Chapter 6
Recent Advances in Schema and Ontology Evolution

Michael Hartung, James Terwilliger, and Erhard Rahm

Abstract Schema evolution is the increasingly important ability to adapt deployed schemas to changing requirements. Effective support for schema evolution is challenging since schema changes may have to be propagated, correctly and efficiently, to instance data and dependent schemas, mappings, or applications. We introduce the major requirements for effective schema and ontology evolution, including support for a rich set of change operations, simplicity of change specification, evolution transparency (e.g., by providing and maintaining views or schema versions), automated generation of evolution mappings, and predictable instance migration that minimizes data loss and manual intervention. We then give an overview about the current state of the art and recent research results for the evolution of relational schemas, XML schemas, and ontologies. For numerous approaches, we outline how and to what degree they meet the introduced requirements.

1 Introduction

Schema evolution is the ability to change deployed schemas, i.e., metadata structures formally describing complex artifacts such as databases, messages, application programs, or workflows. Typical schemas thus include relational database schemas, conceptual ER or UML models, ontologies, XML schemas, software interfaces, and workflow specifications. Obviously, the need for schema evolution occurs very often in order to deal with new or changed requirements, to correct deficiencies in the current schemas, to cope with new insights in a domain, or to migrate to a new platform.

M. Hartung (✉) and E. Rahm
University of Leipzig, Ritterstraße 26, 04109 Leipzig, Germany
e-mail: hartung@informatik.uni-leipzig.de, rahm@informatik.uni-leipzig.de

J. Terwilliger
Microsoft Research, Redmond, WA, USA
e-mail: James.Terwilliger@microsoft.com

Z. Bellahsene et al. (eds.), *Schema Matching and Mapping*, Data-Centric Systems and Applications, DOI 10.1007/978-3-642-16518-4_6,
© Springer-Verlag Berlin Heidelberg 2011

Effective support for schema evolution is challenging since schema changes may have to be propagated, correctly and efficiently, to instance data, views, applications, and other dependent system components. Ideally, dealing with these changes should require little manual work and system unavailability. For instance, changes to a database schema S should be propagated to instance data and views defined on S with minimal human intervention. On the other hand, without sufficient support schema evolution is difficult and time-consuming to perform and may break running applications. Therefore, necessary schema changes may be performed too late or not at all resulting in systems that do not adequately meet requirements.

Schema evolution has been an active research area for a long time and it is increasingly supported in commercial systems. The need for powerful schema evolution has been increasing. One reason is that the widespread use of XML, web services, and ontologies has led to new schema types and usage scenarios of schemas for which schema evolution must be supported. The main goals of this survey chapter are as follows:

- To introduce requirements for schema evolution support.
- To provide an overview about the current state of the art and recent research results on schema evolution in three areas: relational database schemas, XML schemas, and ontologies. For each kind of schema, we outline how and to what degree the introduced requirements are served by existing approaches.

While we cover more than 20 recent implementations and proposals, there are many more approaches that can be evaluated in a similar way than we do in this chapter. We refer the reader to the online bibliography on schema evolution under http://se-pubs.dbs.uni-leipzig.de (Rahm and Bernstein 2006) for additional related work. Book chapter 7 (Fagin et al. 2011) complements our paper by focusing on recent work on mapping composition and inversion that support the evolution of schema mappings.

In Sect. 2, we introduce the main requirements for effective schema and ontology evolution. Sections 3 and 4 deal with the evolution of relational database schemas and of XML schemas, respectively. In Sect. 5, we outline proposed approaches for ontology evolution and conclude in Sect. 6.

2 Schema Evolution Requirements

Changes to schemas and ontologies affect the instances described by these metadata structures as well as other dependent system components. Figure 6.1 illustrates some of these dependencies for the evolution of database schemas that are always tightly connected with the instances of the database. So when the schema S of a database with instances D, described by S, is changed to schema S′ the instances must be adapted accordingly, e.g., to reflect changed data types or added and deleted structures in S′. We assume that schema changes from S to S′ are described by a so-called *evolution mapping* (e.g., a set of incremental changes or a higher-level abstraction).

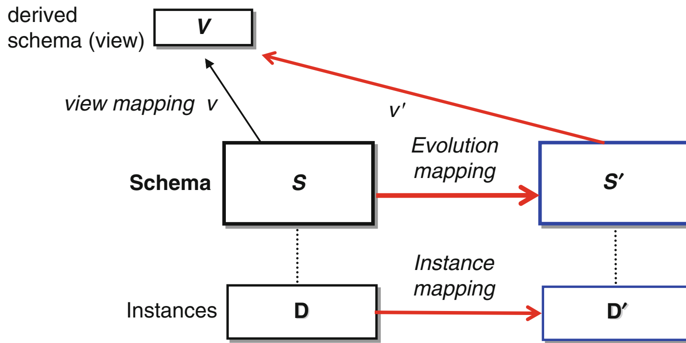


Fig. 6.1 Schema evolution scenario

Similarly, instance changes/migration can be specified by an instance mapping, e.g., a sequence of SQL operations. A main requirement for database schema evolution is thus to propagate the schema changes to the instances, i.e., to derive and execute an instance mapping correctly and efficiently implementing the changes specified in the evolution mapping. Changes to schema S can also affect all other usages of S, in particular the applications using S or other schemas and views related to S. Schema changes may thus have to be propagated to dependent schemas and mappings. To avoid the costly adaptation of applications they should be isolated from schema changes as much as possible, e.g., by the provision of stable schema versions or views. For example, applications using view V remain unaffected by the change from S to S′ if the view schema V can be preserved, e.g., by adapting view mapping v to v′ (Fig. 6.1). There are similar evolution requirements for XML schemas and ontologies, although they are less tightly connected to instance data and have different usage forms than database schemas as we will see (e.g., XML schemas describing web service interfaces; ontologies may only provide a controlled vocabulary).

In the following sections, we discuss in detail general and more specific desiderata and requirements for effective schema and ontology evolution. These requirements are then used in the subsequent sections to review and compare existing and proposed evolution approaches.

We see the following general desiderata for a powerful schema evolution support:

– *Completeness*: There should be support for a rich set of schema changes and their correct and efficient propagation to instance data and dependent schemas.
– *Minimal user intervention*: To the degree possible, ensure that the schema evolution description is the only input to the system and that other artifacts co-evolve automatically.
– *Transparency*: Schema evolution should result into minimal or no degradation of availability or performance of the changed system. Furthermore, applications and other schema consumers should largely be isolated from the changes, e.g., by support for backward compatibility, versioning, or views.

The general desiderata are hard to meet and imply support for a series of more specific, interrelated features:

– *Rich set of simple and complex changes*: Simple changes refer to the addition, modification, or deletion of individual schema constructs (e.g., tables and attributes of relational databases), while complex changes refer to multiple such constructs (e.g., merging or partitioning tables) and may be equivalent to multiple simple changes. There are two main ways to specify such changes and both should ideally be supported. The straightforward approach is to explicitly specify schema modification statements to incrementally update a schema. Alternatively, one can provide the evolved schema, thereby providing an implicit specification of the changes compared to the old schema. This approach is attractive since it is easy to use and since the updated schema version may contain several changes to apply together.
– *Backward compatibility*: For transparency reasons, schema changes should minimally impact schema consumers and applications. We therefore require support for *backward compatibility* meaning that applications/queries of schema S should continue to work with the changed schema S′. This requires that schema changes do not result in an information loss but preserve or extent the so-called information capacity of schemas (Miller et al. 1994). Changes that are potentially lossy (e.g., deletes) should therefore be either avoided or limited to safe cases, i.e., to schema elements that have not yet been used. As we see, the view concept and schema versioning in combination with schema mappings are main approaches to support backward compatibility.
– *Mapping support*: To automatically propagate schema changes to instances and dependent or related schemas, it is necessary to describe the evolution itself as well as schema dependencies (such as view mappings) by high-level, declarative schema mappings. In the simplest case, the *evolution mapping* between the original schema S and the evolved schema S′ consists of the set of incremental changes specified by the schema developer. In case the changes are specified by providing the evolved schema S′, the evolution mapping between S and S′ still needs to be determined. Ideally, this mapping is (semi-)automatically determined by a so-called *Diff(erence) computation* that can be based on schema matching (Rahm and Bernstein 2001; Rahm 2011) but also has to take into account the (added/deleted) schema elements that exist in only one of the two schemas.

There are different possibilities to represent evolution mappings and other schema mappings. A high flexibility and expressive power is achieved by using different kinds of logical and algebraic mapping expressions that have been the focus of a substantial amount of theoretical research (Cate and Kolaitis 2010). The mapping representation should at least be expressive enough to enable the semi-automatic generation of corresponding instance (data migration) mappings. Further mapping desiderata include the ability to support high-level operations such as composition and inversion of mappings (Bernstein 2003) that support the evolution *(adaptation) of mappings* after schema changes. In the example of Fig. 6.1, such operations can

be used to derive the changed view mapping v' by composing the inverse of the evolution mapping with the view mapping v.

- *Automatic instance migration*: Instances of a changed schema or ontology should automatically be migrated to comply with the specified changes. This may be achieved by executing an instance-level mapping (e.g., in SQL or XQuery) derived from the evolution mapping. Database schema evolution also requires the adaptation of affected index structures and storage options (e.g., clustering or partitioning), which should be performed without reducing the availability of the database (online reorganization). There are different options to perform such instance and storage structure updates: either in place or on a copy of the original data. The copy approach is conceptually simpler and keeping the original data simplifies undoing an erroneous evolution. On the other hand, copying is inherently slow for a large amount of data most of which are likely unaffected by the schema change. Furthermore, data migration can be performed eagerly (expensive, but fast availability of changes) or lazily. Instance migration should be undoable if anything goes wrong, which can be achieved by running it under transactional control.
- *Propagation of schema changes to related mappings and schemas*: Schemas are frequently related to other schemas (by mappings) so that schema changes may have to be propagated to these related schemas. This should be performed in a largely automatic manner supporting a maximum of backward compatibility. Important use cases of this general requirement include the maintenance of views, integrated (global) schemas, and conceptual schemas. As discussed view schemas may be kept stable for information-preserving changes by adapting the view mapping according to a schema change; deleted or added schema components on the other hand may also require the adaptation of views. Data integration architectures typically rely on mappings between source schemas and a global target (mediator or warehouse) schema and possibly between source schemas and a shared ontology. Again, some schema changes (e.g., renames) may be covered by only adapting the mappings, while other changes such as the provision of new information in a source schema may have to be propagated to the global schema. Finally, interrelating database schemas with their conceptual abstractions, e.g., in UML or the entity/relationship (ER) model, require evolution support. Changes in the UML or ER model should thus be consistently propagated to the database schema and vice versa (reverse engineering).
- *Versioning support*: Supporting different explicit versions for schemas and ontologies and possibly for their associated instances supports evolution transparency. This is because schema changes are reflected in new versions leaving former versions that are in use in a stable state. Different versioning approaches are feasible, e.g., whether only a sequence of versions is supported or whether one can derive different versions in parallel and merge them later on. For full evolution transparency, it is desirable to not only support backward compatibility (applications/queries of the old schema version S can also use S') but also *forward compatibility* between schema versions S and S', i.e., applications of S' can also use S.

– *Powerful schema evolution infrastructure*: The comprehensive support for schema evolution discussed before requires a set of powerful and easily usable tools, in particular to determine the impact of intended changes, to specify incremental changes, to determine Diff evolution mappings, and to perform the specified changes on the schemas, instances, mappings, and related schemas.

3 Relational Schema Evolution Approaches

By far, the most predominantly used model for storing data is the relational model. One foundation of relations is a coupling between instances and schema, where all instances follow a strict regular pattern; the homogeneous nature of relational instances is a foundational premise of nearly every advantage that relational systems provide, including query optimization and efficient physical design. As a consequence, whenever the logical scheme for a table changes, all instances must follow suit. Similarly, whenever the set of constraints on a database changes, the set of instances must be validated against the new set of constraints, and if any violations are found, either the validations must be resolved in some way or, more commonly, the schema change is rejected.

An additional complication is the usually tight coupling between applications and relational databases or at least between the data access tier and the database. Because the SQL query language is statically typed, application queries and business logic applied to query results are tightly coupled to the database schema. Consequently, after a database schema is upgraded to a new version, multiple applications may still attempt access to that database concurrently. The primary built-in support provided by SQL for such schema changes is *external schemas*, known more commonly as views. When a new version of an application has different data requirements, one has several options. First, one can create views to support the new application version leaving existing structures intact for older versions. Or, one can adapt the existing schema for the new application and maintain existing views to provide backward compatibility for existing applications. In both cases, the views may be virtual, in which case they are subject to the stringent rules governing updatable views, or they may be materialized, in which case the application versions are essentially communicating with different database versions. However, the different views have no semantic relationship and no intrinsic notion of schema version, and thus no clean interoperability.

For the rest of this section, we first consider the current state of the art in relational database systems regarding their support for schema evolution. We examine their language, tool, and scenario support. We then consider recent research revelations in support for relational schema evolution. Finally, we use Table 6.1 to summarize the schema evolution support of the considered approaches w.r.t. requirements introduced in Sect. 2.

Table 6.1 Characteristics of systems for relational schema evolution

	Oracle	Microsoft SQL Server	IBM DB2	Panta Rhei, PRISM Curino et al. (2008)	HECATAEUS Papastefanatos et al. (2008, 2010)	DB-MAIN/MeDEA Hick and Hainaut (2006), Domínguez et al. (2008)
Description/focus of work	Commercial relational database system	Commercial relational database system	Commercial relational database system	Evolution mapping language allowing multiple versions of applications to run concurrently	Propagation of evolution primitives between dependent database objects	Conceptual modeling interface, allowing model-driven evolution
Changes						
(1) Richnless (simple, complex)	(1) Only simple changes (SQL DDL)	(1) Only simple changes (SQL DDL)	(1) Only simple changes (SQL DDL)	(1) Simple and complex changes (table merging/ partitioning)	(1) Simple changes (SQL DDL). Create statement annotated with propagation policies	(1) Simple changes (add/modify/drop entity type, relationship, attribute, etc.)
(2) Specification (incremental, new schema)	(2) Incremental or new schema (table redefinition)	(2) Incremental	(2) Incremental	(2) Incremental	(2) Incremental	(2) Incremental
Evolution mapping						
(1) Representation	(1) Column mapping between old and new single table or set of incremental changes	(1) Set of incremental changes (SQL DDL)	(1) Set of incremental changes (SQL DDL)	(1) Formal logic and SQL mappings, both forward and backward	(1) None beyond standard DDL execution	(1) Set of incremental changes based on conceptual model

(*Continued*)

Table 6.1 (Continued)

	Oracle	Microsoft SQL Server	IBM DB2	Panta Rhei, PRISM Curino et al. (2008)	HECATAEUS Papastefanatos et al. (2008, 2010)	DB-MAIN/MeDEA Hick and Hainaut (2006), Domínguez et al. (2008)
(2) DIFF computation	(2) Comparison functionality for two schema versions (diff expressed in SQL DDL)	(2)–	(2)–	(2)–	(2)–	(2)–
Update propagation						
(1) Instances	(1) Non-transactional, instances migrate if no other objects depend on changed table	(1) Transactional DDL, automatic instance translation for objects with no dependents	(1) Transactional DDL, automatic instance translation for objects with no dependents	(1)–	(1)–	(1) Translation of instance data by a extract-transform-load workflow
(2) Dependent schemas	(2)–	(2)–	(2)–	(2) Query rewriting when possible, notification if queries invalid because of lossy evolution	(2) Policies determine how and when to automatically update dependent objects such as views and queries	(2)–

Versioning support	Editions support multiple versions of nonpersistent objects, with triggers, etc., for interacting with tables	Only internal versioning during instance migration	Only internal versioning during instance migration	Generates views to support both forward and backward compatibility, when formally possible	–	–
Infrastructure/GUI	Oracle change management pack – compares schemas, bundles changes, predicts errors	SSMS creates change scripts from designer changes. DAC packs support schema diff and in-place instance migration for packaged databases	Optim data studio administrator – bundles changes, predicts evolution errors	Web-based tool for demonstrating query rewriting and view generation	–	GUI-based tools used to capture user actions while modifying schemas

3.1 Commercial Relational Systems

Relational database systems, both open-source and proprietary, rely on the DDL statements from SQL (CREATE, DROP, and ALTER) to perform schema evolution, though the exact dialect may vary from system to system (Türker 2000). So, to add an integer-valued column C to a table T, one uses the following syntax:

```
ALTER TABLE T ADD COLUMN C int;
```

Other changes are differently specified in commercial DBMS. For instance, renaming a table in Oracle is performed using the following syntax:

```
ALTER TABLE foo RENAME TO bar;
```

SQL Server uses a stored procedure for that particular change:

```
sp_rename 'foo', 'bar', 'TABLE';
```

Schema evolution primitives in the SQL language and in commercial DBMS are atomic in nature. Unless there is a proprietary extension to the language, each statement describes a simple change to a schema. For instance, individual tables may be added or dropped, individual columns may be added or dropped from a table, and individual constraints may be added or dropped. Additionally, individual properties of a single object may be changed; so, one can rename a column, table, or constraint; one can change individual properties of columns, such as their maximum length or precision; and one can change the data type of a column under the condition that the conversion of data from the old type to the new type can be done implicitly.

However, one cannot specify more complex, compound tasks such as horizontal or vertical splitting or merging of tables in commercial DBMS. Such actions may be accomplished as a sequence of atomic actions – a horizontal split, for instance, may be represented as creating each of the destination tables, copying rows to their new tables, and then dropping the old table. Using this piecemeal approach is always possible; however, it loses the original intent that treats the partition action as a single action with its own properties and semantics, including knowing that horizontal merge is its inverse.

The approach that has been taken by and large by commercial vendors is to include at most a few small features in the DBMS itself and then provide robust tooling that operates above the database. One feature that is fairly common across systems is *transactional DDL*; CREATE, ALTER, and DROP statements can be bundled inside transactions and undone via a rollback. A consequence of this feature is that multiple versions of schemas at a time may be maintained for each table and potentially for rows within the table for concurrent access. Even though multiple versions of schema may exist internally within the engine, there is still only a single version available to the application. PostgreSQL, SQL Server, and DB2 all support this feature; in Oracle, DDL statements implicitly mark a transaction boundary and run independently.

Commercial systems automatically perform update propagation for the simple changes they support. Simple actions, such as column addition, deletion, or type

changes, can frequently be performed while also migrating existing instance data (provided new columns are allowed to be null). Furthermore, online reorganization is increasingly supported to avoid server downtime for update propagation. So, for instance, DB2 offers a feature where renaming or adjusting the type of a column does not require any downtime to complete, and existing applications can still access data in the table mid-evolution (IBM 2009b). The transactional DDL and internal versioning features also promote high server uptime, as alterations may be made lazily after the transaction has completed while allowing running applications access to existing data.

On the other hand, there is little support to propagate schema changes to dependent schema objects, such as views, foreign keys, and indexes. When one alters a table, either the dependent objects must themselves be manually altered in some way, or the alteration must be aborted. The latter approach takes the majority of the time. For instance, SQL Server aborts any attempt to alter a column if it is part of any index, unless the alteration is within strict limits – namely, the alteration is a widening of a text or binary column. Dropped columns simply cannot participate in any index. DB2 has similar restrictions; Oracle *invalidates* dependent objects like views so that they must be revalidated on next use and fails to execute them if they do not compile against the new schema version.

Commercial DBMS do not support abstract schema mappings but only SQL for specifying view mappings and evolution mappings. There is no support for multiple explicit schema and database versions. Once the DDL statements of an evolution step have been executed, the previous version of the evolved objects is gone. There is no support for applications that were developed against previous versions of the database.

For the rest of this subsection, we will focus on vendor-specific features that go above and beyond the standard DDL capabilities for schema evolution.

Oracle provides a tool called *Change Management Pack* that allows some high-level schema change operations. One can compare two database schemas, batch changes to existing database objects, and determine statically if there will be any possible impacts or errors that may need to be mitigated such as insufficient privileges. The tool then creates scripts comprising SQL DDL statements from the schema difference. This capability is similar to those offered by other commercially available schema difference engines (e.g., Altova DiffDog (Altova 2010)), but does not offer the same level of automatic matching capabilities that can be found in schema-matching research tools.

Since the release of version 9i, Oracle also provides a schema evolution feature called *redefinition* (Oracle Database 10g Release 2 2005). Redefinition is performed on single tables and allows the DBA to specify and execute multiple schema or semantic modifications on a table. Changes such as column addition or deletion, changing partitioning options, or bulk data transformation can be accomplished while the table is still available to applications until the final steps of the update propagation.

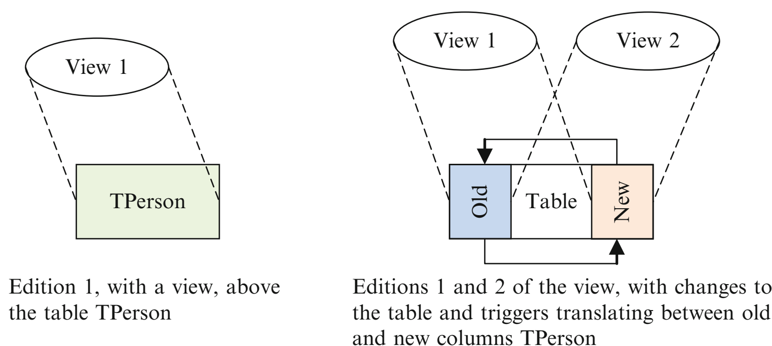
Redefinition is a multistep process. It involves creating an *interim* table with the shape and properties that the table is to have post-redefinition and then interlinking

the interim table with the original table by a *column mapping* specified as a SQL query. The DBA can periodically synchronize data between the two tables according to the query before finally finishing the redefinition. At the end of the redefinition process, the interim table takes the place of the original table. Only the final step requires the table to go offline.

Finally, Oracle supports a feature called *editions* (Oracle Edition-Based Redefinition 2009). An edition is a logical grouping of database objects such as views and triggers that are provided to applications for accessing a database. Using editions, database objects are partitioned into two sets – those that can be editioned and those that cannot. Any object that has a persistent extent, in particular tables and indexes, cannot be editioned. So, an edition is a collection of primarily views and triggers that provide an encapsulated version of the database.

To illustrate the use of editions, consider a simple scenario of a database that handles data about people (Fig. 6.2). In version 1, the database has a table TPerson with a single column Name. Edition 1 also provides applications with an editioned view 1 over TPerson that includes the name column. Schema evolution is triggered by the need to break apart Name into FirstName and LastName. So, version 2 of the database adds two new columns – FirstName and LastName – to TPerson, but leaves column Name present. Edition 2 of the database includes a new view 2 that leaves out Name but includes FirstName and LastName. A background task, run concurrently with the creation of the edition, copies existing data in Name into the new columns but leaves existing data intact. Furthermore, Edition 2 includes two triggers written by the developer to resolve the differences between the two versions. The forward trigger applies to edition 2 and all future editions and takes the data from FirstName and LastName on inserts and updates and applies the data to Name. The reverse trigger applies to all strictly older editions and translates Name data into FirstName and LastName on insert and update. Note that view 1 is still supported on the changed schema so that its applications continue to work.

The resulting database presents two different external faces to different application versions. Version 1 sees edition 1 with a Name column; and version 2 (and beyond) sees edition 2 with FirstName and LastName columns. Both versions can



Edition 1, with a view, above the table TPerson

Editions 1 and 2 of the view, with changes to the table and triggers translating between old and new columns TPerson

Fig. 6.2 Editions in Oracle

create and consume data about a person's name, and that data are visible to the other version as well.

While editions solve a significant process problem, it does not solve data semantics problems. For instance, there is no guarantee that the forward and reverse triggers are in any way inverses of one another and are both essentially opaque program codes to the database system.

SQL Server ships with a tool called SQL Server Management Studio (SSMS) that serves as the GUI front end for a database. The tool can present a database diagram for a given database; the developer can then make changes directly to that diagram, such as adding foreign keys or dropping columns, and the changes are then propagated to the database when the diagram is saved. SSMS also has a *generate change script* feature. While editing a table in a designer, SSMS will track the changes that the developer has made on the table. SSMS packages those changes into a script either on demand or whenever the designer is saved or closed.

SQL Server also includes a feature called *Data-Tier Applications* (Microsoft SQL Server 2008 R2 Data-Tier Applications 2010). At the core of the feature is a distributable file called a DAC pack. A DAC pack is essentially a deployable image of a single version of an application database schema. A typical use case is that a developer packages an application with the schema of a database within such a DAC pack. Initially, when a customer installs the DAC pack an empty database is created with the respective table definitions, views, indexes, etc., from the schema. When the developer creates a new version of the application with an evolved version of the database schema, it is again bundled in a DAC (the old schema version is not considered). When the customer installs the new DAC, the existing database is detected and evolved (upgraded) to the new schema version. The current SQL server version does not do any sophisticated schema-matching heuristics, but also does not make any guesses. If a table has the same name in both the before and after versions, and has columns that are named the same, the upgrade will attempt to transfer the data from the old to the new, failing with a rollback if there are errors like incompatible data types. The resulting evolution process is effectively able to add or drop any objects – tables, columns, indexes, constraints, etc. – but unable to perform any action that requires user intent to capture semantics, such as object renaming (which to an automated process appears like a drop followed by an add). What the approach thus supports are common evolution scenarios of schema element adds and drops for which instance data can be migrated without either developer or user intervention.

IBM DB2 includes a tool called Optim Data Studio Administrator, which is a workbench tool for displaying, creating, and editing database objects with a live connection to a database (IBM 2009a). The interface has a largely hierarchical layout, with databases at the top of the hierarchy moving down to tables and displayed column properties. One can use the tool to manually edit schema objects and commit them to the database. Data Studio Administrator can batch changes together into a script that can subsequently be deployed independently. The script can be statically checked to determine whether the operation can be performed without error. For instance, when changing a column's data type, the default operation is to unload the data from the column's table, make the change to the type, and then reload the data.

If the type changes in such a way that will cause data type conflicts, Data Studio Administrator will alert the user that an error exists and offer the potential solution of casting the column's data on reload.

3.2 Research Approaches

PRISM (Curino et al. 2008) is a tool that is part of a larger project called *Panta Rhei*, a joint project between UCLA, UC San Diego, and Politecnico di Milano investigating schema evolution tools. The PRISM tool is one product of that joint venture that focuses on relational evolution with two primary goals: allow the user to specify schema evolution with more semantic clarity and data preservation and grant multiple versions of the same application concurrent access to the same data.

One contribution of the work on PRISM is a language of schema modification operators (SMOs). The SMO language closely resembles the DDL language in the SQL standard in that it is a textual, declarative language. The two languages also share some constructs, including "CREATE TABLE" and "ADD COLUMN." However, the two languages have two fundamental distinctions.

First, for every statement expressed using the SMO language, there are formal semantics associated with it that describe forward and reverse translation of schemas. The reverse translation defines, for each statement, the "inverse" action that effectively undoes the translation. The only SMO statements that lack these forward and reverse translations are the CREATE TABLE and DROP TABLE operations; logical formalism for these statements is impossible, since one is effectively stating that a tuple satisfies a predicate in the before or after state, but that the predicate itself does not exist in the other state. The work on PRISM describes "quasi-inverses" of such operations; for instance, if one had copied the table before dropping it, one could recover the dropped information from other sources. PRISM offers some support for allowing a user to manually specify such inverses.

Second, SMO and SQL DDL have a different philosophy for what constitutes an atomic change. SQL DDL has a closure property – one can alter any schema S into another schema S′ using a sequence of statements in the language. The statements may be lossy to data, but such a sequence will always be possible. The SMO statements have a different motivation, namely, each statement represents a common database restructuring action that requires data migration. Rather than set the unit of change to be individual changes to individual database elements, the unit of change in PRISM more closely matches high-level refactoring constructs such as vertical or horizontal partitioning. For instance, consider the following statements:

```
MERGE TABLE R, S INTO T
PARTITION TABLE T INTO S WITH T.X < 10, T
COPY TABLE R INTO T
```

These three statements merge two tables, partition a table into two based on a predicate, and copy a table, respectively. Each statement and its inverse can be represented as a logical formula in predicate calculus as well as SQL statements that

describe the alteration of schema and movement of data. For instance, the merge statement above may be represented in SQL as follows:

```
CREATE TABLE T (the columns from either R or S)
INSERT INTO T
  SELECT * FROM R
  UNION
  SELECT * FROM S
DROP TABLE R
DROP TABLE S
```

The second major contribution of PRISM is support for database versioning with backward and forward compatibility. When starting with version N of a database, if one uses SMOs to create version N + 1 of the database, PRISM will provide at least one of the following services to applications whenever the SMOs are deemed to be invertible or the user provides a manual workaround:

– Automatic query rewriting of queries specified against version N into semantically equivalent queries against schema N + 1, and vice versa.
– Views that expose version N of the schema using version N + 1 as a base.

The authors examine the entire schema edit history of the database behind Wikipedia and create a classification of high-level restructuring that covers the vast majority of changes that have occurred in the history of that data repository.

HECATAEUS (Papastefanatos et al. 2010) focuses on the dependencies between schema components and artifacts such as views and queries. Recall that commercial systems have tight restrictions on schema evolution when dependencies exist; one cannot drop a column from a table if a view has been created that references that table. Using HECATAEUS, the developer is given fine-grained control over when to propagate schema changes to an object to the queries, statements, and views that depend on it.

A central construct in HECATAEUS is an *evolution policy* (Papastefanatos et al. 2008). Policies may be specified on creation of tables, views, constraints, or queries. One specifies an evolution policy using a syntactic extension to SQL. For instance, consider the following table definition:

```
CREATE TABLE Person (
  Id INT PRIMARY KEY,
  Name VARCHAR(50),
  DateOfBirth DATE,
  Address VARCHAR(100),
  ON ADD Attribute TO Person THEN Propagate)
```

This DDL statement constructs a table with a policy that states that any added attribute should be automatically added as well to any dependent object. For instance, consider the following view:

```
CREATE VIEW BobPeople AS
SELECT Id, DateOfBirth, Address FROM Person
WHERE Name = 'Bob'
```

If one were to subsequently add a new column "City" to the Person table, the BobPeople view definition would be automatically updated with an additional column Person as well.

Policies are specified on the object to be updated, not the dependent objects. Each policy has three enforcement options: propagate (automatically propagate the change to all dependents), block (which prevents the change from being propagated to dependents), or prompt (meaning the user is asked for each change which action to take). For both propagate and block options, queries are rewritten to either take the new schema semantics into account or preserve the original semantics. The available policies depend on the object being created; for instance, tables may have policies added for adding, dropping, or renaming attributes; drop or rename the relation; or add, drop, or modify constraint.

DB-MAIN is a conceptual modeling platform that offers services that connect models and databases. For instance, one can reverse engineer from a database a conceptual model in an entity–relationship model with inheritance, or one can forward engineer a database to match a given model. The relationship between such models and databases is generally straightforward – constructs like inheritance that exist in the model that have no direct analog in the relational space map to certain patterns like foreign keys in predictable ways (and detectable, in the case of reverse engineering a model from a database).

Research over the last decade from DB-MAIN includes work on ensuring that edits to one of those artifacts can be propagated to the other (Hick and Hainaut 2006). So, for instance, changes to a model should propagate to the database in a way that evolves the database and maintains the data in its instance rather than dropping the database and regenerating a fresh instance. The changes are made in a nonversioning fashion in that, like vanilla DDL statements, the changes are intended to bring the database to its next version and support applications accessing the new version without any guarantee of backward compatibility.

Because DB-MAIN is a tool, it can maintain the history of operations made to the graphical representation of a model. Graphical edits include operations like adding or dropping elements (entity types, relationships, attributes, etc.) as well as "semantics-preserving" operations like translating an inheritance relationship into a standard relationship or reifying a many-to-many relationship into an entity type. Each model transformation is stored in a history buffer and replayed when it is time to deploy the changes to a database instance. A model transformation is coupled with a designated relational transformation as well as a script for translating instance data – in essence, a small extract-transform-load workflow. The set of available translations is specified against the conceptual model rather than the relational model, so while it is not a relational schema evolution language by definition, it has the effect of evolving relational schemas and databases by proxy.

MeDEA (Domínguez et al. 2008) is a tool that, like DB-MAIN, exposes relational databases as conceptual models and then allows edits to the conceptual model to be propagated back to schema changes on the relational database. A key distinction between MeDEA and DB-MAIN is that MeDEA has neither a fixed modeling

language nor a fixed mapping to the database. For instance, the conceptual model for a database may be constructed in UML or an extended ER diagram.

As a result, the relationship between model and database is fluid as well in MeDEA. Given a particular object in the conceptual model, there may be multiple ways to represent that object as schema in the database. Consequently, when one adds a new object to an existing model (or an empty one), the developer has potentially many valid options for persistence. A key concept in MeDEA is the encapsulation of those evolution choices in *rules*. For each incremental model change, the developer chooses an appropriate rule that describes the characteristics of the database change. For instance, consider adding to an ER model a new entity type that inherits from an existing entity type. The developer in that situation may choose as follows:

– To add a new relational table with the primary key of the hierarchy and the new attributes of the type as column, plus a foreign key to the parent type's table (the "table-per-type" mapping strategy).
– To add a new relational table with columns corresponding to all attributes of the new type including inherited attributes (the "table-per-concrete class" mapping strategy).
– To add columns to the table of the parent type, along with a discriminator or a repurposing of an existing discriminator column (the "table-per-hierarchy" mapping strategy).

Each of these strategies may be represented as a rule that may be applied when adding a new type.

Impact Analysis (Maule et al. 2008) is an approach that attempts to bridge the loose coupling of application and schema when the database schema changes. The rough idea is to inform the application developer of potential effects of a schema change at application design time. A complicating factor is that the SQL that is actually passed from application to database may not be as simple as a static string; rather, the application may build such queries or statements dynamically. The work uses dataflow analysis techniques to estimate what statements are being generated by the application, as well as the state of the application at the time of execution so as to understand how the application uses the statement's results.

The database schema evolution language is assumed to be SQL in this work. Schema changes are categorized by their potential impact according to existing literature on database refactoring (Ambler and Sadalage 2006). For instance, dropping a column will cause statements that refer to that column to throw errors when executed, and as such is an error-level impact. Impact analysis attempts to recognize these situations at design time and register an error rather than rely on the application throwing an error at runtime. On the other hand, adding a default value to a column will trigger a warning-level impact notice for any statement referring to that column because the semantics of that column's data has now changed – the default value may now be used in place of null – but existing queries and statements will still compile without incident. DB-MAIN and MeDea focus on propagating changes between relational schemas and conceptual models.

A significant amount of research has recently been dedicated to automatic *mapping adaptation* (Yu and Popa 2005) to support schema evolution and is surveyed in chapter 7 (Fagin et al. 2011). This work mostly assumed relational or nested relational schemas and different kinds of logical schema mappings. For these settings, the definition and implementation of two key operators, composition and inversion of mapping, have been studied. These operators are among those proposed in the context of model management, a general framework to manipulate schemas and mappings using high-level operators to simplify schema management tasks such as schema evolution (Bernstein 2003; Bernstein and Melnik 2007). A main advantage of composition and inversion is that they permit the reuse of existing mappings and their adaptation after a schema evolves. The proposed approaches for mapping adaptation still have practical limitations with respect to a uniform mapping language, mapping functionality, and performance so that more research is needed before their broader usability.

3.3 Summary

Table 6.1 shows a side-by-side comparison of most of the approaches described in this section for key requirements of Sect. 2. With the exception of the Panta Rhei project, all solutions focus on the simple (table) changes of SQL DDL. Oracle is the only system that also allows the specification of changes by providing a new version of a table to be changed as well as a column mapping. Commercial GUIs exist that can support simple diffing and change bundling, but eventually output simple SQL DDL without version mappings or other versioning support. Oracle's edition concept makes versioning less painful to emulate, though underlying physical structures are still not versioned. Overall, commercial DBMS support only simple schema changes and incur a high manual effort to adapt dependent schemas and to ensure backward compatibility. PRISM adds value by enabling versioning through interversion mappings, forward and backward compatibility, and formal guarantees of information preservation when applicable. HECATAEUS improves flexibility by specifying how to update dependent schema objects in a system when underlying objects evolve.

4 XML Schema Evolution

XML as a data model is vastly different than the relational model. Relations are highly structured, where schema is an intrinsic component of the model and an integral component in storage. On the other hand, XML is regarded as a *semi-structured* model. Instances of XML need not conform to any schema, and must only conform to certain well-formed-ness properties, such as each start element having an end tag, attributes having locally distinct names, etc. Individual elements

may contain structured content, wholly unstructured content, or a combination of both. In addition, the initial purpose and still dominant usage of XML is as a document structure and communication medium and not a storage model, and as such, notions of schema for XML are not nearly as intrinsic to the model as with relations. However, a notion of schema for XML is important for application interoperability to establish common communication protocols.

Given that the very notion of XML schemas is relatively new, the notion of schema evolution in XML is equally new. While there have been many proposed schema languages for XML, two have emerged as dominant – Document type definitions (DTDs) and XML Schema, with XML Schema now being the W3C recommendation. Each schema language has different capabilities and expressive power and as such has different ramifications on schema evolution strategies. None of the proposed XML schema languages, including DTDs and XML Schema, have an analogous notion of an "ALTER" statement from SQL allowing incremental evolution. Also unlike the relational model, XML does have a candidate language for referring to schema elements called *component designators* (W3C 2010); however, while the language has been used in research for other purposes, it has to date not been used in the context of schema evolution. Currently, XML schema evolution frameworks either use a proprietary textual or graphical language to express incremental schema changes or require the developer to provide the entire new schema.

The W3C – the official owners of the XML and XML Schema recommendations – have a document describing a base set of use cases for evolution of XML Schemas (W3C 2006). The document does not provide any language or framework for mitigating such evolutions, but instead prescribes what the semantics and behavior should be for certain kinds of incremental schema evolution and how applications should behave when faced with the potential for data from multiple schema versions. For instance, Sect. 2.3 lists use cases where the same element in different versions of a schema contains different elements. Applications are instructed to "ignore what they don't expect" and be able to "add extra elements without breaking the application."

All of the use cases emphasize application interoperability above all other concerns, and in addition that each application be allowed to have a local understanding of schema. Each application should be able to both produce and consume data according to the local schema. This perspective places the onus on the database or middle tier to handle inconsistencies, in sharp contrast to the static, structured nature of the relational model, which generally assumes a single working database schema with homogeneous instances that must be translated with every schema change. Thus, commercial and research systems have taken both approaches from the outset; some systems (e.g., Oracle) assume uniform instances like a relational system, while other systems (e.g., DB2) allow flexibility and versioning within a single collection of documents.

A key characteristic of a schema language such as DTDs and XML Schemas is that it determines what elements may be present in instance documents and in what order and multiplicity. Proprietary schema alteration languages thus tend to

have analogous primitive statements, e.g., change an element's multiplicity, reorder elements, rename an element, insert or remove elements from the sequence, etc. Researchers have created a taxonomy of possible incremental changes to an XML schema (Moto et al. 2007) that is useful for evaluating evolution support in existing systems:

1. Add a new optional or required element to a type.
2. Delete an element from a type.
3. Add new top-level constructs like complex types.
4. Remove top-level constructs.
5. Change the semantics of an element without changing its syntax – for instance, if the new version of an application treats the implicit units of a column to be in metric where previous versions did not.
6. Refactor a schema in a way that does not affect instance validation – for instance, factoring out common local type definitions into a single global type definition.
7. Nest a collection of elements inside another element.
8. Flatten an element by replacing it by its children.
9. Rename an element or change its namespace.
10. Change an element's maximum or minimum multiplicity.
11. Modify an element's type, either by changing it from one named type to another or adding or changing a restriction or extension.
12. Change an element's default value.
13. Reorder elements in a type.

For each class of change, Moto et al. (2007) describe under what conditions a change in that class will preserve forward and backward compatibility. For instance, if in version 2 of a schema one adds optional element X to a type from version 1, any application running against version 1 will be able to successfully run against version 2 and vice versa so long as version 2 applications do not generate documents with element X. If element X is required rather than optional, the two versions are no longer interoperable under this scheme. The same logic can be applied to instances: an instance of schema version 1 will validate against version 2 if X is optional and will not if X is required.

For the rest of this section, we will describe the current state of the art in XML schema evolution as present in commercially available systems and research works. For each solution, in addition to comparing the solution against the requirements outlined in Sect. 2, we describe the classes of incremental changes that the solution supports and in what way it mitigates changes that must be made to either applications or instances. Table 6.2 shows the characteristics of the main approaches considered, which are discussed at the end of this section.

Table 6.2 Characteristics of XML schema evolution systems

	Oracle	Microsoft SQL Server	IBM DB2	Altova Diff Dog	XEM Kramer (2001), Su et al. (2001)	Model-based approaches (X-Evolution, CoDEX, UML)	Temporal XML schema
Description/focus of work	Commercial relational system with XML support	Commercial relational system with XML support	Commercial relational system with XML support	Commercial tool for XML schema diffing	DTD-based incremental changes to schema	View XML schema in a conceptual model	Allow time-varying instances to validate against time-varying schema
Change types							
(1) Richness (simple, complex)	(1) Simple, e.g., addition of an optional element or attribute	(1)–	(1)–	(1) Simple changes like rename, element reordering (additions and multiplicity changes unclear)	(1) Changes determined by DTD data model, e.g., add element or attribute	(1) Changes determined by conceptual model, e.g., rename_type or change_cardinality	(1)–
(2) Specification (incremental, new schema)	(2) Incremental using diffXML language or specification of new schema	(2) Addition of new schemas to existing schema sets	(2) New schema versions are specified as new schemas	(2) Supply of old/new schema	(2) Incremental	(2) Incremental	(2) New versions are added to running temporal schema

(*Continued*)

Table 6.2 (Continued)

	Oracle	Microsoft SQL Server	IBM DB2	Altova Diff Dog	XEM Kramer (2001), Su et al. (2001)	Model-based approaches (X-Evolution, CoDEX, UML)	Temporal XML schema
Evolution mapping							
(1) Representation	(1) Set of changes based on diffXML update language	(1)–	(1)–	(1) Set of element-element correspondences	(1) Set of incremental changes	(1) Set of incremental changes	(1)–
(2) DIFF computation	(2) Manually specified or XMLdiff function	(2)–	(2)–	(2) Semi-automatically derived (manual correction via GUI possible)	(2)–	(2)–	(2)–
Update propagation							
(1) Instances	(1) Specified in XSLT (copyEvolve procedure)	(1) None – documents must validate against new and old schemas	(1) None – documents must validate against the new version that exists at insertion	(1) Generation of XSLT to transform documents	(1) Each schema change is coupled with a series of XML data changes such as addDataEl or destroyDataEl	(1) Each schema change is coupled with a series of XML data changes using XQuery Update or proprietary update languages	(1) No need – documents validate against the version of the document at a given time slice

(2) Dependent schemas	(2)–	(2)–	(2)–	(2)–	(2)–	(2)–	(2)–
Versioning support	–	–	All documents validate against their original schema version	–	–	–	Documents and schemas are both allowed to be versioned
Infrastructure/GUI	Libraries exist to diff two schemas at an XML document level	–	–	GUI to perform diff and to manually correct match results	–	Tool-based generation of changes, capture user actions	–

4.1 Commercial DBMS Systems

All three of the leading commercial database systems at the time of publication – Oracle, Microsoft SQL Server, and IBM DB2 – provide support for storage of XML data validated against an XML schema. Both of the major open-source relational database offerings – PostgreSql and MySql – have support for storing XML, but do not yet support schema validation in their standard configurations. We now briefly describe how each of the three major vendors supports XML schemas in general as well as how each vendor handles changes to those schemas. Furthermore, we discuss evolution support in the native XML database system Tamino.

Oracle offers two very different ways to evolve an XML schema (Oracle XML Schema Evolution 2008). The first is a *copy-based* mechanism that allows a great deal of flexibility. Data from an XML document collection are copied to a temporary location, then transformed according to a specification, and finally replaced in its original location. The second is an *in-place* evolution that does not require any data copying but only supports a limited set of possible schema changes.

Oracle has supported XML in tables and columns since version 9i (9.0.1) as part of XML DB, which comes packaged with Oracle since version 9.2. One can specify a column to have type XMLType, in which case each row of the table will have a field that is an XML document, or one can specify a table itself to have type XMLType, where each row is itself an XML document. In both cases, one can specify a single schema for the entire collection of documents. For instance, one can specify an XML column to have a specified given schema as follows:

```
CREATE TABLE table_with_xml_column
   (id NUMBER, xml_document XMLType)
   XMLTYPE COLUMN xml_document
   ELEMENT "http://tempuri.com/temp.xsd#Global1";
```

Note that when specifying a schema for an XML column or document, one must also specify a single global element that must serve as the document root for each document instance. In the example above, schema `temp.xsd` has a global element `Global1` against which all document roots must validate.

The copy-based version of schema evolution is performed using the `DBMS_XMLSCHEMA.copyEvolve` stored procedure. The procedure takes as input three arrays: a list of schema URLs representing the schemas to evolve, a list of XML schema documents describing the new state of each schema in the first list, and a list of transformations expressed in XSLT. Each transformation corresponds to a schema based on its position in the list; so, the first transformation on the list is used to translate all instances of the first schema to conform to the first new schema definition, and so on.

There are a few restrictions on the usage of `copyEvolve`. For instance, the list of input schemas must include all dependent schemas of anything in the list, even if those schemas have not changed. There are also some additional steps that must be performed whenever global element names change. However, from an expressiveness perspective, one can use the procedure to migrate any schema to any

other schema. There is no correctness validation that the specified transformations actually provide correct instance translation, so in the event that translated documents do not actually conform to the new schema, an error is thrown mid-translation.

The second in-place method of evolution is performed using a different procedure called `DBMS_XMLSCHEMA.inPlaceEvolve`. Because the evolution is performed in place, the procedure does not have any parameters guiding physical migration, given that there is none. The in-place evolution procedure has much less expressive power than the copy version – for this procedure, there is a full reverse-compatibility restriction in place. It is not just the case that all existing instances of the old schema must also conform to the new schema without alteration; it must be the case that all *possible* instances of the old schema must conform to the new one as well. Therefore, the restriction can be statically determined from the schemas and is not a property of the documents currently residing in the database. So, for instance, schema elements cannot be reordered, and elements that are currently singletons cannot be changed to collections and vice versa. The restriction guarantees that the relational representation of the schema does not change, which ensures that the in-place migration does not impose relational disk layout changes.

The kinds of changes that in-place migration does support include as follows:

- Add a new optional element or attribute (a subset of change class 1 from earlier in the section).
- Add a new domain value to an enumeration (subset of change class 11).
- Add a new global element, attribute, or type (change class 3).
- Change the type of an element from a simple type to a complex type with simple content (change class 6).
- Delete a global type, if it does not leave elements orphaned (subset of change class 4).
- Decrease the minOccurs for an instance, or increase the maxOccurs (subset of change class 10).

This list is not comprehensive, but is representative. It is clear from these changes that any valid instance of the old schema will still be valid after any of these changes. To specify these incremental changes, Oracle has a proprietary XML difference language called *diffXML* that is not specific to schemas but rather describe a diffgram between two XML document instances (and XML schemas are, of course, XML documents themselves). Expressions in diffXML loosely resemble expressions in XML update facility in that they have primitives that append, delete, or insert nodes in an XML document. However, diffXML expressions are XML documents rather than XQuery expressions. For instance, one can change the MaxLength restriction facet to 28 in a type using the following sequence of nodes:

```
<xd:delete-node xpath="/schema/complexType
  [@name'Foo']//maxLength/>
<xd:append-node
  parent-xpath = "/schema
    /complexType[@name='Foo']//restriction"
```

```
    node-type = "element">
    <xd:content>
      <xs:maxLength value = "28"/>
    </xd:content>
  < /xd:append-node>
```

Note that the expression language used to navigate an XML schema is vanilla XPath. The `xd` namespace is the namespace for the diffXML language, and `xd:content` nodes contain fragments of XML schema using the `xs` namespace.

One can specify a diffXML document manually, or one can generate it from the XMLDiff function, available both in Oracle's SQL dialect and Java. As mentioned earlier, XMLDiff operates on any XML documents, not just XML schemas, so the in-place evolution is essentially migrating schema by incrementally modifying the schema documents as instances under a guarantee that there will be no cascading effects of the migration.

Microsoft SQL Server, like Oracle, supports storing a collection of homogeneous XML documents in a relation column (Pal et al. 2006). Whereas instances in an XML-typed column or table in Oracle must conform to a specific schema with a specific global element as root, an XML-typed column in SQL Server validates against any schema in a collection of schemas and allows any global element as root. One specifies an XML Schema Collection in SQL server using a DDL statement:

```
CREATE XML SCHEMA COLLECTION [<relational_schema>.]
   sql_identifier AS Expression
```

Once a schema collection has been created, it can be assigned to be the schema for any column whose type is XML. Also, once the collection is created, there are only two operations that can be done on it – drop it or alter it by adding new constructs. The ALTER statement is the only form of schema evolution that SQL Server allows without manually dropping the schema, manually translating instances, and reestablishing the schema. The ALTER statement has only one form:

```
ALTER XML SCHEMA COLLECTION [relational_schema.]
   sql_identifier ADD Expression
```

For both the CREATE and ALTER statements, the expression must be a forest of valid XML schema documents. The ALTER statement can add schema elements to namespaces that already exist in the collection or to new namespaces.

The monotonic nature of alterations to a schema collection X means that, for the most part, documents that conform to collection X will continue to validate against the collection after alteration (maintaining the same reverse-compatibility restriction of the in-place evolution in Oracle). The one exception is if the collection contains a lax validation wildcard or any element whose type is xs:anyType. In such a case, the addition of new global elements to the schema collection could cause documents to fail validation. So, if any existing schema elements include such a construct, revalidation of existing documents will happen any time new global elements are added, and if the revalidation fails, the action is aborted.

IBM DB2 takes a different approach to XML schema validation, one that embraces the XML notion of interoperability rather than instance homogeneity (Beyer et al. 2005). Rather than apply a single schema or schema set against an entire collection of documents in a table or column, DB2 schema validation occurs on a per-document basis. XML documents may be validated against a schema at the time of insertion; however, the schema against which to validate the document is not determined by the schema associated with the column, since there by definition is no such schema. Rather, the schema is determined by attributes within the document to be inserted, or by manually specifying a schema as an argument to the XMLValidate function. Once a document has been validated, the document is adorned with metadata that verifies that the document was validated as well as information to help optimize query processing.

Like Oracle's schema registration service and SQL Server's schema collections, DB2 requires one to register XML schemas in the system prior to use:

```
register xmlSchema 'foo://tempuri.com/schema.xsd'
     from 'schema-v1.xsd' as schemaV1 complete
```

DB2 has no support for schema evolution per se, as different versions of the same schema appear in the database repository as unconnected documents. One also does not update document instances from one version of a schema to another, similar to SQL Server. Researchers from IBM have described how to support schema versioning using a complete scenario (Beyer et al. 2005); the scenario involves a relational table that correlates the currently registered schemas (and thus schema versions) with the applications currently using them. All of the mitigation of schema versioning is handled by the tables and protocols set up in the scenario rather than inside the engine.

Since the engine does not enforce document homogeneity, it allows documents from multiple schemas and thus multiple schema versions to coexist in a single corpus with full fidelity. Rather than automatically evolve instances, the documents exist in their original form, associated with its original schema.

Native XML databases, unlike relational systems, are built from the ground up to support XML storage. Relatively few of these systems support XML schemas or schema evolution. One notable exception is *Tamino* (Software AG 2006).

Like Oracle, Tamino can store XML data in a fashion that is XML schema dependent, i.e., the physical structures may be optimized, possibly by mapping to relations, knowing that the XML data is regularly structured in some way. Also similarly to Oracle, Tamino allows schemas to evolve under the same restrictions as Oracle's in-place migration mechanism. One specifies a new schema version wholesale – no mapping or incremental changes are possible – providing the entire schema document, and passing it to the same `_define` command to define an initial version.

Where Tamino differs from Oracle is that Tamino allows the stored data to determine reverse compatibility rather than the schema document versions themselves. One can pass a parameter to the `_define` command to attempt to do some static validation first – determining just from the documents themselves whether it is possible for reverse compatibility to be guaranteed – but eventually all documents are

validated against the new schema at evolution time and, if any fail validation, the change is rejected.

4.2 Mapping Tools

Altova (Altova 2010) does specialize in XML-specific tools for document and data management. Altova provides a tool called DiffDog that can perform XML schema matching and diffing. The tool takes as input two XML schema instances and performs element-to-element matching. The tool's result can be manually modified to accommodate renames that the automatic algorithm does not immediately catch. From a diff result, the tool generates an XSLT script that translates valid documents of one schema into valid documents of the other schema. The tool can thus handle renaming and reordering of elements in a fairly straightforward manner. It is unclear from documentation whether the tool can handle addition of required elements or changes in multiplicity; such changes would not be straightforward in the user interface of the tool. There is also no mechanism to incrementally alter schemas – schemas are diffed wholesale. A related tool Altova MapForce is used to generate XSLT mappings between different XML schemas that are not in an evolution relationship but may differ to a larger extent. The initial schema matching is therefore to be provided by a human user.

Research on schema matching and mapping has also resulted in several tools to semi-automatically determine executable mappings such as Clio, e.g., for instance migration after schema evolution (Jiang et al. 2007; Bonifati et al. 2011). The tools do not provide for incremental evolutions per se, but map between the old and the evolved schema. None of the existing mapping-based tools provide full support for all of the features of XML Schema; for instance, Clio supports a significant subset of XML Schema but not element order, choice particles, or element multiplicity restrictions other than zero, one, or unbounded.

4.3 Research Approaches

As of the year 2000, the DTD was the predominant method for schematizing XML documents. As the decade progressed, XML Schema became the dominant schematizing technology for XML. That same trend has been mirrored in research; schema evolution techniques introduced earlier in the decade focused more on changes to a DTD, while more recent publications cover the far more expressive XML Schema recommendation.

XEM (Kramer 2001; Su et al. 2001) – XML Evolution Management – is a framework introduced by Worcester Polytechnic Institute in 2001 describing evolution management in DTDs. The approach predates schema evolution in any of the commercial systems introduced in the previous section. The work provides a sound and

complete set of change operations. The set is sound in that each evolution primitive is guaranteed to maintain all validity and integrity properties; post-evolution, all documents will still be well-formed XML and will still validate against the DTD. The set is complete in that one can start with any DTD and arrive at any other valid DTD using only changes from the set. The set of schema changes is as follows:

- Create a DTD element type (change class 3).
- Delete a DTD element type (change class 4).
- Insert DTD element or attribute into an existing element type (change class 1).
- Remove an element or attribute from an existing element type (change class 2).
- Change the quantifier on an element in a type (change class 10, limited to the kinds that DTD is capable of).
- Nest a set of adjacent elements in a type beneath a new element (change class 7).
- Flatten a nested element (change class 8).

Each individual change to a DTD induces a change on all valid documents to maintain document validity. For instance, if one adds a new required element or changes the quantifier on an element so that it becomes required, XEM will automatically add a default element to all instances that lack the element. Note that this evolution scheme takes a relational approach to evolution in the sense that all instances must evolve to match the new schema rather than allowing documents to belong to multiple versions simultaneously.

DTD-Diff (Leonardi et al. 2007) is an algorithm and tool for detecting changes between versions of a DTD. The algorithm takes as input two DTD instances and returns a list of changes from the following categories:

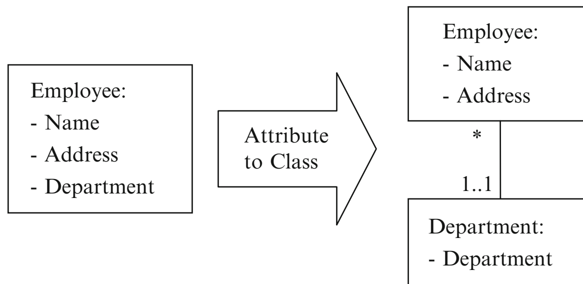
- Adding or deleting element, attribute, or entity declarations (change classes 3 and 4).
- Change the content of an element type by adding, removing, or reordering nodes (change classes 1, 2, 11, and 13).
- Change element cardinality (change class 10, limited to DTD support).
- Update attribute or entity facets such as changing a default value of an attribute or updating the replacement text of an entity declaration (change classes 5 and 12).

The set of supported changes explicitly does not include construct renaming, due to the fully automated nature of the difference engine – one could imagine adding support for allowing the result of a matching graph as additional input to handle such renaming, though. The authors claim that applying existing XML document change detection algorithms to instances of XML Schema (which are themselves XML documents) does not necessarily yield semantically correct or optimal changes.

Diagram-based evolution (Domínguez et al. 2005) is a way to bypass the absence of a standard evolution language by allowing the developer to express evolution intent using a tool. One such effort uses UML diagrams as a front end for an XML schema; in turn, changes to a diagram translate to changes on the associated schema. In that framework, a UML diagram is used as a conceptual model for an XML schema and its corresponding documents. The UML diagrams supported by the

framework do not have the same expressive power as the full XML schema language, and so the work focuses on the subset of XML Schema to which the UML language maps cleanly. Changes to the UML diagrams within a tool then induce changes to the underlying schema and instances in the form of deployable XSLT documents.

The set of changes that the UML framework supports is thus heavily influenced by the tooling support. For instance, the change that is described in depth in Domínguez et al. (2005) is a refactoring operation that translates an attribute in a UML class into its own class:



In general, each class corresponds to a type in an XML schema with an element and a key. Attributes correspond to nested elements, while associations map to key references. The refactoring operation above therefore results in removing the nested element from the Employee type, creating a new type and element with a key for Department, and a key reference between the two types. An XSLT stylesheet is also generated to migrate data to ensure Department data is not lost.

A similar and more recent approach is *CoDEX* (Klettke 2007), which uses a conceptual model that is closely aligned with XML Schema rather than using UML. Again, incremental changes made to the conceptual model result in changes to the associated schema and valid documents. The work on CoDEX also describes an algebra that does preprocessing on incremental changes. As the user edits the model, a log of actions is recorded, which can subsequently be optimized using reduction rules. For instance, adding a new element then renaming it is equivalent to simply adding the element with the new name to begin with.

X-Evolution (Guerrini and Mesiti 2009; Mesiti et al. 2006) is another framework that defines incremental schema evolution in terms of a tool, in this case a graph representation of the schema. Like CoDEX and the UML tools, X-Evolution supports a set of evolution primitives; the list is too long to mention in-line, but covers change classes except 7, 8, and 13 from our running list (and the algorithm in X-Evolution could be altered in a fairly straightforward way to accommodate them). X-Evolution is also able to handle a kind of change not listed in the change taxonomy at all – specifically, changing a content particle's type, say, from ALL to SEQUENCE or CHOICE. A subset of the list of incremental evolutions is classified as having no effect on validation, such as the removal of a global type that has no current element instances. With any such evolution, no document revalidation is necessary – this list of validation-less changes tracks with the research done in Moto et al. (2007).

A key contribution of the work on X-Evolution is *incremental repudiation and revalidation*. Given an incremental change to a schema, X-Evolution runs one of two algorithms at the user's request – one that tests valid documents to see if they are still valid post-validation and one that alters valid documents to make them valid with respect to the new schema. Both algorithms are incremental, as the documents are not re-validated en masse. Instead, only the parts of the document that correspond to the altered part of the document are re-validated (or altered).

Temporal XML Schema (Currim et al. 2009) – also referred to as *τXSchema* – is a way to formalize the temporal nature of schema and document versioning. The framework is assembled by the same research group that helped develop the temporal extensions to SQL. In all other frameworks discussed to date, the relationship between versions of schemas and documents are informal if they exist at all; two versions of the same schema version are considered to be two separate schemas, related to each other only by whatever point-in-time script was used to perform the migration. τXSchema makes evolution over time a first-class concept, modeling different versions of the same conventional XML schema in the same document.

τXSchema enforces the standard constraints of an XML schema. Assuming that a temporal document is valid with respect to a temporal schema, restricting the document to a single point in time produces a document that is valid with respect to its XML schema at that same point in time. Any conventional schema constraint must be valid at all points in time as well, such as keys, key references, and data type restrictions. In addition, temporal documents and schemas are still valid XML documents with additional elements and attributes added to reflect temporal characteristics; τXSchema provides extensions to existing XML tools that perform the additional temporal validation of documents.

4.4 Summary

Table 6.2 shows a comparison of most of the previously mentioned approaches to XML evolution relative to the characteristics laid out in Sect. 2. In general, commercial options support evolution where instances may need to be revalidated but need not be updated. The exception is Oracle, where one can specify XSLT scripts to migrate instances. There is no commonly supported evolution language to specify incremental updates, a shortcoming that research approaches circumvent by inventing proprietary solutions. XEM and model-based solutions attempt to couple incremental schema changes with incremental data changes, which often results in empty or default element generation to fill gaps where a document no longer validates. None of the solutions explicitly support versioning unless they support multiple versions appearing side by side physically in persistent storage, as IBM and temporal XSchema do. Altova presents a dedicated diffing tool with noncomplete capabilities, and model-driven approaches offer a GUI-based method to specify incremental changes. Mapping tools such as Clio also support diff computation and instance migration for XML-like schemas. Currently, there is not yet any support for adapting dependent mappings/schemas for XML schema evolution.

5 Ontology Evolution

Gruber (1993) characterizes an ontology as the explicit specification of a conceptualization of domain. While there are different kinds of ontologies, they typically provide a shared/controlled vocabulary that is used to model a domain of interest using concepts with properties and relationships between concepts. In the recent past, such ontologies have been increasingly used in different domains to semantically describe objects and to support data integration applications. For example, there are a growing number of life science ontologies, e.g., the ontologies managed in the open biomedical ontologies (OBO) Foundry (Smith et al. 2007). The existing ontologies are not static but are frequently evolved to incorporate the newest knowledge of a domain or to adapt to changing application requirements.

There are several differences between ontologies and relational schemas that influence their evolution:

- Ontologies are conceptually more abstract models than database schemas and come in different variations ranging from controlled vocabularies and thesauri over is-a hierarchies/taxonomies and directed a-cyclic graphs (DAG) to frame-based and formal representations (Lassila and McGuinness 2001). For instance, ontology languages such as RDF or OWL allow the specification of concept hierarchies with multiple inheritance, cardinality constraints, inverse or transitive properties, and disjoint classes. The kind and expressiveness of ontologies determine the kind of changes that should be supported for ontology evolution. For instance, Noy and Klein (2004) propose a set of 22 simple and complex ontology change operations such as concept creation, reclassification of a concept, or merge/split of concepts.
- The role of instances differs between ontologies and relational schemas. For example, many ontologies include instances but do not clearly separate them from other parts of the ontologies such as concepts and relationships. In other cases, instances are described by ontologies but are maintained outside the ontology within separate data sources. These differences impact update propagation of ontology changes since the separately maintained instances may not be under the control of the ontology editors.
- In contrast to database schemas, the development and evolution of ontologies is often a collaborative and decentralized process. Furthermore, new ontologies often reuse existing ones, i.e., an ontology engineer uses a common ontology as the basis for domain-specific extensions. These aspects lead to new synchronization requirements for ontology changes. Furthermore, ontologies serving a whole domain likely introduce many usage dependencies, although ontology providers usually do not know which applications/users utilize their ontology. Supporting different ontology versions is a main approach to provide stability for ontology applications. For example, there are daily new versions for the popular Gene Ontology.

Despite these differences, it is easy to see that the schema evolution requirements introduced in Sect. 2 also apply to ontology evolution, in particular support for a rich

set of changes, expressive mappings, update propagation to instances and dependent schemas/ontologies, versioning, and user-friendly tools.

For the rest of this section, we will describe representative approaches on ontology evolution and how they meet the introduced requirements. Table 6.3 comparatively shows selected approaches that are discussed at the end of the section.

5.1 Research Approaches

The *Protégé* system supports different kinds of collaborative ontology evolution meeting varying functional requirements (Noy et al. 2006). First, ontologies can be modified synchronously or asynchronously. Synchronous editing is performed on a centrally stored ontology that can be modified concurrently by several developers. For asynchronous editing collaborators check out the latest ontology version, change it offline, and merge their changes into a common version later on. Second, ontologies may internally be versioned or not. Ontologies may so periodically be archived with the possibility to roll back to a former version. Alternatively, all changes are continuously directed to a single (the most recent) ontology version. Third, ontology changes may be subject to the approval of designated curators to resolve potential problems and maintain a high quality. Usually, such a curation is performed before releasing a new version of an ontology. Finally, ontology changes may be monitored (logged) or not.

The ontology evolution framework supports a rich set of simple and complex changes that can be annotated (Noy et al. 2006). These changes are classified within a change and annotation ontology (CHAO). Annotation includes the type of ontology change, the class/property/instance that was changed, the user and date/time when the change was performed. The two main approaches to specify changes are supported: specification (and logging) of incremental change operations and the provision of a new ontology version. In the latter case, the Diff evolution mapping is semi-automatically determined.

Protégé uses the PROMPTDIFF algorithm (Noy and Musen 2002) to determine an evolution mapping between two input ontology versions. The two versions V1 and V2 are compared using an iterative algorithm combining different heuristic matchers (e.g., single unmatched sibling, unmatched inverse slots, or same type/name) until no more changes are found. The found changes are presented in a so-called difference table containing a set of tuples that interrelate elements of V1 with elements of V2. Each tuple specifies a change operation (add, delete, split, merge, and map) and its parameters.

The different kinds of ontology evolution are implemented in the Protégé ontology editor within two plugins: Change-management plugin and the PROMPT plugin. The Change-management plugin can be used to access a list of changes, allows users to add annotations, and enables to study the history of concepts, i.e., users can examine what modifications happened on a particular concept in the history. The PROMPT plugin implements the PROMPTDIFF algorithm and provides

Table 6.3 Characteristics of selected ontology evolution systems

	Protégé Noy et al. (2004, 2006), Noy and Musen (2002)	KAON Stojanovic et al. (2002)	OntoView Klein et al. (2002)	OnEX Hartung et al. (2008, 2009, 2010), Kirsten et al. (2009)
Description/focus of work	Flexible framework for ontology management and evolution	Process for consistent ontology evolution	Version management and comparison for RDF-based ontologies	Quantitative evolution analysis for life science ontologies and mappings
Supported ontology formats	RDF/OWL, further formats via import plugins	RDF/OWL	RDF	OBO, RDF, CSV; further formats via adaptable import
Change types				
(1) Richness (simple, complex)	(1) Simple and complex (siblings_move, ...)	(1) Simple and complex (merge, copy, ...)	(1) Simple	(1) Simple and complex (merge, split, ...)
(2) Specification (incremental, new schema)	(2) Incremental or specification of new ontology version	(2) Incremental	(2) Integration of new versions	(2) Integration of new versions
Evolution mapping				
(1) Representation	(1) Incremental changes or difference table for two versions	(1) Incremental changes	(1) Set of changes interrelating two versions	(1) Set of changes interrelating two versions
(2) DIFF computation	(2) PROMPTDIFF algorithm	(2)–	(2) Rule-based diff computation	(2) Matching and rule-based diff computation

Update propagation				
(1) Instances	(1)–	(1) Migration of instances managed with the ontology	(1)–	(1) Adaptation of annotations affected by ontology change
(2) Dependent schemas	(2)–	(2) Recursive application of evolution process on dependent ontologies	(2)–	(2)–
Versioning support	Sequential versioning	–	Sequential versioning based on CVS	Supports existing sequential ontology versions
Infrastructure/GUI	Protégé ontology editor with PROMPT and change management plugin	GUI-based editor in KAON infrastructure	Web-based application to access, compare, and version ontologies	Web-based application to explore changes in life science ontologies

facilities to accept/reject performed changes for curators. Besides these two plugins, the Protégé environment provides functionality for editing in a client–server mode as well as transaction and undo support.

The *KAON* prototype (Karlsruhe Ontology and Semantic Web Tool Suite) providing a graphical user interface for incrementally editing ontologies within a process of six phases (Stojanovic et al. 2002). For each change, the following sequential phases are needed: (1) Change Capturing, (2) Change Representation, (3) Semantics of Change, (4) Change Implementation, (5) Change Propagation, and (6) Change Validation. The evolution process can be cyclic, i.e., after the last phase, the process can be re-executed for further ontology changes.

In the first phase (Change Capturing), the ontology engineer decides about the necessary ontology changes, e.g., to delete a concept. In phase 2 (Change Representation), such change requests are translated into a formal change representation. The approach distinguishes between elementary (simple) as well as composite (complex) changes that can be expressed by a series of elementary ones. In total, 16 elementary changes (additions/deletions/modifications of concepts, properties, axioms, and subclass relationships) and 12 composite changes (merging and moving of concepts, concept duplication/extraction, etc.) are distinguished.

Phase 3 uses the formal change representation to identify potential problems (inconsistencies) that the intended changes can introduce within the ontology. For example, the deletion of a concept C impacts its children and instances. Different evolution strategies can be specified to deal with such situations, e.g., one can delete the children as well or move the children to be subconcepts of C's parent concept. To reduce the manual effort for such decisions, different default evolution strategies can be specified. Furthermore, the evolution strategies to resolve inconsistencies may be automatically determined controlled by general goals such as minimizing the number of ontology changes or keeping the ontologies flat.

The resulting changes are presented to the user for confirmation and are then implemented in phase 4. All performed changes are logged in a version log; an explicit versioning does not take place. The following phase 5 (Propagation) is responsible to propagate the ontology changes to dependent applications or other ontologies that extend the modified ontology. This approach assumes that the consumers of the ontology are known and that the ontology evolution process can be recursively applied on the dependent ontologies. The final Validation phase gives ontology engineers the possibility to review the performed changes with the option of undoing changes. Moreover, she can initiate further change requests by starting another evolution cycle.

The *OntoView* system (Klein et al. 2002) focuses on versioning support for RDF-based ontologies. The system is inspired by the concurrent versioning system (CVS), which is used in collaborative software development. One of its core functions is to structurally compare ontology versions to determine different types of changes (representing a Diff evolution mapping). Nonlogical changes denote changes in the label or comment of a concept. Logical definition changes may affect the formal semantics of a concept, e.g., modifications on subClassOf, domain/range of properties, or property restrictions. Further change types include identifier

changes and the addition/deletion of definitions. More complex changes such as merges or splits of concepts are not supported.

The detection algorithm is inspired by the UNIX diff operation but uses the ontology graph structure and RDF triples <subject, predicate, object> as the basis for the version comparison. Change detection between two graphs is based on IF–THEN rules that specify conditions on triples in the old/new ontology and produce resulting changes if the conditions are fulfilled. The authors argue that they can specify and detect almost every change type using this mechanism except identifier changes.

Ontology evolution explorer (OnEX) is a web-based system for exploring changes in numerous life science ontologies (Hartung et al. 2009). It uses existing ontology versions and identifies the differences between succeeding versions of an ontology. The differences are represented by evolution mappings consisting of simple changes (adds, deletes, updates of concepts/relationships, and attributes) that are identified by comparing the unambiguous accession numbers of elements available in life science ontologies (Hartung et al. 2008). OnEX can be used to determine the stability and specific change history of ontologies and selected concepts of interest. Furthermore, one can determine whether given annotations referring to an ontology version have been invalidated, e.g., due to deletes. Such annotations can then be semi-automatically migrated to be consistent with the newest version of the respective ontology.

OnEX uses a tailored storage model to efficiently store all ontology versions in its repository by utilizing that succeeding ontology version differ only to a small degree (Kirsten et al. 2009). Currently, OnEX provides access to about 700 versions of 16 life science ontologies.

The ontology diff algorithm proposed in Hartung et al. (2010) determines an evolution mapping between two ontology versions. The evolution mapping consists of a set of simple as well as complex ontology changes (e.g., merging or splitting of concepts). The approach is based on an initial matching of the ontology version and applies so-called Change Operation Generating Rules (COG rules) for deriving the change operations of the evolution mapping. For instance, the rule for determining a merge of multiple concepts looks as follows:

$$\begin{aligned} &\exists mapC(a,c) \wedge \exists mapC(b,c) \wedge \neg\exists mapC(a,d) \wedge \neg\exists mapC(b,e) \\ &\wedge a \neq b \wedge c \neq d \wedge c \neq e \quad \rightarrow \mathbf{create}[merge(\{a\},c)], \mathbf{create}[merge(\{b\},c)] \end{aligned}$$

The rule derives that concepts a and b are merged into concept c if there are two match correspondences *mapC*(*a,c*) and *mapC*(*b,c*) and if *a* and *b* are not connected to any other concept. The approach could be validated for different kinds of ontologies.

Change detection using a version log: Plessers and De Troyer (2005) builds upon the KAON ontology evolution process (Stojanovic et al. 2002). The proposed evolution process consists of five phases: (1) Change Request, (2) Change Implementation, (3) Change Detection, (4) Change Recovery, and (5) Change Propagation. The main difference is in the Change Detection phase where additional implicit changes are detected based on the history (log) of previous changes as well

as the so-called version log containing the different versions of ontology concepts during their lifetime.

Changes are either basic (simple) or composite and defined declaratively using the change definition language (CDL), which is based on RDF/OWL. Both kinds of changes are determined by evaluating the old and new ontology versions w.r.t. rule-based change definitions. For example, the change definition

$$\forall\ p \in P, A \in C : addDomain(p, A) \leftarrow \neg hasDomain(p, A, v_{i-1}) \\ \wedge\ hasDomain(p, A, v_i)$$

specifies that the basic change *addDomain(p, A)* to add A as the domain of property p has occurred when this domain has not been in the old version v_{i-1} but in the changed version v_i. Composite changes are more difficult to determine since they involve several ontology elements that may be subject to changes themselves that may have to be taken into account. The correct identification of such changes is important to correctly adapt instances of the ontology. For instance, we may have two basic changes to move property p from class *C1* to class *C2* followed by a subclass addition between *C1* and *C2*. Treating these changes independently would first delete all properties p in instances of *C1*. However, the following addition of a subclass relationship between *C1* and *C2* would require the addition of property p to the *C1* instances. By finding out that the two basic changes realize the composite change of moving up p in the class hierarchy, the unnecessary deletions of p values can be avoided.

Detection of high-level changes in RDF/S ontologies: Papavassiliou et al. (2009) focuses on the detection of high-level changes (diff) between two RDF/S-based ontology versions. Their framework uses a formal language to define changes and distinguishes between basic, composite, and heuristic changes. Heuristic changes refer to changes that are detected by matchers employing heuristic techniques to determine that classes have been renamed, merged, or split. The proposed algorithm focuses on the detection of basic and composite changes and utilizes the so-called low-level delta containing the RDF triples that have been added and deleted between two versions V1 and V2 of a RDF/S knowledge base. Changes are described by triples consisting of (1) required added RDF triples, (2) required deleted RDF triples, and (3) a set of conditions that need to be fulfilled. For instance, the change *Delete_Superclass(x,y)*, which removes the is-a relationship between x and y, can be described as follows: (1) no added triple exists, (2) the deletion of a triple (x, subClassOf, y) exists, and (3) x is a class in V1. The detection algorithm first uses the low-level delta and the change descriptions to find potential changes between V1 and V2. The second step then iteratively selects changes that meet the conditions and reduces the set of changes in the low-level delta. The algorithm first identifies composite changes and then basic ones.

5.2 Summary

Table 6.3 shows a comparison of most systems that are discussed. While the first two systems Protégé and KAON support complete processes for ontology evolution, OntoView and OnEX focus on the management of existing ontology versions developed elsewhere. Supported ontology formats are primarily RDF and OWL; Protégé and OnEX can integrate further formats (e.g., OBO). With the exception of OntoView, all systems support both simple and complex changes. The representation and determination of an evolution mapping between two ontology versions differs among the systems. Protégé is most flexible for specifying ontology changes by supporting both incremental changes and the provision of a new ontology version; the other systems follow only one of the two possibilities. A Diff computation is supported by all systems except KAON. The update propagation to instances and related data is partially supported in KAON and OnEX. KAON uses evolution strategies to adapt instances managed together with the ontology. OnEX supports the identification and migration of annotations affected by ontology changes. With the exception of KAON, all systems support sequential versioning. Graphical user interfaces are provided by all systems: Protégé and KAON are editor-like applications, while OntoView and OnEX are web-based.

6 Conclusions

Effective schema evolution is a long-standing problem that is difficult to address since schema changes impact existing instances, index and storage structures as well as applications, and other schema consumers. We introduced the main requirements for effective schema evolution and provided an overview about the current state of the art on the evolution of relational schemas, XML schemas, and ontologies. More than 20 approaches have been analyzed against the introduced requirements and we used several tables to compare most of these approaches side by side. The introduced methodology should be similarly applicable to evaluate further schema or ontology evolution approaches. We summarize some of our observations as follows.

Commercial DBMS currently restrict their support for evolving relational schemas to simple incremental changes and instance migration, while there is not yet support to semi-automatically propagate changes to dependent schemas, mappings, and applications. Filling this gap requires support for the determination and processing of expressive schema mappings that have been studied in recent research approaches such as Pantha Rei/Prism and in model management research (Bernstein and Melnik 2007).

The evolution of XML schemas is easier than for relational schemas since the schemas can be extended by optional components that do not invalidate existing instances. Due to the absence of a standard schema modification language, schema changes are usually specified by providing a new version of the schema. In research approaches, schema matching and mapping techniques are being used

to semi-automatically derive the evolution mapping between two schema versions and to derive a corresponding instance-level mapping for instance migration. Support for propagating changes of XML schemas to dependent schemas or applications have not yet been studied sufficiently.

Research on ontology evolution considers both the adoption of incremental changes and the provision of new schema versions to specify several changes at once. Several approaches have been proposed to semi-automatically determine Diff evolution mappings by comparing two ontology versions. These mappings are usually represented by sets of simple or complex changes. While instance migration has been considered to some extent, the propagation of ontology changes to dependent ontologies/schemas, or applications have not yet found sufficient attention.

Despite recent progress, we therefore see a need for substantially more research on schema evolution, also in areas not discussed in this chapter. For example, distributed architectures with many schemas and mappings need powerful mapping and evolution support, e.g., to propagate changes of a data source schema to merged (global) schemas. New challenges are also posed by dynamic settings such as *stream systems* where the data to be analyzed may change its schema, e.g., by providing new or changed attributes. A first approach in this area is Fernández-Moctezuma et al. (2009). They propose certain extensions for schema consumers such as query operators to deal with changed schemas.

References

Altova DiffDog (2010) http://www.altova.com/diffdog

Ambler SW, Sadalage PJ (2006) Refactoring databases: Evolutionary database design. Addison Wesley, MA

Bernstein PA (2003) Applying model management to classical meta data problems. In: Proceedings of Conference on Innovative Database Research (CIDR) 2003. ACM, NY, pp 209–220

Bernstein PA, Melnik S (2007) Model management 2.0: manipulating richer mappings. In: Proceedings of ACM SIGMOD conference. ACM, NY, pp 1–12

Beyer K, Oezcan F, Saiprasad S, Van der Linden B (2005) DB2/XML: Designing for evolution. In: Proceedings of ACM SIGMOD conference. ACM, NY, pp 948–952

Bonifati A, Mecca G, Papotti P, Velegrakis Y (2011) Discovery and correctness of schema mapping transformations. In: Bellahsene Z, Bonifati A, Rahm E (eds) Schema matching and mapping, Data-Centric Systems and Applications Series. Springer, Heidelberg

Cate BT, Kolaitis PG (2010) Structural characterizations of schema-mapping languages. Comm ACM 53(1):101–110

Curino CA, Moon HJ, Zaniolo C (2008) Graceful database schema evolution: The PRISM workbench. In: Proceedings of VLDB conference. VLDB Endowment. pp 761–772

Currim F, Currim S, Dyreson CE, Joshi S, Snodgrass RT, Thomas SW, Roeder E (2009) tXSchema: Support for data-and schema-versioned XML documents. TimeCenter Technical Report TR-91, Aalborg University, Denmark

Domínguez E, Lloret J, Rubio AL, Zapata, MA (2005) Evolving XML schemas and documents using UML class diagrams. In: Proceedings of DEXA conference. Springer, Heidelberg

Domínguez E, Lloret J, Rubio AL, Zapata MA (2008) MeDEA: A database evolution architecture with traceability. Data Knowl Eng 65(3):419–441

Fagin R, Kolaitis PG, Popa L, Tan W (2011) Schema mapping evolution through composition and inversion. In: Bellahsene Z, Bonifati A, Rahm E (eds) Schema matching and mapping, Data-Centric Systems and Applications Series. Springer, Heidelberg

Fernández-Moctezuma R, Terwilliger JF, Delcambre LML, Maier D (2009) Toward formal semantics for data and schema evolution in data stream management systems. In: Proceedings of ER workshops. Springer, Heidelberg, pp 85–94

Gruber TR (1993) A translation approach to portable ontology specifications. In: Knowledge acquisition, vol 5(2). Academic, London, pp 199–220

Guerrini G, Mesiti M (2009) XML schema evolution and versioning: current approaches and future trends. In: Open and novel Issues in XML database applications. Future directions and advanced technologies. IDEA Group, pp 66–87

Hartung M, Kirsten T, Rahm E (2008) Analyzing the evolution of life science ontologies and mappings. In: Proceedings of 5th international workshop data integration in the life sciences (DILS). LNCS, vol 5109. Springer, Heidelberg

Hartung M, Kirsten T, Gross A, Rahm E (2009) OnEX – Exploring changes in life science ontologies. BMC Bioinformatics 10:250

Hartung M, Gross A, Rahm E (2010) Rule-based determination of Diff evolution mappings between ontology versions. Technical report, University of Leipzig

Hick JM, Hainaut JL (2006) Database application evolution: a transformational approach. Data Knowl Eng 59(3):534–558

IBM (2009a) Database version control with IBM Optim Database Administrator V2.2. http://www.ibm.com/developerworks/data/library/techarticle/dm-0704henry/index.html

IBM (2009b) DB2 9.7: Online schema change. http://www.ibm.com/developerworks/data/library/techarticle/dm-0907db2outages/index.html

Jiang H, Ho H, Popa L, Han WS (2007) Mapping-driven XML transformation. In: Proceedings of WWW conference. ACM, NY, pp 1063–1072

Kirsten T, Hartung M, Gross A, Rahm E (2009) Efficient management of biomedical ontology versions. In: Proceedings on the move to meaningful internet systems (OTM) workshops. Springer, Heidelberg, pp 574–583

Klein M, Fensel D, Kiryakov A, Ognyanov D (2002) Ontology versioning and change detection on the web. In: Proceedings of 13th international conference on knowledge engineering and knowledge management. Ontologies and the semantic web. Springer, Heidelberg

Klettke M (2007) Conceptual XML schema evolution – the CoDEX approach for design and redesign. In: Proceedings of BTW workshops, pp 53–63

Kramer D (2001) XEM: XML evolution management. Master's Thesis, Worcester Polytechnic Institute

Lassila O, McGuinness, D (2001) The role of frame-based representation on the semantic web. Knowledge Systems Laboratory Report KSL-01-02, Stanford University

Leonardi E, Hoaia TT, Bhowmicka SS, Madria S (2007) DTD-Diff: A change detection algorithm for DTDs. Data Knowl Eng 61(2):384–402

Maule A, Emmerich W, Rosenblum DS (2008) Impact analysis of database schema changes. In: Proceedings of international conference on software engineering (ICSE). ACM, NY, pp 451–460

Mesiti M, Celle R, Sorrenti, MA, Guerrini G (2006) X-Evolution: A system for XML schema evolution and document adaptation. In: Proceedings of EDBT, 2006. Springer, Heidelberg

Microsoft SQL Server 2008 R2 Data-Tier Applications (2010) http://msdn.microsoft.com/en-us/library/ee240739(SQL.105).aspx

Miller R, Ioannidis YE, Ramakrishnan R (1994) Schema equivalence in heterogeneous systems: Bridging theory and practice. Inform Syst 19(1):3–31

Moto MM, Malaika S, Lim L (2007) Preserving XML queries during schema evolution. In: Proceedings of WWW conference. ACM, NY, pp 1341–1342

Noy NF, Klein M (2004) Ontology evolution: Not the same as schema evolution. Knowl Inform Syst 6(4):428–440

Noy NF, Musen MA (2002) PromptDiff: A fixed-point algorithm for comparing ontology versions. In: Proceedings of the national conference on artificial intelligence. American Association for Artificial Intelligence, CA, pp 744–750

Noy NF, Kunnatur S, Klein M, Musen, MA (2004) Tracking changes during ontology evolution. In: Proceedings of international semantic web conference (ISWC). Springer, Heidelberg, pp 259–273

Noy NF, Chugh A, Liu W, Musen, MA (2006) A framework for ontology evolution in collaborative environments. In: Proceedings of international semantic web conference (ISWC). Springer, Heidelberg, pp 544–558

Oracle Database 10g Release 2 (2005) Online data reorganization & redefinition, white paper. May 2005

Oracle Edition-Based Redefinition (2009) Whitepaper. Available at http://www.oracle.com/technology/deploy/availability/pdf/edition_based_redefinition.pdf

Oracle XML Schema Evolution (2008) Chapter 9 of Oracle XML DB, Developer's Guide, 11g Release, May 2008

Pal S, Tomic D, Berg B, Xavier J (2006) Managing collections of XML schemas in Microsoft SQL Server 2005. In: Proceedings of EDBT conference. Springer, Heidelberg, pp 1102–1105

Papastefanatos G, Vassiliadis P, Simitsis A, Aggistalis K, Pechlivani F, Vassiliou Y (2008) Language extensions for the automation of database schema evolution. In: Proceedings of the 10th international conference on enterprise information systems (ICEIS). INSTICC, pp 74–81

Papastefanatos G, Vassiliadis P, Simitsis A, Vassiliou Y (2010) HECATAEUS: Regulating schema evolution. In: Proceedings of ICDE, pp 1181–1184

Papavassiliou V, Flouris G, Fundulaki I, Kotzinos D, Christophides V (2009) On detecting high-level changes in RDF/S KBs. In: Proceedings of 8th international semantic web conference (ISWC). Springer, Heidelberg, pp 473–488

Plessers P, De Troyer O (2005) Ontology change detection using a version log. In: Proceedings of 4th international semantic web conference (ISWC). Springer, Heidelberg, pp 578–592

Rahm E (2011) Towards large-scale schema and ontology matching. In: Bellahsene Z, Bonifati A, Rahm E (eds) Schema matching and mapping, Data-Centric Systems and Applications Series. Springer, Heidelberg

Rahm E, Bernstein PA (2001) A survey of approaches to automatic schema matching. VLDB J 10(4):334–350

Rahm E, Bernstein PA (2006) An online bibliography on schema evolution. SIGMOD Rec 35(4):30–31

Smith B, Ashburner M, Rosse C et al (2007) The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. Nat Biotechnol 25(11):1251–1255

Software AG (2006) Tamino XML schema user guide 4.4.1. http://documentation.softwareag.com/crossvision/ins441_j/print/tsl.pdf

Stojanovic L, Maedche A, Motik B, Stojanovic N (2002) User-driven ontology evolution management. In: Proceedings of 13th international conference on knowledge engineering and knowledge management. Springer, London, pp 285–300

Su H, Rundensteiner E, Kramer D, Chen L, Claypool K (2001) XEM: Managing the evolution of XML documents. In: Proceedings international workshop on research issues in data engineering (RIDE). IEEE Computer Society, Washington, DC

Türker C (2000) Schema evolution in SQL-99 and commercial (object-) relational DBMS. Database schema evolution and meta-modeling. LNCS, vol 2065. Springer, Heidelberg, pp 1–32

W3C (2006) XML schema versioning use cases. Framework for discussion of versioning, 2006. http://www.w3.org/XML/2005/xsd-versioning-use-cases

W3C (2010) XML component designators, 2010 http://www.w3.org/TR/xmlschema-ref/

Yu C, Popa L (2005) Semantic adaptation of schema mappings when schemas evolve. In: Proceedings VLDB conference. VLDB Endowment, pp 1006–1017