CHAPTER 8

XML Performance and Size

XML has a reputation for being big and slow.

- An XML document is often larger—sometimes *much* larger—than equivalent files in other formats, requiring more memory, disk storage, and network bandwidth.
- XML files need to go through complex parsing and transformations, which can take up considerable processing power and time.

An XML document can be larger in two ways: (1) in its proper XML form, requiring more storage space and bandwidth; and (2) in its compiled, in-memory form, requiring more computing resources.

Sometimes, XML deserves this reputation. Building a Document Object Model [DOM] tree, for example, or performing an XSLT [XSLT] transformation can use up a surprising amount of time and memory. Often, developers miss these problems while building a proof of concept but then run into them hard while building a production system with a full traffic load.

XML's very nature causes some of these problems—a plaintext format with frequent and repeated labels is bound to be a bit big and a bit slow—but XML's designers decided that it was worth trading some size and efficiency for the advantages of a portable, transparent information format. This has been a winning tradeoff in the past: Most of the popular Internet formats, such as SMTP, FTP, HTTP, POP3, and TELNET, also use plaintext, and their transparency and simplicity caused them to win out over more optimized but less transparent competitors.

On the other hand, many of the worst performance problems people run into with XML are a result of the tools and libraries they choose and the way they use them. Toolkits often hide large size and performance costs behind a simple interface: One or two function calls can trigger an exponential time and space explosion behind the scenes. When this happens, developers do not have to give up on XML, but they sometimes have to abandon their toolkits and do more work by hand. This chapter introduces some of the tips and tricks to work around XML's imperfections in a high-performance environment.

8.1 Advantages of XML for Size and Performance

Because XML markup is text, some developers assume that it will always be more verbose and slower to process than binary format. In fact, an optimized binary format can be efficient when the data structure is highly consistent, but when structure varies—for example, with optional or

repeatable fields or hierarchical relationships—a binary format can end up as large as an XML file, sometimes even larger.

Likewise, people assume that parsing XML will always be slower than parsing a binary format, but in practice, tool support sometimes cancels out that different: Because XML is so widely used, many free-software developers and commercial vendors have put a lot of time into profiling and optimizing the programs that do low-level XML parsing.

8.1.1 Space Efficiency

When data is in a highly consistent, predictable format, such as a table (see Section 4.3.1), it is possible to build efficient binary formats for storing it. For example, a binary file containing a table of 1,000 rows of 8 short integers will require only 16 octets for each row, or 16,000 octets in total, or even less with some compression schemes. Compare that with a row with typical XML markup in Listing 8-1.

Listing 8-1 A Row of Integers in XML

```
<row>
  <num>18</num>
  <num>11</num>
  <num>64</num>
  <num>23</num>
  <num>5</num>
  <num>65</num>
  <num>2</num>
  <num>10</num>
</row>
```

Even using fairly succinct XML names, such as `row` and `num`, the XML example requires 130 octets for each row, or eight times as much storage space as the binary format.

The binary format is space efficient because it can make assumptions about the data: There is no need to label rows or items, because a new item will start every 2 octets, and a new row will start every 16 octets. But what happens if the data is not *quite* so regular? The binary format will have to start adding extra information.

- If the rows have variable lengths—that is, items can be omitted from the end—the binary file will have to store the length of each row, adding an overhead of 1 or 2 octets for each row.
- If individual items have variable lengths, such as textual data, the binary file will have to store the length of each item, adding an overhead of 1 or 2 octets for each item.
- If items can be omitted or repeated in the middle of rows, the binary file will have to label each item so that it will be clear what has been repeated or omitted, adding an overhead of typically 4 octets for each item, assuming pointers into a name table.
- If the data is not simple rows of items but can have a more complex structure of nodes, the binary file may have to maintain navigational points, such as parent, first child, and next sibling, adding typically 4 octets for each node.

- If leaf and branch nodes can be mixed at the same level, as in XML or HTML mixed content, nodes will require type information, adding typically 1 octet for each node.

Accordingly, Table 8.1 shows that the memory requirement for item, or node, is now 17 octets, excluding the overhead of keeping one unique copy of each element name in a lookup table. By comparison, an item as an XML element can take as few as 4 octets `<a/>`, depending on the name length and encoding.

Even when both the start and end tags appear and the name is longer, such as `table`, the overhead for the start and end tags will be only twice the name length plus 5 octets in UTF-8 encoding, or, in this case, 15 octets, still shorter than the overhead in the binary format. Furthermore, the binary format does not yet have any mechanism for representing the equivalent of XML attributes, which would add yet more pointers and other overhead to it.

In the end, then, it is not the fact that XML is a text-based format that makes it verbose; rather, it is the fact that XML can encode very complex structure. When that extra structure is not necessary, an XML document can also be concise:

```
<r>1 2 3 4 5 6 7 8</r>
```

The 22 octets required for that data row in XML compare favorably to the 16 octets required for the data row in binary format. In fact, if the individual numbers were 4-octet integers, the efficient binary encoding would require 32 octets, whereas this particular XML row would still require only 22 octets. It is possible to tune a binary format to use a little less space, say, by stemming, but it is important to recognize that XML itself, text-based as it is, does provide a relatively efficient way to represent complex structures.

8.1.2 Software and Hardware Support

The text-based protocols used on the Internet—SMTP for sending e-mail, HTTP for retrieving Web resources, FTP for downloading files, and TELNET for connecting to remote machines—are all text based, like XML, and they have suffered from the same complaints that XML faces. Because they are text, they require an initial parsing step that slows down processing.

Table 8-1 A Binary Node

Property	Length (octets)
Type	1
Name pointer	4
Parent pointer	4
Next-sibling pointer	4
First-child pointer	4
Total	*17*

However, as the Web grew in popularity, individual developers and networking companies started to make software and hardware especially designed to work with these higher-level protocols. For example, many, if not most routers, can read and understand HTTP as well as the lower-level TCP and IP protocols and can make more intelligent routing choices as a result. Hardware acceleration is available for creating and managing HTTP, and networking libraries are efficient and well debugged. It does not matter so much that HTTP adds a little parsing overhead, because modern software and hardware support more than cancels out that disadvantage.

The same process is starting to take place with XML. Although higher-level tools, such as XSLT engines, remain slow, the low-level tools, especially XML parsers, have become fast and robust. Someone parsing an XML file is taking advantage of thousands of hours of experimentation, debugging, profiling, and optimizing that highly competitive XML parser providers—both vendors and free-software developers—have put into their products. Compared to custom-designed code to read a binary format, an XML parser is less likely to crash or fall into performance traps, such as unintentional buffer copying, and more likely to run fast and efficiently.

Furthermore, like HTTP, XML is starting to get hardware support. Several vendors, such as DataPower, Sarvega, and Reactivity, are releasing products for low-level XML parsing and, sometimes, for such higher-level operations as XSLT. This hardware still needs to be proven in the field and the market, but it suggests that the processing inefficiencies particular to XML will matter even less in the future than they do now.

8.2 Disadvantages of XML for Size and Performance

Despite the advantages mentioned in Section 8.1, XML does sometimes cause a significant increase in data size and processing time. These disadvantages are the result of design decisions and tradeoffs made by XML's original designers. For example, to make XML fully internationalized, the designers chose to require Unicode support, which can increase the memory required for processing and storing information from XML documents. The designers also chose the robustness of redundant labels in start and end tags, increasing the amount of space XML requires in disk storage or the amount of bandwidth for moving it over a network. The most serious performance risk, however, is one that people do not often worry about: XML's ability to include external resources.

8.2.1 Repetition

XML repeats every element and attribute name for every element and attribute instance: In fact, it repeats the element name *twice* for every instance. If a long XML document contains 20,000 nonempty elements named `maintenance-entry`, the string `maintenance-entry` will appear in the document 40,000 times, consuming between 680,000 and 2,720,000 bytes of storage space, depending on the character encoding.

For loosely structured XML, such as human-readable documents (see Chapter 3), this overhead is often not a problem, but for highly structured XML, such as a database dump, these repeated names represent a significant overhead. There is a temptation to use short, cryptic element

and attribute names, such as `c183`, instead of `workflow-approval`, destroying XML's advantage of transparency. There is also a temptation to reduce the amount of tagging, using whitespace and line ends to delimit some fields. These solutions are not particularly good, but they do show the desperation people face when dealing with enormous XML data files.

8.2.2 Encoding

Sometimes, text can be more efficient than binary representations: For example, the long integer "1" requires 1 byte to represent in text using UTF-8 text encoding but 4 bytes to represent in a typical binary encoding. More often, however, the XML textual representation is longer: For example, the short integer "15,383" requires between 5 and 20 bytes in text, depending on character encoding, but only 2 bytes in binary form.

In fact, character encoding itself can cause an enormous size increase for XML documents, both in memory and on disk. The Unicode UCS-4 encoding, which, fortunately, almost no one uses, requires 4 octets for each character, so 100,000 characters become 400,000 bytes of storage. UTF-16, which is more common, requires 2 bytes for most characters. UTF-8 requires only 1 byte for ASCII characters but as many as 6 bytes for some Asian characters.

8.2.3 External References

The biggest performance risk for XML comes not from the fact that it is text based, that it is parsed, or that it can use Unicode but from the fact that XML documents can include external files. To make things worse, the inclusion can take place in the lowest-level XML parsing layer, where it is completely hidden from—and sometimes outside the control of—the application developer. For example, consider Listing 8-2.

Listing 8-2 Referencing an External DTD

```
<!DOCTYPE doc SYSTEM "http://www.example.org/dtds/doc.dtd">

<doc>
...
</doc>
```

By default, almost all validating XML parsers will go to `www.example.org` and download `doc.dtd` every time they parse this document, leading to some serious performance problems.

- Even with a fast network connection, each document will likely require seconds rather than milliseconds to parse.
- If `www.example.org` is slow, possibly because of a heavy network load, parsing will slow down even further, possibly on the order of minutes for each document.
- If `www.example.org` goes offline, parsing will fail completely.
- If `www.example.org` has a security breach, an intruder could modify the DTD to cause denial of service or include false information in XML documents referencing it.

External DTD subsets are the greatest danger, but they are not the only way XML documents can cause files to be downloaded automatically during processing: The documents can also use external parameter entities in the DTD subset and external text entities in the document itself. Some XML parsers will also automatically download schemas, such as XML Schema [XMLSCHEMA] or RelaxNG [RELAXNG], referenced from inside a document.

Organizations can work around this problem by always parsing inside a sandbox that prohibits or limits external network access, providing local copies of required files, such as schemas. Many developers do not consider this step at first, however, and when looking over HTTP server logs, it is not uncommon to see some sites hitting the same online DTD file hundreds or thousands of times a day, almost certainly because of automatic downloading by XML parsers.

8.3 Processing Performance

One of XML's great strengths is the enormous selection of libraries, applications, toolkits, and frameworks available to developers in nearly every programming language and on nearly every platform. Sometimes, especially in the case of such low-level components as parsers, this software has been heavily optimized for size or performance. In other cases, however, publicly available XML software is optimized instead for ease of use; when that happens, the software can hide design choices that trigger enormous size or performance hits while processing XML.

When a high-level component, such as an XSLT engine, becomes an efficiency bottleneck, developers need to understand how their tools work and where the costs come from. This section describes some of the optimizations available to developers to improve XML performance for parsing, querying, and transforming XML and then introduces the technique of XML pipelines.

8.3.1 Parsing Interfaces

Although people have come up with some innovative ideas for XML parsing interfaces, such as pull parsers and query-based parsers, nearly all XML parsing interfaces in common use fall into one of two groups:

1. Tree-based interfaces, such as the Document Object Model [DOM]
2. Event-based interfaces, such as the Simple API for XML [SAX]

DOM-like interfaces allow applications to navigate and modify an XML document in much the same way that they navigate and modify a file system, by moving up and down various branches. As a matter of fact, some XML user interfaces even present XML documents using file system iconography, with folder icons for elements and file icons for text.

Unfortunately, the ease of use of DOM-like interfaces comes at a high price: An XML document's entire structure must be available for random access. In practice, that means that the structure must be in a database or, most commonly, in computer memory. During batch processing, building that in-memory data structure requires a time for allocating objects[1] and memory for

[1] Tree-based interfaces often require several objects for each markup component.

keeping them, typically five to ten times the size of the original XML document. For a desktop application, these requirements are not usually a problem: A user editing a 2MB XML document will not mind waiting a few seconds for the document to load and can easily spare 20MB of memory to edit it. However, on a system handling hundreds or thousands of XML documents simultaneously and quickly, such as a server or high-speed information system—or even worse, a network appliance—the memory and time overhead are entirely unacceptable.

In these cases, switching to an event-based interface, such as SAX, may be a good choice. SAX-like interfaces do not allow random access to an XML document—although a DOM-like interface is similar to a hard drive, a SAX-like interface is more similar to a tape drive—but the SAX-like interface uses near-constant memory no matter how large an XML document may be[2] and tends to use very little processing time, as it does not need to allocate new objects during its run.

Unfortunately, programming with a SAX-like parsing interface is considerably more complicated than programming with a DOM-like interface. The application needs to capture and store any information it requires as the information streams past, and there is no way to follow backward references in a document. When ease of implementation is the priority, a DOM-like interface makes sense; when performance is the priority, a SAX-like interface makes sense.

Often, applications parsing data-oriented XML documents build their own, in-memory data trees.

- An accounting application might build a tree of ledgers, accounts, and entries.
- A geodata application might build a tree of polygons and points.
- A graphics program might build a tree of shapes and lines.

An application that builds a specialized data tree using a DOM-like XML parsing interface takes a double hit: First, the parser uses a lot of time and memory to build a parse tree of the XML document, and then the application uses more time and memory to build its own tree before discarding the parser's tree. In such a case, simply switching to an event-based interface can reduce an application's time and memory requirements for XML processing to a fraction of what they originally were.

Another place that tree-based interfaces often hide performance is in transformations, discussed in Section 8.3.3.

8.3.2 Queries

Sometimes, an application reads an XML document simply to extract a small bit of information hidden somewhere inside it. A bibliographical application might need to extract only the title and the author's name from an XML-encoded research paper, like the one in Listing 8-3.

[2] Memory requirements vary with maximum element nesting depth, not document length: Typically, even long XML documents have elements nested only a few levels deep.

Listing 8-3 Bibliographical Information in a Research Paper

```
<article>
  <articleinfo>
   <title>Frog populations in Frontenac County</title>
   <author>
     <honorific>Dr.</honorific>
     <firstname>Jose</firstname>
     <surname>Schmidt</surname>
     <affiliation>
       <orgdiv>Department of Biology</orgdiv>
       <orgname>King's University</orgname>
     </affiliation>
   </author>
 </articleinfo>

 <sect1>
 <title>Introduction</title>

 <para>During the summers of 1999 and 2000, a team of researchers
   [...]</para>

 [...]

 </sect1>

 [...]

</article>
```

If the paper were 100 pages long, building a DOM tree of the entire XML document simply to extract the title and author information would be extremely wasteful. It is much better to write a short, perhaps 50-line, program in Perl, Java, Python, C, or C++ to extract the required information from an event stream provided by a SAX-like interface. In fact, the application can terminate parsing as soon as it has all the required information, so the parser will not need to read most of the document at all, much less build it into an in-memory tree.

Unfortunately, custom programming is not always a practical solution: Many people need to extract information from XML documents in a more generic way that does not require hard-core computer programming skills. For situations like these, the XML community has largely settled on XPath [XPATH] as a simple query syntax for XML documents.[3] For example, the following

[3] At the time of writing, the XQuery [XQUERY] specification was not yet complete. XQuery builds on top of XPath, making it into a proper, full-featured query language. It is not yet clear whether XQuery will gain the kind of widespread acceptance that XPath has.

XPath expressions select the element containing the article title and the subtree containing the author's name:

```
/article/articleinfo/title
/article/articleinfo/author
```

A general-purpose query application can use these expressions to return matching fragments from an XML document, the same way that the UNIX `grep` command uses regular expressions to return matching fragments from a text file. An interactive XML editor or browser can use these expressions to bring the cursor to the matching points in a document.

Unfortunately, it is necessary to have random access to the original XML document to support all of XPath, and that, once again, requires a DOM-like interface. In a high-speed environment, in which predictable, consistent performance is important, it may be necessary to restrict queries to the subset of XPath that *can* be supported by a streaming interface. Both of the previous XPath expressions, for example, could be resolved by an application using a SAX-like parsing interface, as they do not require any backward references.

The XPath 1.0 expressions that can be matched most efficiently—for time and memory—are the ones that limit themselves mostly to context on the element stack, particularly parent, child, ancestor, and descendant relationships. For example, all the following XPath expressions could be matched against output from an event-based XML parser, such as a SAX parser, storing no context except for a stack of elements and their attributes up to the root element and the index of each element in the stack relative to its siblings:

```
//caution
/book//chapter/title[contains(string(), "Crimson")]
//section[@id="preface"]//character[string()="Joseph"]
```

- The first XPath expression simply requires matching the `caution` element, with no additional context.
- The second expression requires matching the current element against `title` and examining the element stack to ensure that the parent element is `chapter` and that the root element is `book`, then testing whether the string `"Crimson"` appears in the element's content.
- The third expression requires matching the current element against `character`, testing whether that `section` element with the `id` attribute equal to `"preface"` appears anywhere in the element stack, and testing that the element's content is exactly the string `Joseph`.

These are all queries that will use near-constant memory no matter how long an XML document is, as none requires random access. The following inefficient expressions, on the other hand, do require random access to a document or else the caching of a large part of a document's contents:

```
//caution[../para[contains(string(), "oil")]]
/book//chapter[following-sibling:chapter/title[string()="Crimson"]]
//link[id(@ref)/@status="modified"]
```

- The first XPath expression matches any `caution` element that has a sibling named `para` containing `oil` in its content. As a result, a query engine will have to save the content of every `caution` element until the parser has finished reporting all its siblings.
- The second expression matches any chapter that appears before a chapter with the title `"Crimson"`. As a result, the query engine will have to cache the content of all chapters until the parser is finished processing the parent element.
- The third expression is the most inefficient: In the worst case, it will require the query engine to cache every link element until the end of the document.

Fortunately, the efficiency concerns for XPath expressions are not so serious, because many technical specialists, not to mention ordinary users, will find such expressions anywhere from excessively complicated to baffling. Limiting a high-performance query engine to efficient expressions still provides a respectable amount of query capability while also allowing an application all the advantages of a SAX-like interface (see Section 8.3.1): Modifying an application to use only this subset, together with a SAX-like parser, can bring significant speed and memory improvements to an XML application.

8.3.3 Transformations

Performance problems with transformations are closely related to the problems with parsing interfaces (Section 8.3.1) and queries (Section 8.3.2). The most popular specification for transforming XML documents into XML or another format is XSL Transformations [XSLT], but because XSLT supports the full XPath specification, XSLT also requires random access to the original XML document. Therefore, XSLT engines typically use a DOM-like interface with all the extra time and memory overhead.

XSLT's expressive power, and the fact that it is a template language, make it ideal for complex structural transformations; in those cases, extra time and memory might be a reasonable tradeoff for rapid development. On the other hand, many transformations are small and do not require the ability to restructure a document extensively. In these cases, the significant overhead of using XSLT processors other tree-based transformation tools is unacceptable.

Before considering alternatives, this section takes a quick look at the efficiency of XSLT engines. XSLT engines read an XML document as input and create a new document, possibly XML, as output. Although the engines require random access to the input document, they have no such requirement for output, so there is no need to build a *second* in-memory document tree. Command line XSLT utilities are generally well-enough designed that they do not build a second DOM tree, or the equivalent, but simply write out the result as a stream, However, if an XML project happens to use an XSLT library, a developer could end up using the library to generate a DOM tree, for XML, before writing any output. This is, obviously, a bad idea when memory is at

a premium, and fixing this problem is an easy way to halve the memory consumption of any project that uses XSLT.

A second optimization is possible with XSLT, but to date, no one seems to have taken advantage of it. XSLT requires random access to the input document but does not modify the original document; as a result, it is possible to parallelize XSLT processing, assigning different processors to perform different parts of the transformations on the input document.[4] This approach will not solve the memory problem but could bring significant speed improvements for large input documents or complex transformation rules.

As mentioned earlier, however, XSLT's ability to access the input document randomly using XPath expressions is not necessary for many kinds of transformations, and an application working with an event stream, such as SAX, can run much faster, using very little memory. SAX-like transformations are complicated or impossible when the transformation requires major structural changes, but such transformation works well in three cases:

1. Adding information to a document
2. Removing information to a document
3. Modifying information in place, such as renaming elements and attributes or changing text

Many typical transformations fall into these categories. For example, one common transformation for technical and sales publications is adding live data from a database at publication time. The original XML markup might look like that in Listing 8-4.

Listing 8-4 Catalog Entry before Transformation

```
<item>
 <source>Henrickson</source>
 <name>Acoustic Guitar</name>
 <description>This guitar includes a mahogany fret board, gold-plated
  tuning pegs, and a solid top.</description>
 <price dbref="g3905778/price"/>
</item>
```

The transformation engine executes a database query based on the value of the `dbref` attribute in the `price` element and replaces it with the latest price information, as shown in Listing 8-5.

Listing 8-5 Catalog Entry after Transformation

```
<item>
 <source>Henrickson</source>
 <name>Acoustic Guitar</name>
```

[4] Thanks to Ken Holman of Crane Softwrights for pointing this out to me.

```
  <description>This guitar includes a mahogany fret board, gold-plated
   tuning pegs, and a solid top.</description>
  <price>
   <status>sale</status>
   <expiry-date>2005-01-01</expiry-date>
   <currency>USD</currency>
   <amount>1999.00</amount>
  </price>
 </item>
```

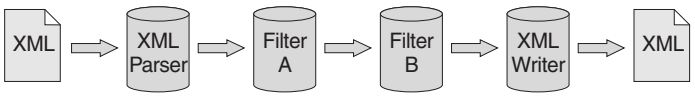
A tree-based transformation engine, such as XSLT, brings little advantage to this kind of work. In fact, because XSLT does not specify a standard method for database access, any XSLT template would not be portable anyway, even if the engine did support database access. On the other hand, using Perl, Python, or Java, a transformation like this is trivial for an experienced developer.

8.3.4 Pipelines

Another performance problem that can creep into an XML processing system is duplicated parsing. An XML document will often pass through several components in a chain from the point it enters a system to the point it leaves; if the XML is written to disk and then reparsed into memory each time a component touches it, a system might slow down.[5] The solution to this problem is to use a processing pipeline like the one in Figure 8-1.

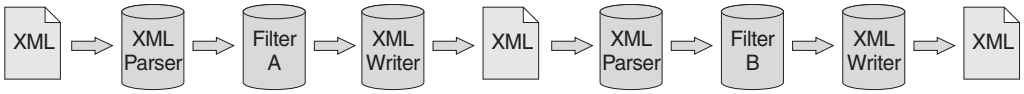
The XML document enters the pipeline through the parser on the left, which converts the document to a series of SAX or SAX-like events. The parser then passes the events on to the first component, which manipulates them as necessary—say, by adding or removing information—then passes the modified events on to another component. The second component knows nothing about the first component: As far as it is concerned, the events could be coming directly from a parser. The second component makes further changes to the event stream and then passes the modified events on to the third component, and so on. At the end of the chain is a component that simply writes the events back out to disk or sends them out over the network in XML format. The alternative to this pipeline would be to parse and rewrite the XML document for each processing component, as in Figure 8-2: Note all the extra work required for processing.

Figure 8-1 XML processing in a pipeline



[5] Notwithstanding this problem, XML parsing is normally very fast: Speed problems typically come from higher-level processes, such as tree building, querying, and transformation, not from the low-level parsing. The fastest software-based XML parsers can read thousands of short documents every second, even on a low-end desktop computer.

Figure 8-2 XML processing without a pipeline



The pipeline concept works with both DOM-like interfaces and SAX-like interfaces. For DOM-like interfaces, the components all work on the same in-memory tree in sequence, making changes and then passing the tree on to the next component. The beginning of the pipeline is also a parser—and tree build, in this case—and the last component is also a writer.

Aside from the fact that, for efficiency, the components need to run on the same system, this approach has one problem: Without XML to examine between each component, it can be difficult to find and fix problems. Fortunately, however, the pipeline components are modular, so it is easy to fit in extra debugging components to write out the events as XML at various points in the processing.

The pipeline model is the most common one used by skilled XML developers working with SAX and other event-based interfaces, and it has proved itself in high-speed, high-demand environments. The model is not quite as common with the DOM and other tree-based interfaces, as people tend not to use those in high-speed environments in the first place, but it can work equally well.

8.4 Size

This section describes some ways to work around XML's size problems, both with XML markup and with internal representations of parsed XML documents.

8.4.1 Unicode and Character Size

XML documents can use many different character encodings, and those affect the size of a document on disk, through the network, and in memory. Choosing the right character encoding for a project can be the cheapest and easiest kind of compression.

Because the W3C designed XML for international use, all XML parsers are *required* to support the standard Unicode encodings UTF-8 and UTF-16. Many XML parsers also accept other popular encodings, such as ASCII, ISO-8859-1—the ISO Latin 1 alphabet, sometimes informally called *8-bit ASCII*—and Shift-JIS. Other than the Unicode UTF and UCS encodings, no character encodings support the entire Unicode character set: For example, ISO-8859-1 does not support Japanese characters, and Shift-JIS does not support accented European characters. In XML document instances, however, it is possible to include characters outside the current character set by using character references; the following example includes the Russian word *мир* in an ASCII-encoded XML document:

```
<?xml version="1.0" encoding="US-ASCII"?>

<trivia>The Russian word <q>&#x43c;&#x438;&#x440;</q>
means both <q>world</q> and <q>peace</q>.</trivia>
```

If space and bandwidth are at a premium, then, it is possible to save some space by choosing the right XML character encoding.

- With UTF-16, all characters—at least, all the ones you're likely to use—require 2 octets, so the phrase "Hello, world!" will use 26 octets to store 13 characters.
- With UTF-8, basic ASCII characters require only 1 octet, so "Hello, world!" would use only 13 octets to store 13 characters; however, Cyrillic or accented European characters require 2 octets each, and other characters for other languages can require up to 6 octets.
- With the ISO-8859 character encodings, each character requires only 1 octet, even accented European characters, but any characters not in the set require XML character references, such as `&#x43c;`.

Of course, many other character encodings are available, but the tradeoffs should be obvious. XML documents containing English-language documents and computer programming code will be smallest with UTF-8 and generally twice as large with UTF-16; non-English European languages will be smallest with one of the ISO-8859 alphabets, if available, and only slightly larger with UTF-8. Many other world languages, such as Chinese, will be smallest with UTF-16 or a specialized encoding and can grow considerably larger with UTF-8. In most cases, where storage space is not severely constrained and every millisecond of bandwidth doesn't count, choosing either UTF-8 or UTF-16 and sticking with it may be the best choice; if size is critical, choosing the right encoding for storage can bring considerable savings.

The second Unicode-related size issues come when the XML document has been parsed and read into the computer: Typically, parsers will convert the original XML document's characters into a standard internal encoding for processing, almost always UTF-8 or UTF-16, so that any Unicode characters can be included. The same size advantages and disadvantages for various languages apply here: English and European languages will generally use less space with UTF-8, whereas nonalphabetic languages, such as Chinese, will use less space with UTF-16. There are also processing considerations: UTF-8 works better with older libraries in some programming languages, such as C and C++, but string manipulation can be tricky because characters do not have a fixed width. UTF-16 gives all characters a fixed width and works well with modern programming languages, such as Java, but it can be difficult to use in older libraries and programming languages.

8.4.2 Internalization

Internalization is a useful trick from older programming languages, such as LISP. Because the same symbol, such as `list`, can appear frequently in a LISP program, implementers can save space if they put exactly one copy of each symbol in a lookup table and always point to that one location, rather than copying the symbol over and over again and wasting memory.

Many, if not most, XML parsers internalize element and attribute names in the same way, as a name may appear hundreds or thousands of times in an XML document. In Listing 8-6, the element name `list` appears twice, and the name `item` appears eight times, but the XML parser would allocate only a single instance of each in memory.

Listing 8-6 Repeated Names

```
<list>
  <item>100</item>
  <item>200</item>
  <item>300</item>
  <item>400</item>
</list>
```

In general, parsers do not internalize other strings, such as attribute values and character data, as they are much less likely to be repeated, and internalizing adds some processing overhead. However, an attribute value or character data content in XML documents is sometimes limited to a set of enumerated values: For example, the status of a task entry in a dictionary might be `"draft"`, `"pending"`, `"approved"`, or `"released"`. Because the value must be one of these and no other, an application can internalize the value of the element or attribute holding the information, as the values are very likely to repeat. Sometimes, XML schemas and DTDs can provide hints to XML processors about what values are enumerated, but applications should pay attention to internalization opportunities as well, as they can provide important memory savings.

8.4.3 Compression

If an XML document is going out over the network or being kept in long-term storage, brute-force compression is sometimes a much better choice than any of the other techniques in this section. XML is text with a lot of repetition, so it compresses surprisingly well. For example, at the point that I'm writing this paragraph, the XML manuscript for this book is 440,320 bytes long. Simply running the manuscript through the UNIX `bzip2 -9` command reduces its size to 98,815 bytes, or only 22 percent of the original. If you are willing to trade a bit of processing time for the sake of reducing storage or network bandwidth requirements, straightforward compression will save far more space than any of the other techniques in this section and will outperform almost any optimized binary format as well.

Unfortunately, brute-force compression is not always the answer. Compressed XML needs to uncompressed before it can be used in any way, and it is virtually useless for in-memory processing. In these cases, specialized compression techniques, such as internalization, can bring some savings, but they are much less dramatic.

8.5 Final Words on Performance and Size

For many XML projects, performance is not as important as people think. Parsing an XML document takes very little time, often a millisecond or less. Typically, an application will spend several orders of magnitude more time processing the information than parsing it, so even a 1,000 percent improvement in parsing time would speed up the typical application too little to matter.

Parsing time does start to matter for applications that need to run at I/O speed, however, and it matters greatly for network appliances that need to run at wire speed. These applications may need specialized solutions, such as XML parsers burned into silicon. Such components are

starting to appear, but it will take time before they have an opportunity to prove their performance and value in the field.

For everyone else, the most important optimizations for size and speed come not in XML parsing but in how applications work with the information once an XML parser has delivered it and what encodings they use to store it on disk. Few of the techniques introduced in this chapter will magically eliminate all performance and size problems; used together, however, they may make an application fast enough for today's requirements and, perhaps, even for tomorrow's.