Service-Oriented Architecture

A Field Guide to Integrating XML and Web Services

Thomas Erl

PRENTICE HALL
PROFESSIONAL TECHNICAL REFERENCE
UPPER SADDLE RIVER, NJ 07458
WWW.PHPTR.COM

Contents

CHAPTER

7

Integrating XML and databases

When talking about XML to DBAs and data analysts in the past, my enthusiasm frequently was greeted with a level of suspicion that often made me feel like I was trying to sell undercoating on a used car. XML's acceptance in the overall IT mainstream has since increased dramatically, because it has become a relatively common part of database environments. Still, there is a significant lack of understanding as to how or why XML can or should integrate with traditional corporate repositories.

One of the obstacles to both conceptualizing and realizing the integration of XML formatted data with traditional databases is simply the fact that XML was not designed with databases in mind. XML's origins lie in document meta tagging, and the XML language was developed to infuse structure and meaning into the vast amount of presentation-oriented content on the Internet.

Now that it has evolved into a core application development technology, it is being used for a variety of sophisticated data representation and transportation purposes. It has found a home in just about every layer of application architecture, except… the relational database tier. Here it fits less comfortably (Figure 7.1).

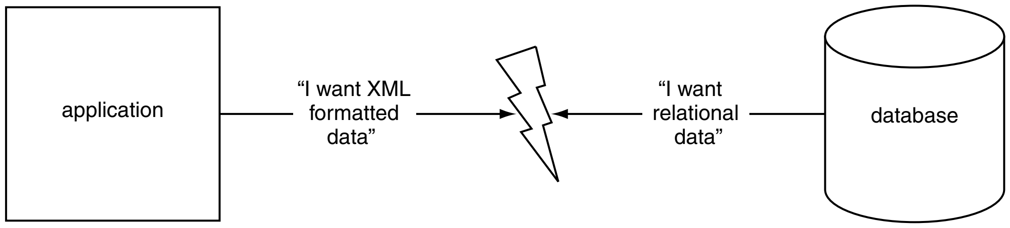


Figure 7.1
Application components and databases have different data format preferences.

XML documents and relational databases represent and structure data in very different ways. This draws an invisible border between the two environments, and getting these data platforms to cooperate efficiently can be as challenging as negotiating a treaty between two very different cultures. And guess what — you're the arbitrator.

7.1 Comparing XML and relational databases

Before creating any sort of data integration strategy, it is important to first understand the fundamental differences between relational databases and the XML technology set, NOTE

Though important to designing a robust service-oriented architecture, integrating XML and databases does not require or depend on the presence of Web services. As a result, there is little reference to Web services in this chapter, allowing you to apply the architectures and strategies to environments outside of SOAs.

and their respective relationships to your corporate data. This section covers some of the major areas of data management, and contrasts how they are addressed by each platform.

NOTE

We are not making this comparison to provide a choice between the two platforms. We are only assessing the features of each to gain an understanding of their differences.

7.1.1 Data storage and security

The most basic feature of any data management system is its ability to securely store information. This is where relational databases provide an unparalleled set of features, barely comparable to XML's simple file format (see Table 7.1).

Table 7.1 Data storage and security comparison

	Databases	XML
Physical storage	Highly controlled storage environment.	Plain text.
Security	Proprietary security, or a security system integrated with the operating system. Provides granular control over most aspects of the data and its structure.	No built-in security. Access control is set on a file or folder level, or is managed by the application.

7.1.2 Data representation

XML documents and relational databases approach the representation of data from different ends of the spectrum. XML documents introduce a cohesive, structured hierarchy, whereas RDBMSs provide a more flexible relational model (see Table 7.2).

These two platforms face significant integration challenges because XML document hiearchies are difficult to recreate within relational databases, and relational data models are difficult to represent within XML documents.

Table 7.2 Data representation comparison

	Databases	XML
Data model	Relational data model, consisting of tabular data entities (tables), with rows and columns.	Hierarchical data model, composed of document structures with element and attribute nodes.
Data types	A wide variety of data types typically are provided, including support for binary data.	XSD schemas are equipped with a comparable set of data types.
Data element relationships	Column definitions can interrelate within and between tables, according to DDL rules.	References can be explicitly or intrinsically defined between elements.

Understanding these simple limitations is the most important part of creating an effective integration strategy.

7.1.3 Data integrity and validation

In Table 7.3, we look at how data management systems provided by databases and XML preserve the integrity of the data they represent.

7.1.4 Data querying and indexing

Next, in Table 7.4, is a comparison of generic searching and indexing features.

7.1.5 Additional features

Rounding up this high-level comparison is Table 7.5, providing a list of features found in both database and XML technologies, most of which are exclusive to their respective environment.

SUMMARY OF KEY POINTS

- XML and relational databases are fundamentally incompatible data platforms, created for different purposes.
- The requirement to integrate these two types of technologies stems from XML's popularity in application environments.
- An understanding of how XML documents differ from relational databases is required in order to devise effective integration strategies.

Table 7.3 Data validation comparison

	Databases	XML
Schema	The loose structure of relational schemas provide a great deal of flexibility as to how data entities can exist and interrelate.	XML schemas are more rigid in that they are restricted to an element hierarchy. More sophisticated schema technologies, such as the XML Schema language, provide functionality that can achieve detachment of elements.
Referential integrity	Extensive support is provided to ensure that relationship constraints are enforced. Typical features include the ability to propagate changes in related columns via cascading deletions and updates.	Although references simulate inter-element relationships to an extent, no comparable enforcement of RDBMS-like referential integrity is provided. Additionally, while relational databases enforce referential integrity at the time data is altered, a separate validation step is required by the XML parser.
Supplemental validation	Validation can be further supplemented through the use of triggers and stored procedures.	XSD schemas can be designed to validate elements and attributes according to custom rules. Additional technologies, such as XSLT and Schematron, can be used to further refine the level of validation.

7.2 Integration architectures for XML and relational databases

As you look through each of the upcoming architecture diagrams, it is worth remembering that there is no one standard approach. The many different integration requirements organizations tend to have in this part of the application architecture demand a flexible set of integration models that will vary in design and scope.

If you are already using XML within your application, or if you already have an application design, follow this short process to best assess the suitability of an alternative architecture:

1. Describe the role XML currently plays within your application. Make sure you have a clear understanding as to how and why XML is being utilized.

Table 7.4 Data querying and indexing comparison

	Databases	XML
Query languages	Most commercial RDBMSs support the industry standard SQL. Many add proprietary extensions.	Single XML documents are most commonly queried via the DOM and SAX APIs, or by using XPath expressions. The XML technology most comparable to SQL is XQuery, which provides a comprehensive syntax that also supports cross-document searches.
Query result manipulation	SQL provides a number of output parameters that can group, sort, and further customize the query results.	XSLT and XQuery can be used to manipulate the output of XML formatted data. Both languages can group, sort, and perform complex data manipulation.
Querying across multiple repositories	Several database platforms allow multi-data source queries, as long as each repository supports the protocol used to issue the query.	XPath cannot query multiple XML documents, however XQuery can. XSLT can also query multiple documents (using the document function).
Indexing	Sophisticated indices are supported, customizable to the column level. RDBMS indices can be fine-tuned for optimized querying.	The XML technology set does not support a comparable indexing extension. XML document indices are often created and maintained by custom applications.

2. Pick a primary business task and map the processing steps between the application components. Show how and where XML data is being manipulated and transported.
3. When you reach the database layer, identify how the XML-formatted data currently is being derived, where inserts and updates are required, and how often the same body of data is reused.
4. Review each of the integration architectures in this section, and identify the one closest to your current or planned design.
5. Study the pros, cons, and suitability guidelines to ensure that your current architecture provides the best possible integration design for your application requirements. If it doesn't, consider one of the alternatives.
6. Finally, if no one architecture adequately meets your requirements, pick the one that is the closest, and modify it to whatever extent you need to.

Table 7.5 Comparison of various additional features

	Databases	XML
Transactions	Most databases provide transaction and rollback support, and most support the common ACID[3] properties. Some RDBMSs also come with two-phase commit capabilities, extending transaction support across multiple databases.	The XML platform does not yet provide an industry standard transaction technology (although support for ACID properties is provided through the use of the WS-Coordination and WS-Transaction second-generation Web services specifications).
Multi-user access, record locking	To preserve the integrity of data during concurrent usage, most databases provide a means of controlling access to data while it is being updated. RDBMSs typically support either page-level or row-level record locking, as well as different locking models (e.g., pessimistic, optimistic).	Access to XML documents existing as physical files is essentially file I/O controlled by the application via the XML parser. There are no comparable locking features.
Platform dependence	Relational databases are commercial products that impose a vendor-dependent storage platform. However, data generally can be easily migrated between databases from different vendors.	The family of XML specifications are open industry standards and are not dependent on any commercial platform.
Schema dependence	Schemas are a required part of a database. The schema features are provided by the database software.	Schemas are an optional part of XML documents. If used, one of several available schema technologies can be chosen.
Schema reuse	The schema is bound to the database instance. Schemas can be encapsulated with DDL, but not easily reused without the help of modeling tools.	The schema exists as an independent entity. Multiple documents can use the same schema.
Nesting	Table columns generally do not provide intrinsic nesting.	XML elements can contain nested child elements.

3. ACID represents the following four standard transaction properties: atomicity, consistency, isolation, and durability.

7.2.1 Storing XML documents as database records

This approach simply places entire XML documents into dedicated database tables, putting the responsibility of validating and processing these documents on the application (Figure 7.1). The new tables are separated from the current data model, and therefore will likely not affect the existing relational data.

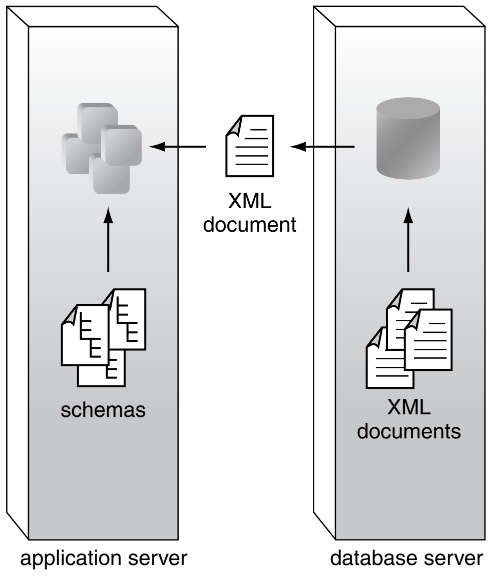


Figure 7.1
A physical architecture illustrating XML documents being stored and retrieved from a relational database.

Corresponding schemas can be stored (and cached) with the application, and linked dynamically upon retrieval of the document. Alternatively, schemas can be placed in the database with the XML documents, where they can be retrieved together, as illustrated in Figure 7.2.

To accommodate this architecture, schemas and XML documents can be placed in separate tables, united by a simple one-to-many relationship. If a single XML document can link to multiple schemas, then this will require a many-to-many relationship.

If XML document and schema constructs (modules) are supported, then additional tables would need to be added in order to support one-to-many relationships between the tables hosting the master XML documents and schemas, and their respective constructs.

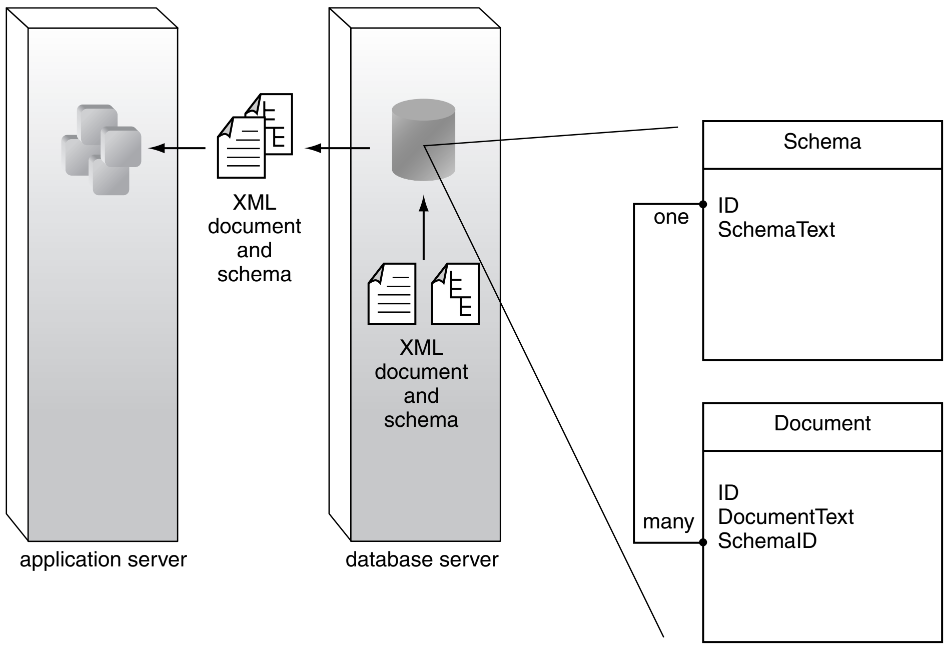


Figure 7.2
A variation of this architecture, where schemas are stored alongside XML documents, within the database.

Also, it is advisable for the primary and foreign key values used by the database to be represented within the document and schema content (perhaps as embedded annotations). This will allow you to efficiently reference the source record of an XML document or schema being processed by the application.

Suitable for:

- Redundant storage of legacy data, preformatted in XML. This would require some method of synchronization, depending on how static the data remains.
- Storing new application data that does not relate to existing legacy data. If there's any chance that this information will need to tie into your existing legacy data, then this may not be such a great idea. However, in situations where you simply want to store complete XML documents that are fully independent of the existing legacy repository, this provides a simple way of adding XML support without having to worry about mapping.
- Storing state information. Session data that needs to be temporarily (or even permanently) persisted fits perfectly into this model. The data may require only a simple database table to store the document within a single column.

Pros:

- XML data is mobile and easily detached. Alternatively, should integration with the existing data model be required, it can still be performed by simply removing the new tables.
- Allows for an easier migration to a native XML database, as these database systems tend to store XML documents in their entirety as well.
- Existing data models will likely not be affected.

Cons:

- Queries against these new tables may be limited to full-text searches, which are notoriously slow.
- Even when using full-text searches, many databases are not XML-aware (cannot distinguish tags from data). This translates into poor search results that require further processing to be useful.
- This design will introduce a level of data modeling disparity.
- If the option to store schemas with XML documents is chosen, more runtime processing will be required to retrieve a single document. (This can be mitigated by introducing an application caching strategy that retrieves the schema periodically.)

7.2.2 Storing XML document constructs as database records

Similar to the previous architecture, this design approach (illustrated in Figure 7.3) also introduces a loosely coupled model that does not affect the existing legacy data. The difference is that here an XML document is divided into logical constructs (decomposed into smaller XML chunks), each of which is stored independently.

To accommodate this architecture, schemas can be divided into multiple granular schemas. These schema modules can also be assembled dynamically into a composite schema instance that matches the structure of the generated XML document (as shown in Figure 7.4). For more information about modular schema design, see the "Building modular and extensible XSD schemas" section in Chapter 5.

Suitable for:

- Object-based or class-based application interfaces.
- New application data requiring flexible document structures that are determined at runtime. This type of requirement typically is encountered when supporting parameter-driven business rules.

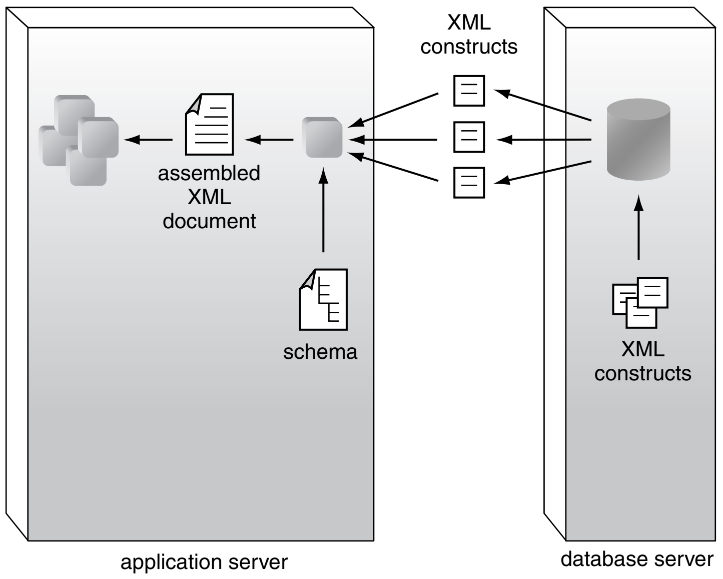


Figure 7.3
A physical architecture where XML constructs are stored independently in a relational database, and then assembled into a complete XML document at runtime by the application.

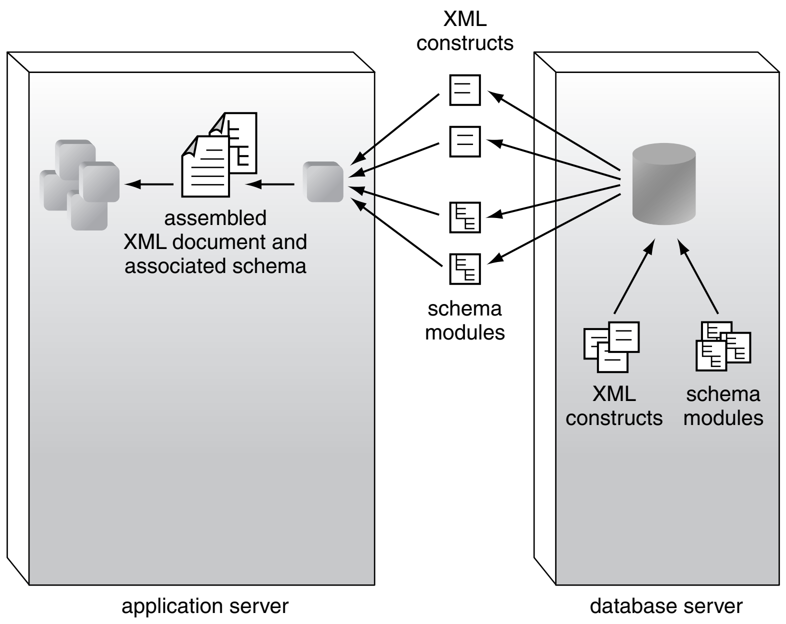


Figure 7.4
Schema modules are stored and assembled alongside XML document constructs in the database.

Pros:

- Establishes a highly reusable data platform.
- Existing data model likely will not be impacted.

Cons:

- Introduces a potentially complex data model extension, and places the burden of document assembly on the application.
- Can complicate validation, as creating and maintaining schemas for all possible construct combinations may become an unwieldy task.
- All of the cons listed under the previous section.

7.2.3 Using XML to represent a view of database queries

This model allows for dynamically created views of database queries, represented as XML documents. The resulting architecture (shown in Figure 7.5) is most common when working with the proprietary XML extensions provided by database vendors.

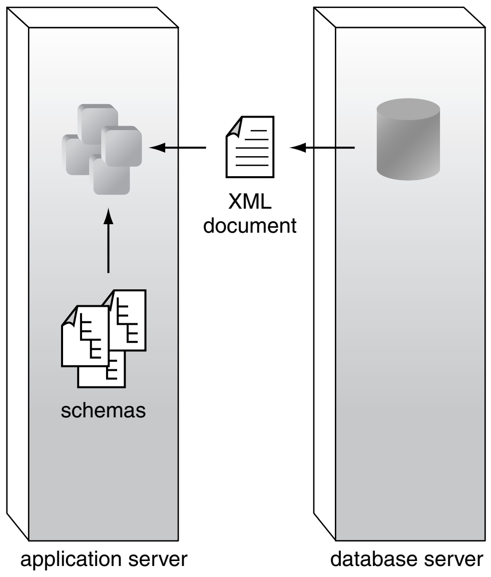


Figure 7.5
In this physical architecture, the requested data is returned in the format of an XML document by the database.

Suitable for:

- Dynamically created XML documents with a limited lifespan. In other words, for when XML is used as a transport format for legacy data.
- XML documents auto-generated by proprietary database extensions. Most database vendors provide a way of outputting legacy data into an XML format.
- Lookup tables and static data retrieved and stored in memory at runtime, by the application.

Pros:

- If using proprietary database extensions, this architecture is relatively easy to implement.
- Does not affect existing legacy data model.

Cons:

- When using proprietary database extensions, the degree to which this output format can be utilized is limited to the features of the database product. Some databases output unrefined (and sometimes cryptic) XML markup. These documents often require further processing and filtering by the application. (See the "Database extensions" section later in this chapter for more information.)
- Proprietary database extensions tie you to a database platform, and much of XML's mobility is lost.

7.2.4 Using XML to represent a view of a relational data model

In this architecture, XML documents are modeled to accurately represent portions of the legacy data model (see Figure 7.6). This approach requires the most up-front design work to properly map relational data structures to XML's hierarchical format.

Suitable for:

- Applications requiring an accurate representation of select relational data entities.
- Applications that need to perform granular updates of legacy data hosted in XML documents.

Pros:

- Establishes a highly flexible and mobile data transport mechanism that accompanies data with a self-contained data model.

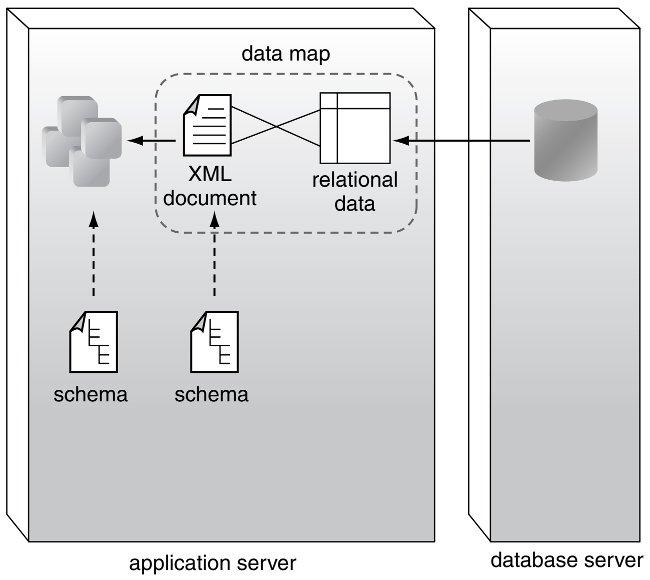


Figure 7.6
A physical architecture illustrating the use of a data map to represent relational data within an XML document.

- RDBMS-comparable support for inserts and updates.
- Focus on XML parsing provides a comprehensive abstraction of RDBMS data access.

Cons:

- Complex to design and maintain.
- Sophisticated approach, but still does not remove the need to perform queries against the database.

7.2.5 Using XML to represent relational data within an in-memory database (IMDB)

XML actually can be utilized to increase application performance by caching relational data on the application server. This architecture can be combined with others, depending on how you want to represent and map the relational data bodies to XML documents, and whether you need to preserve inter-table relationships.

Once you've defined how XML is to represent your data, you can have a utility component retrieve the information once, upon the start of an application, or periodically, based on preset refresh intervals. As explained in Figure 7.7, this component can then parse and load XML documents into a globally accessible memory space.

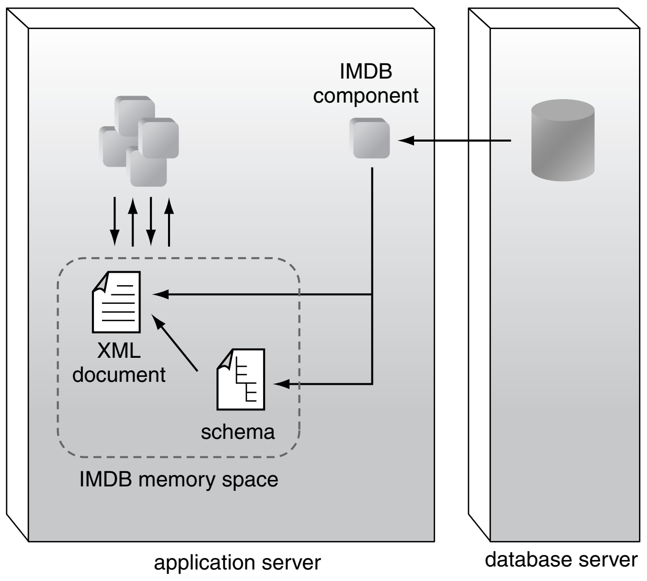


Figure 7.7
A physical architecture in which XML data is cached in memory.

Suitable for:

- Applications with high-usage volumes.
- Relatively static data, such as lookup tables.

Pros:

- Shifts processing load from database to application server, which can dramatically increase performance.
- Cost-effective method of scaling an application (memory is cheaper than database licenses).

Cons:

- May introduce vertical scaling requirements.
- Although faster, the overall architecture is less robust (memory space is more volatile and environmentally sensitive than a hard drive).

7.3 Strategies for integrating XML with relational databases

Section 7.4 is dedicated to data mapping, and provides numerous techniques for overcoming the inherent differences between XML and relational data models. Before you

SUMMARY OF KEY POINTS (Section 7.2)

- There is no one standard integration architecture for XML and relational databases.
- Architectures often will need to be designed around the features and limitations of the proprietary database platform.
- The storage of XML documents may vary from the storage of schemas. Sometimes XML documents are auto-generated, and need to be linked dynamically to existing schema files.

delve into the mysterious world of data maps, relationship pointers, and relational hierarchies, here are more high-level integration strategies to keep in mind.

7.3.1 Target only the data you need

There is one simple design principle that you should take to heart: Whenever possible, restrict application data to the context of the current business task. In other words, only encapsulate the pieces of data and the parts of the data model relevant to the immediate application function.

There are two reasons this one design point is important:

Performance

Keeping the data model on the XML side of your architecture lean is a crucial step to designing performance optimization into your application. It is very easy simply to take the results of a query and stuff them into a generic or dynamically generated document structure, especially when working with database extensions that already do this for you.

Designing XML documents to represent too much data is a surprisingly common pitfall, which almost always results in eventual performance challenges. It's more effort to discern the parts of data you really need from those that can be discarded. The time you invest in order to put together a proper document model design up-front is nothing compared to the effort required to redesign and re-implement a new document model once the application has been deployed. (See Chapter 5 for many XML document modeling guidelines and standards.)

Task-oriented data representation

Your legacy repository may not have been designed to accommodate your current or future application tasks. Each task performed by an application represents a business function that involves a subset of your corporate data within a unique context.

By the mere fact that your application architecture is already XML-enabled, you have the opportunity to model your documents so that they can best represent only the data relevant to the business task at hand. This means that you can custom-tailor your data representation (while still preserving its validation and relational rules) to establish a data view that relates to the requirements of the application function currently being executed.

This moves into the area of object- or class-based data mapping, but we'll call it task-oriented data representation for now. All that I recommend is that you present the data in such a way that it can easily be consumed and processed by the application, while also being associated with a business task.

Task-oriented data representation also can improve application maintenance. Since your data is uniquely identified with a specific business task, it can be more easily traced and logged.

7.3.2 Avoiding relationships by creating specialized data views

If you are repeatedly working with the same sets of legacy data, you can save a great deal of integration effort by pre-consolidating this information into a database view. Instead of having to map to, extract, and assemble multiple tables every time, you can simply map the one view of data to one or more XML documents.

Additionally, if your database provides XML support, and your column names are self-descriptive, you can have the database auto-generate relatively optimized XML markup, on demand.

Finally, you can supplement this approach by creating XSD schemas or DTDs in support of each view. Do this, however, only if you are certain that the views are fairly permanent. Also, note that views are often read-only. In this case, view-derived documents would not be suitable for updates and inserts, as your data will not be accompanied by the necessary data model rules.

7.3.3 Create XML-friendly database models

If you are in a position to build a brand new database, you can take a number of steps to streamline the integration process with XML documents. Here are some suggestions.

Avoid granular tables and relationships

This isn't to suggest that you should compromise the integrity of your data model, but if you do have the choice between creating a series of larger tables and a multitude of smaller joined tables, you will save yourself a great deal of mapping effort by cutting down on inter-table relationships.

Support preformatted XML views
Consider adding tables to store for XML documents representing redundant views of static legacy data. Your database may provide support for automatically generating and synchronizing these views, via the use of stored procedures, triggers, or other extensions.

Consider descriptive column names
If you will be using proprietary XML extensions provided by your database, you may be subjected to auto-generated XML documents based entirely on your existing DDL syntax. By having self-descriptive column names, you will end up with more self-descriptive XML documents. Consider this only if you actually need your XML documents to contain descriptive element-type names. Read through the section, "Naming element-types: performance vs. legibility," in Chapter 5 to learn more about the implications of using self-descriptive element-types.

Avoid composite keys
If you will be mapping your data to DTDs, avoid composite keys. For the purpose of mapping relationships within XML documents, it is preferable to uniquely identify a record by adding a primary key rather than by defining a key based on a combination of multiple column values. If you are working with XSD schemas, however, recreating composite keys will be less of an issue.

7.3.4 Extending the schema model with annotations

Regardless of the schema technology you end up using, a hierarchical data representation can only reflect the complexities of a relational data model to a certain extent. You often will find yourself compensating for gaps by writing application routines that supplement the schema validation with custom data rules and processing.

A classic example is the enforcement of referential integrity. A relational database will typically allow you to propagate and cascade updates or deletions to column values involved in a relationship. Deleting an invoice record, for instance, automatically will delete all associated invoice detail records.

Even though this type of rule enforcement will need to be processed by the application, it is often preferable for these rules to still exist within the schema file itself, as opposed to residing independently in application components. This is where schema annotations are very useful.

By creating a standard set of codes to represent common processing rules, you can embed processing statements as comments or annotations within each schema file. Especially when using the `appinfo` element in XSD schemas, these annotations are easily parsed, retrieved, and processed by the application at runtime. (For more information on annotating XSD schemas, refer to the "Supplementing XSD schema validation" section in Chapter 5.)

7.3.5 Non-XML data models in XML schemas

Continuing from the previous section, let's take this technique a step further. If your application environment consists of a mixture of data formats, you could place validation rules and processing instructions within your schema annotation that apply to non-XML formatted data. In this case, a schema would typically be related to a business task, and could then encompass the validation rules of any data involved in the execution of that task, regardless of format.

By centralizing all data-related rules into one file, you retain the mobility and extensibility of the XML application model. By creating standards around the code syntax, you also establish a loosely coupled relationship between the application and your data model.

7.3.6 Developing a caching strategy

Retrieving and composing XML documents at runtime can be a processor-intensive task, especially if you need to perform some form of dynamic linking. To minimize the amount of times a particular body of application data is generated this way, develop a caching strategy to hold an XML document in memory as long as possible.

The in-memory database architecture (as illustrated in the "Using XML to represent relational data within an in-memory database (IMDB)" section) demonstrates the performance benefits of caching legacy data on the application server within XML documents. XML data is very well suited for storage in memory, and even if you don't build a formal architecture around the use of an IMDB, you should consider developing a strategy for caching documents (or perhaps constructs) whenever possible.

Suitable types of data include:

- static report data
- lookup tables
- state and session information
- application configuration parameters
- processing instructions and validation rules

…and pretty much any other piece of (relatively static) information that will need to be accessed throughout the lifetime of an application instance. Note that security requirements and memory limitations may restrict the type and amount of data you can place in memory.

7.3.7 Querying the XSD schema

When working with XSD schemas, the data model established by the schema is open to be queried and parsed, as any other XML document. This gives developers a standard API into the structure, constraints, and validation characteristics of any piece XML formatted data (see Figure 7.8).

By querying schema files at runtime, applications can dynamically retrieve the data model. This facilitates the development of highly intelligent and adaptive application components that can respond to changes in auto-generated schema files.

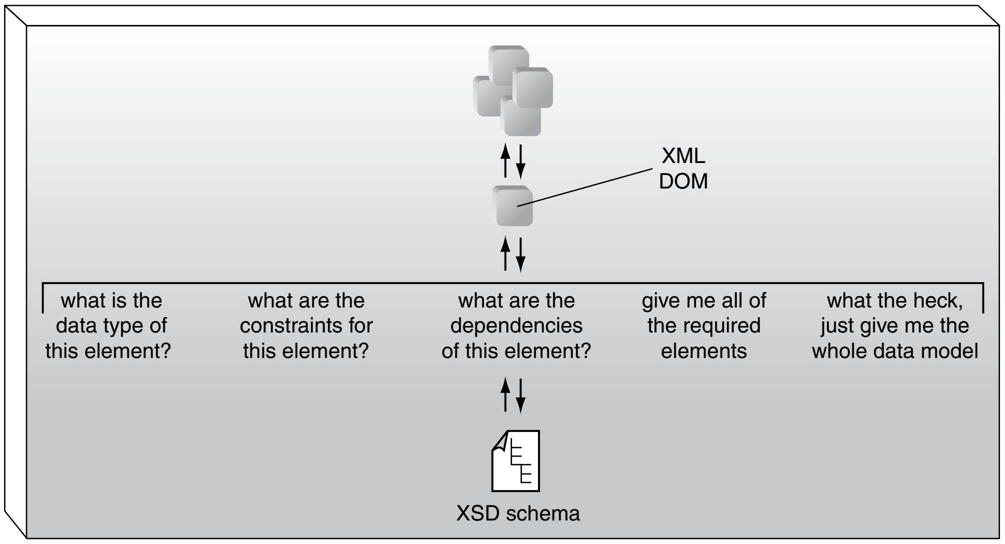

XML DOM
what is the data type of this element?
what are the constraints for this element?
what are the dependencies of this element?
give me all of the required elements
what the heck, just give me the whole data model
XSD schema

Figure 7.8
Application components can query the XSD schema file as they would any other XML document.

NOTE

If you have an XML processing library that supports a schema object model, you can also interface with an XSD schema programmatically. This would allow you to modify or even generate schema data models dynamically.

7.3.8 Control XML output with XSLT

One of the limitations of any hierarchy is that the order in which items are structured is generally fixed. If you are mapping table columns to elements or attributes within your

XML document, you may be limited to the sequence in which these elements or attributes are declared within your schema.

XSLT can provide a convenient way to:

- alter the document structure
- change the sorting order (as in Figure 7.9)
- introduce a series of logical element groups

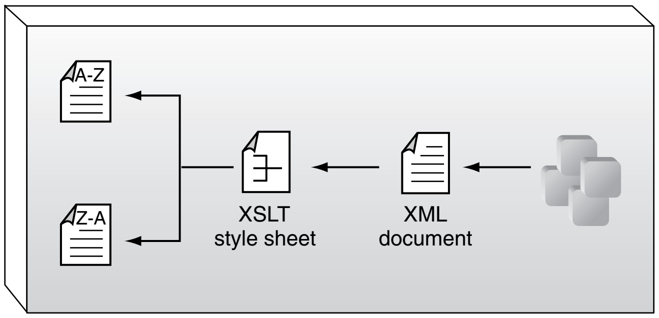


Figure 7.9
The structure of an XML document is transformed to represent different sort orders.

By dynamically creating structure-oriented XSLT style sheets and associating them with your newly populated XML documents, you can build a flexible data manipulation system that can accommodate a number of different output formats from the same data source.

7.3.9 Integrate XML with query limitations in mind

It’s no secret that querying XML documents can be a slow and inefficient means of data retrieval. Where RDBMSs have indices they can utilize for nearly instant access to key pieces of data, XML parsers are forced to iterate through the document nodes in order to locate the requested information.

It is therefore preferable to have the database do as much of the querying prior to subsequent application processing of the data. If you are considering preserving relationships and other aspects of your relational data within XML documents, then try to incorporate specialized views that pre-query the data you want to represent.

If you are unsure of how data will be queried once it is returned to the application, try to model your XML documents into granular sections that can be searched faster.

7.3.10 Is a text file a legitimate repository?

After establishing the limitations of XML documents as a storage medium for corporate data, is there a point in ever considering XML documents as a valid repository? The answer is “yes, but only to a limited extent.”

XML never aspired to replace the data storage capabilities of traditional databases. When XML was originally conceived, it was intended to host document-centric Web content in a structured manner, supplemented by descriptive and contextual meta information. Within an application architecture, its strength is providing a highly flexible and mobile data representation technology that can be applied in many different ways throughout a technical environment.

There are a number of situations when it may be appropriate to persist XML documents as physical files, including:

- static report data
- lookup tables
- state and session information
- application configuration parameters
- processing instructions and validation rules

If this list looks familiar, it’s because it’s identical to the list of data recommended for use with an IMDB. Although you may not necessarily need to load all of the data kept in physical XML documents into memory (or vice versa), the general rules apply for each approach, because your data is being hosted on the application server either way.

One additional item that can be added to this list is a configuration file in support of IMDBs, in which refresh-and-upload intervals are stored.

7.3.11 Loose coupling and developer skill sets

One of the often-overlooked benefits of abstracting data access from the database to XML is that developers no longer need to concern themselves (as much) with vendor-specific data access technology.

Once you’ve built an integration architecture that accomplishes a high level of independence from platform-specific technologies, you will create an environment where developers can concentrate on the management and manipulation of data with only the XML technology set. Web services can play a key role in achieving this level of platform detachment. (Read Chapter 9 for more information.)

SUMMARY OF KEY POINTS

- Performance is, as always, an important consideration when integrating and modeling XML data representation. Caching is a key strategy to overcoming potential bottlenecks.
- Since relational databases are typically an established part of a legacy application environment, most of the integration focus is on designing XML documents around the existing relational data model. Relational databases, however, can also be “adjusted” to contribute to an improved integration.

7.4 Techniques for mapping XML to relational data

Perhaps the most challenging and awkward part of integrating relational databases with XML documents is trying to recreate relationships between database tables within the hierarchical model of XML documents.

Especially when trying to integrate complex and extensive data models, it will often feel like you’re forcing a round peg into a square hole (many times over). Well, life isn’t always easy, and integration projects are no exception. The point — there’s a hole, there’s a peg, let’s grab that hammer and deal with it.

7.4.1 Mapping XML documents to relational data

To integrate a relational data model with XML, some form of data map generally will be required. This map will associate the relevant parts of your legacy data model with the corresponding parts of your XML schema.

There are several tools that can assist this process, some of which even auto-generate the XML schema files for you. You will find, however, that more often than not, an accurate mapping requires hands-on attention and manual changes to the schema markup.

There are several approaches to mapping data, depending on the nature of your data model and the design of your application components. Here are some guidelines for devising a mapping strategy.

Table-based mapping

Mapping tables to parent elements within XML documents is the most common approach. Depending on the nature of the data, columns can be represented as child elements or attributes to the parent record elements.

Template-based mapping

This is a popular alternative, supported by several middleware products. The design provides an effective means of generating XML formatted data, by embedding SQL statements in “hollowed” XML document templates. These statements then are processed by the product at runtime, the database is queried, and the document is populated dynamically.

Class-based mapping
A less frequently used approach, class- or object-based mapping may become more common once Web services establish themselves as a standard part of application architecture. The format of a class-based XML document allows data to be mapped according to class objects and their attributes or method parameters.

7.4.2 The Bear Sightings application

Throughout section 7.4 we will be referencing a simple data model (Figure 7.10) for an application used to keep track of bears that roam into mining camps in the Yukon. This is a common problem with placer mines located in remote areas of the wilderness, and the information gathered by such an application can assist in broadcasting alerts to camp sites, especially if bears exhibiting dangerous behavior are encountered.

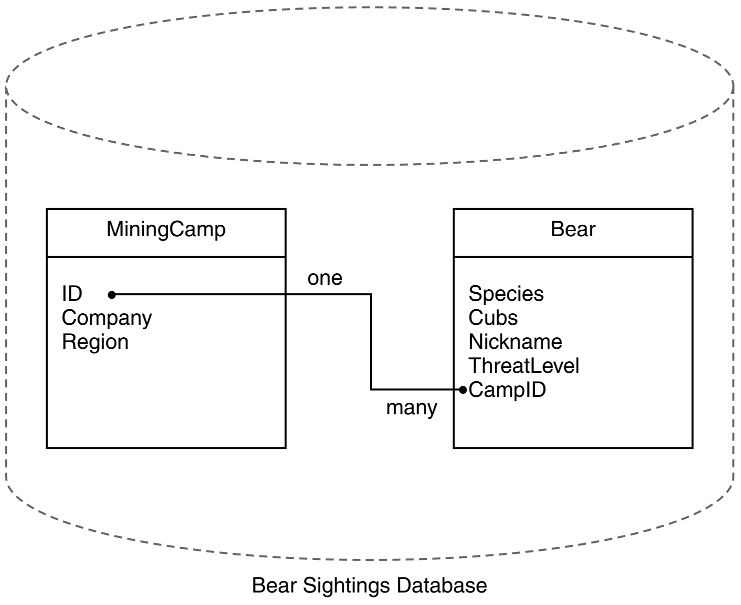
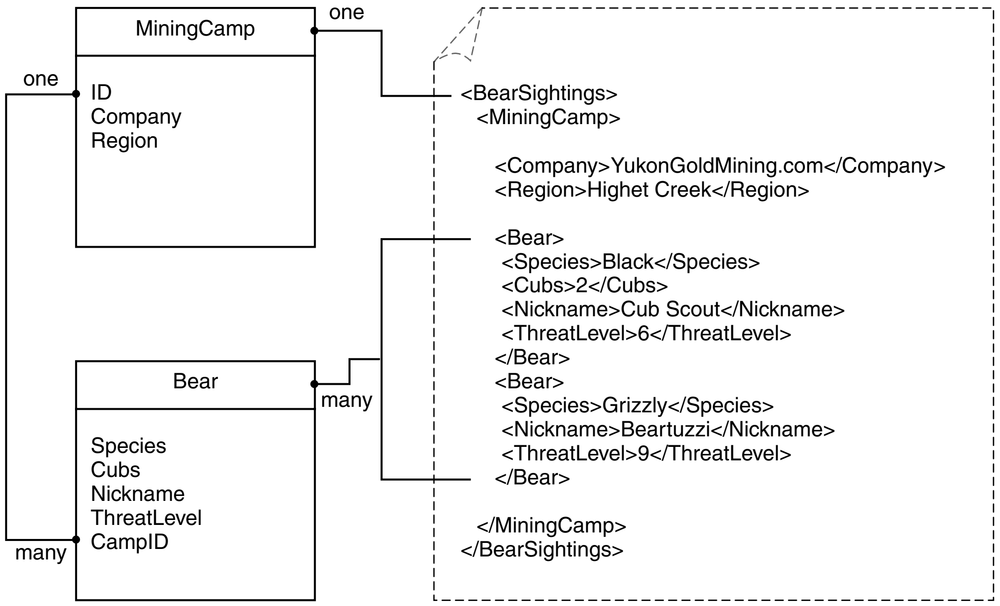


Figure 7.10
The application data model of the Bear Sightings database consists of two simple, related tables.

7.4.3 Intrinsic one-to-one and one-to-many relationships with XML

The hierarchical XML document structure provides natural one-to-one and one-to-many relationships, where single or multiple instances of a child element can be nested within one parent element. As long as the child element requires only a single parent element, you need to do nothing more than define this parent-child hierarchy as you would any other.

Any schema you use should be able to easily enforce a one-to-one relationship. DTDs enable this via the "`?`" symbol (or the absence of a symbol) within the element declaration, and XSD schemas use the `maxOccurs` attribute.

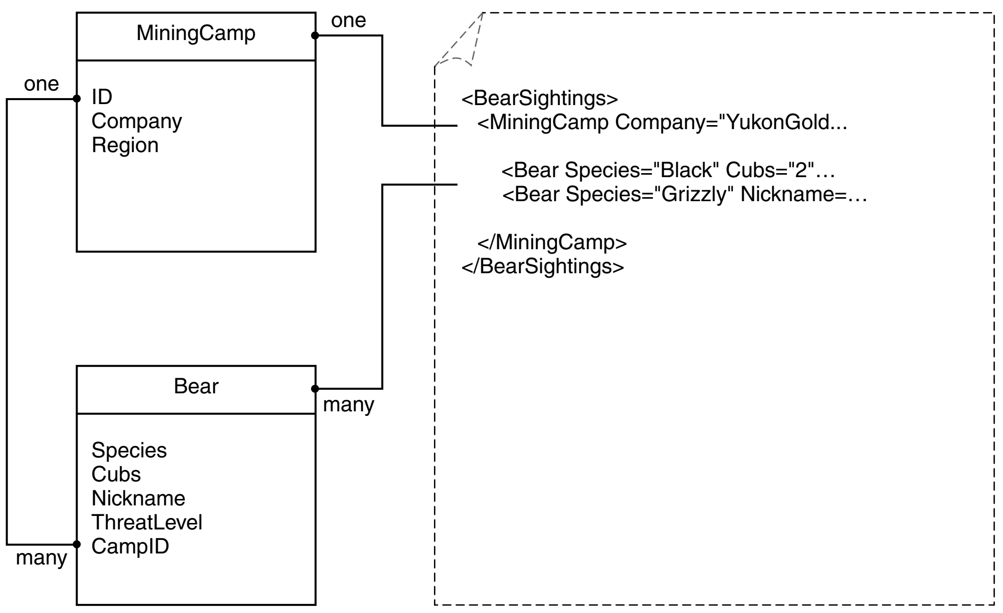


```
<BearSightings>
  <MiningCamp>

    <Company>YukonGoldMining.com</Company>
    <Region>Highet Creek</Region>

    <Bear>
     <Species>Black</Species>
     <Cubs>2</Cubs>
     <Nickname>Cub Scout</Nickname>
     <ThreatLevel>6</ThreatLevel>
    </Bear>
    <Bear>
     <Species>Grizzly</Species>
     <Nickname>Beartuzzi</Nickname>
     <ThreatLevel>9</ThreatLevel>
    </Bear>

  </MiningCamp>
</BearSightings>
```

Figure 7.11
An intrinsic one-to-many relationship established within an element-centric XML document instance.

The example in Figure 7.11 illustrates an element-centric one-to-many relationship, where table columns are represented as child elements. Columns of the MiningCamp table are mapped to child elements of the `MiningCamp` element, and columns of the Bear table are mapped to child elements of the `Bear` element. Figure 7.12 shows the same data represented in an attribute-centric model.

The point at which this intrinsic relationship becomes insufficient is when you need to represent a child element that requires more than one parent element in the same document. That's when it's time to roll up your sleeves and delve into the world of DTD or XSD schema pointers. The following two sections will show you how.



```
<BearSightings>
  <MiningCamp Company="YukonGold...
     <Bear Species="Black" Cubs="2"…
     <Bear Species="Grizzly" Nickname=…
  </MiningCamp>
</BearSightings>
```

Figure 7.12
An intrinsic one-to-many relationship established within an attribute-centric XML document instance.

7.4.4 Mapping XML to relational data with DTDs

Provided here are techniques to accomplish rudimentary relational functionality within DTDs.

NOTE

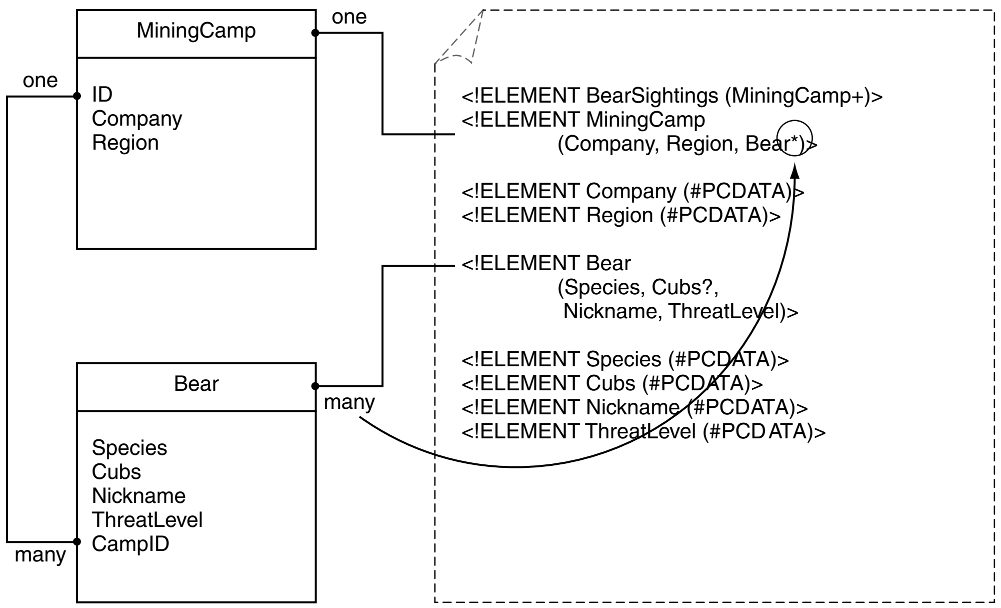
XSD schemas provide more advanced relationship mapping features, as described in the “Mapping XML to relational data with XSD schemas” section.

Basic table mapping with DTDs

Put simply, you can map tables to individual DTDs, or group tables logically into one DTD. A deciding factor is whether or not you also intend to represent relationships between tables. If you do, all tables involved in a relationship will need to be contained within one DTD schema.

As the next few sections thoroughly explore, DTDs generally rely on a series of attributes that simulate pointers, and those pointers apply only within the boundary of an XML document.

Let’s begin by revisiting our previous data model, and associating it with a basic DTD.



```
<!ELEMENT BearSightings (MiningCamp+)>
<!ELEMENT MiningCamp
            (Company, Region, Bear*)>

<!ELEMENT Company (#PCDATA)>
<!ELEMENT Region (#PCDATA)>

<!ELEMENT Bear
            (Species, Cubs?,
             Nickname, ThreatLevel)>

<!ELEMENT Species (#PCDATA)>
<!ELEMENT Cubs (#PCDATA)>
<!ELEMENT Nickname (#PCDATA)>
<!ELEMENT ThreatLevel (#PCDATA)>
```

Figure 7.13
A DTD containing a parent element for each table.

As the example in Figure 7.13 demonstrates, the one-to-many relationship is established by the nesting of the `Bear` element. The asterisk symbol in the `MiningCamp` element is used to enable multiple occurrences of the `Bear` child element.

NOTE

If you are not mapping relationships, you have the flexibility to control the granularity of your DTDs and their corresponding XML documents. This will allow you to accommodate application performance requirements. High data volumes and complex document structures can justify distributing a relational model across multiple document types.

Data type restrictions with DTDs

A significant limitation to representing a relational data model within a DTD is the inability of DTD schemas to type data properly. The DTD language supports only four types of data: ANY, EMPTY, PCDATA, and element-only content.

Most of the time you will find yourself lumping your database columns into elements that are of type PCDATA. This places the burden of figuring out the nature of your data on the application.

One method of countering this limitation is to annotate the DTD with data type information for each element. Alternatively, you can add custom attributes to elements that identify the native database data types, as demonstrated here.

```
<ThreatLevel DataType="integer">9</ThreatLevel>
```

Example 7.1 An element instance with a custom attribute identifying the original data type

The problem with these types of workarounds is that they introduce a non-standard solution to a common problem. Outside of applications that are aware of the purpose behind the custom attributes, this solution is not useful.

Null restrictions with DTDs

DTDs have no concept of null values, which can turn into another challenge when wanting to accurately represent the data values found in databases. Since at least some of your database tables will likely allow and contain null values, you need to establish a way of expressing them within your DTD schemas.

Here are some suggestions:

- Nulls can be represented by an absence of a child element or attribute. Essentially, if the element or attribute is present and empty, it is displaying an empty value. If the element or attribute is not included in an instance of the parent element, then that indicates a null value.
- The value of null can be represented by a keyword. You can create a standard code, say "NULL," that your application can look for and interpret as a null value, as shown in this example.

```
<ThreatLevel>NULL</ThreatLevel>
```

Example 7.2 An element instance indicating a null value through the use of a pre-assigned code

As with using custom attributes to represent non-DTD supported data types, these solutions are non-standard. If your application will need to interoperate with others, the implemented method of null value management will not be evident, and may very well be ignored.

Representing relational tables with DTDs

Using the `ID`, `IDREF`, and `IDREFS` attributes provided by the XML specification (and further explained in subsequent sections), DTDs can define and enforce the uniqueness of an element, as well as constraints between elements.

If database tables are represented as separate XML constructs within an XML document, DTDs can simulate basic inter-table relationships, as well as the use of primary and foreign keys. When using DTDs for this purpose, however, you are restricted to representing database tables involved in relationships within one XML document.

NOTE

In order to achieve cross-document relationships you may need to consider using XLink and XPointer. With these supplementary technologies, the ID attribute can still be used to identify the element being referenced.

Primary keys with DTDs

The XML specification provides the `ID` attribute to allow unique identifiers to be assigned to XML elements. This can be useful to the application parsing the XML document, because it can search for and identify elements based on this value.

In the following example, we declare an attribute of type `ID` and also call it "id" (we could just as easily call it "ReferenceID" or "MiningCampID").

```
<!ELEMENT MiningCamp (Company, Region, Bear*)>
<!ATTLIST MiningCamp id ID #REQUIRED>
```

Example 7.3 An element type declaration with an ID attribute

For the purpose of establishing relationships, the `ID` attribute can simulate a primary key for an element construct that represents a database table.

There are two major limitations when using the `ID` attribute:

- The attribute value cannot begin with a number. Since database tables frequently use incrementing numeric values for keyed columns, you will often need to programmatically modify this data before placing it into the ID attribute.
- ID values need to be unique within the entire XML document, regardless of element type. Since you are restricted to representing all the tables involved in a relationship within the scope of one XML document, you may run into ID value collisions. (Incidentally, this makes the ID attribute useful as an element index value.)

The easiest way to solve both of these issues is to prefix your key values with a code that relates the ID value to its associated table.

For instance, imagine you are representing both Invoice and PO tables within one document. In your database, each table's primary key is a column called "Number," which uniquely identifies a record. An Invoice record that has a Number value of 1001 and a

PO record that also has a Number value of 1001 are legitimate within a database, however they are not within a DTD-validated XML document.

To incorporate Invoice and PO Number keys as `ID` attributes within an XML document, these values could instead be represented as “INV1001” and “PO1001.” This is obviously not an accurate representation of table data, but it does achieve the functionality required to simulate primary keys (to an extent).

For the purpose of our example, we are prefixing the MiningCamp table’s numeric ID values with the word “Camp.” Our sample mining camp therefore has an `ID` value of “Camp1.”

```
<MiningCamp ID="Camp1">
```

Example 7.4 A primary key represented by the ID attribute

Foreign keys with DTDs

The XML specification enables cross-element referencing of the `ID` attribute by providing the `IDREF` and `IDREFS` attributes. An element can assign the ID value of another element to its `IDREF` attribute, thereby establishing a relationship between the two, similar to the relationship between a primary key and a foreign key within a database.

```
<!ELEMENT Bear (Species, Cubs?, Nickname, ThreatLevel)>
<!ATTLIST Bear CampID IDREF #REQUIRED>
```

Example 7.5 An element type declaration containing a foreign key reference

The `IDREFS` attribute is identical to `IDREF`, except that it allows you to reference multiple ID values. One element, therefore, can have references to multiple others.

Since XML documents typically represent a portion of a database’s relational model, they will often evolve. If the scope of your application grows, you may find your DTD expanding as well, as it needs to represent more relationship information. You can therefore use the `IDREFS` attribute, even if you are referencing only one value initially. This way you can add references as required without changing the original element definition.

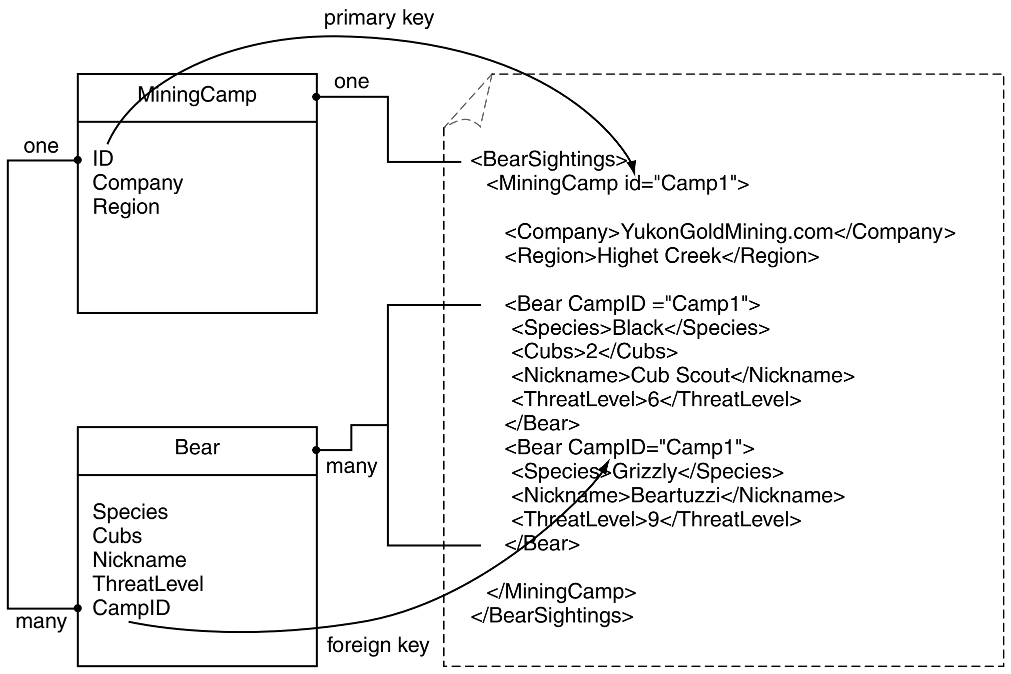
NOTE

The XML schema language also supports attributes of type `IDREF` and `IDREFS`. However, since XSD schemas introduce more sophisticated ways of establishing relationships between elements, these attributes are rarely used. The subsequent section in this chapter is dedicated to relationship mapping with XSD schemas.

Relationships with DTDs

The pointing mechanism established in the previous sections (using the `ID`, `IDREF`, and `IDREFS` attributes) provides you with the ability to set up a series of sequential element constructs within a DTD. This allows you to identify how these elements relate, and enables you to simulate various database table relationships. All of this can lead to the creation of a relatively normalized DTD schema.

The intrinsic one-to-many relationship illustrated in the "Intrinsic one-to-one and one-to-many relationships with XML" section can be recreated using DTD pointers, as illustrated in Figures 7.14 and 7.15.

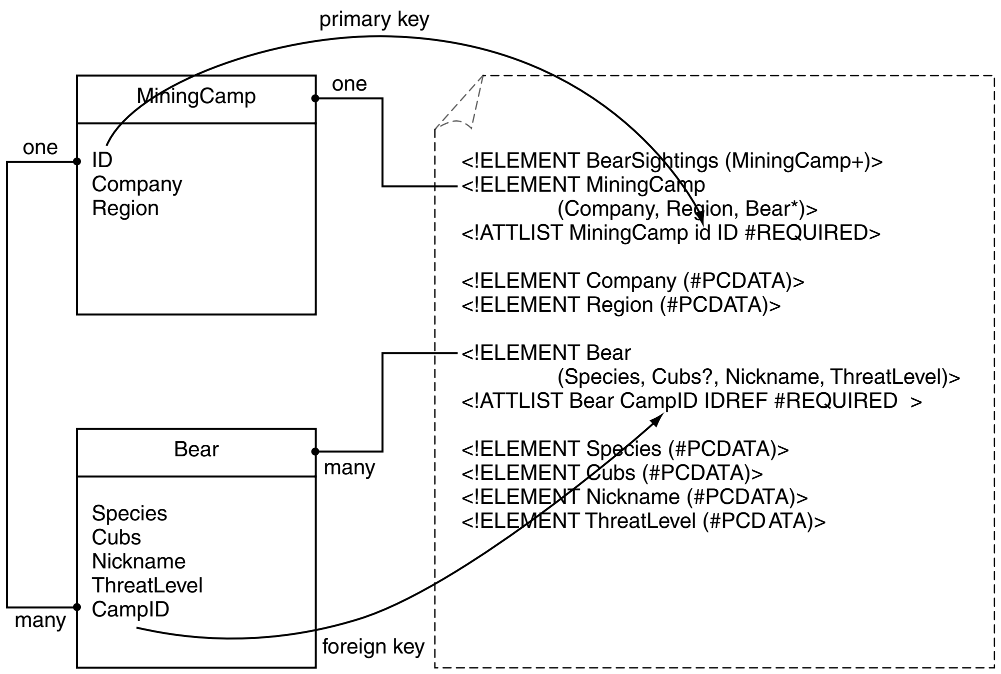


```
<BearSightings>
  <MiningCamp id="Camp1">

    <Company>YukonGoldMining.com</Company>
    <Region>Highet Creek</Region>

    <Bear CampID ="Camp1">
     <Species>Black</Species>
     <Cubs>2</Cubs>
     <Nickname>Cub Scout</Nickname>
     <ThreatLevel>6</ThreatLevel>
    </Bear>
    <Bear CampID="Camp1">
     <Species>Grizzly</Species>
     <Nickname>Beartuzzi</Nickname>
     <ThreatLevel>9</ThreatLevel>
    </Bear>

  </MiningCamp>
</BearSightings>
```

Figure 7.14
A one-to-many relationship using ID and IDREF.

This example may be interesting, but it's not really that useful. We have not gained anything over representing the one-to-many relationship without the use of pointers. The real power of DTD pointers is realized when you have a set of child elements that are required to relate to multiple parent elements.

In the example provided in Figure 7.16 the `Bear` elements are not explicitly nested within the parent `MiningCamp` element. Instead, they exist as separate constructs and



```
<!ELEMENT BearSightings (MiningCamp+)>
<!ELEMENT MiningCamp
          (Company, Region, Bear*)>
<!ATTLIST MiningCamp id ID #REQUIRED>

<!ELEMENT Company (#PCDATA)>
<!ELEMENT Region (#PCDATA)>

<!ELEMENT Bear
          (Species, Cubs?, Nickname, ThreatLevel)>
<!ATTLIST Bear CampID IDREF #REQUIRED  >

<!ELEMENT Species (#PCDATA)>
<!ELEMENT Cubs (#PCDATA)>
<!ELEMENT Nickname (#PCDATA)>
<!ELEMENT ThreatLevel (#PCDATA)>
```

Figure 7.15
A DTD establishing ID and IDREF attributes.

reference the corresponding `ID` attribute (the primary key of the MiningCamp table), using their own `IDREF` attribute (which acts as the foreign key of the Bear table). Figure 7.17 provides the corresponding DTD.

When defining a one-to-one relationship, ensure that the declaration syntax allows a maximum of one instance of the child element within the parent element construct. This is accomplished by using the "?" symbol in the declaration statement, as shown here.

```
<!ELEMENT Bear (Cubs?)>
```

Example 7.6 An element type declaration restricting a child element to zero or one occurrence

Alternatively, you can also establish a one-to-one relationship by embedding the column values of the database record into the element as a series of attributes.

Referential integrity restrictions within DTDs

When using `IDREF` and `IDREFS`, a DTD cannot enforce an erroneous occurrence of these attributes. For instance, let's say an element representing a Country Code lookup table contains an `IDREF` value that corresponds to a valid Invoice ID value, like

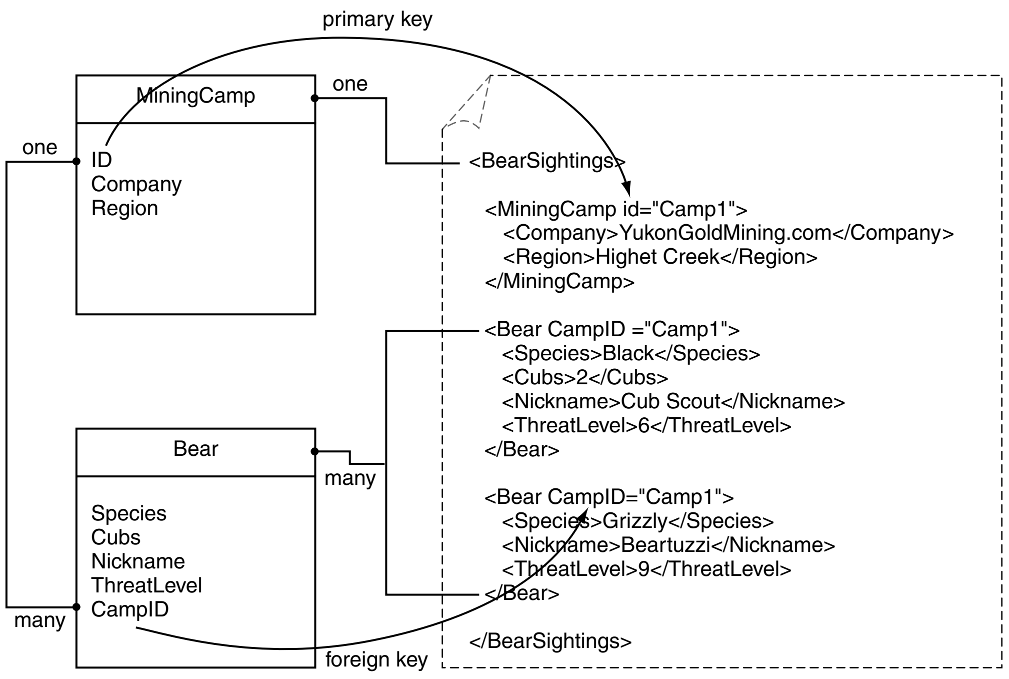


Figure 7.16
A different one-to-many relationship using ID and IDREF.

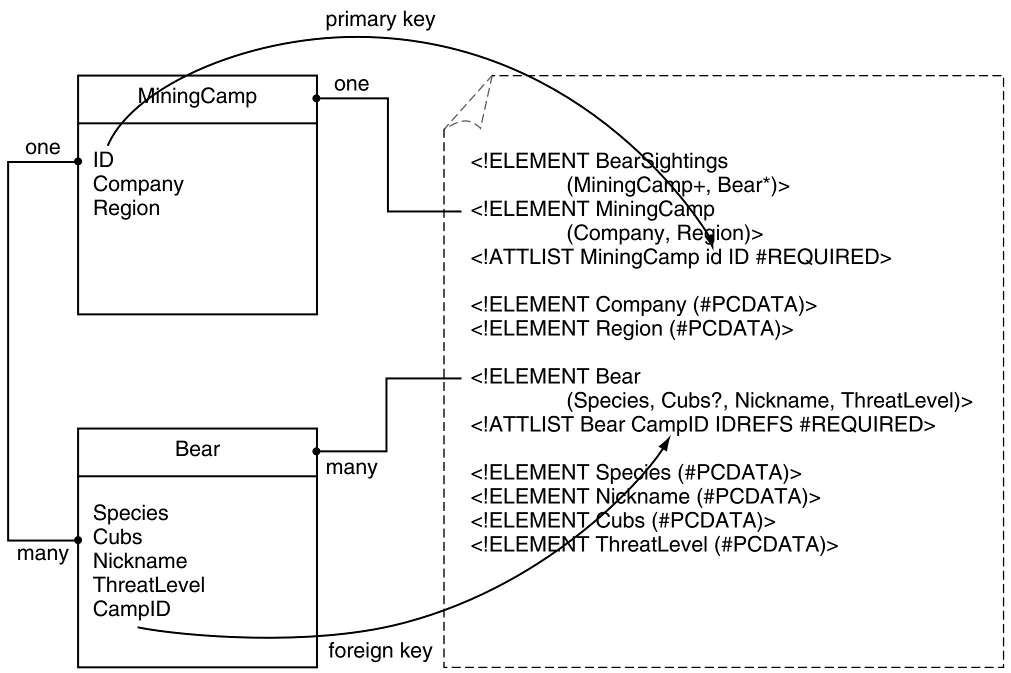
"INV1001." In databases we can set up constraints to enforce foreign key relationships between two tables. If no such relationship exists between the Invoice table and the Country Code table, the Country Code table cannot reference a primary key value from the Invoice table.

In an XML document, however, the DTD has no concept of explicit relationships between two types. It simply keeps track of `ID` and `IDREF`/`IDREFS` attributes, and makes sure that all `IDREF` and `IDREFS` values consist of valid ID values somewhere in the document. It doesn't matter where the `IDREF` or `IDREFS` values are located, as long as they are present and the value is unique.

It is therefore important to understand that DTDs cannot provide true referential integrity. DTDs allow for a system of pointers that can be utilized to simulate database table relationships to a limited extent.

7.4.5 Mapping XML to relational data with XSD schemas

The XML schema language provides a number of features that are very useful for representing relational data. One notable difference in how XSD schemas approach the



```
<!ELEMENT BearSightings
            (MiningCamp+, Bear*)>
<!ELEMENT MiningCamp
            (Company, Region)>
<!ATTLIST MiningCamp id ID #REQUIRED>

<!ELEMENT Company (#PCDATA)>
<!ELEMENT Region (#PCDATA)>

<!ELEMENT Bear
            (Species, Cubs?, Nickname, ThreatLevel)>
<!ATTLIST Bear CampID IDREFS #REQUIRED>

<!ELEMENT Species (#PCDATA)>
<!ELEMENT Nickname (#PCDATA)>
<!ELEMENT Cubs (#PCDATA)>
<!ELEMENT ThreatLevel (#PCDATA)>
```

Figure 7.17
DTD providing ID and IDREFS attributes.

definition of keys, is that they incorporate XPath statements to address the shortcomings of DTDs we just discussed.

Although XSD schemas do still support the `ID`, `IDREF` and `IDREFS` attributes discussed in the previous section, here we focus on the parts of the XML Schema language that were added specifically to address relational data mapping requirements.

Basic table mapping with XSD schemas

When mapping database tables to XSD schemas, you generally represent tables as elements with complex types, where each column exists as a simple type element (or as a nested complex type element, when required).

```
<element name="Bear">
   <complexType>
     <attribute name="Species" type="string" />
     <attribute name="Nickname" type="string" />
     <attribute name="ThreatLevel" type="integer" />
   </complexType>
</element>
```

Example 7.7 An element declaration representing three columns from a relational table

Null restrictions with XSD schemas

Even though the XML schema language supports the null value, it does so only for elements. Attributes cannot contain nulls, and therefore any table column to which you map an attribute should not allow nulls to avoid validation conflicts.

NOTE

If you really do need to map a null-allowed column to an attribute, refer to the "Null restrictions with DTDs" section for customized null value management techniques that can also be applied to XSD schemas.

Primary keys with XSD schemas

Elements within XSD schemas can be exclusively identified using the `unique` element. Similar in nature to the `ID` attribute, this element provides more flexibility, and uses XPath to define the scope of its uniqueness.

```
<unique name="MiningCampID">
  <selector xpath=".//MiningCamp" />
  <field xpath="PrimaryKey" />
</unique>
```

Example 7.8 The XSD schema unique element

For the purpose of representing relational data, however, the `key` element is more suitable. As with `unique`, the `key` element enforces a level of uniqueness among the elements or attributes returned by an XPath statement.

```
<key name="MiningCampPrimaryKey">
  <selector xpath=".//MiningCamp" />
  <field xpath="PrimaryKey" />
</key>
```

Example 7.9 The XSD schema key element

The `key` element is specifically intended to be referenced by the `keyref` element. This establishes a primary-to-foreign key relationship.

NOTE

Unlike the `ID` attribute in DTDs, `key` element values can be numeric.

Foreign keys with XSD schemas

Elements that need to reference other elements can use `keyref`. This element defines a foreign key that points to a primary key, based on the `key` element just explained.

```
<keyref name="MiningCampForeignKey" refer="x:MiningCampPrimaryKey">
  <selector xpath=".//Bear" />
  <field xpath="ForeignKey" />
</keyref>
```

Example 7.10 The XSD schema keyref element

As with the `unique` and `key` elements, `keyref` relies on XPath statements to define the region of an XML document to which it applies.

Composite keys with XSD schemas

Databases allow for the creation of composite keys, which derive the key value from a combination of table columns. As long as that combination is unique throughout the table, the key is valid.

XSD schemas provide the same functionality. Whether declaring `unique`, `key`, or `keyref` elements, you can define multiple elements or attributes by adding a `field` element for each.

```
<key name="BearKey">
 <selector xpath=".//Bear" />
 <field xpath="Nickname" />
 <field xpath="Species" />
</key>
```

Example 7.11 A composite key created by multiple field elements

Note that composite keys can consist of elements with different data types.

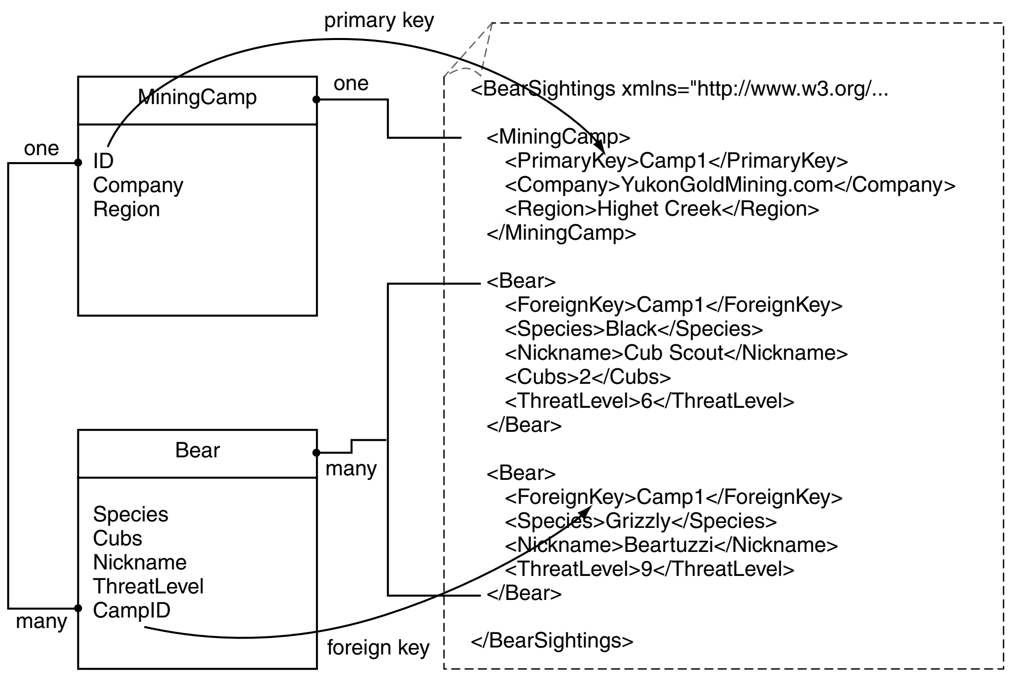
Relationships with XSD schemas

Intrinsic one-to-one and one-to-many relationships are adequate for when child elements have only one parent. For a more flexible schema that allows an element to be referenced by multiple others, you will need to use the `key` and `keyref` elements explained in the previous sections.

By establishing the primary key of your child element with a key value, you will be able to add a corresponding `keyref` element to each parent that needs to reference it. You can set the `maxOccurs` indicator to control how many instances of the child element you want to allow.

Figure 7.18 provides an example that demonstrates an XSD schema-based constraint. Note that we have named the elements representing table keys “PrimaryKey” and “ForeignKey,” respectively.

Next are the contents of the corresponding XSD schema file, followed by the diagram in Figure 7.19 that illustrates the relationship between the schema and table keys.

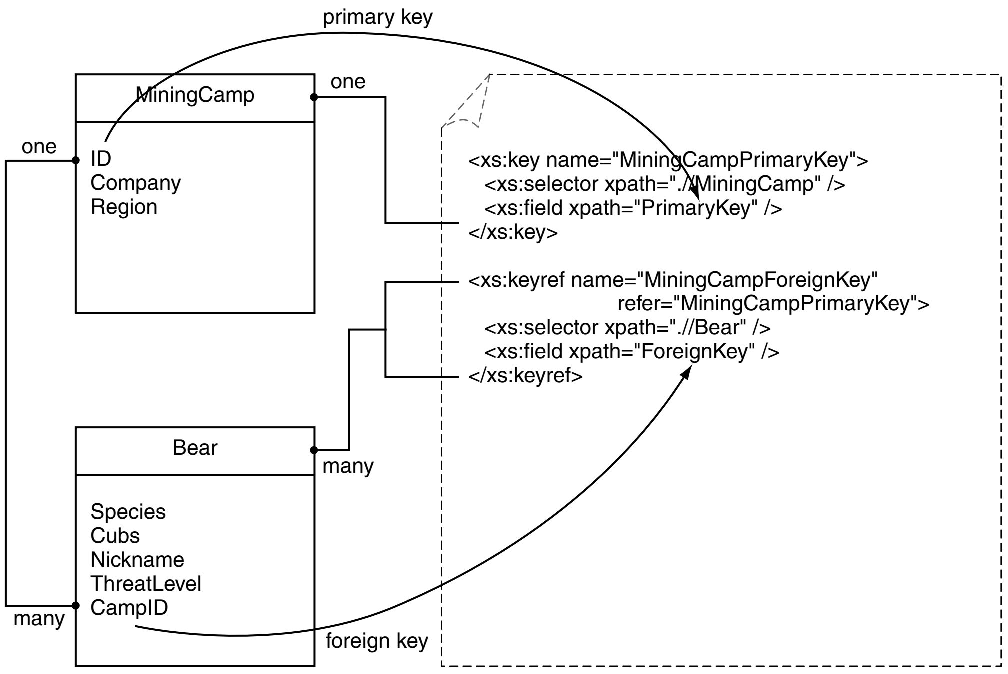


```
<BearSightings xmlns="http://www.w3.org/...

  <MiningCamp>
    <PrimaryKey>Camp1</PrimaryKey>
    <Company>YukonGoldMining.com</Company>
    <Region>Highet Creek</Region>
  </MiningCamp>

  <Bear>
    <ForeignKey>Camp1</ForeignKey>
    <Species>Black</Species>
    <Nickname>Cub Scout</Nickname>
    <Cubs>2</Cubs>
    <ThreatLevel>6</ThreatLevel>
  </Bear>

  <Bear>
    <ForeignKey>Camp1</ForeignKey>
    <Species>Grizzly</Species>
    <Nickname>Beartuzzi</Nickname>
    <ThreatLevel>9</ThreatLevel>
  </Bear>

</BearSightings>
```

Figure 7.18
A document instance with a one-to-many relationship enforced by an XSD schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <xs:element name="BearSightings">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">

  <xs:element name="MiningCamp">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="PrimaryKey" type="xs:string" minOccurs="1" />
     <xs:element name="Company" type="xs:string" />
     <xs:element name="Region" type="xs:string" />
    </xs:sequence>
   </xs:complexType>
  </xs:element>

   <xs:element name="Bear">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="ForeignKey" type="xs:string" minOccurs="0" />
      <xs:element name="Species" type="xs:string" />
      <xs:element name="Nickname" type="xs:string" />
      <xs:element name="Cubs" type="xs:integer" minOccurs="0" />
      <xs:element name="ThreatLevel" type="xs:integer" />
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:choice>
 </xs:complexType>

 <xs:key name="MiningCampPrimaryKey">
   <xs:selector xpath=".//MiningCamp" />
   <xs:field xpath="PrimaryKey" />
 </xs:key>

 <xs:keyref name="MiningCampForeignKey" refer="MiningCampPrimaryKey">
   <xs:selector xpath=".//Bear" />
   <xs:field xpath="ForeignKey" />
 </xs:keyref>

 </xs:element>
</xs:schema>
```

Example 7.12 An XSD schema enforcing constraints with key and keyref



```
<xs:key name="MiningCampPrimaryKey">
  <xs:selector xpath=".//MiningCamp" />
  <xs:field xpath="PrimaryKey" />
</xs:key>
<xs:keyref name="MiningCampForeignKey"
                     refer="MiningCampPrimaryKey">
  <xs:selector xpath=".//Bear" />
  <xs:field xpath="ForeignKey" />
</xs:keyref>
```

Figure 7.19
A one-to-many relationship using key and keyref.

As with the one-to-many relationship, you can easily define a one-to-one relationship by creating a parent-child association between two element declarations within the schema. The one-to-one relationship can be enforced by setting the `maxOccurs` indicator to "1" on the child element declaration.

For one-to-one relationships where the child element will have multiple parents, you can use the `key` and `keyref` elements as you would in a one-to-many relationship.

SUMMARY OF KEY POINTS

- XML provides natural (intrinsic) one-to-many and one-to-one relationships through its hierarchical nesting structure.
- DTDs can supply basic data mapping functionality by simulating primary keys, foreign keys, and various relationships.
- XSD schemas are equipped with more sophisticated features designed specifically for relational data mapping requirements.

7.5 Database extensions

Chances are, one of the first places you'll look for XML integration features is to your existing database. IBM, Microsoft, Oracle, and just about every other major database vendor are providing some level of XML support. Very few, however, have made any attempt to make these new features anything more than proprietary enhancements that further tie you to their platform.

This is not to say that your database's XML features are not useful or should not be used. It is just important to integrate proprietary extensions with an awareness of how they may limit you in the future.

Following are some common ways in which database products extend their data access to XML.

7.5.1 Proprietary extensions to SQL

Some database vendors simply add XML-specific commands to their version of SQL. An SQL query, for instance, can be formulated with an extra parameter indicating that the query output should be an XML document, instead of the traditional result-set format.

Embedding proprietary SQL statements into your application code will likely make your application no less independent than it was before. However, you will be missing out on one of the fundamental benefits of an XML-enabled environment: an abstraction of data access. Keeping the XML aspect of your application architecture generic establishes a foundation for many interoperability opportunities.

Encapsulating proprietary extensions in a separate component layer or within a Web service can protect your application from becoming too dependent on the database platform. (Chapter 9 is dedicated to exploring the use of Web services within numerous legacy architecture models.)

7.5.2 Proprietary versions of XML specifications

Database vendors that create proprietary versions of XML specifications tend to be pretty up-front about it. Even to the extent of creating new acronyms for their unique implementations. Building on vendor-driven standards may be a better solution than providing non-XML-based extensions altogether, but it will still tie you to a product platform.

7.5.3 Proprietary XML-to-database mapping

Some databases provide data mapping facilities that allow you to associate XML documents to existing relational data models. Data maps can be generated through either a front-end tool or a programmatic API.

Regardless of how they are accessed, proprietary data maps will restrict you to the interface of the tool or API, which can often be quite limited. Additionally, a common problem here is that the map itself is stored in a proprietary format. This creates further dependencies between your architecture and a single database platform.

It is much more desirable for a mapping tool to generate its output into XML standard syntax, such as XSD schema and XSLT files. This will allow you to migrate the data maps to another product, and also gives you the freedom of editing them yourself.

7.5.4 XML output format

One of the biggest complaints relating to XML support in current databases is the format and syntax of the XML markup generated by the database extensions. Quite often, the document structure will be based on the existing relational model, resulting in creatively awkward hierarchies. Also, carrying database column names forward to XML elements can lead to cryptic naming conventions. Add to that a slew of proprietary markup and annotated commands that some products also insert, and you may be hard-pressed to recognize what you requested as even being XML anymore.

Some database vendors mitigate this problem by giving the developer the option of supplying parameters to predetermine the naming of elements and the overall format of the requested XML document. For instance, you may be able to tell the database to output a query as an element-centric document, as opposed to one that is attribute-centric. This gives you more control, but it can also impose a great deal of runtime processing, directly proportional to the size and complexity of the document you are building.

The best way to assess whether a database's XML output will do more harm than good, is simply to execute a range of commands and study the markup that gets returned. Keep in mind that databases are performing this translation at runtime, so if the output is only marginally useful, it may not be worth the processing cycles it is consuming. You may very well be better off writing a custom routine to create exactly the output you want.

7.5.5 Stored procedures

If you read the technical documentation carefully enough, you might notice that a significant amount of a database's XML support may be occurring through the use of stored procedures. Database vendors simply have added a set of system stored procedures to perform the runtime manipulation of data between XML and the native data format.

From the vendor's perspective, this approach makes a lot of sense. They are simply building on their existing platform, and by adding features with new stored procedures, they are not required to make major changes to their existing database software.

Again, though, this design may have implications in terms of your architecture's dependence on a vendor-specific technology.

7.5.6 Importing and exporting XML documents

Most XML extensions provided by database vendors focus on the translation of XML to and from existing relational data. Some provide utilities for importing and exporting XML documents as a whole. There is less emphasis on this aspect, as it is moving the database into the realm of content management, an uncomfortable place for many relational databases.

Regardless of whether the database product actually provides extensions to explicitly store and retrieve XML documents, you can always alter the data model yourself to add a character or LOB (Large Object) typed column that can contain the document text.

The key issue here is that, though most databases provide full-text searching capabilities, very few actually support XML-aware querying. XML tags are considered part the data, and will therefore be included in the results of full-text searches. Traditional relational repositories can adequately store XML documents only for retrieval, as long as the querying of the document is performed by the application.

7.5.7 Encapsulating proprietary database extensions within Web services

All the issues raised in the previous section build a case for avoiding proprietary database extensions altogether. Instead, it supports the idea of building your own interface to repositories, and using only those extensions that allow for a loose coupling between application and data source.

This is an area where Web services fits in nicely. A service-oriented architecture can introduce an interoperability layer that achieves platform independence and mobility by encapsulating any code required to interact with proprietary database extensions. (Read Chapters 6 and 9 to learn more about how Web services can facilitate data abstraction.)

SUMMARY OF KEY POINTS

- Extensions to your existing database that provide XML support are tempting, because they provide a convenient way to get a limited amount of XML output from existing relational data.
- Since most extensions are highly proprietary, they will further tie your application to a specific database platform, potentially nullifying the mobility benefits of an XML architecture.
- Web Services can provide a suitable abstraction layer, by encapsulating application code to interact with proprietary extensions. The result is a data platform-independent application core.

7.6 Native XML databases

Even in XML-centric environments, many organizations continue to rely exclusively on the well-established relational database platforms that have seen them through a number of changes in architecture and development technology. When moving your applications to XML-compliant and service-oriented architectures, you will find the relational data model to still be very much a part of your core data access technologies.

There is, however, a place for native XML databases. An understanding of what these products can offer will allow you to place them strategically within your environment. This can lead to a number of improvements, foremost of which are performance and protection of data integrity. Next is an exploration of how and where native XML databases can be utilized.

7.6.1 Storage of document-centric data

This is where native XML databases can immediately impact an organization. If you've standardized a body of documents using XML, you will need a storage and retrieval system that can handle the unique characteristics of the XML data format. In fact, a number of content management products that use XML as the underlying document format, also utilize native XML databases for storage.

Native XML databases are designed to accommodate and properly manage an XML document structure independently from its content. This is an area where relational databases often fail. Being able to differentiate the actual data from markup (that can include processing instructions and entity references) is beyond the ability of typical relational database platforms. For data-centric XML documents, however, relational databases that have been extended with XML support are still the way to go.

7.6.2 Integrated XML schema models

In some of the architectures explored earlier in this chapter, we placed schema files in relational repositories as entire documents or construct fragments. This is an extremely loose form of integration. The database is not aware of the schema content, and therefore views these schema models as any other piece of textual data.

Some relational databases do provide conversion features and other extensions that support a level of DTD or XSD schema integration. Few, however, come close to the depth at which a native XML database represents and manages XML schema models. Not only are XML schemas used to validate the integrity of data, the native database can actually build an index around the schema structure itself.

7.6.3 Queries and data retrieval

Here's where some analysis can result in significant performance improvements. Native XML databases index content differently from their relational counterparts. This relates back to their respective data structures: the tree/node hierarchy versus open, tabular data entities. When working with documents, as opposed to pieces of data, queries typically will result in larger amounts of data being requested.

XML-aware indices can provide a faster data retrieval mechanism when a large amount of document data is requested. The parsing of large XML data constructs is faster than the equivalent processing required when retrieving and then assembling this information from a relational data source. This is another reason document-centric content is more suitable for native XML environments.

Additionally, the XML-aware nature of native XML databases supports query technologies designed specifically for the XML representation format. This opens the door to sophisticated query statements that would not be possible with many of the XML-enabled relational database platforms.

7.6.4 Native XML databases for intermediary storage

One popular use for native XML repositories is to supplement application environments that already rely on relational databases. A common challenge with XML-enabled applications is the constant conversion between relational data structures and XML document formats. In previous chapters we explored some strategies for mitigating the performance overhead this runtime conversion process can impose, including the use of in-memory databases for caching purposes.

If, however, you need to provide a permanent storage facility for cached, non-durable, and document-centric XML data, then what better place than a repository specifically designed to store XML in its native format. Figure 7.20 illustrates this architecture.

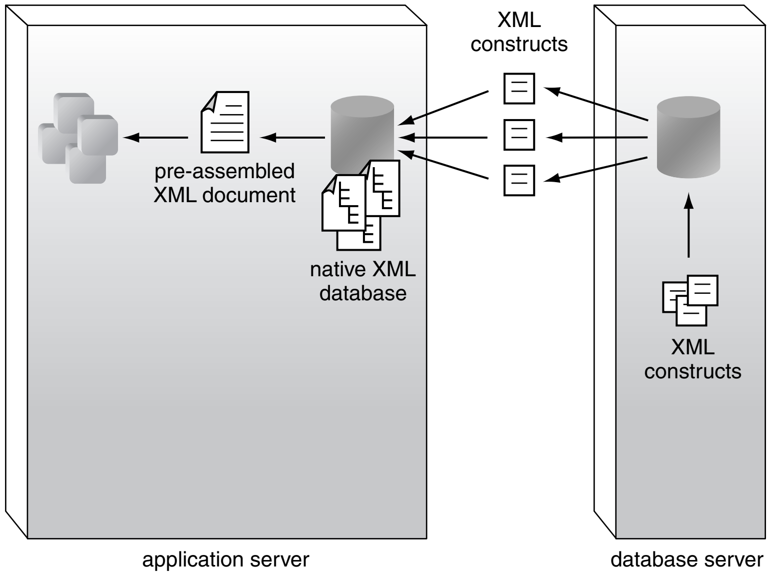


Figure 7.20 A native XML database acting as a physical cache for non-durable XML document data.

There may be situations where this architecture may even be useful for data-centric XML documents. It really comes down to what you're trading off. If you can take advantage of the integrated XML schemas, then this database can act as a pre-validator for documents that remain fairly static throughout the lifetime of the currently executing business task.

Also, if you need to cache XML formatted data for extended periods, the additional administration features offered by native XML databases may be more attractive. Finally, placing data in a native XML cache repository can open it up to a wide variety of data access opportunities that may not exist while the data is residing in a relational database.

SUMMARY OF KEY POINTS

- Native XML databases are most suitable for the storage of document-centric XML data.
- The XML-aware indices provided by native XML databases can provide faster data retrieval for large amounts of document data.
- Native XML databases can be positioned strategically alongside relational repositories.

About the Author

Thomas Erl is an independent consultant with XMLTC Consulting in Vancouver, Canada. His previous book, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, became the top-selling book of 2004 in both Web Services and SOA categories. This guide addresses numerous integration issues and provides strategies and best practices for transitioning toward SOA.

Thomas is a member of OASIS and is active in related research efforts, such as the XML & Web Services Integration Framework (XWIF). He is a speaker and instructor for private and public events and conferences, and has published numerous papers, including articles for the *Web Services Journal*, *WLDJ*, and *Application Development Trends*.

For more information, visit `http://www.thomaserl.com/technology/`.

About SOA Systems

SOA Systems Inc. is a consulting firm actively involved in the research and development of service-oriented architecture, service-orientation, XML, and Web services standards and technology. Through its research and enterprise solution projects SOA Systems has developed a recognized methodology for integrating and realizing service-oriented concepts, technology, and architecture.

For more information, visit `www.soasystems.com`.

One of the consulting services provided by SOA Systems is comprehensive SOA transition planning and the objective assessment of vendor technology products.

For more information, visit `www.soaplanning.com`.

The content in this book is the basis for a series of SOA seminars and workshops developed and offered by SOA Systems.

For more information, visit `www.soatraining.com`.