PostgreSQL 9.4 正在加载一项新功能叫jsonb，是一种新型资料，可以储存支援GIN索引的JSON 资料。换言之，此功能，在即将来临的更新中最重要的是，如果连这都不重要的话，那就把Postgres 置于文件为本数据库系统的推荐位置吧。

自从9.2开始，一个整合JSON 资料类型已经存在，带有一整套功能（例如资料产生和资料解构功能），还有9.3新增的操作者。当使用JSON 资料类型，资料的被存储成一完全一样的副本，功能还在此之上运作，还另外需要后台运作的重新分析。

这心得JSONB 资料类型以已降解的2元格式存储，所以，插入此资料会比JSON高效，因为后台不再需要重新分析，因此让它更快速运行，而且还兼顾GIN 索引。就是因为最后这个原因，我们实际上建议读者使用jsonb来代替json制作程式（当然你还可以因应需要而使用json）。请记住jsonb使用相同的操作者和功能，读者们可以看我之前的帖子去令你得到些什么启发（或者干脆看Postgres的文件）。

现在让我们看一下JSONB是如何工作的，同时和JSON比较一下。采用的测试数据是860万的[geobase](http://www.geonames.org/export/)类型数据，大概1.1G大小，包括了城市名，国家代码（可以[在这](http://download.geonames.org/export/dump/readme.txt)参见完整列表）等很多字段。首先通过底层复制(raw copy)来把这些数据存储到数据库的一个新表里面，之后把这张表通过一组填充因子是100的表转换成JSON/JSONB，之后来看它们各占多少空间。

=# COPY geodata FROM '$HOME/Downloads/allCountries.txt';

COPY 8647839

=# CREATE TABLE geodata\_jsonb (data jsonb) with (fillfactor=100);

CREATE TABLE

=# CREATE TABLE geodata\_json (data json) with (fillfactor=100);

CREATE TABLE

=# \timing

Timing is on.

=# INSERT INTO geodata\_json SELECT row\_to\_json(geodata) FROM geodata;

INSERT 0 8647839

Time: 287158.457 ms

=# INSERT INTO geodata\_jsonb SELECT row\_to\_json(geodata)::jsonb FROM geodata;

INSERT 0 8647839

Time: 425825.967 ms

生成JSONB数据花费稍微长一点时间，大小有没有区别呢？

=# SELECT pg\_size\_pretty(pg\_relation\_size('geodata\_json'::regclass)) AS json,

     pg\_size\_pretty(pg\_relation\_size('geodata\_jsonb'::regclass)) AS jsonb;

 json  | jsonb

---------+---------

 3274 MB | 3816 MB

(1 row)

在JSON数据上面做索引从9.3版本开始，比如用操作符（注意 因为它返回文本，所以'->>'被采用；并且根据查询不同，索引采用不同的关键字）

=# CREATE INDEX geodata\_index ON

  geodata\_json ((data->>'country\_code'), (data->>'asciiname'));

CREATE INDEX

=# SELECT pg\_size\_pretty(pg\_relation\_size('geodata\_index'::regclass))

  AS json\_index;

 json\_index

------------

 310 MB

(1 row)

=# SELECT (data->>'population')::int as population,

     data->'latitude' as latitude,

     data->'longitude' as longitude

  FROM geodata\_json WHERE data->>'country\_code' = 'JP' AND

    data->>'asciiname' = 'Tokyo' AND

    (data->>'population')::int != 0;

 population | latitude | longitude

------------+----------+-----------

  8336599 | 35.6895 | 139.69171

(1 row)

=# -- Explain of previous query

                            QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on geodata\_json (cost=6.78..865.24 rows=215 width=32)

  Recheck Cond: (((data ->> 'country\_code'::text) = 'JP'::text) AND ((data ->> 'asciiname'::text) = 'Tokyo'::text))

  Filter: (((data ->> 'population'::text))::integer <> 0)

  -> Bitmap Index Scan on geodata\_index (cost=0.00..6.72 rows=216 width=0)

     Index Cond: (((data ->> 'country\_code'::text) = 'JP'::text) AND ((data ->> 'asciiname'::text) = 'Tokyo'::text))

 Planning time: 0.172 ms

(6 rows)

在这个例子里，计划(planner)可以使用bitmap索引扫描，同时使用了之前产生的索引。

现在，JSONB的一个新特点就是检查包含带有操作符@>的数据容量，这种数据是可以用GIN来索引的，这种操作符数据也包括了?，?|和?&(为了检查给定的关键字是否存在)。 GIN索引对两类操作符起作用：

    缺省操作符类，之前列出的四个；

    jsonb\_hash\_ops，仅支持@>，但是当搜索数据时性能表现不错，而且所占磁盘空间较小；

下面是它如何工作：

=# CREATE INDEX geodata\_gin ON geodata\_jsonb

   USING GIN (data jsonb\_hash\_ops);

CREATE INDEX

=# SELECT (data->>'population')::int as population,

   data->'latitude' as latitude,

   data->'longitude' as longitude

  FROM geodata\_jsonb WHERE data @> '{"country\_code": "JP", "asciiname": "Tokyo"}' AND

    (data->>'population')::int != 0;

 population | latitude | longitude

------------+----------+-----------

  8336599 | 35.6895 | 139.69171

(1 row)

 =# SELECT pg\_size\_pretty(pg\_relation\_size('geodata\_gin'::regclass)) AS jsonb\_gin;

 jsonb\_gin

-----------

 1519 MB

(1 row)

=# -- EXPLAIN of previous query

                   QUERY PLAN

-------------------------------------------------------------------------------------

 Bitmap Heap Scan on geodata\_jsonb (cost=131.01..31317.76 rows=8605 width=418)

  Recheck Cond: (data @> '{"asciiname": "Tokyo", "country\_code": "JP"}'::jsonb)

  Filter: (((data ->> 'population'::text))::integer <> 0)

  -> Bitmap Index Scan on geodata\_gin (cost=0.00..128.86 rows=8648 width=0)

     Index Cond: (data @> '{"asciiname": "Tokyo", "country\_code": "JP"}'::jsonb)

 Planning time: 0.134 ms

根据应用的需求，你或许想采用空间消耗低的索引，比如BTree建立在JSON数据上的索引类型；GIN索引有着更多的优点，因为它覆盖了所有的JSON字段，并且检查容量；

--------------------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------------------

如何使用PostgreSQL中的JSONB数据类型(PG JSON系列2)

上一篇 [使用PostgreSQL中的row\_to\_json()直接获得JSON（PG JSON系列1）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1007839)主要针对现有的非json/jsonb类型如何输出json结果，这一篇主要写一下在数据库设计中直接使用json/jsonb数据类型（而不是json/jsonb处理函数）带来的问题。我们的问题是：

1. 设定一个典型的JSON/JSONB使用场景，以及演示如何做增删改查

首先来看JSON/JSONB的区别，参考<https://www.postgresql.org/docs/9.6/static/datatype-json.html>  ：

There are two JSON data types: json and jsonb. They accept **almost** identical sets of values as input. The major practical difference is one of efficiency. The json data type stores an exact copy of the input text, which processing functions must reparse on each execution; whilejsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. jsonb also supports indexing, which can be a significant advantage.

Because the json type stores an exact copy of the input text, it will preserve semantically-insignificant white space between tokens, as well as the order of keys within JSON objects. Also, if a JSON object within the value contains the same key more than once, all the key/value pairs are kept. (The processing functions consider the last value as the operative one.) By contrast, jsonb does not preserve white space, does not preserve the order of object keys, and does not keep duplicate object keys. If duplicate keys are specified in the input, only the last value is kept.

In general, most applications should prefer to store JSON data as jsonb, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

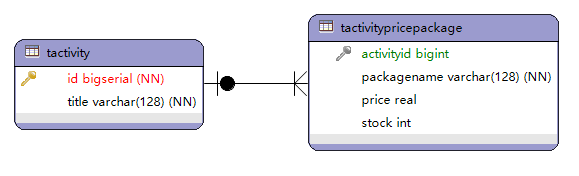
最后一句话很明确，基本上使用的时候，都用JSONB。原因可以这么理解json存储是纯粹字符串，jsonb的存储和访问做了解析和优化，等后面我有时间专门写一篇jsonb和json是如何在内核里实现的，以及他们到底有啥区别，此处不做纠结。

**首先用JSONB来存储什么呢，我感觉和多维数组相关的比较合适，用ARRAY类型吧，输出的时候还得做个json转换，麻烦。**

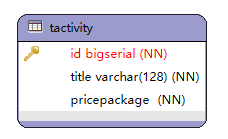
在数据库设计的时候，有时候为了一小件事情设置个外键关联表有点不值得，感觉有点大材小用，例如下面的演出活动的票价套餐信息(父母帮应该给我广告费，顺便也提一下我们的“两小童”)：



那设计数据库的时候，就有两种方案，一种是常规的做一个关联表（TActivityPricePackage），如下图：



另外一种就是用jsonb来表示票价套餐（pricepackage），这种方法感觉到设计又简洁，还避免获取的时候做JOIN（乖乖，还能减少JAVA程序员工作，是不是也搞好了团队协作了），如下图：



为了简洁的数据库设计的理想而奋斗，下面我们来做个实验：

运行环境：PG 9.6，WIN8.1 64，Tomcat 8.0，JDK 8

数据库表：

|  |
| --- |
| -- Table: TActivity  -- 演出活动表  DROP TABLE IF EXISTS TActivity CASCADE;  CREATE TABLE TActivity (   id bigint DEFAULT nextval('jsontest\_uuid\_seq') PRIMARY KEY,-- 活动id   title character varying(128) NOT NULL,-- 活动名称   pricepackage jsonb NOT NULL -- 价格套餐,格式如：[{"packagename":"成人票","price":25,"stock":1000},{"packagename":"儿童票（12岁以下）","price":15,"stock":1000}]  )WITH (   OIDS=FALSE  ); |

测试数据：

|  |
| --- |
| -- Table: TActivity  -- 演出活动表  insert into TActivity values(1,'演出活动标题1','[{"packagename":"成人票","price":189,"stock":100},{"packagename":"儿童票（12岁以下）","price":66,"stock":20},{"packagename":"成人+儿童套票","price":128,"stock":10}]'); |

好，我分别演示如何做增删改查。

**1.先说增**

**第一次Insert的时候**，除了上面测试数据里面直接字符串的方式：

insert into TActivity values(1,'演出活动标题1','[{"packagename":"成人票","price":189,"stock":100},{"packagename":"儿童票（12岁以下）","price":66,"stock":20},{"packagename":"成人+儿童套票","price":128,"stock":10}]');

以外，根据（<https://www.postgresql.org/docs/9.6/static/functions-json.html> ）还可以有如下方法：

json\_build\_object

json\_object  
json\_build\_array  
json\_build\_object和json\_object生成一个名值对的json对象比较方便，而json\_build\_array只能生成比较简单的json array，正如官方文档例子里所展示的：

|  |  |
| --- | --- |
| json\_build\_array(1,2,'3',4,5) | [1, 2, "3", 4, 5] |

**综上我们发现**，**在第一次insert含有数组的时候，直接用字符串吧。**

**那如果想给jsonb数组增加一条数据怎么办？**

我们可以使用连接操作符||：

update tactivity set pricepackage = pricepackage || '{"packagename":"成人票新增3","price":189,"stock":100}'  where id = 1

或者新值在前面：

update tactivity set pricepackage = '{"packagename":"成人票新增3","price":189,"stock":100}' || pricepackage    where id = 1

也可以使用函数jsonb\_insert，如在数组的最后增加一个：

update tactivity set pricepackage = jsonb\_insert(pricepackage, '{-1}', '{"packagename":"成人票新增100","price":189,"stock":100}', true)    where id = 1

其中-1表示从数组后面数第一个，即倒数第一个，true表示在该元素的后面插入。

或在数组的头部插入：update tactivity set pricepackage = jsonb\_insert(pricepackage, '{0}', '{"packagename":"成人票新增00","price":189,"stock":100}', false)    where id = 1

**综上我们发现，给jsonb数组最后面或者最前面增加一条数据时，还是使用操作符 || 使用起来更加自然，不易出错。而如果想在jsonb数组的中间增加一条记录，则使用jsonb\_insert()。**

**2.改**

如果我们想修改jsonb中的一个数据怎么办？

假设我们现在pricepackage中jsonb数组已有两条数据：

update tactivity set pricepackage = '[{"price":189,"packagename":"成人票0","stock":100},{"price":189,"packagename":"成人票1","stock":100}]'   where id = 1

**我们先看第一种情况，如何修改JSONB数组中的第1条完整的记录呢？**

1. 即我想把：{"price":189,"packagename":"成人票0","stock":100} 替换成：{"price":189,"packagename":"成人票00","stock":100}

如何写呢？

正常写法应该是：

update tactivity set pricepackage = jsonb\_set(pricepackage,'{0}','{"price":189,"packagename":"成人票00","stock":100}',false) where id = 1

但是我发现9.6.0关于jsonb\_set()修改数组顶级元素的时候有个bug，参考我的另外一篇博文：

[9.6.0版本里jsonb\_set()有个bug，截至到2016.10.12](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1008263)

这里的false得替换成true才行，即：

update tactivity set pricepackage = jsonb\_set(pricepackage,'{0}','{"price":189,"packagename":"成人票00","stock":100}',true) where id = 1

而在9.5.4里却是正常的，即直接使用false即可。

**我们再看第二种情况，如何修改JSONB数组中的第1条记录中的某个值呢？**

如，我想把：{"price":189,"packagename":"成人票0","stock":100} 替换成：{"price":189,"packagename":"成人票000","stock":100}

如何写呢？

正常写法应该是：

update tactivity set pricepackage = jsonb\_set(pricepackage,'{0,packagename}','"成人票000"',false) where id = 1

这回PG 9.6.0跑对了。

**我们再看第三种情况，如果我不知道是数组中的哪一条记录呢？即我想通过一个查找定位到该记录呢，其实这个问题是json的查找定位问题，我们放在JSON的查询那一部分讲。**

**3.删**

官方文档里关于JSONB的删除只有操作符：

|  |  |  |  |
| --- | --- | --- | --- |
| - | text | Delete key/value pair or **string** element from left operand. Key/value pairs are matched based on their key value. | '{"a": "b"}'::jsonb - 'a' |
| - | integer | Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array. | '["a", "b"]'::jsonb - 1 |
| #- | text[] | Delete the field or element with specified path (for JSON arrays, negative integers count from the end) | '["a", {"b":1}]'::jsonb #- '{1,b}' |

这里针对第2，3种情况做个演示：

update tactivity set pricepackage = pricepackage - 0  where id = 1

以及：

update tactivity set pricepackage = pricepackage #- '{0,stock}'  where id = 1

**4.查**

JSON查找访问方面的操作，我们从字符串匹配/数值比较/ 这两类比较典型的来演示。

假设数据库表TActivity有如下记录：

delete from tactivity;

insert into TActivity values(1,'演出活动标题1','[{"packagename":"成人票","price":189,"stock":100},{"packagename":"儿童票（5-15岁）","price":66,"stock":20},{"packagename":"成人+儿童套票","price":128,"stock":10}]');

insert into TActivity values(2,'演出活动标题2','[{"packagename":"成人票","price":99,"stock":100},{"packagename":"儿童票（3-5岁）","price":58,"stock":20},{"packagename":"成人+儿童套票","price":99,"stock":10}]');

**关于字符串匹配，我们演示的问题是：**

1. 找出符合条件“packagename = 儿童票（3-5岁）”的那些活动？

那SQL语句为:

select \* from TActivity a where a.pricepackage @> '[{"packagename":"儿童票（3-5岁）"}]'::jsonb;

这里的 @> '[{"packagename":"儿童票（3-5岁）"}]'::jsonb，注意必须是'[]'括起来的，即比较时不仅比较内容还要比较结构，参考：<https://www.postgresql.org/docs/9.6/static/datatype-json.html>  里面的这句话：

The general principle is that the contained object must match the containing object as to **structure and data contents**, possibly after discarding some non-matching array elements or object key/value pairs from the containing object. But remember that the order of array elements is not significant when doing a containment match, and duplicate array elements are effectively considered only once.

注意上面这句SQL语句，是字符串相等比较，没有通配符的概念，相比于普通字符串比较强大的功能，这或许也是JSONB的缺点（或许以后可以改进）。

**关于数值比较，我们演示的问题是：**

* 找出符合条件“数组中第0个元素的price < 100”的那些活动？
* 找出符合条件“数组中所有元素的price < 100”的那些活动？

关于第1个问题，SQL语句为:

select \* from TActivity a where cast(a.pricepackage->0->>'price' as int) <100

关于第2个问题，由于JSONB处理函数不能就数组里面的对象（而不是简单的数值）遍历中比较，所以我们得先把jsonb中的值取出来：

CREATE type json\_type\_pricepackage AS (packagename text, price numeric, stock numeric);

SELECT id , (jsonb\_populate\_recordset(null::json\_type\_pricepackage, pricepackage)).\* FROM tactivity;

返回结果为：



然后再做过滤，获得结果：

WITH rows\_filtered AS (

   SELECT DISTINCT id FROM (

SELECT id ,(jsonb\_populate\_recordset(null::json\_type\_pricepackage, pricepackage)).\* FROM tactivity ) a

WHERE price < 100)

SELECT \* FROM TActivity a WHERE a.id IN (SELECT \* FROM rows\_filtered);

是不是有点绕？确实有点。

当然如果不希望用CREATE type json\_type\_pricepackage AS (packagename text, price numeric, stock numeric);创建类型的话，可以使用函数jsonb\_to\_recordset()：

select a.id, b.\* from tactivity a, jsonb\_to\_recordset(a.pricepackage) as b(packagename text, price numeric, stock numeric);

然后类似的：

WITH rows\_filtered AS (

   SELECT DISTINCT id FROM (

select a.id, b.\* from tactivity a, jsonb\_to\_recordset(a.pricepackage) as b(packagename text, price numeric, stock numeric)

   ) c

WHERE c.price < 100)

SELECT \* FROM TActivity a WHERE a.id IN (SELECT \* FROM rows\_filtered);

来获得想要的结果。

本节小结：我们发现，对于模糊匹配的字符串查询，JSONB还是欠缺的，除非把字符串取出来再做比较，但是又感觉不是很方便。对于数值比较，非==比较，如<，>等比较，还是需要把值从JSONB里取出来在做比较，还是感觉略微不方便。

综上：感觉JSONB更加适合一些特定情况，如字符串模糊检索少，数值非==比较少的情况下。关于JSONB的存储效率和访问效率，专门写一篇博客说明一下。

--------------------------------------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------------------------------------------------------------------

# PG数据库JSonb类型数据测试

## 1. 测试大纲

1.1. 数据表写入速率对比测试。

1.2. 数据表容量对比测试。

1.3. 全表扫描查询性能对比测试。

1.4. 索引扫描查询性能对比测试。

## 2. 测试步骤

使用定时记录数据对比测试两种表之间的差别，即普通表定时记录月表和jonsb表。两种表结构定义如下：

**普通定时记录月表：**

CREATE TABLE pd\_tb\_12\_test ( sourceid integer NOT NULL, logtime timestamp without time zone NOT NULL, msec smallint NOT NULL, data1 double precision, data2 double precision, data3 double precision, data4 double precision, data5 double precision, data6 double precision, data7 double precision, data8 double precision, data9 double precision, data10 double precision, data11 double precision, data12 double precision, data13 double precision, data14 double precision, data15 double precision, data16 double precision, status smallint, CONSTRAINT pk\_pd\_tb\_12\_test PRIMARY KEY (sourceid, logtime, msec), CONSTRAINT ck\_pd\_tb\_12\_test\_time CHECK (logtime >= to\_date ('2015-12-01'::text, 'YYYY-MM-DD'::text) AND logtime < to\_date ('2016-01-01'::text, 'YYYY-MM-DD'::text)))

**Jsonb类型表：**

create table pd\_tb\_12\_test\_jsonb(datalog jsonb);

**说明：**未对Jsonb类型表创建主键和检查约束，在数据写入测试时，普通表会吃亏。

### 2.1. 数据表写入速率测试

测试数据规模：4000组定时记录，每5min产生一次数据，制造12月份一个月的数据。总数据条数为：4000\*6\*24\*31 = 35712000（条）。

**普通表写入测试：**

临时表中存储4000条12月1号0点0分的源数据，使用此源数据向测试表pd\_tb\_12\_test中循环插入数据，每次循环logtime增加5min，直到12月31号23点55分。   
写入耗时：1608s。

**JSonb表写入测试：**

直接把pd\_tb\_12\_test表中的数据转换为jsonb类型写入到pd\_tb\_12\_test\_jsonb表中，写入语句为：

insert into pd\_tb\_12\_test\_jsonb(datalog) select row\_to\_json(pd\_tb\_12\_test)::jsonb from pd\_tb\_12\_test; 写入耗时：2639s。

**说明：**此时未对jsonb表创建主键和检查约束，从这方面来说jsonb表占了一些便宜。

**结论：**单从写入同样数据量的耗时来看，jsonb表的写入速度比普通表要慢很多。

### 2.2. 数据表容量测试

使用上述写入数据后的表进行对比。

普通表pd\_tb\_12\_test的数据表大小为：6488MB；

普通表pd\_tb\_12\_test的主键索引大小为：1383MB；

Jsonb表pd\_tb\_12\_test\_jsonb的数据表大小为：18GB；

Jsonb表pd\_tb\_12\_test\_jsonb此时无索引。

**结论：**可以看出，同样的数据量Jsonb表的大小为普通表的3倍。

### 2.3. 全表扫描查询性能测试

为了保证查询测试的公平性，在测试查询性能之前，先删除了普通表pd\_tb\_12\_test的主键索引，然后再进行查询性能的测试。

普通表查询测试：

PECSTAR\_DATA\_GZZD=# explain analyze select \* from pd\_tb\_12\_test where sourceid= 20 and logtime >= '2015-12-12' and logtime <= '2015-12-13'; QUERY PLAN---------------------------------------------------------------------------------------------------------------------------------------------------------------------Seq Scan on pd\_tb\_12\_test (cost=0.00..1455472.00 rows=283 width=144) (actual time=136292.211..144357.127 rows=289 loops=1) Filter: ((logtime >= '2015-12-12 00:00:00'::timestamp without time zone) AND (logtime <= '2015-12-13 00:00:00'::timestamp without time zone) AND (sourceid = 20)) Rows Removed by Filter: 35711711 Planning time: 0.223 ms Execution time: 144357.435 ms(5 行记录)

**查询总耗时144.357s；**

Jsonb表查询测试：

PECSTAR\_DATA\_GZZD=# explain analyze select \* from pd\_tb\_12\_test\_jsonb where datalog->>'sourceid'='20' and datalog->>'logtime' >= '2015-12-12T00:00:00' and datalog->>'logtime' <= '2015-12-13T00:00:00'; QUERY PLAN ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------- Seq Scan on pd\_tb\_12\_test\_jsonb (cost=0.00..3276593.50 rows=893 width=506) (actual time=219834.645..241907.125 rows=289 loops=1) Filter: (((datalog ->> 'logtime'::text) >= '2015-12-12T00:00:00'::text) AND ((datalog ->> 'logtime'::text) <= '2015-12-13T00:00:00'::text) AND ((datalog ->> 'sourceid'::text) = '20'::text)) Rows Removed by Filter: 35711711 Planning time: 1.479 ms Execution time: 241907.384 ms (5 行记录)

**查询总耗时241.907s；**

### 2.4. 索引扫描查询性能测试

对两张表的sourceID，logtime，msec三个字段创建索引，具体如下：

PECSTAR\_DATA\_GZZD=# alter table pd\_tb\_12\_test add constraint pk\_pd\_tb\_12\_test primary key (sourceid,logtime,msec); ALTER TABLE 时间：458263.365 ms PECSTAR\_DATA\_GZZD=# create index ix\_pd\_tb\_12\_test\_jsonb on pd\_tb\_12\_test\_jsonb using btree((datalog->>'sourceid'),(datalog->>'logtime'),(datalog->>'msec')); CREATE INDEX 时间：2298165.514 ms

索引创建完成后，再次执行查询操作，查询语句和2.3节中查询语句相同。   
普通表查询测试：

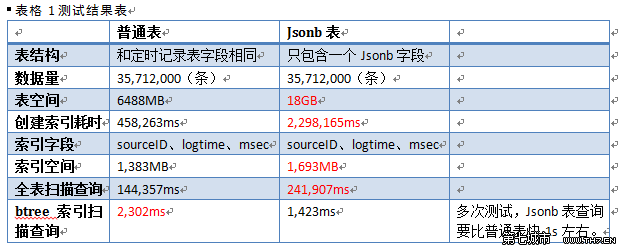
PECSTAR\_DATA\_GZZD=# explain analyze select \* from pd\_tb\_12\_test where sourceid= 20 and logtime >= '2015-12-12' and logtime <= '2015-12-13'; QUERY PLAN ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------- Bitmap Heap Scan on pd\_tb\_12\_test (cost=12.17..1133.45 rows=283 width=144) (actual time=44.777..2277.358 rows=289 loops=1) Recheck Cond: ((sourceid = 20) AND (logtime >= '2015-12-12 00:00:00'::timestamp without time zone) AND (logtime <= '2015-12-13 00:00:00'::timestamp without time zone)) Heap Blocks: exact=289 -> Bitmap Index Scan on pk\_pd\_tb\_12\_test (cost=0.00..12.10 rows=283 width=0) (actual time=35.126..35.126 rows=289 loops=1) Index Cond: ((sourceid = 20) AND (logtime >= '2015-12-12 00:00:00'::timestamp without time zone) AND (logtime <= '2015-12-13 00:00:00'::timestamp without time zone)) Planning time: 51.055 ms Execution time: 2302.445 ms (7 行记录)

**时间：2302.445 ms**

Jsonb表查询测试：

PECSTAR\_DATA\_GZZD=# explain analyze select \* from pd\_tb\_12\_test\_jsonb where datalog->>'sourceid'='20' and datalog->>'logtime' >= '2015-12-12T00:00:00' and datalog->>'logtime' <= '2015-12-13T00:00:00'; QUERY PLAN ----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------- Bitmap Heap Scan on pd\_tb\_12\_test\_jsonb (cost=35.96..3578.43 rows=893 width=506) (actual time=29.621..1423.518 rows=289 loops=1) Recheck Cond: (((datalog ->> 'sourceid'::text) = '20'::text) AND ((datalog ->> 'logtime'::text) >= '2015-12-12T00:00:00'::text) AND ((datalog ->> 'logtime'::text) <= '2015-12-13T00:00:00'::text)) Heap Blocks: exact=289 -> Bitmap Index Scan on ix\_pd\_tb\_12\_test\_jsonb (cost=0.00..35.73 rows=893 width=0) (actual time=20.306..20.306 rows=289 loops=1) Index Cond: (((datalog ->> 'sourceid'::text) = '20'::text) AND ((datalog ->> 'logtime'::text) >= '2015-12-12T00:00:00'::text) AND ((datalog ->> 'logtime'::text) <= '2015-12-13T00:00:00'::text)) Planning time: 1.319 ms Execution time: 1423.766 ms (7 行记录)

**时间：1423.766 ms**



**结论：**可以看出，在数据写入速率、数据容量、创建索引耗时、索引容量、全表扫描速率上，jsonb表都处于劣势，但是在btree索引查询上，jsonb表却比普通表快很多。所以，如果是对查询速率要求较高，对容量要求低的需求，可以考虑使用jsonb表，而不使用普通表。

-----------------------------------------------------------------------------------------------------------------------------------------------

关于pgsql 的json 和jsonb 的数据处理笔记

1. json 和jsonb 区别  
两者从用户操作的角度来说没有区别，区别主要是存储和读取的系统处理（预处理）和耗时方面有区别。json写入快，读取慢，jsonb写入慢，读取快。

2. 常用的操作符

操作符：

-> // 右边传入整数（针对纯数组），获取数组的第n个元素，n从0开始算，返回值为json

示例： select '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2 // 输出 {"c":"baz"}

-> // 右边传入键值（针对关联数组），获取数组的第n个元素，n从0开始算，返回值为json

示例： select '{"a": {"b":"foo"}, "c":{"a": "aaa"}}'::json->'a' // 输出 {"b":"foo"}

->> // 右边传入整数（针对纯数组），获取数组的第n个元素，n从0开始算，返回值为文本

示例： select '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->>2 // 输出 {"c":"baz"}

->> // 右边传入键值（针对关联数组），获取数组的第n个元素，n从0开始算，返回值为文本

示例： select '{"a": {"b":"foo"}, "c":{"a": "aaa"}}'::json->>'a' // 输出 {"b":"foo"}

#> // 获取json子对象，传入数组，返回json

示例： select '{"a": {"b":{"c": "foo"}}}'::json#> '{a,b}' // 输出 {"c": "foo"}

#>> // 获取json子对象并转换为文本，传入数组，返回文本

示例： select '{"a": {"b":{"c": "foo"}}}'::json#>> '{a,b}' // 输出 {"c": "foo"}

3. 操作函数  
目前pgsql版本提供了两套函数分别处理，可以通用，名称也差不多，比如 json\_each 和 jsonb\_each ， json\_array\_elements 和 jsonb\_array\_elements 。

json相关的处理函数比较多，常用的有如下三个，这三个基本够用了

json\_object\_keys  // 返回json的键（多层只返回第一层），该函数不能用于纯数组.

json\_array\_elements  // 提取转换纯数组元素

json\_extract\_path   // 返回JSON值所指向的某个键元素（相当于 #> 操作符），该函数不能直接操作纯数组。

需要注意的是如果你创建字段用的是json就用json相关函数，如果创建字段用的是jsonb就用jsonb相关函数。

json\_object\_keys 函数示例：

select json\_object\_keys ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": "10"  
}  
')

输出：



json\_object\_keys 函数示例：

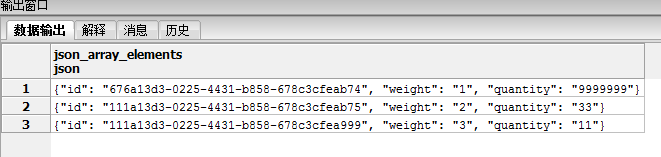
select json\_object\_keys ('  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
')

输出：

ERROR:  cannot call json\_object\_keys on an array     // 不能用于数组

json\_array\_elements  函数 示例:

select json\_array\_elements ('  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"},  
{"id": "111a13d3-0225-4431-b858-678c3cfea999", "weight": "3", "quantity": "11"}  
]  
')

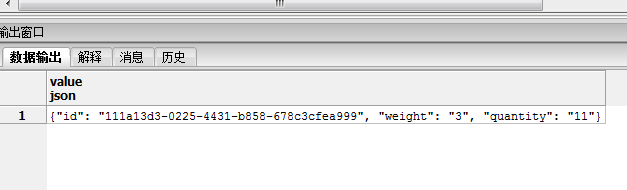


我们看到json数据被分离成三条记录，这时我们就可以对其进行查询操作，

比如查询是否包含了weight=3的数据。

select \* from json\_array\_elements ('  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"},  
{"id": "111a13d3-0225-4431-b858-678c3cfea999", "weight": "3", "quantity": "11"}  
]  
') as jae  
where jae::jsonb->>'weight' = '3'

#输出：



我们看到这样就可以到对json数据内部进行查询了。

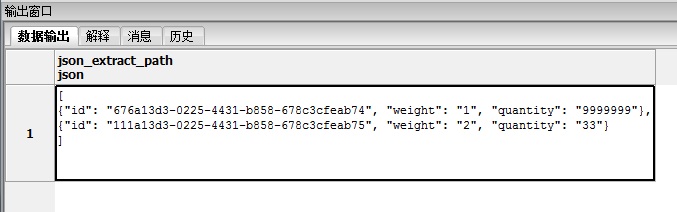
json\_extract\_path   函数示例：

比如要获取键 ‘goods’ 的值：

{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}

select json\_extract\_path   ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}  
' , 'goods' )  ;   // 第二个参数表示获取键为goods的值

#输出：



json\_extract\_path   函数和  pgsql  提供的操作符  #>  是一样的。

select ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}  
') ::json #> '{goods}'

两者的输出是一致的。

同样我们要输出 键quantity 下键max 的值：

select json\_extract\_path   ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}  
' , 'quantity','max' ) ;

-- 或

select ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}  
') ::json #> '{quantity, max}'

两者输出是一样的。

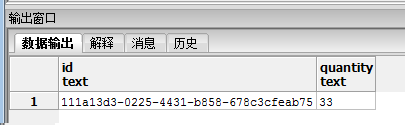


这几个函数是可以联合使用的。

比如我们要查询 键“goods” 下weight =2 的id 和quantity 值，语句如下：

select jae::json->>'id' as id, jae::json->>'quantity' as quantity from json\_array\_elements (  
json\_extract\_path ('  
{  
"goods":  
[   
{"id": "676a13d3-0225-4431-b858-678c3cfeab74", "weight": "1", "quantity": "9999999"},  
{"id": "111a13d3-0225-4431-b858-678c3cfeab75", "weight": "2", "quantity": "33"}  
],   
"quantity": {"max": "150", "min": "2"}  
}  
' , 'goods' ) ) as jae where jae::json->> 'weight' = '2'

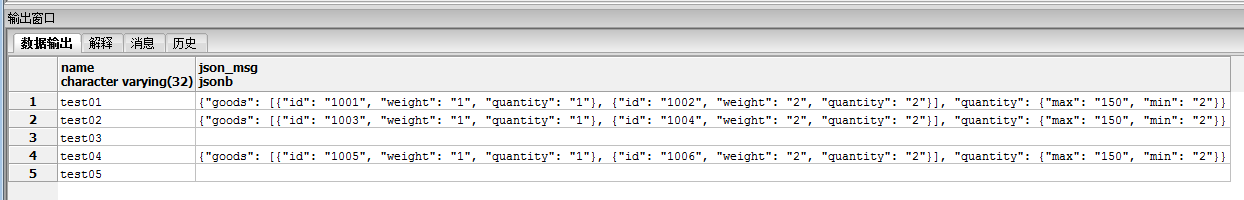
#输出：



 上述的json语句我们可以当做字段来使用，就相当于对表记录进行操作了。

接下来我们同个一个例子讲解json在表中的用法：

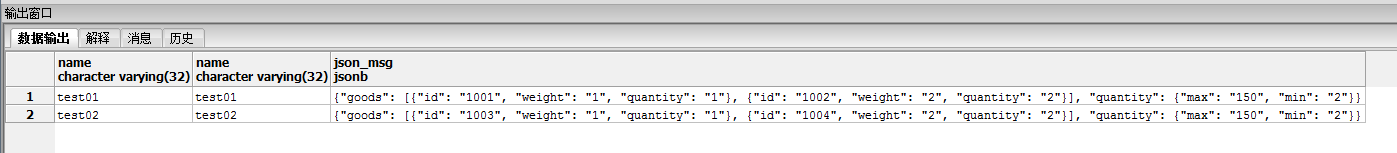
示例：查询表中jsonb\_msg字段中goods下id=1003和1002的记录，表中输入如下图：



 sql查询语句：

select name,\* from upgrade\_test.test1 test  
where   
( select count(\*) from jsonb\_array\_elements (  
jsonb\_extract\_path(test.json\_msg , 'goods' ) ) as jae where jae::json->> 'id' in ('1001','1003') ) > 0 ;

#输出：



效率还行：

Total query runtime: 11 msec  
检索到 2 行。

**--6 关于索引**

  GIN  索引支持 jsonb 类型，支持大的 jsonb 表中基于 keys 或者 key/values 模式的检索。

   默认的 GIN 索引模式支持带有 @>, ?, ?& 和 ?| 操作的查询，关于这些操作符的含义参考本文的附录。

   假如有一个文档：

{

"guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",

"name": "Angela Barton",

"is\_active": true,

"company": "Magnafone",

"address": "178 Howard Place, Gulf, Washington, 702",

"registered": "2009-11-07T08:53:22 +08:00",

"latitude": 19.793713,

"longitude": 86.513373,

"tags": [

    "enim",

    "aliquip",

    "qui"

]

}

我们将表名定义为 api,  jsonb 字段为 jdoc，创建如下索引

CREATE INDEX idx\_gin\_api\_jdoc ON api USING gin (jdoc);

那么如下的查询可以使用索引

-- Find documents in which the key "company" has value "Magnafone"

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';

备注：上面这个例子来自手册。

# [PostgreSQL的jsonb类型中包含数组的 增加、遍历、查找、修改](http://www.cnblogs.com/23cm/p/5527050.html)

Postgresql 版本 ：9.5

**表结构**

CREATE TABLE person(id serial, info jsonb);

JSONB结构

[{"num":"学号","name":"姓名","score":"成绩"}, {"num":"学号","name":"姓名","score":"成绩"}]

**插入**

INSERT INTO person (info) VALUES ('[{"num":"1","name":"张三","score":"90"}]'::jsonb);

增加

UPDATE person SET info = info || '[{"num":"2","name":"李四","score":"91"}]'::jsonb;

**遍历**

SELECT t.\* FROM person, jsonb\_to\_record(info) AS t(num text, name text, score text) WHERE person.id=**1**;

或

CREATE TABLE rt\_person\_info(no text, name text, score text);

SELECT t.\* FROM person, jsonb\_populate\_recordset(null::rt\_person\_info, info) WHERE person.id=**1**;

查找

SELECT \* FROM person WHERE info @> '[{"num":"1"}]'::jsonb;

修改

UPDATE person t1 SET info = jsonb\_set(info, array[(SELECT ORDINALITY::INT - 1 FROM person t2, jsonb\_array\_elements(info) WITH

ORDINALITY WHERE t1.id = t2.id AND value->>'num' = '1')::text, 'score'::text], '"92"') WHERE id = '1'

首先， **hstore** 是一个扩展模块，它允许你保存 **key=>values** 键值对，且键值都只能是 **text** s类型（但是，值也允许sql的NULL）

json与jsonb 允许你保存一个有效的json值( [***定义***](http://json.org/) ).

例如，以下都是有效的json表示方式: **null** , **true** , **[1, false, "string", {"foo":"bar"}]** ,**{"foo":"bar", "baz":[null]}** .

相比json, hstore只是它的一个很小的子集(但是，如果你只需要这个子集，也OK的.)

json与jsonb的区别主要是它们的存储方式：

* **json** 是保存为文本格式的
* **jsonb** 是保存为二进制格式的

这主要有三方面的影响：

* **jsonb** 通常比 **json** 占用更多的磁盘空间（有些情况不是）
* **jsonb** 比 **json** 的写入更耗时间
* **json** 的操作比 **jsonb** 的操作明显更耗时间（在操作一个 **json** 类型值时需要每次都去解析）

当 **jsonb** 将在未来稳定版发行可用时，这有两个主要使用情况，你很容易在他们之间选择的：

**json**

jsonb

官方文档上说:

有两个JSON数据类型：json和jsonb。它们接受几乎 相同的值组作为输入。它们实际的主要差别是效率。json 数据类型存储输入文本的精确拷贝，处理函数必须在每个执行上重新解析；而jsonb数据以分解的二进制格式存储，这使得它由于添加了转换机制而在输入上稍微慢些，但是在处理上明显更快，因为不需要重新解析。jsonb也支持索引，这也是一个明显的优势。

因为json类型存储输入文本的精确拷贝，它将保存令牌间语义上无关紧要的空格，和JSON对象中键的顺序。另外，如果值中的一个JSON对象多次包含相同的键，那么保存所有的键/值对。（处理函数将最后一个值当做操作值。）相比之下， jsonb不保存空格，也不保存对象键的顺序，并且不保存重复对象键。如果在输入中指定了重复的键，那么只保存最后一个值。

## json（jsonb) 的常用函数及操作符

[***functions-json***](https://www.postgresql.org/docs/current/static/functions-json.html)

### ->

右操作符为int： 获取JSON数组元素（索引从0开始）

**select** '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2;

?column?

-------------

{"c":"baz"}

(1 row)

Time: 1.240 ms

右操作符为text： 通过键获取json值.

SELECT '{"a": {"b":"foo"}}'::json->'a';

?column?

-------------

{"b":"foo"}

(1 row)

Time: 0.685 ms

### ->>

右操作符为int： 获取JSON数组元素为text

**SELECT** '[1,2,3]'::**json**->>2;

?column?

----------

3

(1 row)

Time: 0.530 ms

右操作符为text： 通过键获取json值为text

**SELECT** '{"a":1,"b":2}'::**json**->>'b';

?column?

----------

2

(1 row)

Time: 0.585 ms

### #>

右操作符为: **text[]** , 在指定的路径获取JSON对象。

SELECT '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}';

?column?

--------------

{"c":"foo"}

(1 row)

Time: 0.665 ms

即在获取 **a.b** 的值

### #>>

右操作符为: **text[]** , 在指定的路径获取JSON对象为text

**SELECT** '{"a":[1,2,3],"b":[4,5,6]}'::**json**#>>'{a,2}';

?column?

----------

3

(1 row)

Time: 0.556 ms

即获取 **a[2]** 的值并转为text.

### jsonb 操作符

#### @>

右操作数的类型： **jsonb** , 左侧的JSONB的是否包含右侧的.

**SELECT** '{"a":1, "b":2}'::jsonb @>'{"b":2}'::jsonb;

?column?

----------

t

(1 row)

Time: 0.599 ms

#### <@

右操作数的类型： **jsonb** , 右侧的JSONB的是否包含左侧的.

**SELECT** '{"a":1, "b":2}'::jsonb <@'{"b":2}'::jsonb;

?column?

----------

f

(1 row)

Time: 0.435 ms

#### ?

右操作符: **text** , 该字符串是否存在于json的顶级key中.

**SELECT** '{"a":1, "b":2}'::jsonb ?'b';

?column?

----------

t

(1 row)

Time: 0.551 ms

#### ?|

右操作符： **text[]** ，这些元素之一是否存在于json的顶级key中.

**SELECT** '{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'ceeee', 'e'];

?column?

----------

t

(1 row)

Time: 0.315 ms

#### ？&

右操作符： **text[]** ，所有这些元素是否存都在于json的顶级key中.

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb ?& array['a', 'b'];

?column?

----------

t

(**1** row)

Time: 36.143 ms

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb ?& array['a', 'b', 'c'];

?column?

----------

f

(**1** row)

Time: 0.370 ms

#### ||

右操作符: **jsonb** , 拼接两个 **jsonb** 生成一个新的 **jsonb**

[**local**]:5432 sky@sky=# SELECT '["a", "b", {"hello":"world"}]'::jsonb || '["c", "d", {"hello":"world"}]'::jsonb;

?column?

--------------------------------------------------------------

["a","b", {"hello":"world"},"c","d", {"hello":"world"}]

(**1** row)

Time: 0.359 ms

[**local**]:5432 sky@sky=#

#### -

右操作符： **text** ，从左操作数中删除K/V或者字符串元素。

[**local**]:5432 sky@sky=# SELECT '{"a": "b"}'::jsonb - 'a';

?column?

----------

{}

(**1** row)

Time: 0.357 ms

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb - 'a';

?column?

----------

["b"]

(**1** row)

Time: 0.359 ms

```

右操作符：`int`， 删除指定索引的元素（负数表示从结尾开始）

```bash

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb - (**-1**);

?column?

----------

["a"]

(**1** row)

Time: 0.319 ms

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb - 0;

?column?

----------

["b"]

(**1** row)

Time: 0.319 ms

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb - 1;

?column?

----------

["a"]

(**1** row)

Time: 0.305 ms

[**local**]:5432 sky@sky=# SELECT '["a", "b"]'::jsonb - 2;

?column?

------------

["a","b"]

(**1** row)

Time: 0.312 ms

[**local**]:5432 sky@sky=#

#### #-

右操作符： **text[]** ， 删除字段或指定路径的元素.

[**local**]:5432 sky@sky=# SELECT '["a", {"b":1}]'::jsonb #- '{1,b}';

?column?

-----------

["a", {}]

(**1** row)

Time: 0.460 ms

[**local**]:5432 sky@sky=# SELECT '["a", {"b":1}]'::jsonb #- '{0}';

?column?

------------

[{"b": 1}]

(**1** row)

Time: 0.329 ms

## 常用json函数

### row\_to\_json()

localhost:5433 sky@sky=# **SELECT** \* from test\_json ;

id | hello

----+---------

1 | hello

1 | hello2

2 | hello-2

2 | hello-3

(4 rows)

Time: 0.203 ms

localhost:5433 sky@sky=# **SELECT** row\_to\_json(test\_json) from test\_json ;

row\_to\_json

----------------------------

{"id":1,"hello":"hello"}

{"id":1,"hello":"hello2"}

{"id":2,"hello":"hello-2"}

{"id":2,"hello":"hello-3"}

(4 rows)

Time: 0.229 ms

localhost:5433 sky@sky=# SELECT \* from article ;

id | content

----+-----------------------

1 | hello article **content**

(1 row)

Time: 0.199 ms

localhost:5433 sky@sky=# SELECT \* from tags ;

aid | name

-----+------

1 | tag1

1 | tag2

(2 rows)

Time: 0.210 ms

localhost:5433 sky@sky=# select row\_to\_json(t)

**from** (

**select** id,

(

**select** **array\_to\_json**(array\_agg(row\_to\_json(d)))

**from** (

**select** name

**from** tags

wheretags.aid = article.id

) d

) **as** tags

**from** article

whereid = 1

) t;

row\_to\_json

---------------------------------------------------

{"id":1,"tags":[{"name":"tag1"},{"name":"tag2"}]}

(1 row)

Time: 0.349 ms

localhost:5433 sky@sky=#

localhost:5433 sky@sky=# select row\_to\_json(t) from ( select \*, ( SELECT array\_to\_json(array\_agg(name)) **as** name from tags where aid = article.id) **as** tags from article ) **as** t ;

row\_to\_json

-------------------------------------------------------------------

{"id":1,"content":"hello article content","tags":["tag1","tag2"]}

(1 row)

Time: 0.304 ms

localhost:5433 sky@sky=#

## json(jsonb)中的CRUD

添加jsonb的字段

localhost:5433 sky@sky=# create TABLE test\_json(hello jsonb);

CREATE TABLE

Time: 5.642 ms

localhost:5433 sky@sky=# INSERT INTO test\_json VALUES ('{"hello":"hello-value", "wolrd":"world-value"}');

INSERT 0 1

Time: 1.722 ms

localhost:5433 sky@sky=# SELECT \* from test\_json ;

hello

--------------------------------------------------

{"hello":"hello-value","wolrd":"world-value"}

(1 row)

Time: 0.179 ms

localhost:5433 sky@sky=# UPDATE test\_json set hello = jsonb\_set(hello, '{hello}', '"hello-new-value"'::text::jsonb, true);

UPDATE 1

Time: 0.994 ms

localhost:5433 sky@sky=# SELECT \* from test\_json ;

hello

------------------------------------------------------

{"hello":"hello-new-value","wolrd":"world-value"}

(1 row)

Time: 0.174 ms

localhost:5433 sky@sky=#

删除jsonb的某字段

localhost:5433 sky@sky=# UPDATE test\_json set hello = (hello - 'hello');

UPDATE 1

Time: 0.883 ms

localhost:5433 sky@sky=# SELECT \* from test\_json ;

hello

--------------------------

{"wolrd":"world-value"}

(1 row)

Time: 0.185 ms

localhost:5433 sky@sky=#

# PG中JSONB存储和访问效率（PG JSON系列5）

已有 766 次阅读 2016-10-18 04:12 |个人分类:[postgresql](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&classid=130971&view=me)|系统分类:[科研笔记](http://blog.sciencenet.cn/home.php?mod=space&do=blog&view=all&uid=419883&catid=1)|关键词:POSTGRESQL, JSONB

前4篇博文，大概写了一下如何使用PG的JSONB类型：

1. [使用PostgreSQL中的row\_to\_json()直接获得JSON（PG JSON系列1）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1007839)
2. [JSON/JSONB如何做增删改查（PG JSON系列2）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1008040)
3. [组合使用PostgreSQL的ARRAY和JSONB数据类型（PG JSON系列3）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1008644)
4. [PG中验证JSON Schema（PG JSON系列4）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1009248)

这些数据量都还没上规模，这一篇写写上规模之后的JSONB的存储和访问效率。

测试环境：CENTOS 7 64bit, PG 9.6.0，三星SSD 840，CPU 酷睿G640,2.8GHz, VirtualBox 5.1.6

我们的第1个问题是：

1. **相对于普通表，JSONB的存储大小是否过大？还是变化不大。**

Alexander Korotkov在 <https://www.pgcon.org/2015/schedule/track/Hacking/784.en.html> 说：

The first problem of jsonb is its size overhead (4-5 times) in comparison with storing decomposed json data in multiple plain tables, whereas binary format of jsonb has very low storage overhead with regard to a plain text (<4%). The overhead comes mainly from redundant storing of keys names, which can be quite long, in each document.

可以这么理解：二进制形式存储的jsonb的存储比普通表大4-5倍，事实如何？我们来做个实验。

回到[组合使用PostgreSQL的ARRAY和JSONB数据类型（PG JSON系列3）](http://blog.sciencenet.cn/home.php?mod=space&uid=419883&do=blog&quickforward=1&id=1008644)的例子，即照片中的每个人的信息，如下图：



如果用普通表设计为（我们略作修改，把attendanceYear，graduationYear设置为数值型）：

|  |
| --- |
| -- Table: R\_PIC\_PERSON\_PLAINTABLE  -- 照片中的人的信息表,用普通关联表实现  DROP TABLE IF EXISTS R\_PIC\_PERSON\_PLAINTABLE CASCADE;  CREATE TABLE R\_PIC\_PERSON\_PLAINTABLE (  id serial,-- 照片id  position text, -- 如"前1排左1"  name text,--如"张三"  phone text,  city text,  organization text,  attendanceYear numeric,  graduationYear numeric  )WITH (OIDS=FALSE); |

我们使用两种JSONB的存储形式，第1种：

|  |
| --- |
| -- Table: R\_PIC\_PERSON\_JSONB\_NOARRAY  -- 照片中的人的信息表，用JSONB实现，非JSONB数组形式，即一行对应普通表的一行  DROP TABLE IF EXISTS R\_PIC\_PERSON\_JSONB\_NOARRAY CASCADE;  CREATE TABLE R\_PIC\_PERSON\_JSONB\_NOARRAY (  id serial,-- 照片id  person jsonb NOT NULL -- 格式为：{"position":"前1排左1","name":"张晓陆","phone":"18910180100","city":"济南","organization":"中科院计算所","attendanceYear":2012,"graduationYear":2016}  )WITH (OIDS=FALSE); |

第2种：

|  |
| --- |
| -- Table: R\_PIC\_PERSON\_JSONB\_USEARRAY  -- 照片中的人的信息表，用JSONB实现，JSONB数组形式，即一行对应普通表的多行  DROP TABLE IF EXISTS R\_PIC\_PERSON\_JSONB\_USEARRAY CASCADE;  CREATE TABLE R\_PIC\_PERSON\_JSONB\_USEARRAY (  id serial,-- 照片id  persons jsonb NOT NULL -- 格式为：[{"position":"前1排左1","name":"张晓陆","phone":"18910180100","city":"济南","organization":"中科院计算所","attendanceYear":2012,"graduationYear":2016},{"position":"前1排左2","name":"张晓陆","phone":"18910180100","organization":"中科院计算所","attendanceYear":2012,"graduationYear":2016}]  )WITH (OIDS=FALSE); |

第1种JSONB的形式，一行对应一行普通表，好比1张照片中有20个人，则分别有20行。第2种JSONB的形式，即一张照片只有一行。

然后我们灌测试数据（参考francs的 [PostgreSQL9.4: jsonb 性能测试](http://francs3.blog.163.com/blog/static/40576727201452293027868/) ）：

20万条，普通表R\_PIC\_PERSON\_PLAINTABLE（假设平均1张照片20个人）：

|  |
| --- |
| insert into R\_PIC\_PERSON\_PLAINTABLE  select r/20,  round(random()\*200000) || '\_position',  round(random()\*200000) || '\_name',  round(random()\*200000) || '\_phone',  round(random()\*200000) || '\_city',  round(random()\*200000) || '\_organization',  round(random()\*200000) ,  round(random()\*200000)      from generate\_series(1,200000) as r; |

20万条，第1类JSONB表R\_PIC\_PERSON\_JSONB\_NOARRAY：

|  |
| --- |
| insert into R\_PIC\_PERSON\_JSONB\_NOARRAY  SELECT id, row\_to\_json(R\_PIC\_PERSON\_PLAINTABLE)::jsonb - 'id' from R\_PIC\_PERSON\_PLAINTABLE; |

假设平均一张照片20个人，插入20万/20=1万条JSONB表R\_PIC\_PERSON\_JSONB\_USEARRAY：

|  |
| --- |
| insert into R\_PIC\_PERSON\_JSONB\_USEARRAY  select id,array\_to\_json(array\_agg(person))     from R\_PIC\_PERSON\_JSONB\_NOARRAY     group by id; |

上面是基于20万条，我们分别就2000，2万，20万，200万，2000万，分别做几组实验，分别用：

jsontestdb=> \dt+ R\_PIC\_PERSON\_PLAINTABLE;

jsontestdb=> \dt+ R\_PIC\_PERSON\_JSONB\_NOARRAY;

jsontestdb=> \dt+ R\_PIC\_PERSON\_JSONB\_USEARRAY;

获得各自的存储大小，得出如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 2K | 2万 | 20万 | 200万 | 2000万 |
| PLAINTABLE | 248 | 2264 | 23 | 240 | 2521 |
| JSONB\_NOARRAY | 496 | 4856 | 49 | 488 | 5042 |
| JSONB\_USEARRAY | 136 | 1176 | 13 | 156 | 异常 |
| 单位 | KB | KB | MB | MB | MB |

最后2000万那列的R\_PIC\_PERSON\_JSONB\_USEARRAY表，生成时出现了异常（回头查一下原因，大家也测试）：

LOG:  redo starts at 2/5918E7B8

invalid record length at 2/591902D8: wanted 24, got 0

redo done at 2/5918E7B8

**结论：总体上来讲本例中，在没有建索引的情况下，R\_PIC\_PERSON\_JSONB\_NOARRAY表大概是普通的R\_PIC\_PERSON\_PLAINTABLE表的2倍大的空间，R\_PIC\_PERSON\_JSONB\_USEARRAY表则为R\_PIC\_PERSON\_JSONB\_NOARRAY表的50-65%左右。**

那我们看看建立索引的情况下的大小，由于JSONB可建立Btree，Hash和GIN索引，我们先看看在三张表上均建立Btree索引的情况：

|  |
| --- |
| CREATE INDEX idx\_btree\_plaintbl ON R\_PIC\_PERSON\_PLAINTABLE USING BTREE (name);  CREATE INDEX idx\_btree\_jsonb\_noarray ON R\_PIC\_PERSON\_JSONB\_NOARRAY USING BTREE ((person->'name'));  CREATE INDEX idx\_btree\_jsonb\_userarray ON R\_PIC\_PERSON\_JSONB\_USEARRAY USING BTREE ((persons)); |

获得各自的索引存储大小，得出如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2K | 2万 | 20万 | 200万 |
| PLAINTABLE | 0.078 | 0.617 | 6.039 | 60.17 |
| JSONB\_NOARRAY | 0.093 | 0.789 | 7.773 | 77.48 |
| JSONB\_USEARRAY | 0.156 | 1.609 | 19.539 | 244.11 |
| 单位 | MB | MB | MB | MB |

Hash索引就不测了，然后在JSONB上测一下GIN：

|  |
| --- |
| CREATE INDEX idx\_btree\_plaintbl ON R\_PIC\_PERSON\_PLAINTABLE USING BTREE (name);  CREATE INDEX idx\_gin\_jsonb\_noarray ON R\_PIC\_PERSON\_JSONB\_NOARRAY USING GIN ((person->'name'));  CREATE INDEX idx\_gin\_jsonb\_userarray ON R\_PIC\_PERSON\_JSONB\_USEARRAY USING GIN ((persons)); |

获得各自的索引存储大小，得出如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2K | 2万 | 20万 | 200万 |
| PLAINTABLE | 0.078 | 0.617 | 6.039 | 60.17 |
| JSONB\_NOARRAY | 0.109 | 0.960 | 9.578 | 62.58 |
| JSONB\_USEARRAY | 0.617 | 6.117 | 44.97 | 476.8 |
| 单位 | MB | MB | MB | MB |

**综上，在单一字段上建立的Btree和GIN索引，R\_PIC\_PERSON\_PLAINTABLE表和R\_PIC\_PERSON\_JSONB\_NOARRAY表建立索引占用的存储空间略大一些（如上表中标红色的部分），但差别不大。**

然后我们比较一下GIN索引在R\_PIC\_PERSON\_JSONB\_NOARRAY和R\_PIC\_PERSON\_JSONB\_USEARRAY表上的所有字段上建立的索引，所占空间有何区别。由于2K,2万，20万，200万呈现出良好的线性，所以这回我们仅对200万行记录进行测试。

|  |
| --- |
| CREATE INDEX idx\_gin\_jsonb\_all\_noarray ON R\_PIC\_PERSON\_JSONB\_NOARRAY USING GIN ((person));  CREATE INDEX idx\_gin\_jsonb\_all\_userarray ON R\_PIC\_PERSON\_JSONB\_USEARRAY USING GIN ((persons)); |

结果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | 200万 |
| JSONB\_NOARRAY | |  |  | 481.3906 |
| JSONB\_USEARRAY | |  |  | 475.4844 |
| 单位 |  |  |  | MB |

**发现二者区别不大，JSONB\_NOARRAY全字段索引所占空间略大些。**

我们的第2个问题是：

**2.JSONB的建立索引后，访问效率提升是多少？**

在上面200万行的记录里，创建了如下全字段索引后:

|  |
| --- |
| CREATE INDEX idx\_gin\_jsonb\_all\_noarray ON R\_PIC\_PERSON\_JSONB\_NOARRAY USING GIN ((person));  CREATE INDEX idx\_gin\_jsonb\_all\_userarray ON R\_PIC\_PERSON\_JSONB\_USEARRAY USING GIN ((persons)); |

在R\_PIC\_PERSON\_JSONB\_NOARRAY 上面执行：

|  |
| --- |
| explain analyze select \* from R\_PIC\_PERSON\_JSONB\_NOARRAY where person @> '{"name":"1349018\_name"}';  QUERY PLAN  ---------------------------------------  Bitmap Heap Scan on r\_pic\_person\_jsonb\_noarray  (cost=59.50..6912.05 rows=2000 width=222  ) (actual time=0.167..0.170 rows=2 loops=1)    Recheck Cond: (person @> '{"name": "1349018\_name"}'::jsonb)    Heap Blocks: exact=2    ->  Bitmap Index Scan on idx\_gin\_jsonb\_all\_noarray  (cost=0.00..59.00 rows=2000 width=  0) (actual time=0.155..0.155 rows=2 loops=1)          Index Cond: (person @> '{"name": "1349018\_name"}'::jsonb)  Planning time: 0.065 ms  Execution time: 0.196 ms  (7 rows)  Time: 0.593 ms |

在R\_PIC\_PERSON\_JSONB\_USEARRAY 上面执行：

|  |
| --- |
| explain analyze select \* from R\_PIC\_PERSON\_JSONB\_USEARRAY where personS @> '[{"name":"1349018\_name"}]';        QUERY PLAN  -----------------------------  Bitmap Heap Scan on r\_pic\_person\_jsonb\_usearray  (cost=44.78..424.81 rows=100 width=1365  ) (actual time=0.099..0.114 rows=2 loops=1)    Recheck Cond: (persons @> '[{"name": "1349018\_name"}]'::jsonb)    Heap Blocks: exact=2    ->  Bitmap Index Scan on idx\_gin\_jsonb\_all\_userarray  (cost=0.00..44.75 rows=100 width  =0) (actual time=0.077..0.077 rows=2 loops=1)          Index Cond: (persons @> '[{"name": "1349018\_name"}]'::jsonb)  Planning time: 0.066 ms  Execution time: 0.140 ms  (7 rows)  Time: 0.571 ms |

二者均走了索引。

下面这个SQL语句，没走索引：

针对R\_PIC\_PERSON\_JSONB\_NOARRAY ：

|  |
| --- |
| explain analyze SELECT count(\*) FROM R\_PIC\_PERSON\_JSONB\_NOARRAY a WHERE (a.person->>'attendanceyear')::integer <1998;    QUERY PLAN  --------------  Aggregate  (cost=104167.56..104167.57 rows=1 width=8) (actual time=1356.895..1356.895 ro  ws=1 loops=1)    ->  Seq Scan on r\_pic\_person\_jsonb\_noarray a  (cost=0.00..102500.90 rows=666665 width=  0) (actual time=0.224..1355.766 rows=1991 loops=1)          Filter: (((person ->> 'attendanceyear'::text))::integer < 1998)          Rows Removed by Filter: 1998009  Planning time: 0.052 ms  Execution time: 1356.922 ms  (6 rows)  Time: 1357.409 ms |

针对R\_PIC\_PERSON\_JSONB\_USEARRAY：

|  |
| --- |
| explain analyze WITH rows\_filtered AS (    SELECT DISTINCT id FROM (  select a.id, b.\* from R\_PIC\_PERSON\_JSONB\_USEARRAY a, jsonb\_to\_recordset(a.persons) as b(position text,name text, phone text, city text, organization text, attendanceYear numeric, graduationYear numeric)    ) c WHERE c.attendanceYear < 1998)  SELECT count(\*) FROM R\_PIC\_PERSON\_JSONB\_USEARRAY a WHERE a.id IN (SELECT \* FROM rows\_filtered);  QUERY PLAN  ------------------------  Aggregate  (cost=666790.68..666790.69 rows=1 width=8) (actual time=5306.279..5306.279 ro  ws=1 loops=1)    CTE rows\_filtered      ->  Unique  (cost=626533.97..643034.14 rows=100001 width=4) (actual time=5214.361..5  215.161 rows=1975 loops=1)            ->  Sort  (cost=626533.97..634784.06 rows=3300033 width=4) (actual time=5214.3  59..5214.633 rows=1991 loops=1)                  Sort Key: a\_1.id                  Sort Method: quicksort  Memory: 142kB                  ->  Nested Loop  (cost=0.00..179001.59 rows=3300033 width=4) (actual tim  e=1.558..5212.525 rows=1991 loops=1)                        ->  Seq Scan on r\_pic\_person\_jsonb\_usearray a\_1  (cost=0.00..21000  .01 rows=100001 width=1365) (actual time=0.002..140.555 rows=100001 loops=1)                        ->  Function Scan on jsonb\_to\_recordset b  (cost=0.00..1.25 rows=3  3 width=0) (actual time=0.050..0.050 rows=0 loops=100001)                              Filter: (attendanceyear < '1998'::numeric)                              Rows Removed by Filter: 20    ->  Hash Join  (cost=2254.52..23631.54 rows=50000 width=0) (actual time=5217.060..5305  .864 rows=1975 loops=1)          Hash Cond: (a.id = rows\_filtered.id)          ->  Seq Scan on r\_pic\_person\_jsonb\_usearray a  (cost=0.00..21000.01 rows=100001  width=4) (actual time=0.028..68.248 rows=100001 loops=1)          ->  Hash  (cost=2252.02..2252.02 rows=200 width=4) (actual time=5216.963..5216.9  63 rows=1975 loops=1)                Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage:  86kB                ->  HashAggregate  (cost=2250.02..2252.02 rows=200 width=4) (actual time=5  216.258..5216.586 rows=1975 loops=1)                      Group Key: rows\_filtered.id                      ->  CTE Scan on rows\_filtered  (cost=0.00..2000.02 rows=100001 width  =4) (actual time=5214.363..5215.643 rows=1975 loops=1)  Planning time: 0.171 ms  Execution time: 5306.357 ms  (21 rows)  Time: 5307.197 ms |

**结论：建立JSONB的索引后，对字符串相等匹配会有效，数值类操作无效，这是因为就算取到数值型的jsonb ，还得要做转换（如上面的(a.person->>'attendanceyear')::integer <1998)。在上面的例子里，从200万行的数据里查询，使用索引后，从全表扫描的Execution time为1356.922 ms降低到0.196 ms，提升还是巨大的。**

PG 9.4版本里面，增强了对json数据的支持，受到了很大关注。9.4之前，PG已经原生支持json数据类型了，但只是用字符串的形式存储和处理。这样做天然有性能上的缺点：每次对json字符串里面的数据进行查询，一般需要全表扫描加字符串匹配，效率很低。当然也可以在存储json的字符串字段上创建GIN索引，但需要对查询中用到的json的key或value创建单独索引，造成要被动维护很多索引。所以，这种json类型，只适用于把PG单纯作为数据存储，只读入读出数据，不对数据进行限定key或value查询的场景。  
  
PG 9.4中引入了jsonb类型。其特点是，将json数据中的key和value进行解析，转换为PG的基本数据类型，包括数字，字符串和布尔类型等；同时，增加了对应的GIN处理函数，可以将json中的所有key和value转换为GIN索引的key。这样，只用一个GIN索引，即可实现对所有key或value的条件查询。下面我们分析一下jsonb的使用方法和内核实现。  
使用  
创建含jsonb类型的表方法如下所示：  
  
=> create table test\_jsonb(col\_jsonbjsonb);  
CREATE TABLE  
=> insert into test\_jsonbvalues('{"product": "PostgreSQL", "version": 9.4,"platform":["win", "linux","unix"]}'::jsonb);  
INSERT 0 1  
  
创建GIN索引的方法如下：  
--创建jsonb\_ops索引：  
=> create index idx\_jsonb on test\_jsonbusing gin (col\_jsonb);  
CREATE INDEX  
--创建jsonb\_ops\_path索引：  
=> create index idx\_jsonb\_path ontest\_jsonb using gin (col\_jsonb jsonb\_path\_ops);  
CREATE INDEX  
  
可以使用下面的查询得到含有<product, ProgreSQL>键值对的行：  
=>select \* from test\_jsonb wherecol\_jsonb @> '{"product" : "PostgreSQL"}';  
                                    col\_jsonb  
---------------------------------------------------------------------------------  
{"product":"PostgreSQL", "version": 9.4, "platform":["win", "linux", "unix"]}  
  
内核实现  
先分析一下jsonb是如何从字符串，变成特殊的二进制形式存入磁盘的。追踪一下jsonb插入的过程，可以看到PG所调用的函数流程如下。  
jsonb\_in->jsonb\_from\_cstring->pg\_parse\_json->JsonbValueToJsonb  
  
其中，pg\_parse\_json先把用户输入的字符串，通过编译器转换为一个树形结构（每个节点的类型为JsonbValue）。然后JsonbValueToJsonb在这个结构基础上，转换为存入磁盘的格式。从convertJsonbObject函数可以看出，转换为磁盘格式的策略为：从树形结构的根部开始遍历，递归进行广度优先遍历。对于同一父亲下面的子键值，将所有键名（字符串）长度写入buffer中预留的头部，随后将键名依次写入buffer中。最后再以相似的方式写入键所对应的所有值（值如果是json对象，则递归调用）。这样，读入buffer的头部，就可以遍历出所有键名的位置，得到键名。再从读第一个键值开始，读入对应的值或子键，最终得到整个树（见JsonbIteratorNext）。  
  
采用这种存储方式，jsonb所占用的存储空间比原来支持的json类型要多一些。其实，jsonb的核心优势在于快速和灵活的索引。从前面创建index的语句可以看到，jsonb支持两种特有的GIN索引jsonb\_ops和jsonb\_path\_ops。我们知道，GIN索引建立时，会先通过内建函数从表中每行数据的索引字段的值中，抽取键（key），一个字段值一般可抽取多个key。然后，将每个key与含有此key的所有行的ID组成键值对，再将它们插入b树索引供查询。那么这两种GIN索引有什么区别呢？  
  
它们的区别在于，生成GIN key的方式不同。jsonb\_ops调用gin\_extract\_jsonb函数生成key，这样每个字段的json数据中的所有键和值都被转成GIN的key；而jsonb\_path\_ops使用函数gin\_extract\_jsonb\_path抽取：如果将一个jsonb类型的字段值看做一颗树，叶子节点为具体的值，中间节点为键，则抽取的每个键值实际上时每个从根节点到叶子节点的路径对应的hash值。  
  
不难推测，jsonb\_path\_ops索引的key的数目和jsonb的叶子节点数有关，用叶子节点的路径做查询条件时会比较快（这也是这种索引唯一支持的查询方式）；而jsonb\_ops索引的key的数目与jsonb包含的键和值（即树形结构的所有节点）的总数有关，可以用于路径查询之外的其他查询。

[关于PostgreSQL中的多字段索引之三(gin篇)](http://blog.chinaunix.net/uid-20726500-id-5090449.html)2015-06-21 00:27:49

分类： Mysql/postgreSQL

前两篇讲的btree和gist的多字段索引，本篇顺理成章地讲一下gin的多字段索引。  
前两篇请参考这里:  
<http://blog.chinaunix.net/uid-20726500-id-5088527.html>  
<http://blog.chinaunix.net/uid-20726500-id-5090166.html>

## 1. gin多字段索引的特征

不像gist和btree，gin天生就适合做多字段索引，不管查询条件覆盖所有索引字段还是仅仅覆盖一个子集，它都可以胜任。  
<http://www.postgres.cn/docs/9.3/indexes-multicolumn.html>  
---------------------------------------------------------------  
一个多字段的 GIN 索引可以用于那些查询条件包含索引字段子集的查询中。 不像B-tree 或 GiST，除了查询条件使用的索引字段外，索引的搜索效率是相同的。  
---------------------------------------------------------------

## 2. gin多字段索引的原理

gin多字段索引的原理是，索引中的每个key由字段编号加上元素值构成，这样的key作为一个整体构成倒排索引树。  
  
参考：src/backend/access/gin/README  
---------------------------------------------------------------  
...  
\* In a single-column index, a key tuple just contains the key datum, but  
**in a multi-column index, a key tuple contains the pair (column number,**  
**key datum) where the column number is stored as an int2**.  This is needed  
to support different key data types in different columns.  This much of  
the tuple is built by index\_form\_tuple according to the usual rules.  
The column number (if present) can never be null, but the key datum can  
be, in which case a null bitmap is present as usual.  (As usual for index  
tuples, the size of the null bitmap is fixed at INDEX\_MAX\_KEYS.)  
...  
---------------------------------------------------------------

## 3. 性能对比测试

为了和前面的btree和gist有个可比性，测试环境和数据还采用上一篇的标量数据，而不是采用gin特有的集合数据，所以要使用btree-gin扩展。  
测试环境详见  
<http://blog.chinaunix.net/uid-20726500-id-5088527.html>

### 3.1 创建c1+c2多字段gin索引

**点击(此处)折叠或打开**

### postgres=# create extension btree\_gin;

### CREATE EXTENSION

### postgres=# \timing

### Timing is on.

### postgres=# create index tb1\_idx5 on tb1 using gin(c1,c2);

### CREATE INDEX

### Time: 23119.722 ms

### postgres=# select pg\_size\_pretty(pg\_relation\_size('tb1\_idx5'));

### pg\_size\_pretty

### ----------------

### 129 MB

### (1 row)

### 3.2 c1+c2多字段gin索引处理多字段查询

**点击(此处)折叠或打开**

### postgres=# explain (analyze,buffers) select count(\*) from tb1 where c1=99 and c2=999;

### QUERY PLAN

### ----------------------------------------------------------------------------------------------------------------------------

### Aggregate (cost=404.70..404.71 rows=1 width=0) (actual time=15.797..15.798 rows=1 loops=1)

### Buffers: shared hit=179 read=2

### -> Bitmap Heap Scan on tb1 (cost=21.01..404.45 rows=99 width=0) (actual time=14.545..15.769 rows=92 loops=1)

### Recheck Cond: ((c1 = 99) AND (c2 = 999))

### Buffers: shared hit=179 read=2

### -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..20.99 rows=99 width=0) (actual time=14.144..14.144 rows=92 loops=1)

### Index Cond: ((c1 = 99) AND (c2 = 999))

### Buffers: shared hit=87 read=2

### Total runtime: 16.199 ms

### (9 rows)

这个速度没有btree和gist的多字段索引快，因为它的原理和BitmapAnd组合索引相似，要扫描2次索引再取交集，但比组合索引的50ms快。

### 3.3 c1+c2多字段gist索引处理单字段查询

**点击(此处)折叠或打开**

1. postgres=# explain (analyze,buffers) select count(\*) from tb1 where c1=99;
2. QUERY PLAN
3. -------------------------------------------------------------------------------------------------------------------------------------
4. Aggregate (cost=46880.08..46880.09 rows=1 width=0) (actual time=1569.289..1569.290 rows=1 loops=1)
5. Buffers: shared hit=142 read=39638
6. -> Bitmap Heap Scan on tb1 (cost=1097.08..46624.25 rows=102333 width=0) (actual time=37.484..1548.405 rows=99886 loops=1)
7. Recheck Cond: (c1 = 99)
8. Rows Removed by Index Recheck: 7363157
9. Buffers: shared hit=142 read=39638
10. -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..1071.50 rows=102333 width=0) (actual time=35.034..35.034 rows=99886 loops=1)
11. Index Cond: (c1 = 99)
12. Buffers: shared hit=78
13. Total runtime: 1569.548 ms
14. (10 rows)
15. postgres=# explain (analyze,buffers) select count(\*) from tb1 where c2=999;
16. QUERY PLAN
17. -------------------------------------------------------------------------------------------------------------------------------
18. Aggregate (cost=23488.40..23488.41 rows=1 width=0) (actual time=79.362..79.362 rows=1 loops=1)
19. Buffers: shared hit=1974 read=6910
20. -> Bitmap Heap Scan on tb1 (cost=110.81..23464.27 rows=9652 width=0) (actual time=7.228..77.332 rows=9892 loops=1)
21. Recheck Cond: (c2 = 999)
22. Buffers: shared hit=1974 read=6910
23. -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..108.39 rows=9652 width=0) (actual time=4.468..4.468 rows=9892 loops=1)
24. Index Cond: (c2 = 999)
25. Buffers: shared hit=1 read=11
26. Total runtime: 79.408 ms
27. (9 rows)

查询速度和gist的多字段索引差不多。

## 4. PostgreSQL9.4的测试结果

PostgreSQL 9.4对gin的性能做了很大提升，下面再在9.4上重复一下上面的测试。

### 4.1 数据准备

**点击(此处)折叠或打开**

1. chenhj=# create table tb1(c1 int,c2 int);
2. CREATE TABLE
3. chenhj=# insert into tb1 select round(random()\*100),round(random()\*1000) from generate\_series(1,10000000);
4. INSERT 0 10000000
5. chenhj=# select pg\_size\_pretty(pg\_table\_size('tb1'));
6. pg\_size\_pretty
7. ----------------
8. 346 MB
9. (1 row)
10. chenhj=# create extension btree\_gin;
11. CREATE EXTENSION
12. chenhj=# create index tb1\_idx5 on tb1 using gin(c1,c2);
13. CREATE INDEX
14. chenhj=# select pg\_size\_pretty(pg\_relation\_size('tb1\_idx5'));
15. pg\_size\_pretty
16. ----------------
17. 47 MB
18. (1 row)

9.4里gin索引的大小比9.3小了很多，只有9.3的三分之一。

### 4.2 性能测试

### ****查询条件包含c1+c2****

**点击(此处)折叠或打开**

1. chenhj=# explain (analyze,buffers) select count(\*) from tb1 where c1=99 and c2=999;
2. QUERY PLAN
3. ----------------------------------------------------------------------------------------------------------------------------
4. Aggregate (cost=423.95..423.96 rows=1 width=0) (actual time=3.399..3.399 rows=1 loops=1)
5. Buffers: shared hit=166
6. -> Bitmap Heap Scan on tb1 (cost=25.05..423.69 rows=103 width=0) (actual time=3.151..3.375 rows=117 loops=1)
7. Recheck Cond: ((c1 = 99) AND (c2 = 999))
8. Heap Blocks: exact=117
9. Buffers: shared hit=166
10. -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..25.03 rows=103 width=0) (actual time=3.125..3.125 rows=117 loops=1)
11. Index Cond: ((c1 = 99) AND (c2 = 999))
12. Buffers: shared hit=49
13. Planning time: 0.099 ms
14. Execution time: 3.448 ms
15. (11 rows)

比9.3快了很多，但仍然没有btree和gist的多字段索引快，不过差别也不是太大。  
  
**查询条件只包含c1**

**点击(此处)折叠或打开**

1. chenhj=# explain (analyze,buffers) select count(\*) from tb1 where c1=99;
2. QUERY PLAN
3. ------------------------------------------------------------------------------------------------------------------------------------
4. Aggregate (cost=46819.50..46819.51 rows=1 width=0) (actual time=590.436..590.437 rows=1 loops=1)
5. Buffers: shared hit=105 read=39583 written=700
6. -> Bitmap Heap Scan on tb1 (cost=981.50..46554.50 rows=106000 width=0) (actual time=49.895..570.300 rows=99790 loops=1)
7. Recheck Cond: (c1 = 99)
8. Heap Blocks: exact=39665
9. Buffers: shared hit=105 read=39583 written=700
10. -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..955.00 rows=106000 width=0) (actual time=31.225..31.225 rows=99790 loops=1)
11. Index Cond: (c1 = 99)
12. Buffers: shared hit=23
13. Planning time: 0.112 ms
14. Execution time: 590.496 ms
15. (11 rows)

速度是9.3的2倍多，也比gist和btree都快。  
  
**查询条件只包含c2**

**点击(此处)折叠或打开**

1. chenhj=# explain (analyze,buffers) select count(\*) from tb1 where c2=999;
2. QUERY PLAN
3. -------------------------------------------------------------------------------------------------------------------------------
4. Aggregate (cost=23539.39..23539.40 rows=1 width=0) (actual time=92.894..92.894 rows=1 loops=1)
5. Buffers: shared hit=1950 read=7130 written=3
6. -> Bitmap Heap Scan on tb1 (cost=95.12..23515.16 rows=9692 width=0) (actual time=5.672..90.431 rows=10178 loops=1)
7. Recheck Cond: (c2 = 999)
8. Heap Blocks: exact=9073
9. Buffers: shared hit=1950 read=7130 written=3
10. -> Bitmap Index Scan on tb1\_idx5 (cost=0.00..92.69 rows=9692 width=0) (actual time=2.671..2.671 rows=10178 loops=1)
11. Index Cond: (c2 = 999)
12. Buffers: shared hit=6 read=1
13. Planning time: 0.187 ms
14. Execution time: 94.261 ms
15. (11 rows)

和9.3差不多，和gist也差不多。

## 4.总结

1）当查询条件中同时出现多字段索引的所有条件时，多字段索引的速度优于多个字段单独建索引然后通过BitmapAnd组合起来。btree和gist的多字段索引又快于gin多字段索引。  
2）多字段索引用在只包含全部索引字段的子集时，除了btree在查询条件中多字段的第一个字段缺席时效率很差外，其它都表现良好。  
3）9.4的gin索引的大小比btree和gist小了太多。（但gin有别的问题，在做比较查询时有时效率不高，后面准备再通过例子讲解）  
4）9.4的gin索引比9.3的gin优化了太多了  
  
最后引用一下手册中关于使用多字段索引还是多索引组合的说明。加黑的那个限制似乎只对btree的多字段索引有效。  
-----------------------------------------------------------------------------------  
在大多数最简单的应用里，可能有多种索引组合都是有用的，数据库开发人员必须在使用哪个索引之间作出平衡。 有时候多字段索引是最好的，有时候创建一个独立索引并依靠索引组合是最好的。比如， 假如你的查询有时候只涉及字段x，有时候只涉及字段y，有时候两个字段都涉及， 那么你可能会选择在x和y上创建两个独立的索引， 然后依靠索引组合来处理同时使用两个字段的查询。**你也可以在(x, y)上创建一个多字段索引， 它在同时使用两个字段的查询通常比索引组合更高效，但是，正如我们在**[**Section 11.3**](http://postgres.cn/docs/9.3/indexes-multicolumn.html)**里面讨论的，它对那些只包含y的查询几乎没有用**，因此它不能是唯一一个索引。 一个多字段索引和y上的独立索引可能会更好。因为对那些只涉及x的查询， 可以使用多字段索引，但是它会更大，因此也比只在x上的索引更慢。最后一个选择是创建三个索引， 但是这种方法只有在表的更新远比查询少得多，并且所有三种查询都很普遍的情况下才是合理的。 如果其中一种查询比其它的少很多，那么你可能更愿意仅仅创建两种匹配更常见查询的索引。  
-----------------------------------------------------------------------------------

## PostgreSQL9.4: jsonb 性能测试

        json 特性的提升是9.4 的关键特性之一, 本人对于 json 的关注较少, 一方面由于之前版本的 json 并不十分成熟, 使用时需要配合使用外部模块如 PLV8, PLPerl 来弥补 JSON 功能的不足,一方面由于太懒没花精力研究; 但 9.4 版本的 JSON 功能完善很多,  jsonb 的出现带来了更多的函数, 更多的索引创建方式, 更多的操作符和更高的性能. 接下来通过一些例子来讲解, 希望更多的朋友能够了解并[**测试**](http://lib.csdn.net/base/softwaretest) PostgreSQL 的 json 功能.

**一 环境信息**

**--1.1 之前写的关于 json 的博客**

· [PostgreSQL 9.3Beta1：JSON 功能增强](http://francs3.blog.163.com/blog/static/40576727201341613630793/)

· [PostgreSQL 9.4: 新增 JSONB 数据类型](http://francs3.blog.163.com/blog/static/40576727201442264738357/)

**--1.2 测试环境**

硬件: 笔记本虚拟机

系统: RHEL 6.2

PG 版本: 9.4Beta1

**二生成测试数据**

**--2.1 测试表**

user\_ini:              基础数据表, 200 万数据.

tbl\_user\_json:    含有 json 数据类型表, 200 万数据

tbl\_user\_jsonb:  含有 jsonb 数据类型表,  200 万数据

**--2.2 创建基础数据测试表**

francs=> create table user\_ini(id int4 ,user\_id int8, user\_name character varying(64),create\_time timestamp(6) with time zone default clock\_timestamp());

CREATE TABLE

francs=> insert into user\_ini(id,user\_id,user\_name) select r,round(random()\*2000000), r || '\_francs' from generate\_series(1,2000000) as r;

INSERT 0 2000000

备注: 生成 200 万测试数据.

**--2.3 生成 json 测试数据**

francs=>  create table tbl\_user\_json(id serial, user\_info json);

CREATE TABLE

francs=>  insert into tbl\_user\_json(user\_info) select row\_to\_json(user\_ini) from user\_ini;

INSERT 0 2000000

Time: 63469.336 ms

**--2.4 生成 jsonb 测试数据**

francs=> create table tbl\_user\_jsonb(id serial, user\_info jsonb);

CREATE TABLE

francs=> insert into tbl\_user\_jsonb(user\_info) select row\_to\_json(user\_ini)::jsonb from user\_ini;

INSERT 0 2000000

Time: 78300.553 ms

备注:  从时间来看, jsonb 插入速度比 json 插入速度稍慢, 再来看下两个表的大小如何?

**--2.5 比较表大小**

francs=> \dt+ tbl\_user\_json

                       List of relations

 Schema |     Name      | Type  | Owner  |  Size  | Description

--------+---------------+-------+--------+--------+-------------

 francs | tbl\_user\_json | table | francs | 269 MB |

(1 row)

francs=> \dt+ tbl\_user\_jsonb

                        List of relations

 Schema |      Name      | Type  | Owner  |  Size  | Description

--------+----------------+-------+--------+--------+-------------

 francs | tbl\_user\_jsonb | table | francs | 329 MB |

(1 row)

**--2.6 查看几条测试数据**

francs=> select \* from tbl\_user\_jsonb limit 3;

 id |                                               user\_info

----+-------------------------------------------------------------------------------------------------------

  1 | {"id": 1, "user\_id": 116179, "user\_name": "1\_francs", "create\_time": "2014-06-21 11:54:38.371774+00"}

  2 | {"id": 2, "user\_id": 956659, "user\_name": "2\_francs", "create\_time": "2014-06-21 11:54:38.373425+00"}

  3 | {"id": 3, "user\_id": 1017031, "user\_name": "3\_francs", "create\_time": "2014-06-21 11:54:38.37344+00"}

(3 rows)

备注: 以上是生成的测试数据, 列几条出来,方便查阅, 接下来看一个查询.

**三 基于 jsonb 字段 key 值的检索效率**

**--3.1 根据 user\_info 字段的 user\_name key 检索**

francs=>  select \* from tbl\_user\_jsonb where user\_info->>'user\_name'= '1\_francs';

 id |                                               user\_info

----+-------------------------------------------------------------------------------------------------------

  1 | {"id": 1, "user\_id": 116179, "user\_name": "1\_francs", "create\_time": "2014-06-21 11:54:38.371774+00"}

(1 row)

**--3.2 执行计划和执行时间**

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->>'user\_name'= '1\_francs';

                                                     QUERY PLAN

---------------------------------------------------------------------------------------------------------------------

 Seq Scan on tbl\_user\_jsonb  (cost=0.00..72097.82 rows=10000 width=140) (actual time=0.033..2965.837 rows=1 loops=1)

   Filter: ((user\_info ->> 'user\_name'::text) = '1\_francs'::text)

   Rows Removed by Filter: 1999999

 Planning time: 1.657 ms

 Execution time: 2966.380 ms

(5 rows)

备注:  此时还没建索引,走的全表扫, 花了将近 3 秒.

**--3.3 创建索引**

francs=> create index idx\_gin\_user\_infob\_user\_name on tbl\_user\_jsonb using btree ((user\_info ->> 'user\_name'));

CREATE INDEX

**--3.4 再次查看  plan**

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->>'user\_name'= '1\_francs';

                                                                QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on tbl\_user\_jsonb  (cost=233.93..23782.62 rows=10000 width=140) (actual time=0.046..0.047 rows=1 loops=1)

   Recheck Cond: ((user\_info ->> 'user\_name'::text) = '1\_francs'::text)

   Heap Blocks: exact=1

   ->  Bitmap Index Scan on idx\_gin\_user\_infob\_user\_name  (cost=0.00..231.43 rows=10000 width=0) (actual time=0.035..0.035 rows=1 loops=1)

         Index Cond: ((user\_info ->> 'user\_name'::text) = '1\_francs'::text)

 Planning time: 0.144 ms

 Execution time: 0.101 ms

(7 rows)

备注: 创建索引后,  上述查询走了索引, 仅花  0.101 ms 完成检索, 挺给力!

**--3.5 根据 user\_info 字段的 user\_id 检索**

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->>'user\_id'= '1';

                                                       QUERY PLAN

------------------------------------------------------------------------------------------------------------------------

 Seq Scan on tbl\_user\_jsonb  (cost=0.00..72098.00 rows=10000 width=140) (actual time=2483.198..4289.888 rows=1 loops=1)

   Filter: ((user\_info ->> 'user\_id'::text) = '1'::text)

   Rows Removed by Filter: 1999999

 Planning time: 3.304 ms

 Execution time: 4292.158 ms

(5 rows)

Time: 4321.349 ms

备注: 没走索引,花了 4 秒多,因为没建这个 key 上的索引.

**四 使用 GIN 索引**

     可以给 jsonb 字段创建 GIN 索引, GIN 索引有两种模式, 默认模式支持  @>, ?, ?& 和 ?| 的索引查询, 我们这里使用默认模式.

**--4.1 删除之前索引,新建 gin 索引**

francs=> create index idx\_tbl\_user\_jsonb\_user\_Info on tbl\_user\_jsonb using gin (user\_Info);

CREATE INDEX

Time: 214253.873 ms

francs=> \di+ idx\_tbl\_user\_jsonb\_user\_Info

                                       List of relations

 Schema |             Name             | Type  | Owner  |     Table      |  Size  | Description

--------+------------------------------+-------+--------+----------------+--------+-------------

 francs | idx\_tbl\_user\_jsonb\_user\_info | index | francs | tbl\_user\_jsonb | 428 MB |

(1 row)

备注: 索引很大,创建很慢,一般不会这么建索引.

**--4.2 基于 key/value 检索可以使用索引**

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info @> '{"user\_id": 1017031}';

QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on tbl\_user\_jsonb  (cost=59.50..6637.58 rows=2000 width=140) (actual time=0.340..0.345 rows=1 loops=1)

   Recheck Cond: (user\_info @> '{"user\_id": 1017031}'::jsonb)

   Rows Removed by Index Recheck: 1

   Heap Blocks: exact=2

   ->  Bitmap Index Scan on idx\_tbl\_user\_jsonb\_user\_info  (cost=0.00..59.00 rows=2000 width=0) (actual time=0.319..0.319 rows=2 loops=1)

         Index Cond: (user\_info @> '{"user\_id": 1017031}'::jsonb)

 Planning time: 0.118 ms

 Execution time: 0.391 ms

(8 rows)

**--4.3 以下查询不走索引**

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->>'user\_name' ='4\_francs';

                                                     QUERY PLAN

---------------------------------------------------------------------------------------------------------------------

 Seq Scan on tbl\_user\_jsonb  (cost=0.00..72098.00 rows=10000 width=140) (actual time=0.036..4640.794 rows=1 loops=1)

   Filter: ((user\_info ->> 'user\_name'::text) = '4\_francs'::text)

   Rows Removed by Filter: 1999999

 Planning time: 1.101 ms

 Execution time: 4640.851 ms

(5 rows)

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->'user\_name' ?'4\_francs';

                                                     QUERY PLAN

--------------------------------------------------------------------------------------------------------------------

 Seq Scan on tbl\_user\_jsonb  (cost=0.00..72098.00 rows=2000 width=140) (actual time=0.187..5387.658 rows=1 loops=1)

   Filter: ((user\_info -> 'user\_name'::text) ? '4\_francs'::text)

   Rows Removed by Filter: 1999999

 Planning time: 0.382 ms

 Execution time: 5387.762 ms

(5 rows)

备注: 以上的 ? 操作没走索引, 但 ? 操作支持索引检索,创建以下索引.

**--4.4 删除之前索引并新建以下索引**

francs=> create index idx\_gin\_user\_info\_user\_name on tbl\_user\_jsonb using gin((user\_info -> 'user\_name'));

CREATE INDEX

francs=> explain analyze select \* from tbl\_user\_jsonb where user\_info->'user\_name' ?'4\_francs';

                                                               QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on tbl\_user\_jsonb  (cost=35.50..6618.58 rows=2000 width=140) (actual time=0.067..0.069 rows=1 loops=1)

   Recheck Cond: ((user\_info -> 'user\_name'::text) ? '4\_francs'::text)

   Heap Blocks: exact=1

   ->  Bitmap Index Scan on idx\_gin\_user\_info\_user\_name  (cost=0.00..35.00 rows=2000 width=0) (actual time=0.037..0.037 rows=1 loops=1)

         Index Cond: ((user\_info -> 'user\_name'::text) ? '4\_francs'::text)

 Planning time: 0.151 ms

 Execution time: 0.129 ms

(7 rows)

备注: 速度很快.

**五 对比 json 和 jsonb 的检索性能**

       文档上提到了 jsonb 的检索效率要高于 json 的检索效率, 下面通过例子测试.

**--5.1 删除之前创建的所有索引并创建函数索引**

francs=> create index idx\_gin\_user\_info\_id on tbl\_user\_json using btree (((user\_info ->> 'id')::integer));

CREATE INDEX

francs=> create index idx\_gin\_user\_infob\_id on tbl\_user\_jsonb using btree (((user\_info ->> 'id')::integer));

CREATE INDEX

备注: 为什么使用函数索引?  由于 --> 操作返回的是 text 类型,  接下来的查询会用到 id 字段比较, 需要转换成整型.

**--5.2 json 表范围扫描**

francs=>  explain analyze select id,user\_info->'id',user\_info->'user\_name' from tbl\_user\_json where (user\_info->>'id')::int4 > '1' and (user\_info->>'id')::int4 < '10000';

                                                              QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on tbl\_user\_json  (cost=190.94..22275.60 rows=10000 width=36) (actual time=2.417..60.585 rows=9998 loops=1)

   Recheck Cond: ((((user\_info ->> 'id'::text))::integer > 1) AND (((user\_info ->> 'id'::text))::integer < 10000))

   Heap Blocks: exact=167

   ->  Bitmap Index Scan on idx\_gin\_user\_info\_id  (cost=0.00..188.44 rows=10000 width=0) (actual time=2.329..2.329 rows=9998 loops=1)

         Index Cond: ((((user\_info ->> 'id'::text))::integer > 1) AND (((user\_info ->> 'id'::text))::integer < 10000))

 Planning time: 0.183 ms

 Execution time: 64.116 ms

(7 rows)

**--5.3 jsonb 表范围扫描**

francs=>  explain analyze select id,user\_info->'id',user\_info->'user\_name' from tbl\_user\_jsonb where (user\_info->>'id')::int4 > '1' and (user\_info->>'id')::int4 < '10000';

                                                              QUERY PLAN

---------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on tbl\_user\_jsonb  (cost=190.94..23939.63 rows=10000 width=140) (actual time=2.593..24.308 rows=9998 loops=1)

   Recheck Cond: ((((user\_info ->> 'id'::text))::integer > 1) AND (((user\_info ->> 'id'::text))::integer < 10000))

   Heap Blocks: exact=197

   ->  Bitmap Index Scan on idx\_gin\_user\_infob\_id  (cost=0.00..188.44 rows=10000 width=0) (actual time=2.494..2.494 rows=9998 loops=1)

         Index Cond: ((((user\_info ->> 'id'::text))::integer > 1) AND (((user\_info ->> 'id'::text))::integer < 10000))

 Planning time: 0.142 ms

 Execution time: 27.851 ms

(7 rows)

备注: 这里实验发现, jsonb 检索确实比 json 要快很多,  而本文开头插入数据时 jsonb 比 json 稍慢, 这也正好验证了 "jsonb 写入比 json 慢,但检索较 json 快的说法.", 我在之前的博客  [PostgreSQL 9.4: 新增 JSONB 数据类型](http://francs3.blog.163.com/blog/static/40576727201442264738357/)  有提到过.

**--六 附**

**6.1 json and jsonb Operators**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator | Right Operand Type | Description | Example | Example Result |
| -> | int | Get JSON array element (indexed from zero) | '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2 | {"c":"baz"} |
| -> | text | Get JSON object field by key | '{"a": {"b":"foo"}}'::json->'a' | {"b":"foo"} |
| ->> | int | Get JSON array element as text | '[1,2,3]'::json->>2 | 3 |
| ->> | text | Get JSON object field as text | '{"a":1,"b":2}'::json->>'b' | 2 |
| #> | text[] | Get JSON object at specified path | '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}' | {"c": "foo"} |
| #>> | text[] | Get JSON object at specified path as text | '{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}' | 3 |

**6.2  Additional jsonb Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Right Operand Type | Description | Example |
| = | jsonb | Are the two JSON values equal? | '[1,2,3]'::jsonb = '[1,2,3]'::jsonb |
| @> | jsonb | Does the left JSON value contain within it the right value? | '{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb |
| <@ | jsonb | Is the left JSON value contained within the right value? | '{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb |
| ? | text | Does the key/element string exist within the JSON value? | '{"a":1, "b":2}'::jsonb ? 'b' |
| ?| | text[] | Do any of these key/element strings exist? | '{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c'] |
| ?& | text[] | Do all of these key/element strings exist? | '["a", "b"]'::jsonb ?& array['a', 'b'] |