Using Relational Data in XML Applications

Michael Kay

In previous articles in this series, I've stayed firmly within the boundaries of the XML world: not always a cosy world in which everything integrates smoothly, but at least a world in which different tools can usually share the same data without too much difficulty, because that data can always be expressed in XML.

For this article, the folks at Stylus Studio have persuaded me to tackle something more challenging: to describe the various different ways in which you can bridge the gap between the XML world and the world of relational databases.

Because this is such a common requirement, there's a vast range of options available, ranging from very low-tech "cobble it together yourself" approaches, to hi-tech packaged software like the DataDirect XQuery engine, which integrates data seamlessly from a variety of sources. Somewhere in this spectrum is the answer that's right for your particular problem.

I'm going to tackle this subject by first outlining the different things you might be trying to do. With luck this will then provide a framework on which we can organize the various kinds of technology available. But before we start, it's worth looking at why there's a problem.

Impedance Mismatch

As someone who once studied physics, I can't say I'm particularly fond of the way the term *impedance mismatch* is used in computing: but the phrase seems to have stuck, so I'll go with it. It was first used to describe the mismatch between the tabular relational model used for databases and the much more flexible models used in object programming languages; today the term is also used to describe the mismatch between the relational and XML models.

As we all know, the relational model represents data in terms of tables and columns. Some kinds of information lend themselves well to this treatment: what I call the "ledger book" data. Other kinds of information are very difficult to represent in tabular form. The text of this article or the content of the Stylus Studio web site are obvious examples. In between are many data structures that can be stored in relational form if you have to, but where it might not have been your first choice: geographical data and engineering data come to mind as examples. During the 1990s, under pressure from a new breed of object databases, relational databases evolved to become better at handling some of these data types, while still fitting them into the same basic rows-and-columns framework.

In integrating relational data into XML applications, one of the factors we will have to consider is whether the data is "naturally relational" or not. (Perhaps it never is: the data we store in relational databases is self-selected to be the kind of data that fits in rows and columns. We're so accustomed to leaving the less tractable information on one side that we forget we are doing it.)

The mismatch between the relational and XML models happens on several levels:

- At the level of data structure, relational data is tabular whereas XML is hierarchic. It's easy to represent a table in XML as a two-level hierarchy consisting of rows and cells within rows, but it's harder to represent a hierarchy as a table: there are many different ways of doing it, and they all have different strengths and weaknesses.

  (The story is complicated by the fact that modern relational databases are not strictly tabular any more – in fact, they aren't truly relational any more. Many products have added object-relational features which effectively turn them into hierarchic databases.)

- At a more abstract level, relational data is rigid whereas XML is flexible. In a relational database there is always a schema, and you can't store data unless the schema defines somewhere to store it. With XML, schemas are optional, and even if you do have a schema, it can be a permissive one that allows data to be stored (perhaps in different namespaces) that's outside the scope of the schema. Also, it's common for XML schemas to describe large numbers of optional fields that might be very rarely used in practice. It's possible in theory to have a database schema in which there are hundreds of unpopulated columns, but it's not a design I would recommend to anyone.

- At a more detailed level, the data types of XML Schema and the data types of the SQL standard don't have a precise correspondence. For example, xs:dateTime is almost the same as a SQL TIMESTAMP, but not quite: the details of how timezones are handled differ from one to the other (as do esoteric things such as support for leap seconds). Even basic types such as character strings and numbers are not exactly the same. They're close enough that you won't encounter problems very often, but that can make the problems especially bewildering when you hit an edge case.

Some Use Cases

Use cases (example scenarios) are a good way of analyzing requirements, so let's look at a few.

Use case 1: occasional reference to SQL data

Let's say we have an HR application for managing employees' annual performance appraisals, and that these appraisals are captured as XML documents (perhaps via a web interface using XForms).

The application might want to make occasional read-only reference to an operational database, for example to extract details such as the employee's date of joining and period of notice.

There might also be a need to make occasional updates to the relational database, for example recording that an appraisal for a given year has been completed, or recording the performance level arrived at by the appraisal process.

Use case 2: document management using a relational database

We'll stick with the HR application. The appraisals are still XML documents, but with tens of thousands of employees we want to manage these documents more effectively than simply keeping them as operating system files on some server. We want to retain the flexibility that XML provides (for example, the format of the documents might vary between countries or operating divisions, and might change from one year to the next), but we want to be sure that we can easily locate the appraisal for a given employee in a given year.

One solution to this is an XML database such as Mark Logic. But there are good reasons for trying to avoid proliferation of different kinds of database in an organization. Perhaps you have carefully built up a skilled Oracle team over the years and you don't want to dilute or duplicate this investment. So you want to use your existing relational database to provide storage and retrieval of the XML documents.

Use case 3: extracting data from XML documents

Perhaps using the relational database to manage XML documents isn't enough. You want to make the data in the XML documents accessible to relational applications. Let's say, for the sake of variety, that the XML documents this time round are reports of financial data from your operating divisions, perhaps using the XBRL business reporting language. You don't simply want to store these as XML and manage them as documents, you want to extract the figures from the incoming XML messages and hold the data in your relational database. Perhaps you also want to use the power of the relational database to slice and dice the figures so you can produce new XBRL documents for the next stage of reporting: which leads us into Use Case 4…

Use case 4: extracting XML documents from relational data

This is the converse of the previous case. Every quarter, as a vendor based in the UK, I have to report to the tax authorities a summary of the sales I have made in other European Union countries, and I am expected to submit this report in XML. Details of my sales are held in a relational database, so I need to extract an XML document, in the required format, to send to the government.

In this example the XML data is intended for transmission to another software application. But it might equally well be a document intended for human consumption. Perhaps you might want to extract it directly in XHTML, XSL-FO, or SVG; or perhaps you would prefer to extract the raw data in XML, for subsequent conversion into presentable form using XSLT stylesheets.

Use case 5: query across XML and relational data sources

Sometimes the data held in XML documents is sufficiently complex (or variable) that you don't want to transfer it all into your relational database. But you do still want to be able to query it; and you also want these queries to have access to your relational data. For example suppose you are creating a web site for a soccer tournament (there's more to life, after all, than appraisals, tax, and accounts…). You might well have a collection of player biographies held in XML, and details of the results of

matches held in a relational database. To build a particular page on the web site, you might well want to combine information from a player's biography with database information about who scored the goals in each match.

A Survey of Technical Solutions

We've taken a quick look at some of the things you might want to achieve. Now let's take a look at the technical options available. There's a wide range of choice, with different levels of flexibility. A word of warning though: database vendors are notorious for announcing technology long before it is really available in production quality on every platform.

Extracting data as XML

All the relational database vendors offer some kind of XML extraction capability. Let's look at these facilities briefly.

Most databases now have the ability to return the results of a query in XML format. For example, in SQL Server you can enter the query:

```
SELECT * FROM EMPLOYEE FOR XML RAW
```

which will give you a dump of the entire EMPLOYEE table in "raw" XML format: this means you get no control over the representation. Each row in the result is output as an element, with the column values represented as attributes. Of course you can then put it through an XSLT transformation to turn it into something else.

Instead of RAW you can specify AUTO. This option is rather interesting, because although the SQL specification says that the result of a query is always a table, the AUTO option actually returns a hierarchy based on the structure of the joins in your query, which avoids outputting redundant data. The result is much closer to the natural XML representation of the data.

Finally there's an option EXPLICIT that allows you to control the structure of the results yourself. But personally I find this difficult to get to grips with: it's easier to take what Microsoft chooses to give you, and then transform it using XSLT. But perhaps that's because I'm an XSLT fan!

There are various ways you can use such extended SQL statements, depending on the environment you're in. One mechanism is to embed the SQL statement in a `<sql:query>` element within a stylesheet-like XML document called an XML Template. When you execute this, the query is replaced by its XML result. This gives you a simple alternative to ASP or JSP pages when all you need to do is to substitute some current data into an HTML page.

For Oracle the equivalent is the XML SQL utility. Using XSU, you can enter a standard SQL query such as `SELECT * FROM EMPLOYEE WHERE EMPLOYEE_NR='517541'`, and get back the answer in the form of an XML document such as:

```
<ROWSET>
  <ROW num="1">
    <EMPLOYEE_NR>517541</EMPLOYEE_NR>
    <NAME>Michael Kay</NAME>
```

```
  </ROW>
</ROWSET>
```

The result doesn't need to be raw lexical XML, of course; it can be delivered as a DOM document, or as a stream of SAX events, which means you could send it directly into an XSLT transformation without the need for further parsing. As with the Microsoft tools, there are facilities to control the format of the returned XML so as to remove the need for subsequent transformation. Because Oracle support object-relational structures, one way to do this is to define a hierarchic representation of the data as an object-relational view. When you query this, the returned XML will reflect the hierarchic structure of the view.

Oracle also has a utility, called XSQL pages, that allows you to embed SQL statements in a skeletal XML document. A request from a browser to this document is directed to a servlet, which executes the SQL statements and enters the results into the page before delivering it back to the browser. Formatting of the page can then be controlled on the client side using either CSS or client-side XSLT.

As an alternative to the utilities offered by your database vendor, there are similar tools available from third parties. A number of XSLT processors have similar facilities developed as XSLT extensions. This gives you a capability independent of what your particular relational database vendor provides – though there's no free lunch, because it locks you into your XSLT processor instead. For simple tasks, it's not all that difficult to do the same thing yourself, just by writing some Java code that sends the query to the database using JDBC, and then formats the result as XML. It's more code to write, but it saves the trouble of learning another new technology.

These tools also have the ability to transfer data in the reverse direction, from an XML document into relational tables. However, database updates are always more complex, because there is no unique mapping from a hierarchical structure to a flat one. As there are no standards in this area, you will need to study the documentation for your chosen product.

Storing XML in Relational Databases

There are basically two ways of storing XML in a relational database: you can store an entire XML document as a single object in a cell of a table, or you can take the document apart and store the individual pieces – which is known by the picturesque name of *shredding*.

I'll look first at storing documents in their entirety. One way to do this, of course, is simply to pretend that the document is a character string (or a byte string) and store it as an anonymous BLOB or CLOB, in much the same way as you might store an image. The disadvantage of this approach is that you get no help from the database management system: you won't be able to search effectively on the document contents, or to do a query that aggregates across multiple documents, or to retrieve a small part of the document, except by pulling everything you need into the application and doing the work there. Nevertheless, this is sometimes a viable strategy. I once used this approach in a system that had to store some tens of thousands of product bulletins: these were accessed by date, by title, and by product

family, and the data needed to support these access routes was stored redundantly in separate columns of the database making them accessible to regular SQL queries. We also indexed the XML as if it were plain text to provide a keyword search capability.

A more sophisticated approach is to let the database system know that what you're storing in the column is in fact XML. That is, XML becomes another data type supported by the SQL database engine, with its own set of associated operations. This is the approach taken by the ISO SQL/XML specification, which we'll look at in the next section.

ISO SQL/XML

Names are confusing, especially when there are lots of different things with similar names. The Microsoft product features that I talked about in the previous section are often referred to as SQLXML (with no slash). In this section, I'll be talking about a new part of the SQL standard called SQL/XML (with a slash). Oracle's implementation of SQL/XML is also often referred to as SQL/XML (with a slash) despite the fact that it includes a number of features which are not in the standard and omits some that are (spot the key phrase in the marketing literature "*based on the emerging SQL XML standard*", which a cynic can read as "has some resemblance to an old draft of the standard").

ISO SQL/XML, in its 2003 incarnation, does three things:

- It defines XML as a new data type for the columns of a relational database, and provides some basic operations on that data type
- It provides mechanisms for exporting relational data as XML
- It defines mappings between relational data types and XML Schema data types

This allows you to store an XML document in your relational database, but it doesn't actually do much else. There are no search or comparison operations, so the only way you will be able to find your XML documents is by storing additional information in other columns of the table or in "side tables".

One thing you can do, though, is to construct XML documents within a SQL query. The XMLELEMENT function constructs an XML element. Here's an example:

```
SELECT E.EMPLOYEE_NR,
       XMLELEMENT(NAME "e", E.FIRST || ' ' || E.LAST) AS "result"
FROM EMPLOYEE E
WHERE E.LOCATION='Reading'
```

(In case your SQL is rusty, || is the string concatenation operator.)

This query might produce the answer:

EMPLOYEE_NR	result
517541	<e>Michael Kay</e>
573924	<e>John Smith</e>

Note that the result of the SQL query is (as always) a table, not an XML document; but it may contain XML documents in the cells of the table.

Calls on XMLELEMENT may be nested to produce a hierarchic structure. For example we could refine the above example to read:

```
SELECT E.EMPLOYEE_NR,
       XMLELEMENT(NAME "e",
         XMLELEMENT(NAME "first", E.FIRST)
         XMLELEMENT(NAME "last", E.LAST)) AS "result"
FROM EMPLOYEE E
WHERE E.LOCATION='READING'
```

and the output would then be:

EMPLOYEE_NR	result
517541	<e><first>Michael</first><last>Kay</last></e>
573924	<e><first>John</first><last>Smith</last></e>

In my example I've shown the element values being calculated from rather simple expressions, but in fact you can use any SQL subquery, which can also produce results of type XML. There are similar functions to generate attributes and namespaces, and some more complex functions to generate sequences of elements.

The 2003 version of the specification is really just a first step. Work is well advanced on a revised version that will incorporate support for XQuery so that you will be able to search the XML documents as well as constructing them and serializing them. In practice, of course, you're probably more interested in what you can do with your favourite relational database product today, rather than what's in the next version of the standard. The good news is that all the major vendors have announced facilities that do much of what you would want; the bad news is that the products are not necessarily shipping yet.

The most advanced implementation so far in terms of deliverable product (and any connection with Stylus Studio here is purely coincidental, of course!) comes not from any of the relational vendors, but from a third party, DataDirect. DataDirect has a high reputation as a supplier of ODBC and JDBC drivers, which one might think of as fairly mundane component software, but the experience has given them a good insight into the intricacies of interfacing into different vendor's database products, and the detailed tricks that are needed to get respectable performance; the other thing they bring to the table, which is important for some users and not for others, is the ability to avoid lock-in to any particular database product by using a common front-end that talks to all of them. (End of commercial.)

The new version of the ISO spec allows you to embed XQuery code inside SQL queries. Here is a sample:

```
SELECT E.EMPLOYEE_ID,
```

```
       XMLQUERY('$appraisal//performance-level'
         PASSING BY REF E.APPRAISAL AS 'appraisal'
         RETURNING SEQUENCE) AS "managers"
```

In this case the query is a simple path expression, but it can of course be any legal query. Notice how the XQuery variable `$appraisal` is initialized from the surrounding SQL code.

This isn't exactly seamless, and you can see that both the SQL statement and the enclosed XQuery code could get much more complex than this. The mixture of different syntax conventions will no doubt cause enormous confusion, especially when the combined statement gets embedded in another language such as Java or ASP.NET or XSLT. When you consider that there might also be regular expressions nested within the XQuery code, sorting out the multilayered escaping of special characters such as quotation marks is likely to prove a nightmare. But we're dealing with heterogeneous technologies here, so the fact that they can be made to work together at all is good news.

Shredded storage

The other way to store your XML data in a relational database is to shred it: that is, to split it up into its smallest parts, and store these individually. There are two variations on this: you can try to preserve fidelity to the structure of the XML document that you started with (so that it can be reconstructed more-or-less exactly), or you can try to produce a faithful normalized representation of the information content of the document, that looks as much like ordinary relational data as possible.

As with the approach of storing the entire document, there are two ways of doing shredding: you can do it yourself, in your application code, or you can use tools supplied by the database vendor or by third parties.

Shredding works well if your XML document has a very regular structure: for example, if it is simply being used to export data from one relational database and import it into another. It works much less well when the XML has deep levels of nesting, mixed content, recursive content models, heavy use of optional and repeated data, and exotica like comments and processing instructions.

On SQL Server, shredding can be achieved using the OPENXML construct (a Microsoft SQL extension), which defines the rowset you want to create, using XPath expressions to define where each column of the data comes from in the original XML document.

The reverse process to shredding (putting the XML document back together again) is sometimes called XML publishing. This can be done using the SQL/XML operators discussed in the previous section. Here's an example, which uses several more of the functions in SQL/XML (you'll have to look elsewhere to get the details of how they work):

```
SELECT XMLSERIALIZE
       (XMLELEMENT(NAME "DEPARTMENT",
            XMLATTRIBUTES (d.deptid, d.deptname ),
```

```
            XMLAGG(
              XMLELEMENT(NAME "EMPLOYEE",
                XMLFOREST (e.empno, e.firstname,
                           e.lastname, e.birthdate, e.salary)))))
         AS "deptdoc"
FROM department d, employee e
WHERE d.deptid = e.deptid
GROUP BY e.dept;
```

One significant drawback here is that there's no guarantee that the publishing code is the inverse of the shredding code: there's lots of scope for the two to get out of sync as the schema evolves. This clearly calls for some higher-level tooling. I looked around on the web to see what might exist, and it was no great surprise to find myself right back on the Stylus Studio pages with their SQL/XML Editor, which seems expressly designed to tackle this problem. I haven't used it myself, but it looks very promising.

Shredding is appropriate when the relational form of the data is the primary form, and XML is being used merely to get relational data from one place to another. It's less appropriate where the design of the data is primarily an XML design. For example, if you are in the financial services business, you might be exchanging business data with other companies in the FpML format. Although this is structured data, it's a very complex format, and in any given message most of the potential elements are likely to be missing. You might want to extract the fields of interest into your relational database as part of a business process, but I doubt you would want to store a shredded representation of the original message, because it would simply be too complex.

I've talked here about shredding approaches that preserve the semantic information content of the XML. There are other approaches that try to retain everything in the original XML document so it can be faithfully reconstructed: element order, node identity, comments and processing instructions, namespace prefixes, and so on. This is effectively a third-normal-form representation of a DOM tree. The resulting structure might represent the original XML faithfully, but it is likely to be very unwieldy (and inefficient) to query from SQL.

XQuery access to relational databases

We've seen one approach to querying XML held in relational databases: write a SQL query, and within the SQL, use the XMLQUERY function to execute an embedded XQuery on the XML data held in a table cell. If you want to construct XML output, use the XMLELEMENT function. This is pretty cumbersome, and one feels there must be a better way. There is: write the whole thing in XQuery, treating the entire relational database as if it were a collection of XML documents (including, of course, the bits that really are XML). Clearly this offers the promise of a much cleaner solution than the mixed bag of technologies we've been talking about so far.

The relational vendors themselves have been slow to ship their products in this area, though they all have announced plans. Meanwhile, third parties such as DataDirect and BEA have been jumping in on the act. Paradoxically, the ordinary relational data is more accessible to these external XQuery engines than data stored using the

native XML type, simply because the only way to get inside the native XML data is by using the database vendor's own XQuery engine, which in many cases hasn't been shipped yet.

The obvious way to expose the relational data to XQuery is by making each table into a document, with an element to represent each row and within that, an element (or attribute) to represent each cell. This is the default mapping used in DataDirect XQuery. You can use the XQuery `collection()` function to access a particular table. The result is an XML document in which the rows of the table form the children of the document node. Perhaps you imagined that XML requires the document to contain a single enclosing element? Well, in "lexical XML" or XML-as-angle-brackets, that's true. But the XQuery data model dispenses with this requirement—you're allowed to store multiple elements directly under the document node. So the content of a color table can be accessed as `collection('color')`, and is presented as a document containing a sequence of elements, like this:

```
<color>
      <name>AliceBlue</name>
      <code>F0F8FF</code>
</color>
<color>
      <name>AntiqueWhite</name>
      <code>FAEBD7</code>
</color>
<color>
      <name>Aqua</name>
      <code>00FFFF</code>
</color>
<color>
      <name>Aquamarine</name>
      <code>7FFFD4</code>
</color>
```

Here's a query that determines the color code for DeepPink:

```
collection('color')/color[name='DeepPink']/code
```

Or if you prefer to use FLWOR expressions, you can write:

```
for $c in collection('color')/color
where $c/name = 'DeepPink'
return $c/code
```

It's worth noting that even without any XML Schema present, the data in this view is typed according to the original SQL data types. For example, a numeric column might contain values of type `xs:integer` or `xs:decimal`. This means that operations that rely on the data type, such as comparison and sorting, are all taken care of automatically. It also means that you get all the advantages of writing schema-aware queries.

And of course, you could easily write a query that outputs the data as XHTML. Ignoring the boilerplate, here's the essence of it:

```
<table>{
      for $c in collection('color')/color return
      <tr>
            <td>{$c/name}</td>
            <td>#{$c/code}</td>
            <td style="background-color:#{$c/code}"> </td>
```

```
      </tr>
}</table>
```

Clearly, you're not confined to writing your queries against a single table. You can access any number of tables you like, and use joins to connect them just as you would in SQL. For example, if we have a table of products in which one of the columns is the color name, we can join this with the color table when displaying details of the products:

```
<table>{
      for $p in collection('product')/product, $c in
collection('color')/color
      where $p/color = $c/name
      return
      <tr>
            <td>{$p/product-number}</td>
            <td>{$p/description</td>
            <td style="background-color:#{$c/code};
color:#{f:contrastColor($c/code)}"> {$c/name} </td>
      </tr>
}</table>
```

Notice that there's a call on a function here to compute a contrasting color, to ensure that the text is always visible on its background. Here's a rather crude implementation of the function:

```
declare function f:contrastColor($color as xs:string) as xs:string {
      translate($color, '0123456789ABCDEF', 'FFFFFFFF00000000')
}
```

So far, XQuery here is looking simply like an alternative to SQL for querying the relational data, perhaps with better facilities for generating XML output. Where the approach really comes into its own, however, is where the tabular data in the relational database needs to be combined with "real" XML, which might be in free-standing documents or in the database (using any of the approaches discussed earlier in the article).

For example, suppose that your product specifications are XML documents held in filestore, and you want to produce a report giving the short descriptions of all products that you sell in Liechtenstein. This would involve a join between data in the database and data loaded from filestore, something like this:

```
<body> {
for $p in collection('product')/product
      $t in collection('product-territory')/product-territory
      $d in collection('http://intranet/product-descriptions')
where $t/product-code = $p/product-code
     and $t/country = 'Liechtenstein'
     and $d/product-code = $p/product-code
return {
   <h1>{$p//product-title}</h1>,
   <p>{($p//product-description/para)[1]}</p>
} </body>
```

There are two big gains in adopting this approach.

- Firstly, you get access to all your data, structured and unstructured, using a single query language

- Secondly, you get XML formatting (which includes HTML, XSL-FO and so on) integrated with the query language

And of course a mature product like DataDirect XQuery will give you update capabilities as well – but that's another story.

DataDirect isn't the only company tackling this challenge: BEA's Liquid Data product, for example, has similar aspirations. The database vendors too will be offering direct XQuery access to their relational data in due course, though on past experience they are unlikely to offer products that make coexistence easier with their competitors. The whole area is likely to see a lot of growth in the next couple of years.

Conclusions

I'm aware that this article has been short on recommendations: instead, it's been more of a lightning tour of a cluttered workshop littered with tools that might help you do part of the job.

I tried to present a number of scenarios at the start, and in an ideal world I would now tell you what the right answer is for each of these scenarios. Unfortunately life isn't that simple, because no-one's problem fits exclusively into one of the five use cases I outlined, and everyone has different constraints.

One piece of advice I would love to give about integrating different technologies is: *don't do it!* That's hopelessly idealistic, of course. But when you see a big system implemented using a single architecture and a single coherent design, you realize how much more effective it can be than one whose parts are tied together with pieces of string. Sometimes it can be cost-effective to throw the old technology out and start again. Not many people are brave enough to do it, and that's understandable, but when it works, it can work really well. It's a high-risk, high-return strategy. In this case, of course, this means designing your applications to use wall-to-wall XML, databases and all.

But if you can't get rid of heterogeneous technologies behind the covers, at least you can try and create a homogenous environment for your developers. Writing applications in a mixture of half a dozen languages (PHP/ASP, SQL/XQuery, HTML, regular expressions, Ant…) often nested inside each other and requiring multiple escaping of special characters, can't be good for productivity or for the robustness of the final system. With a product like DataDirect's, you can do nearly all the development in XQuery. And if you like it as much as I do, you might well be throwing out those SQL databases in a few years from now.

But for other projects, a more ad-hoc approach might be appropriate. If you just have occasional requirements to move data in or out of your SQL database, there are low-tech solutions that will do the job. For example, most XSLT processors allow you to call out to Java methods which in turn can call JDBC to access your database. In the end, an application that contains hundreds of low-tech connections to the outside world will be a maintenance nightmare, but in the short term, it might meet all your requirements.

I've tried in this article to start by describing the problems you might be tackling, because so often people seem to start by describing the technology. I hope you can relate to one or more of the use cases, and although my survey of the technical choices was lamentably brief, I hope that having got this far you can relate the tools available to the needs they are trying to address. Personally, I would be happy to see relational databases disappear in a cloud of smoke (along with C, Windows, Unix, and the QWERTY keyboard): but it ain't going to happen. The new has to live alongside the old, and the good news is that vendors are falling over each other trying to make that possible.