Native XML Databases: Death or Coming of Age?

Craig Brown
<cmb@qti.qualcomm.com>
Xavier Franc
<xfranc@qti.qualcomm.com>
Michael Paddon
<mwp@qti.qualcomm.com>

Abstract

Back in the early 2000s Native XML Databases (NXDbs) promised a bright future. Today, adoption falls short of those promises and some have even suggested that the technology is in decline. We analyze this "trough of disillusionment" with reference to the Gartner Hype Cycle and compare it to the history of relational databases.

Despite the slow maturation of NXDbs, XML itself is now well established and here to stay. The need for repository systems able to store and query large document collections is likely to increase as ever more XML data is generated. XML is an important technology in the field of publishing and digital preservation, and is especially effective when it comes to handling a mix of text and data.

Native XML Databases might represent only a niche market, but may prove to be irreplaceable in some applications.

To illustrate this, we present a use case in which an NXDb turned out to be a good solution for a massive data warehouse problem. Why other solutions fell short is discussed, as well as the specific features that made the product we selected, the Qualcomm® Qizx™ database[1], suitable. To conclude, we provide our thoughts about which features are desirable or even possibly indispensable in an NXDb from the perspective of making native XML Databases more attractive to application designers.

Keywords: XML, database, query optimization, scalability

[1]Qualcomm Qizx is a product of Qualcomm Technologies, Inc. Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Qizx is a trademark of Qualcomm Incorporated. All trademarks of Qualcomm Incorporated are used with permission.

1. Where are all the XML Databases?

Native XML Databases (NXDbs) were a hot topic 10-15 years ago. Many believe the buzz peaked around 2003-2004. The use case seemed as obvious then as now: “if you have more than a handful of XML documents that you need to store, you should store them in a native XML Database” [7]. And yet as of 2014, we have not seen mainstream adoption of the technology.

W3C standards development has continued slowly and steadily, however, and important standards such as XQuery Update [2] were only completed in the past 5 years.

The number of published research papers in the NXDb field appears to have steadily decreased since the last decade. However research into non-relational databases continues to bloom, and many of the fundamental results are directly applicable to NXDbs. This suggests that the term “NoSQL” may have displaced “NXDb” in grant applications but that current research continues to be relevant to XML Databases.

The number of available NXDb products reached a peak of about 40 at the end of the last decade, and some are no longer maintained or commercially available [7]. Bourret's website has listed no new products over the preceding 5 years. An informal survey by the authors suggests there are currently a few actively developed open-source products and a handful of commercial offerings.

RDBMS vendors have quickly reacted to the threat of NXDbs by offering XML extensions in their products, typically as an "XML column" feature. This has blurred the distinction between relational and XML Databases, and created uncertainty in the market about the benefits of purely native XML technologies. To date, this has been effective at defending the market share of incumbent products against potential disruption.

The very expression "XML Database" has nearly disappeared from marketing literature, in favor of *"NoSQL"*. The ongoing investment in the NoSQL field points to continuing and growing demand for non-relational data models (key/value, document-oriented, graph) and yet XML Databases do not seem to benefit from this trend.

A perception exists that NXDbs lack maturity for use by enterprises. There is a kernel of truth to this: relational database products are decades old and contain many more features than the first generation of XML Databases. Of course, they also contain much more legacy technical debt. As XML offerings mature, more enterprise-ready features may be expected.

There is no well-known theory about how to model data in a non-relational way. Some relational database architects shy away from XML to represent data in new applications since there are few good examples to emulate. The presence of hierarchy and the lack of traditional schema normalization also can seem unsettling. This is a chicken and egg problem that we last saw in the 1980s. One of the authors recalls when developers were wary of the newfangled relational databases versus tried and trusted raw ISAM files.

XML standards are often regarded as too complex. Some enterprises and developers are slow to embrace radically new technologies. Adoption of XQuery [1] is a good example. As a functional language, it is relatively difficult for a programmer from an imperative background to pick up. Nevertheless, there are benefits to be realized once that learning curve has been climbed. For instance, the natural composability of XQuery functions allows the easy creation of domain specific query languages in a way that might not be obvious to an imperative programmer.

JSON has taken much of the focus away from XML when it comes to representing semi-structured data. JSON is perceived as much lighter and easier to master, especially in web environments. However, XML already has many of the components that enterprises will demand long term such as standardized schema/query/update/processing languages, which JSON still lacks.

2. Are we in the "Trough of Disillusionment"?

According to the Gartner Hype Cycle [6], after reaching the “Peak of Inflated Expectations,” the visibility of new technologies frequently goes through a “Trough of Disillusionment” before possibly scaling the “Slope of Enlightenment” or maturation phase.

The history of relational databases is instructive: the first paper was written in 1969, lots of research and investment occurred through the 1970s; finally credible products emerged, starting around the early 1980s. The market went on to mature, and by the early 1990s many early entrants had retired their products [8]. The survivors of this shakeout enjoyed strong growth from the mid 1990s onwards. Looked at through this lens, XML databases are still very young. If the past is a guide, one would expect credible products to be emerging sometime this decade. XML Databases might only now be starting up Gartner's slope of enlightenment. If that is the case, then one would expect a future phase of mass adoption.

Obviously the NXDb situation is not exactly the same, since RDBMS had no incumbent to challenge at the time, but this observation confirms that the actual pace of adoption of a new technology can be much slower than initially imagined by its proponents.

This begs the question: why would anyone choose an NXDb over the alternatives? Some key indicators that an NXDb may be the most suitable platform are:

- Where XML is used as representation in the first place.
- Data where the schema is significantly evolving over time.
- Information merged from multiple, disparate sources (with schema evolution being common here, too).
- Complex hierarchical relationships in data.

The authors encountered a use case exhibiting all of these characteristics. We present it as an example of the sorts of problems that we believe will drive NXDb adoption in the future, and recount our experiences in building a solution.

3. Use Case: a World Patent Database

It is common for an inventor (or the company they work for) to apply for patents to protect the fruits of their research.

An invention may only be patented if (amongst other criteria) it is novel. Anyone applying for a patent is therefore likely, at some point, to consider searching through existing patents (and related publications) to see if prior art exists. But there are millions of active patents nowadays, and without effective tools that task can become extremely time-consuming and costly.

This is a quintessential big data problem. There is an enormous amount of information available in a vast number of documents published by patent offices around the world. Some is textual. Some is structural. While simplistic key word searches are applicable to this data, modern machine learning techniques (e.g. clustering and classification) are also valuable. Our use case essentially consists of a massive document warehouse that must be queried in many different ways, not all of which are foreseeable. Some of these queries are made by humans looking for a small result set. Others are made by machines and may return extraordinary amounts of data.

Patents (and related documents such as published filings) are extremely hierarchical, and exhibit an evolving schema. XML is a natural format for their representation, transmission and storage, and we were not surprised to discover this was indeed common practice. Figure 1 depicts some structure from a sample document, from which a sense of the schema may be inferred. Since representation is a solved problem, our challenge reduces to effectively storing, correctly indexing and efficiently querying a huge corpus of complex XML documents.

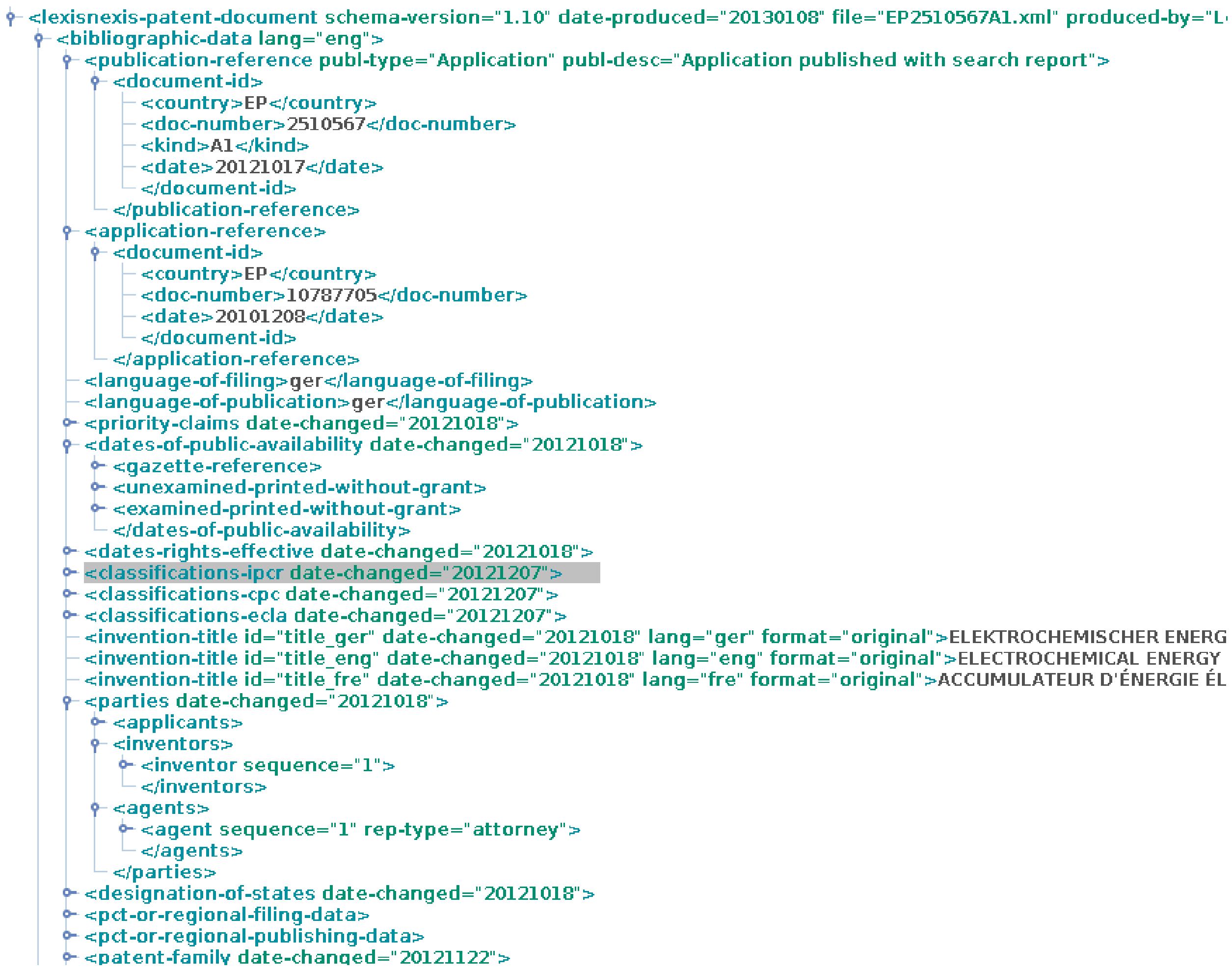
How large? If one takes all the published patents and filings from just three patent offices (USPTO, EPO and WIPO/PCT) there are nearly 20 million documents. That's over 2.5 terabytes of data (about 125 kilobytes per document on average) and growing fast. Over 850,000 new or updated documents are issued per month.

How big is 2.5 terabytes? Merely to copy from hard disk to hard disk (at circa 100 Mb/s) takes about 7 hours; In our tests, parsing that much XML in order to extract fields took days on a contemporary CPU.

All patent documents use a common schema in principle, but there are variations according to time and originating office. This schema defines about 200 elements and 50 attributes and includes MathML. From time to time new elements are added. Any solution must deal gracefully with this change, including in updates to old documents.

The types of queries that must be supported for our use case are:

- Large batch queries for statistics and machine learning.
- Small queries generated by interactive user interfaces (with many concurrent users).
- Custom ad hoc queries for data exploration. Such queries can be complex with multiple conditions and constraints.



```
<lexisnexis-patent-document schema-version="1.10" date-produced="20130108" file="EP2510567A1.xml" produced-by="L
  <bibliographic-data lang="eng">
    <publication-reference publ-type="Application" publ-desc="Application published with search report">
      <document-id>
        <country>EP</country>
        <doc-number>2510567</doc-number>
        <kind>A1</kind>
        <date>20121017</date>
        </document-id>
      </publication-reference>
    <application-reference>
      <document-id>
        <country>EP</country>
        <doc-number>10787705</doc-number>
        <date>20101208</date>
        </document-id>
      </application-reference>
    <language-of-filing>ger</language-of-filing>
    <language-of-publication>ger</language-of-publication>
    <priority-claims date-changed="20121018">
    <dates-of-public-availability date-changed="20121018">
      <gazette-reference>
      <unexamined-printed-without-grant>
      <examined-printed-without-grant>
      </dates-of-public-availability>
    <dates-rights-effective date-changed="20121018">
    <classifications-ipcr date-changed="20121207">
    <classifications-cpc date-changed="20121207">
    <classifications-ecla date-changed="20121207">
    <invention-title id="title_ger" date-changed="20121018" lang="ger" format="original">ELEKTROCHEMISCHER ENERG
    <invention-title id="title_eng" date-changed="20121018" lang="eng" format="original">ELECTROCHEMICAL ENERGY
    <invention-title id="title_fre" date-changed="20121018" lang="fre" format="original">ACCUMULATEUR D'ÉNERGIE ÉL
    <parties date-changed="20121018">
      <applicants>
      <inventors>
        <inventor sequence="1">
        </inventors>
      <agents>
        <agent sequence="1" rep-type="attorney">
        </agents>
      </parties>
    <designation-of-states date-changed="20121018">
    <pct-or-regional-filing-data>
    <pct-or-regional-publishing-data>
    <patent-family date-changed="20121122">
```

Figure 1. A patent document sample (styled view with some nodes folded)

4. The Journey to an NXDb

In order to find a solution for our use case, we embarked on a journey in which we experimented with many different database technologies. It was this process that eventually led us to NXDbs, and in particular Qizx, as a viable solution.

4.1. Key/Value stores: the joins killed us

Key/Value stores basically offer to retrieve some application-defined data (value) from a key. It is often also possible to scan keys matching a particular condition. Some KVS allow retrieval by content, using secondary indexes.

There are quite a few key/value storage systems available. From the perspective of the difficulties we encountered, they are all essentially similar as the root cause is an "impedance mismatch" between the primitives and the problem to be solved.

- The simplest approach is that the key is a field of interest inside a document and the value is a reference (DocId) to a document. We discuss in the next section what it takes to extract and maintain 'fields of interest'.
- In all cases, it is necessary to create key/value tables ahead to time to support all possible queries. As a consequence custom queries are impractical on large data sets.
- Due to rudimentary query primitives, most joins must be done by business logic: this is time consuming to implement, error prone, and inefficient (much data has to be read and conveyed from storage to client).
- Most joins or constraint checks (e.g structure based) require pulling and parsing entire documents: this is terribly slow. Extracting specific parts of documents is also cumbersome.
- No free text search: necessity to add a third-party full-text engine.
- Some sort of structure index could have been implemented as key/values. Then we would have built a kind of native XML database!

4.2. Relational Databases: schema maintenance killed us

Using a Relational DBMS seemed a viable approach, building on a mature technology and query language. Mapping XML nodes to a relational schema requires data to be decomposed into tables and fields. This process is generally called *shredding*.

Some databases can automatically perform the decomposition task, based on a XML schema:

- The problem here is to have an up-to-date schema.
- Automatic shredding is notoriously slow.
- The resulting relational schema is quite complex. Writing queries against such a complex schema is really hard.
- We could just have used an "XML column" in a relational database. In that case however, indexing of our content would have been limited or non-existent, and querying would be impractical beyond a point.

Alternatively, it is possible to define a partial relational schema: this means defining entities (or types) of interest in the XML document (e.g. patent, inventor, claim), and for each entity, defining named properties as a function of the contents.

- Defining a complete relational schema is an enormous challenge, that requires hundreds of tables and thousands of constraints.
- Without automatic shredding, custom code is required to parse documents, extract and index fields: this is expensive and error prone.
- There is always a new type or field that has not been seen yet: that induces a Sisyphean task of continual rewrites of database schema, parse/store software with days/weeks of reloading.

4.3. Full-text search engine with non-text fields

Many full-text search engines offer the possibility of handling several "Fields" which can even have non-text value types like numbers or dates. A typical example is Lucene. An example of this approach applied to patents is Google Patents[2]. It offers a limited search interface with links providing a kind of drill-down functionality.

As with relational databases, each document has to be decomposed into a set of fields.

- Each change in the field-set implies rescanning all the documents and re-indexing, a task usually requiring several days of processing.
- It is unwieldy or impossible to deal with entities in the document that do not decompose cleanly into flat fields.
- We lose the ability to leverage hierarchy in any but the most basic ways. Often this is very important. We might want to query all the names in a patent filing. Tomorrow, we might want just the inventors' names. The relative position of elements carries important information.

4.4. Other non-XML systems

We also considered a NoSQL document-oriented database such as MongoDB, as it can handle semi-structured data in the form of JSON.

However there was a major issue: JSON cannot represent XML with mixed content.

4.5. Native XML Databases

Storing XML documents directly as XML, being able to retrieve the documents in their original form, then querying this data using a powerful and standardized

[2] http://patent.google.com

query language (XQuery), seemed a natural and promising way to solve our problem. But the devil was in the details.

To determine which NXDbs were suitable to our problem, we ran a set of trials.

Open-source products were examined initially, since by nature they have evaluation versions without limitations.

The problems we encountered fell in several categories:

- **Stability**: several products terminated with software crashes or data corruption issues under intense upload strain. Durability was fundamental to us since it can take weeks to import really large datasets.
- **Scalability**: we encountered two classes of problem as the database grew large:
  - Update performance dropped off severely. A slow decay in speed is naturally expected (typically `1/log(n)`) since many algorithms involved have a `n×log(n)` complexity at best. But in some trials we observed such a rapid fall in performance that it became clear that we would be unable to update documents as fast as they were being produced.
  - Some internal limit was reached: for example in one trial the database could not store more than 4 billion nodes (2^{32}).
- **Query speed**: this was always going to be a major challenge for a multi-terabyte document corpus. We observed that many products became unable to return queries fast enough for our needs when our collection grew beyond a few million documents. In some cases, this may have been improved by defining a number of custom indexes - but that would have created an ongoing issue of future ad hoc queries requiring new indexes.

Our trials led us to conclude that, at that time, none of the open source candidates were a good fit for our use case. Different products had different pros and cons, but none met our exact needs. We note that many of these products have continued to rapidly mature in the 2-3 years since our study. The reader is encouraged to run their own trials for their own use cases.

We then began to assess commercial offerings. A key requirement for us was to avoid proprietary query languages and consequential vendor lock-in.

4.6. Finally, Qizx

Qizx was one of the commercial offerings we chose to trial. Qizx is an NXDb with several interesting features that was initially developed by a small European company, and has since been acquired by Qualcomm Technologies, Inc.

Qizx is a lightweight product, in the sense that it has a focused feature set, is parsimonious with resources and runs fast. It has good documentation, tools to make evaluation easy, and transparent licensing. In addition it has the following particular characteristics:

- Stability and Scalability: Qizx is able to run for more than one week ingesting 2 terabytes of XML.
- **Automatic indexing**: There is no need to define ad hoc indexes in order for new queries to be executed efficiently. Qizx creates all necessary indexes and uses them transparently. We found that this feature is a game-changer.
- Fast, standard queries: Without requiring any particular effort, except writing good XQuery, Qizx has excellent response times against large collections. It supports XQuery Full-Text and XQuery Update.
- Application-specific key-value properties (basic data or XML fragments) can be attached to documents, and used for querying. This support for metadata makes it possible to perform pre-computation on documents and store the results without touching the original data.
- Data type conversion: During the storing/indexing phase, Qizx automatically recognizes basic types such as numbers and dates in attribute or element content, and indexes them both in text form and in the recognized type. For example an element such as `<item price='1.0'/>` can be found by a query such as `//item[@price = 1]` (numeric value) as quickly as `//item[@price = '1.0']` (text value).

Qizx turned out to be an excellent fit for our use case.

5. Reflections

Our journey provided us first-hand experience as to why a native XML database can be a superior solution for handling a massive XML corpus with complex structure. We also discovered just how difficult it was to find the right technology for our needs despite there being quite a few options to choose from. That might explain why some give up and write off native XML solutions! That nearly happened to us, but we are pleased that we persisted.

Along the way, we learned a lot about what might make an NXDb more attractive to an application architect.

Automatic indexing

The problem with most non-XML technologies is that they require re-scanning of the entire dataset in order to extract the new fields or to build new specialized indexes, a process that can take days of computation in our case. This is not only very costly, it can be Sisyphean at development time. It can also imply database downtime if such changes are required during the production phase.

Storing in XML form provides the benefit that all data is available for query and that no data is lost in the translation phase. But this is not sufficient: If new specialized indexes have to be defined because of new queries then some of the benefit is lost, because adding new indexes also requires a re-indexing phase where all the data present in the database has to be scanned. The cost is likely smaller than res-canning data from source XML, but it is still significant.

Therefore it appears that the feature 'automatic indexing' is fundamental: it means "store and index" once, then execute all the queries needed without any other overhead. If in addition the queries run fast, then the benefit over other solu-tions is tremendous.

Actually there are already several NXDbs that support such a feature in some way. In our opinion, all native XML Databases should support some form of this feature, because 1) XML makes it possible and 2) without this feature, a native XML Database is much less useful.

Use of XML Schema (or not)

XML Schema can be used in several ways:

- For validating documents on creation or update. This is obviously a desirable feature.
- For determining the data type of element or attribute content. But this is a rather heavy approach for users. Requiring a schema in order to determine the types of pieces of XML data is also somewhat at odds with the "self-describing" promise of XML.
- To optimize queries. This idea has been advocated since the beginnings of XQuery. We believe that it is of limited effectiveness:
    - It is highly complex, and indeed it has never been implemented as far as we know.
    - It requires a correct association between document and schema, without version issues. This is error prone and expensive to manage for database users.

  In contrast, optimization methods that use the knowledge of the *actual* structure of documents, inferred from the data and without need for a schema have been proposed [4][5].

  The next version of Qizx will implement this type of optimization and deliver significant speedups. Our preliminary testing suggests a 2 to 10 times speedup in typical cases.

Fast queries

XML Databases should be optimized for fast querying, not for massive updating, because they are likely to be used for applications where data does not change much but rather is accumulated (historical data, publications, logs).

Ideally, an XML Database supporting automatic indexing should be able to compile and optimize any standard XQuery using the automatic indexes and then return results at optimal speed.

In practice, this is not always possible. Some queries cannot be optimized. In addition XQuery is a rich and complex language that makes optimization difficult.

For example, a direct exploitation of automatic indexes in Qizx allows optimizing a query such as:

```
//inventor[ .//name = 'John Smith' ]
```

But if the inventor name in the database has not the exact specified form, the query will fail. So we can rewrite it like this:

```
//inventor[ normalized-name(.) = 'john smith' ]
```

where normalized-name() is some function that returns a normalized form of the name of an inventor. It can be as simple as lowercase() or more complex (the inventor name can also appear as a pair of elements first-name, last-name).

Unfortunately, the query compiler in Qizx currently cannot fully optimize this query using indexes only. Therefore it must back off on the predicate and optimize only `//inventor`, which is much slower on a large database.

It is interesting to notice that automatic indexes offer great possibilities of optimizations:

- For example it is possible to apply the function to all index keys and find those that match the normalized name, then merge the corresponding results. It can be costly, but if it is done once and cached in a map, then subsequent queries will execute much faster.
- By building specialized indexes derived from basic indexes, at much lower cost.

Full-text search

Because it is the nature of XML to mix text and data, it seems very natural and useful to offer full-text searches on XML documents.

Full-text capabilities have been standardized in XQuery some years after the core language. The XQuery Full-Text standard [3] offers a very rich set of features, but strangely lacks an essential functionality: how to quickly get the N most relevant documents or nodes for a given full-text query (provided by any full-text search engine).

This can be expressed in XQuery:

```
subsequence( for $hit score $score in ...full-text-query...
             order by $score descending
             return $hit,
             1, N)
```

Yet this is not only embarrassingly cumbersome, but also quite difficult to optimize by a database system.

The only solution for XML Database implementers is to resort to proprietary functions. It seems very desirable to define a standard function that would do the same thing in a simple way.

Faceted search

Another capability that is very useful for real applications but has no support yet in XQuery standards is *faceted search*.

There are many definitions of faceted search (and a myriad patents around it...), but here is an outline of this feature, applicable to XML Databases:

1. A faceted search returns an *Entity*, which is typically an XML element; for example an entity would be `patent, inventor, claim, legal-event`...
2. For each Entity type, a set of *Facets* is defined. Facets are properties of the entity that can be used either for searching or for sorting.

   Each facet has a name and is computed from the Entity, for example as a relative path, but more generally by any function.

   The value can be transformed for presentation to a search user: for example dates can be mapped to a set of months or years.
3. A search specifies an Entity type and a subset of facets with values (or more generally with predicates on values). Other facets can be used for sorting the results.

   Search can also be combined with full-text search and sorted by relevance.
4. For each facet not used in the search, the search function computes the distinct values of that facet over the result set with occurrence counts.

   The facet values can be presented to a user in order to "drill down" by using particular facet values to refine the search.

Standardizing faceted search for XML represents a fairly significant step forward that we believe the XML community can collaborate to achieve.

6. Conclusions

In conclusion, we believe that there is a strong argument to make that NXDbs have been languishing in the Gartner “Trough of Disillusionment” for some time. However relevant research has continued and the surviving products have continued to mature. In the experience of the authors, NXDbs now outperform competing technologies when used in their areas of strength.

Our trials suggest that products which support automatic indexing and effective query optimization are the most likely to be attractive to application architects. Simple and efficient full-text search and faceted search are also highly desirable.

Products with these features will arguably be well placed to take advantage of future market growth.

Bibliography

[1] XQuery 3.0: An XML Query Language. W3C Recommendation, 08 April 2014. http://www.w3.org/TR/xquery-30/

[2] XQuery Update Facility 1.0. W3C Recommendation, 17 March 2011. http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/

[3] XQuery and XPath Full Text 1.0. W3C Recommendation, 17 March 2011. http://www.w3.org/TR/2011/REC-xpath-full-text-10-20110317/

[4] DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Roy Goldman , Jennifer Widom - Stanford University 1997

[5] Faster path indexes for search in XML data Nils Grimsmo, Norwegian University of Science and Technology, Trondheim. 2007 - ISBN: 978-1-920682-56-9

[6] Gartner Hype Cycle. http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp

[7] XML and Databases, R.Bourret. http://www.rpbourret.com/index.htm

[8] Timeline of Database History. http://quickbase.intuit.com/articles/timeline-of-database-history