Implementace funkce XMLTABLE v PostgreSQL

Pavel Stěhule @2017

Motivace

- **Aktuálně chybí jednoduchý způsob, jak parsovat XML dokumenty v PostgreSQL**
  - S použitím untrusted funkcí (XML reader API) - pracné, a ne každý má dostatečné zkušenosti s Perlem, Pythonem (běžné jazyky jsou untrusted)
  - Funkce XPath - opakované volání může být pomalé, i poměrně jednoduché transformace nepřehledné

Motivace - funkce XPath

```
<data>
   <user id="1">
     <jmeno>Pavel</jmeno>
     <prijmeni>Stehule</prijmeni>
   </user>
   <user id="2">
     <jmeno>Tomas</jmeno>
     <prijmeni>Vondra</prijmeni>
   </user>
 </data>
```

Motivace - funkce XPath

```
SELECT d[1]  id, d[2]  jmeno, d[3]  prijmeni
  FROM
  (SELECT xpath('/user/@id|/user/jmeno/text()|'
                '/user/prijmeni/text()',x) d
    FROM
     (SELECT unnest(xpath('/data/user', a)) x
        FROM foo) s) s2;
```

SQL/XML

- **ANSI SQL 2003**
  - funkce XMLELEMENT, XMLFOREST
- **příprava XMLTABLE 2004**
- **ANSI SQL 2006 obsahuje XMLTABLE**
- **PostgreSQL 8.3 (2008)**
  - podpora SQL/XML
- **PostgreSQL 10 (2017)**
  - podpora XMLTABLE

Motivace - funkce XMLTABLE

```
SELECT xmltable.*
  FROM foo,
       LATERAL xmltable('/data/user'
                        PASSING a
                        COLUMNS id int PATH '@id',
                                jmeno text,
                                prijmeni text);
```

id	jmeno	prijmeni
1	Pavel	Stehule
2	Tomas	Vondra

(2 rows)

Výhody

- Zjednodušení zápisu
- Efektivní zpracování dostupné všem uživatelům
- Zlepšení shody se standardem
- Implementace funkce občas požadované při migraci z Oracle

Pro detaily implementace

- **Standard - nepokrývá všechny implementační detaily - málo čitelný**
- **Oracle, DB2 - potřeba odstínit specifika, občas protichůdná implementace - Oracle v detailu nesplňuje standard (byť je autorem této části)**
- **Je nutné respektovat specifika Postgresu - nová funkce musí zapadat do kontextu celé db - ANSI/SQL, Oracle - XML je blob, v PostgreSQL XML je nativní typ nevyžadující speciální transformaci**

Implemetace

Jak by mohla tato funkce vypadat v aktuální verzi PostgreSQL?

Implementace

```
CREATE OR REPLACE FUNCTION fx(a xml,
                              rowf text,
                              colf text[])
RETURNS SETOF record as $$
BEGIN END
$$ LANGUAGE plpgsql;

SELECT *
  FROM foo,
       LATERAL fx(a,
                  '/data/user',
                  ARRAY['@id','jmeno','prijmeni'])
                AS (id int,
                    jmeno text, prijmeni text);
```

Implementace v klasické funkci

- **Není ve shodě se standardem**
- **Nelze jednoduše nastavit defaultní hodnoty**
- **Zápis je roztahaný - definice typů mimo volání funkce**

- **+ lze napsat jako extenzi**
- **+ pravděpodobně plnohodnotná implementace bez zásahu do Postgresu**

Nativní implementace

- konzistentní úspornější čistý zápis s možností definovat defaultní hodnoty a podmínky NOT NULL
- podpora standardu
- pravděpodobně velice dobrý výkon

- - poměrně velký patch .. cca 100-150KB

Možné komplikace při implementaci

- **libxml2 - plochá učící křivka, málo příkladů**
  - odladěný software, ale občas je nutné se podívat do zdrojáků
- **dvě různé správy paměti**
  - PostgreSQL palloc
  - libxml2
- **nezbytné konverze mezi libxml2 a Postgresem**
  - xmlChar <=> char <=> text
- **generické řešení - podobná funkce JSON_TABLE**

Generování XML - pro úplnost

```
SELECT xmlelement(
           NAME data,
           xmlagg(xmlelement(
                      NAME user,
                      xmlattributes(id),
                      xmlforest(jmeno, prijmeni))))
  FROM foo_x;
```

Implementace - části k modifikaci

- **parser - stávající verze neumožňují definovat výstupní sloupce uvnitř funkce**
- **executor - je nutné někde spočítat výsledek**

Postgres jako překladač SQL

- **Překladač SQL do AST (parser)**
- **Transformace AST (analyzer/rewriter)**
- **Generátor variant (cest) prováděcího plánu (planner)**
- **Interpret prováděcích plánů (executor)**
- **Executor implementuje operace nad relacemi (join, agg) a získáva data z datových zdrojů - halda, různé typy indexů**

- **Programování PostgreSQL je podobné programování překladače programovacího jazyka.**

Postgres jako překladač SQL

- **Při správném použití je režie SQL zanedbatelná**
  - práce s IO, zajištění konzistence (zámky)
- **I při nesprávném použití je režie SQL obvykle zanedbatelná (pozor na ISAM přístup).**
- **Postgres má interní ISAM API - nikdo nenapsal interface, kterým by se toto API zpřístupnilo uživatelům**
  - preferuje se komfortní SQL
  - až na vyjímky zanedbatelný výkonostní benefit
  - ohledně výkonu není snaha konkurovat zavedeným key/value databázím (ACID versus CAP)

Historie - Lisp v PostgreSQL

- Původně byl optimalizátor v Lispu
- Rychlost dostatečná
- Oceňovaná rychlost vývoje - interně je zpracování SQL dotazu několik transformací stromů (a následně evaluace stromu)
- Odstraněno kvůli problémům s integrací správy paměti v C a GC v Lispu
- Pozůstatek - intenzivně používaná struktura node

Historie - Lisp v PostgreSQL

```
typedef struct Expr
{
	NodeTag	type;
}Expr;

typedef struct Param
{
	Expr		xpr;
	int		paramid;
	...
}

-- Param lze bezpecne pretypovat na Expr
Expr *ex = (Expr *) par;
```

Historie - Lisp v PostgreSQL

```
-- funkce pro praci se seznamy
List *l = NIL;
l = make_list1(expr);
l = lappend(l, expr);

-- iterace
ListCell *lc;
foreach(lc, l)
{
   Expr *x = (Expr *) lfirst(lc);

   ...
}
```

PostgreSQL Parser

- **Používá překladač gramatik bison**
- **Gramatika musí být jednoznačná (bez kolizí)**
- **Pokud možno bez nových rezervovaných klíčových slov (nerezervovaná klíčová slova nejsou problém)**
- **První pravděpodobný blocker**

Gramatika pro XMLTABLE

```
xmltable:
            XMLTABLE '(' c_expr xmlexists_argument ')'
            | XMLTABLE '(' c_expr xmlexists_argument COLUMNS xmltable_column_list ')'
            | XMLTABLE '(' XMLNAMESPACES '(' xml_namespace_list ')' ','
                c_expr xmlexists_argument ')'
            | XMLTABLE '(' XMLNAMESPACES '(' xml_namespace_list ')' ','
        ;

xmltable_column_list: xmltable_column_el                    {  $$ = list_make1($1); }
            | xmltable_column_list ',' xmltable_column_el   {  $$ = lappend($1, $3); }
        ;

xmltable_column_el:
            ColId Typename
            | ColId Typename xmltable_column_opt_list
            | ColId FOR ORDINALITY
        ;
```

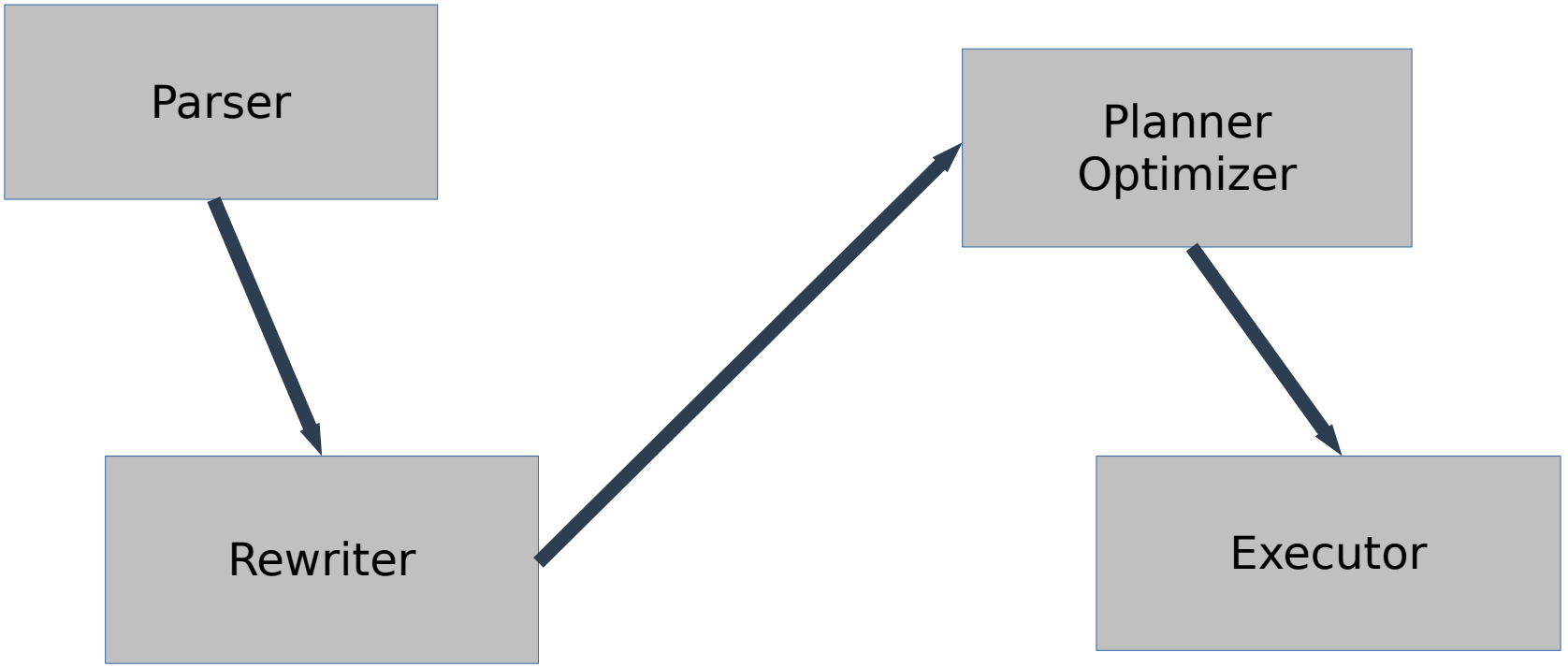
Výkonná část

- **Implementace jako variace SRF funkce**
  - výrazně kratší kód - mírné znásilnění SRF API
  - část interpretu výrazů
- **Implementace jako executor scan node**
  - generická čistá implementace
  - výrazně delší kód + zásahy (většinou triviální/mechanické) do planneru

Postup

- **POC - ukázat, že to lze**
  - nujsurovější možná forma
- **Kompletní patch**
  - včetně dokumentace
  - a testů
- **Commitfest patch**
  - začišťování dokumentace a komentářů
  - rozšiřování testů
  - pročišťování kódu

Query pipeline



Postup

1. **funkční parser,**
2. **funkční parser, NULL executor (0 řádek)**
   - propojení obou komponent
   - musí fungovat vytvoření a aplikace dynamického compozitního datového typu - TupleDesc
3. **funkční parser, konstantní executor, fixní typy**
4. **funkční parser, funkční executor**
5. **regresní testy**
6. **dokumentace**

PostgreSQL je komunitní software

- **občas o váš kód nejeví nikdo zájem**
  - může se jednat opravdu o hloupost
  - může odradit rozsahem, komplikovaností
  - přínosy nemusí být zřejmé
- **vysvětlovat, komunikovat, být trpělivý, asertivní ale respektovat, že ostatní mohou mít jiný názor**
- **některé patche potřebují uzrát**
  - podpora XML - 3 roky po prototypu
  - GROUPING SETS - 10 let po prototypu
  - psql \if command - 10 let po prototypu

Možné pokračování

- **Podpora JSONu**
  - SQL/JSON
  - funkce JSON_TABLE - v Postgresu je nutné uvažovat i jsonb
  - složitější - umožňuje zanoření, pracuje se s poli
  - čeká se na implementaci JSONPath
    - chybí knihovna jako je libxml2 (jsonb je proprietární)

JSON_TABLE

```
-- design neni konzistentni s xmltable()

SELECT * FROM JSON_TABLE( JSON_VAR,
    '$'
    COLUMNS(
         first VARCHAR(10) PATH 'lax $.name.first',
         last VARCHAR(10) PATH 'lax $.name.last' ,
         NESTED '$.phones[*]'COLUMNS (
             "type" VARCHAR(10),
             "number" VARCHAR(10)
         )
    )
) as t;
```