规则更准确地说是查询重写规则。 从使用上来说， 规则系统上的一些功能也可以通过函数和触发器来实现， 但规则系统与触发器完全不同， 它是把用户发过来的SQL命令在执行前通过内部的规则定义改编成另一个SQL命令后再执行的一种方式。

**SELECT规则**

PostgreSQL的视图是通过SELECT规则来实现的。

创建一张测试表：

CREATE TABLE users (id int,name text,password text,age int);

在users表上建一个SELECT规则：

create table v\_user(id int,name text,age int);

CREATE RULE "\_RETURN" AS ON SELECT TO v\_user DO INSTEAD SELECT id,name,age FROM users;

SELECT规则的后续动作只能是INSTEAD SELECT， SELECT规则的名称只能是“\_RETURN”

\d v\_user

查看v\_user定义，显示为视图

**更新规则**

**语法格式：**

CREATE [ OR REPLACE ] RULE name AS ON event

TO table\_name

DO [ ALSO | INSTEAD ] { NOTHING | command }

event 可以取以下值：

1.SELECT： 当SQL的查询计划中存在查询表的操作时会重写查询计划。

2.INSERT： 当SQL的查询计划中存在向表中插入数据的操作时会重写查询计划。

3.UPDATE： 当SQL的查询计划中存在向表更新数据的操作时会重写查询计划。

4.DELETE： 当SQL的查询计划中存在从表中删除数据的操作时会重写查询计划。

更新规则指事务为“INSERT”“UPDATE”“DELETE”的这3种规则。

ALSO”与“INSTEAD”的区别：

1.ALSO： 除执行原操作外还执行一些附加操作， 这些附加操作由后面的“command”指定。

2.INSTEAD： 把原操作替换为后面的“command”的操作。

创建规则语法中的“NOTHING”表示什么都不执行。

**示例**

1.对表的操作记录日志表中：

创建测试表：

CREATE TABLE t1(id int, name text,age int);

创建日志表：

CREATE TABLE t1\_log(

id serial primary key,

opertype text,

opertime timestamp,

old\_id int,

new\_id int,

old\_name text,

new\_name text,

old\_age int,

new\_age int);

创建3个规则：

CREATE RULE rule\_t1\_insert AS ON INSERT TO t1

DO ALSO INSERT INTO t1\_log(opertype, opertime, new\_id, new\_name,new\_age) VALUES('insert', now(), new.id, new.name,new.age);

CREATE RULE rule\_t1\_update AS ON UPDATE TO t1

DO ALSO INSERT INTO t1\_log(opertype, opertime, old\_id, new\_id, old\_name, new\_name,old\_age,new\_age)

VALUES('update', now(), old.id, new.id, old.name, new.name,old.age,new.age);

CREATE RULE rule\_t1\_delete AS ON DELETE to t1

DO ALSO INSERT INTO t1\_log(opertype, opertime, old\_id, old\_name,old\_age) VALUES('delete', now(), old.id, old.name,old.age);

测试插入、 更新、 删除效果：

insert into t1 values(1, 'zhangsan',22);

insert into t1 values(2, 'lisi',24);

update t1 set name ='aaa' where id=1;

update t1 set name ='bbb' where id=2;

delete from t1;

select \* from t1\_log;

**规则和权限**

规则（RULE） 是从属于表或视图的。 如果一张表属于一个用户， 则这张表上的所有规则都是属于这个用户的。 由于PostgreSQL规则系统对查询进行了重写， 因此通过规则系统用户可间接访问到其他表， 如果用户对这些间接表没有权限是否会报权限错误呢？

**示例1：**

用户user1有一张视图v\_user：

create view v\_user as select id from users;

其中表users的属主也是user1用户。

把视图查询的权限赋于用户user2：

grant select on v\_user to user2;

以user2身份登录系统执行：
select \* from v\_user;--SUCCESS
select id from users;--ERROR

user2可以查询v\_user视图， 而user2对视图的底层表users没有权限， 直接查询users时报错。

**示例2：**

在user2用户下创建一张视图v\_t1， 而此视图的底层表为t1，t1的属主是user1：

create view v\_t1 as select id from t1;

用户user2不能访问底层表t1， 但可以在这张表上建视图。 查询视图：

select \* from v\_t1;--ERROR

user1不能访问v\_t1，因为v\_t1的属主是user2

select \* from v\_t1;--ERROR

user2用户可以在不能访问的表上建视图， 但查询视图时会因无底层表的查询权限而查询失败。 这时再让user2把视图 v\_t1 的查询权限赋予user1，现在来看看此时user1登录系统后能否查询视图 v\_t1：

select \* from v\_t1;--ERROR

select \* from t1;--SUCCESS

在规则系统中检测权限， 是对规则的属主进行权限检查。 因 v\_t1 的属主为user2， 在检查权限时， 是检查user2的权限， 而不是 user1 的权限， 所以用户 user1 查询视图 v\_t1 的操作也会失败。

**规则与触发器的比较**

从外部导入数据的命令 COPY FROM 会让触发器执行， 但不会调用规则系统。

创建测试表：

create table t2(id int,name text,age int);

insert into t2 select n,md5(n::text),random()\*30 from generate\_series(1,100) n;

copy t2 to '/tmp/t2.sql';

示例：对表 student 的更新情况记录log

CREATE TABLE log\_student(

update\_time timestamp,

db\_user varchar(40),

oper\_type varchar(6));

创建函数：

CREATE FUNCTION log\_student\_function()

RETURNS trigger

LANGUAGE plpgsql

AS $$

BEGIN

INSERT INTO log\_student values(now(), user, TG\_OP);

RETURN NULL;

END $$ ;

创建触发器：

CREATE TRIGGER log\_student\_trigger

AFTER INSERT OR DELETE OR UPDATE ON student

FOR ROW EXECUTE PROCEDURE log\_student\_function();

copy student from '/tmp/t2.sql';

触发器能做的很多事情使用PostgreSQL的规则系统也可以完成， 使用哪种方法取决于具体的应用场景。 规则系统是通过查询重写来实现的， 而触发器通常是为每个行都触发执行一次， 所以对于批量操作， 使用规则可能会生成更好的执行计划， 从而效率更高一些。

insert into student select n,md5(n::text),random()\*30 from generate\_series(1,100000) n;

insert into t1 select n,md5(n::text),random()\*30 from generate\_series(1,100000) n;