RatSWD Working Paper Series

www.ratswd.de

RatSWD
German Data Forum

191

Representing and Utilizing DDI in Relational Databases

Alerk Amin, Ingo Barkow, Stefan Kramer, David Schiller, Jeremy Williams

January 2012

Federal Ministry of Education and Research

Working Paper Series of the German Data Forum (RatSWD)

The *RatSWD Working Papers* series was launched at the end of 2007. Since 2009, the series has been publishing exclusively conceptual and historical works dealing with the organization of the German statistical infrastructure and research infrastructure in the social, behavioral, and economic sciences. Papers that have appeared in the series deal primarily with the organization of Germany's official statistical system, government agency research, and academic research infrastructure, as well as directly with the work of the RatSWD. Papers addressing the aforementioned topics in other countries as well as supranational aspects are particularly welcome.

RatSWD Working Papers are non-exclusive, which means that there is nothing to prevent you from publishing your work in another venue as well: all papers can and should also appear in professionally, institutionally, and locally specialized journals. The *RatSWD Working Papers* are not available in bookstores but can be ordered online through the RatSWD.

In order to make the series more accessible to readers not fluent in German, the English section of the *RatSWD Working Papers* website presents only those papers published in English, while the the German section lists the complete contents of all issues in the series in chronological order.

Starting in 2009, some of the empirical research papers that originally appeared in the *RatSWD Working Papers* series will be published in the series *RatSWD Research Notes*.

The views expressed in the *RatSWD Working Papers* are exclusively the opinions of their authors and not those of the RatSWD.

Contact: German Data Forum (RatSWD) | Mohrenstraße 58 | 10117 Berlin | office@ratswd.de

Representing and Utilizing DDI in Relational Databases*

Alerk Amin[1], Ingo Barkow[2], Stefan Kramer[3], David Schiller[4], Jeremy Williams[5]

* Previously published at http://dx.doi.org/10.3886/DDIOtherTopics02.

Acknowledgments

The Leibniz Institute for Educational Research and Educational Information (DIPF) hosted a workshop including the topic of DDI in relational databases in Frankfurt, Germany, on April 7-8, 2011, during which the development of this document was begun. Thanks to the following colleagues for providing input on a late-Oct. 2011 draft: Sanda Ionescu, University of Michigan (USA); Jeremy Iverson, Algenta Technologies (USA); Johanna Vompras, University of Bielefeld (Germany); and to Mary Vardigan, University of Michigan (USA), who edited the final version of this paper previously published at http://dx.doi.org/10.3886/DDIOtherTopics02.

Abstract

This document is primarily intended for implementers of DDI-based metadata stores who are considering different technical options for housing and managing their metadata.

The Data Documentation Initiative (DDI) metadata specification[6] is expressed in the form of XML schema. With version 3, the DDI specification has become quite complex, including 21 namespaces and 846 elements[7]. Organizations employing DDI, or considering doing so, may want to

1. store and manage the metadata elements in relational databases, for reasons of integration with existing systems, familiarity with the concepts of relational databases (such as Structured Query Language), systems performance, and/or other reasons;
2. select only the subset of the available DDI metadata elements that is of utility to their work, and have the flexibility of capturing metadata they need that would not fit into the DDI model.

This paper discusses advantages and disadvantages of the relational database approach to managing DDI. It also describes methods for modeling DDI in relational databases and for formally defining subsets of DDI to employ in this environment.

1 CentERdata [a.amin@uvt.nl]
2 DIPF – Leibniz Institute for Educational Research and Educational Information [barkow@dipf.de]
3 Cornell Institute for Social and Economic Research [stefan.kramer@cornell.edu]
4 Research Data Center (FDZ) of the German Federal Employment Agency (BA) at the Institute for Employment Research (IAB) [david.schiller@iab.de]
5 Cornell Institute for Social and Economic Research [jw568@cornell.edu]
6 http://www.ddialliance.org/Specification
7 Numbers gleaned from DDI 3.1 field-level XML Schema Documentation: http://tinyurl.com/6sm2koq

Introduction

Data constitute a valuable, perhaps the most valuable, commodity in scientific research. Therefore, the potential for reusing generated data for future projects is an important consideration in the conduct of research. But data can only be reused if they can be sufficiently interpreted and understood, and that requires that they be well documented. The documentation challenge becomes even greater in comparative research when international standardization of documentation is required. A further challenge is the growing demand for merged datasets from different data sources. As a result, and to enable sound scientific research in the future, a documentation standard for research data that can address all these challenges is vital. The DDI metadata specification offers a solution, and many important data providers are already using DDI, or are about to use it. The DDI development is supported by an active community that steadily works on improvements.

DDI provides a means to represent metadata about data collected in the social sciences, and potentially other disciplines[8], in a meaningful and structured manner. The DDI Alliance[9], which develops the DDI specification and promotes its worldwide adoption and implementation, currently uses XML-based structures to describe the content of the model. For DDI version 2[10], a Document Type Definition (DTD) was used; this was changed to an XML Schema (XSD) for DDI version 3[11]. These schemas are employed to structure metadata content in the form of DDI instances. Essentially DDI can represent metadata in the form of XML files based on the DDI XML Schema stored on a common file share, or can be put into an XML database (like BaseX[12] or eXist[13]) to enable collaborative work with multiple users. Another possibility is to represent DDI in relational databases (RDBs).

It is obvious that DDI can only serve the scientific community if it is actively used by a sufficient number of stakeholders. In order to achieve this goal, the DDI-based documentation has to be easy to understand and easy to integrate into the existing data structure of the data providers. It also has to be compatible with future developments in the area of data storage. Relational databases are a widely used and flexible solution for data storage. Bringing DDI together with the capability of relational database systems will promote both data storage for the purpose of scientific research and the DDI standard itself.

8 See, for example: Documenting a Wider Variety of Data Using the Data Documentation Initiative 3.1: http://dx.doi.org/10.3886/DDILongitudinal01
9 http://www.ddialliance.org/alliance
10 http://www.ddialliance.org/Specification/DDI-Codebook/
11 http://www.ddialliance.org/Specification/DDI-Lifecycle/
12 http://basex.org/
13 http://exist.sourceforge.net/

This paper outlines the advantages and disadvantages of representing DDI in relational databases as an alternative to an XML structure. In addition, it discusses the benefits and drawbacks of using relational databases for the DDI model, gives some hints about future solutions, provides a short introduction on the topic of how to model DDI, discusses application compatibility, and points out some challenges in "advanced cases."

DDI in Relational Databases vs. XML: Pros and Cons, and Other Approaches

The idea of storing DDI instances in a relational database, as opposed to a XML database, is often a hot topic among developers. From the perspective of DDI solely as a "storage" standard, an XML database has certain advantages. But when thinking of DDI as a transport format between applications, the actual storage format for each application should be the one that best meets that application's needs. In many cases, a relational database is the better option. The following section of the paper demonstrates the advantages of using a relational database.

Representing the DDI model within a relational database

The first reason to consider a relational database model for DDI arises from an organizational point of view. Many agencies have been storing primary data and associated metadata for timespans measured in decades, and a very common storage method is the relational database, as its tabular structure is ideal for storing rectangular data resulting from data collection activities. Therefore, those agencies have a high level of expertise in using the relational database model. Changing their present table-based metadata standard (whatever that may be) to a DDI representation which is also table-based should thus be intuitive to them. Using XML for storage, on the other hand, might be problematic as these agencies do not have the experience or resources to convert the metadata and change the surrounding tools to the new structure. XML may be known to them, but mostly as an import or export format. They might therefore be reluctant to utilize DDI in XML format for reasons of transformation costs or leaving their area of expertise.

In addition to organizational considerations, there are also structural advantages to using a relational database. Therefore, agencies often represent their microdata internally in the form of a relational database as a central storing mechanism because it is ideal for processing rectangular data (e.g., SPSS data files, ASCII data files) in tables and can manage the file structures of multiple studies by input and output processes. If the metadata are stored in

the same database as the microdata, the movement from metadata to data output works seamlessly as native database methods such as connecting tables by referential integrity can be used. The metadata can be linked to the associated research data. A user can therefore first search the metadata and then move easily to the connected data. This model can even be extended to create custom data extracts (like a variable shopping basket), where an extract of the dataset, including the related metadata as a kind of codebook, can be selected and downloaded, e.g., via a Web interface. In an XML-based DDI environment this can also be done, but with much more effort, as two different structural models have to be merged. In a worst case scenario, an external service has to link between an XML metadata structure based on DDI and an ASCII file containing the microdata.

Relational databases have existed on the market for decades, and have led to the development of many tools for working with them. If one extends the idea of combining metadata and microdata into a relational database model, then the next step can be changing the database model into an analytical one. Relational databases can be enhanced to become analytical or multidimensional databases (e.g., online analytical processing [OLAP] cubes[14]). With this model, enhanced analytical or statistical methods from the area of Business Intelligence (e.g., data mining, process mining) can be applied to the data. These methods might lead to completely new research questions and new knowledge. This change of model would be difficult to realize in a complete XML-based environment.

A less complex example is storing more than one survey in a structure. In a relational database, the tabular structure can be designed to support multiple surveys in one structure by adding additional administrative tables. In a DDI structure based on XML files, this is difficult to represent; and it is difficult in an XML database, as the structure is largely based on the original DDI XML schema, which normally (as it is file-based) demands a separate XML file for each survey. In an XML database structure each survey on its own has to be represented as a separate XML database or at least as a separate instance of an XML database (if the XML database supports instances). The problem can be solved by adding additional programming routines surrounding the XML structure to emulate referential integrity by XML database linkage. Nevertheless, the relational database offers these possibilities natively or with much less effort.

Performance is not addressed in this paper, as the authors currently cannot prove that working with DDI in a relational database is always faster than in an XML database or XML file structure. The performance of DDI within different systems depends heavily on the structure used, and therefore benchmarking would not make much sense, as the results would not be representative of different instances or different database products. The authors believe the

14 http://www.mendeley.com/research/providing-olap-online-analytical-processing-to-useranalysts-an-it-mandate/

relational database *might* have some advantages because of its long existence and heavy performance optimizations (e.g., indexes, stored procedures, user-defined functions, managed code, file groups, raw device mapping), which for the most part do not exist in XML databases, but this impression cannot be verified and is therefore not further discussed here.

A representation of metadata within a relational database can also be independent of the DDI version or instance. Some agencies use an internal structure for their metadata that is not based on DDI but contains all the necessary information needed to exchange data with other agencies. For them, DDI in its XML form can be used as an import and export format, where the necessary files are read or created by extract, transform or load processes (so called ETL-processes). For example, ICPSR offers an "Export Study-level metadata" (of DDI 2.1 or 3.1, as of Oct. 2011) function for studies in its data archive[15] in this manner. A possible advantage of this method would be that the surrounding processes can always be adapted to the desired or required DDI version(s), which is far less challenging than updating native DDI XML instances to the appropriate version. Nevertheless, a major drawback of relational databases importing XML file structures is the possibility for information loss. If for some nodes within the XML instance there is no representation within the database structure, this content will simply be lost during the import process, or the import will not work at all if there is a structural check disallowing these kinds of partial imports.

In a DDI-RDB model all import and export processes have to be handled by ETL (Extract-Transform-Load) processes. This means DDI XML structures have to be parsed and transformed. If an unknown element comes up there are essentially two strategies to handle this – discard or store. Discard means the loss of information which can be considered bad tooling. Storing also causes a problem as this involves a high degree of program logic. A strategy could be that the original DDI XML structure is kept as backup and can be attached to a later export. However, here the danger of creating errors in the new DDI-XML structure is even higher as there might be also problems with DDI versioning when re-assembling the metadata.

Representing the DDI model in XML instances

Although the relational database contains a lot of additional features, the "native" way to represent the DDI content is to store DDI as an instance specified by the XML schema. This leads to the logical advantage of a direct representation of the content in the correct schema. A DDI instance using the full set of DDI elements will be far superior to a construct within a relational database, as not all functionalities of DDI can be represented easily in the latter. Problems arise with a relational database, as will be shown further

15 http://www.icpsr.umich.edu/icpsrweb/ICPSR/access/index.jsp

below, in representing versioning in DDI[16], or pointing to another agency by using referential URNs. In native XML the solution can be quite easily expressed, but in relational databases this is possible only with heavy additional programming (e.g., incrementing versioning by surrounding Web services or using analytical databases with slowly changing dimensions to represent the time or version). However, most agencies do not use DDI in its full specification, but only a small subset of elements; here, the advantages of the XML approach may not weigh heavily. Essentially, if an agency uses the full DDI specification, the XML implementation is superior as this is the best possibility to express DDI as designed by the DDI Alliance.

Issues with DDI specification changes in relational databases *and* in XML

A problem all implementations of DDI share is handling new versions of the specification (e.g., DDI 3.1 to DDI 3.2). If a new version of DDI is extended with new structures, or there are changes in the structure itself, this causes significant problems in implementation. In the case of the DDI-RDB, this means constructing a new import and export mechanism for the new version. Furthermore it might lead to a change in the overall database model to support both versions. In a worst case scenario, the structures are not compatible anymore, leaving the organization with two different databases or at least database partitions for storing the information, which is a considerable problem in data management.

But the DDI-XML method faces challenges with specification changes as well. Either the DDI-XML structure has to be transformed, or there have to be multiple versions of DDI-XML in the XML database. This leads also to changes in the application logic of the associated tool. If one chooses the simple solution from above and only changes the nodes which are known to the agency, again this leads to the inconsistency problems mentioned before.

A sizable advantage for the DDI-XML representation here is its hierarchical structure. DDI-XML is capable of expressing complex structures in an organized manner and can use built-in XML features like inheritance or validation against the schema. A DDI-RDB has to use additional program logic to emulate this behavior. In some cases, inheritance can only be expressed by using complex join operations between tables or self-joins within a table, leading to a decrease in speed while accessing the information. These performance issues can be decreased by using advanced database optimization techniques like partitioned view, partitioned tables or managed code, but in the end there is still a structural disadvantage.

16 see also: DDI Best Practice on Versioning and Publication – http://dx.doi.org/10.3886/DDIBestPractices08

Considering a hybrid RDB-XML database approach for DDI

As described before, the import process into a relational database from an XML structure might lead to problems, or fail altogether, if there is not sufficient mapping between certain nodes and the tabular structure. The question is what will happen to the parts which cannot be imported? Do they get discarded or are they stored externally (e.g., in a table containing strings which were not imported or keeping the original file as backup)? The disadvantage of handling this information externally would be very complex import and export handling. Furthermore, searching long strings within one table cell leads to a major loss in performance as relational tables are optimized for short cell lengths.

Another way to keep the imported XML structure intact without losing performance or logical losses would be to use the XML features of commercial databases. Some database systems, such as Microsoft SQL Server 2008 R2[17] or Oracle 11g[18], have added support for managing XML natively within the cell structure of their tables. This includes advanced features like XML indexes, XML data type (thus XML will not be handled as string, but recognized as XML) and XPath search expressions within table cells.

Using the hybrid approach, the advantages of relational databases (e.g., multiple studies, high performance) can be combined with the flexibility of XML databases and enable easier handling of DDI between different systems. A thorough evaluation of this approach, however, is outside of the scope of this paper.

Modeling DDI in Relational Databases

There are many strategies for representing DDI, based on its XML expression, in a relational database. This section provides some information about how to model various DDI elements and relationships. These ideas can be incorporated into the strategy chosen for a particular application. However, there are many other factors that should also be considered. These include requirements for performance and scalability and which DDI versions to support, as well as support within the chosen database, programming language, and application framework. All of these must be considered when designing a database schema for use within a particular application. The following examples are based on DDI version 3[19], but the techniques used should apply to future versions of DDI.

17 http://msdn.microsoft.com/en-us/library/ms189887.aspx
18 http://www.oracle.com/technetwork/database/features/xmldb/index.html
19 http://www.ddialliance.org/Specification/DDI-Lifecycle/

DDI elements

Most DDI elements consist of a number of attributes and sub-elements. For instance, the Variable element has several attributes of different types, including id (string), isGeographic (boolean), urn (URI), and more. Additionally, the Variable element contains many sub-elements. These include the VariableName, Label, Description, ConceptReference, Representation, and more.

The natural representation for these types of elements in a relational database is a table. In a Variables table, each row would represent a single DDI Variable. The columns in the Variables table would be used to store the various attributes and sub-elements of the DDI Variable element. For most simple types (such as strings, numbers, booleans, dates/times), this can easily be represented via the corresponding database field types (varchars/texts, ints, booleans, datetime). This works well for fields that can only be used once (required or optional) in a table. For example, the id and isGeographic attributes are only used once in the Variable element. For sub-elements such as Label, DDI allows 0 to unlimited Labels for each Variable. Modeling this in a relational database will require a more complex structure than a simple string field. The following sections describe how to model these complex relationships.

XML hierarchy

The most basic relationship between DDI elements comes from the XML structure. In the simplest case, there are many Schemes in DDI (such as VariableScheme or ControlConstructScheme) which are lists. Each VariableScheme can have any number of Variables, but each Variable belongs to only one VariableScheme. To model these types of relationships, the best option is usually a one-to-many relationship, where the foreign key relates the two tables. For example, to model VariableScheme and Variable, the Variables table would have a variable_scheme_id field that references the appropriate record in the VariableSchemes table. This strategy works not only for DDI Schemes, but for most elements in the DDI hierarchy. For example, a StudyUnit has many DataCollections, and a DataCollection has many CollectionEvents. These can also be modeled with one-to-many relationships.

References

A major feature of DDI is the ability to reuse elements, usually via References. For example, a Variable can reference the QuestionItem (or QuestionItems) it is based upon. Additionally, a QuestionItem can be referenced by many Variables. A many-to-many relationship is required to properly model DDI References. This is accomplished in a RDB using a join table. For each pos-

sible DDI reference, a separate join table is needed to store the reference. While a join table works fine for specific references, it does not work well for "late-bound" references, which are supported in DDI. These are discussed in the Advanced Cases section of this paper.

Recursive structures

There are some DDI elements which can have sub-elements of the same type. Some examples of this are Groups and ControlConstructs. These types of elements have a one-to-many relationship with themselves. While this can be modeled like any other one-to-many element, there are several other possibilities that may improve performance. Some common strategies for implementing trees in relational databases are Path Enumeration, Nested Sets, Nested Intervals, or solutions using Common Table Expressions. The best option depends on many factors, including the nature of the application, programming language/library support, and database support.

Substitution groups

There are many DDI elements that serve as placeholders for which several options are possible. One example is ResponseDomain. A ResponseDomain may not be directly used in a DDI Instance. Instead, it may be substituted with the appropriate element, such as a CategoryDomain, CodeDomain, DateTimeDomain, GeographicDomain, NumericDomain, or TextDomain. Other examples of substitution groups include ControlConstruct and ValueRepresentation.

DDI substitution groups can be implemented in a RDB using inheritance. The placeholder element (such as ResponseDomain) becomes the parent class in the RDB, and the other substitution elements (such as CategoryDomain, etc.) become the child classes. There are several options for implementing inheritance, involving either a single table to hold all classes, or multiple tables.

In the single-table solution, one should create a single table to hold the entire class hierarchy. There should be columns for all properties of all of the possible child classes. Many of these columns will be NULL because they do not apply to a particular record. For example, a row that is a CategoryDomain will have NULL fields for all of the columns that apply to CodeDomains, TextDomains, etc. This type of solution is inefficient with respect to space, but it eliminates joins and unions as all properties for a record are in a single row in the table.

In a multiple-table solution, a table can be created for each child class, with the appropriate columns to store the fields for that class. Additionally, each child table will have a foreign key pointing to a record in a parent-class table. For example, the ResponseDomain table will be pointed to by each of the

child tables. This solution is more space efficient, but may require several joins to retrieve records.

The best solution depends on many factors, including the needs of the application, as well as the programming language and database support.

Controlled vocabularies

There are several DDI fields whose values should come from a Controlled Vocabulary, which is managed by the DDI Alliance's Controlled Vocabularies Working Group[20]. As the vocabulary is a list of items, each Controlled Vocabulary should be represented in its own table. The various elements which need to refer to a row in this table will use a foreign key field, using a standard one-to-many relationship.

Database IDs

Each record in a table needs an ID. This can be an internal database ID (often created with auto increment), or it can be a DDI ID. For performance reasons, it is often better to use an internal database ID, using the fastest type for joins. This allows for auto increment to create unique IDs when creating new records, and leads to extremely fast joins when the ID columns are indexed. The internal database ID can be an int (or unsigned int), though many databases support UUIDs. Using UUIDs has certain advantages, as these can be used to generate completely unique DDI IDs.

The DDI ID can be stored in an additional column. As DDI IDs can contain both letters and numbers, they would have to be stored as varchar or text. This makes them less fast for joins, but still useful for searching for elements, based on DDI ID.

Advanced Cases

The following discussion provides more depth on some aspects of DDI management that were addressed earlier.

Versioning (including late-bound references)

When there are multiple versions of an element (such as a QuestionItem), another element (such as a Variable) can reference a specific version of the

20 http://www.ddialliance.org/alliance/working-groups#cvwg

QuestionItem, or it can use a "late-bound" reference. In DDI, this is done by using the letter "L" in the version (e.g., "1.L" or "3.0.0.L"), to reference the latest version of an element. DDI supports both version numbering, and also version timestamps. The timestamp aspect can be automatically managed by some databases, such as SQL Server 2008 R2 or Oracle 11i. Nevertheless there is currently no similar mechanism for representing the version numbering. Furthermore, changing the version number of an element in DDI very often leads to a cascading effect where the version number of other elements has to be increased as well.

The version numbers will therefore have to be handled programmatically. Basically there are three options to solve this:

1. A mechanism native to databases would be to create an array of different triggers on the tables to increment the version numbers of elements as well as maintain copies of the old elements in history tables (as there might be references to older versions of the elements). This leads to a lot of performance issues in running SQL INSERT or UPDATE statements. Furthermore, as triggers can be considered unmanaged code (very often they are only present in the database and not in the source code control system of the outer programming framework), they are hard to document and regularly cause problems in larger programming teams.
2. A way to solve this issue would be the option of using managed code (e.g., an external Web service programmed in C# accessing a SQL Server via the .NET Framework or similar solutions in JAVA or PHP frameworks). This does not eliminate the problem of performance issues, but at least makes the code more manageable and allows the usage of repositories.
3. The last option would be using a feature of Data Warehousing to represent the version and validity by slowly changing dimensions (see, e.g., Kimball 2002)[21]. This means the tabular structure gets additional fields which specify the start and end point of the validity of an object as well as the version number. Although this can also be done in relational databases, this technique is originally designed for the dimension tables within analytical databases and therefore might cause huge SQL statements to be represented in a fully relational structure. A solution could be to use an analytical database for the history data or ETL processes to DDI and a relational database for the current version.

Another problem of versioning is that DDI allows for unlimited length version numbers (e.g., a.b.c.d.e.f...), which would normally be implemented as a text field. String processing within table cells (even using indexing) is generally

21 Ralph Kimball, Mary Ross: The Data Warehouse Toolkit. The Complete Guide to Dimensional Modeling. 2nd Edition. John Wiley & Sons, New York: 2002.

slow within databases and should therefore be avoided. A way to limit the search burden would be to limit the depth of the version numbers so integer fields can be used. This will also allow late binding to be done via SQL WHERE statements, rather than via string processing.

As versioning is one of the key problems in using DDI in a relational database infrastructure, many of these problems have to be discussed among application developer, database designer and database administrator to find the fitting solution between functionality and performance in the software environment in question. As different databases have different features to increase performance (e.g., partitioned tables for history processing in SQL Server), the choice of the best of these three options has to be clarified.

Modeling schemes which include other schemes

There are many schemes in DDI which can include other schemes by reference. This feature can be used when creating a new version of a scheme, or even when including elements from a completely unrelated scheme.

There are two main methods for implementing this in a relational database. The first is to implement a structure very similar to the DDI XML structure. The second is to "resolve" all of the included schemes, and just store the "complete" version. The descriptions below will use VariableSchemes as an example, but the methods can be extended to other schemes.

Implementing the first method involves several tables. A VariableSchemeReferences table is used to store the references. Each VariableSchemeReference should include both the "target" VariableScheme that is being referenced, as well as the "source" VariableScheme that will include the target. This makes the VariableSchemeReferences table in effect a many-to-many join table for the VariableSchemes table to itself.

Each VariableSchemeReference can contain several Exclude elements, which should be stored in a VariableExcludes table. Each VariableExclude belongs to a VariableSchemeReference, and has a many-to-many relationship with the Variables table, which should be modeled with a join table. The VariableSchemeReference should also include an ItemMap, which points to the changed elements. This is modeled by a VariableItemMaps table, where each row contains the ID of a VariableSchemeReference, a source variable in the Variables table, a target variable in the Variables table, and a Correspondence. The correspondence can either be modeled with one or more text fields, or its own table.

While the above method of modeling the Reference closely follows DDI, it comes at a cost. There are a number of tables to manage, which increases application complexity. Additionally, trying to figure out the elements in a VariableScheme becomes a very complex and resource-intensive process, with

many queries required to combine all of the elements to get one list of variables.

An alternative method is to store only the "resolved" schemes and instead of a one-to-many relationship between VariableSchemes and Variables, use a many-to-many relationship. In this manner, each Variable will belong to all of the schemes that reference it. Using just the join table, it is very easy to list all of the Variables that belong to the VariableScheme. All important descriptive fields such as the ItemMap, Correspondence, etc., can be stored in the join table, documenting how a Variable is included in the VariableScheme. While this second method deviates more from DDI, it has many advantages for application development and performance. Most read/write operations on Variables and VariableSchemes become much faster, and the model is much simpler to understand and maintain.

Multiple language support

Most text fields in DDI have support for multiple languages. This is usually implemented by repeating the element, with a different locale for each element.

Support for multiple languages is extremely important in applications, and therefore almost every database and application framework has some support for holding localized version of strings. Because of the prevalence of support for this, the best option is usually to use the mechanism supported by the application framework. These text strings can be processed by the tools offered within the framework and stored in additional tables attached to the RDB structure. Many-to-many relationships can furthermore provide the means to store multiple translations in multiple languages for the individual item.

In DDI, one should first identify the text strings that will be translatable in the application – this will usually be Labels, Descriptions, QuestionText, and other fields – and then implement these fields as the application framework recommends to enable multiple language support. An alternative method would be using an external standard for translation like the XML Localization Interchange File Format (XLIFF). This XML schema is used by several translation software suites (e.g., Trados or OLT – Open Language Translator). A process between the relational database and the XLIFF format could work in a manner similar to exporting a DDI structure out of the database. The export creates an XLIFF file with the strings to be translated, and an external tool like Trados can be used to translate the content. Afterward the strings can be re-imported to the appropriate places in the corresponding translation tables.

Handling unknown or external elements in DDI

From the perspective of a relational database user, the biggest advantage of using the DDI-XML model is easier exchange of metadata with another agency by importing and exporting that metadata into/from the database. Nevertheless, there are also some limitations of this process.

As long as the XML schema of DDI is not violated, a DDI-XML based implementation can simply import all the code from other agencies, although the elements of the DDI schema might be unknown as the agency chose not to implement the full set of DDI. These unknown DDI elements in the XML file structure will not be processed by the tools of the importing agency, but simply ignored. However, they can still reside untouched in the XML file or XML database, ready to be added when an export takes place. When the metadata content is modified and later exported, all the information from the original import will still be available in the structure ready to be processed by the next agency which might support these elements. The XML code can therefore more or less "pass through" agencies as long as it is not modified or referenced.

Nevertheless, there is a danger of inconsistency. The previous agency might have changed elements which could have in the full set of DDI an impact on other elements. As the tools of the importing agency are only aware of the elements they know and do not modify the unknown elements, but only export them as they were imported originally, the result is an internal inconsistency between those elements, which does not cause a problem for the exporting agency, but might have huge impacts on the importing agency.

In a relational database this kind of behavior is much trickier to emulate. If the DDI XML structure from another agency is imported, the content has to be processed and divided as it has to be converted from a tree structure to a tabular structure. For the conversion there cannot be unknown elements as every single one of them has to be parsed.

Essentially there are three ways to handle the process.

1. The relational database contains the full DDI set of a specific version (e.g., DDI Lifecycle 3.1) which means at least within this setting there cannot be unknown elements.
2. The relational database discards all unknown elements in the import process; therefore they cannot be forwarded anymore, resulting in a loss of original metadata.
3. The relational database stores the unknown elements as text strings or in the case of enterprise databases as internal XML structures to provide them later for exporting. Nevertheless, the mechanism to perform this export is much more difficult to develop in a RDB environment and has the same risk of inconsistency described in the XML world.

A special case of unknown elements is when external resources embedded into the DDI structure have to be imported (e.g., multimedia resources used in educational sciences) as the import process in both cases (relational or XML) cannot really handle them because the storage structure is normally not prepared for external elements. Here the importing application has to be adapted individually to fulfill those needs.

Ensuring Application Compatibility in Transferring DDI Between Databases

Using a relational data model for storage (or a hybrid RDB-XML approach) has the disadvantage of not being able to easily store the entire DDI schema, which often warrants the dynamic generation of DDI for transmission and consumption by applications. Thus, it is important to establish the set of elements being used in a given instance. DDI application compatibility can be defined as the ability for software applications driven by DDI XML to predictably understand expected inputs and yield predictable outputs for potentially disparate instances of DDI. Due to the flexible nature of the DDI schema, a machine-actionable means of defining used and unused elements in a given instance is necessary in order to validate whether different instances of DDI are compatible. The DDI Profile module has been developed to facilitate the automation of this requirement using XPath statements[22]. Use of this module is not mandated by the DDI specification, but best practices have been established about the creation of DDI Profiles[23]. There is also a best practice document about high-level architecture for DDI application developers, which suggests the use of DDI Profiles in the context of application interoperability[24]. This paper does not aim to reiterate best practice, but to further elucidate how one might use DDI Profiles in the context of building software applications capable of communicating compatibility across organizations as well as within organizations' instances of DDI metadata.

The DDI Profile is defined in the DDI documentation as a simple collection of XPaths that describe the objects within DDI that are either used or not used for particular purposes. The DDI Profile facilitates sharing by clearly stating what is expected in the DDI metadata received or sent by an organization and defines which parts of DDI an organization or system can handle[25]. The use of a Profile is not mandatory, but when one is being used, it should be referenced in all of the DDI instances that conform to it. This is done using the

22 http://www.w3.org/TR/xpath/
23 Best Practice on Creating a DDI Profile (2009-02-15): http://dx.doi.org/10.3886/DDIBestPractices06
24 Best Practice on High-Level Architectural Model for DDI Applications (2009-02-22): http://dx.doi.org/10.3886/DDIBestPractices12
25 DDI Profile module schema: http://www.ddialliance.org/Specification/DDI-Lifecycle/3.1/XMLSchema/ddiprofile.xsd

URN of the profile in the DDIProfileReference element declared at the end of the StudyUnit module as follows[26]:

```
<?xml version="1.0" encoding="UTF-8"?>
<DDIInstance xmlns="ddi:instance:3_1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="ddi:instance:3_1 http://sampleDDI/instance.xsd"
isMaintainable="true" id="">
        <StudyUnit xmlns="ddi:studyunit:3_1" id="">
        <Citation xmlns="ddi:reusable:3_1">
        <Title></Title>
        </Citation>
        <Abstract id="">
        <Content xmlns="ddi:reusable:3_1"></Content>
        </Abstract>
        <UniverseReference xmlns="ddi:reusable:3_1"></UniverseReference>
        <Purpose id="">
        <Content xmlns="ddi:reusable:3_1"></Content>
        </Purpose>
        <DDIProfileReference><ID
xmlns="ddi:reusable:3_1">URN_of_profile</ID></DDIProfileReference>
        </StudyUnit>
</DDIInstance>
```

Adding this reference is foundational to application compatibility as it provides access to the output logic contained in the profile, informing developers interacting with the instance of which elements from the greater DDI Schema to expect as input. (The proper use of identifiers and URNs is outside the scope of this discussion, but can be found described in the DDI Identifier[27] and DDI URN Resolution[28] best practice documents.)

Whether generating DDI dynamically from a relational data model, or transmitting it from an XML database, there are several types of validation required for interoperability. Beyond XML validation and DDI Schema validation, an application serving DDI should provide facilities by which other applications can validate compatibility between systems. For example, an application developer from organization ABC interacting with a DDI store from organization XYZ via a Web service to create a cross-organization metadata search would most likely be delighted to find functionality that would analyze the profiles from their organization and communicate incompatibilities between the instances that needed to be addressed before implementation. This functionality would expedite the development process and could be built relatively easily by looping through the XPath statements contained in ABC's profile and returning the results when performed on XYZ's instance of DDI.

26 Correspondences with Sanda Ionescu, May 25, 2011 - Aug. 22, 2011
27 Management of DDI 3.0 Unique Identifiers (2009-02-15): http://dx.doi.org/10.3886/DDIBestPractices10
28 DDI 3.0 URNs and Entity Resolution (2009-03-21): http://dx.doi.org/10.3886/DDIBestPractices11

Applications between organizations are not the only architecture where DDI profiles would be very useful. Another case to consider is one in which a large organization is made up of many smaller autonomous, DDI-generating units. This organization would like to find a way to merge datasets into a model derived from common fields from each unit. This task would be extremely expensive and tedious without a way to communicate used and unused fields in a standardized way. DDI Profiles provide that method in the same way that was described in the previous example. Even greater efficiency can be found when each unit establishes common profiles with the other units before generating DDI, so that subsequent generation of metadata in one unit will be interoperable as long as it adheres to the agreed upon field set.

Use of DDI Profiles to communicate elements used and not used in an organization has many benefits besides interoperability between autonomous units' instances of DDI. It would also be pivotal in facilitating metadata tool development throughout the DDI Lifecycle[29]. In the case of many survey agencies, the creation of metadata involves a chain of tools, the output of one feeding into the next. For example, the output of a questionnaire designer would feed into a questionnaire engine, sending its output through a data cleaner, the output of which is finally ingested by tools related to dissemination.

A Look into the Future: A More Abstract Representation of DDI

While the previous sections discussed issues of handling DDI-based metadata in a relational database, XML structure, or hybrid systems, the future may hold another possibility. DDI as a standard underwent significant changes in its history, e.g., DDI Codebook (prior to 2.5) is expressed as a DTD while DDI Lifecycle (all versions of DDI 3.x) is represented as an XML schema. Some agencies, as already mentioned, use the XML format only for import and export purposes, while they internally map the metadata to their own proprietary relational database standard. The underlying principle this reveals is that DDI does not need to rely upon a particular technical representation, but is valuable as an abstract model, the manifestation of which can be in different representations, such as UML, RDF, etc. This means an abstraction layer could exist between the relations and nodes of a possible "DDI 4" and the technical representations used in a given system. The advantage is that a technical representation can be generated out of the abstract model. At the time of the writing of this paper, discussions within the DDI Alliance were occurring to

29 see graphic depiction at: http://www.ddialliance.org/sites/default/files/what-is-ddi-diagram.jpg

take exactly this step, towards conceptualizing the specification as an abstract model.

Conclusion

This paper discussed various aspects, pros, and cons of managing DDI metadata in relational databases vs. XML structures, and issues of application compatibility when transferring DDI metadata between different stores and agencies. The authors invite discussion of this paper on the DDI users' email discussion list[30], and at future meetings of the DDI and broader social science data communities.

30 http://www.ddialliance.org/community/listserv