Current, Legacy, and Invalid Tuples in Conditionally Evolving Databases

Ole Guttorm Jensen and Michael Böhlen*

Department of Computer Science, Aalborg University,
Frederik Bajers Vej 7E, DK–9220 Aalborg Øst, Denmark,
{guttorm | boehlen}@cs.auc.dk

Abstract. After the schema of a relation has evolved some tuples no longer fit the current schema. The mismatch between the schema a tuple is supposed to have and the schema a tuple actually has is inherent to evolving schemas, and is the defining property of legacy tuples. Handling this mismatch is at the very core of a DBMS that supports schema evolution. The paper proposes tuple versioning as a structure for evolving databases that permits conditional schema changes and precisely keeps track of schema mismatches at the level of individual tuples. Together with the change history this allows the DBMS to correctly identify current, legacy, and invalid tuples. We give an algorithm that classifies tuples, in time and space proportional to the length of the change history. We show how tuple versioning supports a flexible semantics needed to accurately answer queries over evolving databases.

1 Introduction

A conditionally evolving database permits schema changes that apply to selected tuples only. We define *conditional schema changes* to selectively evolve the schema of tuples that satisfy a change condition. Different change conditions are orthogonal to each other, and select tuples based on their attribute values. An immediate consequence of conditional schema changes is that there are multiple current versions. After n conditional schema changes there will be up to 2^n current versions. The exponential growth of the number of versions has several consequences. For example, it is infeasible to compute all current versions, let alone the entire evolution history. Similarly, an application cannot be expected to explicitly specify the version to be used when updating or querying the database. The DBMS must handle versions transparently, and all computations must be based on the *change history*, i.e., the sequence of conditional schema changes that has been applied to the original schema. We define *evolving schemas* to define a precise semantics for conditionally evolving databases.

When the schema evolves the *intended schema* of a tuple and the *actual schema* of the tuple, are no longer synchronized. For example, let $E_{\text{mployee}} =$

* This research was supported in part by the Danish Technical Research Council through grant 9700780 and Nykredit, Inc.

T. Yakhno (Ed.): ADVIS 2002, LNCS 2457, pp. 65–82, 2002.

($N_{\text{ame}}, U_{\text{nit}}$) model the name and unit of university employees, and let a conditional schema change add a G_{roup} attribute to the employees in the database unit. Obviously, all tuples that have been recorded before the schema change have no G_{roup} attribute. Because missing attribute values cannot be recovered in the general case, the only accurate solution is to precisely keep track of the mismatch. We introduce *tuple versioning*, where each tuple is associated with two schemas to keep track of actual and intended schema, respectively. In our example, we want to assert ($N_{\text{ame}}, U_{\text{nit}}$) as the actual schema and ($N_{\text{ame}}, U_{\text{nit}}, G_{\text{roup}}$) as the intended schema. Note that this only applies to tuples that model employees in the database unit. Tuples modeling employees in other units will have ($N_{\text{ame}}, U_{\text{nit}}$) as their actual and intended schema.

Tuple versioning eliminates the need to migrate data in response to schema changes. Migrating the data is a common approach to keep the extension and intension synchronized (cf. [17]). Unfortunately, data migration leads to subtle problems [14,19], and a general solution requires an advanced and complex management of multiple null values [19]. To illustrate the problems of data migration consider our initial example and the following queries: 1) Are employee tuples supposed to have a group attribute? 2) Are there employee tuples that are recorded with a group attribute? and 3) Are there employee tuples that are supposed to have a group attribute but are recorded without one? Note that these queries can be issued either before or after the schema change. To answer the first query the intended schema has to be considered. To answer the second query the actual schema has to be considered. To answer the third query both the intended and actual schema have to be considered. Committing to one of the schemas does not allow to accurately answer all queries (unless schema semantics is encoded in the attributes values and query evaluation is changed accordingly).

An evolving relation schema also requires the DBMS to handle legacy tuples. There are several sources for such tuples. If tuples are not migrated to make them fit the new schema, legacy tuples are already stored in the database. Legacy applications, which continue to issue queries and assertions after a schema change has been committed, are another source for legacy tuples. Based on the schema it is natural to distinguish three types of tuples: a *current tuple* is a tuple that fits the schema, a *legacy tuple* is a tuple that at some point in the past fitted the schema, and an *invalid tuple* is a tuple that is neither a current nor a legacy tuple. In the presence of evolving schemas it is one of the key tasks of a DBMS to correctly identify current, legacy, and invalid tuples, and to resolve the mismatch between intended and actual schema for legacy tuples. In this paper we investigate the first issue. The correct classification of tuples is non-trivial and is needed to identify mismatches. We show that besides the current state of the evolving schema it is necessary to consider the history of the evolving schema to correctly identify current, legacy, and invalid tuples. The correct classification of tuples is reduced to path properties in the evolution tree. We give the ClassifyTuple algorithm to classify tuples without materializing the evolution tree. Given an initial schema and a change history the algorithm classifies tuples in time and space proportional to the length of the change history.

The paper proceeds as follows. Section 2 gives an overview of our solution and sets up a running example. Evolving schemas and conditional schema changes are defined in Sections 3 and 4, respectively. In Section 5 the properties of evolving schemas are explored. Section 6 defines change histories and evolution trees. The evolution tree defines the semantics of a change history and can be used to identify legacy tuples. In Section 7 we show how to accurately answer queries over evolving schemas by providing a flexible semantics for resolving attribute mismatches. Section 8 discusses related work. Conclusions and future work are given in Section 9.

2 Overview and Running Example

We use an evolving employee schema for illustration purposes throughout. Detailed definitions follow in the next sections. For simplicity we consider a database with a single relation. The solution is easily generalized and applies to each relation individually. We also restrict the discussion to schema changes. Updates of the data are completely orthogonal. Updates of the data change the attribute values and possibly the actual schema of tuples.

Assume a schema that models university employees:

$$E_{\text{mployee}} = (N_{\text{ame}}, P_{\text{osition}}, U_{\text{nit}}, R_{\text{ank}}).$$

This schema requires a name, position, unit, and rank for each employee. The rank is a number that determines the salary of an employee. As the number of members in the database unit grows it is decided to sub-divide the database unit into groups. We use a *conditional schema change* to change the schema for employees in the database unit to $(N_{\text{ame}}, P_{\text{osition}}, U_{\text{nit}}, G_{\text{roup}}, R_{\text{ank}})$. The schema $(N_{\text{ame}}, P_{\text{osition}}, U_{\text{nit}}, R_{\text{ank}})$ remains valid for all employees in other units. Note that we are left with *two* current and equally important schemas. A *qualifier* is associated with each schema to determine the association between tuples and schemas. A (schema, qualifier)-pair is a *schema segment*.

The second schema change allows full professors to negotiate individual salaries with the university. As a result full professors need to be recorded with a S_{alary} attribute instead of a R_{ank} attribute. Obviously, full professors can be members of any unit. Therefore, the schema change applies to the schema of the database unit as well as to the schema of the other units. This yields an evolving schema with a total of four schema segments, S, S', S'', and S''', as illustrated in Figure 1.

In Figure 1 each tuple is associated with an actual and an intended schema, respectively. The actual schema of a tuple (indicated by the solid line) is the schema that was used when the tuple was added to the database. The intended schema of a tuple (indicated by the dashed line) denotes the schema the tuple is supposed to have according to the initial DDL statement and the subsequent schema changes. For example, tuples t_1 and t_2 in Figure 1 have

$$(N_{\text{ame}}, P_{\text{osition}}, U_{\text{nit}}, R_{\text{ank}})$$

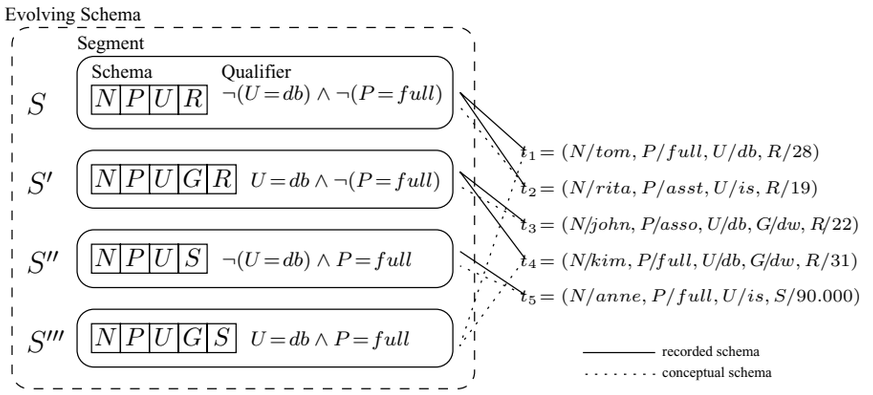


Fig. 1. The Evolving E_{mployee} Schema

as their actual schema because that was the schema that was used when they were asserted. The intended schema of t_1 is $(N_{\text{ame}}, P_{\text{osition}}, U_{\text{nit}}, G_{\text{roup}}, S_{\text{alary}})$ because t_1 models a full professor in the database unit and, according to our schema changes, this means that the tuple is supposed to have a G_{roup} attribute and the R_{ank} attribute should be replaced by a S_{alary} attribute. The intended schema of t_2 is the same as its actual schema because Rita is not in the database unit and is not a full professor. Note that the actual schema of a tuple is never affected by schema changes. In contrast, the intended schema of a tuple changes whenever a conditional schema change applies to the tuple.

3 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 $\mathcal{A}_S$ (P_S) to directly refer to the schema (qualifier) of segment S. As usual, a *schema* is defined as a set of attributes: $\mathcal{A} = \{A_1, \ldots, A_n\}$. A tuple t is a set of attributes where each attribute is a name/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 $dom(A_i)$ denotes the domain of attribute A then $\forall A, v(A/v \in t \Rightarrow v \in dom(A))$.

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. Note that because of attribute deletions some attributes may be missing from the schema. To allow for this, $A\theta c$ is an abbreviation for $\exists v(A/v \in t \wedge v\theta c)$, and $\neg(A\theta c)$ is an abbreviation for $\neg\exists v(A/v \in t \wedge v\theta c)$. Note that this implies that the constraints $\neg(A{=}c)$ and $A{\neq}c$ are not equivalent. A *qualifier* P is either TRUE, FALSE, or a conjunction of attribute constraints.

A tuple t *qualifies* for a segment S, $qual(t, S)$, iff t satisfies the qualifier P_S. A tuple satisfies a qualifier iff the qualifier is TRUE or the tuple satisfies each

attribute constraint in the qualifier. An attribute constraint $\exists v(A/v \in t \wedge v\theta c)$ is satisfied iff the tuple makes the attribute constraint true under the standard interpretation. A tuple t *matches* a segment S iff the schema of S and t are identical: $match(t, S)$ iff $\forall A(A \in \mathcal{A}_S \Leftrightarrow \exists v(A/v \in t))$. A tuple t *fits* a segment S, $fit(t, S)$, iff $qual(t, S)$ and $match(t, S)$. $\mathcal{A}_S$ is the *intended schema* of t iff $qual(t, S)$. $\mathcal{A}_S$ is the *actual schema* of t iff $match(t, S)$

Example 1. We use the E_{mployee} schema from Figure 1 to illustrate these definitions.

- E_{mployee} is an evolving schema with four segments, S, S', S'', and S'''.
- Segment $S' = (\{N, P, U, G, R\}, U{=}db \wedge \neg(P{=}full))$ states that the schema for employees who are working in the database unit and are not full professors is (N, P, U, G, R).
- Tuple t_1 qualifies for S''' but not for any other segment: $\neg qual(t_1, S)$, $\neg qual(t_1, S')$, $\neg qual(t_1, S'')$, and $qual(t_1, S''')$. Thus, the schema of S''', (N, P, U, G, S), is the intended schema of t_1.
- Tuple t_1 matches segment S but does not match any of the other segments: $match(t_1, S)$, $\neg match(t_1, S')$, $\neg match(t_1, S'')$, and $\neg match(t_1, S''')$. Thus, (N, P, U, R), the schema of S, is the actual schema of t_1.
- Tuple t_1 does not fit any segment: $\forall S_i(S_i \in E_{\text{mployee}} \Rightarrow \neg fit(t_1, S_i))$.

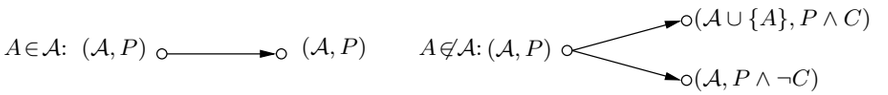
4 Conditional Schema Changes

A *conditional schema change* is an operation that changes the set of segments of an evolving schema. The condition C determines the tuples that are affected by the schema change. A condition is either TRUE, FALSE, or an attribute constraint. For the purpose of this paper we consider two conditional schema changes: adding an attribute, α_A, and deleting an attribute, β_A. An extended set of primitives that includes mappings between attributes and a discussion of their completeness can be found elsewhere [10].

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.

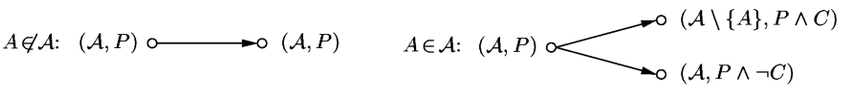
An attribute A is deleted from the schemas of all segments that include the attribute. For each such segment two new segments are generated: a segment with a schema that still includes the attribute and a segment with a schema that does not include the attribute. Segments with a schema that does not include A are not changed.

Example 2. Let $E = \{\ (\{N, P, U, R\}, \text{TRUE})\ \}$ be an evolving schema, and let $\beta_R(E, P{=}full)$ be the conditional schema change that deletes the R_{ank} attribute for full professors. This yields the evolving schema $E' = \{\ (\{N, P, U, R\}, \neg(P{=}full)), (\{N, P, U\}, P{=}full)\ \}$.



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

Fig. 2. Adding Attribute A on Condition C: $\alpha_A(E, C)$



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

Fig. 3. Deleting Attribute A on Condition C: $\beta_A(E, C)$

Note that conditional schema changes properly subsume regular (i.e., unconditional) schema changes. It is always possible to have the condition (TRUE) select the entire extent of a relation.

5 Properties of Evolving Schemas

This section establishes a set of properties for evolving schemas. In the next section these properties are generalized to path properties in evolution trees. Path properties make it possible to correctly classify tuples, and they are used to derive an efficient classification algorithm.

A tuple t is *current* iff it fits a segment in the evolving schema. A tuple t is a *legacy* tuple, iff it is not a current tuple, but at some point in the past (i.e., before some schema changes were applied) was current. If a tuple is neither a current nor a legacy tuple, it is an *invalid* tuple. Note that a conditional schema change may change the status of a current tuple to become a legacy tuple. However, a current or legacy tuple never becomes invalid.

Lemma 1. *An evolving schema cannot identify legacy tuples.*

Proof. The proof is by counterexample. Let $E = \{(\{A_1\}, \text{TRUE})\}$ be an evolving schema, and assume two attribute additions: α_{A_2} and α_{A_3}. Let

$$E_1 = \alpha_{A_3}(\alpha_{A_2}(E, \text{TRUE}), \text{TRUE})$$

and

$$E_2 = \alpha_{A_2}(\alpha_{A_3}(E, \text{TRUE}), \text{TRUE}).$$

Clearly, $E_1 = E_2$. However, the tuple $t = (A_1/c_1, A_2/c_2)$ is a legacy tuple of E_1 but an invalid tuple of E_2.

We show that conditional schema changes preserve matching and qualification, but do not preserve fitting. Thus, tuples that match and qualify for a segment are guaranteed to match and qualify for a segment after the schema has evolved. However, they are not guaranteed to match and qualify for the *same* segment (since schema changes do not preserve fitting).

Lemma 2. *Let E be an evolving schema, t be a tuple, and C be a condition. A conditional schema change Γ is*

1. match-preserving*:*

$$\forall E, S, t, C, \Gamma (S \in E \wedge match(t, S) \Rightarrow \exists S' (S' \in \Gamma(E, C) \wedge match(t, S')))$$

2. qual-preserving*:*

$$\forall E, S, t, C, \Gamma (S \in E \wedge qual(t, S) \Rightarrow \exists S' (S' \in \Gamma(E, C) \wedge qual(t, S')))$$

3. not fit-preserving*:*

$$\forall E, S, t, C, \Gamma (S \in E \wedge fit(t, S) \not\Rightarrow \exists S' (S' \in \Gamma(E, C) \wedge fit(t, S')))$$

Proof. Match-preservation: The definition in Figure 2 guarantees that when adding an attribute to a segment a new segment with the exact same schema is included in the result. Thus, tuple matching is preserved. The same holds for attribute deletion (cf. Figure 3).

Qual-preservation: If P is the qualifier for a segment then an attribute addition or deletion yields either a segment with the qualifier P or two segments with qualifiers $P \wedge C$ and $P \wedge \neg C$, respectively. Clearly, if P holds then either $P \wedge C$ or $P \wedge \neg C$ holds as well.

To show that conditional schema changes are not fit-preserving it is sufficient to give a counterexample. Assume an evolving schema with one segment: $S_1 = (\{A_1\}, \text{TRUE})$, and a tuple $t = (A_1/v)$ that fits S_1. Adding attribute A_2 on condition TRUE yields an evolving schema with two segments: $S_2 = (\{A_1\}, \neg\text{TRUE})$ and $S_3 = (\{A_1, A_2\}, \text{TRUE})$. Clearly, t fits neither segment because it does not qualify for S_2 and does not match S_3.

Lemma 3. *Let E and E' be evolving schemas, C be a condition, and Γ be a conditional schema change such that $\Gamma(E, C) = E'$. Let t be a tuple that fits a single segment in E.*

1. *The qualifying segment of t in E' is unique.*
2. *The matching segment of t in E' may not be unique.*

Proof. To prove that the qualifying segment is unique we show that 1) if t does not qualify for a segment S then t cannot qualify for any segment that results from applying Γ to S, and 2) if t qualifies for a segment S then the application of Γ results in exactly one segment that t qualifies for. Note that if P is the

qualifier for a segment, then the application of Γ yields either a segment with the qualifier P or two segments with qualifiers $P \wedge C$ and $P \wedge \neg C$, respectively. It follows that if P does not hold then neither does $P \wedge C$ nor $P \wedge \neg C$, and if P holds then so does either $P \wedge C$ or $P \wedge \neg C$, but not both.

To prove that the matching segment is not unique assume an evolving schema with two segments: $S = (\{A_1\}, C_1)$ and $S' = (\{A_1, A_2\}, \neg C_1)$, and a tuple t matching S'. The addition of A_2 on condition C_2 yields three segments:

$$\alpha_{A_2}(\{S, S'\}, C_2) = \{(\{A_1, A_2\}, C_1 \wedge C_2), (\{A_1\}, C_1 \wedge \neg C_2), (\{A_1, A_2\}, \neg C_1).$$

Clearly, the matching segment of t is no longer unique.

6 Histories of Evolving Schemas

In the previous section we saw that an evolving schema cannot be used to identify legacy tuples. This section shows that we also have to consider the change history of an evolving schema to correctly classify legacy tuples.

6.1 Change Histories and Evolution Trees

Definition 1. (change history) *Let $E = \{S\}$ be an evolving schema with a single segment, and $[\Gamma_1, \ldots, \Gamma_n]$ be a sequence of conditional schema changes. $H = (E, [\Gamma_1, \ldots, \Gamma_n])$ is a* change history *for E.*

The semantics of a change history is defined in terms of an evolution tree. An evolution tree is a binary tree (we assume an evolving schema with only one segment in the beginning) that models the current state of an evolving schema and keeps track of all prior states.

Definition 2. (evolution tree) *Let $H = (E, [\Gamma_1, \ldots, \Gamma_n])$ be a change history with change conditions C_i. $T_E = (\mathcal{S}^*, E^*)$ is the* evolution tree *of E, iff*

$$\mathcal{S}^* = \bigcup_{i=0}^{n} E_i \text{ , where } E_i = \Gamma_i(E_{i-1}, C_i)$$
$$E^* = \{e(S, S') \mid S \in E_{i-1} \wedge E' = \Gamma_i(\{S\}, C_i) \wedge E' \neq \{S\} \wedge S' \in E'\}$$

$\mathcal{S}^*$ denotes the vertices and E^* the edges in the evolution tree. Each vertex is a segment. An edge, $e(S, S')$, relates parent segment S to child segment S'. The leaf segments of an evolution tree define the current segments, i.e., the evolving schema (cf. Figure 1).

Figure 4 illustrates the evolution tree for the change history of the E_{mployee} schema. Segment S_0 with the initial schema (N, P, U, R) and qualifier TRUE is the root of the evolution tree. The first schema change sub-divides the database unit into groups. This splits segment S_0 into segments S_x and S_y. The second schema change replaces the R_{ank} attribute with a S_{alary} attribute for full professors. This change leads to the four segments shown at the bottom of the tree. They define the current state of the evolving E_{mployee} schema.

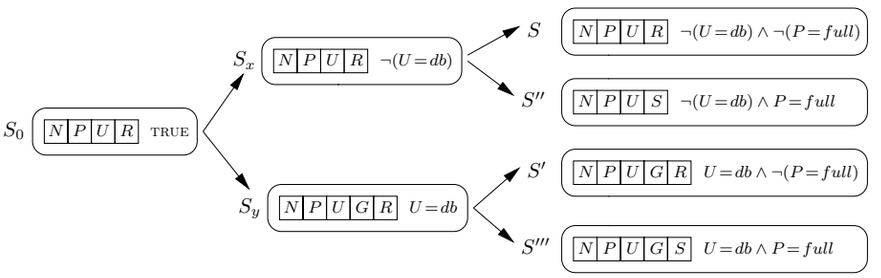


Fig. 4. The E_{employee} History and its Evolution Tree.

6.2 Path Properties

A *path* $p = [S_1, \ldots, S_k]$ in an evolution tree T_E is a sequence of vertices in T_E such that S_i is the parent of S_{i+1} for $1 \leq i < k$. We generalize the properties of an evolving schema to path properties. Path properties allow us to identify legacy tuples and provide the basis for an efficient algorithm.

Definition 3. (qualifying path) *Let $T_E = (\mathcal{S}^*, E^*)$ be an evolution tree, $p = [S_1, \ldots, S_n]$ be a non-empty path in T_E, and t be a tuple. Path p is a* qualifying path *for tuple t, $p_Q(t, p, T_E)$, iff*

$$\forall S((S \in S^* \wedge S \in p) \Rightarrow qual(t, S)) \wedge$$
$$\forall S((S \in S^* \wedge S \notin p \wedge (e(S, S_1) \in E^* \vee e(S_n, S) \in E^*)) \Rightarrow \neg qual(t, S))$$

Thus, a qualifying path is a maximal path of segments a tuple qualifies for. The first line requires that the tuple qualifies for all segments in the path and the second line guarantees that the path is maximal.

Lemma 4. *Let $T_E = (\mathcal{S}^*, E^*)$ be an evolution tree and t be a tuple. The qualifying path of t is unique:*

$$\forall t, p, p', T_E(p_Q(t, p, T_E) \wedge p_Q(t, p', T_E) \Rightarrow p = p').$$

Proof. Follows by induction from Lemma 2 (qual-preservation) and Lemma 3 (unique qualifying segment).

Lemma 5. *Let T_E be an evolution tree and t be a tuple. The qualifying path of t extends from the root to a leaf:*

$$\forall t, T_E, S_1, \ldots, S_n(p_Q(t, [S_1, \ldots, S_n], T_E) \Rightarrow root(S_1, T_E) \wedge leaf(S_n, T_E)).$$

Proof. On the one hand, qualification is propagated to the leaves because of qual-preservation (Lemma 2). On the other hand, if the qualifier of a parent segment is P then the qualifier of the two child segments is $P \wedge C$ and $P \wedge \neg C$, respectively. Obviously, if a tuple satisfies $P \wedge C$ (or $P \wedge \neg C$) it also satisfies P. Thus the qualifying path always extends to the root of the evolution tree.

Definition 4. (matching path) *Let $T_E = (\mathcal{S}^*, E^*)$ be an evolution tree, $p = [S_1, \ldots, S_n]$ be a non-empty path in T_E, and t be a tuple. Path p is a* matching path*, $p_M(t, p, T_E)$, iff*

$$\forall S((S \in S^* \wedge S \in p) \Rightarrow match(t, S)) \wedge$$
$$\forall S((S \in S^* \wedge S \notin p \wedge (e(S, S_1) \in E^* \vee e(S_n, S) \in E^*)) \Rightarrow \neg match(t, S))$$

In other words, a matching path for a tuple is a maximal path of segments the tuple matches.

Lemma 6. *Let T_E be an evolution tree and t be a tuple. Each matching path of t includes a leaf:*

$$\forall t, T_E, S_1, \ldots, S_n(p_M(t, T_E, [S_1, \ldots, S_n]) \Rightarrow leaf(S_n, T_E)).$$

Proof. By induction from Lemma 2. If t matches a non-leaf segment then t matches one of the children of that segment (match-preservation). Note that if a tuple matches the schema of a segment it does not necessarily match the schema of the parent segment (cf. Figure 4). Thus, the root may not be included in the matching path.

A matching path is not necessarily unique. Different sequences of schema changes may lead to the same schema in different branches of the evolution tree (cf. Lemma 3). For example, assume that the sub-division of units into groups is also applied to the information systems unit. This means we also add the G_{roup} attribute on condition $U = is$. The conditional schema change applies to S and S'' in Figure 4 and results in segments with the same schema as S' and S''', respectively.

Qualifying and matching paths can be used to identify current, legacy, and invalid tuples. Consider the three evolution trees in Figure 5. The qualifying path is indicated by the thin line from the root to a leaf, whereas the matching path is indicated by the bold line.

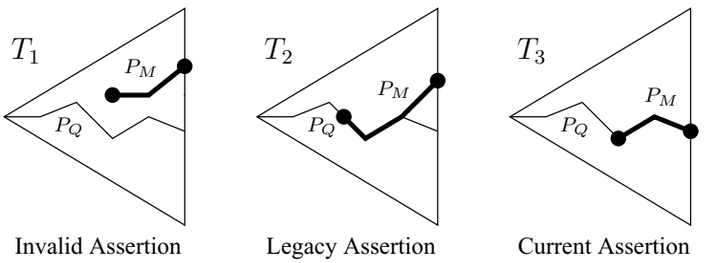


Fig. 5. Possible Relationships between Qualifying and Matching Paths

In T_1 the qualifying and matching paths do not overlap. This means that, at no point in the evolution of the evolving schema, there was a segment that

t matches and qualifies for. Thus, t does not fit any segment in the evolution tree and is therefore invalid. In T_2 the matching and qualifying paths partially overlap. This means that t fits all segments that are part of both the matching and the qualifying path. Because t does not fit a leaf node it is a legacy tuples. In T_3 tuple t also fits a leaf, which makes it a current tuple.

Theorem 1. *Let $T_E = (\mathcal{S}^*, E^*)$ be an evolution tree. Tuple t is* invalid *iff the qualifying and matching path of t do not share any segment, i.e.,*

$$\forall t, T_E, p, p'((p_Q(t, T_E, p) \wedge p_M(t, T_E, p') \wedge \neg\exists S(S \in p \wedge S \in p')) \\ \Leftrightarrow \neg\exists S(S \in \mathcal{S}^* \wedge \mathit{fit}(t, S))).$$

Proof. Tuple t is invalid iff t does not fit any segment S in the evolution tree. From $fit(t, S) \Leftrightarrow qual(t, S) \wedge match(t, S)$ and the fact that the qualifying path of t is unique (Lemma 4), it follows that t is invalid iff no matching path of t shares a segment with the qualifying path of t.

Theorem 1 leads to an algorithm for tuple classification. The ClassifyTuple algorithm in Figure 6 takes as input a tuple and a change history and determines the class of the tuple. The complexity of the algorithm is linear in the length of the history.

```
ClassifyTuple(t, H) → {C_current, L_egacy, I_nvalid}
   Input:
      tuple t
      history H = ({S_0}, [Γ_1, ..., Γ_n])
   Output:
      assertion C_current, L_egacy, or I_nvalid
   Method:
      if ¬qual(t, S_0) return I_nvalid
      let i:=1
      let S:=S_0
      let overlap:=FALSE
      while i ≤ n do
         S:={S'|S' ∈ Γ_i({S}, C_i) ∧ qual(t, S')}
         overlap:=overlap ∨ match(t, S)
         i:=i+1
      end while
      if match(t, S) return C_urrent
      else if overlap return L_egacy
      else return I_nvalid
```

Fig. 6. The ClassifyTuple Algorithm

The algorithm unfolds the qualifying path of the given tuple and evolution tree by applying the conditional schema changes in sequence to the unique qualifying segment. By following the qualifying path and checking for overlap with

a matching path, the algorithm can determine whether the tuple fitted a prior version of the evolving schema. If the tuple matches the current qualifying segment, it is a current tuple. Otherwise, an overlap with a matching path indicates a legacy tuple, and if no overlap was found the tuple is invalid.

7 Attribute Mismatches

This section investigates the mismatch between the actual and intended schema of tuples. We illustrate the four kinds of mismatches that may occur at the level of individual attributes, and use a series of examples to argue for the need of a parametric approach to resolve mismatches according to the need of the application.

7.1 Mismatch Types

Given an attribute A and the intended schema $\mathcal{A}_c$ and actual schema $\mathcal{A}_r$ of a tuple t there are four possible types of attribute mismatches:

Table 1. The Four Types of Attribute Mismatches

	$A \in \mathcal{A}_r$	$A \notin \mathcal{A}_r$
$A \in \mathcal{A}_c$	M_1 (No mismatch)	M_2 (Not recorded)
$A \notin \mathcal{A}_c$	M_3 (Not available)	M_4 (Not applicable)

- **No mismatch** (M_1) The attribute appears in both the intended and actual schema of a tuple. For example, for Tom (tuple t_1 in Figure 1) there is no mismatch on attributes N, P, and U.
- **Not recorded** (M_2) The attribute appears only in the intended schema of the tuple. These mismatches occur when schema changes add new attributes. For example, Tom was not recorded with a G_{roup} or a S_{alary} attribute, which are attributes that were added after the tuple was inserted into the database.
- **Not available** (M_3) A tuple is recorded with the attribute, but it does not appear in the intended schema of the tuple. Mismatches of this kind are caused by attribute deletions. E.g., Tom has a rank of 28, but according to its intended schema is not supposed to have this attribute.
- **Not applicable** (M_4) The attribute appears in neither the actual nor the intended schema of a tuple. Such mismatches occur for tuples that do not satisfy the condition for an attribute addition. For example, the G_{roup} and S_{alary} attributes are not applicable to Rita.

Table 2 shows all attribute mismatches for each employee tuple in Figure 1.

Table 2. Attribute Mismatches of the Evolving Employee Schema

Name	Position	Unit	Group	Rank	Salary
M_1/tom	$M_1/full$	M_1/db	M_2	$M_3/28$	M_2
$M_1/rita$	$M_1/asst$	M_1/is	M_4	$M_1/19$	M_4
$M_1/john$	$M_1/asso$	M_1/db	M_1/dw	$M_1/22$	M_4
M_1/kim	$M_1/full$	M_1/db	M_1/dw	$M_3/31$	M_2
$M_1/anne$	$M_1/full$	M_1/is	M_4	M_4	$M_1/90.000$

7.2 Mismatch Resolution

When querying an evolving database, the DBMS has to systematically resolve attribute mismatches. We discuss three sensible and intuitive resolution strategies (it would be easy to add other strategies).

- **Projection:** Resolves the mismatch by using the stored attribute value. Clearly, this is only possible if the attribute appears in the actual 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 strategies we provide a series of examples.

Example 3. Assume we want to see all the ranks ever recorded for employees. This means that we also want to see the ranks of employees who had a rank before they were allowed to negotiate individual salaries. With the resolutions M1:projection, M2:exclusion, M3:projection, and M4:exclusion, the query $\pi[R](employees)$ answers the query. The result is $\{28, 19, 22, 31\}$.

Example 4. Assume we want to count the number of employees recorded with a salary but who are not supposed to have one. This means that we have to count the number of tuples with an M_3 mismatch for the S_{alary} attribute. The query $\pi[cnt(S)](employees)$ with the resolutions M1:exclusion, M2:exclusion, M3:projection, and M4:exclusion answers the query. The result is 0 (cf. Table 2).

Example 5. Assume we want to count the number of employees who are allowed to negotiate individual salaries. This means that we want to count employees with either M_1 or M_2 mismatches for the S_{alary} attribute. With the resolutions M1:projection, M2:replacement, M3:exclusion, and M4:exclusion, the query $\pi[cnt(S)](employees)$ answers the query. The result is 3.

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) be in a group does not matter. Queries and resolution strategies are conceptual solutions that do not depend on the conditional schema changes. This is a major difference to approaches that exploit conditions of the schema changes (or other implicit schema information) to formulate special purpose queries to answer the example queries.

The examples illustrate that it is well possible to apply different resolution strategies to the exact same query yielding different results. The resolutions provides the semantics of the query. Therefore, an approach that resolves mismatches when the schema is changed (it has a fixed resolution strategy) has to make compromises with respect to the queries that it supports. We propose a parametric approach where attribute mismatches are resolved according to the need of the application.

The above queries cannot be answered correctly with a fixed semantics for resolving attribute mismatches when a schema is changed. A flexible and context-dependent semantics is required to correctly answer queries in the presence of schema evolution. Tuple versioning supports such semantics in a natural way.

8 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 [13,18,21] or more time dimensions [7]. Because schema change conditions are restricted to the time the exponential explosion of the number of schemas segments are avoided. The reason is that the qualifier of a segment only contains predicates on the timestamp attribute, thus effectively defining an interval. Any schema change with a condition based on a point in time will therefore fall within exactly one interval, splitting it into two intervals. So, while several schema segments may change due to a schema change, the *number* of segments will only increase by one. As shown in this paper, schema changes with conditions on different attributes are orthogonal and result in an exponential growth of the number of segments. The effect is that existing techniques, such as query processing strategies that require a user to specify a target schema version, do not apply to conditional schema changes.

Unconditional schema evolution has receive much attention by both the temporal and object-oriented community (cf. [17,19] for an overview and an annotated bibliography).

In temporal databases schema evolution has been analyzed in the context of temporal data models [8,1], and schema changes are applied to explicitly specified versions [23,6]. This requires an extension to the query language and forces schema semantics (such as missing or inapplicable information) down into attribute values [19,14]. In order to preserve the convenience of a single global schema for each relation null values have been used [14,9]. In particular, it is possible to use inapplicable nulls if an attribute does not apply to a specific

tuple [2,3,11,12,25,26]. This leads to completed schemas [19] 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 [20]. Also, note that the completed schema by itself, i.e., the unification of all attributes ever to appear in the schema, is insufficient, because the change history will be lost. This makes the correct tuple classification impossible.

Temporal schema evolution proposes two different ways of coordinating schemas and instances. In synchronous management data can only be updated through the schema with the same temporal pertinence. This constraint is lifted in asynchronous management, where data can be updated through any schema. To illustrate synchronous and asynchronous management assume a schema $S = (A, B, T)$, where T is a timestamp attribute. At $t = 3$ attribute B is dropped and attribute C is added. This yields the schema $S' = (A, C, T)$. Consider the following four tuple assertions: $t_1 = (a, b, 2)$, $t_2 = (a, c, 4)$, $t_3 = (a, b, 5)$, and $t_4 = (a, c, 1)$.

Under synchronous management t_1 is asserted to S, because its timestamp is 2 and S was the current schema until time 3. Similarly, t_2 is asserted to S'. Both t_3 and t_4 are invalid assertions under synchronous management, since their attributes correspond to the one schema and their timestamps indicate the other schema. However, note that t_3 is a legacy assertion, i.e., it was a valid assertion to S before the schema change was applied. Thus, under synchronous management only assertions to the current schema are valid, and legacy assertions are rejected.

In asynchronous management t_1 and t_3 are asserted to S while t_2 and t_4 are asserted to S'. In effect, the constraint based on the timestamp attribute is disregarded and the assertion is based solely on matching the tuples with the schema. While asynchronous management supports both current and legacy assertions it also permits the assertion of invalid tuples. For example, the assertion of t_4 is not a legacy assertion because it does not match S, and is not a current assertion because according to the schema change only tuples with a timestamp attribute larger than 3 have schema S'.

Unconditional schema evolution has also been investigated in the context of OODBs, where several systems have been proposed. *Orion* [4], CLOSQL [15], and *Encore* [22] 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 [5,24]. Ra and Rundersteiner [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. Since a view has to be selected for each schema, applying conditional schema changes result in an exponential number of views. Clearly, this is not tractable. New techniques, such as the ClassifyTuple algorithm proposed in this paper, that rely on the history of schema changes rather than the result of the conditional schema changes are required to ensure tractable support.

9 Conclusions and Future Research

In this paper we introduce conditional schema changes that can be applied to a subset of the extension of a relation. Conditional schema changes result in evolving schemas with several current schemas. We present tuple versioning to precisely keep track of schema mismatches at the level of individual tuples. Tuple versioning eliminates the need to migrate the data in response to schema changes.

A key task of the DBMS is to correctly identify current, legacy, and invalid tuples. We show that this cannot be done solely based on the current evolving schema. The change history is needed to correctly identify current, legacy, and invalid tuples. Finally, we give an algorithm, which, given a tuple and a change history, determines the classification of that tuple. The algorithm exploits the properties of change histories to provide a time and space complexity proportional to the length of the history.

Ongoing work includes processing queries over evolving schemas. We are investigating a parameterized semantics for query processing based on the strategy for resolving attribute mismatches presented in Section 7. Part of this research focuses on the problem of view evolution where both the underlying relations and the view itself are subject to evolution.

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. Elisa Bertino. A view mechanism for object-oriented databases. In Alain Pirotte, Claude Delobel, and Georg Gottlob, editors, *Advances in Database Technology - EDBT'92, 3rd International Conference on Extending Database Technology, Vienna, Austria, March 23-27, 1992, Proceedings*, volume 580 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 1992.
6. C.D. Castro, F. Grandi, and M.R. Scalas. *On Schema Versioning in Temporal Databases. In: Recent Advances in Temporal Databases*. Springer, 1995.
7. C.D. Castro, F. Grandi, and R.R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
8. 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.
9. G. Grahne. The Problem of Incomplete Information in Relational Databases. In *Springer LNCS No. 554*, 1991.
10. 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.
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. Schema Evolution in Database Systems an Annotated Bibliography. *ACM SIGMOD Record*, 21(4):35–40, 1992.
18. J.F. Roddick. SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.
19. J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information Software Technology*, 37(7):383–393, 1995.
20. 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.
21. J.F. Roddick and R.T. Snodgrass. *Schema Versioning. In: The TSQL92 Temporal Query Language*. Noewell-MA: Kluwer Academic Publishers, 1995.
22. 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.

23. R.T. Snodgrass et al. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1), 1994.
24. Markus Tresch and Marc H. Scholl. Schema transformation without database reorganization. *SIGMOD Record*, 22(1):21–27, 1993.
25. Y. Vassiliou. Null values in Database Management: A Denotational Semantics Approach. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 162–169, 1979.
26. Y. Vassiliou. Functional Dependencies and Incomplete Information. In *International Conference on Very Large Databases*, pages 260–269, 1980.