GJXDM Query Options

(Draft)

March 12, 2004

Benjamin Shrom
Jack Wallace

© Georgia Tech Research Corporation
Atlanta, Georgia 30332-0832

Preface:
This paper was developed by an independent group of GTRI researchers not funded or directly affiliated with the Global Justice XML Data Model team. This group has gained practical experience through implementation initiatives utilizing the GJXDM. This paper is an expansion of a research paper written in August 2003. The intent of this paper is to share lessons learned and to engage the community in dialogue leading to a sound set of recommendations and best practices.

Table of Contents:

1 Introduction

The GJXDM provides a standard way for the Justice community to represent data, including relationships among various "pieces" of data. However, the use of GJXDM to represent data solves only part of the information-sharing problem. How can someone ask a question? The GJXDM is a data dictionary and data model; it does not have any inherent data query capabilities. As a result, query mechanisms must be devised that can utilize the GJXDM for the stating of queries. There are a number of issues regarding how to generate queries based on the GJXDM. These queries are intended to be passed between enterprises, agencies or even applications. How the responder processes these queries is up to the responder; queries can be processed by custom code and converted to a native query, passed through middleware, or perhaps even run directly against a database. Some queries may be processed so they can be passed to another application, for example converting an incoming query into an NLETS transaction. This paper uses the term data source to refer to the logical data repository or the application that is being asked the question.

Query mechanisms range from simple templates to full-blown query languages. Is there a one-size-fits-all solution, or are there different solutions for different cases? If the Justice community standardizes on one (or a small number) of query mechanisms, does that mean that anyone could technically query anyone else who maps their data to the GJXDM? How do different enterprises utilize the GJXDM and a query mechanism to facilitate information sharing across enterprises, or agencies?

This paper provides background on a number of standard or proposed query mechanisms that could be used by the Justice community in conjunction with GJXDM. There are many query mechanisms out there in various stages of completion. We are not trying to cover every single candidate in depth, but to evaluate the most promising. This paper also discusses issues regarding information sharing across enterprises using one or more standard query mechanisms.

2 Cross-Enterprise Information Sharing Issues

The use of GJXDM in conjunction with one (or more) "standard" query mechanism(s) simplifies data sharing across enterprises, but does not completely solve the problem. (We use the term "enterprise" here, but the same issues may arise if two applications or data sources in the same enterprise want to share information.) There are two primary technical issues that must be addressed for cross-enterprise information sharing. There are undoubtedly more technical and political issues, but only these two are addressed here.

First, the GJXDM is a very large model, with many data elements. Most, if not all, users of the GJXDM will subset the model in some fashion to make the model more manageable for their purposes. So if two enterprises don't have the same "set" of elements in their schemas, how can they query each other? There are a couple of ways to approach this. For the purposes of this paper, we use the term "client application" to mean the originator of the query. This may be a user tool forming the question and sending it out, or it may be a system forming a question to send to another system.

Option A: Each could build the query using the responder's schema, so that no question is asked that is not valid on the responder side. This requires that each enterprise have knowledge of other's schema. This also means that if many enterprises want to share information, each client application has to be able to build slightly different queries using many different schemas. This is illustrated in Figure 2-1. As a

rudimentary example, assume that Enterprise A uses a GJXDM subset that includes a person's birth date but not the person's age, and Enterprise B uses a GJXDM subset that includes age but not birth date. Depending on how much additional logic has been put into Enterprise A's client application, it can choose not to ask the question at all since it cannot ask it exactly as specified, or ask it without birth date, or it could calculate a rough birth date based on the age information it has. This may not seem like a big problem, but if there are many places where the schemas for Enterprise A and B do not agree, then there are either a lot of questions that cannot be asked, or there has to be a lot of extra logic in the code that generates the query. When you add in more than two enterprises that want to pass queries to each other, then there is even a bigger problem when the schemas do not exactly match. So if you add in Enterprise C that does not include a middle name, then Enterprise A may have to formulate one query to Enterprise B that does not include birth date but does include middle name, and a different query to Enterprise C that does not include middle name but does include birth date. All of this extra formulation comes at a price.

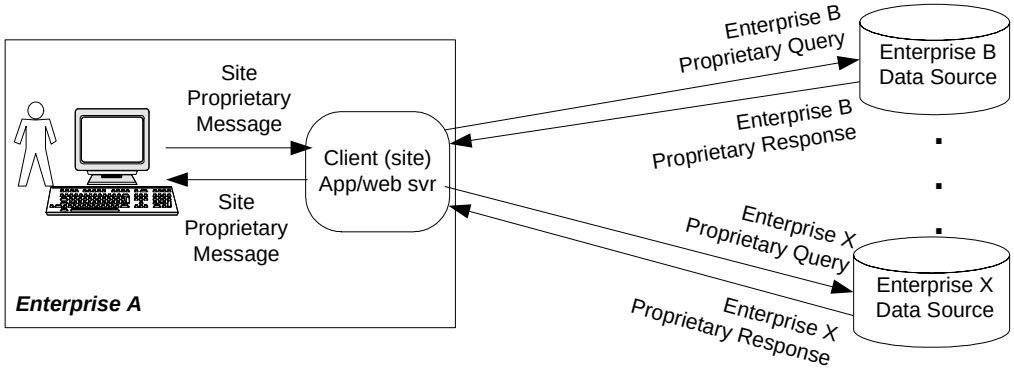


Figure 2-1 - Query/Response Flow using Responder's Schema

Option B: Each could build the query using their own (or some common, mutually agreed on) schema, which means the responding enterprise would have to do their best to respond properly. This is illustrated in Figure 2-2. In order for the responder to validate the incoming query, the originator's schema would have to be available. In addition, the responder would have to be robust enough to handle a query that includes elements that are not mapped to the responder's data. This does not mean it has to do any fancy handling or guessing of what the originator wants, but it needs to either do the best it can with the fields it understands or to respond with some usable message that basically says it could not handle the query as stated. The response could come back using the originator's schema or the responder's schema, depending on the implementation. It makes more sense to use the responder's schema, since the client application can ignore the fields that are not in it's schema, saving the responder from having to do special processing for each originator. Using the example above, Enterprise A would ask the query using both middle name and birth date; Enterprise B does what it can with the fields it understands, and Enterprise C does the best it can as well. What these enterprises do may be to ignore incoming fields and process what they can, or they may just bounce back the query with some message that says they could not process the question as asked. But ignoring fields on the back end or returning "I cannot handle what you sent" seems much easier to implement than a client application that has to look in multiple schemas to form multiple questions.

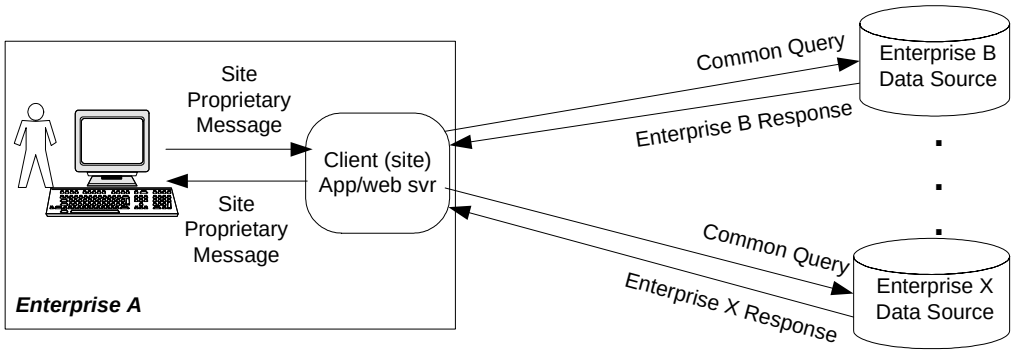


Figure 2-2 - Query/Response Flow using Originator (or Common) Schema

The second big issue is that the GJXDM provides a great deal of flexibility, which is an asset as well as a liability. The GJXDM allows objects to be "connected" in more than one way. There are 3 basic relationship mechanisms in GJXDM, each with its own strengths and weaknesses. There is a separate paper discussing these mechanisms, detailing advantages and disadvantages, so we will only cover the basics here. Assume we are trying to find a person from a particular state. This query could be formed by a user application where the user fills in a form to do this, or maybe from a system reporting tool that generates reports on the number of sex offenders in a state. One way of representing this query in GJXDM is to use the Person element and Person/Residence element. A second way to represent it is to use Person and Person/ResidenceReference, where the reference points to a Residence element in the instance. A third way is to use the element Relationship to refer to a Person element and a Residence element. If Enterprise A utilizes the Person and Person/Residence in their schema, and Enterprise B utilizes Relationship, neither will be able to properly handle data from the other without developing extra logic that handles both methods.

3 Query Mechanisms

There are two basic paradigms for stating queries: query templates and query languages. Both are valid ways of forming queries, depending on the nature of the questions being asked and in some cases the answer expected. This chapter describes each paradigm in a general way, and then summarizes decision points that can be used to determine which solution is best for a given situation. In order to simplify notation, the examples shown below use generic names instead of GJXDD element names or XPaths.

3.1 Templates

A query template, in its simplest form, is a list of fields or elements that can be queried on. It is analogous to a form that a user would fill out. Templates have minimal structure; maybe nothing more complicated that punctuation or some sort of delimiter. There is some rudimentary syntax, maybe as simple as a specific ordering or a list of field names followed by the values to be matched.

3.1.1 Usage

Templates provide a simple mechanism for stating questions. Templates are used in many data applications today, whether the underlying data model is XML or not. Templates work very well if there are a limited number of simple queries that need to be performed. What do we mean by simple? In essence, a simple query is one that can be conclusively defined, either by the use of a list or a template

where the application essentially fills in the blank. For example, let's say an application requires first name and last name in a query, and there is a fixed list of optional fields that can be included in the query, such as sex, hair color, eye color, height, weight, and age. That is still a pretty simple query, even though there are a lot of possible permutations. You could define that query as a list:

First name – required
Last name – required
Sex – optional
Hair color – optional
Eye color – optional
Height – optional
Weight – optional
Age – optional

There are any number of ways to state the syntax for a template, from using XML-like tags to an ordered list with some sort of delimiters. One possible syntax for the template might be:

First name, last name [,sex] [,hair color] [,eye color] [,height] [,weight] [,age]

Where the [ ] denote optional elements.

Templates lose some of their allure when the queries start to get more complicated. Templates may become difficult to manage if the list of possible elements starts to get really long, for example 10 possible person fields seems a lot easier to manage than 100; although the number of fields does not add greatly to the complexity of the template. Templates may not be the best solution if some elements can only be included if another element is included (if you provide age, you have to provide a +/- range), or if you want to put ranges on dates or numbers, or you want to use an OR (last name Smith or Jones), or you want a best match (last name, first name, middle name and not all fields have to be matched), or you want to do modifiers like soundex, etc. This is not to say templates cannot be used in any of these cases, but that as things start to add up, templates start getting more complex. For example, using the previously noted example, you could allow soundex on last name and a range value for the age:

First name, last name [,soundex], [,sex] [,hair color] [,eye color] [,height] [,weight] [,age] [,range]

A query using this template might look something like (note the extra comma since the sex field is not used):

Bill, smith, soundex, , brown, blue, 67, 180, 33, 3

If queries are allowed on any element in a GJXDM Person, the template becomes very large and unwieldy. However, if queries are limited to a small subset of Person, such as shown above, a template would not be so unwieldy.

Templates also become unwieldy when there are a lot of modifiers and/or a lot of mixing and matching. For example, an application allows all the fields listed above, but soundex is allowed on both or either the first and last name, age, height, and weight can have ranges, queries on partial matches such as "begins with" or "contains" are allowed, multiple colors can be provided (e.g. blue or green eyes), etc. These could still be stated as templates, but the templates look suspiciously like a language. The example below is just one of many ways to build a template for this query, this syntax happens to depend on a specific order and delimiter.

First name soundex | begins with | contains | =
,last name soundex | begins with | contains | =
,[sex]
,[hair color] [,hair color] …
,[eye color] [,eye color] …
,[height [range]]
,[weight [range]]
,[age [range]]

Using line breaks for clariry, a query stated using this template might look like:

b begins with
,smith soundex
,
,brown
,blue
,67 3
,180 10
,33 3

The expected response may also impact whether a template is practical. If a template must include not just a definition of what can be in the query, but also what can be requested in the response, templates again start looking like a language. For many queries, what you get back is what the responder says you get back. In other words, you cannot specify what elements you want back, you just get what the query processor on the back end gives you based on the kind of query you did. For example, if you have an application that takes a name in and produces a response that provides the address that corresponds to the name, the templates do not need to take into account what the requestor wants back; they get an address. If the requestor provides a name, and can specify that they want address or vehicle information back, the template starts getting more complex.

If an application allows, or will allow in the future, ad hoc queries where the requestor can build any kind of query, templates are impractical.

3.1.2 Processing

Since templates are defined based on the needs of an application or a user community, there are no industry standard templates for querying data. Standard templates could be defined for a user community, such as the Justice community, which could allow for some re-usable components. But otherwise, any development done using templates requires complete custom coding. If someone has worked with query templates in one job, that experience probably does not significantly reduce the learning curve of working with templates at another job.

Since templates are customized for an application or a user community, there is not going to be any industry support for a particular set of templates. In other words, companies like Oracle and IBM are not going to build processing capabilities into their products to support a particular set of templates. So implementers have to work from scratch. However, since templates are generally simple, the programming should not be terribly complex. If a well-defined set of templates can be established for a

certain set of transactions, such as NLETS, industry vendors might build templates into the domain specific product lines.

3.2 Language

What we think of as a language is much more formal than a template. A language has syntax and semantics, may allow much more complex content, and generally has more flexibility in what can be stated.

3.2.1 Usage

A query language provides a very powerful mechanism for stating queries, but this power comes at a price of added complexity. A full query language provides mechanisms for doing very complex queries, and even ad hoc queries, against a data source. For example, SQL is an industry standard query language for use against relational databases. As an example, if we want to do a SQL query against a person and get back address information, and we want to use of lots modifiers, a SQL query against a database might look something like:

```
SELECT P.firstname, P.lastname, A.streetname, A.city, A.state, A.zipcode
FROM PERSON_TABLE   P, ADDRESS_TABLE   A,  PERSON_ADDRESS_LINK  L
WHERE firstname LIKE 'b%'
        AND SOUNDEX(P.lastname) = SOUNDEX('smith')
        AND P.haircolor = 'brown'  AND  P.eyecolor = 'blue'
        AND P.height BETWEEN 64 AND 70
        AND P.weight BETWEEN 170 AND 190
        AND P.age BETWEEN 30 AND 36
        AND  L.person_id = P.id
        AND L.address_id = A.id;
```

This is certainly not a simple query, and could be done with a template. However, the language syntax is well defined and just about any question can be stated, and just about any response can be requested. Building a template that can do the query above results in a template that is complex enough that it really is a language.

Use of a language also provides the capability to perform ad hoc queries. For example, a requestor could state a query using any elements in the model, and depending on the power of the query processor on the back end, the query could be processed. Even if initial implementations of an application use simple queries, if the application is expected to expand to allow complex or ad hoc queries, implementers may want to use a query language from start to finish.

Combining the query language with templates may mitigate the complexity of a query language, while allowing for relatively complex queries and providing a growth path for even more powerful queries in the future. For example, if an SQL application did not want to support the full power of SQL nor allow access to all fields in the database, SQL queries could be "templated". For example, the fields in the SELECT clause of the SQL statement could be limited to a certain subset of fields, which also limits the FROM clause. In addition, the fields allowed in the WHERE clause could be limited to certain fields, and perhaps further limited to only using "=" and "soundex"; no "between" or "like" allowed. This has two potential benefits. (1) If the language used is a standard, enterprises may be able to find designers and developers familiar with the language, which may reduce the learning cycle. (2) The application could be

expanded in the future to allow more complex queries and even ad hoc queries without having to re-do as much code or documentation.

3.2.2 Processing

Depending on the language, support may be available from vendors. SQL is an industry standard (and supported) query language for relational databases, and once it became a standard, database and middleware vendors incorporated direct support into their products. At least one query language for XML data does have industry support, and therefore some vendor products could be used in implementation; reducing the amount of custom code that has to be written. Other languages are either too new, or are used by too small an audience, to be supported directly by available middleware or databases. Some languages may not be directly supported by industry, but tools and application may exist from the language proponent or the open source community. If direct support and/or tools are not available for a language, then all processing must be done manually. This may not be as onerous as it sounds. For example, if a language is stated as XML, then an XML parser can be used to assist with processing so that a language-specific parser does not have to be developed.

Even if a query language has industry support and is included as a feature in database products, implementers may not want to just pass a query instance directly to the database. Business rules must generally be applied to queries, depending on their source. For example, an enterprise would probably not want a SQL query generated by an outside enterprise or application to be run directly against a database; so just because a database supports a query language does not mean queries should be run directly against the database. So rules may need to be applied to the incoming query to modify it to include business rules, or perhaps the results from running the query must be processed. Implementers may want to insert custom logic prior to processing queries in order to ensure proper business rules are applied rather than passing the query into an engine and doing post-processing.

One other benefit to using a standard language, versus templates or a proprietary language, is that if a developer used that language in a previous job, the learning curve is reduced in a new job. For example, if someone used to use SQL on an Oracle database at one job, there would not be as great a learning curve if they needed to use Oracle against DB2 (or better yet Oracle again) at a new job.

3.3 Decision Points: Templates versus Language

The following summarizes decision points that may be useful when deciding whether to implement using templates or a full-blown language.

When to use Templates

Simple queries
Requestor cannot specify response fields
Limited number of fields that can be queried
Limited number of fields with modifiers such as soundex, begins with, date/number range, etc.
Do not need to do fuzzy queries such as ORs or best match

When to use a Language
Complex queries or ad hoc queries
Requestor can specify response fields
Large number of fields that can be queried
Large number of fields with modifiers such as soundex, begins with, date/number range, etc.
Need to do fuzzy queries such as ORs or best match

There may not be a cut-and-dried answer to whether to use one or the other; an implementation may have some characteristics that align better with templates and other characteristics that align with using a language. Some points above are stronger indicators than others. For example, a long list of fields in a template does not complicate matter nearly as much as a long list of fields with modifiers. There may also not be a "yes" or "no" answer to the questions. For example, maybe an application accepts name information as a query, and can provide either person information or address information in the response. The decision points above indicate using a language in that case, but it is a very limited specification of response fields and can certainly be handled by a template if other indicators point to using templates. In addition, an evaluator might look at these decision points and decide that what they are planning to implement this year meets all the criteria for using templates. However, if what is planned for further releases falls more into the language criteria, a query language may be the best way to go even in the first phase to reduce code and documentation changes in later phases.

4 Query Language Layers

RDF and OWL are less mature and less well-known technologies than XML and XML Schema. Therefore, this chapter provides a short overview of RDF and OWL, which form the basis for a number of the query languages discussed later.

4.1 RDF and RDF Query

Since the first release of RDF specifications in 1999 by the W3C, there have been many reports published on RDF query languages. However, none of these reports are directly applicable regarding the feasibility of using any RDF query language for querying the GJXDM. At the present moment there is no "official" RDF query language because there is no consensus about what RDF query is. Only one RDF query language has been proposed to the W3C, and that was very recently. However, several implementations and specifications are available, primarily from the academic community.

Some RDF background may be in order before we introduce specific RDF Query Languages. Figure 4-1 shows how RDF builds upon XML and XML Schema.

RDF Schema	*Semantics*
RDF	*Instances*
XML Schema	*Structure*
XML	*Syntax, Data*

Figure 4-1- RDF and XML Building Blocks

Every point on the 2-dimensional geometric plane can be described by the pair of coordinates X and Y. In a similar fashion we can use the Element Name and Element Data pair to describe our data in XML (for example, <email>chief@foo.com</email>). XML Schema defines encoding of Data Elements in XML and the description of the structure fore XML. XML Schema provides structural cardinality and data typing for XML such as element types, element names, content model, structure, and local element usage constraints. However to show the relative location using the intersection of geometric planes, we have to introduce another coordinate Z (a third dimension). The same way, in order to show the relationships between various objects/classes in XML, we have to add another value to our XML data pair: "predicate". XML then becomes a Resource Description Framework (RDF). In the GJXDM case, RDF helps to define the relationship between objects, such as a Person and a Vehicle. For example: a Vehicle has an Owner (which is a Person), or Person owns a Vehicle. RDF does not stop at high-level objects (such as Car and Person). The framework goes all the way to the deepest level of every object, describing relationship between them. For example "Person has SSID".

Each RDF statement can consists from one of the following triplets:

<subject, verb/predicate , object>

<object1, relation, object2>

<resource, property, property-value>

Where: Subject is a noun in the phrase and is the doer of the action; Object is a noun as well, and is what is being acted upon; the Predicate is the verb that provides the action. In "Person has SSID" the Subject is defined by Person, Predicate is the verb "has", and the role of Object is assigned to "SSID". It is important to point out that all RDF statements are "directional". For example in the statement "SSID belongs to Person" , a Subject role will be performed by SSID element, Object will be assigned to the Person element, and Predicate will be "belongs to".

An RDF/RDFS statement has a very specific graphical representation: a directed, labeled, possibly cyclic, graph, where nodes are the objects and each edge is a predicate/verb, describing the relationships between the objects. That leads us to the simple idea for the structure of the RDF query language. In order to state the question we need to "describe an RDF graph with parts missing, assign those parts variable names, and specify the binding between the elements". Several RDF query languages, such as RDFdb QL, Algae, RDQL, RDFql, have different syntaxes, however they employ the same idea. RDF Query languages also contain various methods for querying RDF schema information such as subclass and subproperty. [1].

A question that comes up is – an RDF instance is XML, right? Then, why don't we use an XML query language like XQuery/XPath for querying an RDF model, instead of reinventing the wheel? The answer lies in the flexibility of RDF model. The same RDF statement can be expressed by many different XML forms, which makes an XPath/XQuery question valid in one case and invalid in another. In order to be able to ask a question in XPath/XQuery, it is necessary to normalize RDF. The closest analogy would be comparison of a tree representation versus a graph. In the tree there is only one unique path to get to a specific leaf node, while in a directed cyclic graph (such as RDF) there can be several paths leading to the same end node. The normalization problem in GJXDM can be solved by subsetting to eliminate multiple paths to the leaf nodes in the model.

The RDF community has been quiet on the RDF query language standardization issue. Even after a proposal is submitted, it will take several years before any standard will be specified, and resemblance of the final result to the original submission may be superficial at best. Due to the lack of a standardization effort, many RDF query languages have emerged, and almost every language has a corresponding project implementation. A majority of the RDF query language projects have been developed by European academic community, with a few from the United States academic community; there have been few submissions by industry. Languages can be categorized by a large number of characteristics, however we will concentrate on the Query Model, Goal (variable object/predicate, literal evaluations, Boolean expressions, Node selection methods such as pattern matching, SOUNDEX), Returned Results, and Query Serialization characteristics.

The criteria for the selection of the "best of the bunch" includes:

- Capability to query GJXDM (vs. RDF)
- Extensibility of the language. (If the language does not embed node pattern matching constructs such as "soundex", "begins-with", "contains", etc.)
- Simplicity and ease of implementation
- Commercial and Open Source product availability
- Industry standard ("to be" – how long before it matures, will it ever mature)

RDF queries have two major forms of serialization: XML (RDF, non – RDF), and ASCII (N3 notation, s-expression, SQL). ASCII serialization is human friendly, however, it is preferable to have XML based syntax, since there are no reliable industry available tools or RDF query processing engines (each available engine is using proprietary RDF query language).

As it was mentioned earlier, majority of the RDF Query languages share the same Query Model, where the question is stated in the form of the Graph with missing Nodes (Subjects, Objects) or Edges (Predicates).

4.2 OWL Web Ontology Language

OWL is a W3C Recommendation dated February 2004. OWL was been developed by the Web Ontology Working Group as part of Semantic Web research. OWL is a revision of the DAML+OIL web ontology language and incorporates lessons learned from the design and application of DAML+OIL. "OWL is intended to be used when the information contained in documents needs to be processed by applications, as opposed to situations where the content only needs to be presented to humans. OWL can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms. This representation of terms and their interrelationships is called an ontology. OWL has more facilities for expressing meaning and semantics than XML, RDF, and RDF-S, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web." [17] In terms of the diagram in Figure 4-1, OWL sits on top of RDF Schema, providing higher semantics. "The Semantic Web is a vision for the future of the Web in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. The Semantic Web will build on XML's ability to define customized tagging schemes and RDF's flexible approach to representing data. The first level above RDF required for the Semantic Web is an ontology language what can formally describe the meaning of terminology used in Web documents. If machines are expected to perform useful reasoning tasks on these documents, the language must go beyond the basic semantics of RDF Schema." [17]

Since OWL is built upon RDF and RDF builds upon XML, a query language that works on OWL should be applicable to the GJXDM. However, since OWL is so new, little has been done on developing query mechanisms for OWL representations. We review one such OWL query language in this paper.

5 Query Language Review

This chapter provides information on a number of query languages that could be used by implementers utilizing the GJXDM. This chapter does not discuss every possible language that could be used to query GJXDM-formatted data. XQuery (XML Query Language) and SQL/XML (an ANSI/ISO standard) are XML-focused languages that are emerging standards with relatively broad industry support; these are covered in detail here. While the GJXDM is not RDF, it incorporates features from RDF. Therefore, query languages targeted for RDF may be appropriate for queries against the GJXDM. There are many RDF-focused query languages that have been proposed in recent years as part of Semantic Web research. Since there is not a standard RDF query language, this paper covers a number of RDF query languages that we have discovered in our research. In addition, GTRI initiated preliminary design effort in the summer of 2003 on a query language, called JXQL (for Justice XML Query Language), which was intended to meet the specific needs of the JusticeXML community; this language is also covered. This chapter provides an overview of what appears to be the strongest candidates and discusses possible methods to access and query heterogeneous data sources. Advantages and disadvantages of each candidate will be provided. Discussion of each candidate also includes a sample of commercial tools that could be used for implementation, and available information on vendor support from a short list of major database and middleware vendors; specifically IBM, Oracle and BEA.

The features required for a query language depend on the needs of the application or the user community. We have tried to determine a list of query features that appear to provide the kind of query flexibility that can be found in other query languages like SQL. There are a number of other query capabilities that are not standard in industry query products such as SQL, but that are used in query applications through

custom coding. We have attempted to determine how well each query language handles these desired query features. These features are described below.

- **Exact match on a single field** – This is the typical query capability, for example: Last name equals Smith. Exact on a single field.
- **Matches using Or** – This allows fuzzier searches. For example, searches on a person whose last name is Smith and first name is either John or James. Some query applications also allow flip-flop queries, which are generally handled by the use of OR. A flip-flop query is useful if the searcher is not sure how someone's name was entered into the system, because of different representations for some foreign names, or the use of dashes in names, or even confusion over whether something is a first or middle name. For example, someone whose name is Jane Marie Doe might be in a data source as Jane Marie Doe or Marie Jane Doe. Sometimes UNION is used in place or OR.
- **Soundex (or Sounds Like)** – This allows matching on words that sound like other words. There are algorithms that calculate a soundex value for a word, and these values can be compared instead of comparing the exact word. Not all databases use the same algorithm, so queries need to state that they want to match using soundex rather than specifying the soundex value.
- **Begins with** – This allows searches on partial words or text. For example, someone might want to search on license plate numbers that start with a set of characters.
- **Ends with** – This allows searches on partial words or text, in this case at the end of a word.
- **Contains** – This allows searches on partial words or text, in this case anywhere in the word. For example a license plate that has "123" in it somewhere.
- **Range** – Dates and numbers may have ranges. For example, someone who is 30 years old, plus or minus 3 years; or someone who weighs 200 pounds, plus or minus 10 pounds.
- **Diminutive** – Names frequently have diminutive versions. For example, someone named Bill might also be knows as Billy, or William, or Will. This feature is not part of any database products we are aware of, but is usually done as a look up in a table; i.e. look up diminutive versions of Bill and perform the search using all of them.
- **All fields must match** – Generally, searches are done where all specified fields must be matched. For example, if someone specifies last name Smith and first name John, they want all data returned to be people with the first name John and last name Smith. This is how most databases work.
- **Best match** – In a best match, not all fields must match. So for a best match search on first name John and last name Smith, the data source would return first all data where the person has the name John Smith. Then the data source could return all records where the first name is John OR the last name is Smith. None of the major database vendors support this directly. Generally multiple searches have to be performed on the various combinations. So even if a query language does not support this in its syntax, a query could be built using OR, however, the query could become very complex depending on the number of search fields. Best match usually implies a scoring or weight capability, so that matches on some things count more than matches on other. For example, a match on hair color is not as "strong" a match as one on last name. Best match without weights does not seem particularly useful.

5.1 XQuery/XQueryX

5.1.1 Overview

XQuery is a standard query language published by the W3C, with the latest working draft dated November 2003. [2] XQuery was designed to query a broad spectrum of XML information sources such as XML enabled databases and XML documents. Additional middleware enables the XQuery language to operate against relational databases. XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL, SQL, and OQL. XQuery Version 1.0 is an extension of XPath Version 2.0. The type system of XQuery is based on XML Schema. XQuery uses "FLWOR" syntax format (`for`, `let`, `where`, `order by`, and `return`), which makes the language look more like standard SQL.

The XML Query Language has more than one syntax binding. The XQuery language syntax is convenient for humans to read and write. The XQueryX syntax is expressed solely in XML in a way that reflects the underlying structure of the query. [3] Work on the XQueryX standard is somewhat behind work on XQuery, although a new working draft was released in December 2003. The remainder of this paper focuses on XQuery.

In June 2003, IBM and Oracle submitted JSR 225 (Java Community Process, http://www.jcp.org) to "Develop a common API that allows an application to submit queries conforming to the W3C XQuery 1.0 specification and to process the results of such queries". [4] JSR implementation will bring XQuery to the new level of officially implemented standard (like JDBC).

Issues with XQuery include:

- XQuery by itself requires the constructor of a query to know the structure of the data source. To overcome this problem, additional abstraction layers of "data views" can be introduced (see XQuery engine implementations from IBM and BEA). Optionally, an implementer can parse the XQuery and convert it directly to query against a specific data source. So for the Justice community, queries would be constructed using GJXDM instead of an XML representation of the underlying database.
- The introduction of additional abstraction layers decreases the performance of the system. Currently there is no data available to estimate the performance impact of using XQuery.
- Most implementations of the XQuery engine are not mature and do not represent a complete solution to the problems listed above.
- The current version of XQuery does not support the use of weights, which are necessary to directly handle queries such as best match, where not all search terms have to be matched to get data back. There is a proposal for a "SCORE" (weighting) capability to be added to a future version of XQuery.
- The current version of XQuery does not support soundex or diminutive queries although there are possible workarounds. There are proposals for handling queries using dictionaries and thesauri (which would handle soundex and diminutive) to be added to a future version of the XQuery.
- The language is relatively complex since it was devised as a processing language instead of just a way to state a query. Therefore, there are multiple ways to state the same question.

5.1.2 Example

The example listed below shows access to the data in a relational database using XQuery. “A relational database system might present a view in which each table (relation) takes the form of an XML document. One way to represent a database table as an XML document is to allow the document element to represent the table itself, and each row (tuple) inside the table to be represented by a nested element. Inside the tuple-elements, each column is in turn represented by a nested element. Columns that allow null values are represented by optional elements, and a missing element denotes a null value.” [5]

Suppose there is online auction relational database that contains three tables:

```
USERS ( USERID, NAME, RATING )
ITEMS ( ITEMNO, DESCRIPTION, OFFERED_BY, START_DATE, END_DATE, RESERVE_PRICE )
BIDS  ( USERID, ITEMNO, BID, BID_DATE )
```

Tables can be converted to the XML view by using 1-to-1 default table-to-xml mapping provided by most database manufacturers (for example XSU utility from Oracle):

```
<items>
  <item_tuple>
    <itemno>1001</itemno>
    <description>Red Bicycle</description>
    <offered_by>U01</offered_by>
    <start_date>1999-01-05</start_date>
    <end_date>1999-01-20</end_date>
    <reserve_price>40</reserve_price>
  </item_tuple>
  <!-- !!! Snip !!! -->

<users>
  <user_tuple>
    <userid>U01</userid>
    <name>Tom Jones</name>
    <rating>B</rating>
  </user_tuple>
  <!-- !!! Snip !!! -->

<bids>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1001</itemno>
    <bid>35</bid>
    <bid_date>1999-01-07</bid_date>
    </bid_tuple>
  <bid_tuple>
  <!-- !!! Snip !!! -->
```

Suppose someone needs to list the item number, description, and highest bid (if any), for all bicycles, ordered by item number. The resulting XQuery would look like:

```
<result>
  {
    for $i in doc("items.xml")//item_tuple
    let $b := doc("bids.xml")//bid_tuple[itemno = $i/itemno]
    where contains($i/description, "Bicycle")
    order by $i/itemno
    return
        <item_tuple>
            { $i/itemno }
            { $i/description }
            <high_bid>{ max($b/bid) }</high_bid>
        </item_tuple>
  }
</result>
```

The XML result would look like:

```
<result>
  <item_tuple>
    <itemno>1001</itemno>
    <description>Red Bicycle</description>
    <high_bid>55.0</high_bid>
  </item_tuple>
</result>
```

For more examples see http://www.w3.org/TR/xquery-use-cases/ website.

5.1.3 Available Tools and Support

5.1.3.1 ORACLE OJXQI

OJXQI is a Java API for XQuery proposed by Oracle that can be used to execute and fetch results from XQuery queries against single XML documents or an Oracle database.

Oracle has introduced several enhancements to the standard XQuery, which can be considered drawbacks since it then becomes a proprietary Oracle XQuery feature:

- Support for SQL queries to be embedded inside XQuery.
- Support for bind variables, so as to not re-execute the XQuery for every constant change.
- Support for XQueryX - an XML representation of the XQuery language - for ease of mechanical generation and translation of XQuery.

The product depends on the following: XDK 9.2.0.1.0, XSU 9.2.0.1.0, JDK 1.3, and JDBC1.3 (if used with Oracle).

It is unclear from the prototype whether a user can run a query against an XQuery defined view and how it will be processed.

Oracle does not provide any information about upcoming releases.

5.1.3.2 IBM XML FOR TABLES

“XML for Tables provides functions for creating XML views of relational tables in such a way that the SQL data are treated as if they are virtual XML documents and they can be queried in XQuery.

XML for Tables does this by automatically mapping the data of the underlying relation database system to a low-level default XML view. User can then create application-specific XML view on top of the

default XML view. These application specific view are created using XQuery. Another significant feature provided by XML for Tables is the ability to query User defined XML views of relational databases using XQuery.

XML for Tables translates XQuery into SQL and pushes down SQL to DB2. SQL queries produce output in tuple format; XML for Tables tags the tuple result into XML; so the XQuery results are in an XML document.

XML for Tables is wrapped as DB2 stored procedures, and queries are submitted by calling the stored procedures. ” [6]

There are two major disadvantages in XML for Tables

- DB2 specific (may be possible to overcome by using the Information Integration DB2 suite)
- Released under limited license. It is currently a beta product and expires 90 days from download. It is not a commercial product yet and there is no technical support.

The example listed below [7] shows use of XQuery with XML for Tables:

a) Purchase Order Database and it’s default XML view

order

id	custname	custnum
10	Foo Construction	7734
9	Western Builders	7725

item

oid	desc	cost
10	generator	8000
10	Pump	24000

payment

oid	due	cost
10	1/10/01	20000
10	6/10/01	12000

```
<db>
    <order>
        <row> <id>10 </id> <custname> Foo Construction </custname> <custnum> 7734 </custnum> </row>
        <row> <id> 9 </id> <custname> Western Builders </custname> <custnum> 7725 </custnum> </row>
    </order>
    <item>
        <row> <oid> 10 </oid> <desc> generator </desc> <cost> 8000 </cost> </row>
        <row> <oid> 10 </oid> <desc> backhoe </desc> <cost> 24000 </cost> </row>
    </item>
    <payment>
        …
    </payment>
</db>
```

b) User defined XML view and resulting XML

```
create view orders as (
    for $order in view(“default”)/order/row
    return
        <order>
            <customer> $order/custname </customer>
            <items>
                for $item in view(“default”)/item/row
                where $order/id = $item/oid
                return
                <item>
                    <description> $item/desc </description> <cost> $item/cost </cost>
                </item>
                    </items>
            <payments>
                for $payment in view(“default”)/item/row
                where $order/id = $payment/oid
                return
                <payment due=$payment/date>
                    <amount> $payment/amount </amount>
                </payment> sortby(@due)
            </payments>
        </order>
)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
        <customer> Foo Construction </customer>
        <items>
                <item>
                        <description> generator </description>
                        <cost> 8000 </cost>
                </item>
                <item>
                        <description> Pump </description>
                        <cost> 24000 </cost>
                </item>
        </items>
        <payments>
                <payment due='1-10-01'> <amount> 20000 </amount> </payment>
                <payment due='6/10/01'> <amount> 12000 </amount> </payment>
        </payments>
</order>
```

c) Query over user defined XML view.

```
for $order in view(“orders”)

let $items = $oder/items

where $order/customer like “Foo%”

return $items
```

XML for Tables allows queries of the GJXDD model using XQuery and can be implemented for broad use if XTABLES is removed from IBM:Alphaworks development state to production and support for other information sources is enabled and/or if use of the Information Integration IBM software suite proves feasible. Since XTables pushes much processing down to the database engine, this solution has the potential for fast performance. This also makes it more difficult to provide for other databases, at least with the same performance advantages, since versions for other databases would require extensive customization to push the processing down into another vendor’s database engine.

5.1.3.3 IBM XML EXTENDER/NET.DATA

DB2's XML Extender provides new data types to store XML documents in DB2 databases along with functions that assist in working with these structured documents. Retrieval functions (based on DB2 stored procedures) allow retrieval of either the entire XML document or individual elements or attributes.

XML Extender uses a document access definition (DAD) to map XML elements and attributes to DB2 tables. The DAD can be used for indexing elements and attributes for fast structural search, or for creating XML documents from DB2 data, or shredding and storing XML data in DB2.

XML Extender supports sending and retrieving XML documents from MQSeries message queues. The software also supports Web Services with Web Services Object Runtime Framework (WORF) Beta.

The software is included in DB2 Universal Database, Version 7.1.

Also, an XML document can be generated from SQL queries against DB2 or any ODBC compliant databases using Net.Data. Net.Data is a full-featured scripting language, which allows access to the following information sources: DB2, Oracle, DRDA-enabled data sources, ODBC data sources, flat files, and web registry data. Net.Data is a no-cost feature of most versions of DB2.

This approach could be used in multiple GJXDD solutions: adaptation of XTABLES for other relational information sources, or in JXQL query engine implementation.

System drawbacks are:

- Unknown performance of the system
- XML Extender implementation requires use of XTABLES (see below), which is still in the alpha development stage (not commercially released and not supported product).
- Solution could lock implementers into IBM's proprietary solution.

5.1.3.4 BEA LIQUID DATA FOR WEBLOGIC

BEA offers a single product solution for access and querying of heterogeneous data sources – "BEA Liquid Data for WebLogic is a data access and aggregation product for *Information Visibility*, allowing a real-time unified view of disparate enterprise data." [8]

BEA Liquid Data for WebLogic provides the following capabilities:

- Universal data access – Using XML translators and optimized XML queries, Liquid Data can retrieve and query data from relational databases, Web Services, Web sites, flat files, XML files, and other data sources, returning results in XML format.
- Abstract Data Views – Liquid Data provides a virtual abstraction layer to aggregate distributed data sources as an integrated, logical view. For developers, these logical views of aggregated datasets can be thought of as a single, virtual database.
- Expose and Share Web Services – Liquid Data allows deployment of Data Views as a Web Services
- Liquid Data provides developer with GUI tools for rapid application development.
- BEA Liquid Data for WebLogic is based on the BEA WebLogic Server.
- Liquid Data implements XQuery standard as a basic coding language, providing a platform for building and processing XQuery queries.

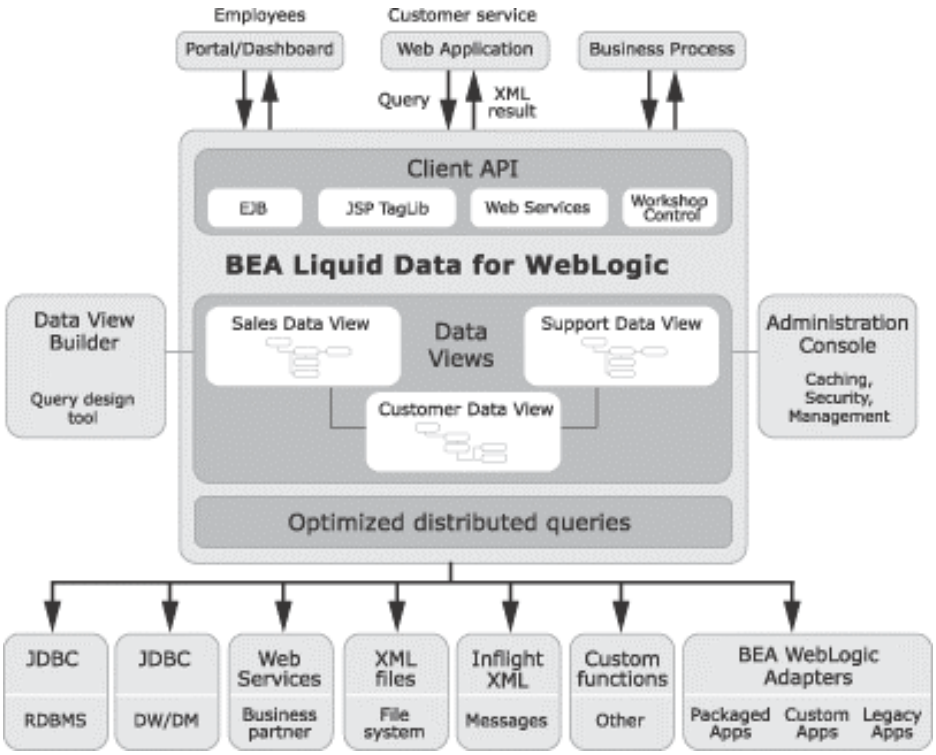


Figure 3

BEA Liquid Data represents a possible implementation solution for XQuery and GJXDD. BEA Liquid Data has several appealing features that IBM's XTABLES is missing:

- Fully released and supported product. BEA has already released a second version of Liquid Data.
- Heterogeneous data source integration.

The following disadvantages exist in the Liquid Data solution:

- License fees
- Unknown performance of the system – the degree of optimization for specific databases is questionable
- Locks implementers into a solution available only from one software vendor
- Requires use of the BEA Weblogic Application Server

For more information, see the documentation page at:
http://dev2dev.bea.com/products/liquiddata81/index.jsp
Or the product page at:
http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/liquid_data/

5.1.4 Conclusion

XQuery represents a generic XML query language, which is expected to be widely adopted. However due to the recent release of the standard, not many vendors support XQuery to the full extent. XQuery can be used for querying databases with the help of middleware tools such as IBM's XTABLES and BEA's

Liquid Data. These tools are in early stages and in general have not matured yet. There are no estimates for the performance impact of using these middleware tools. XQuery can also be used without middleware tools and converted directly into native queries, although this requires potentially significant custom coding. Regardless of the implementation method, XQuery instances will have to be built assuming the GJXDD as the underlying data source to ensure isolation of the query from the data source implementation. Workarounds will be required to handle soundex and/or diminutive queries. The language is relatively complex and may be difficult to convert directly into native queries.

5.2 SQL/XML

5.2.1 Overview

SQL/XML is a standard that emerged from ANSI and ISO SQL. [9] SQL/XML is an extension to SQL -- using functions and operators -- to include processing of XML data in relational stores. SQL/XML is the preferred way to query and manipulate XML when data is a mix of structured and semi-structured, and there is a need to use enterprise (SQL) tools with existing standards (SQL and SQL/XML). The definition of SQL/XML is driven by the SQLX Group, which is sponsored by IBM, Oracle (XDK for 9i), Microsoft (SQLXML 3.0), and Sybase.

"SQL/XML defines a mapping from tables to XML documents. The mapping may take as its source an individual table, all of the tables in a schema, or all of the tables in a catalog. The mapping takes place on behalf of a specific user, so only those tables that contain a column for which the user has SELECT privilege will be included in this mapping. This mapping produces two XML documents, one that contains the data in the table or tables that were specified, and another that contains an XML Schema that describes the first document." [10]

Major drawbacks of the standard are following:

- SQL/XML query requires detailed knowledge of the information source.
- SQL/XML query is written to run against relational structure and is not capable of querying a generic GJXDD model (see example below). Nevertheless, SQL/XML can be effectively used in the backend for implementation of other query engines as it is shown in Figure 5-1.
- SQL/XML is not extensible – current implementations do not allow introduction of additional layers of abstraction for mapping between GJXDD and various database instances.
- SQL/XML query engine implementations do not exist for older database releases.

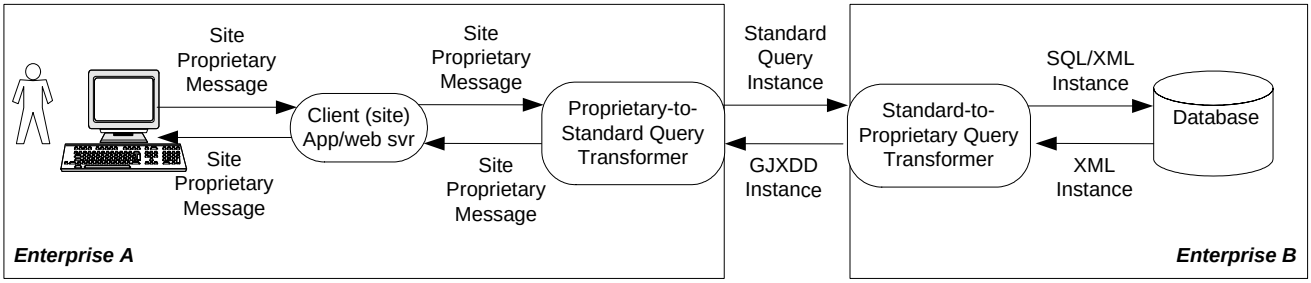


Figure 5-1 - Query/Response Flow Using SQL/XML on the Backend

Even though SQL/XML is designed to run against a relational model, it is technically possible to use SQL/XML as the generic query. A standard relational model would have to be developed that all

implementers would generate queries against, similar to how XQuery utilizes GJXDD as the model to query against. However, this relational model would have to be based upon GJXDD so that the resulting data could be expressed as a GJXDD instance. Switching between relational in one direction and GJXDD in the other could potentially make the mapping more difficult since the underlying database would have to be mapped to both models. It seems much more practical to use SQL/XML on the backend.

5.2.2 Example

The Oracle XDK package takes an arbitrary SQL query and converts the results to XML. Here is a hierarchical XML output example (taken from the Oracle Code Examples webpage), which shows the master-detail output for the "`emp`" and "`dept`" tables in Oracle's sample database. This creates a department element with a list of employees.

```
SELECT  XMLElement("Department",
     XMLForest (deptno "DeptNo", d.dname "DeptName", d.loc "Location"),
           (SELECT XMLAGG(XMLElement("Employee",
                  XMLForest ( e.empno "EmployeeId",
                       e.ename "Name",
                        e.job "Job",
                        e.mgr "Manager",
                        e.hiredate "Hiredate"),
                        e.citezen "Citezen"))
                 FROM emp e
            WHERE e.deptno = d.deptno))
FROM dept d;
```

The resulting XML would look like:

```
   <department>
          <deptno>23</deptno>
          <deptname>Accounting</deptname>
          <location>Campus</location>
          <employee>
                 <employeeid>4568</employeeid >
                 <name>George Burdell</name>
                 <job>TA</job>
                 <manager>John Wandelt</manager>
                 <citizen>US</citizen>
          </employee>
   </department>
```

5.2.3 Available Tools and Support

5.2.3.1 ORACLE XML SQL UTILITY (XSU)

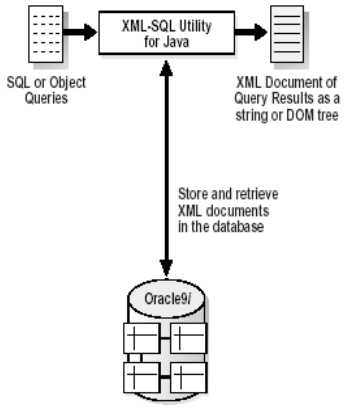
XSU is comprised of core Java class libraries for automatically and dynamically rendering the results of arbitrary SQL queries into canonical XML. XSU includes the following features:

- Supports queries over richly-structured user-defined object types and object views.
- Supports automatic XML Insert of canonically-structured XML into any existing table, view, object table, or object view.

XSU Java classes can be used for the following tasks:

- Generate from an SQL query or result set object a text or XML document, a Document Object Model (DOM), Document Type Definition (DTD), or XML Schema.
- Load data from an XML document into an existing database schema or view.

Figure 5-2 shows how XSU processes SQL queries and returns the results as an XML document.



The structure of the resulting XML document is based on the internal structure of the database schema that returns the query results:

- Columns are mapped to top level elements
- Scalar values are mapped to elements with text-only content
- Object types are mapped to elements with attributes appearing as sub-elements
- Collections are mapped to lists of elements

Figure 5-2 - Oracle XML SQL Utility (XSU)

XSU is compatible with older versions of Oracle (above 8.0.6).

XSU can be successfully used for implementation of query engines as a database SQL to XML adapter for different Oracle information data sources.

5.2.3.2 ORACLE DATABASE SUPPORT FOR SQL/XML

The Oracle9i Database implements a number of standard-based functions enabling direct query of relational data, returning XML documents using the emerging ANSI/ISA SQL/XML standard. Oracle versions prior to 9i are not supported.

5.2.4 Conclusion

SQL/XML enables XML support for relational databases, creating a bridge between XML and relational data. Implementers can create XML views of existing relational data and work with them as if they were XML files – a feature which can be effectively used in the backend for implementation of other query languages. However, SQL/XML cannot be used by itself as the primary query solution since it requires extensive knowledge of the underlying database and there are no options for creating an abstraction later to hide the underlying database. Development of a relational model of the GJXDD in order to use SQL/XML as a generic query language does not seem practical. In addition, SQL/XML is only supported in the latest releases of major databases and is not compatible with older database versions that may be in use in the Justice community.

5.3 Justice XML Query Language (JXQL)

5.3.1 Overview

The Justice XML Query Language (JXQL) is a language devised by the Georgia Tech Research Institute based upon an early version of the GJXDM. JXQL was designed to provide a mechanism to query disparate data sources that may exist in the Justice community, regardless of the underlying database or structure of the data source. A query application can use JXQL to build a query without any knowledge of the data source since JXQL references GJXDD types and properties instead of data source fields.

Knowledge of the underlying data source is pushed down to the responding application, where the mapping between the GJXDD and the data source is handled. JXQL is defined as an XML schema, and can therefore be validated by a standard XML validator. Other XML communities could use JXQL, replacing the GJXDD component with their own data dictionary, as long as those communities adhered to the concepts defined in the GJXDM. (I.e. the GJXDM defines a set of relationships between types and properties; each type "has a" set of properties that define its nature, and each property exists because it is a property of some type or types.)

The JXQL schema incorporates three schemas: criteria, weights and results. The criteria schema defines a data set and fulfills two needs. It can be used to specify a query, containing criteria that the data must meet to be in the set; such as people whose last names equal "Smith". It can also be used to represent data that cannot be represented using the GJXDD alone; such as a last name that starts with 'S'. For the purposes of this document, the descriptions and examples are limited to queries. The weights schema allows the data set to be sorted based upon weights defined by the query originator. This also allows a "best match" type of query where the results may include data that only matches some criteria instead of all criteria. The results schema allows specification of the data to be returned, specified as types or properties from the GJXDD.

The primary drawback to JXQL is that it was designed as part of a research effort, and as such has not been visible outside the Justice XML community. In fact, many inside the Justice XML community may not be aware of it. Even though JXQL could be used outside the Justice XML community, it is unlikely that JXQL would gain sufficient momentum to become an industry-recognized standard like XQuery. Therefore, JXQL would have to be considered a proprietary product. Any tools required for usage would have to be developed from scratch. It should also note that the design was preliminary, and some details were never completed.

The primary advantage to JXQL is that it was designed to handle the specific problem of defining a query against disparate data sources that can be mapped to the GJXDM. Therefore it is a less complex language than XQuery, which was designed to be a processing language against XML documents.

5.3.2 Example

The following example defines a relatively complex query looking for people using a search on last name "Lee", first name "Jung" whose residence is on a street "Main". The JXQL instance also requires that matches return the person's last name, first name, middle name, date of birth, social security number, height, hair color code and residence address data street name, city and state. This example does not include the weights schema. It should be noted that this example was developed based on the older JXDD 3.0.0.1 pre-release.

```
<q:query
    xmlns:q="http://justicexml.gtri.gatech.edu/query/1/query"
    xmlns:c="http://justicexml.gtri.gatech.edu/query/1/criteria"
    xmlns:a="http://justicexml.gtri.gatech.edu/query/1/assembly"
    xmlns:justice="http://www.it.ojp.gov/jxdd/prerelease/3.0.0.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://justicexml.gtri.gatech.edu/query/1/query
        query.xsd">
  <c:instance>
    <c:variable name="A"/>
    <c:has-type type="justice:PersonType"/>
```

```
    <c:has-property property="PersonName">
      <c:has-property property="justice:PersonSurName">
        <c:value-matches-caseless match="lee"/>
      </c:has-property>
      <c:has-property property="justice:PersonGivenName">
        <c:value-matches-caseless match="jung"/>
      </c:has-property>
    </c:has-property>
    <c:has-property property="justice:Residence">
      <c:has-property property="justice:LocationAddress">
        <c:has-property property="justice:AddressStreet">
          <c:value-matches-caseless-pattern match=".*main.*"/>
        </c:has-property>
      </c:has-property>
    </c:has-property>
  </c:instance>
  <a:result>
    <a:for-each select="A">
      <a:include-selected property="justice:PersonName">
        <a:include-selected property="justice:PersonSurName"/>
        <a:include-selected property="justice:PersonGivenName"/>
        <a:include-selected property="justice:PersonMiddleName"/>
      </a:include-selected>
      <a:include-selected property="justice:PersonBirthDate"/>
      <a:include-selected property="justice:PersonSocialSecurityNumber"/>
      <a:include-selected property="justice:PersonPhysicalDetails">
        <a:include-selected property="justice:PersonHeightMeasure"/>
        <a:include-selected property="justice:PersonHairColorCode"/>
      </a:include-selected>
      <a:include-selected property="justice:Residence">
        <a:include-selected property="justice:LocationAddress">
          <a:include-selected property="justice:AddressStreet"/>
          <a:include-selected property="justice:AddressCityName"/>
          <a:include-selected property="justice:AddressStateName"/>
        </a:include-selected>
      </a:include-selected>
    </a:for-each>
  </a:result>
</q:query>
```

5.3.3 Available Tools and Support

This is a proprietary language, so there are no tools specific to JXQL. However, since JXQL is XML, XML parsers and validators can be used against instances.

5.3.4 Conclusion

JXQL represents an XML query language that is targeted at the Justice community with their requirements to search disparate data sources. JXQL was designed to eliminate the need for the generator of the query to have any knowledge of the underlying data source, so that a query application could query many data sources without having to customize the query for each source. The drawback to JXQL is that it may not be utilized outside the Justice XML community, requiring those utilizing JXQL to develop all tools required to implement complete solutions.

5.4 OWL-QL (Formerly DQL – DAML Query Language).

Website: http://ksl.stanford.edu/projects/owl-ql/
Company: Stanford Knowledge Systems Laboratory
License: n/a.

5.4.1 Overview

OWL-QL is being developed by the SKSF and the members of Joint United States/European Union ad hoc Agent Markup Language Committee. The query language has emerged from the DAML Query Language and probably will be a winning candidate for "standard language and protocol for query-answering dialogues among Semantic Web computational agents". While OWL-QL is designed for query of the Semantic Web and to be used with OWL, the language is extremely flexible and is capable of querying RDF/RDFS model as well. The current OWL-QL specification is using XML syntax format for query expression. However, the specification provides an outline for structural format of the language only, reserving the space for many different syntactic forms.

Details on this language can be found in the paper by Richard Fikes, Pat Hayes, Ian Horrocks: "OWL-QL - A Language for Deductive Query Answering on the Semantic Web", which can be found at: ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-03-14.pdf.gz

5.4.2 Examples

Q: "Find all people who own red car"

```
Q: (owns ?p ?c) (type ?c Car) (has-color ?c Red)
   must-bind ?p don't-bind ?c
A: (exists ?c (and (owns Joe ?c) (type ?c Car) (has-color ?c Red)))
```

```
<owl-ql:query xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
        xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
  <owl-ql:queryPattern>
    <rdf:RDF>
      <rdf:Description rdf:about="http://www.w3.org/2003/10/owl-ql-variables#p">
        <owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
      </rdf:Description>
      <Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
        <has-color rdf:resource="#Red"/>
      </Car>
    </rdf:RDF>
  </owl-ql:queryPattern>
  <owl-ql:mustBindVars>
    <var:p/>
  </owl-ql:mustBindVars>
  <owl-ql:answerKBPattern>
    <owl-ql:kbRef rdf:resource="http://joedata/joe.owl"/>
  </owl-ql:answerKBPattern>
  <owl-ql:answerSizeBound>5</owl-ql:answerSizeBound>
</owl-ql:query>

<owl-ql:answerBundle xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
                xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
  <owl-ql:queryPattern>
    <rdf:RDF>
      <rdf:Description rdf:about="http://www.w3.org/2003/10/owl-ql-variables#p">
        <owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
      </rdf:Description>
      <Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
```

```
      <has-color rdf:resource="#Red"/>
    </Car>
   </rdf:RDF>
  </owl-ql:queryPattern>
  <owl-ql:answer>
   <owl-ql:binding-set>
    <var:p rdf:resource="#Joe"/>
   </owl-ql:binding-set>
   <owl-ql:answerPatternInstance>
    <rdf:RDF>
     <rdf:Description rdf:about="#Joe">
      <owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
     </rdf:Description>
     <Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
      <has-color rdf:resource="#Red"/>
     </Car>
    </rdf:RDF>
  </owl-ql:answer>
  <owl-ql:continuation>
   <owl-ql:none/>
  </owl-ql:continuation>
</owl-ql:answerBundle>
```

5.4.3 Available Tools and Support

Tools and support are meager at best. The OWL-QL author provides an XML Schema definition for the language syntax. He also presents numerous example queries, which can be run on the java based web application (.war file) prototype or via Java API supplied in the form of library archive. However, the author does not release source code for the application prototype.

5.4.4 Conclusion

At the present, there is no official submission to the W3C for OWL-QL language standardization; however it seems likely that OWL-QL will become a standard language for Semantic Web Query. OWL-QL has a long road ahead and would require additional customization for querying GJXDM model.

5.5 QEL (Query Exchange Language)

Company:	SUN Microsystems
Project:	JXTA/Edutella
Website:	http://edutella.jxta.org
License:	The Sun Project JXTA Software License

5.5.1 Overview

QEL is being developed by SUN Microsystems for collaboration on JXTA Peer-To-Peer (P2P) networks. The language is part of the Edutella P2P project. “QEL is used to express queries against data sources using the Resource Description Framework (RDF).” QEL is based on relational calculus and has a predicate expression as a basic language construct. Predicate expressions consist of a predicate symbol, followed by an argument list (for example *owner(“Red Car”, “John Doe”)*). QEL uses RDF syntax for database queries, which are expressed in predicate logic and are very similar to classical Prolog expressions. Currently, the language provides a number of build-in predicates to be used for RDF query, such as “qel:s” – used to match RDF triples, “qel:nodeType” – used to test RDF container membership, “qel:equals” – used to determine if two RDF nodes are the same, “qel:stringValue” – used to test the string value of an RDF literal, ignoring any language or datatype. The language explicitly specifies the

form or the result set. “Results in QEL are returned in a tabular format, one row at a time. This makes it possible to return results in independent batches.”

The QEL specification does not enforce implementation of all listed constructs and provides a list of actions to be taken by the system if the query is not supported. The specification also provides the vocabulary for description of the query capabilities implemented by the model. For example QEL implementations will be capable of reporting whether it can support “like”, “equals”, “greaterThan”, or “member” build in queries. However, it is not clear whether the language can be extended with the new predicates, such as “soundex”, “begins-with”, etc.

5.5.2 Examples:

```
<!--
name(P,N) :-
qel:s (P, <jxdm:name>, N)
qel:s(C, <jxdm:color>, "red")
qel:s(C, <jxdm:owner>, P),
-->
<rdf:RDF>

 <qel:Rule rdf:nodeID="rule1">
  <qel:head>
   <qel:QueryLiteral rdf:nodeID="nameliteral">
    <qel:predicate rdf:nodeID="name"/>
        <qel:arguments>
      <rdf:Seq>
       <rdf:li rdf:nodeID="P"/>
       <rdf:li rdf:nodeID="N"/>
      </rdf:Seq>
     </qel:arguments>
   </qel:QueryLiteral>
  </qel:head>

  <!-- Reuse literal from above -->
  <qel:literal rdf:nodeID="rs1">

  <qel:outerJoinLiteral>
   <qel:StatementLiteral>
     <rdf:subject   rdf:nodeID="X"/>
     <rdf:predicate rdf:resource="&jxdm;name"/>
     <rdf:object    rdf:nodeID="N"/>
   </qel:StatementLiteral>
  </qel:outerJoinLiteral>
   <qel:StatementLiteral>
     <rdf:subject   rdf:nodeID="C"/>
     <rdf:predicate rdf:resource="&jxdm;owner"/>
     <rdf:object    rdf:nodeID="P"/>
    </qel:StatementLiteral>
  </qel:outerJoinLiteral>
   <qel:StatementLiteral>
     <rdf:subject   rdf:nodeID="C"/>
     <rdf:predicate rdf:resource="&jxdm;color"/>
     <rdf:object    rdf:nodeID="red"/>
   </qel:StatementLiteral>
  </qel:outerJoinLiteral>
 </qel:Rule>

</rdf:RDF>
```

5.5.3 Available Tools and Support

Sun Microsystems has started the Edutella project as "RDF-based Metadata Infrastructure for P2P Applications". Project has released several application prototypes, however they are based solely on RDF data sources, and the currently do not support connection to a back end database.

5.5.4 Conclusion

The QEL language provides a solid framework and functionality for RDF model queries, and can be used for GJXDM queries as well. At the present moment the language specification is not complete. There is no implementation that would provide the translation of the QEL queries to native database queries based on the defined mapping between RDF model and existing RDB schema. Since it has been designed for collaboration on P2P network, it is not clear whether QEL will become a standard RDF query language.

5.6 SQL-like RDF query languages (RDQL, SquishQL, rdfDB, RQL, SeRQL).

Language:	rdfDB
Project:	rdfDB
Website:	http://guha.com/rdfdb/
License:	Mozilla Public License 1.0 (MPL)

Language:	SquishQL
Company:	Hewlett Packard
Project:	Inkling
Website:	http://swordfish.rdfweb.org/rdfquery/
License:	GPL License

Language:	RDQL (RDF Data Query Language), SquishQL
Company:	Hewlett Packard
Project:	Jena
Website:	http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/ http://www.hpl.hp.com/semweb/doc/tutorial/RDQL/index.html
License:	BSD License

Language:	RQL (RDF Query Language)
Company:	ICS-FORTH - Greece
Website:	http://139.91.183.30:9090/RDF/RQL/

Language:	SeRQL (Sesame RDF Query Language)
Project:	Sesame (http://www.openrdf.org/)
Website:	http://www.openrdf.org
Company:	Aduna
License:	GNU Library or Lesser General Public License (LGPL)

5.6.1 Overview

In November 1998, the W3C posted a position paper on Query Languages and specified a query framework for RDF – subgraph matching. [18] As result of this, several RDF query languages have emerged. The language developed first was rdfDB – a simple SQL-like query language, designed by Ramanathan V. Guha for the rdfDB open-source database. rdfDB served as a basis for development of a

more advanced RDF query language called SquishQL by the Information Infrastructure Laboratory at Hewlett Packard in April 2002. [20] SquishQL has incorporated a majority of the features available in rdfDB and added new options such as an optional 'FROM' clause, a new notation for shortened namespaces, and patterns and constraints clauses for result filtering. SquishQL syntax was implemented in Inkling RDF query engine with PostgreSQL as backend support. Later, SquishQL was integrated with Jena Semantic Web applications framework and received a new name – RDQL. RDQL served as a source for member language submission to W3C in January 2004 by Hewlett Packard.

RDQL has data-oriented model queries and does not support inference – RDQL system does not infer new RDF statements based on the conclusion from other RDF statements. "RDQL provides a way of specifying a graph pattern that is matched against the graph to yield a set of matches. It returns a list of bindings - each binding is a set of name-value pairs for the values of the variables. All variables are bound (there is no disjunction in the query)." [19] Being extremely generic, the W3C submission does not specify mechanisms for extending the language.

SeRQL RDF query language introduced the next step in the evolution of the RDQL language by adding support for subClassOf and subPropertyOf, allowing simple inferences. SeRQL is being developed by Aduna as part of the Sesame project. "It combines the best features of other (query) languages (RQL, RDQL, N-Triples, N3) and adds some of its own. The important features are:

- Graph transformation.
- RDF Schema support.
- XML Schema datatype support.
- Expressive path expression syntax.
- Optional path matching."

Following the similar path of introducing inferences, another language is being developed by the Institute of Computer Science FORTH academic community – RQL. "RQL is a typed language following a functional approach (a la ODMG-OQL). RQL supports generalized path expressions (GPE) featuring variables on both labels for nodes (i.e., classes) and edges (i.e., properties). RQL relies on a formal graph model (as opposed to other triple-based RDF QLs) that captures the RDF modeling primitives and permits the interpretation of superimposed resource descriptions by means of one or more schemas.

The novelty of RQL lies in its ability to smoothly combine schema and data querying while exploiting the taxonomies of labels and multiple classification of resources, using advanced pattern-matching facilities (i.e. GPEs). The RQL Interpreter has been implemented in C++ on top of an object oriented database using standard client-server architecture for Solaris and Linux platforms. It consists of four modules (a) the Parser, analyzing the syntax of queries; (b) the Graph Constructor, capturing the semantics of queries in terms of typing and interdependencies of involved expressions; (c) the SQL Translator, which rewrites RQL to efficient SQL queries; and (d) the Evaluation Engine, accessing the underlying database via SQL queries."

5.6.2 Examples

Q: "Find all people who own red car"

RDQL:

```
SELECT ?person, ?name
WHERE    (?person, <jxdm:FullName>, ?name),
```

```
        (?car, <jxdm:Owner>, ?person),
        (?car, <jxdm:Color>, “red”)
      USING  jxdm  FOR http://justicexml.gtri.gatech.edu/3.0#
```

SeRQL:

```
      SELECT DISTINCT
         Person, Name
      FROM
         {Person} <jxdd:name > {} {Name} ,
         {Car} <jxdd:color> {} <Color> ,
         {Car} <jxdd:owner> {Person}
      WHERE
         isLiteral(Color) AND
         lang(Color) = "en" AND
         label(Color) LIKE "*Red*"
      USING NAMESPACE
         jxdd = <!http://justicexml.gtri.gatech.edu/3.0/>
```

5.6.3 Available Tools and Support

See list of projects in the beginning of the section. SeRQL appears to be the only language for which a commercial product is available, and there is only one product.

5.6.4 Conclusion

From the evolution of SQL-like RDF query languages, it is clear that inference enabled languages (SeRQL, RQL) eventually will take over from their predecessors (RDQL, SquishQL). The current submission to W3C does not provide any guarantee that RDQL language will become an RDF query language standard due to lack of important features. The RQL language represents a prototype that has emerged from the academic community, while SeRQL is already integrated in the corporate environment and is used in the commercial applications.

5.7 Turtle - Terse RDF Triple Language

Company:	Institute for Learning and Research Technology University of Bristol
Website:	http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/
Project:	Redland RDF Application Framework (http://www.redland.opensource.ac.uk/)
License:	LGPL, Mozilla Public License V1.1

5.7.1 Overview

Formerly: N-Triples Plus. Turtle is an extension of N-Triples. Project architect Dave Beckett participates in the development of Jena Semantic Web Framework and plans implementation of RDQL in the Redland framework.

5.7.2 Example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix jxdm: <http://example.org/jxdm/3.0/>.
<http://www.w3.org/TR/rdf-syntax-grammar>
  jxdm:Car [
    jxdm :Color "Red";
    jxdm :Owner [
        jxdm :PersonName [
```

```
                jxdm :PersonFullName "John Doe";
        ]
    ]
  ].
```

5.7.3 Available Tools and Support

The Turtle RDF query language is implemented on the Redland RDF Application Framework which is written in C. Author claims that the project was successfully compiled and run on all major OSes. Turtle has interfaces for all major programming languages such as Perl, Python, Tcl, Java, Ruby, and PHP.

5.7.4 Conclusion

The Turtle language appears to represent mostly academic interest and there is no evidence that language will become the RDF/Semantic Web query standard.

5.8 D2R (Database 2 RDF Map)

Website: http://www.wiwiss.fu-berlin.de/suhl/bizer/d2rmap/D2Rmap.htm
Company: Freie Universität Berlin
License: GNU LGPL

5.8.1 Overview

"D2R MAP is a declarative language to describe mappings between relational database schemata and OWL ontologies. The mappings can be used by a D2R processor to export data from a relational database into RDF." The project specification claims that the D2R processor prototype is capable of exporting data as RDF, N3, N-TRIPLES and Jena models. Also, the prototype is compliant with all relational databases offering JDBC or ODBC access and is implemented in Java, Jena API.

5.8.2 Example

```
<d2r:Map d2r:versionInfo="$Id: eShopDB.Map.d2r, v 1.0 2003/01/20 19:44:09 Chris Exp $">
  <!-- <d2r:ProcessorMessage d2r:outputFormat="N3"/> -->
  <!-- <d2r:ProcessorMessage d2r:outputFormat="RDF/XML-ABBREV"/> -->
  <d2r:DBConnection d2r:odbcDSN="eShopDB"/>
  <d2r:Namespace d2r:prefix="eShop" d2r:namespace="http://www.wiwiss.fu-berlin.de/suhl/bizer/eShop/eShop#"/>
  <d2r:Namespace d2r:prefix="rdf" d2r:namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <!-- Categories -->
  <d2r:ClassMap d2r:type="eShop:category" d2r:sql="SELECT catid, catname, description  FROM categories" d2r:groupBy="catid">
    <d2r:DatatypePropertyBridge d2r:property="eShop:name" d2r:column="catname"/>
    <d2r:DatatypePropertyBridge d2r:property="eShop:description" d2r:pattern="Description of Category '@@catname@@':
@@description@@."/>
  </d2r:ClassMap>
  <!-- CDs -->
    <d2r:ClassMap d2r:type="eShop:cd" d2r:sql="SELECT article.articleno, interpret, album, coverlink, category, trackno, price FROM
article, tracks where article.articleno = tracks.articleno and offer=True" d2r:groupBy="article.articleno"
d2r:uriPattern="eShop:CdNo@@article.articleno@@">
    <d2r:DatatypePropertyBridge d2r:property="eShop:interpret" d2r:column="article.interpret"/>
    <d2r:DatatypePropertyBridge d2r:property="eShop:album" d2r:column="article.album"/>
    <d2r:DatatypePropertyBridge d2r:property="eShop:price" d2r:pattern="$ @@price@@"/>
    <d2r:ObjectPropertyBridge d2r:property="eShop:cover" d2r:pattern="http://www.wiwiss.fu-
berlin.de/suhl/bizer/eShop/images/@@article.coverlink@@"/>
    <d2r:ObjectPropertyBridge d2r:property="eShop:category" d2r:referredClass="eShop:category" d2r:referredGroupBy="category"/>
    <d2r:ObjectPropertyBridge d2r:property="eShop:track" d2r:referredClass="eShop:track" d2r:referredGroupBy="articleno,trackno"/>
  </d2r:ClassMap>
  <!-- Tracks -->
```

```
  <d2r:ClassMap d2r:type="eShop:track" d2r:sql="SELECT  articleno, trackno, name  FROM tracks" d2r:groupBy="articleno, trackno"
d2r:uriPattern="eShop:track@@articleno@@-@@trackno@@">
      <d2r:DatatypePropertyBridge d2r:property="eShop:name" d2r:column="name"/>
   </d2r:ClassMap>
</d2r:Map>
```

5.8.3 Available Tools and Support

D2R has a working prototype available at the project web site.

5.8.4 Conclusion

The language is essentially an RDF version of SQLXML – "Send SQL query to RDBMS, Receive RDF back". This represents the major problem for the GJXDM domain – this language is tightly coupled to the backend database. The queries are database specific and should be formulated against particular database implementation, not against GJXDM or any other desired abstract model. Therefore, language does not appear to be useful in conjunction with GJXDM.

5.9 Versa

Versa

Website: http://uche.ogbuji.net/tech/rdf/versa/
http://uche.ogbuji.net/tech/rdf/versa/versa.doc?xslt=/ftss/data/docbook_html1.xslt
http://4suite.org/index.xhtml

Project: 4Suite

Company: N/A

License: N/A

5.9.1 Overview

Versa is an RDF query language based on the graph data model and developed by Uche Ogbuji. Versa query language is modeled after XPath and follows functional paradigm (LISP look alike). Queries in Versa are traversal expressions and allow forward and backward processing of arcs, node content filtering, and general expression evaluation. Versa includes support for aggregate functions, sorting functions, and has build in data conversion functions.

5.9.2 Examples

head(h:FullName(type(h:Car) - dc:Color -> eq("Red") - h:Person -> *))

5.9.3 Available Tools and Support

Currently there is no prototype available for the Versa language.

5.9.4 Conclusion

This is interesting as an academic language. It does have more match capabilities than some of the other RDF query languages, although exactly what it has is unclear since it is still under development.

5.10 Other Languages

These languages have not yet been reviewed. None have been submitted to the W3C. Versa, like SeRQL, seems to have more match capabilities than other RDF query languages.

Buchingae, LogicML (Another Rule-Markup Language)
Website: http://mknows.etri.re.kr/bossam/docs/logicml.html
Project: Bossam

KAON
Website: http://kaon.semanticweb.org/
http://wim.fzi.de:8080/kaon/readme.html
Project: KAON
Company: FZI, AIFB
License: LGPL

Algae2
Website: http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html
Project:
Company:
License:

6 Comparison of Query Languages

This chapter summaries the advantages and disadvantages of the reviewed query languages. The table below shows how languages compare on query features. Note that there are two columns for XQuery: one for the current working draft (1.0) and one that includes proposed features for the next version (which we have labeled as 1.x). All of the RDF query languages, except SeRQL, have the same query features, and are combined into one column.

Match Capability	JXQL	XQuery 1.0	XQuery 1.x	OWL-QL	SeRQL	Other RDF Languages
Exact on a single field	Yes	Yes	Yes	Yes	Yes	Yes
OR	Yes	Yes	Yes	Yes	Yes	Yes
Soundex (Text)	Yes	No	Yes	No	No	No
Begins with (text)	Yes	Yes	Yes	No	Yes	No
Ends with (text)	Yes	Yes	Yes	No	Yes	No
Contains (text)	Yes	Yes	Yes	No	Yes	No
Range (dates/numbers)	Yes	Yes	Yes	No	Yes	No
Diminutive (names)	Yes	No	Yes	No	No	No
All fields must match	Yes	Yes	Yes	Yes	Yes	Yes
Best match - weights	Yes	No	Yes	No	No	No

Figure 6-1 – Supported Search Features

SQL/XML is not a reasonable solution for the Justice community due to the requirement that queries be written against a relational data model, although it may be useful on the backend.

XQuery is a W3C working draft that already have industry support, in the form of tools, middleware and database support. This is the most mature of the query languages. XQueryX may be easier to use than XQuery, since the XML syntax allows the use of existing XML tools and bindings, although there is not currently much industry support for this syntax. XQuery is very powerful, but also very complex. The complexity could be mitigated through the use of templates that limit what features can be stated in a query. XQuery's missing query features, such as weights and soundex, are expected to be addressed in later versions of the specification. In the interim, there are workarounds for soundex and diminutive.

JXQL was designed from scratch with GJXDD in mind. It is simpler than XQuery, but still supports all query features. However, the drawback to JXQL is that it may not be utilized outside the Justice XML community, requiring those utilizing JXQL to develop all tools required to implement complete solutions.

Most of the remaining query languages are very generic. Other than SeRQL, none provide very many match capabilities, such as are found in query languages like SQL. For example, you cannot state a question where you want to match on a partial word (i.e. find a person whose first name starts with 'B'). RDQL is the only one that has been submitted to the W3C, and that submission was in January, so even if it eventually becomes a standard, it will be years. In fact, SeRQL is based on RDQL, so RDQL does not even appear to be the most advanced RDF query language. RQL and SeRQL are the only RDF or OWL query languages that actually have products that can be downloaded and used for real projects. However, these are proprietary products and are not supported by any major vendors.

7 Conclusions

It appears that templates and a query language have different roles in the information-sharing realm. Both fit certain needs, and we believe the Justice community would be well served by supporting both.

In terms of templates, the development of a standard template syntax is desirable; something that handles simple queries and does not become a full-blown language definition. It would be beneficial to base these templates on XML-syntax, unlike the rudimentary examples earlier in this paper.

We cannot make any definitive recommendations at this time on what the Justice XML community should use as a standard query language. Out of the reviewed query languages, only XQuery (and its XQueryX sibling) can be considered an industry standard – and they are only in the late working draft stage and cannot be considered mature. However, no other query language will reach XQuery's level of maturity and industry support for years. RDQL has been proposed to the W3C, but is not even in the working draft stage. XQuery is very complex, and if it is selected as the GJXDM-standard query language, templates should be supported for simpler needs. The RDF and OWL query languages are new and no standard has emerged in that domain. So if the JusticeXML community selects an RDF or OWL language for use now, that will essentially tie the GJXDM to a proprietary solution since even if the Justice XML community picks the language that ends up being selected as the standard, the language is certain to change before the standard is completed. Since it may be years before an RDF or OWL standard query language emerges, picking any of the RDF or OWL candidates now seems to be a very high risk position. SeRQL has the potential for immediate use, since there are products available and development is being supported by the open source community and a commercial company. But SeRQL has not been proposed to the W3C as a standard, so using it would still seem to tie the Justice XML community into a proprietary solution.

The Justice XML community could also develop their own query language, either starting from the preliminary work on JXQL, or by starting with one of the other reviewed query languages. This effort will take time, and will require a great deal of collaboration to ensure that the language is powerful enough to handle any queries that are expected to be required. If a simple enough language could be devised, templates may not need to be supported. However, we would also have to guard against "feature-creep" to avoid developing a language that even more complex than XQuery. While development of a Justice-specific query language appears desirable on the surface, evaluators need to keep in mind the effort required, and the fact that development of this language would revisit a lot of the same ground already covered by the designers of the reviewed query languages. So we may end up reinventing the wheel.

XQuery or XQueryX appear to be the best solution for implementers who are going to need something in the next few years. Resolution for shortcomings in the match capabilities are planned for later releases, and workarounds are available for soundex and diminutive. The complexity of XQuery can be reduced by limiting the features to be used in GJXDM queries, although some effort will have to be expended to determine how best to limit queries. Further review of XQuery versus XQueryX may also be practical since the latest update to XQueryX is so new. While industry support is broader for XQuery, the use of XML syntax in XQueryX may still make it the more usable language.

In order to facilitate information sharing across enterprises, the Justice XML community may want to consider selecting a single method for the representation of relationships. Even with a standard query

language or templates, if a query is stated using one relationship mechanism, a responder that uses a different relationship mechanism may not be able to respond.

8 References

1. Databases, Query, API, Interfaces report on Query languages (2003-04-01). http://www.w3.org/2001/sw/Europe/reports/rdf_ql_comparison_report/

2. W3C: XQuery 1.0 – An XML Query Language http://www.w3.org/TR/xquery/

3. W3C: XML Syntax for XQuery 1.0 (XQueryX) http://www.w3.org/TR/xqueryx

4. JSR 225: XQuery API for Java™ (XQJ); http://www.jcp.org/en/jsr/detail?id=225

5. W3C: XML Query Use Cases http://www.w3.org/TR/xquery-use-cases/

6. IBM: XML for Tables http://www.alphaworks.ibm.com/tech/xtable

7. IBM: XTABLES: Bridging relational technology and XML
   http://www.research.ibm.com/journal/abstracts/sj/414/fan.html
   http://www.research.ibm.com/journal/sj/414/funderburk.pdf

8. BEA: BEA Liquid Data for WebLogic™ http://edocs.bea.com/liquiddata/docs81/index.html

9. XSQL Group: SQL/XML http://sqlx.org

10. Oracle: Oracle Technology Network (OTN) http://www.otn.oracle.com

11. "Querying XML documents" Miller, J.A. Sheth, S. Department of Computer Science, University of Georgia, Athens, GA; Potentials, IEEE, Feb/Mar 2000; On page(s): 24-26; Volume: 19, Issue: 1 ; ISSN: 0278-6648; CODEN: IEPTDF; INSPEC Accession Number: 6540107

12. "Bidirectional conversion between XML documents and relational databases" Jacinto, M.H.; Librelotto, G.R.; Ramalho, J.C.; Henriques, P.R.; Computer Supported Cooperative Work in Design, 2002. The 7th International Conference on CSWD Design, 25-27 Sept. 2002 Page(s): 437 -443.

13. Some XQuery Implementations:
    GALAX http://www-db.research.bell-labs.com/galax/
    XQEngine http://www.fatdog.com/
    Qexo http://www.gnu.org/software/qexo/
    Ipedo http://www.ipedo.com/

14. Mapping Semantic Web Data with RDBMSes (2003-01-23). http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/

15. "The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management." Michael C. Daconta, Leo J. Obrst and Kevin T. Smith: (Wiley 2003).

16. Libby Miller: RDF Query and Rules: A Framework and Survey (2001- http://www.w3.org/2001/11/13-RDF-Query-Rules/

17. “OWL Web Ontology Language Overview”, Deborah L. McGuinness, Frank van Harmelen; Feb 10, 2004. http://www.w3.org/TR/owl-features/

18. R.V. Guha, Ora Lassila, Eric Miller, Dan Brickley, Enabling Inference, W3C Query Language Meeting, Boston, Dec 3-4, 1998 http://www.w3.org/TandS/QL/QL98/pp/enabling.html

19. Jena Semantic Web Framework http://jena.sourceforge.net/index.html