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. Connect to your database.
5. **`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`

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

Special node type *document fragment* exists to represent a set of nodes.

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`

`xml.doc` represents well-formed XML document. [3]

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`

`xml.pathval` represents a result of `xml.path()` functions (see bellow). Depending on the XPath used for a search, the `xml.pathval` value contains value of one of the following types: number, string, boolean, `xml.node`.

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.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)
```

4.3. `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)
```

4.4. `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)
```

4.5. `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)
```

4.6. `xml.remove()`

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

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

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)
```

4.7. `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.

4.8. `xml.path_debug_print()`

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

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

A. Release Notes

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.
- Fixed several bugs and eliminated some inefficiencies

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 instead, 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. 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.