pg_xnode - user documentation

1. Introduction

pg_xnode is PostgreSQL extension. The purpose is to

- provide the PostgreSQL database server with ability to query and modify XML documents in an efficient and convenient way.
- introduce a set of data types to represent XML document (DOM) tree, its parts (nodes, subtrees) and XPath expressions.
- provide an extensible platform for non-traditional approaches to XML storage and processing.

pg_xnode doesn't rely on any third-party library.

> **Caution**
>
> The current version is not ready for use in production environments. The extension may be subject to significant changes, including those of the binary data format. Migration from versions lower than 1.0 won't be supported.
>
> In addition, comprehensive testing has to be performed before 1.0 gets released.

2. Installation

Make sure PostgreSQL database server is installed (the current version of pg_xnode has been developed on top of PostgreSQL 9.1).

1. Extract the archive.
2. **`make`**
3. **`sudo make install`** [1]
4. Start the PostgreSQL cluster.
5. Optional verification:

   `make installcheck`
6. Connect to your database.
7. **`CREATE EXTENSION xnode;`** [2]

3. Data types

pg_xnode defines data types to store various XML-related objects in binary form. This helps to avoid (repeated) unnecessary parsing and serialization of those objects and thus provides potential for efficient data processing.

3.1. `xml.node`

`node` type represents a node of XML document tree. Following are the valid node kinds: *document*, *document type declaration (DTD)* [3], *element*, *attribute*, *comment*, *character data (CDATA)*, *processing instruction (PI)* and *text node*.

Special node type *document fragment* exists to represent a set of nodes. To add document fragment to some location in the tree in fact means to add its children. The document fragment node is actually just a kind of node container.

Example:

```
CREATE TABLE nodes (
        data    xml.node
);

INSERT INTO nodes
VALUES ('<element/>'), ('<!--comment-->'), ('plain text'), ('<a/><b/>');

SELECT *
FROM nodes;


          data
----------------
 <element/>
 <!--comment-->
 plain text
 <a/><b/>
(4 rows)
```

3.2. `xml.doc`

`doc` represents well-formed XML document. [4]

If namespace prefixes are used, each must be bound, i.e. each namespace used must be declared at the appropriate level of the document tree. This restriction does not apply to `xml.node` type.

Example:

```
CREATE TABLE ecosystems (
 id int,
 data xml.doc
);

INSERT INTO ecosystems VALUES (1,
 '<zoo city="Wien"><penguin name="Pingu"/><elephant name="Sandeep"/></zoo>');

INSERT INTO ecosystems VALUES (2,
 '<africa><hipo weight="2584"/><elephant age="28"/></africa>');

INSERT INTO ecosystems VALUES (3,
 '<zoo city="Dublin"><rhino/><giraffe/><elephant name="Yasmin"/></zoo>');
```

3.3. `xml.pathval`

`pathval` represents a result of `xml.path()` functions (see bellow). Depending on particular XPath expression, the resulting `pathval` value contains binary value of the appropriate XPath data type: number, string, boolean, `node`.

3.4. `xml.xnt`

`xnt` stands for *XML Node Template (XNT)*. Such a template may be undersotood as incomplete `node`. Instead of nodes, it contains *parameters* at some locations of the tree.

The template gets converted to `node` when valid `pathval` values are substituted for all parameters. `xml.node(xnt, text[], record)` is the function to perform the substitution.

See Section 5 for more information and some examples.

4. Functions

4.1. `xml.node_kind()`

```
xml.node_kind(xml.node node) returns text
```

Returns textual expression for the kind of `node`, e.g. element, comment, etc.

Example:

```
SELECT xml.node_kind(data)
FROM nodes;
```

```
           node_kind
-------------------------
 XML element
 XML comment
 text node
 document fragment
(4 rows)
```

4.2. `xml.element()`

```
xml.element(text name, text[][2] attributes, xml.node nested) returns xml.node
```

Returns XML element having name equal to `name` and attributes listed in `attributes` array: one row per attribute where the first column is attribute name and the second column is the attribute value. `nested` parameter may contain other node(s) to be nested in the new XML element.

Example:

```
SELECT xml.element('city', '{{"name", "Tokyo"}, {"population", 13185502}}', NULL);
```

```
                  element
-------------------------------------------
 <city name="Tokyo" population="13185502"/>
 (1 row)
```

Example:

```
SELECT xml.element('book', NULL, xml.element('chapter', '{{"id", 1}}', 'some text'));


                      element
-------------------------------------------------
 <book><chapter id="1">some text</chapter></book>
(1 row)
```

4.3. `xml.children()`

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

Returns array of `node`'s child nodes or empty array if `node` is a leaf node.

Only node at the immediate lower level of the tree is considered a child. Recursive calls of the function on results has to be used to get nested nodes from the deeper levels (i.e. *descendants*).

Example:

```
SELECT xml.children('<house><cellar><wine/></cellar><floor/><garret/></house>');
                   children
-----------------------------------------------
 {<cellar><wine/></cellar>,<floor/>,<garret/>}
(1 row)
```

4.4. `xml.path()` - scalar

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

Returns result of XPath expression (passed as `xpath`) applied to `document`. If `xpath` is a location path and there are multiple qualifying nodes in the document then the returned`xml.pathval` contains a document fragment containing all the qualifying nodes.

Example:

```
SELECT xml.path('//elephant', e.data)
FROM ecosystems e;
```

```
         elephants
---------------------------
 <elephant name="Sandeep"/>
 <elephant age="28"/>
 <elephant name="Yasmin"/>
(3 rows)
```

4.5. `xml.path()` - vector

```
xml.path(xml.path basePath, xml.path[] columnPaths, xml.doc doc) returns xml.pathval[]
```

Returns table-like output. For each occurrence of `basePath` in `doc` an array of `xml.pathval` values is returned where n-th element is value of relative XPath expression passed as n-th element of `columnPaths`. All values of `columnPaths` array are relative to `basePath`.

Example:

```
SELECT xml.path('/zoo',
 '{"@city", "count(elephant)", "count(penguin)"}', e.data) as counts
FROM ecosystems e;
```

```
           counts
---------------------------
 {Wien,1.000000,1.000000}
 {Dublin,1.000000,0.000000}
(2 rows)
```

4.6. `xml.add()`

```
xml.add(xml.doc doc, xml.path path, xml.node newNode, xml.add_mode mode) returns xml.doc
```

Adds `newNode` to all occurrences of `path` in `doc`. Depending on `mode` value, the new node can be added before (`b`) or after (`a`) the *target node* (where target node is the the node `path` points at).

If target node kind is element and `mode` is `i`, then the new node is added into that element. If that element is not empty, the new node is added as the last.

If `mode` is `r` then the target node is replaced with `newNode`.

A document is returned where all the additions have been done as required in the input parameters.

Example:

```
UPDATE ecosystems e
SET data=xml.add(e.data, '/africa', '<gorilla/>', 'i')
where e.id=2;

SELECT data FROM ecosystems e where e.id=2;


                                    data
-------------------------------------------------------------------------
 <africa><hipo weight="2584"/><elephant age="28"/><gorilla/></africa>
(1 row)
```

4.7. `xml.remove()`

```
xml.remove(xml.doc doc, xml.path path) returns xml.doc
```

Removes all occurrences of `path` from `doc`.

A document is returned where all the removals have been done as required in the input parameters.

Example:

```
UPDATE ecosystems e
SET data=xml.remove(e.data, '/zoo/elephant')
WHERE xml.path('/zoo', data);

SELECT xml.path('count(/zoo/elephant)', data)
FROM ecosystems e
WHERE xml.path('/zoo', data);


   path
----------
 0
 0
(2 rows)
```

4.8. `xml.fragment()` - aggregate

```
xml.fragment(node) returns xml.node
```

Aggregate function to turn group of nodes into a single node - a document fragment. (What document fragment means is explained in Section 3.1.) The function can be combined with XNT templates, to turn relational data into XML. See Section 5.3 for an example.

4.9. `xml.node()`

```
xml.node(xml.xnt template, text[] paramNames, record paramValues)
returns xml.node
```

Constructs a new `node` out of `template` by substituting parameters.

`paramNames` is an array of all parameter names that the template contains, while `paramValues` is a row containing the corresponding values. Ordering is crucial to associate values and parameters correctly. For example, the first column in `paramValues` is understood to be the value of the parameter whose name is the first item of `paramNames` array.

Each parameter value is cast to the most suitable XPath data type. For example `varchar` SQL type is converted to `string` XPath type, `integer` SQL type is converted to `number` XPath type, etc.

See Section 5 for examples.

4.10. `xml.node_debug_print()`

```
xml.node_debug_print(xml.node node) returns text
```

Shows tree structure of `node` tree. Instead of content, position (offset) of each node in the binary value is displayed, as well as its size.

4.11. `xml.path_debug_print()`

```
xml.path_debug_print(xml.path xpath) returns text
```

Returns text string showing structure of XPath expression passed as `xpath`.

5. XML Node Template (XNT)

XML Node Template (XNT) is a means to turn a table row (i.e. instance of `record` PostgreSQL type) into a node of XML document tree (i.e. instance of `node`). If acompanied with `fragment()` function, the template can be used to create a single XML node or document out of many table rows.

The XNT syntax and its tags strongly resemble template (stylesheet) for XSL Transformations (http://http://www.w3.org/TR/xslt), however there is one fundamental difference: there's no `select` attribute in XNT. The reason is that XNT processing function (i.e. `node()`) does not receive any input XML document to browse in and to modify (by applying various templates). The only thing the XNT processing function does is to substitute particular values for the template parameters. Another significant difference is that XSLT template may define many templates, whereas XNT template only has one - located in the root of the tree.

5.1. Expressions and Parameters

XNT template uses (XPath) expressions in the same way as XSLT template does. There's however difference in the expression itself: XNT template does not accept location paths, whether absolute or relative. The reason is that - as mentioned above - there's no source XML document the location path could reference.

Besides string or numeric literals, expression can contain parameters. Parameter starts with dollar sign and its usage is identical to that in XSLT templates.

Examples of expressions accepted by XNT:

```
2 * $pi * $r
count($nodes)
contains($sentence, $word)
```

Expressions NOT accepted by XNT:

```
contains(/article/sect1/title/text, $word)
name(/article/sect1/@id)
```

Expression in XNT template also corresponds to XSLT in the way it's used to construct attribute value: it has to be enclosed in curly braces. For example:

```
<ticket price="{$p}" destination="{$d}" departure="{$month}/{$day}/{$year}"/>
```

5.2. XNT Tags

Besides complete nodes or nodes where just attribute value is parametrized, the XNT template may contain some of the following special tags. As mentioned earlier in this chapter, the XNT tags are in general similar to XSLT tags and sometimes even identical. However they never have `select` attribute to get XML node (node-set) from input XML document (because XNT does not know the concept of input XML document). Instead, if XNT tag needs an XML node to work with, it must always receive it as template parameter.

> **Note:** It's necessary for the `xnt` namespace to be bound to `http://www.pg-xnode.org/xnt` Any tag failing to do so won't be considered XNT tag and parser will report error where XNT tag is the only option (typically `xnt:template`).

5.2.1. `xnt:template`

Mandatory root tag of the template. The template document must contain exactly one `xnt:template`.

Currently this tag has one (optional) attribute: `preserve-space`. If it's set to `true` then whitespace-only descendant (i.e. both direct and indirect children of the template) text nodes are copied to the resulting node. Otherwise they are discarded.

5.2.2. `xnt:copy-of`

This tag has a single attribute `expr`. XNT expression is expected as the attribute value.

If the expression evaluates to boolean, number or string XPath type (see Section 3.3), then the value is cast to string and a text node containing is substituted for the `xnt:copy-of` node. If the expresion evaluates to node, the resulting node itself is used for the substitution. (Special case is that document fragment is the result. In such a case only child nodes of the fragment node are inserted.)

Example:

```
select xml.node(
'<xnt:template xmlns:xnt="http://www.pg-xnode.org/xnt">
   <chapter>
     <xnt:copy-of expr="$paragraphs"/>
   </chapter>
</xnt:template>',
'{paragraphs}',
ROW('<para>Just a few words.</para><para>The end.</para>'::xml.node)
);
```

```
                                        node
-------------------------------------------------------------------------
 <chapter><para>Just a few words.</para><para>The end.</para></chapter>
(1 row)
```

5.2.3. `xnt:element` and `xnt:attribute`

`xnt:element` has a single attribute `name`. String (possibly containg string-returning expression(s)) is expected as the value. If the string represents a valid XML name, a new XML element having this name is substituted for the `xnt:element` node.

`xnt:attribute` has 2 attributes: `name` and `value`. Both expect string value and these values are used as name and value of a new XML attribute respectively. (Similar to `xnt:element`, value of `name` must be a valid XML name.) The new XML attribute is substituted for the `xnt:attribute` tag.

Example:

```
TABLE trees;

  name   | height
---------+--------
 sequoia |  115.5
 palm    |     60
 ash     |   99.6
(3 rows)


INSERT INTO trees_xml
select xml.node(
'<xnt:template xmlns:xnt="http://www.pg-xnode.org/xnt">
   <xnt:element name="{$name}">
     <xnt:attribute name="height" value="{$height}"/>
   </xnt:element>
</xnt:template>',
'{name, height}',
ROW(name, height))
FROM trees;


TABLE trees_xml;


               node
---------------------------
 <sequoia height="115.5"/>
 <palm height="60"/>
```

```
 <ash height="99.6"/>
(3 rows)
```

5.3. Example: Aggregate Multiple Rows Into a Single XML Node

The example in Section 5.2.3 shows how to generate one XML node per table row. Sometimes it may be desired to create a single XML node out of multiple rows (and possibly cast it to XML document). This is the typical use case for `xml.fragment()` function.

```
SELECT xml.node(
'<xnt:template xmlns:xnt="http://www.pg-xnode.org/xnt">
   <forest>
     <xnt:copy-of expr="$trees"/>
   </forest>
</xnt:template>',
'{trees}',
ROW(xml.fragment(tree)))
FROM trees_xml;
```

```
                                                    node
-------------------------------------------------------------------------------
 <forest><sequoia height="115.5"/><palm height="60"/><ash height="99.6"/></forest>
```

5.4. XNT and XHTML

As XHTML is understood by XML parsers, it's possible to use pg_xnode and its templates (XNT) to generate XHTML web pages. Web site of the pg_xnode project (https://github.com/pg-xnode/web) uses this approach and as such it can be used for reference.

A. Release Notes

The following is a summary of new features each version brought. Particular bug fixes are not listed here until stable release is approached.

A.1. Release 0.6.0

The first release to be published on www.pgxn.org

A.2. Release 0.6.1

- Added XPath core library functions `true()`, `false()`, `last()`, `concat()`, `boolean()`, `number()` and `string()`
- Added DOM function `xml.children()`.
- Added casts from `xml.pathval` to Postgres types.

A.3. Release 0.7.0

- Function `xml.element()`
- XPath core library function `starts-with()`
- XPath operators: `|`, `+`, `-` (both binary and unary), `*`, `div`, `mod`.

A.4. Release 0.7.1

- Namespace validation.
- XML Node Template (XNT)
- `xml.fragment()` aggregate function.

A.5. Release 0.7.2

- Ensured that all data types are correctly aligned.
- Added XPath functions `local-name()` and `sum()`.

Notes

1. If *PATH* environment variable doesn't seem to contain pg_xnode, specify the full path, e.g. **`sudo env PG_CONFIG=/usr/local/pgsql/bin/pg_config make install`**
2. If earlier version already exists where version number <=1.0, then the extension must be dropped (with `CASCADE` option) and re-created. If the `CREATE` command step is skipped in such a case, then data in obsolete format remain in the database and pg_node's behaviour becomes undefined.

   This only applies to pre-releases. Migration functionality will be delivered for versions > 1.0;
3. The current version of pg_xnode does recognize syntax of the DTD, but does not perform any validation.
4. Unlike other node kinds (comment, element, etc.) there's no polymorphism between `xml.node` and `xml.doc`. That is, functions do not consider `xml.doc` a special case of `xml.node`. However an implicit `xml.node:xml.doc` cast exists for cases where conversion to a well-formed XML document does make sense.