developerWorks®

6 tips for optimizing a native XML database

Common sense guidelines for using XQuery with native XML databases

Donnie Cameron

December 15, 2009

RSS, Atom, mashups, extraordinary search requirements and other developments are making native XML databases an important part of search applications and services. These types of databases excel at efficiently searching through large collections of semi-structured data. In this article, you'll find some common sense guidelines to maximize the performance of applications that use XQuery and native XML databases.

XQuery and native XML databases

Frequently used acronyms

- RAM: Random-access memory
- RSS: Really Simple Syndication
- XML: Extensible Markup Language
- XSLT: Extensible Stylesheet Language Transformations

Using XQuery (a functional language designed to query collections of XML data) with native XML database systems can be extremely useful in some situations. When servicing queries that are complex and mostly read-only, compared to standard relational databases, native XML databases provide faster response times and faster development times. wit the simplest and most powerful data-transformation system available today built right into the query language, you gain faster development times because you don't need to design a separate full-text indexing system or assemble much of the data for the user.

At the cost of slower inserts and updates, native XML databases can provide vastly superior out-of-the-box response times because they keep their data largely denormalized, provide default indexes, and make excellent use of available RAM. However, when dealing with very large data sets, you can further improve the query response times of a native XML database by following a few general common sense guidelines:

- Avoid normalization
- Employ unique element names
- Precompute values
- Transform data with your queries

- Profile the XQuery code
- Keep a list of optimizations

These guidelines are general and applicable to many of the native XML databases that are available today, including IBM DB2® Express-C, Mark Logic Server, eXist, and even Oracle Berkeley DB XML (see Resources for links). Now let's look at the optimization guidelines in more detail.

Avoid normalization

The most important thing you can do when you design a native XML database schema is to avoid the temptation to normalize the data in the same way that you do when you design a relational database.

Normalizing data for a native XML database consists of designing multiple XML document types that link to each other in ways that are similar to the ways that relational-model tables link to each other. However, in most cases, you'll need to normalize little if any of the data for a native XML database. It's quite common to put data that would reside in tens of relational-model tables into a single XML document type.

Most implementations of XQuery that exist today perform joins so inefficiently that even a simple query involving a few thousand records can take an unacceptable amount of time to process. This makes the criteria for deciding if you should normalize data straightforward: Never normalize data such that a supported query will need to perform a join operation to select records.

A supported query is a query that you can reasonably expect the users to make of your data. For example, if you build an application to sell video tapes, you might expect a user to query for all the videos that have a certain keyword in the title and that were directed by a given person. Because of this, you'd definitely want the XML document that represents the video to contain the title of the video and the name of its director. On the other hand, for this particular application, you might not want to support a query for all the videos that have a certain keyword in the title and that were directed by a person born in New York. In other words, for the video application example, if you have detailed information about the director (beyond the director's name), its okay to consider keeping it in a separate XML document.

Picture a database with two linked XML document types, `video-rec` and `director-rec`, the former with detailed information about videos including a `director-rec` identifier, the latter with detailed information about directors. The query for a record with a keyword in the title and a director born in New York is a query that has to perform a join operation to select records. As mentioned, it might be okay not to support this type of query because it's more of a data-mining query, not really the type of query that most users browsing an online video store make. However, unless you have concrete reasons to move the director's details to a separate document type, you should keep it in the `video-rec` document.

Although in a native XML database it's never efficient to perform a join operation to select records, it's often perfectly okay to fetch data from multiple XML document types when transforming data from search results. The video store I've described can easily and efficiently present results that included the place of birth of the director, even though obtaining the location requires fetching data from a document that's not among the original search results. The operation necessary to assemble the results in this way is limited to the few records that the application has already selected and that it plans to display; it's computing and memory requirements are negligible compared to what's needed to join multiple document types in a general search query.

Employ unique element names

Unique elements are elements that always refer to the same element in an XML document. *Non-unique elements* are elements that can appear anywhere in the XML document and that have to be preceded by a path to be meaningful. For example, if an XML document contains 10 entirely different types of nodes and each type includes a date element as a descendant, then the date element is a non-unique element name. Employing non-unique element names can hinder your ability to evaluate or profile some XQuery or XPath alternatives for locating your data. For example, non-unique element names can keep you from properly evaluating code that performs fewer index look-ups. Also, non-unique element names can get in the way of emerging support for faceted search results.

The following sections provide examples of the kinds of optimizations that you can support by changing the design of a document like the one in Listing 1 so that it uses distinct element names.

Listing 1. Base document with non-unique element names

```
#+BEGIN_SRC nxml-mode
<class-info>
  <school>Lusher Elementary School</school>
  <grade>10</grade>
  <teachers>
    <teacher>
      <name>
        <first>Carol</first>
        <last>Osborne</last>
      </name>
    </teacher>
    <teacher>
      <name>
        <first>Dan</first>
        <last>Silver</last>
      </name>
    </teacher>
  </teachers>
  <students>
    <student>
      <name>
        <first>Barrie</first>
        <last>Stoff</last>
      </name>
    </student>
    <student>
      <name>
        <first>Andrew</first>
        <last>Silver</last>
      </name>
    </student>
    <student>
      <name>
        <first>Larry</first>
        <last>Cracchiolo</last>
      </name>
    </student>
    <student>
      <name>
        <first>Richard</first>
        <last>Hughes</last>
      </name>
    </student>
    <student>
      <name>
        <first>Bruce</first>
        <last>Silver</last>
      </name>
    </student>
    .
    .
    .
  </students>
</class-info>
#+END_SRC
```

Performing fewer index lookups

To get the first names of the students that have the last name Silver, you can use an XPath expression like the one in Listing 2.

Listing 2. XPath expression to find last name Silver

```
: /class-info/students/student/name[last = "Silver"]/first
```

If you limited the data to the visible data in the single document in Listing 1, then evaluating the XPath expression in Listing 2 always correctly returns the result in Listing 3.

Listing 3. The XPath result

```
<first>Andrew</first>
<first>Bruce</first>
```

If the data is not indexed, then Listing 2 is always the fastest way to get those results. The expression limits the number of branches that the database has to search to find the relevant elements.

If the data is indexed, however, then depending on the specific database implementation that you use, and provided that you have a very large data set, an expression like the one in Listing 4 might resolve consistently faster.

Listing 4. XPath expression to use if data is indexed

```
: //name[last = "Silver"]/first
```

The reason for the potential improvement is that the system has to look through fewer elements in the index. However, because of the design of the document in Listing 1, which uses non-unique element names, the XPath in Listing 4 returns incorrect results; the results include a teacher's name, Dan. The design precludes you from profiling queries that make use of fewer indexes. A better design is to replace the non-unique element names in Listing 1 with unique element names, as Listing 5 describes.

Listing 5. Replacing non-unique element names from Listing 1 with unique element names

```
   //teacher/name => //teacher/teacher-name
   //teacher/name/first => //teacher/teacher-name/teacher-first
   //teacher/name/last => //teach/teacher-name/teacher-last
   //student/name => //student/student-name
   //student/name/first => //student/student-name/student-first
   //student/name/last => //student/student-name/student-last
```

Supporting faceted search results

The goal of a faceted search is to display links that allow the user to quickly and intuitively narrow the search results along various axes. In an application that supports faceted search results, a query that lists all of the teachers in the database might return information like this in the user interface (see Listing 6).

Listing 6. Faceted search

```
•Tabor, Gavin
•Nance, Jamey
•Haas, Carlene
•Davies, Yesenia
•Singer, Lupe

Narrow your search:

•School

   •Lusher Elementary School (35)
   •Academy of the Sacred Heart (34)
   •Isidore Newman School (32)
   •Audubon Charter School (28)
   •Benjamin Franklin Elementary Math-Science Magnet (25)

•Grades

    •9 (5)
    •10 (6)
    •11 (6)
    •12 (6)
```

Listing 6 provides two facets: `School` and `Grades`. Each facet contains four or five values that link to a search that narrows the most recent search. Next to each facet value is a number, in parentheses, that indicates the total number of teachers that you'll end up with if you click the link. Faceted search results typically display only a few possible values for each facet. When the number of distinct values for a facet is small, as is the case for the `Grades` facet, the application typically displays all the facets in the order in which they make the most sense. However, when a facet includes many possible values, then the application often displays only the values that will return the most results and normally displays those values in descending order of number of results.

Some native XML databases are incorporating support for faceted searches, but they require special indexes to deliver the best performance. A typical XQuery algorithm for obtaining the display values for a facet becomes a bottleneck quickly as the number of records in the database increase and as the number of possible values for a facet increase. For a large database that has facets with thousands of values, such an algorithm just won't do. To deliver the power of faceted searches, native XML engines need to be able to build lexicons from the values that an element takes on in the database. These lexicons can be implemented from special indexes, which in turn can require unique element names.

If you have a relatively small native XML database that lacks support for faceted searches and you need to write your own code to support such functionality, you'll see how unique element names are just as essential to your code as they are to the faceted-search support code that's already in existence in the more advanced databases.

Precompute values

The thought of adding redundant data to an XML document is heresy to a seasoned relational database administrator. However, when your primary concern is performance—for example, when you have to return faceted search results to queries that run against tens of millions of records—precomputing some values based on data that exists in the XML document and then adding the results to the XML document can help improve response times dramatically. Native XML databases are all about sacrificing storage and tolerating redundance in exchange for performance.

Suppose that you have a bunch of image meta data XML documents. Every one of these XML documents has one or more of the elements `camera`, `device`, and `scanner`, which all hold information about the device that created the image. The `device` element represents a complex node that includes an element with the name of the device and several other elements with additional information. In this example, all of the `device` elements are needed in other parts of the application and thus cannot be discarded. The application implements faceted searches and calls for a facet named `scanning device` that shows the name of the device that created the image.

Similarly, the image meta data documents have height and width elements, but the application calls for a facet called `size`, which can easily be derived from the `height` and `width` elements.

Listing 7 is one example.

Listing 7. First image meta data document example

```
<image>
  <id>123456789</id>
  <date>2009-11-16T03:14:42</date>
  <description>Eiffel Tower</description>
  <device>
    <device-name>Scanmelter 2000</device-name>
    <device-resolution>300dpi</device-resolution>
    <device-manufacturer>Scanners Inc.</device-manufacturer>
    <service-tag>ASDFQWER</service-tag>
  </device>
  <width>1200</width>
  <height>1024</height>
</image>
```

Listing 8 shows a second example.

Listing 8. Second image meta data document example

```
<image>
  <id>123456788</id>
  <date>2009-11-16T03:14:42</date>
  <description>Empire State Building</description>
  <scanner>Pixel Maker LS</scanner>
  <width>800</width>
  <height>600</height>
</image>
```

Now imagine a database with enough of these records that a query for images taken on 2009-11-16 returns 5,000 images. Of these, the application displays 30. The search-results view displays various facets, including `scanning device` and `size`, with each facet providing a short list of values. The `scanning device` facet values include `Scanmelter 2000 (1202)` and `Pixel Maker LS (207)`. The `size` facet values include `1200x1024 (2302)` and `800x600 (113)`.

Think about the code that you might write to satisfy these requirements. The code is fairly easy to write, but it does not scale well because of the amount of work that it must do to count the number of records that satisfy the query that each facet value represents. There might be hundreds of facet values; the code needs to calculate result counts for each one of them in order to determine which five facet values to list for the facet. The situation gets worse quickly with the number of records in the database, with the number of facets that your application is displaying, and with the number of possible facet values for each facet. If your application is displaying 50 facets and deals with millions of records, then you don't really have an option but to precalculate the facet values and include them in the record.

The XML documents in Listing 7 and Listing 8 would each take on two new elements: `scanner-name` and `size`. That simple change will allow the implementation to scale much better.

Transform data with your queries

One of the biggest strengths of XQuery is its ability to deliver data exactly as the caller needs it. But this strength is possibly the most underused. Often, architects are tempted to treat a native XML database as a back-end XML Web service, that returns XML documents that the front end is supposed to transform and render as necessary.

Businesses that already use XQuery to retrieve data from a native XML database are always quick to point out all of the reasons for returning the data in XML format and then using XSLT (for example) to convert the data at the front end. Here are some of the more common reasons:

- We plan to create other products that use the same data for other purposes
- We have front-end people that already know how to use the XSLT, Perl, PHP, JavaScript, and Java™ languages
- We want a Service-Oriented Architecture
- None of us know XQuery and we want to limit its use as much as possible
- We have a data pipeline in place
- XQuery looks complicated
- XQuery can't do X

There's not enough room here to rebut each of these statements, but keep the following points in mind:

- Data that is ready to render to a browser is often much smaller than the original data that the database stores
- XQuery is easier, more powerful, and more concise than XSLT, by any standard
- Transforming your data on the way out is much faster than transforming it later using XSLT (or just about anything else)
- You can write XQuery such that clients can request data in various formats
- The general complexity of the application will decrease considerably if the code that transforms the data is near the code that locates the data and both are written in the same language

The first of those points—about the size of the data to be rendered versus the size of the original, untransformed data—bears a little extra consideration. Beware of large XML records. Try to avoid sending large records to the front end to be transformed there. Putting the smallest possible chunks of data on the network can often improve the responsiveness and scalability of an application appreciably. And you'll often be doing exactly that: Putting smaller chunks of data on the network, when you transform the data on its way out of the database.

The real reason that businesses fail to take advantage of the full power of XQuery is fear of the unknown. This is completely understandable, but if you already use a native XML database, then using its full capabilities will only make things better in the end.

Profile the XQuery code

In general, profiling your code means determining how much time the computer spends in each part of your code. The idea is to identify the parts of your code that might benefit most from optimization. These are not necessarily the parts that run the slowest; sometimes, a piece of code that is already quite efficient will be the one you want to focus on the most. For example, one piece of code that runs in 10 seconds can optimized rather easily to run in less than a second. If that code runs only once per day, then your efforts might be better spent improving the speed (even ever so slightly) of a function that runs 1,000,000 times per day.

Most native XML databases have tools to benchmark or profile code. Use them. Sometimes these tools will not measure the performance of some particular piece of code in the way you want. In that case, don't hesitate to create your own code to benchmark a process. There's nothing wrong with inserting code to mark and measure time within your XQuery modules, especially if you can disable that benchmarking code in production.

Also, because XQuery is a functional programming language, each function stands on its own. To a large degree, the return sequence of an XQuery function depends exclusively on the parameters with which you call it. It's easier therefore to develop unit tests and performance tests to assess XQuery functions than it is to develop tests for functions in standard procedural programming languages such Java, Python, Perl, C, and PHP. You can easily time functions in your XQuery code with external processes such as a quick script. A short and clever Emacs script, for example, will allow you to run and time XQuery code that you are editing merely by highlighting the code and hitting a key combination. The script can send the code to the server, have the server evaluate it, then return the results to a new buffer with an execution-time stamp.

Keep a list of optimizations

Keep a list of potential optimizations that apply to your platform and look through the entire list every time you need to improve the performance of an application. In addition to the optimizations that this article has already covered, I keep the following items on my list.

Precompile code where possible

Some native XML databases have the ability to precompile or parse XQuery code. For code that runs frequently in a server, you'll see measurable gains in performance if you can ensure that the server doesn't have to parse or compile the code upon each encounter.

Code against the index

In many cases, a native XML database can evaluate an XQuery or XPath expression and resolve the query directly from the indexes, without ever having to retrieve a document. Where possible, you should strive to write queries that do this. Look for a database option or search-function option that allows your search to retrieve its results directly from the indexes and that avoids any kind of validation-filtering of the results.

Retrieving the results from the index without filtering has its drawbacks. You have to be careful if the nodes you're searching are not top-level nodes (or fragment roots). Be prepared to understand the results, which might include nodes that a filtered search would not include.

Consider XQuery extensions

Most native XML databases provide XQuery extensions that are designed to run fast. When you stray from strict XQuery, your application becomes more tied to a specific product, but in practice, in production, when dealing with a great deal of data, you definitely want to consider the advantage of performance extensions. Portability often comes at a price.

Understand forests, stands, and merges

Some native XML databases keep their data in binary format in forests (directories), which contain stands (files). New records often go into new stands. The system (or an administrator) merges stands periodically to improve performance—the fewer stands in a forest, the better the query response times. But you don't want forests to grow to be too large. When you optimize your system, investigate the maximum optimal size for forests and the maximum number of stands that you want before you start (or configure the system to start) a merge.

So, data loads trigger merges. And merges can bring a down a system that is operating near the limit of its throughput capabilities. When possible, you should schedule data loads and merges to happen when the system load is at its minimum. Data loads have a much smaller impact on performance than do merges. If your data loads run for long periods of time, then consider forbidding only merges during peak load times.

Summary

Native XML databases are currently powering sites that support complex searches against databases with tens of millions of records. Under the right conditions, for some applications, these databases can give organizations considerable competitive advantages over slower adopters. But just like any other database technology, native XML databases will provide the most benefit to those who understand well how to optimize their systems for efficiency and responsiveness.