pg_xnode (extension)

New implementation of XML for PostgreSQL

Antonin Houska

October 26, 2012

XML - Why a new implementation?

PostgreSQL currently relies on libxml2 library. That conforms to the XML standards very well, but has a few drawbacks:

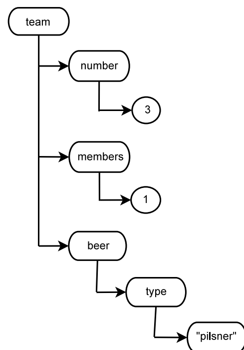
- Plain text is the only format to interchange data between PostgreSQL and libxml2
- Difficult memory management
- Limited control over the behaviour of parser, XPath processor, etc.

Effort can be spent to make the integration more seamless, but the dependency on a separate project would remain.

XML Node as a fundamental concept

- Various programming languages have API to work with the document tree as specified by Document Object Model (DOM).
- The possibility to store just a subtree or a single node might be attractive for application programming.

```
<team number=3 members="1">
  <beer type="pilsner"/>
</team>
```



Data types

xml.node

- Node or subtree retrieved from XML document that can be stored alone (and possibly added to another document later)
- `xml.node_kind()` function shows the actual node kind
- Document fragment is in fact a **parent** of the node(s) it contains (basically a document containing more than one element)

```
      node        |     node_kind
------------------+---------------------
 <a/>             | XML element
 <a><b/></a>      | XML element
 <!--comment-->   | XML comment
 some text        | text node
 <a/><b/>         | document fragment
```

Data types

xml.doc

- Well-formed XML document (That does not imply DTD or XSD validation!)
- Unlike **xml.node** it ensures that all namespaces are bound
- There's an implicit cast between **xml.node** and **xml.doc** in either direction, but no polymorphism

xml.path

- Parsed XPath expression that can be inserted into a table and used later
- Ready for use as soon as fetched (and writable copies of some parts are made)
- Type checking at parse time

```
postgres=# SELECT xml.path('count(true())');
ERROR:  boolean type cannot be cast to nodeset.
check argument 1 of count() function
```

Data types

xml.pathval

- Output of XPath search functios is the primary purpose.
- Binary storage for XPath value, i.e. one of the following: **boolean**, **number**, **string**, **node-set**
- Implicit casts to the appropriate SQL types (bool, numeric, etc.) or to **xml.node**.

XPath evaluation: simple

```
xml.path(xml.path xpath, xml.doc document)
returns xml.pathval
```

```
SELECT xml.path(
'count(/team/beer) div /team/@members',
'<team number="5" members="3">
     <beer type="pilsner"/>
     <beer type="guinness"/>
     <wine type="white"/>
</team>'
);
       path
-------------------
 0.666666666666667
(1 row)
```

XPath evaluation: table-like

```
xml.path(xml.path base, xml.path[] columns,
xml.doc doc)
returns xml.pathval[]
```

```
SELECT xml.path(
'/team/node()',
'{"name()", "@type"}',
'<team number="5" members="3">
     <beer type="pilsner"/>
     <beer type="guinness"/>
     <wine type="white"/>
</team>'
);
      path
-----------------
 beer,pilsner
 beer,guinness
 wine,white
(3 rows)
```

Structure of the XPath expression

```
SELECT xml.path_debug_print (
  '/team[count(beer) div @members >= 1.5]'
);
                   path_debug_print
-------------------------------------------------------
 main expr.: (paths / funcs: 1 / 0, val. type: 3)      +
     <path 1>                                          +
                                                       +
   variables:                                          +
     path: 1                                           +
                                                       +
                                                       +
 <path 0>                                              +
 relative xpath                                        +
 node test:      beer                                  +
                                                       +
                                                       +
 <path 1>                                              +
 absolute xpath                                        +
 node test:      team                                  +
   predicate expr.: (paths / funcs: 1 / 1, val. type: 0)+
     subexpr. implicit: (val. type: 1)                 +
       count(                                          +
         <path 0>                                      +
       )                                               +
       div                                             +
       @members                                        +
     >=                                                +
     1.50                                              +
                                                       +
   variables:                                          +
     path: 0                                           +
     attribute: members                                +
```

Performance of the XPath processor

Text-to-text

```
doc text;
expr text;
result text;
result := xml.path(expr::xml.path, doc::xml.doc)::text;
```

Binary-to-text

```
doc xml.doc;
expr xml.path;
result text;
result := xml.path(expr, doc)::text;
```

Binary-to-binary

```
doc xml.doc;
expr xml.path;
result xml.pathval;
result := xml.path(expr, doc);
```

Performance of the XPath processor

- Documentation the `pg_xnode` (XML DocBook) used, copied to achieve the desired size
- Only execution time of the `xml.path()` function measured, i.e. not that the whole SQL statement
- Size of the document queried: 1, 2, 5, 10, 20, 50 and 100 kB respectively
- XPath query `//sect2` used for all tests. The amount of data returned was always "nearly identical" to the size of the original document

Performance of the XPath processor

Frequency of execution by the PostgreSQL in-core (libxml2) processor considered a unit

Test type	1k	2k	5k	10k	20k	50k	100k
text→text	1.4	1.3	1.1	1.0	0.9	0.9	0.8
binary→text	2.3	2.4	2.6	3.2	3.6	4.3	4.0
binary→binary	3.2	3.4	3.6	4.8	5.5	6.4	5.7

TODO

- Bigger documents (what's the reasonable size of XML?)
- Evaluate memory consumption
- Testing by 3rd party (i.e. not the author)

Custom XPath functions

1. **Implement the logic (Treat all objects as READ ONLY!)**

```
extern void xpathSum(
    XPathExprState exprState,
    unsigned short nargs, XPathExprOperandValue args,
    XPathExprOperandValue result) {

    XPathNodeSet nodeSet = XFUNC_GET_ARG_NODESET(args, 0);
    XMLNodeHdr *nodes = getArrayFromNodeSet(exprState, nodeSet);
    unsigned int i;
    float8 sum = 0.0f;

    for (i = 0; i < nodeSet->count; i++) {
        char *nodeStr = XNODE_GET_STRING(nodes[i]);
        bool isNum = false;
        double numValue = xnodeGetNumValue(nodeStr, false, &isNum);

        if (isNum)
            sum += numValue;
        else {
                result->isNull = true;
                XFUNC_SET_RESULT_NULL(result);
                return;
        }
    }

    XFUNC_SET_RESULT_NUMBER(result, sum);
}
```

Custom XPath functions

2. Register the function

```
XPathFunctionData xpathFunctions[] = {

    ...

    {
        XPATH_FUNC_SUM,             /* Function ID */
        "sum",                      /* Function name*/
        1,                          /* Number of arguments */
        {XPATH_VAL_NODESET},        /* Argument types */
        false,                      /* Variadic? */
        {.args = xpathSum},         /* The implementation */
        XPATH_VAL_NUMBER,           /* Return type */
        false                       /* Requires context? */
    }
}
```

Custom XPath functions

3. Use it

```
SELECT xml.path(
'sum(//@price)',
'<team number="5" members="3">
    <beer type="pilsner" price="0.8"/>
    <beer type="guinness" price="2.8"/>
    <wine type="white" price="2.3"/>
</team>');
 path
------
 5.9
```

Future work – Ideas

- Other aggregate functions, e.g. `avg()`, `min()`, `max()`, ...
- Set operations, e.g. `sort()`, `unique()`, ...
- Make sure that function can be used as operator
- Containment operators: `<@`, `@>`

Document Object Model

Add node to a document

```
xml.add (
  xml.doc,         -- Source document
  xml.path,        -- Add the node here
  xml.node,        -- Node to be added
  xml.add_mode     -- How it should be added
) returns xml.doc
```

Addition mode

Position of the new node, relative to the target

- 'b' – before
- 'a' – after
- 'i' – into
- 'r' – replace

Document Object Model

Example: Add a new node INTO the target node

```
SELECT xml.add (
'<day date="2012-10-26"/>',
'/day',
'<talk subj="pg_xnode"/>',
'i'
);
                           add
----------------------------------------------------------
 <day date="2012-10-26"><talk subj="pg_xnode"/></day>
```

Document Object Model

Example: Add a new node AFTER the target node

```
SELECT xml.add(
'<day date="2012-10-26"><talk sub="pg_xnode"/></day>',
'/day/talk',
'<break/>',
'a'
);
                              add
-------------------------------------------------------------
 <day date="2012-10-26"><talk sub="pg_xnode"/><break/></day>
```

Document Object Model

Remove node from a document

```
xml.remove (
  xml.doc,          -- Source document
  xml.path          -- Node to be removed
) returns xml.doc
```

Example: All occurrences of the target path are removed

```
SELECT xml.remove (
'<dba id="6">
  <phone home="777815361" work="251050128"/>
</dba>',
'//text()'
);
                              remove
--------------------------------------------------------------
 <dba id="6"><phone home="777815361" work="251050128"/></dba>
```

Document Object Model

Get array of child nodes

```
xml.children(
  xml.node
) returns xml.node[];
```

Example:

```
SELECT xml.children (
'<dba id="6"><phone home="777815361" work="251050128"/></dba>'
);
                       children
------------------------------------------------------
 6,"<phone home="777815361" work="251050128"/>"
```

Document Object Model – Access the nodes in C

```
static unsigned int
countDescendants(XMLNodeHdr node) {
  unsigned int result = 1;

  if (XNODE_IS_COMPOUND(node)) {
    XMLCompNodeHdr compNode = (XMLCompNodeHdr) node;
    XMLNodeIteratorData iterator;
    XMLNodeHdr child;

    initXMLNodeIterator(&iterator, compNode, true);

    while ((child = getNextXMLNodeChild(&iterator))
      != NULL) {
      result += countDescendants(child);
    }
  }
  return result;
}
```

Document Object Model – Access the nodes in C

```
PG_FUNCTION_INFO_V1(descendants);

Datum
descendants(PG_FUNCTION_ARGS) {
  xmlnode nodeStorage = (xmlnode) PG_GETARG_VARLENA_P(0);
  XMLNodeHdr rootNode = XNODE_ROOT(nodeStorage);
  unsigned int result = countDescendants(rootNode);
  PG_RETURN_INT32(result);
}
```

```
CREATE FUNCTION descendants(node)
  RETURNS int
  as 'MODULE_PATHNAME', 'descendants'
  LANGUAGE C
  IMMUTABLE
  STRICT;
```

Document Object Model – Access the nodes in C

```
SELECT xml.descendants(
'<team number="5" members="3">
    <beer type="pilsner" price="0.8"/>
    <beer type="guinness" price="2.8"/>
    <wine type="white" price="2.3"/>
</team>'
);
 descendants
-------------
          16
```

Note: This is just an example, we don't really need such a function. XPath expression

```
count(//node()) + count(//@*)
```

can get the number of descendants too.

Convert table data to XML using node template

```
CREATE TABLE templates (
        name varchar NOT NULL PRIMARY KEY,
        data xml.xnt NOT NULL
);
```

```
INSERT INTO templates
VALUES (
'talk',
'<xnt:template
  xmlns:xnt="http://www.pg-xnode.org/xnt"
  preserve-space="true"
><talk speaker="{$s}" room="{$r}" time="{$t}">
  <xnt:copy-of expr="$cnt"/>
</talk></xnt:template>');
```

Convert table data to XML using node template

```
SELECT xml.node(
  t2.data, '{"cnt", "s", "r", "t"}',
  (t1.subject, t1.speaker, t1.room,
    to_char(t1.start, 'FMDy HH:MI')))
FROM talks t1, templates t2
WHERE t2.name='talk';
```

```
                              node
------------------------------------------------------------
 <talk speaker="@fuzzychef" room="Vltava" time="Thu 11:50">+
   Elephants and Windmills                                 +
 </talk>
 <talk speaker="@jkatz05" room="Vltava" time="Thu 09:30">  +
   Marketing PostgreSQL                                    +
 </talk>
 <talk speaker="@c2main" room="Seine" time="Thu 09:30">    +
   How fast is PostgreSQL?                                 +
 </talk>
```

Aggregate XML nodes – xml.fragment()

```
SELECT xml.fragment(
  xml.node(
    t2.data, '{"cnt", "s", "r", "t"}',
    (t1.subject, t1.speaker, t1.room,
      to_char(t1.start, 'FMDy HH:MI'))
)) FROM talks t1, templates t2
WHERE t2.name='talk';
```

```
                              fragment
---------------------------------------------------------------
<talk speaker="@fuzzychef" room="Vltava" time="Thu 11:50">
  Elephants and Windmills
</talk><talk speaker="@jkatz05" room="Vltava" time="Thu 09:30">
  Marketing PostgreSQL
</talk><talk speaker="@c2main" room="Seine" time="Thu 09:30">
  How fast is PostgreSQL?
</talk>
(1 row)
```

XML Node Template (XNT) – Summary

- Unlike the SQL constructs `xmlelement()`, `xmlattributes()`, etc., the XNT is data
- Easier to read, especially if the resulting document is complex
- Can be used to generate one or multiple nodes, as opposed to the whole XML document
- Everything (including expressions) is stored in binary format. Conversion to text only takes place if user (application) needs it.
- TODO: tags `xnt:if`, `xnt:for-each`, etc.

Use XHTML to generate web pages

```
<xnt:template xmlns:xnt="http://www.pg-xnode.org/xnt">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>pg_xnode</title>
    <link rel="stylesheet" type="text/css"
      href="layout.css"/>
  </head>
  <body>
    <div class="main">
      <div class="menu">
        <xnt:copy-of expr="$menu"/>
      </div>
      <div class="page">
        <xnt:copy-of expr="$body"/>
      </div>
  </body>
</html>
</xnt:template>
```

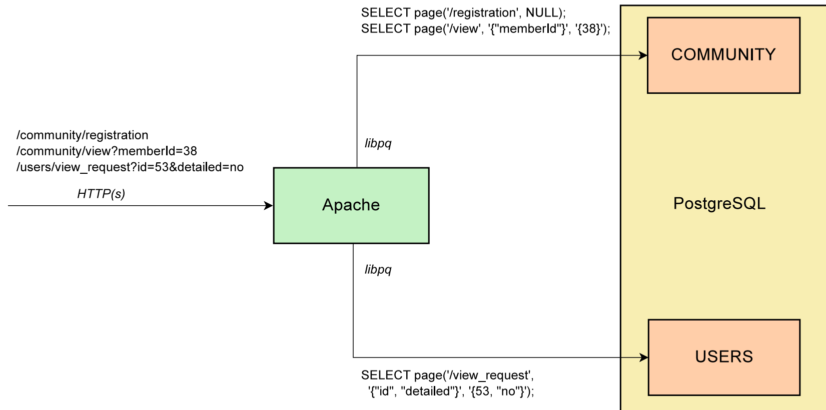
Use XHTML to generate web pages

Get the static content to be uploaded to server

```
psql -d my_web -tAc "select page('/')" > index.html
psql -d my_web -tAc "select page('/download')" > download.html
psql -d my_web -tAc "select page('/doc')" >doc.html
```

Complete example: https://github.com/pg-xnode/web

Proposal: SQL-only web applications



Proposal: SQL-only web applications

Apache module mod_pg

1. Checks the application context and decides which database/schema should render it
2. Connects to that database and authenticates
3. Uses `page()` function to generate the XHTML code for given page URI and parameters
4. Returns the page to user. If it's not one that changes often, cache it

Convert XML to JSON

```
SELECT
xml.node(
  '<xnt:template xmlns:xnt="http://www.pg-xnode.org/xnt">
    <talk subject="{$s}" room="{$r}" time="{$t}"/>
  </xnt:template>', '{s, r, t}',
  (t.subject, t.room, to_char(start, 'FMDy HH:MI'))
)::json
FROM talks t;
```

```
                                         node
-----------------------------------------------------------------------------------------
 {"talk": {"@subject": "Elephants and Windmills", "@room": "Vltava", "@time": "Thu 11:50"}}
 {"talk": {"@subject": "Marketing PostgreSQL", "@room": "Vltava", "@time": "Thu 09:30"}}
 {"talk": {"@subject": "How fast is PostgreSQL?", "@room": "Seine", "@time": "Thu 09:30"}}
```

Conclusions

Future work

- Refine conformity to the XML standard
- New developers, testers, ideas ...
- Implement other related standards (XSD, XSLT, ...) ?

Resources

- Website: http://www.pg-xnode.org
- Repository: https://github.com/pg-xnode/extension/

Thanks!

(http://2012.pgconf.eu/feedback/)