Greenplum 对 JSON 的支持

1 JSON 与 JSONB 概述

1.1 JSON 的概述

JSON 作为结构化的数据，目前越来越受到开发者的爱戴，它简单灵活易于理解。是作为储存数据的一种比较使用的一种格式，greenplum 最新版本已经很好的支持了 JSON 和 JSONB 类型的数据

参考资料:https://hashrocket.com/blog/posts/faster-json-generation-with-postgresql#how-to
Greenplum 官网介绍:https://gpdb.docs.pivotal.io/530/admin_guide/query/topics/json-data.html

1.2 JSONB 的概述

JSONB 同时属于 JSON(JavaScript Object Notation)数据类型,存储的是分解的 binary 格式数据,查询时不需要再次解析，效率非常高。缺点是在写入数据时需要转换为 binary 格式的数据，速度相对会慢一些。

1.3 JSON 与 JSONB 的区别

1、json 储存的是文本格式的数据，jsonb 储存的是 binary 格式的数据。
2、json 插入速度快，查询速度慢，原因是处理函数必须在每次执行时重新解析该数据。jsonb 插入速度慢，而查询速度快，原因是 jsonb 数据被存储在一种分解好的二进制格式中，因为需要做附加的转换，它在输入时要稍慢一些。但是 jsonb 在查询数据时快很多，因为不需要重新解析。
3、json 储存的数据是对数据的完整拷贝，会保留源数据的空格/重复键以及顺序等，如果一个值中的 JSON 对象包含同一个键超过一次，所有的键/值对都会被保留。而 jsonb 在解析时会删除掉不必要的空格/数据的顺序和重复键等，如果在输入中指定了重复的键，只有最后一个值会被保留。

2 JSON 与 JSONB 常用操作符与函数

2.1 JSON 与 JSONB 常用操作符

操作符	操作数据类型	描述	例子
->	int	得到 Json 数组的元素 (索引从 0 开始，负整数结束)	'[1,2,3]'::json->2
->	text	得到 Json 对象的域值	'{"a":1,"b":2}'::json->'b'
->>	int	得到 Json 数组的元素（text 格式输出）	[1,2,3]'::json->>2
->>	text	得到 Json 对象的域值（text 格式输出）	'{"a":1,"b":2}'::json->>'b'
#>	array of text	得到指定位置的 Json 对象	'{"a":[1,2,3],"b":[4,5,6]}'::json#>'{a,2}'
#>>	array of text	得到指定位置的 Json 对象（text 格式输出）	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'

注意:

1、使用->>操作符查询出来的数据为 text 格式而使用->查询出来的是 json 对象
2、使用#>>查询出来的数据是 text 格式的数据，而使用#>查询出来的数据为 json 数据

2.2 JSON 常用的创建函数

```
to_json(anyelement)

array_to_json(anyarray [, pretty_bool])

row_to_json(record [, pretty_bool])

json_build_array(VARIADIC "any")

json_build_object(VARIADIC "any")

json_object(text[])
```

json_object(keys text[], values text[])

2.3 JSON 聚合函数

json_agg(record)

json_object_agg(name, value)

2.4 JSON 处理函数

json_array_length(json)

jsonb_array_length(jsonb)

json_each(json)

jsonb_each(jsonb)

json_each_text(json)

jsonb_each_text(jsonb)

json_extract_path(from_json json, VARIADIC path_elems text[])

jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])

json_extract_path_text(from_json json, VARIADIC path_elems text[])

jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])

json_object_keys(json)

jsonb_object_keys(jsonb)

json_populate_record(base anyelement, from_json json)

jsonb_populate_record(base anyelement, from_json jsonb)

json_populate_recordset(base anyelement, from_json json)

jsonb_populate_recordset(base anyelement, from_json jsonb)

json_array_elements(json)

jsonb_array_elements(jsonb)

json_array_elements_text(json)

jsonb_array_elements_text(jsonb)

json_typeof(json)

jsonb_typeof(jsonb)

json_to_record(json)

jsonb_to_record(jsonb)

json_to_recordset(json)

jsonb_to_recordset(jsonb)

2.5 JSONB 操作符

操作符	操作类型	描述
@>	jsonb	左边的 JSON 值是否包含顶层右边 JSON 路径/值项
<@	jsonb	左边的 JSON 路径/值是否包含在顶层右边 JSON 的值中
?	text	字符串是否作为顶层键值存在于 JSON 中
?|	text[]	这些数组字符串中的任何一个是否作为顶层键值存在
?&	text[]	这些数组字符串是否作为顶层键值存在
||	jsonb	链接两个 jsonb 值到新的 jsonb 值

-	text	层左操作中删除键/值对会字符串元素，基于键值匹配键/值对
-	integer	删除制定索引的数组元素(负整数结尾)，如果顶层容器不是一个数组，那么抛出错误。
#-	text[]	删除制定路径的区域 元素(JSON 数组,负整数结尾)

2.6 常用的操作运算符

操作符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于
<>或！=	不相等

2.7 Greenplum 对 JSONB 支持的说明

目前 Greenplum 对 JSONB 格式的数据只支持简单的查询,接下来就不过多的介绍 JSONB 数据了。

3 JSON 运算符常用实例

3.1 单组 JSON 解析

```
 select '{"a":1}'::json ->>'a'    as jsondata;

 jsondata
----------
 1
(1 row)
```

3.2 多组 JSON 解析

```
select '{"a":1,"b":2}'::json->>'b'    as jsondata;
 jsondata
----------
 2
(1 row)
```

3.3 复杂的 JSON 解析

3.3.1 多个 JSON 子集的解析

```
select '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2    as jsondata;
   jsondata
-------------
 {"c":"baz"}
(1 row)
```

注意以上结果查询的坐标是从 0 开始的，查询条件必须是索引

3.3.2 获取 JSON 子集的数据

```
select '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'    as jsondata;
 jsondata
--------------
 {"c": "foo"}
(1 row)
```

注意#>'{a,b}的使用，表示嵌层查询

3.3.3 获取一个 JSON 集合的子元素

```
select '{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'    as jsondata;
 jsondata
----------
 3
(1 row)
```

注意这个 JSON 写的格式，以及获取的顺序

3.3.4 获取数值进行判断

```
select '{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'='3';
 ?column?
----------
 t
(1 row)
```

4 JSON 创建函数的使用

4.1 创建 int 类型的 JSON 格式数据

```
select array_to_json('{{1,5},{99,100}}'::int[])    as jsondata;
     jsondata
------------------
 [[1,5],[99,100]]
(1 row)
```

注意 int 数组的 json 数据已经把原本的格式转换了。

4.2 把行的数据转化为 JSON 类型的数据

```
select row_to_json(row(1,2,'foo'))    as jsondata;
                jsondata
-----------------------------
 {"f1":1,"f2":2,"f3":"foo"}
(1 row)
```

注意查看以上的结果可以看出 row 是行的数据，结果中 f1,f2,f3 是默认的字段的名，在后面将会介绍怎样获取字段名转化为 JSON。

4.3 把字段转化为 json 类型

```
dw=# create table test_table(id int,name varchar(50)) DISTRIBUTED BY(id);
CREATE TABLE
dw=# insert into test_table values(1,'xiaozhang'),(2,'xiaowang'),(3,'xiaoli');
INSERT 0 3
dw=# select array_to_json(array_agg(t),true) from    (select id,name    from test_table) t;
       array_to_json
------------------------------
[{"id":1,"name":"xiaozhang"},{"id":3,"name":"xiaoli"},{"id":2,"name":"xiaowang"}]

(1 row)
```

5 JSON 处理函数的使用

5.1 json_each(json) 把一个 Json 最外层的 Object 拆成 key-value 的形式

```
select * from json_each('{"a":"foo", "b":"bar"}');
  key | value
-----+-------
  a      | "foo"
  b      | "bar"
(2 rows)
```

以上结果只显示出了 key 与 value 的值，value 返回的是带双引号的值。

5.2 获取 JSON 中的数据(去除双引号)

```
select * from json_each_text('{"a":"foo", "b":"bar"}')
;
  key | value
-----+-------
  a      | foo
  b      | bar
```

```
(2 rows)
```

可以注意到与上一个比较 value 的值去除了双引，这个数据是比较使用的。

5.3 获取 JSON 数据中的 KEY 的值

```
select * from json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}') as jsondata;
 jsondata
----------
 f1
 f2
(2 rows)
```

只返回 json 数据的 key 值

5.4 返回 JSON 的文本值

```
select * from json_array_elements_text('["foo", "bar"]');
 value
-------
 foo
 bar
(2 rows)
```

6 查询 JSON 数据的方式

6.1 创建支持 JSON 数据的表

6.1.1 创建表的 SQL

创建带有自增长主键的表

```
create table test_json (
      id serial not null primary key,
      info json not null
) DISTRIBUTED BY(id);
```

6.1.2 插入数据 SQL

```
insert into test_json (info)
values
	(
		'{ "customer": "john doe", "items": {"product": "beer","qty": 6}}'
	),
	(
		'{ "customer": "lily bush", "items": {"product": "diaper","qty": 24}}'
	),
	(
		'{ "customer": "josh william", "items": {"product": "toy car","qty": 1}}'
	),
	(
		'{ "customer": "mary clark", "items": {"product": "toy train","qty": 2}}'
	);
```

6.1.3 获取 JSON 数据的 KEY 值

```
SELECT info -> 'customer' AS customer FROM test_json;
	customer
----------------
  "josh william"
  "mary clark"
  "john doe"
  "lily bush"
(4 rows)
```

以上数据只把制定 KEY 的 VALUE 获取出来，注意使用-> 是不把双引号去掉的。

```
SELECT info ->> 'customer' AS customer FROM test_json;
    customer
--------------
  lily bush
  john doe
  josh william
  mary clark
(4 rows)
```

使用->> 就可以把双引去掉了。

6.2 按照条件查询数据

```
select info ->> 'customer' as customer from test_json where info -> 'items' ->> 'product' = 'diaper';
  customer
-----------
  lily bush
(1 row)
```

查询条件也可以作为解析的对象。

也可以写成以下的形式

```
select info ->> 'customer' as customer,info -> 'items' ->> 'product' as product from test_json where cast ( info -> 'items' ->> 'qty' as integer ) = 2;
```

```
   customer    |    product
---------------+-----------
  mary clark | toy train
(1 row)
```

info -> 'items' ->> 'qty' AS INTEGER 是获取 json 集合中元素是 qty 的数据 转化为 INTEGER, case() 是把数值转化为 int4 类型

6.3 集合函数查询 JSON 数据

```
select min(cast( info -> 'items' ->> 'qty' as integer)), max (cast (info -> 'items' ->> 'qty' as integer)), sum (cast (info -> 'items' ->> 'qty' as integer)), avg (cast (info -> 'items' ->> 'qty' as integer)) from test_json;
```

```
  min | max | sum |              avg
-----+-----+-----+--------------------
    1 |    24 |    33 | 8.2500000000000000
```

(1 row)

6.4 获取 JSON 结构中的数据

select info -> 'items' ->> 'product' as product from test_json order by product;

```
   product
-----------
 beer
 diaper
 toy car
 toy train
(4 rows)
```

SQL 中可以->与->>一起使用，区别就是结果有无双引的问题。

6.5 使用默认的函数查找数据

6.5.1 JSON_EACH 函数的使用

select json_each(info) from test_json;

```
                      json_each
-------------------------------------------------
 (customer,"""josh william""")
 (items,"{""product"": ""toy car"",""qty"": 1}")
 (customer,"""mary clark""")
 (items,"{""product"": ""toy train"",""qty"": 2}")
 (customer,"""lily bush""")
 (items,"{""product"": ""diaper"",""qty"": 24}")
 (customer,"""john doe""")
 (items,"{""product"": ""beer"",""qty"": 6}")
(8 rows)
```

json_each 函数把含有 key 与 value 的数据全部取了出来，如果一行有多个 key 与 value 则会把分行显示出来。

6.5.2 JSON_OBJECT_KEYS 函数的使用

```
select json_object_keys (info->'items') as jsondata from test_json;
  jsondata
----------
  product
  qty
  product
  qty
  product
  qty
  product
  qty
(8 rows)
```

6.6 把查询数据转化为 JSON

创建表并构造数据

```
CREATE OR REPLACE FUNCTION "public"."uuid_python"()
   RETURNS "pg_catalog"."varchar" AS $BODY$
import uuid
return uuid.uuid1()
$BODY$
   LANGUAGE 'plpythonu' VOLATILE COST 100;


create table test_json_data(
filed1 varchar(50),
filed2 varchar(50)
) DISTRIBUTED BY(filed1);

insert into test_json_data select md5(uuid_python()) as filed1, md5(uuid_python()) as   filed2
from generate_series(1,10);
```

6.6.1 查看原始数据

```
select * from test_json_data;
```

```
                filed1                |                filed2
----------------------------------+----------------------------------
 02a49d4ddba8dec994df27908eb98003 | e0266c5b9095e3d7660e1841a5be3cfd
 6415540deaea3cb2279ba2b4b2199c05 | d4e1f83bda7f37d2ed4b388de70bf470
 d63738f834be2cce38d6aa1802f674c8 | eced478050de3ac7fc366f1e309ff9ca
 d4b9576198948233d95ac6495e332fdb | 603b5c38fdd577c6ba43cd791e01c6ae

(4 rows)
```

6.6.2 把查询的数据转化为 JSON

6.6.2.1 把字段的名字作为 JSON 对象

```
select row_to_json(test_json_data) from test_json_data;
                                    row_to_json
-------------------------------------------------------------------------------------------

{"filed1":"d63738f834be2cce38d6aa1802f674c8","filed2":"eced478050de3ac7fc366f1e309ff9ca"
}

{"filed1":"02a49d4ddba8dec994df27908eb98003","filed2":"e0266c5b9095e3d7660e1841a5be3c
fd"}

{"filed1":"6415540deaea3cb2279ba2b4b2199c05","filed2":"d4e1f83bda7f37d2ed4b388de70bf4
70"}

{"filed1":"d4b9576198948233d95ac6495e332fdb","filed2":"603b5c38fdd577c6ba43cd791e01c6
ae"}
(4 rows)
```

把 test_json_data 表全部转化成了 json 的格式了

6.6.2.2 使用默认的 JSON 字段名字

```
select row_to_json(row(filed1, filed2)) from test_json_data;
                                          row_to_json
-----------------------------------------------------------------------------------
 {"f1":"02a49d4ddba8dec994df27908eb98003","f2":"e0266c5b9095e3d7660e1841a5be3cfd"}
 {"f1":"6415540deaea3cb2279ba2b4b2199c05","f2":"d4e1f83bda7f37d2ed4b388de70bf470"}
 {"f1":"d4b9576198948233d95ac6495e332fdb","f2":"603b5c38fdd577c6ba43cd791e01c6ae"}
 {"f1":"d63738f834be2cce38d6aa1802f674c8","f2":"eced478050de3ac7fc366f1e309ff9ca"}
```

(4 rows)

可以看出已使用默认的字段作为 JSON 的对象了。

或写成一下的形式

```
select row_to_json(t) from    (select filed1 f1,filed2 f2 from test_json_data ) t;
                                                  row_to_json
--------------------------------------------------------------------------------
 {"f1":"d63738f834be2cce38d6aa1802f674c8","f2":"eced478050de3ac7fc366f1e309ff9ca"}
 {"f1":"02a49d4ddba8dec994df27908eb98003","f2":"e0266c5b9095e3d7660e1841a5be3cfd"}
 {"f1":"6415540deaea3cb2279ba2b4b2199c05","f2":"d4e1f83bda7f37d2ed4b388de70bf470"}
 {"f1":"d4b9576198948233d95ac6495e332fdb","f2":"603b5c38fdd577c6ba43cd791e01c6ae"}
(4 rows)
```