Managing Schema Evolution in a Federated Spatial Database System

Xiaoying Wu
Curtin University of Technology, Australia

Jianhong (Cecilia) Xia
Curtin University of Technology, Australia

Geoff West
Curtin University of Technology, Australia

Lesley Arnold
Landgate, Australia

Bert Veenendaal
Curtin University of Technology, Australia

ABSTRACT
A federated spatial database is the integration of multiple spatial data sources and the realisation of effective spatial data sharing. However, in a federated database environment, database schemas are subject to change and the management of these changes is complex and inefficient. This is because schema changes in one local database will result not only the applications built against this local schema becoming invalid, but also applications built against the global schema.

In this research, an Automatic Schema Evolution (ASE) Framework has been developed to more effectively manage schema evolution in a federated spatial database system (FSDBS) automatically. This framework provides a Schema Element Dependency (SED) metamodel, a set of Spatial Schema Change Operations and incorporates view generation and rewriting, and query rewriting as solutions. These methods combined, ensure user applications are immune to schema evolution.

INTRODUCTION
A spatial database schema is the description of the structure and behaviours of a spatial database. Spatial database schemas are subject to change due to changes in how reality is modelled and end-user requirements. Changes also occur as a consequence of integration with other systems, compliance to new regulations and the implementation of new security requirements.

The task of managing schema evolution is to ensure schema, data and application consistency and integrity after change. Schema changes often result in the applications built against the schema being invalid due to loss of data. This is particularly the case in federated database environments.

In a federated spatial database system, spatial data are shared by multiple organizations and user applications. Therefore, a local database schema change will affect not only the database locally but also the integrated schema and thus the applications built on the federated schema. Effective management of schema evolution is problematic and there are several challenges to managing schema evolution in a federated spatial database systems including:
how to evaluate the impact of schema changes on local and global databases,
how to propagate the schema change across databases in a federated environment, and
how to automatically rewrite run time queries, and rewrite and generate views (spatial views and traditional views) so the user applications are immune to schema changes.

Effective method to managing schema evolution in a federated environment is much sort after. Effective management implies schema evolution transparency to end-users, avoidance of loss of data, increase in the reusability of spatial data, ensuring the longevity of GIS products and services, and improving spatial data management and collaboration within and between organisations.

The objective of this chapter is to explore an Automatic Schema Evolution (ASE) framework for managing schema evolution in a federated spatial database system. The approach seeks to manage evolving database schemas so that applications will be immune to change.

In order to achieve this, mismatch between evolved schema and queries need to be overcome. This includes queries generated by applications and stored queries (views and spatial views) as seen by users. The ASE framework defines the components and methods for automatic run time query rewriting and view rewriting, as well as, new view generation when a schema changes.

This chapter includes the following: (i) Section BACKGROUND: information on the definitions and concepts embedded in the conceptual ASE framework; (ii) Section MAIN FOCUS: the development of the ASE framework and its component parts; (iii) Section CONCLUSION & FUTURE WORK: conclusion and future research.

BACKGROUND

In order to help the reader to better understand the Automated Schema Evolution Framework, this section describes the concepts involved in the framework and how they apply to managing schema evolution in a Federated Spatial Database System (FSDBS).

Spatial DBMS

A Spatial Database Management System (SDBMS) is a database system that deals with data related to objects in space. With the development of database technologies, spatial data and spatial query are supported by Relational Database Management System (RDBMS), which have extended spatial capabilities to enable spatial and non-spatial data to be stored and managed in one database system (Rigaux, Scholl, & Voisard, 2002).

In a basic relational data model, only primitive data types are supported, such as string, numeric and date. In order to model spatially referenced objects new data types need to be introduced. According to Güting (1994), a spatial database system has all the advantages of a database system with the added capability that spatial data types (SDT), such as point, line and polygon, are supported in the data model for modelling geometric objects. Spatial query language is also supported along with spatial indexing and algorithms to process spatial queries and spatial data.

Spatial data type is an abstract data type (ADT) that provides an abstract view of objects. It defines a set of operations on a set of objects with similar behaviours or same type. The actual data structure and implementation are hidden from users. It is an implementation independent data type. The implementation of SDT may be different depending on the underlying DBMS. Data models that support SDTs are called spatial data models (Rigaux et al., 2002).

The Open GeoSpatial Consortium (OGC) has developed the Simple Features Access (SFA) specification, also called ISO 19125. This specification defines a geometry object model and its SQL implementation. The geometry object model defines a set of spatial data types such as
point, curve and surface; the relationships between them, and the functions and methods for each type that are used for the storage, query and analysis of geometry (OGC, 2006a).

The SQL implementation defines the SQL schema for storage, query, retrieval and update of spatial data via SQL interface (OGC, 2006b). With the development of database technologies, user-defined data types (UDT) that define complex data structures are supported in object-relational and object-oriented database systems. With this capability, geometry data types can be extended in a DBMS.

The OGC SFA specification defines two types of SQL implementations. One implements the primitive data types in the database such as binary data type; the other supports the implementation of the extended geometry data types in the database system (OGC, 2006a, 2006b).

Spatial database systems are distinguished from traditional (non-spatial) database systems in two ways (Yeung & Hall, 2007):

- the capability to store spatial data such as objects modelled as point, line and polygon; and
- the functionality required to process spatial data such as support of spatial indexes, extended spatial query language and effective algorithms for spatial operations.

**Federated Spatial Database System**

Spatial data sharing is becoming more and more important in social, economic and political decision making. Data sharing is motivated by (i) cost savings on data collection and maintenance, and software; (ii) improved spatial data quality and consistency; and (iii) enhanced inter-organisational relationships and best use of scarce GIS resources.

The aim of spatial data sharing is to realise interoperability among different database systems at the data, applications and business levels (Harvey & Tulloch, 2006). Currently, the formats and data structures of spatial data are often different. This is due to different perceptions of the real world by end-users, different sources and types of spatial data acquisition, varying levels of capture precision, variations in the use of terminology, and different levels of abstraction and data model adopted. Therefore, database mediation and information brokering methods are often used for data sharing at the data, applications and business levels (Yeung & Hall, 2007).

Spatial data federation is one of the approaches to realise spatial data sharing. It has been an active research topic in dealing with database heterogeneity since it was proposed (Litwin, Mark, & Roussoupoulos, 1990; Sheth & Larson, 1990).

In 2011, federated database environments are now more common and are used effectively share spatial data. This has been achieved through advancements in network, communication and distributed computing technologies.

A federated spatial database is a virtual integrated database system realised by integrating multiple autonomous spatial data sources into a single federated database and providing a unified data access mechanism (Yeung & Hall, 2007). It hides the heterogeneity of the component data sources from end users and provides access to multiple data sources at the same time. The features of a federated database system include:

- **Autonomy**, which means each component data source is independent in its own domains and can be operated on independently by local users. At the same time, each data source can cooperate with other systems to provide data and information to global users.

- **Heterogeneity** is realised by the different data semantics, data structures, and query languages and data models adopted.
Distribution, which means those multiple component data sources are stored in different geographical locations but are interconnected via a network (Sheth & Larson, 1990)

Building a federated spatial database system can produce significant cost savings for data acquisition, data management and software development; and can improve the value of data by sharing through cooperation. The benefits of such a database system include:

- Realisation of data sharing across databases.
- Avoid replication.
- Keep consistency between databases and applications.
- Simplicity for application development.
- Enhanced availability and reliability.
- Improved flexibility.

However, the federation of spatial databases is more challenging than with non-spatial databases. Data involved in the federation include vector, raster and non-spatial data. In order to federate spatial data, the schema level correspondences have to be built and then the corresponding instance level detected. For example, the same object (instance) in different datasets are matched (Goesseln & Sester, 2003).

While the methodology to federate spatial data is out of the scope of this research, schema evolution management will be evaluated in a complex federated spatial database environment using existing federated system technology.

**Query & Spatial Query**

In a DBMS, the retrieval of information from database is achieved through queries. Applications access a database in the form of queries. The queries are then sent to the DBMS server and processed there to retrieve data from the database. Simply speaking, a query is a question imposed on data (Ramakrishnan & Gehrke, 2003). In a relational DBMS, a query is a function to extract information from a relational database and produces a result relation consisting of a set of tuples.

Queries languages are needed to express queries. The Structured Query Language (MySQL) is the most popular query language in commercial DBMSs. SQL is a declarative language where the users only specify what they want and do not have to worry about how to get it. The basic form of a SQL query is Select-From-Where.

In addition to SQL, there are two other paradigms of query languages developed for relational databases (Abiteboul, 1995): relational algebra and relational calculus. Each of these languages can be translated into another for most expressions. This is because relational algebra and relational calculus are logically equivalent and they can be translated into SQL (Date, 2003).

Relational algebra queries are expressed by a set of algebra operators along with relations. Operations in relational algebra include set operations and tuple operations. Set operators include union (U), intersection (∩), and difference (−). Tuple operators include selection (σ), projection (π), join (⋈), division (/) and rename (ρ).

Relational calculus is a subset of predicate calculus. A relational calculus query consists of variables and a formula that describes the variables. The result of a calculus query is obtained when the evaluation of the formula is true. The formula includes atomic operators (>, <, =, ≤, ≥, ≠), logical connectives such as ∧ (and), ∨ (or) and ¬ (negation), and quantifiers ∃, ∀.

A spatial query is a special type of query and is more complex than a non-spatial query. This is because it deals with data of 2 or 3 dimensions. The features of a spatial query include: the support of spatial data types, and the support of spatial operators and spatial joins.
Types of spatial operators classified by OGC (2006) include: basic operators, query operators and analysis operators. Basic operators deal with the general properties of a geometric object. Examples of such operators are boundary, envelop and spatial reference. Query operators test spatial relationships and include overlap, touch and cross between geometric objects. Analysis operators are used to perform spatial analysis such as distance, difference, buffer and so on. Like a join in a regular relational query, a spatial join is an important part of a spatial query. Spatial joins compare two or more feature classes in terms of their location using spatial operators, such as overlay, cross and difference (Yeung & Hall, 2007).

**Views & Spatial Views**

In a relational DBMS, a view is defined as a stored query, which can be accessed as a derived table containing result sets of the query. Views can be used to provide flexible representation of the underlying database so that cohabitation of the database schema is supported (Claramunt & Mainguenaud, 1995). Views also ensure the security of the database by providing users with only data of interest and concealing the rest (Date, 2003)

A spatial view is an extension of a classical view. In a spatial database, a view containing a spatial column is called a spatial view. Spatial views are created the same way as traditional views. A spatial view is a stored spatial query which contains spatial columns and spatial operations. It involves spatial selection and spatial join on the underlying spatial data and has the capability to retrieve the results for applications to display.

**Query Rewriting**

In a relational DBMS, a query from an application is written against the logical schema of the database and is called the logical query (Popa, 2001). This high level query is then processed by Query Processing and Optimization in the DBMS and the result of the query is retrieved. The process of a query includes the following procedures: query parsing and translating, query optimisation, and, query execution (Date, 2003). The steps to process a query in relational DBMSs are depicted by Figure 1.

This process applies to both spatial data and non-spatial query processing. When a logical query access the database in a query language such as SQL, the syntax of the query is checked and the query is converted into an internal form that can be manipulated by the DML processor. The internal form is then further translated into a canonical form - normally relational algebra or relational calculus. The optimizer then generates and evaluates different equivalent query plans according to the actual physical storage, access path and index information provided by the metadata and evaluation formulas. The most efficient plan will then be chosen. At this stage, a logical query is converted into an equivalent counterpart - a physical plan. Finally, the chosen query is executed to retrieve the results from the database and return it to the application (Date, 2003).

**Figure 1: Query Processing**

In a federated database system, the steps for processing a query against the federated database include: firstly, when the query comes to the federated database, it is decomposed into subqueries that relate to the local schemas according to the mapping information between the global schema and local schemas stored in the system catalogue; secondly, the subqueries are sent to the related local databases and processed there; finally, the local databases send back the results to the federated database where all results are combined and then returned to the application (Sheth & Larson, 1990).

Query rewriting is the process of rewriting query statements with different expressions while still keeping the logical structure of the query. Equivalent query rewriting occurs when the result of a rewritten query is equivalent to the results of the original query. Query rewriting has been widely studied for different applications such as query optimisation, data integration,
data exchange and schema evolution (Y. H. Alon, 2001). For example, rewriting queries with materialised views has attracted attention for performance improvement of a database system. Different query rewriting algorithms have been developed to rewrite queries with views (Y. L. Alon, Anand, & Joann, 1996; Oliver & Michael, 1997; Popa, 2001). Rewriting queries against one schema into equivalent ones against another schema has been adopted as a solution to schema integration, schema transformation and schema evolution (Carlo A. Curino, Moon, Tanca, & Zaniolo, 2008; Deutsch, Popa, & Tannen, 2006; Gio, 1992).

Schema Mapping
Schema mapping specifies the correspondence between schemas. A schema mapping can be seen as a binary relation on instances of database schemas. More specifically speaking, schema mappings are a set of mapping rules between two schemas, which maps one schema to another. Schema mapping specifies the relationships between two schemas including both structural and semantic relationships.

Schema mapping is fundamental for query rewriting. In an integrated system, the global schema provides a unified interface to end users. A query imposed on the global schema is typically decomposed into a number of sub-queries imposed on those local data schemas. The decomposition here depends on the mapping information typically stored as metadata.

Schema mapping can also provide substituent query plans. For example, suppose that there are two schemas that have some part overlapping each other - this means that a schema mapping exists between them. If some deletion occurs to the overlapping part in one schema, then queries imposed on that part can be rewritten against the schema as defined by the mapping.

In data integration, views are often used to represent the mapping between schemas. There are two main approaches: Global-as-view (GAV) and Local-as-view (LAV) (Andrea, Diego, Giuseppe De, & Maurizio, 2002). GAV represents the global schema in terms of local schemas while LAV represents local schemas in terms of the global schema.

The form of a GAV mapping is: $\forall x(\varphi(x) \rightarrow P(x))$, where $\varphi(x)$ is a conjunction of atoms over source schema and $P(x)$ is the atom over the target schema.

The LAV mapping is of the form: $\forall x(Q(x) \rightarrow \exists y \psi(x, y))$, where $Q(x)$ is the atom over the source schema and $\psi(x, y)$ is the conjunction of atoms over the target schema.

The advantage of GAV is simplicity. Query rewriting under this approach can be achieved simply by unfolding. However, GAV doesn't have same extensibility as LAV (Andrea et al., 2002).

GAV and LAV can be generalised into a mixed approach, Global-local-as-view (GLAV) mapping, also known as tuple-generating-dependency (Maurizio, 2002). The form of GLAV is $\forall x(\varphi(x) \rightarrow \exists y \psi(x, y))$ which has more expressive power than GAV and LAV. GAV and LAV are special cases of GLAV.

In a federated database system, schema mapping consists of (i) mapping between local database schemas and the federated schema, and (ii) mapping between different versions of one system.

Related Work
Schema evolution has been widely studied (Bounif & Pottinger, 2006; Zhou, Rundensteiner, & Shin, 1997 1997). Research on the impact of schema changes has generated different methodologies and tools to identify the impact of schema evolution on applications (Karahasanovic & Sjoberg, 2001; Maule, Emmerich, & Rosenblum, 2008; Sjoberg, 1993).

Rather than modifying the code in applications, different approaches have been developed to provide transparent evolution to users. For example, schema versioning has developed a
version-control mechanism where different versions of schema and their corresponding data exist concurrently so applications against different schemas are left untouched (Monk & Sommerville, 1993; Roddick, 1995).

In addition, a view mechanism, based on the concept of versioning, is another approach that provides schema evolution transparency to end users (Bellahsene, 1996; Young-Gook & Rundensteiner, 1997). In this approach, different views are created for particular applications. When a change is needed for an application, only the view corresponding to the application will be updated to reflect the change requirement. Other views remain unchanged and thus other applications are also unchanged.

Curino et al. (2009) automated schema evolution for the Wikipedia system by adopting a query rewriting approach. A predictive approach has been proposed by Bounif (2006), which makes a database schema ready for evolutions by anticipating further possible change requirements before they occur.

In a federated database system, schema evolution in one database will also affect other databases as the schema mapping between the two schemas will become invalid. To overcome this inconsistency, a few approaches have been developed. Among them, is an approach that regenerates a schema mapping using mapping tools.

Version Management has also been proposed in a federated system to propagate a new version of schema between local schema and federated schema (Schonhoff, Strassler, & Dittrich, 2001). However, the approach has a few drawbacks. Firstly, it is a time consuming process and costly in terms of human effort; and secondly, the semantics of the original mapping may become lost.

In order to overcome these drawbacks, an incremental mapping adaptation approach has been developed in which the original mapping will be taken into consideration. For example, a metamodel of MoCAs (Model Correspondence Assertions) has been developed. This relates one schema to other schemas in a federated system (Busse & Pons, 2001). Schema evolution can be propagated by means of the MoCAs model along with the evolution actions and their specifications. An incremental schema integration approach can propagate change of the local schema to the federated schema by identifying sub-schemas affected and that are then re-integrated that (Motz, 2005).

Another approach for schema mapping adaption is to treat the schema change itself as the mapping (Carlo A. Curino et al., 2008). In this case, any schema change corresponds to a certain mapping rule. Schema mapping adaption can then be achieved by composing the original mapping and the mapping of schema change. Moreover, the semantics of schema mapping can be preserved in this approach.

Most research on schema evolution only focuses on the theoretical level. There is still lack of automatic procedure and schema evolution still requires a high degree of manual work. In addition, very little research has been done on schema evolution in a spatial database system, particularly a federated spatial database system. As a consequence this PhD research focuses on managing schema evolution in a federated spatial database system in an automatic/semiautomatic manner.

A Schema Element Dependency metamodel and a rich set of schema change operations based on the spatial data model change taxonomy have been developed. These are the basis for schema change impact analysis, view generation and rewriting, and query rewriting. The research incorporates a

devices on query rewriting and mapping, so that schema evolution can be managed effectively in a federated spatial database system. The conceptual framework is explained in the following section.
MAIN FOCUS OF THE CHAPTER

This section explains the conceptual Automated Schema Evolution (ASE) Framework and its component parts.

Overview of the ASE Framework

The main task of the ASE framework is to provide application immunity to schema changes in a federated spatial database system. The main components included the global and local databases, the Metadata Repository, the SED Metamodel, Schema Change Operations, view generation and rewriting, query rewriting, schema mapping composition and adaptation as illustrated in Figure 2.

Figure 2: the ASE Framework for Managing Schema Evolution in a Federated Spatial Database Environment

There are three layers involved in the ASE framework. They are separated by the horizontal dashed lines and include: the database layer, the middle layer and the user interface.

The database layer includes the global database and all local spatial data sources participating in the federation. A metadata repository is also included to store all metadata needed for this framework.

The middle layer provides application immunity to changes that occur in the database layer. Two scenarios are included: (i) view generation or query rewriting for applications built on base tables to keep applications unaffected by database changes; and (ii) view redefinition against the new schema for applications built against a view to keep consistency between the view and the evolved database schema.

The user interface layer mainly deals with processes involving interaction with the DBA. It facilitates the DBA to perform and manage schema changes. The DBA can specify the schema change and perform the schema change analysis. Once a schema change is initiated, scripts for the schema change and data migration can be generated automatically. Schema version mapping is also generated according to the mapping rules defined by schema change operations.

The steps to perform a schema change include:

1. **Schema change analysis**: By specifying the schema change and consulting Schema Element Dependency information, the DBA evaluates the impact of the schema change.

2. **Schema change design**: If the DBA decides to move forward with the change planned, the schema change can then be implemented in terms of Schema Change Operations. Each Schema Change Operation represents a certain change in a spatial database. A schema change operation might consist of one or more primitive schema changes. The Schema Change Operations also capture the semantics of changes. A schema version change is achieved by specifying the sequence of Schema Change Operations.

3. **Schema change implementation**: Scripts for each Schema Change Operation are generated in order to implement schema changes.

4. **Schema mapping generation and composition**: The schema change operations produce a logical mapping between schema versions. All schema change and schema version mapping metadata is stored in system table as a guide to view and query rewriting. Each Schema Change Operation for one schema change may yield a mapping constraint between two versions. So for a schema change, which involves a
number of schema change operations, mapping composition is exploited to derive the 
mapping between the schemas before and after the change. By doing this, space used 
to store mapping information can be reduced and the semantics for mapping can be 
reserved.

5. **Mapping validation**: Mappings between schema versions can be overridden by users. 
   Users can review and check the mapping to validate.

6. **View rewriting**: According to Schema Element Dependencies and schema versions 
mapping, views that have dependencies on the changed schema elements are 
rewritten into the equivalent expressions under the new database schema.

7. **Schema Element Dependency update**: After a view is redefined, the dependency 
   information is recomputed according to the previous dependency and the schema 
   change. The new dependency information replaces the old one stored in the spatial 
database.

8. **Query rewriting**: For applications built on base tables, schema changes may break the 
   link between queries and the old version of schema. This will result in the removal of 
   features from the map. In order to solve this, run time query rewriting is required. 
   When a query against an old schema is processed, the query will be translated into an 
equivalent query expressed against the new evolved schema by means of query 
rewriting and according to the schema version mapping

Spatial Databases
In local databases, there are base tables that store data, views (both spatial and non-spatial), 
and, system tables that provide schema information. Similarly, the global database consists of 
tables, views and system tables. However, there is no real data stored in the tables since the 
federated database is a virtual database. System tables in the global database provide the 
global schema information as well as mapping between the global and the local schemas.

Schema Change Operations
A key part of this research is to identify the schema change scenarios that can occur in a 
spatial database environment and generate a corresponding rich set of schema change 
operations(or templates) to satisfy the many different schema changes. These schema change 
operations can be specified at simple feature object spatial data model level first and then 
mapped into schema changes on the underlying RDBMS according to the mapping rules 
between these two data models. The mapping between the Simple Feature Spatial Data Model 
and the underlying Relational Data Model is shown in Table 1. More details with regards to 
these two models are explained in SDE Metamodel.

**Table 1: Mapping between the Spatial Data Model and the Relational Data Model**

<table>
<thead>
<tr>
<th>Spatial Data Model</th>
<th>Relational Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Class</td>
<td>Table (one to two tables depending on the data types in the underlying RDBMS)</td>
</tr>
<tr>
<td>Non-spatial Object Class</td>
<td>Table</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column</td>
</tr>
<tr>
<td>Relationship</td>
<td>FK and PK (1:1 or 1:M)</td>
</tr>
<tr>
<td></td>
<td>Table with two Foreign Keys (M:N)</td>
</tr>
</tbody>
</table>

The schema change taxonomy in a spatial data model and the implementation of 
corresponding change in the underlying RDBMSs are listed in Table2. The schema change 
operations, according to the taxonomy, are generalised in Table3. Each schema change 
operation consists of one or more ordered schema primitive changes, as well as, data 
migration. Schema primitive changes include addition or deletion of attributes, addition or 
deletion of feature classes etc.
Schema change operations also capture the semantics of change. Any schema change can be achieved by employing one or more schema change operations. Furthermore, schema change operations specify whether or not a view should be generated. For example, after renaming a table, a view with the same name of the original table can be generated. In order to simplify schema changes, each schema change operation can generate scripts automatically to implement schema change and data migration. In this research, schema change operations are the basis for schema versioning mapping because schema change is treated as the mapping. This will be explained later.
<table>
<thead>
<tr>
<th>Feature Class</th>
<th>Modification</th>
<th>Underlying RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Rename a Feature Class</td>
<td>Rename a Table</td>
</tr>
<tr>
<td></td>
<td>Add an Attribute</td>
<td>Add a Column</td>
</tr>
<tr>
<td></td>
<td>Delete an attribute</td>
<td>Delete a column</td>
</tr>
<tr>
<td></td>
<td>Rename an attribute</td>
<td>Rename a column</td>
</tr>
<tr>
<td></td>
<td>Change the data type of an attribute</td>
<td>Data type change</td>
</tr>
<tr>
<td></td>
<td>Set a default value to an attribute</td>
<td>Add default value</td>
</tr>
<tr>
<td></td>
<td>Change the default value of an attribute</td>
<td>Modify default value</td>
</tr>
<tr>
<td></td>
<td>Delete the default value of an attribute</td>
<td>Delete default value</td>
</tr>
<tr>
<td>Deletion</td>
<td>Rename a Feature Class</td>
<td>Rename a Table</td>
</tr>
<tr>
<td></td>
<td>Add an Attribute</td>
<td>Add a Column</td>
</tr>
<tr>
<td></td>
<td>Delete an attribute</td>
<td>Delete a column</td>
</tr>
<tr>
<td></td>
<td>Rename an attribute</td>
<td>Rename a column</td>
</tr>
<tr>
<td></td>
<td>Change the data type of an attribute</td>
<td>Data type change</td>
</tr>
<tr>
<td></td>
<td>Set a default value to an attribute</td>
<td>Add default value</td>
</tr>
<tr>
<td></td>
<td>Change the default value of an attribute</td>
<td>Modify default value</td>
</tr>
<tr>
<td></td>
<td>Delete the default value of an attribute</td>
<td>Delete default value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-spatial Object class</th>
<th>Modification</th>
<th>Underlying RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Rename feature class</td>
<td>Rename a table</td>
</tr>
<tr>
<td></td>
<td>Add an attribute</td>
<td>Add a column</td>
</tr>
<tr>
<td></td>
<td>Delete an attribute</td>
<td>Delete a column</td>
</tr>
<tr>
<td></td>
<td>Rename an attribute</td>
<td>Rename a column</td>
</tr>
<tr>
<td></td>
<td>Change the data type of an attribute</td>
<td>Data type change</td>
</tr>
<tr>
<td></td>
<td>Set a default value to an attribute</td>
<td>Add default value</td>
</tr>
<tr>
<td></td>
<td>Change the default value of an attribute</td>
<td>Modify default value</td>
</tr>
<tr>
<td></td>
<td>Delete the default value of an attribute</td>
<td>Delete default value</td>
</tr>
</tbody>
</table>

| Join a feature class with an object class | Join two tables |
| Join two object classes | Join two tables |
| Decompose an object class into two object classes | Split one table into two |
| Decompose a feature class into one feature class and one object class | Split one table into two |
| Move an attribute | Move a column from one table to another table |

<table>
<thead>
<tr>
<th>Relationship Class</th>
<th>Underlying RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a relationship</td>
<td>Add a foreign key column (1:1 or 1:M) or add a relationship table with two foreign keys (M:N)</td>
</tr>
<tr>
<td>Delete a relationship</td>
<td>Delete the foreign key column or the relationship table</td>
</tr>
<tr>
<td>Modify the cardinality</td>
<td>Add or delete the relationship table</td>
</tr>
</tbody>
</table>
Table 3: Schema Change Operations

<table>
<thead>
<tr>
<th>Schema Operation</th>
<th>Input Schema</th>
<th>Output Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a column</td>
<td>R(A)</td>
<td>R(A,b)</td>
</tr>
<tr>
<td>Add a table</td>
<td>T(A)</td>
<td></td>
</tr>
<tr>
<td>Rename a table</td>
<td>R(A)</td>
<td>T(A)</td>
</tr>
<tr>
<td>Merge tables</td>
<td>R(A), S(A)</td>
<td>T(A,b)</td>
</tr>
<tr>
<td>Join tables</td>
<td>R(a,B), S(a,C)</td>
<td>T(a,B,C)</td>
</tr>
<tr>
<td>Decompose tables</td>
<td>R(a,B,C)</td>
<td>S(a,B), T(a,C)</td>
</tr>
<tr>
<td>Move a column</td>
<td>R(a,B,C), S(a,D)</td>
<td>R(a,B), S(a,c,D)</td>
</tr>
<tr>
<td>Split a column</td>
<td>R(a,B)</td>
<td>R(c,d,B)</td>
</tr>
<tr>
<td>Merge columns</td>
<td>R(a,b,C)</td>
<td>R(d,C)</td>
</tr>
</tbody>
</table>

A,B,...: a set of attributes;

a,b,...: a single attribute

1 In order to be able rewrite queries, a column is added as the condition when two tables are merged;

2 To avoid data loss, when join tables, outer join is needed

Metadata Repository

Metadata is treated as the first class for schema evolution. Currently, the most common strategy for metadata management is to create a repository to store metadata called the metadata repository. The metadata repository can store different types of metadata such as structural, descriptive and administrative metadata.

A metadata repository provides a consistent and unified access mechanism to data so as to improve the effectiveness of information management. It is not only used to store metadata, but also to perform other functions such as metadata definition; metadata transformation; metadata mapping; metadata synchronization; metadata versioning; impact analysis of metadata change etc. (Sen, 2004).

There are four parts to the metadata stored in the metadata repository. They are (i) metadata of schema element dependency; (ii) schema mapping; (iii) schema change history; and (iv) original relational and geo-database system tables (views) from which the schema element dependency and mapping are derived.

System Tables (Views)

Relational database system tables (views) in a RDBMS are often called the data catalogue and store metadata related to the relational database. The metadata contains information on the database schema such as database tables, views, attributes, relationships, indexes and security.

The federated spatial database system tables (views) may also store metadata relating to the mapping between the federated database schema objects and the local database schema objects.

The SDE Metamodel and SDE metadata

The Schema Element Dependency (SED) Metamodel (illustrated in Figure 3) is used to represent column level schema element dependency in a federated spatial database system. The graph of this metamodel is depicted using UML syntax. This conceptual metamodel consists of three parts. The bottom part defines the schema elements of a spatial data model. The middle part defines the schema elements of the relational data model. The top part
represents the schema element dependency. The three parts of the metamodel will be explained below.

**Figure 3: SED Metamodel**

The spatial data model is an abstract data model. In a spatial data model, schema elements include feature classes, non-spatial object classes, attributes and relationships. Feature class represents a collection of features of the same type. Non-spatial object class is the abstraction of a category of objects with no spatial features. Object class consists of feature class and non-spatial object class. An object class has names and are composed of attributes including spatial (for feature class) and non-spatial attributes. Each attribute has its own properties including name, data type etc. Data types of spatial attribute are spatial data types like point, line or polygon.

The middle part is the implementation model of a spatial data model in a relational database. In a relational data model, the schema elements mainly include tables and views. For a table, it contains name of the table, names of all columns and data types for these columns. Similarly, a view includes name of the view and names of columns because a view is a virtual table that is derived from base tables or views by means of stored queries. A view also includes other clauses in its definition such as `where`, `having` etc.

There are three types of tables in a relational DBMS: feature table, geometry table and non-spatial table. A feature table stores a collection of features of the same type. Attributes of a feature class including the geometry attribute are columns of a feature table. A geometry table stores the geometric information of the geometric objects. Tables with no any spatial element are called non-spatial tables.

The relationship between the spatial data model and the relational model can be seen in Figure 3. Object classes are stored as tables in the relational database. More specifically, the implementation of a non-spatial object class is a non-spatial table while the implementation of a feature class varies depending on the data types supported by the underlying relational DBMS. For example, a feature class will be implemented by two tables - feature table and geometry table with a key reference between them if predefined data types are used. However, only one table - feature table is needed, if geometry data types are supported by the underlying database. Attributes of an object class are stored as columns of the corresponding table. A relationship between two object classes is implemented as either a table or the primary key and foreign key column of two tables.

The top part shown in Figure 3 is the core of the whole metamodel. It builds on the system tables (views). The constructs here consist of schema elements and the relationships between them. Schema elements are divided into two parts: compound and part. The Part component denotes atomic elements such as column. The Compound component represents the elements which contains parts like tables and views. Relationships include Referential Integrity (RI), DerivedFrom and ReferencedBy. A relationship connects two schema elements. The dependency information which is termed a column level dependency can be derived from the relationships between the source and the target elements that are connected by the relationship. Depending on the databases containing the source and target elements, the dependency will be identified as an internal or external dependency.

In a database schema, some schema elements may refer to other element (elements) when they are defined. For example, a view is defined by a query which refers to another table (or tables) or another view (or views). Normally, the element that is being referred to is called a referenced element. The one which references the other element is called a dependent element, because it depends on the other element (or referencing element).
If one schema element is changed, the dependent elements may become invalid. For example, if a table is renamed, the views which are defined on this table will become invalid if the definitions of the views are not updated to the new name of the corresponding table.

There are two kinds of dependencies in the database schema, direct dependency and indirect dependency. Direct dependency is where the referenced element is referred by the dependent element directly. Indirect dependency is where the referenced element is referred to by an element which is referred to by another element.

The dependencies derived according to the SED Metamodel and the system metadata are direct dependencies. Indirect dependencies can be calculated iteratively based on the direct dependencies. SED provides the basis for impact analysis of schema change.

**Schema Mapping**

Schema in this research can be either global schema or local schema. Schema mapping stored in the metadata repository is divided into horizontal and vertical mapping. Horizontal mapping refers to mapping between different systems such as the global schema and a local schema while vertical mapping refers to schema version mapping of one database system after schema changes.

In this research, the approach of describing schema change as schema mapping is adopted. When a database schema changes, the schema change itself is treated as the schema mapping between the old schema and the new schema. Each change operation has a corresponding mapping rule between schema versions before and after the change.

The corresponding schema mapping of each schema change operation is shown in Table 4. In order to avoid loss of information, deletion of schema elements is not allowed. Under such a condition, the expressive power of GAV is enough for describing schema mapping where the source and target schemas are database schema after and before evolved respectively. Language used to describe mapping here is relational algebra but other representations could be used such as relational calculus. Schema change can be achieved by applying one or more schema change operations. All the mapping rules from schema change operations form the schema version mapping.

**Table 4: Schema mapping corresponding to schema change**

<table>
<thead>
<tr>
<th>Schema change</th>
<th>Schema Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a Column</td>
<td>R(A) ← π R(A,b)</td>
</tr>
<tr>
<td>Add a Table</td>
<td>R(A) ← T(A)</td>
</tr>
<tr>
<td>Rename a Table</td>
<td>R(A) ← σ (T(A,b)), S(A) ← σ(T(A,b))</td>
</tr>
<tr>
<td>Merge Tables</td>
<td>R(a,B) ← π T(a,B,C)</td>
</tr>
<tr>
<td></td>
<td>S(a,C) ← π T(a,B,C)</td>
</tr>
<tr>
<td>Join Tables</td>
<td>R(a,B,C) ← S(a,B) ( \bowtie ) T(a,C)</td>
</tr>
<tr>
<td>Decompose Tables</td>
<td>R(a,B,C) ← S(a,B) ( \bowtie ) T(a,C)</td>
</tr>
<tr>
<td>Move a Column</td>
<td>R(a,B,c) ← π σ ( R(a,B) ( \bowtie ) S(a,c,D))</td>
</tr>
<tr>
<td></td>
<td>S(a,D) ← π S(a,c,D)</td>
</tr>
<tr>
<td>Split a Column</td>
<td>R(a) ← F(R(c,d))</td>
</tr>
<tr>
<td>Merge Columns</td>
<td>R(a) ← F_1(R(d))</td>
</tr>
<tr>
<td></td>
<td>R(b) ← F_2(R(d))</td>
</tr>
<tr>
<td>Data Type Change</td>
<td></td>
</tr>
<tr>
<td>Change Default Value</td>
<td></td>
</tr>
</tbody>
</table>

* F(), F_1(), F_2() are functions for splitting or merging columns

Schema evolution can result in invalid schema mapping. In order to ensure the consistency of schema mapping, schema mapping composition is used to adapt schema mapping. Schema mapping composition is to compose two schema mappings into one. This approach can be
applied to vertical and horizontal mapping. When a data source schema changes, a new schema mapping between the global schema and the changed schema can be derived from the original mapping and schema changes (mapping) by schema mapping composition. Also, for the local schema, schema mapping between older versions and the current one can be derived from the latest schema version mapping and the old schema version mapping.

Let $M = (S, T, \Sigma)$ be a schema mapping model, where $S$ and $T$ are the source schema and target schema respectively. $\Sigma$ is a set of logical formulas over $<S,T>$. Let $M_{12}=(S_1,S_2,\Sigma_{12})$ and $M_{23}=(S_2,S_1,\Sigma_{23})$. Then the mapping between $S_1$ and $S_3$ $M_{13}=(S_1,S_3,\Sigma_{13})$ is the composition of $M_{12}$ and $M_{23}$ that can be denoted as $M_{13}=M_{12} \cdot M_{23}$. $M_{13} = \{(S_1,S_3) : \exists S_2 (S_1,S_2) \in M_{12}$ and $(S_2,S_3) \in M_{23}\}$.

**Schema Change History**

In the metadata repository, the schema change history keeps the record of each schema change. Each schema change operation yields a record of schema change history. The schema change history provides clear information of schema versions and how the schema evolves. The semantics of the schema change can be understood. Also, schema changes can be reversed by looking up the history records if any error occurs.

**Query/View Rewriting and View Generation**

For a spatial database system, the map view (applications) access data stored in the underlying relational database. Similar to other database systems, in a spatial database system, the users can build queries against the base tables or views (spatial views if there is a spatial column).

In order to manage schema evolution in a spatial database system, two kinds of scenarios have to be considered. One is a query using base tables and the other is a query using views. For a query using views, the views need to be redefined with the change of the schema. When a query is built against the base tables, two ways can be used in order to support old queries expressed against old schema. They are (i) query rewriting; and (ii) the generation of a new view.

View generation does not apply to all schema changes. Only certain changes can generate a view to keep the application immune to schema change such as the renaming of a table. View generation can be included in the schema change operation template.

Query rewriting in this research is to ensure the queries generated from the applications against the old schema can be translated into queries against the new schema. By doing this, the application will be still valid.

Once an application is built on views, changes of schema might invalidate the view definition. Therefore, view rewriting is to redefine and recompile the view after schema changes. Since a view is a stored query, view rewriting is very similar to query rewriting. However, there are differences between them.

Firstly, view rewriting is a one-off operation. It only occurs when schema changes whereas query rewriting happens every time the query is processed. This is because any affected view can be detected by SED whereas the queries from the application can only be detected when they are processed.

Secondly, view rewriting must ensure the view schema is unchanged as the change of the view schema will invalidate the applications built on that view. For example, the view name and its column names should not be changed. However, query rewriting only needs to provide equivalent results.

Different algorithms have been developed for query rewriting such as bucket algorithm, inverse rules and the C&B (Chase & BackChase) algorithm. Among them, the C&B algorithm has attracted lots of attention. It was first developed to rewrite queries under a set of
constraints in order to improve query optimisation (Deutsch, Popa, & Tannen, 1999; Deutsch et al., 2006). The set of constraints include both logical and physical schema constraints and can be classified as four types: key constraints; schema mapping constraints; domain constraints; and, materialised view constraints (Deutsch et al., 2006). All constraints can be expressed as DEDs (Disjunctive Embedded Dependency) which consist of disjunctive tgds (tuple-generating dependency) and egds (equality-generating-dependency) (Deutsch et al., 2006). Due to the wide range of constraints, C&B algorithm has been adopted and extended by other applications such as schema evolution, query optimisation and data exchange (Carlo A Curino et al., 2009; Deutsch & Tannen, 2003; Popa, 2001).

In this research, query rewriting can be achieved by simply unfolding. As mentioned before, the advantage of GAV is the simplicity for query rewriting. Also, the composition of two GAV mappings is still a GAV mapping. Therefore, schema mapping in this research is restricted in the form of GAV mapping. Under such circumstance, a query can be rewritten by unfolding (Cali, 2003). With substitution, queries expressed against the global schema can be rewritten as an equivalent one against the source schema. Similarly, queries expressed against old schema can be rewritten against the new schema.

**CONCLUSION & FUTURE WORK**

Managing schema evolution in a federated spatial database environment is a significant challenge. This research aims to develop the methodologies to manage schema evolution in a federated spatial database system environment automatically so that applications will be immune to schema changes. To achieve this, view generation/rewriting and query rewriting are proposed as the solution.

In order to implement the methodology, a SED Metamodel and a set of schema change operations are developed. The SED Metamodel defines the constructs to build column level dependency in the federated spatial database system.

Taking the input as the SED Metamodel and system tables of the spatial database, the schema element dependencies can be derived.

Schema element dependencies provide the basis for impact analysis and determine which view definition needs to be refined when the schema changes. The schema change operations define a rich set of schema change scenarios in a spatial database environment and provide the mapping rules for each change.

By incorporating the mapping composition, schema version mappings between the versions of a local schema can be derived. Schema mapping adaptation between the global and the local schema can be achieved as well. Schema change operations also provide the solutions on whether or not a view should be generated. Based on the schema mapping information, runtime queries and views that expressed against an old schema version can be rewritten into expressions in terms of the current schema by unfolding. The end result is that applications will be immune to schema changes.

Further research is required to test the effectiveness of the methods developed including the ASE Framework, SED Metamodel and SED metadata and spatial database schema change taxonomy. The following will be conducted:

- Embed the framework with tools to automate queries rewriting and views rewriting/generation
- Build a prototype federated geodatabase ArcSDE environment as the test environment. This includes building local spatial data sources and the federated database, creation of maps as the applications of the database. The test-bed data models and spatial data are provided with courtesy by Landgate
- Implementation and testing of the proposed methodology in the prototype system.
REFERENCES


KEY TERM & DEFINITIONS

**Federated Spatial Database**: a virtual integrated database system. It integrates multiple autonomous spatial data sources into a single federated database and provides a unified data access mechanism to end users.

**Spatial Query**: the type of query deals with data of 2 or 3 dimensions by supporting spatial data types, spatial operators, spatial operations and spatial joins.

**Spatial View**: an extension of a classical view which contains a spatial column.

**Schema Mapping**: specification of the correspondences between schemas

**Schema Mapping Composition**: to compose two schema mappings into one

**Query Rewriting**: the process to rewrite query statements with different expressions while still keeps the logical structure of the query.