

分 区 表

分区表是关系型数据库提供的一个亮点特性，比如 Oracle 对分区表的支持已经非常成熟，广泛使用于生产系统，PostgreSQL 也支持分区表，只是道路有些曲折，早在 10 版本之前 PostgreSQL 分区表一般通过继承加触发器方式实现，这种分区方式不能算是内置分区表，而且步骤非常烦琐，PostgreSQL 10 版本一个重量级的新特性是支持内置分区表，在分区表方面前进了一大步，目前支持范围分区和列表分区。为了便于说明，继承加触发器方式实现的分区表称为传统分区表，10 版本提供的分区表称为内置分区表，本节将介绍这两种分区表的创建、性能测试和注意点。

8.1 分区表的意义

分区表主要有以下优势：

- 当查询或更新一个分区上的大部分数据时，对分区进行索引扫描代价很大，然而，在分区上使用顺序扫描能提升性能。
- 当需要删除一个分区数据时，通过 DROP TABLE 删除一个分区，远比 DELETE 删除数据高效，特别适用于日志数据场景。
- 由于一个表只能存储在一个表空间上，使用分区表后，可以将分区放到不同的表空间上，例如可以将系统很少访问的分区放到廉价的存储设备上，也可以将系统常访问的分区存储在高速存储上。

分区表的优势主要体现在降低大表管理成本和某些场景的性能提升，相比普通表性能有何差异？本章将对传统分区表、内置分区表做性能测试。



8.2 传统分区表

传统分区表是通过继承和触发器方式实现的，其实现过程步骤多，非常复杂，需要定义父表、定义子表、定义子表约束、创建子表索引、创建分区插入、删除、修改函数和触发器等，可以说是在普通表基础上手工实现的分区表。在介绍传统分区表之前先介绍继承，继承是传统分区表的重要组成部分。

8.2.1 继承表

PostgreSQL 提供继承表，简单地说是首先定义一张父表，之后可以创建子表并继承父表，下面通过一个简单的例子来理解。

创建一张日志模型表 `tbl_log`，如下所示：

```
mydb=> CREATE TABLE tbl_log(id int4,create_date date,log_type text);
CREATE TABLE
```

之后创建一张子表 `tbl_log_sql` 用于存储 SQL 日志，如下所示：

```
mydb=> CREATE TABLE tbl_log_sql(sql text) INHERITS(tbl_log);
CREATE TABLE
```

通过 `INHERITS(tbl_log)` 表示表 `tbl_log_sql` 继承表 `tbl_log`，子表可以定义额外的字段，以上定义了 `sql` 为额外字段，其他字段则继承父表 `tbl_log`，查看 `tbl_log_sql` 表结构，如下所示：

```
mydb=> \d tbl_log_sql
          Table "pguser.tbl_log_sql"
      Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----+
      id   | integer |           |          |
 create_date | date   |           |          |
 log_type   | text    |           |          |
      sql   | text    |           |          |
Inherits: tbl_log
```

从以上看出 `tbl_log_sql` 表有四个字段，前三个字段和父表 `tbl_log` 一样，第四个字段 `sql` 为自定义字段，以上 `Inherits: tbl_log` 信息表示继承了表 `tbl_log`。

父表和子表都可以插入数据，接着分别在父表和子表中插入一条数据，如下所示：

```
mydb=> INSERT INTO tbl_log VALUES (1,'2017-08-26',null);
INSERT 0 1
mydb=> INSERT INTO tbl_log_sql VALUES(2,'2017-08-27',null,'select 2');
INSERT 0 1
```

这时如果查询父表 `tbl_log` 会显示两表的记录，如下所示：

```
mydb=> SELECT * FROM tbl_log;
      id | create_date | log_type
-----+-----+-----+
```



```
-----+-----+-----+
 1 | 2017-08-26 |
 2 | 2017-08-27 |
(2 rows)
```

尽管查询父表会将子表的记录数也列出，但子表自定义的字段没有显示，如果想确定数据来源于哪张表，可通过以下 SQL 查看表的 OID，如下所示：

```
mydb=> SELECT tableoid, * FROM tbl_log;
tableoid | id | create_date | log_type
-----+-----+-----+-----+
 16854 | 1 | 2017-08-26 |
 16860 | 2 | 2017-08-27 |
(2 rows)
```

tableoid 是表的隐藏字段，表示表的 OID，可通过 pg_class 系统表关联找到表名，如下所示：

```
mydb=> SELECT p.relname,c.*
      FROM tbl_log c, pg_class p
      WHERE c.tableoid = p.oid;
relname | id | create_date | log_type
-----+-----+-----+-----+
tbl_log | 1 | 2017-08-26 |
tbl_log_sql | 2 | 2017-08-27 |
(2 rows)
```

如果只想查询父表的数据，需在父表名称前加上关键字 ONLY，如下所示：

```
mydb=> SELECT * FROM ONLY tbl_log;
 id | create_date | log_type
-----+-----+-----+
 1 | 2017-08-26 |
(1 row)
```

因此，对于 UPDATE、DELETE、SELECT 操作，如果父表名称前没有加 ONLY，则会对父表和所有子表进行 DML 操作，如下所示：

```
mydb=> DELETE FROM tbl_log;
DELETE 2
mydb=> SELECT count(*) FROM tbl_log;
 count
-----
 0
(1 row)
```

从以上结果可以看出父表和所有子表数据都被删除了。

 **注意** 对于使用了继承表的场景，对父表的 UPDATE、DELETE 的操作需谨慎，因为会对父表和所有子表的数据进行 DML 操作。



8.2.2 创建分区表

接下来介绍传统分区表的创建，传统分区表创建过程主要包括以下几个步骤。

- 步骤 1** 创建父表，如果父表上定义了约束，子表会继承，因此除非是全局约束，否则不应该在父表上定义约束，另外，父表不应该写入数据。
- 步骤 2** 通过 INHERITS 方式创建继承表，也称之为子表或分区，子表的字段定义应该和父表保持一致。
- 步骤 3** 给所有子表创建约束，只有满足约束条件的数据才能写入对应分区，注意分区约束值范围不要有重叠。
- 步骤 4** 给所有子表创建索引，由于继承操作不会继承父表上的索引，因此索引需要手工创建。
- 步骤 5** 在父表上定义 INSERT、DELETE、UPDATE 触发器，将 SQL 分发到对应分区，这步可选，因为应用可以根据分区规则定位到对应分区进行 DML 操作。
- 步骤 6** 启用 constraint_exclusion 参数，如果这个参数设置成 off，则父表上的 SQL 性能会降低，后面会通过示例解释这个参数。

以上六个步骤是创建传统分区表的主要步骤，接下来通过一个示例演示创建一张范围分区表，并且定义年月子表存储月数据。

首先创建父表，如下所示：

```
CREATE TABLE log_ins(id serial,
user_id int4,
create_time timestamp(0) without time zone);
```

创建 13 张子表，如下所示：

```
CREATE TABLE log_ins_history(CHECK ( create_time < '2017-01-01' )) INHERITS(log_ins);
CREATE TABLE log_ins_201701(CHECK ( create_time >= '2017-01-01' and create_time
< '2017-02-01')) INHERITS(log_ins);
CREATE TABLE log_ins_201702(CHECK ( create_time >= '2017-02-01' and create_time
< '2017-03-01')) INHERITS(log_ins);
...
CREATE TABLE log_ins_201712(CHECK ( create_time >= '2017-12-01' and create_time
< '2018-01-01')) INHERITS(log_ins);
```

中间省略了部分脚本，给子表创建索引，如下所示：

```
CREATE INDEX idx_his_ctime ON log_ins_history USING btree (create_time);
CREATE INDEX idx_log_ins_201701_ctime ON log_ins_201701 USING btree (create_time);
CREATE INDEX idx_log_ins_201702_ctime ON log_ins_201702 USING btree (create_time);
...
CREATE INDEX idx_log_ins_201712_ctime ON log_ins_201712 USING btree (create_time);
```

由于父表上不存储数据，可以不用在父表上创建索引。

创建触发器函数，设置数据插入父表时的路由规则，如下所示：

```

CREATE OR REPLACE FUNCTION log_ins_insert_trigger()
RETURNS trigger
LANGUAGE plpgsql
AS $function$
BEGIN
    IF      ( NEW.create_time < '2017-01-01' ) THEN
        INSERT INTO log_ins_history VALUES (NEW.*);
    ELSIF ( NEW.create_time>='2017-01-01' and NEW.create_time<'2017-02-01' ) THEN
        INSERT INTO log_ins_201701 VALUES (NEW.*);
    ELSIF ( NEW.create_time>='2017-02-01' and NEW.create_time<'2017-03-01' ) THEN
        INSERT INTO log_ins_201702 VALUES (NEW.*);
    ...
    ELSIF ( NEW.create_time>='2017-12-01' and NEW.create_time<'2018-01-01' ) THEN
        INSERT INTO log_ins_201712 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'create_time out of range. Fix the log_ins_insert_
                           trigger() function!';
    END IF;
    RETURN NULL;
END;
$function$;

```

函数中的 new.* 是指要插入的数据行，在父表上定义插入触发器，如下所示：

```

CREATE TRIGGER insert_log_ins_trigger BEFORE INSERT ON log_ins FOR EACH ROW
EXECUTE PROCEDURE log_ins_insert_trigger();

```

触发器创建完成后，往父表 log_ins 插入数据时，会执行触发器并触发函数 log_ins_insert_trigger() 将表数据插入到相应分区中。DELETE、UPDATE 触发器和函数创建过程和 INSERT 方式类似，这里不再列出，这步完成之后，传统分区表的创建步骤已全部完成。



注意 父表和子表都可以定义主键约束，但会带来一个问题，由于父表和子表的主键约束是分别创建的，那么可能在父表和子表中存在重复的主键数据，这对整个分区表说来做不到主键唯一，举个简单的例子，假如在父表和所有子表的 user_id 字段上创建主键，父表与子表及子表与子表之间可能存在相同的 user_id，这点需要注意。

8.2.3 使用分区表

往父表 log_ins 插入测试数据，并验证数据是否插入对应分区，如下所示：

```

INSERT INTO log_ins(user_id,create_time)
SELECT round(100000000*random()),generate_series('2016-12-01'::date,
                                              '2017-12-01'::date, '1 minute');

```

这里通过 round(100000000*random()) 随机生成 8 位整数，generate_series 函数生成时间数据，数据如下所示：

```
mydb=> SELECT * FROM log_ins LIMIT 2;
```

```

      id | user_id |      create_time
-----+-----+-----
 570242 | 24040985 | 2016-12-01 00:00:00
 570243 | 10814368 | 2016-12-01 00:01:00
(2 rows)

```

查看父表数据，发现父表里没有数据，如下所示。

```
mydb=> SELECT count(*) FROM ONLY log_ins;
      count
-----
```

```
      0
(1 row)
```

```
mydb=> SELECT count(*) FROM log_ins;
      count
-----
```

```
      525601
(1 row)
```

查看子表数据，如下所示：

```
mydb=> SELECT min(create_time),max(create_time) FROM log_ins_201701;
      min           |      max
-----+-----
```

```
2017-01-01 00:00:00 | 2017-01-31 23:59:00
(1 row)
```

这说明子表里可查到数据，查看子表大小，如下所示：

```
mydb=> \dt+ log_ins*
          List of relations
 Schema |      Name      | Type | Owner | Size | Description
-----+-----+-----+-----+-----+-----+
 pguser | log_ins     | table | pguser | 0 bytes | 
 pguser | log_ins_201701 | table | pguser | 1960 kB | 
 pguser | log_ins_201702 | table | pguser | 1768 kB | 
 ...
 pguser | log_ins_201712 | table | pguser | 8192 bytes |
 pguser | log_ins_history | table | pguser | 1960 kB |
(14 rows)
```

由此可见数据都已经插入到子表里。

8.2.4 查询父表还是子表

假如我们检索 2017-01-01 这一天的数据，我们可以查询父表，也可以直接查询子表，两者性能上是否有差异呢？查询父表的执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM log_ins WHERE create_time > '2017-01-01' AND
          create_time < '2017-01-02';
                                     QUERY PLAN
```

```

-----  

Append (cost=0.00..45.97 rows=1435 width=16) (actual time=0.025..0.425 rows=1439  

    loops=1)  

    -> Seq Scan on log_ins (cost=0.00..0.00 rows=1 width=16) (actual time=0.004..  

        0.004 rows=0 loops=1)  

        Filter: ((create_time > '2017-01-01 00:00:00'::timestamp without time  

            zone) AND (create_time < '2017-01-02 00:00:00'::times  

            tamp without time zone))  

        -> Index Scan USING idx_log_ins_201701_ctime on log_ins_201701 (cost=0.29..  

            45.97 rows=1434 width=16) (actual time=0.020..0.285  

rows=1439 loops=1)  

        Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp without  

            time zone) AND (create_time < '2017-01-02 00:00:00'::t  

            imestamp without time zone))  

        Planning time: 0.581 ms  

        Execution time: 0.515 ms  

(7 rows)

```

从以上执行计划看出在分区 log_ins_201701 上进行了索引扫描，执行时间为 0.515 毫秒，接着查看直接查询子表 log_ins_201701 的执行计划，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT * FROM log_ins_201701 WHERE create_time > '2017-01-  

01' AND create_time < '2017-01-02';  

          QUERY PLAN  

-----  

Index Scan USING idx_log_ins_201701_ctime on log_ins_201701 (cost=0.29..45.97  

    rows=1434 width=16) (actual time=0.017..0.254 rows=1  

439 loops=1)  

    Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp without time  

        zone) AND (create_time < '2017-01-02 00:00:00'::timestamp  

        without time zone))  

    Planning time: 0.142 ms  

    Execution time: 0.337 ms  

(4 rows)

```

从以上执行计划看出，直接查询子表只需要 0.337 毫秒，性能上有一定提升，如果并发量上去的话，这个差异将更明显，因此在实际生产过程中，对于传统分区表分区方式，不建议应用访问父表，而是直接访问子表，也许有人会问，应用如何定位到访问哪张子表呢？可以根据预先的分区约束定义，本节这个例子 log_ins 是根据时间范围分区，那么应用可以根据时间来判断查询哪张子表，当然，以上是根据分区表分区键查询的场景，如果根据非分区键查询则会扫描分区表的所有分区。

8.2.5 constraint_exclusion 参数

constraint_exclusion 参数用来控制优化器是否根据表上的约束来优化查询，参数值为以下值：

- on：所有表都通过约束优化查询；

- off: 所有表都不通过约束优化查询;
- partition: 只对继承表和 UNION ALL 子查询通过检索约束来优化查询;

简单地说, 如果设置成 on 或 partition, 查询父表时优化器会根据子表上的约束判断检索哪些子表, 而不需要扫描所有子表, 从而提升查询性能, 接下来在会话级别将参数 constraint_exclusion 设置成 off, 进行测试, 如下所示:

```
mydb=> SET constraint_exclusion =off;
SET
```

接下来查询父表, 如下所示:

```
mydb=> EXPLAIN ANALYZE SELECT * FROM log_ins WHERE create_time > '2017-01-01' AND
create_time < '2017-01-02';
                                         QUERY PLAN
-----
Append  (cost=0.00..94.40 rows=1447 width=16) (actual time=0.029..0.534 rows=1439
loops=1)
    -> Seq Scan on log_ins  (cost=0.00..0.00 rows=1 width=16) (actual
time=0.005..0.005 rows=0 loops=1)
        Filter: ((create_time > '2017-01-01 00:00:00'::timestamp without time
zone) AND (create_time < '2017-01-02 00:00:00'::timestamp without time
zone))
    -> Index Scan USING idx_his_ctime on log_ins_history  (cost=0.29..4.31
rows=1 width=16) (actual time=0.008..0.008 rows=0 loops=1)
        Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp
without time zone) AND (create_time < '2017-01-02 00:00:00'::
timestamp without time zone))
    -> Index Scan USING idx_log_ins_201701_ctime on log_ins_201701
(cost=0.29..45.97 rows=1434 width=16) (actual time=0.016..0.293
rows=1439 loops=1)
        Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp
without time zone) AND (create_time < '2017-01-02 00:00:00'::
timestamp without time zone))=0 loops=1
    ...
    -> Seq Scan on log_ins_201712  (cost=0.00..1.01 rows=1 width=16) (actual
time=0.011..0.011 rows=0 loops=1)
        Filter: ((create_time > '2017-01-01 00:00:00'::timestamp without
time zone) AND (create_time < '2017-01-02 00:00:00'::
timestamp without time zone))
Rows Removed by Filter: 1
Planning time: 1.344 ms
Execution time: 0.685 ms
(32 rows)
```

从以上执行计划看出, 查询父表时扫描了所有分区, 执行时间上升到了 0.685 毫秒, 性能下降不少, 假如一张分区表有成百上千个分区, 扫描所有分区带来的性能下降将会非常大, 因此, 这个参数建议设置成 partition, 不建议设置成 on, 因为优化器通过检查约束来

优化查询的方式本身就带来一定开销，如果所有表都启用这个特性，将加重优化器的负担。

8.2.6 添加分区

添加分区属于分区表维护的常规操作之一，比如历史表范围分区到期之前需要扩分区，`log_ins` 表为日志表，每个分区存储当月数据，假如分区快到期了，可通过以下 SQL 扩分区，首先创建子表，如下所示：

```
CREATE TABLE log_ins_201801(CHECK ( create_time >= '2018-01-01' and create_time
< '2018-02-01')) INHERITS(log_ins);
...
```

通常会多定义一些分区，这个操作要根据具体场景来进行。

之后创建相关索引，如下所示：

```
CREATE INDEX idx_log_ins_201801_ctime ON log_ins_201801 USING btree
(create_time);
...
```

然后刷新触发器函数 `log_ins_insert_trigger()`，添加相应代码，将符合路由规则的数据插入新分区，详见之前定义的这个函数，这步完成后，添加分区操作完成，可通过 `\d+ log_ins` 命令查看 `log_ins` 的所有分区。

这种方法比较直接，创建分区时就将分区继承到父表，如果中间步骤有错可能对生产系统带来影响，比较推荐的做法是将以上操作分解成以下几个步骤，降低对生产系统的影响，如下所示：

```
--创建分区
CREATE TABLE log_ins_201802(LIKE log_ins INCLUDING ALL );

--添加约束
ALTER TABLE log_ins_201802 ADD CONSTRAINT log_ins_201802_create_time_check
CHECK ( create_time >= '2018-02-01' AND create_time < '2018-03-01');

--刷新触发器函数log_ins_insert_trigger()
函数刷新前建议先备份函数代码。

--所有步骤完成后，将新分区log_ins_201802继承到父表log_ins
ALTER TABLE log_ins_201802 INHERIT log_ins;
```

以上方法是将新分区所有操作完成后，再将分区继承到父表，降低了生产系统添加分区操作的风险，当然，在生产系统添加分区前建议在测试环境事先演练一把。

8.2.7 删除分区

分区表的一个重要优势是对于大表的管理上十分方便，例如需要删除历史数据时可以直接删除一个分区，这比 `DELETE` 方式效率高了多个数量级，传统分区表删除分区通常有两种方法，第一种方法是直接删除分区，如下所示：

```
DROP TABLE log_ins_201802
```

就像删除普通表一样删除分区即可，当然删除分区前需再三确认是否需要备份数据；另一种比较推荐的删除分区方法是先将分区的继承关系去掉，如下所示：

```
mydb=> ALTER TABLE log_ins_201802 NO INHERIT log_ins;
ALTER TABLE
```

执行以上命令后，`log_ins_201802` 分区不再属于分区表 `log_ins` 的分区，但 `log_ins_201802` 表依然保留可供查询，这种方式相比方法一提供了一个缓冲时间，属于比较稳妥的删除分区方法，因为在拿掉子表继承关系后，只要没删除这个子表，还可以使子表重新继承父表。

8.2.8 分区表相关查询

分区表创建完成后，如何查看分区表定义、分区表分区信息呢？比较常用的方法是通过 `\d` 元命令，如下所示：

```
mydb=> \d log_ins
           Table "pguser.log_ins"
   Column | Type      | Collation | Nullable | Default
-----+-----+-----+-----+
    id   | integer   | not null  | nextval('log_ins_id_seq'::regclass)
 user_id | integer   |            |          |
create_time | timestamp(0) without time zone |          |
Triggers:
    insert_log_ins_trigger BEFORE INSERT ON log_ins FOR EACH ROW EXECUTE
PROCEDURE log_ins_insert_trigger()
Number of child tables: 14 (Use \d+ to list them.)
```

以上信息显示了表 `log_ins` 有 14 个分区，并且创建了触发器，触发器函数为 `log_ins_insert_trigger()`，如果想列出分区名称可通过 `\d+ log_ins` 元命令列出。

另一种列出分区表分区信息方法是通过 SQL 命令，如下所示：

```
mydb=> SELECT
  nmsp_parent.nspname AS parent_schema ,
  parent.relname AS parent ,
  nmsp_child.nspname AS child_schema ,
  child.relname AS child_schema
FROM
  pg_inherits JOIN pg_class parent
  ON pg_inherits.inhparent = parent.oid JOIN pg_class child
  ON pg_inherits.inhrelid = child.oid JOIN pg_namespace nmsp_parent
  ON nmsp_parent.oid = parent.relnamespace JOIN pg_namespace nmsp_child
  ON nmsp_child.oid = child.relnamespace
WHERE
  parent.relname = 'log_ins';
parent_schema | parent | child_schema | child_schema
-----+-----+-----+
```

```

pguser      | log_ins | pguser      | log_ins_history
pguser      | log_ins | pguser      | log_ins_201701
pguser      | log_ins | pguser      | log_ins_201702
...
pguser      | log_ins | pguser      | log_ins_201801
(14 rows)

```

`pg_inherits` 系统表记录了子表和父表之间的继承关系，通过以上查询列出指定分区表的分区。如果想查看一个库中有哪些分区表，并显示这些分区表的分区数量，可通过以下 SQL 查询：

```

mydb=> SELECT
    nspname ,
    relname ,
    count(*) AS partition_num
  FROM
    pg_class c ,
    pg_namespace n ,
    pg_inherits i
 WHERE
    c.oid = i.inhparent
  AND c.relnamespace = n.oid
  AND c.relhassubclass
  AND c.relkind in ('r','p')
 GROUP BY 1,2 ORDER BY partition_num DESC;
   nspname | relname | partition_num
-----+-----+
  pguser  | log_ins |          14
  pguser  | tbl_log |           1
(2 rows)

```

以上结果显示当前库中有两个分区表，`log_ins` 分区表有 14 个分区，`tbl_log` 分区表只有一个分区。

8.2.9 性能测试

基于分区表的分区键、非分区键查询和普通表性能有何差异呢？本节继续进行测试，将 `create_time` 字段作为传统分区表 `log_ins` 的分区键，`user_id` 字段作为分区表的非分区键。

首先创建一张普通表 `log`，表结构和 `log_ins` 完全一致，并插入测试数据，如下所示：

```

CREATE TABLE log(id
serial,user_id int4,
create_time timestamp(0) without time zone);

INSERT INTO log(user_id,create_time)
SELECT round(100000000*random()),generate_series('2016-12-01'::date,
'2017-12-01'::date, '1 minute');

```

查看两表记录数，如下所示：



```
mydb=> SELECT count(*) FROM log_ins;
+-----+
| count |
+-----+
| 525601 |
+-----+
(1 row)

mydb=> SELECT count(*) FROM log;
+-----+
| count |
+-----+
| 525601 |
+-----+
(1 row)
```

两表数据量是一样的，普通表 log 创建索引，如下所示：

```
CREATE INDEX idx_log_userid ON log USING btree(user_id);
CREATE INDEX idx_log_create_time ON log USING btree(create_time);
```

在分区表 log_ins 父表和所有子表的 user_id 上创建索引，如下所示：

```
CREATE INDEX idx_log_ins_userid ON log_ins USING btree(user_id);
CREATE INDEX idx_his_userid ON log_ins_history USING btree (user_id);
CREATE INDEX idx_log_ins_201701_userid ON log_ins_201701 USING btree (user_id);
CREATE INDEX idx_log_ins_201702_userid ON log_ins_201702 USING btree (user_id);
...
CREATE INDEX idx_log_ins_201801_userid ON log_ins_201801 USING btree (user_id);
```

接下来根据 user_id 进行检索，对于分区表 log_ins 来说，这是非分区键，在根据 user_id 检索的场景下，普通表和分区表性能差异如何呢？设置场景一测试 SQL，如下所示：

```
--场景一：根据user_id检索
SELECT * FROM log WHERE user_id=?;
SELECT * FROM log_ins WHERE user_id=?;
```

首先查找一个在表 log 和 log_ins 都存在的 user_id，如下所示：

```
mydb=> SELECT a.* FROM log a ,log_ins b WHERE a.user_id=b.user_id LIMIT 1;
+-----+-----+-----+
| id | user_id | create_time |
+-----+-----+-----+
| 67286 | 51751630 | 2017-01-16 17:25:00 |
+-----+-----+-----+
(1 row)
```

根据 user_id=51751630 进行检索，普通表 log 上的执行计划如下所示：

```
mydb=> EXPLAIN SELECT * FROM log WHERE user_id=51751630;
+-----+
| QUERY PLAN |
+-----+
| Index Scan using idx_log_userid on log  (cost=0.42..4.44 rows=1 width=16)
|   Index Cond: (user_id = 51751630) |
+-----+
(2 rows)
```

以上查询进行了索引扫描，根据非分区键 user_id 进行检索，分区表执行计划则完全不一样，如下所示：



```

mydb=> EXPLAIN SELECT * FROM log_ins WHERE user_id=51751630;
          QUERY PLAN
-----
Append  (cost=0.00..63.18 rows=23 width=16)
-> Seq Scan on log_ins  (cost=0.00..0.00 rows=1 width=16)
  Filter: (user_id = 88258037)
-> Index Scan USING idx_his_userid on log_ins_history
  (cost=0.29..4.31 rows=1 width=16)
  Index Cond: (user_id = 88258037)
-> Index Scan USING idx_log_ins_201701_userid on log_ins_201701
  (cost=0.29.. 4.31 rows=1 width=16)
  Index Cond: (user_id = 88258037)
-> Index Scan USING idx_log_ins_201702_userid on log_ins_201702  (cost=0.29..
  4.31 rows=1 width=16)
  Index Cond: (user_id = 88258037)
...
-> Bitmap Heap Scan on log_ins_201801  (cost=2.22..10.48 rows=9 width=16)
  Recheck Cond: (user_id = 88258037)
    -> Bitmap Index Scan on idx_log_ins_201801_userid  (cost=0.00..2.22
      rows=9 width=0)
      Index Cond: (user_id = 88258037)
(33 rows)

```

从以上执行计划看出，根据非分区键查询则扫描了分区表所有分区，对 log 表执行场景一 SQL，执行三次，最小执行时间为 0.050 毫秒；对 log_ins 表执行场景一 SQL，执行三次，最小执行时间为 0.184 毫秒；

create_time 字段是分区表 log_ins 分区键，设置场景二测试 SQL，如下所示：

```
--场景二：根据create_time检索；
SELECT * FROM log WHERE create_time > '2017-01-01' AND create_time < '2017-01-02';
SELECT * FROM log_ins WHERE create_time > '2017-01-01' AND create_time < '2017-01-02';
```

对 log 表执行场景二 SQL，执行三次，最小执行时间为 0.339 毫秒；对 log_ins 执行场景二 SQL，执行三次，取最小执行时间为 0.503 毫秒。表 8-1 为场景一、场景二测试结果数据汇总。

表 8-1 普通表、传统分区表性能对比

查询场景	普通表 log 执行时间	分区表：查询 log_ins 父表执行时间	分区表：查询 log_ins 子表执行时间
根据非分区键 user_id 查询	0.05 毫秒	0.184 毫秒	不支持
根据分区键 create_time 范围查询	0.339 毫秒	0.503 毫秒	0.325 毫秒

从以上测试结果来看，在根据 user_id 检索的场景下，分区表的性能比普通表性能差了 2.68 倍；在根据 create_time 范围检索的场景下，分区表的性能比普通表性能差了 0.4 倍左右，如果查询能定位到子表，则比普通表性能略有提升，从分区表的角度来看，create_time 作为分区键，user_id 作为非分区键，从这个测试可以得出以下结论：





1) 分区表根据非分区键查询相比普通表性能差距较大，因为这种场景分区表的执行计划会扫描所有分区；

2) 分区表根据分区键查询相比普通表性能有小幅降低，而查询分区表子表性能比普通表略有提升；

以上两个场景除了场景二直接检索分区表子表，性能相比普通表略有提升，其他测试项分区表比普通表性能都低，因此出于性能考虑对生产环境业务表做分区表时需慎重，使用分区表不一定能提升性能，如果业务模型 90%（估算的百分比，意思是大部分）以上的操作都能基于分区键操作，并且 SQL 可以定位到子表，这时建议使用分区表。

8.2.10 传统分区表注意事项

传统分区表的使用有以下注意事项：

- 当往父表上插入数据时，需事先在父表上创建路由函数和触发器，数据才会根据分区键路由规则插入到对应分区中，目前仅支持范围分区和列表分区。
- 分区表上的索引、约束需要使用单独的命令创建，目前没有办法一次性自动在所有分区上创建索引、约束。
- 父表和子表允许单独定义主键，因此父表和子表可能存在重复的主键记录，目前不支持在分区表上定义全局主键。
- UPDATE 时不建议更新分区键数据，特别是会使数据从一个分区移动到另一分区的场景，可通过更新触发器实现，但会带来管理上的成本。
- 性能方面：根据本节的测试数据和测试场景，传统分区表根据非分区键查询相比普通表性能差距较大，因为这种场景下分区表会扫描所有分区；根据分区键查询相比普通表性能有小幅降低，而查询分区表子表性能相比普通表略有提升；

8.3 内置分区表

PostgreSQL 10 一个重量级新特性是支持内置分区表，用户不需要预先在父表上定义 INSERT、DELETE、UPDATE 触发器，对父表的 DML 操作会自动路由到相应分区，相比传统分区表大幅度降低了维护成本，目前仅支持范围分区和列表分区，本小节将以创建范围分区表为例，演示 PostgreSQL 10 内置分区表的创建、使用与性能测试。

8.3.1 创建分区表

创建分区表的主要语法包含两部分：创建主表和创建分区。

创建主表语法如下：

```
CREATE TABLE table_name ( ... )
[ PARTITION BY { RANGE | LIST } ( { column_name | ( expression ) }
```





创建主表时须指定分区方式，可选的分区方式为 RANGE 范围分区或 LIST 列表分区，并指定字段或表达式作为分区键。

创建分区的语法如下：

```
CREATE TABLE table_name  
    PARTITION OF parent_table [ ( ) ] FOR VALUES partition_bound_spec
```

创建分区时必须指定是哪张表的分区，同时指定分区策略 partition_bound_spec，如果是范围分区，partition_bound_spec 须指定每个分区分区键的取值范围，如果是列表分区 partition_bound_spec，需指定每个分区的分区键值。

PostgreSQL10 创建内置分区表主要分为以下几个步骤：

1) 创建父表，指定分区键和分区策略。

2) 创建分区，创建分区时须指定分区表的父表和分区键的取值范围，注意分区键的范围不要有重叠，否则会报错。

3) 在分区上创建相应索引，通常情况下分区键上的索引是必须的，非分区键的索引可根据实际应用场景选择是否创建。

接下来通过创建范围分区的示例来演示内置分区表的创建过程，首先创建一张范围分区表，表名为 log_par，如下所示：

```
CREATE TABLE log_par (  
    id serial,  
    user_id int4,  
    create_time timestamp(0) without time zone  
) PARTITION BY RANGE(create_time);
```

表 log_par 指定了分区策略为范围分区，分区键为 create_time 字段。

创建分区，并设置分区的分区键取值范围，如下所示：

```
CREATE TABLE log_par_his PARTITION OF log_par FOR VALUES FROM (UNBOUNDED)  
    TO ('2017-01-01');  
CREATE TABLE log_par_201701 PARTITION OF log_par FOR VALUES FROM ('2017-01-01')  
    TO ('2017-02-01');  
CREATE TABLE log_par_201702 PARTITION OF log_par FOR VALUES FROM ('2017-02-01')  
    TO ('2017-03-01');  
...  
CREATE TABLE log_par_201712 PARTITION OF log_par FOR VALUES FROM ('2017-12-01')  
    TO ('2018-01-01');
```

注意分区的分区键范围不要有重叠，定义分区键范围实质上给分区创建了约束。

给所有分区的分区键创建索引，如下所示：

```
CREATE INDEX idx_log_par_his_ctime ON log_par_his USING btree(create_time);  
CREATE INDEX idx_log_par_201701_ctime ON log_par_201701 USING btree(create_time);  
CREATE INDEX idx_log_par_201702_ctime ON log_par_201702 USING btree(create_time);  
...  
CREATE INDEX idx_log_par_201712_ctime ON log_par_201712 USING btree(create_time);
```





以上三步完成了内置分区表的创建。

8.3.2 使用分区表

向分区表插入数据，如下所示：

```
INSERT INTO log_par(user_id,create_time)
SELECT round(100000000*random()),generate_series('2016-12-01'::date,
'2017-12-01'::date, '1 minute');
```

查看表数据，如下所示：

```
mydb=> SELECT count(*) FROM log_par;
count
```

```
-----  
525601  
(1 row)
```

```
mydb=> SELECT count(*) FROM ONLY log_par;
count
-----  
0  
(1 row)
```

从以上结果可以看出，父表 log_par 没有存储任何数据，数据存储在分区中，通过分区大小也可以证明这一点，如下所示：

```
mydb=> \dt+ log_par*
          List of relations
 Schema |      Name       | Type  | Owner |     Size    | Description
-----+-----+-----+-----+-----+-----+
 pguser | log_par      | table | pguser | 0 bytes   |
 pguser | log_par_201701 | table | pguser | 1960 kB   |
 pguser | log_par_201702 | table | pguser | 1768 kB   |
 ...
 pguser | log_par_201712 | table | pguser | 8192 bytes |
 pguser | log_par_his   | table | pguser | 1960 kB   |
```

8.3.3 内置分区表原理探索

内置分区表原理实际上和传统分区表一样，也是使用继承方式，分区可称为子表，通过以下查询很明显看出表 log_par 和其分区是继承关系：

```
mydb=> SELECT
  nmsp_parent.nspname AS parent_schema ,
  parent.relname AS parent ,
  nmsp_child.nspname AS child_schema ,
  child.relname AS child_schema
FROM
  pg_inherits JOIN pg_class parent
```





```
ON pg_inherits.inhparent = parent.oid JOIN pg_class child
ON pg_inherits.inherelid = child.oid JOIN pg_namespace nmsp_parent
ON nmsp_parent.oid = parent.relnamespace JOIN pg_namespace nmsp_child
ON nmsp_child.oid = child.relnamespace

WHERE
    parent.relname = 'log_par';
parent_schema | parent | child_schema | child_schema
-----+-----+-----+-----
pguser      | log_par | pguser      | log_par_his
pguser      | log_par | pguser      | log_par_201701
pguser      | log_par | pguser      | log_par_201702
...
pguser      | log_par | pguser      | log_par_201712
(13 rows)
```

以上 SQL 显示了分区表 log_par 的所有分区，也可以通过 \d+ log_par 元子命令显示 log_par 所有分区。

8.3.4 添加分区

添加分区的操作比较简单，例如给 log_par 增加一个分区，如下所示：

```
CREATE TABLE log_par_201801 PARTITION OF log_par FOR VALUES FROM ('2018-01-01')
TO ('2018-02-01');
```

之后给分区创建索引，如下所示：

```
CREATE INDEX idx_log_par_201801_ctime ON log_par_201801 USING btree(create_time);
```

8.3.5 删除分区

删除分区有两种方法，第一种方法通过 DROP 分区的方式来删除，如下所示：

```
DROP TABLE log_par_201801;
```

DROP 方式直接将分区和分区数据删除，删除前需确认分区数据是否需要备份，避免数据丢失；另一种推荐的方法是解绑分区，如下所示：

```
mydb=> ALTER TABLE log_par DETACH PARTITION log_par_201801;
ALTER TABLE
```

解绑分区只是将分区和父表间的关系断开，分区和分区数据依然保留，这种方式比较稳妥，如果后续需要恢复这个分区，通过连接分区方式恢复分区即可，如下所示：

```
mydb=> ALTER TABLE log_par ATTACH PARTITION log_par_201801 FOR VALUES FROM
('2018-01-01') TO ('2018-02-01');
ALTER TABLE
```

连接分区时需要指定分区上的约束。





8.3.6 性能测试

检索 2017-01-01 这一天的记录数据，执行如下 SQL：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM log_par WHERE create_time > '2017-01-01' AND
create_time < '2017-01-02';
                                         QUERY PLAN
-----
 Append  (cost=0.29..45.21 rows=1396 width=16) (actual time=0.019..0.425 rows=1439
 loops=1)
   -> Index Scan using idx_log_par_201701_ctime on log_par_201701
       (cost=0.29.. 45.21 rows=1396 width=16) (actual time=0.019..0.288
rows=1439 loops=1)
         Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp without time
zone) AND (create_time < '2017-01-02 00:00:00'::timestamp without time zone))
Planning time: 0.461 ms
Execution time: 0.510 ms
(5 rows)
```

从以上执行计划看出仅扫描了分区 log_par_201701，进行了索引扫描，执行时间为 0.510 毫秒。

同样，我们将内置分区表 log_par 的性能和 log 表进行对比，create_time 作为分区表 log_par 的分区键，user_id 作为分区表的非分区键，基于分区表的分区键、非分区键查询和普通表性能有何差异呢？本节将做进一步测试。

在分区表 log_par 所有子表的 user_id 上创建索引，如下所示：

```
CREATE INDEX idx_log_par_his_userid ON log_par_his using btree (user_id);
CREATE INDEX idx_log_par_201701_userid ON log_par_201701 using btree (user_id);
CREATE INDEX idx_log_par_201702_userid ON log_par_201702 using btree (user_id);
...
CREATE INDEX idx_log_par_201712_userid ON log_par_201712 using btree (user_id);
```

根据 user_id 进行检索，对于分区表 log_par 而言这是非分区键，设置场景一测试 SQL，如下所示：

```
--场景一：根据user_id检索
SELECT * FROM log WHERE user_id=?;
SELECT * FROM log_par WHERE user_id=?;
```

我们首先查找一个在表 log 和 log_par 都存在的 user_id，如下所示：

```
mydb=> SELECT a.* FROM log a ,log_par b WHERE a.user_id=b.user_id LIMIT 1;
          id | user_id |      create_time
-----+-----+-----+
      258926 | 70971018 | 2017-05-29 19:25:00
(1 row)
```

根据 user_id=70971018 进行检索，普通表 log 上的执行计划如下所示：





```
mydb=> EXPLAIN SELECT * FROM log WHERE user_id=70971018;
          QUERY PLAN
```

```
-----  
Index Scan using idx_log_userid on log  (cost=0.42..4.44 rows=1 width=16)  
  Index Cond: (user_id = 70971018)  
(2 rows)
```

可以看出以上查询进行了索引扫描。

根据非分区键 user_id 进行检索，分区表 log_par 上的执行计划则完全不一样，如下所示：

```
mydb=> EXPLAIN SELECT * FROM log_par WHERE user_id=70971018;
          QUERY PLAN
```

```
-----  
Append  (cost=0.29..52.70 rows=13 width=16)  
  -> Index Scan using idx_log_par_his_userid on log_par_his  (cost=0.29..4.31 rows=1  
    width=16)  
    Index Cond: (user_id = 70971018)  
  -> Index Scan using idx_log_par_201701_userid on log_par_201701  
    (cost=0.29.. 4.31 rows=1 width=16)  
    Index Cond: (user_id = 70971018)  
  -> Index Scan using idx_log_par_201702_userid on log_par_201702  
    (cost=0.29.. 4.31 rows=1 width=16)  
    Index Cond: (user_id = 70971018)  
  ...  
  -> Seq Scan on log_par_201712  (cost=0.00..1.01 rows=1 width=16)  
    Filter: (user_id = 70971018)  
(27 rows)
```

从以上执行计划看出，根据非分区键 user_id 检索分区表 log_par 扫描了整个分区，接着 log 表执行场景一 SQL，执行三次，最小执行时间为 0.047 毫秒；log_par 表执行场景一 SQL，执行三次，最小执行时间为 0.139 毫秒；

create_time 字段是分区表 log_par 分区键，设置场景二测试 SQL，如下所示：

```
--场景二：根据create_time检索；  
SELECT * FROM log WHERE create_time > '2017-01-01' AND create_time < '2017-01-02';  
SELECT * FROM log_par WHERE create_time > '2017-01-01' AND create_time < '2017-01-02';
```

对 log 表执行场景二 SQL，执行三次，最小执行时间为 0.340 毫秒；对 log_par 执行场景二 SQL，执行三次，最小执行时间为 0.503 毫秒。表 8-2 为场景一、场景二测试结果数据汇总。

表 8-2 普通表、内置分区表性能对比

查询场景	普通表 log 执行时间	分区表：查询 log_par 父表执行时间	区表：查询 log_par 子表执行时间
根据非分区键 user_id 查询	0.047 毫秒	0.139 毫秒	不支持
根据分区键 create_time 范围查询	0.340 毫秒	0.503 毫秒	0.319 毫秒

根据以上测试结果，在根据 user_id 检索的测试场景下，内置分区表的性能比普通表性能差了 1.95 倍；根据 create_time 范围检索的场景下，分区表的性能比普通表性能差了





0.47倍左右，如果查询能定位到子表，则比普通表性能略有提升，从分区表的角度来看，`create_time`作为分区键，`user_id`作为非分区键；结合之前测试的传统分区表性能，将表8-1和表8-2的数据合成一张表格，如表8-3所示。

表8-3 普通表、传统分区表、内置分区表性能对比

查询场景	普通表(log)	传统分区表(log_ins)		内置分区表(log_par)	
		查询父表	查询子表	查询父表	查询子表
根据非分区键 <code>user_id</code> 查询	表8-1: 0.05毫秒 表8-2: 0.047毫秒	0.184毫秒	不支持	0.139毫秒	不支持
根据分区键 <code>create_time</code> 范围查询	表8-1: 0.339毫秒 表8-2: 0.340毫秒	0.503毫秒	0.325毫秒	0.503毫秒	0.319毫秒

从上表看出传统分区表和内置分区表的在两个查询场景的性能表现一致，根据测试结果同样能得出以下结论：

- 内置分区表根据非分区键查询相比普通表性能差距较大，因为这种场景分区表的执行计划会扫描所有分区；
- 内置分区表根据分区键查询相比普通表性能有小幅降低，而查询分区表子表性能比普通表略有提升；

以上两个场景除了场景二直接检索分区表子表，性能相比普通表略有提升，其他测试项分区表比普通表性能都低，因此出于性能考虑对生产环境业务表做分区表时需慎重，使用分区表不一定能提升性能，但内置分区表相比传统分区表省去了创建触发器路由函数、触发器操作，减少了大量维护成本，相比传统分区表有较大管理方面的优势。

8.3.7 constraint_exclusion参数

内置分区表执行计划依然受`constraint_exclusion`参数影响，关闭此参数后，根据分区键查询时执行计划不会定位到相应分区。先在会话级关闭此参数，如下所示：

```
mydb=> SET constraint_exclusion =off;
SET
```

执行以下SQL来查看执行计划，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM log_par WHERE create_time > '2017-01-01' AND
create_time < '2017-01-02';
                                         QUERY PLAN
-----
Append  (cost=0.29..104.16  rows=1417  width=16)  (actual time=0.024..0.460
rows=1439 loops=1)
->  Index Scan using idx_log_par_his_ctime on log_par_his  (cost=0.29..4.31
rows=1 width=16)  (actual time=0.007..0.007 rows=0 loops=1)
Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp without
time zone) AND
```



```
(create_time < '2017-01-02 00:00:00'::timestamp without time zone))
-> Index Scan using idx_log_par_201701_ctime on log_par_201701
   (cost=0.29..45.21 rows=1396 width=16)
   (actual time=0.016..0.280 rows=1439 loops=1)
Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp without
   time zone) AND (create_time < '2017-01-02 00:00:00'::timestamp
   without time zone))
...
-> Bitmap Index Scan on idx_log_par_201801_ctime  (cost=0.00..2.24
   rows=9 width=0) (actual time=0.002..0.002 rows=0 loops=1)
Index Cond: ((create_time > '2017-01-01 00:00:00'::timestamp
   without time zone) AND
   (create_time < '2017-01-02 00:00:00'::timestamp without time zone))
Planning time: 0.792 ms
Execution time: 0.607 ms
(34 rows)
```

从以上执行计划看出扫描了分区表上的所有分区，执行时间上升到了 0.607 毫秒，同样，这个参数建议设置成 partition，不建议设置成 on，优化器通过检查约束来优化查询的方式本身就带来一定开销，如果所有表都启用这个特性，将加重优化器负担。

8.3.8 更新分区数据

内置分区表 UPDATE 操作目前不支持新记录跨分区的情况，也就是说只允许分区内的更新，例如以下 SQL 会报错：

```
mydb=> SELECT * FROM log_par_201701 LIMIT 1;
 id | user_id |      create_time
----+-----+-----
 44641 | 16965492 | 2017-01-01 00:00:00
(1 row)

mydb=> UPDATE log_par SET create_time='2017-02-02 01:01:01' WHERE user_id=16965492;
ERROR: new row for relation "log_par_201701" violates partition constraint
DETAIL: Failing row contains (44641, 16965492, 2017-02-02 01:01:01).
```

以上 user_id 等于 16965492 的记录位于 log_par_201701 分区，将这条记录的 create_time 更新为 '2017-02-02 01:01:01' 由于违反了当前分区的约束将报错，如果更新的数据不违反当前分区的约束则可正常更新数据，如下所示：

```
mydb=> UPDATE log_par SET create_time='2017-01-01 01:01:01' WHERE user_id=16965492;
UPDATE 1
```

目前内置分区表的这一限制对于日志表影响不大，对于业务表有一定影响，使用时需注意。

8.3.9 内置分区表注意事项

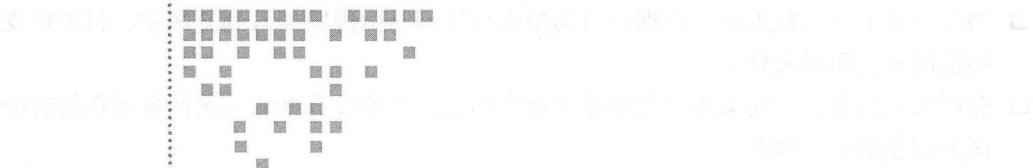
本节简单介绍了内置分区表的部署、使用示例，使用内置分区表有以下注意事项：



- 当往父表上插入数据时，数据会自动根据分区键路由规则插入到分区中，目前仅支持范围分区和列表分区。
- 分区表上的索引、约束需使用单独的命令创建，目前没有办法一次性自动在所有分区上创建索引、约束。
- 内置分区表不支持定义（全局）主键，在分区表的分区上创建主键是可以的。
- 内置分区表的内部实现使用了继承。
- 如果 UPDATE 语句的新记录违反当前分区键的约束则会报错，UPDAET 语句的新记录目前不支持跨分区的情况。
- 性能方面：根据本节的测试场景，内置分区表根据非分区键查询相比普通表性能差距较大，因为这种场景分区表的执行计划会扫描所有分区；根据分区键查询相比普通表性能有小幅降低，而查询分区表子表性能相比普通表略有提升。

8.4 本章小结

本章介绍了传统分区表和内置分区表的部署、分区维护和性能测试，传统分区表通过继承和触发器实现，创建过程非常复杂，维护成本很高，内置分区表是 PostgreSQL 10 新特性，用户不再需要创建触发器和函数，省去了大量维护成本，性能方面两者几乎无差异。分区表和普通表间的性能差异本章通过两个查询场景进行了性能对比，一个是基于非分区键查询的场景，另一个是基于分区键查询的场景，从测试结果来看，基于非分区键的查询场景分区表性能比普通表低很多，基于分区键查询分区表父表比普通表性能略低，基于分区键查询分区表子表比普通表性能略有提升，读者在生产系统中使用分区表时需考虑分区后的 SQL 性能变化。



PostgreSQL 的 NoSQL 特性

PostgreSQL 不只是一个关系型数据库，同时支持非关系特性，而且逐步增强对非关系特性的支持，第 3 章数据类型章节中介绍了 PostgreSQL 的 json 和 jsonb 数据类型，本章将进一步介绍 json、jsonb 特性，内容包括：为 jsonb 类型创建索引、json 和 jsonb 读写性能测试、全文检索对 json 和 jsonb 数据类型的支持。

9.1 为 jsonb 类型创建索引

这一节介绍为 jsonb 数据类型创建索引，jsonb 数据类型支持 GIN 索引，为了便于说明，假如一个 json 字段内容如下所示，并且以 jsonb 格式存储：

```
{
  "id": 1,
  "user_id": 1440933,
  "user_name": "l_francs",
  "create_time": "2017-08-03 16:22:05.528432+08"
}
```

假如存储以上 jsonb 数据的字段名为 user_info，表名为 tbl_user_jsonb，在 user_info 字段上创建 GIN 索引的语法如下所示：

```
CREATE INDEX idx_gin ON tbl_user_jsonb USING gin(user_info);
```

jsonb 上的 GIN 索引支持 “@>” “?” “?&” “?!” 操作符，例如以下查询将会使用索引：

```
SELECT * FROM tbl_user_jsonb WHERE user_info @> '{"user_name": "l_francs"}'
```

但是以下基于 jsonb 键值的查询不会走索引 idx_gin：

```
SELECT * FROM tbl_user_jsonb WHERE user_info->>'user_name' = 'l_francs';
```

如果要想提升基于 jsonb 类型的键值检索效率，可以在 jsonb 数据类型对应的键值上创建索引，如下所示：

```
CREATE INDEX idx_gin_user_infob_user_name ON tbl_user_jsonb USING btree
((user_info ->> 'user_name'));
```

创建以上索引后，上述根据 user_info->>'user_name' 键值查询的 SQL 将会走索引。

9.2 json、jsonb 读写性能测试

上一节介绍了 jsonb 数据类型索引创建的相关内容，本节将对 json、jsonb 读写性能进行简单对比，在第 3 章数据类型章节中介绍 json、jsonb 数据类型时提到了两者读写性能的差异，主要表现为 json 写入时比 jsonb 快，但检索时比 jsonb 慢，主要原因为：json 存储格式为文本而 jsonb 存储格式为二进制，存储格式的不同使得两种 json 数据类型的处理效率不一样。json 类型存储的内容和输入数据一样，当检索 json 数据时必须重新解析；而 jsonb 以二进制形式存储已解析好的数据，当检索 jsonb 数据时不需要重新解析。

9.2.1 创建 json、jsonb 测试表

下面通过一个简单的例子测试 json、jsonb 的读写性能差异，计划创建以下三张表：

- user_ini：基础数据表，并插入 200 万测试数据；
- tbl_user_json：json 数据类型表，200 万数据；
- tbl_user_jsonb：jsonb 数据类型表，200 万数据；

首先创建 user_ini 表并插入 200 万测试数据，如下所示：

```
mydb=> 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

mydb=> 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
```

计划使用 user_ini 表数据生成 json、jsonb 数据，创建 user_ini_json、user_ini_jsonb 表，如下所示：

```
mydb=> CREATE TABLE tbl_user_json(id serial, user_info json);
CREATE TABLE
mydb=> CREATE TABLE tbl_user_jsonb(id serial, user_info jsonb);
CREATE TABLE
```

9.2.2 json、jsonb 表写性能测试

根据 user_ini 数据通过 row_to_json 函数向表 user_ini_json 插入 200 万 json 数据，如下所示：

```
mydb=> \timing
Timing is on.

mydb=> INSERT INTO tbl_user_json(user_info) SELECT row_to_json(user_ini)
FROM user_ini;
INSERT 0 2000000
Time: 13825.974 ms (00:13.826)
```

从以上结果可以看出 tbl_user_json 插入 200 万数据花了 13 秒左右；接着根据 user_ini 表数据生成 200 万 jsonb 数据并插入表 tbl_user_jsonb，如下所示：

```
mydb=> INSERT INTO tbl_user_jsonb(user_info)
      SELECT row_to_json(user_ini)::jsonb FROM user_ini;
INSERT 0 2000000
Time: 20756.993 ms (00:20.757)
```

从以上结果可以看出 tbl_user_jsonb 表插入 200 万 jsonb 数据花了 20 秒左右，正好验证了 json 数据写入比 jsonb 快。比较两表占用空间大小，如下所示：

```
mydb=> \dt+ tbl_user_json
          List of relations
 Schema |      Name       | Type | Owner | Size | Description
-----+-----+-----+-----+-----+
 pguser | tbl_user_json | table | pguser | 281 MB |
(1 row)

mydb=> \dt+ tbl_user_jsonb
          List of relations
 Schema |      Name       | Type | Owner | Size | Description
-----+-----+-----+-----+-----+
 pguser | tbl_user_jsonb | table | pguser | 333 MB |
(1 row)
```

从占用空间来看，同样的数据量 jsonb 数据类型占用空间比 json 稍大。

查询 tbl-user-json 表的一条测试数据，如下所示：

```
mydb=> SELECT * FROM tbl_user_json LIMIT 1;
          id           |          user_info
-----+-----+
 2000001 | {"id":1,"user_id":1182883,"user_name":"l_francs","create_time":
 "2017-08-03T20:59:27.42741+08:00"}
(1 row)
```

9.2.3 json、jsonb 表读性能测试

对于 json、jsonb 读性能测试我们选择基于 json、jsonb 键值查询的场景，例如，根据

user_info 字段的 user_name 键的值查询，如下所示：

```
mydb=> 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..72859.90 rows=10042 width=143) (actual
      time=0.023..524.843 rows=1 loops=1)
       Filter: ((user_info ->> 'user_name'::text) = '1_francs'::text)
       Rows Removed by Filter: 1999999
 Planning time: 0.091 ms
 Execution time: 524.876 ms
(5 rows)
```

上述 SQL 执行时间为 524 毫秒左右。基于 user_info 字段的 user_name 键值创建 btree 索引，如下所示：

```
mydb=> CREATE INDEX idx_jsonb ON tbl_user_jsonb USING btree
      ((user_info->>'user_name'));
```

再次执行上述查询，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM tbl_user_jsonb WHERE user_info->>'user_
      name'='1_francs';
                                         QUERY PLAN
-----
 Bitmap Heap Scan on tbl_user_jsonb  (cost=155.93..14113.93 rows=10000 width=143)
      (actual time=0.027..0.027 rows=1 loops=1)
       Recheck Cond: ((user_info ->> 'user_name'::text) = '1_francs'::text)
       Heap Blocks: exact=1
      -> Bitmap Index Scan on idx_jsonb   (cost=0.00..153.43 rows=10000
          width=0) (actual time=0.021..0.021 rows=1 loops=1)
           Index Cond: ((user_info ->> 'user_name'::text) = '1_francs'::text)
 Planning time: 0.091 ms
 Execution time: 0.060 ms
(7 rows)
```

根据上述执行计划可以看出走了索引，并且 SQL 时间下降到 0.060ms。为了更好地对比 `tbl_user_json`、`tbl_user_jsonb` 表基于键值查询的效率，我们根据 user_info 字段 id 键进行范围扫描以对比性能，首先创建索引，如下所示：

```
mydb=> CREATE INDEX idx_gin_user_info_id ON tbl_user_json USING btree
      (((user_info ->> 'id')::integer));
CREATE INDEX

mydb=> CREATE INDEX idx_gin_user_infob_id ON tbl_user_jsonb USING btree
      (((user_info ->> 'id')::integer));
CREATE INDEX
```

索引创建后，查询 `tbl_user_json` 表，如下所示：

```
mydb=> 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=166.30..14178.17 rows=10329 width=68)
(actual time=1.167..26.534 rows=9998 loops=1)
  Recheck Cond: (((user_info ->> 'id'::text))::integer > 1) AND (((user_
info ->> 'id'::text))::integer < 10000))
  Heap Blocks: exact=338
-> Bitmap Index Scan on idx_gin_user_info_id  (cost=0.00..163.72 rows=10329
width=0) (actual time=1.110..1.110 rows=19996 loops=1)
  Index Cond: (((user_info ->> 'id'::text))::integer > 1) AND (((user_
info ->> 'id'::text))::integer < 10000))
Planning time: 0.094 ms
Execution time: 27.092 ms
(7 rows)

```

根据以上结果可以看出，查询表tbl_user_json的user_info字段id键值在1到10000范围内的记录走了索引，并且执行时间为27.092毫秒，接着测试对tbl_user_jsonb表执行同样SQL的检索性能，如下所示：

```

mydb=> 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=158.93..14316.93 rows=10000 width=68)
(actual time=1.140..8.116 rows=9998 loops=1)
  Recheck Cond: (((user_info ->> 'id'::text))::integer > 1) AND (((user_
info ->> 'id'::text))::integer < 10000))
  Heap Blocks: exact=393
-> Bitmap Index Scan on idx_gin_user_infob_id  (cost=0.00..156.43 rows=
10000 width=0) (actual time=1.058..1.058 rows=18992 loops=1)
  Index Cond: (((user_info ->> 'id'::text))::integer > 1) AND (((user_
info ->> 'id'::text))::integer < 10000))
Planning time: 0.104 ms
Execution time: 8.656 ms
(7 rows)

```

根据以上结果可以看出，查询表tbl_user_jsonb的user_info字段id键值在1到10000范围内的记录走了索引并且执行时间为8.656毫秒，从这个测试看出jsonb检索比json效率高。

从以上两个测试看出，正好验证了“json写入比jsonb快，但检索时比jsonb慢”的观点，值得一提的是如果需要通过key/value进行检索，例如：

```
SELECT * FROM tbl_user_jsonb WHERE user_info @> '{"user_name": "2_francs"}';
```

这时执行计划为全表扫描，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM tbl_user_jsonb WHERE user_info @> '{"user_name": "2_francs"}';
                                         QUERY PLAN
-----
 Seq Scan on tbl_user_jsonb  (cost=0.00..67733.00 rows=2000 width=143) (actual
 time=0.018..582.207 rows=1 loops=1)
   Filter: (user_info @> '{"user_name": "2_francs"}'::jsonb)
   Rows Removed by Filter: 1999999
 Planning time: 0.065 ms
 Execution time: 582.232 ms
(5 rows)
```

从以上结果可以看出执行时间为 582 毫秒左右。在 `tbl_user_jsonb` 字段 `user_info` 上创建 gin 索引，如下所示：

```
mydb=> CREATE INDEX idx_tbl_user_jsonb_user_Info ON tbl_user_jsonb USING gin
          (user_Info);
CREATE INDEX
```

索引创建后，再次执行查询，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM tbl_user_jsonb WHERE user_info @> '{"user_name": "2_francs"}';
                                         QUERY PLAN
-----
 Bitmap Heap Scan on tbl_user_jsonb  (cost=37.50..3554.34 rows=2000 width=143)
   (actual time=0.079..0.080 rows=1 loops=1)
   Recheck Cond: (user_info @> '{"user_name": "2_francs"}'::jsonb)
   Heap Blocks: exact=1
     -> Bitmap Index Scan on idx_tbl_user_jsonb_user_info  (cost=0.00..37.00
           rows=2000 width=0) (actual time=0.069..0.069 rows=1 loops=1)
       Index Cond: (user_info @> '{"user_name": "2_francs"}'::jsonb)
 Planning time: 0.094 ms
 Execution time: 0.114 ms
(7 rows)
```

从以上看出走了索引，并且执行时间下降到了 0.114 毫秒。

这一节测试了 json、jsonb 数据类型读写性能差异，验证了 json 写入时比 jsonb 快，但检索时比 jsonb 慢的观点。

9.3 全文检索对 json 和 jsonb 数据类型的支持

前两小节介绍了 jsonb 索引创建以及 json、jsonb 读写性能的差异，这一小节将介绍 PostgreSQL10 的一个新特性：全文检索对 json、jsonb 数据类型的支持，本小节分两部分，第一部分简单介绍 PostgreSQL 全文检索，第二部分演示全文检索对 json、jsonb 数据类型的支持。

9.3.1 PostgreSQL 全文检索简介

对于大多数应用来说全文检索很少在数据库中实现，一般使用单独的全文检索引擎，例如基于 SQL 的全文检索引擎 Sphinx。PostgreSQL 支持全文检索，对于规模不大的应用如果不想搭建专门的搜索引擎，PostgreSQL 的全文检索也可以满足需求。

如果没有使用专门的搜索引擎，大部分检索需要通过数据库 like 操作匹配，这种检索方式的主要缺点在于：

- 不能很好地支持索引，通常需全表扫描检索数据，数据量大时检索性能很低。
- 不提供检索结果排序，当输出结果数据量非常大时表现更加明显。

PostgreSQL 全文检索能有效地解决这个问题，PostgreSQL 全文检索通过以下两种数据类型来实现。

1. tsvector

tsvector 全文检索数据类型代表一个被优化的可以基于搜索的文档，要将一串字符串转换成 tsvector 全文检索数据类型，代码如下所示：

```
mydb=> SELECT 'Hello,cat,how are u? cat is smiling!' ::tsvector;
          tsvector
-----
      'Hello,cat,how' 'are' 'cat' 'is' 'smiling!' 'u?'
(1 row)
```

可以看到，字符串的内容被分隔成好几段，但通过 ::tsvector 只是做类型转换，没有进行数据标准化处理，对于英文全文检索可通过函数 to_tsvector 进行数据标准化，如下所示：

```
mydb=> SELECT to_tsvector('english','Hello cat;');
          to_tsvector
-----
      'cat':2 'hello':1
(1 row)
```

2. tsquery

tsquery 表示一个文本查询，存储用于搜索的词，并且支持布尔操作“&”、“|”、“!”将字符串转换成 tsquery，如下所示：

```
mydb=> SELECT 'hello&cat'::tsquery;
          tsquery
-----
      'hello' & 'cat'
(1 row)
```

上述只是转换成 tsquery 类型，而并没有做标准化，使用 to_tsquery 函数可以执行标准化，如下所示：

```
mydb=> SELECT to_tsquery( 'hello&cat' );
          to_tsquery
```

```
-----  
'hello' & 'cat'  
(1 row)
```

一个全文检索示例如下所示，用于检索字符串是否包括“hello”和“cat”字符，本例中返回真。

```
mydb=> SELECT to_tsvector('english','Hello cat,how are u') @@  
to_tsquery( 'hello&cat' );  
?column?  
-----  
t  
(1 row)
```

检索字符串是否包含字符“hello”和“dog”，本例中返回假，代码如下所示。

```
mydb=> SELECT to_tsvector('english','Hello cat,how are u') @@  
to_tsquery( 'hello&dog' );  
?column?  
-----  
f  
(1 row)
```

有兴趣的读者可以测试 tsquery 的其他操作符，例如“|”“!”等。

 **注意** 这里使用了带双参数的 to_tsvector 函数，函数 to_tsvector 双参数的格式如下所示：to_tsvector([config regconfig ,] document text)，本节 to_tsvector 函数指定了 config 参数为 english，如果不指定 config 参数，则默认使用 default_text_search_config 参数的配置。

3. 英文全文检索例子

下面演示一个英文全文检索示例，创建一张测试表并插入 200 万测试数据，如下所示：

```
mydb=> CREATE TABLE test_search(id int4,name text);  
CREATE TABLE  
mydb=> INSERT INTO test_search(id,name) SELECT n, n||'_francs'  
FROM generate_series(1,2000000) n;  
INSERT 0 2000000
```

执行以下 SQL，查询 test_search 表 name 字段包含字符 1_francs 的记录。

```
mydb=> SELECT * FROM test_search WHERE name LIKE '1_francs';  
 id |   name  
-----+-----  
 1 | 1_francs  
(1 row)
```

执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM test_search WHERE name LIKE '1_francs';  
 QUERY PLAN
```

```

Seq Scan on test_search  (cost=0.00..38465.04 rows=204 width=18) (actual
    time=0.022..261.766 rows=1 loops=1)
  Filter: (name ~ 'l_francs'::text)
  Rows Removed by Filter: 1999999
Planning time: 0.101 ms
Execution time: 261.796 ms
(5 rows)

```

以上执行计划进行了全表扫描，执行时间为 261 毫秒左右，性能很低，接着创建索引，如下所示：

```

mydb=> CREATE INDEX idx_gin_search ON test_search USING gin
(to_tsvector('english',name));
CREATE INDEX

```

执行以下 SQL，查询 test_search 表 name 字段包含字符 l_francs 的记录。

```

mydb=> SELECT * FROM test_search WHERE to_tsvector('english',name) @@ 
to_tsquery('english','l_francs');
   id |      name
-----+-----
   1 | l_francs
(1 row)

```

再次查看执行计划和执行时间，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT * FROM test_search WHERE to_tsvector('english',
    name) @@
to_tsquery('english','l_francs');
                                QUERY PLAN
-----
Bitmap Heap Scan on test_search  (cost=18.39..128.38 rows=50 width=36) (actual
    time=0.071..0.071 rows=1 loops=1)
  Recheck Cond: (to_tsvector('english'::regconfig, name) @@ ''''l'' &
    ''franc'''::tsquery)
  Heap Blocks: exact=1
-> Bitmap Index Scan on idx_gin_search  (cost=0.00..18.38 rows=50
    width=0) (actual time=0.064..0.064 rows=1 loops=1)
  Index Cond: (to_tsvector('english'::regconfig, name) @@ ''''l'' &
    ''franc'''::tsquery)
Planning time: 0.122 ms
Execution time: 0.104 ms
(7 rows)

```

创建索引后，以上查询走了索引并且执行时间下降到 0.104 毫秒，性能提升了 3 个数量级，值得一提的是如果将 SQL 修改为不走索引，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT * FROM test_search
      WHERE to_tsvector(name) @@ to_tsquery('l_francs');
                                QUERY PLAN
-----
Seq Scan on test_search  (cost=0.00..1037730.00 rows=50 width=18) (actual

```

```

      time=0.036..10297.764 rows=1 loops=1
      Filter: (to_tsvector(name) @@ to_tsquery('l_francs'::text))
      Rows Removed by Filter: 1999999
Planning time: 0.098 ms
Execution time: 10297.787 ms
(5 rows)

```

由于创建索引时使用的是 `to_tsvector('english',name)` 函数索引，带了两个参数，因此 where 条件中的 `to_tsvector` 函数带两个参数才能走索引，而 `to_tsvector(name)` 不走索引。

9.3.2 json、jsonb 全文检索实践

在 PostgreSQL10 版本之前全文检索不支持 json 和 jsonb 数据类型，10 版本的一个重要特性是全文检索支持 json 和 jsonb 数据类型，这一小节将演示 10 版本的这个新特性。

1. PostgreSQL10 版本与 9.6 版本 `to_tsvector` 函数的差异

先来看看 9.6 版本的 `to_tsvector` 函数，如下所示：

```
[postgres@phghost1 ~]$ psql francs francs
psql (9.6.3)
Type "help" for help.
```

```
mydb=> \df *to_tsvector*
```

List of functions

Schema	Name	Result data type	Argument data types	Type
pg_catalog	array_to_tsvector	tsvector	text[]	normal
pg_catalog	to_tsvector	tsvector	regconfig, text	normal
pg_catalog	to_tsvector	tsvector	text	normal
(3 rows)				

从以上看出 9.6 版本 `to_tsvector` 函数的输入参数仅支持 `text`、`text[]` 数据类型，接着看看 10 版本的 `to_tsvector` 函数，如下所示：

```
[postgres@phghost1 ~]$ psql mydb pguser
psql (10.0)
Type "help" for help.
mydb=> \df *to_tsvector*
```

List of functions

Schema	Name	Result data type	Argument data types	Type
pg_catalog	array_to_tsvector	tsvector	text[]	normal
pg_catalog	to_tsvector	tsvector	json	normal
pg_catalog	to_tsvector	tsvector	jsonb	normal
pg_catalog	to_tsvector	tsvector	regconfig, json	normal
pg_catalog	to_tsvector	tsvector	regconfig, jsonb	normal
pg_catalog	to_tsvector	tsvector	regconfig, text	normal
pg_catalog	to_tsvector	tsvector	text	normal
(7 rows)				



从以上看出，10 版本的 to_tsvector 函数支持的数据类型增加了 json 和 jsonb。

2. 创建数据生成函数

为了便于生成测试数据，创建以下两个函数用来随机生成指定长度的字符串，random_range(int4, int4) 函数的代码如下所示：

```
CREATE OR REPLACE FUNCTION random_range(int4, int4)
RETURNS int4
LANGUAGE SQL
AS $$
    SELECT ($1 + FLOOR(( $2 - $1 + 1) * random() ))::int4;
$$;
```

接着创建 random_text_simple(length int4) 函数，此函数会调用 random_range(int4, int4) 函数，其代码如下所示：

```
CREATE OR REPLACE FUNCTION random_text_simple(length int4)
RETURNS text
LANGUAGE PLPGSQL
AS $$
DECLARE
    possible_chars text := '0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ';
    output text := '';
    i int4;
    pos int4;
BEGIN

    FOR i IN 1..length LOOP
        pos := random_range(1, length(possible_chars));
        output := output || substr(possible_chars, pos, 1);
    END LOOP;

    RETURN output;
END;
$$;
```

random_text_simple(length int4) 函数可以随机生成指定长度字符串，下列代码随机生成含三位字符的字符串：

```
mydb=> SELECT random_text_simple(3);
random_text_simple
-----
LL9
(1 row)
```

随机生成含六位字符的字符串，如下所示：

```
mydb=> SELECT random_text_simple(6);
random_text_simple
```



```
-----  
B81BPW  
(1 row)
```

后面会使用这个函数生成测试数据。

3. 创建 json 测试表

创建 user_ini 测试表，并通过 random_text_simple(length int4) 函数插入 100 万随机生成的六位字符的字符串，作为测试数据，如下所示：

```
mydb=> 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  
  
mydb=> INSERT INTO user_ini(id,user_id,user_name)  
SELECT r,round(random()*1000000), random_text_simple(6)  
FROM generate_series(1,1000000) as r;  
INSERT 0 1000000
```

创建 tbl_user_search_json 表，并通过 row_to_json 函数将表 user_ini 的行数据转换成 json 数据，如下所示：

```
mydb=> CREATE TABLE tbl_user_search_json(id serial, user_info json);  
CREATE TABLE  
  
mydb=> INSERT INTO tbl_user_search_json(user_info)  
SELECT row_to_json(user_ini) FROM user_ini;  
INSERT 0 1000000
```

所生成的数据如下所示：

```
mydb=> SELECT * FROM tbl_user_search_json LIMIT 1;  
 id | user_info  
----+-----  
 1 | {"id":1,"user_id":186536,"user_name":"KTU89H","create_time":"2017-  
 08-05T15:59:25.359148+08:00"}  
(1 row)
```

4. json 数据全文检索测试

使用全文检索查询表 tbl_user_search_json 的 user_info 字段中包含 KTU89H 字符的记录，如下所示：

```
mydb=> SELECT * FROM tbl_user_search_json  
WHERE to_tsvector('english',user_info) @@ to_tsquery('ENGLISH','KTU89H');  
 id | user_info  
----+-----  
 1 | {"id":1,"user_id":186536,"user_name":"KTU89H","create_time":"2017-  
 08-05T15:59:25.359148+08:00"}  
(1 row)
```



以上 SQL 能正常执行说明全文检索支持 json 数据类型，只是上述 SQL 进行了全表扫描，性能较低，执行时间为 8061 毫秒，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM tbl_user_search_json
      WHERE to_tsvector('english',user_info) @@ to_tsquery('ENGLISH','KTU89H');
          QUERY PLAN
-----
Seq Scan on tbl_user_search_json  (cost=0.00..279513.00 rows=5000 width=104)
  (actual time=0.046..8061.858 rows=1 loops=1)
    Filter: (to_tsvector('english'::regconfig, user_info) @@ '''ktu89h'''::tsquery)
    Rows Removed by Filter: 999999
Planning time: 0.091 ms
Execution time: 8061.880 ms
(5 rows)
```

创建如下索引：

```
mydb=> CREATE INDEX idx_gin_search_json ON tbl_user_search_json USING
      gin(to_tsvector('english',user_info));
      CREATE INDEX
```

索引创建后，再次执行以下 SQL，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM tbl_user_search_json WHERE to_tsvector('english',
      user_info) @@ to_tsquery('ENGLISH','KTU89H');
          QUERY PLAN
-----
Bitmap Heap Scan on tbl_user_search_json  (cost=50.75..7876.06 rows=5000 width=104)
  (actual time=0.024..0.024 rows=1 loops=1)
    Recheck Cond: (to_tsvector('english'::regconfig, user_info) @@ '''ktu89h'''::tsquery)
    Heap Blocks: exact=1
    -> Bitmap Index Scan on idx_gin_search_json  (cost=0.00..49.50 rows=5000
        width=0) (actual time=0.018..0.018 rows=1 loops=1)
      Index Cond: (to_tsvector('english'::regconfig, user_info) @@ '''ktu89h'''::tsquery)
      Planning time: 0.113 ms
      Execution time: 0.057 ms
(7 rows)
```

从上述执行计划看出走了索引，并且执行时间降为 0.057 毫秒，性能非常不错。

这一小节前一部分对 PostgreSQL 全文检索的实现做了简单介绍，并且给出了一个英文检索的例子，后一部分通过示例介绍了 PostgreSQL10 的一个新特性，即全文检索对 json、jsonb 类型的支持。

9.4 本章小结

本章进一步介绍了 PostgreSQL 的 NoSQL 特性，首先介绍了 jsonb 数据类型索引相关



的内容，之后通过示例对比 json、jsonb 两种 json 数据类型读写性能的差异，最后介绍了 PostgreSQL 全文检索以及全文检索对 json、jsonb 类型的支持（PostgreSQL10 新特性），通过阅读本节读者对 json、jsonb 的使用有了进一步理解。本章给出了 PostgreSQL 英文全文检索的示例，值得一提的是，PostgreSQL 对中文检索也是支持的，有兴趣的读者可自行测试。