 2. Assume we want to count the number of students who got assigned a supervisor although they are not intended to have one. This means that we need to count the supervisors of tuples with an M_3 mismatch for the supervisor attribute. The query $\pi[cnt(U)](students)$ together with the policies M1:exclusion, M2:exclusion, M3:projection, and M4:exclusion answers the query, since only tuples with M_3 mismatches are included after resolution of the policies. The result is 0 (cf. Table 1).

Example 3. Assume we want to print all the grades that ever have been assigned. This means we also want to see the grades of students who got a grade before they transitioned to the credit system. With the policies M1:projection, M2:exclusion, M3:projection, and M4:exclusion for G_{rade}, the query $\pi[G](students)$ answers the query. After resolution only tuples with an actual value of the G_{rade} attribute are included. The result is $\{9.2, 8.7, 5.9, 7.1\}$.

Example 4. Assume we want to count the number of students who are supposed to have a supervisor but have not got assigned one yet. With the policies: M1:exclusion, M2:replacement, M3:exclusion, and M4:exclusion for ${}_{\text{s}}U_{\text{pervisor}}$, $\pi[cnt(U)](students)$ answers the query. Only tuples with M_2 mismatches are included after resolution of the policies. Since M_2 mismatches indicate that the tuples have no actual value stored, a replacement policy is used to introduce a value that can be counted by the query. The result is 1.

Note the generality of our approach. The key advantage of the proposed resolution approach is that it *decouples* schema definition and querying phases. This means that the above examples do not depend on the specifics of the database schema. For example, the exact reason why someone should (not) have a supervisor does not matter. Queries and resolution policies are conceptual solutions that do not depend on the conditional schema changes. This is a major difference to approaches that exploit the conditions of the schema changes (or other implicit schema information) to formulate special purpose queries to answer the example queries.

6 Mismatch Extended Completed Schema

In this section we introduce the *mismatch extended completed schema* (MECS). To support conditional schema evolution and policies, the DBMS must be able to determine and maintain the recorded and intended schema of tuples. We show that relations with MECSs (referred to as evolving instances) can accomplish this task, and give the algorithms to perform both conditional schema evolution and mismatch resolution. Finally, we show that an evolving instance is a lossless evolution model.

A MECS is a schema $\{A_1, \ldots, A_n, M_1, \ldots, M_n\}$ where for each attribute A_i there is an attribute M_i recording the attribute mismatch of A_i. The domain of M_i indicate the current type of attribute mismatch of A_i: 1 (no mismatch), 2 (not recorded), 3 (not available), and 4 (not applicable). MECSs are used by *evolving instances* to record both the attribute values and the attribute mismatches of tuples:

Definition 2. (evolving instance) *Let H be a history and $\{A_1, \ldots, A_n\}$ be the schema containing all attributes added by schema changes in H. Let I be a relation with the MECS $(A_1, \ldots, A_n, M_1, \ldots, M_n)$ as its schema. If, for all tuples $t \in I$ and all $i \in \{1, \ldots, n\}$, M_i is the attribute mismatch given by $m(A_i, t, E_H)$, then I is an* evolving instance *of H.*

Table 2 shows the evolving instance for the Student example.

In Section 4 we showed that the attribute mismatches of a tuple encode both the recorded and the intended schema of that tuple. Moreover, conditional schema changes

Table 2. The Evolving Student Instance

N	M	L	G	C	U	M_N	M_M	M_L	M_G	M_C	M_U
$john$	bio	$ugrad$	9.2			1	1	1	3	2	4
$anne$	$math$	$grad$	8.7			1	1	1	1	4	2
tom	$math$	$ugrad$	5.9			1	1	1	1	4	4
kim	$math$	$grad$	7.1		$rick$	1	1	1	3	2	1
$rita$	bio	$grad$		31	$mike$	1	1	1	4	1	1

can be applied directly to the attribute mismatches. It is therefore sufficient for a DBMS to maintain the attribute mismatches of tuples instead of their intended and recorded schemas. This is desireable for two reasons. First, to apply policies the DBMS needs to determine the attribute mismatches of tuples anyway. Second, while the mismatch type for an attribute may differ between tuples, each tuple in the same evolving relation has defined a mismatch type for the exact same set of attributes; namely one for each attribute added by conditional schema changes. The attribute mismatches of all tuples can therefore be recorded in a single relation. In contrast, tuples have different recorded and intended schemas with respect to both number and composition of attributes.

By definition, an evolving instance records the tuples and all their attribute mismatches. The missing attribute values in Table 2 occur only for attributes with mismatch type M_2 or M_4. We require that attribute mismatches are resolved before queries are answered, and since the projection policy is only applicable to attribute mismatches of type M_1 and M_3, the missing attributes in an evolving instance are never directly accessed, so their content is irrelevant.

6.1 Applying Conditional Schema Changes to Evolving Instances

Lemma 3 and 4 from Section 4.2 establish the relationship between conditional schema changes and attribute mismatches. The operations for conditional attribute addition and deletion on evolving instances are based on those two lemmas. The main point is that conditional schema changes do not modify the schema of the evolving instance, but rather update the attribute values for the mismatch attributes of tuples satisfying the change condition. Conditional schema changes can therefore be handled by the DBMS in the same way as standard tuple updates.

Figure 3 gives the formal definitions of conditional addition and deletion of an attribute A_i for a tuple t in an evolving instance. Both operations assume that the attribute A_i is in the schema of the evolving instance. Intuitively, deleting an attribute that does not appear in the schema has no effect. Moreover, if we consider the set of all possible attributes $\mathbf{A}$, then any schema is a subset of $\mathbf{A}$. The MECS of an evolving instance I contains exactly all the attributes (and their corresponding mismatches) added by conditional schema changes applied to I. Therefore, only attributes in the MECS of I can appear in the intended schemas of tuples in I. The recorded schemas of tuples in I are similarly bounded. This means that for any attribute $A \in \mathbf{A}$ that does not appear in the MECS of I, the attribute mismatch is M_4 for all tuples in I. According to Figure 3 applying an attribute deletion to tuples with an M_4 mismatch type for that attribute has no effect.

$$\alpha(A_i, t, C) = \begin{cases} (t \setminus \{M_i/v_i\}) \cup \{M_i/v_i - 2\} & \text{iff } t[M_i] \in \{3, 4\} \wedge C(t) \\ t & \text{otherwise} \end{cases}$$

$$\beta(A_i, t, C) = \begin{cases} (t \setminus \{M_i/v_i\}) \cup \{M_i/v_i + 2\} & \text{iff } t[M_i] \in \{1, 2\} \wedge C(t) \\ t & \text{otherwise} \end{cases}$$

Fig. 3. Attribute Addition and Deletion Operations on Tuples in Evolving Instances

When adding an attribute A_{n+1} that does not already appear in the evolving instance I, we have to extend the MECS of I with that attribute and its corresponding mismatch attribute M_{n+1}, before applying the attribute addition:

$$\alpha_{A_{n+1}}(t, C) = \alpha_{A_{n+1}}(t \cup \{A_{n+1}/\omega, M_{n+1}/4\}, C) \text{ iff } A_{n+1}/v_{n+1} \notin t$$

Note that the mismatch type for the new attribute is M_4 (as described in the previous paragraph) and ω can be any value in the domain of the attribute (which value is irrelevant as it will never be used). Operations that add a new column (either empty or with a default value) to an existing and populated table are already supported by commercial DBMSs such as Oracle9.

Theorem 1. *An evolving instance is lossless.*

6.2 Mismatch Resolution for Evolving Instances

The three policies presented in the previous section are functions that resolve attribute mismatches at the level of individual attributes.

Let I be an evolving instance and t be a tuple in I. Let A_i be an attribute in the schema of I and M be a mismatch type, then projection, replacement, and exclusion policies are functions defined as follows:

$$f_{proj}(t, A_i, M) = \begin{cases} undef & \text{iff } t[M_i] \in \{2, 4\} \\ t & \text{otherwise} \end{cases}$$

$$f^c_{repl}(t, A_i, M) = \begin{cases} (t \setminus \{A_i/v_i\}) \cup \{A_i/c\} & \text{iff } t[M_i] = M \\ t & \text{otherwise} \end{cases}$$

$$f_{excl}(t, A_i, M) = \begin{cases} \emptyset & \text{iff } t[M_i] = M \\ t & \text{otherwise} \end{cases}$$

An *attribute policy* $P = (A, [f_1, \ldots, f_n])$ where $f_i \in \{f_{proj}, f^c_{repl}, f_{excl}\}$ specifies a policy for each of the four mismatch types of A, i.e., $f_1, \ldots, f_4$ are used to resolve the attribute mismatches of A of type $M_1, \ldots, M_4$, respectively. Intuitively, an attribute policy specifies how the DBMS resolves all the attribute mismatches of a given attribute. We write P_A as a shorthand for $P = (A, [f_1, \ldots, f_4])$.

We can now define the *mismatch resolution* $\rho[P_A]I$ of an evolving instance I given attribute policy P_A.

Definition 3. (mismatch resolution) *Let I be an evolving instance, A_i be an attribute in the MECS of I, P_{A_i} be the attribute policy of A_i, and t be a tuple in I. Then the* mismatch resolution ρ *is given by:*

$$\rho[P_{A_i}]I = \{t' | t \in I \wedge M = t[M_i] \wedge t' = f_M(t, A_i, M)\}$$

Intuitively, for each tuple in the evolving instance the mismatch resolution considers the mismatch type of attribute A_i and applies the corresponding policy function (f_1 for M_1 mismatches, f_2 for M_2 mismatches, etc.) to derive the resolved tuple t'.

To illustrate mismatch resolution and attribute policies, we review the examples from Section 5.

Example 5. Example 2 resolves mismatches with the attribute policy $P_U = (U, [f_{excl}, f_{excl}, f_{proj}, f_{excl}]$. Table 2 contains no tuples with 3 as their M_U attribute value, so all tuples are excluded by the mismatch resolution. The result to $\pi[cnt(U)]I_{Student}$ is therefore 0.

Example 6. Example 3 uses an attribute policy on G: $P_G = (G, [f_{proj}, f_{excl}, f_{proj}, f_{excl}])$, excluding tuples with M_2 and M_4 mismatches for attribute G. Mismatch resolution results in Table 3. The answer to the query $\pi[G]I_{Student}$ is $\{9.2, 8.7, 5.9, 7.1\}$.

Table 3. Mismatch Resolution for Example 3

N	M	L	G	C	U	M_N	M_M	M_L	M_G	M_C	M_U
$john$	bio	$ugrad$	9.2			1	1	1	3	2	4
$anne$	$math$	$grad$	8.7			1	1	1	1	4	2
tom	$math$	$ugrad$	5.9			1	1	1	1	4	4
kim	$math$	$grad$	7.1		$rick$	1	1	1	3	2	1

Example 7. Example 4 uses the same query as Example 2, but resolves mismatches according to a different attribute policy: $P_U = (U, [f_{excl}, f^{jd}_{repl}, f_{excl}, f_{excl}])$. The result is shown in Table 4. The policy has replaced the missing $_{s}U_{\text{pervisor}}$ attribute value with jd (john doe) for all tuples with M_2 mismatches, and excluded all other tuples. The result of the query is 1.

Table 4. Mismatch Resolution for Example 4

N	M	L	G	C	U	M_N	M_M	M_L	M_G	M_C	M_U
$anne$	$math$	$grad$	8.7		jd	1	1	1	1	4	2

Mismatch resolution can apply any number of attribute policies (on different attributes) to an evolving instance at the same time (cf. Lemma 5).

Lemma 5. *Let $P_{A_1}, \ldots, P_{A_m}$ be policies on different attributes. Let I be an evolving instance and t be a tuple in I. Then:*

$$\rho[P_{A_1}] \ldots \rho[P_{A_m}]I = \{t' | t \in I \wedge t' = \rho_{A_1} \times \ldots \times \rho_{A_m}(\{t\})$$

Lemma 5 also establishes the upper bound on time complexity of mismatch resolution of evolving instances to be proportional to the size of the evolving instance, i.e., $O(|I|)$.

For query purposes, this corresponds to resolving all attribute mismatches and then apply the query to the resolved relation. For applications with fixed policies, a view can be created for the resolved relation. However, for applications using ad hoc attribute policies, we want to minimize the number of attributes and tuples that have to be resolved in order to answer a given query.

7 Related Work

Conditional schema evolution has been investigated in the context of temporal databases, where proposals have been made for the maintenance of schema versions along one [13, 17, 20] or more time dimensions [6]. Because schema change conditions are restricted to one or two time recording attributes the exponential explosion of the number of schemas segments are avoided.

In temporal databases schema evolution has been analyzed in the context of temporal data models [7, 1], and schema changes are applied to explicitly specified versions [22, 5]. This requires an extension to the query language and forces schema semantics (such as missing or inapplicable information) down into attribute values [18, 14]. In order to preserve the convenience of a single global schema for each relation null values have been used [14, 8]. In particular, it is possible to use inapplicable nulls if an attribute does not apply to a specific tuple [2, 3, 11, 12]. This leads to completed schemas [18] with an enriched semantics for null values. The approach does not scale to an ordered sequence of conditional evolution steps with multiple current schemas. It is also insufficient for attribute deletions if we do not want to overwrite (and thus loose) attribute values. In response to this it has been proposed to activate and deactivate attributes rather than to delete them [19].

Unconditional schema evolution has also been investigated in the context of OODBs, where several systems have been proposed. *Orion* [4], CLOSQL [15], and *Encore* [21] all use a versioning approach. Typically, a new version of the object instances is constructed along with a new version of the schema. The *Orion* schema versioning mechanism keeps versions of the whole schema hierarchy instead of the individual classes or types. Every object instance of an old schema can be copied or converted to become an instance of the new schema. The class versioning approach CLOSQL provides update/backdate functions for each attribute in a class to convert instances from the format in which the instance is recorded to the format required by the application. The *Encore* system provides exception handlers for old types to deal with new attributes that are missing from the instances. This allows new applications to access undefined fields of legacy instances. In general, the versioning approach for unconditional schema changes cannot be applied to conditional schema changes, because the number of versions that has to be constructed grows exponentially.

Views have been proposed as an approach to schema evolution in OODBs [23]. [16] propose the Transparent Schema Evolution approach, where schema changes are specified on a view schema rather than the underlying global schema. They provide

algorithms to compute the new view that reflects the semantics of the schema change. The approach allows for schema changes to be applied to a single view without affecting other views, and for the sharing of persistent data used by different views.

8 Summary and Future Research

The paper defines the lossless property for general models of evolving schemas, and show that the solutions offered by current commercial database systems are not lossless. The main problem is to resolve the mismatches between the recorded and intended schemas of tuples systematically. We propose a parametric approach where mismatches are resolved according to the needs of the query. We propose the mismatch extended completed schema (MECS) which records both attributes and their mismatches. We show that relations with MECS (called evolving instances) are lossless, and by exploiting the relationship between attribute mismatches and conditional schema changes we can treat conditional schema changes as tuple updates. Finally, we establish an upper bound on the cost of parametric mismatch resolution of evolving instances.

Ongoing and future work includes the optimization of mismatch resolution. Specifically, we investigate techniques to minimize the amount of data that has to be resolved. By defining mismatch resolution as an algebraic operator, we are currently developing algebraic transformation rules to be used by the query optimizer. Preliminary results indicate that mismatch resolution can be delayed until the recorded attribute values are required by other operators which can substantially reduce the cost of resolution.

References

1. G. Ariav. Temporally Oriented Data Definitions: Managing Schema Evolution in Temporally Oriented Databases. *Data Knowledge Engineering*, 6(6):451–467, 1991.
2. P. Atzeni and V. de Antonellis. *Relational Database Theory*. Benjamin/Cummings, 1993.
3. P. Atzeni and N. M. Morfuni. Functional dependencies in relations with null values. *Information Processing Letters*, 18(4):233–238, 1984.
4. J. Banerjee, W. Kim, H.-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD International Conference on Management of Data*, pages 311–322. ACM Press, 1987.
5. C.D. Castro, F. Grandi, and M.R. Scalas. *On Schema Versioning in Temporal Databases. In: Recent Advances in Temporal Databases*. Springer, 1995.
6. C.D. Castro, F. Grandi, and R.R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
7. J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra based on Lifespans. In *3rd International Conference of Data Engineering, Los Angeles, California, USA, Proceedings*, pages 528–537. IEEE Computer Society Press, 1987.
8. G. Grahne. The Problem of Incomplete Information in Relational Databases. In *Springer LNCS No. 554*, 1991.
9. O. G. Jensen and M. H. Böhlen. Evolving Relations. In *Database Schema Evolution and Meta-Modeling*, volume 9th International Workshop on Foundations of Models and Languages for Data and Objects of *Springer LNCS 2065*, page 115 ff., 2001.
10. O.G. Jensen. *Multi-Dimensional Conditional Schema Evolution in Relational Databases*. PhD thesis, Aalborg University, 2004.

11. A. M. Keller. Set-theoretic problems of null completion in relational databases. *Information Processing Letters*, 22(5):261–265, 1986.
12. N. Lerat and W. Lipski. Nonapplicable Nulls. *Theoretical Computer Science*, 46:67–82, 1986.
13. L.E. McKenzie and R.T. Snodgrass. Schema Evolution and the Relational Algebra. *Information Systems*, 15(2):207–232, 1990.
14. R. van der Meyden. *Logical Approaches to Incomplete Information: a Survey. In: Logics for Databases and Information Systems (chapter 10)*. Kluwer Academic Publishers, 1998.
15. Simon R. Monk and Ian Sommerville. Schema Evolution in OODBs using Class Versioning. *SIGMOD Record*, 22(3):16–22, 1993.
16. Young-Gook Ra and Elke A. Rundensteiner. A transparent object-oriented schema change approach using view evolution. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 165–172. IEEE Computer Society, 1995.
17. J.F. Roddick. SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.
18. J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information Software Technology*, 37(7):383–393, 1995.
19. J.F. Roddick, N.G. Craske, and T.J. Richards. A Taxonomy for Schema Versioning based on the Relational and Entity Relationship Models. In *12th International Conference on Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings*, pages 137–148. Springer-Verlag, 1993.
20. J.F. Roddick and R.T. Snodgrass. *Schema Versioning. In: The TSQL92 Temporal Query Language*. Noewell-MA: Kluwer Academic Publishers, 1995.
21. Andrea H. Skarra and Stanley B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *OOPSLA, 1986, Portland, Oregon, Proceedings*, pages 483–495, 1986.
22. R.T. Snodgrass et al. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1), 1994.
23. Markus Tresch and Marc H. Scholl. Schema transformation without database reorganization. *SIGMOD Record*, 22(1):21–27, 1993.