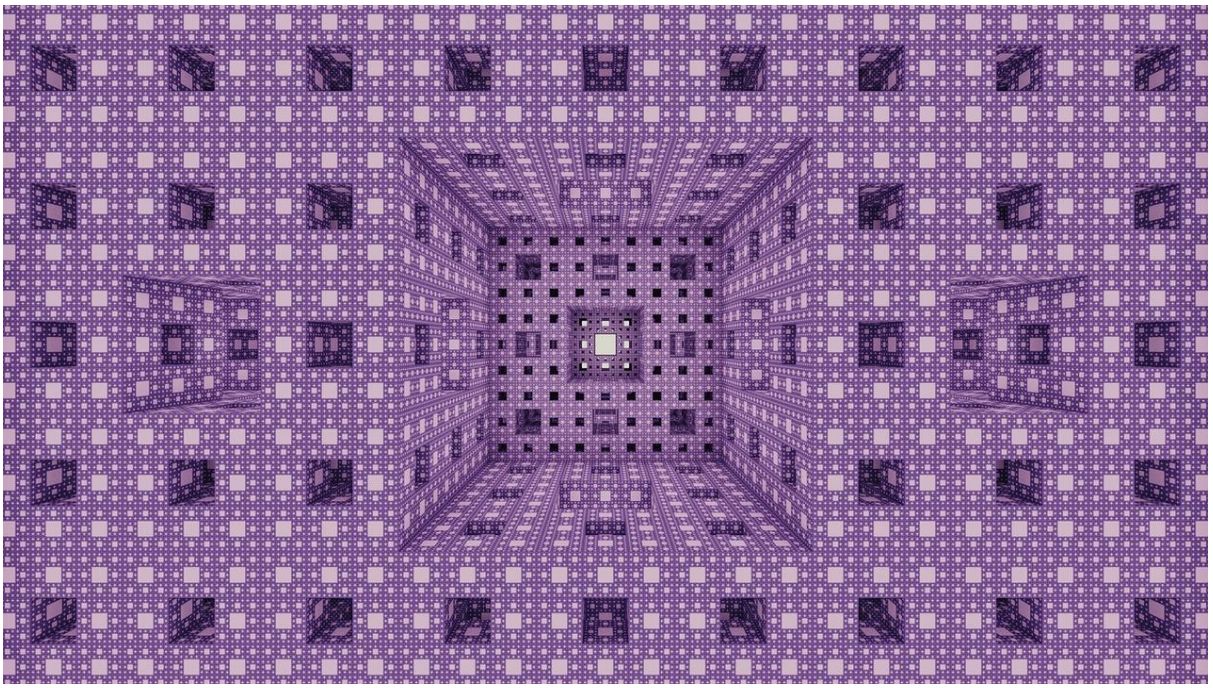
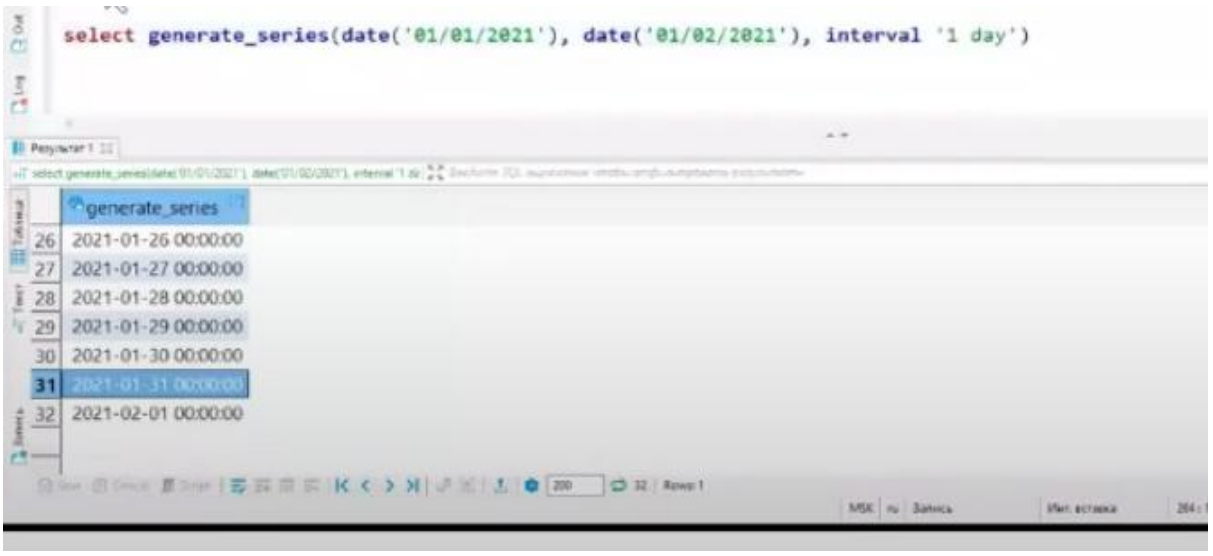
5. Различие между generate_series и рекурсией



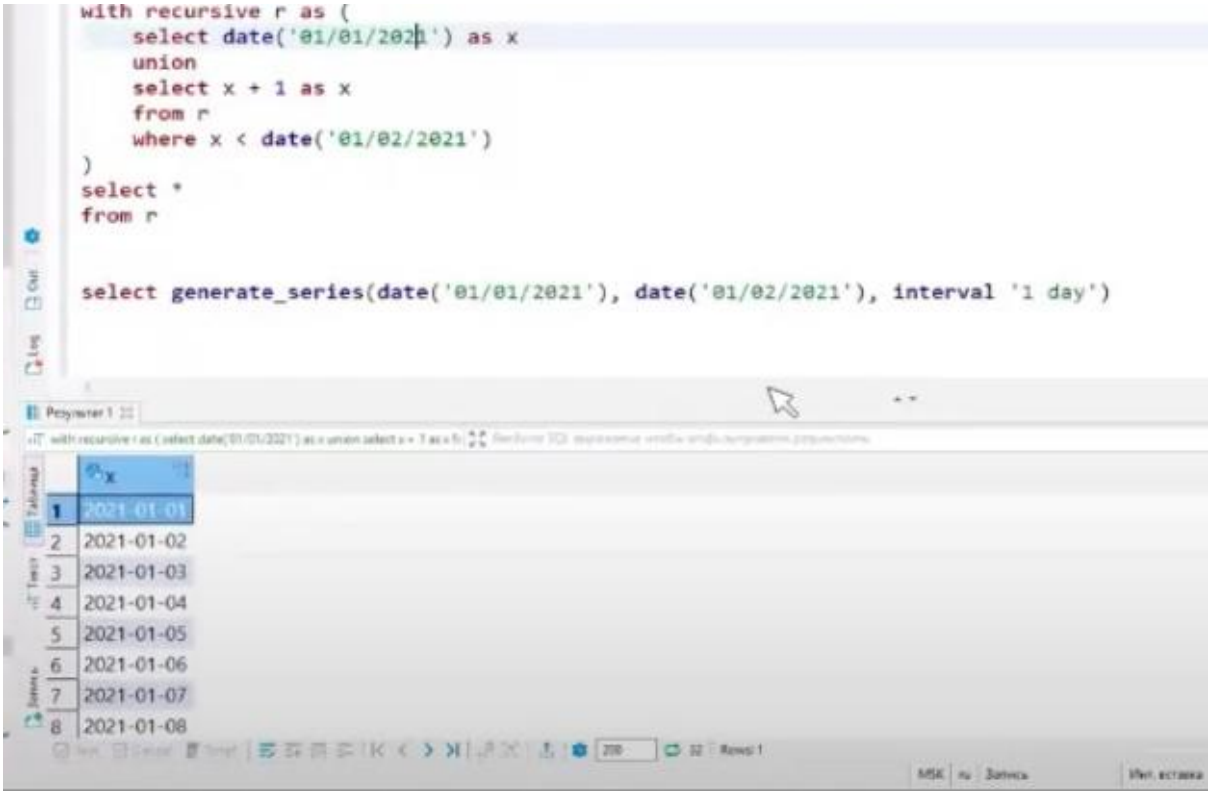
Смысл **рекурсии** в том, что мы получили стартовую часть, осуществили цикл, получили измененные данные и работаем дальше с измененными данными.
Гоняем данные по циклу (кругу), пока не дойдем до условия, которое и будет выполнение останавливать.

Допустим, мы хотим сформировать календарь с 01.01.2021 по 01.02.2021 с интервалом в 1 день.

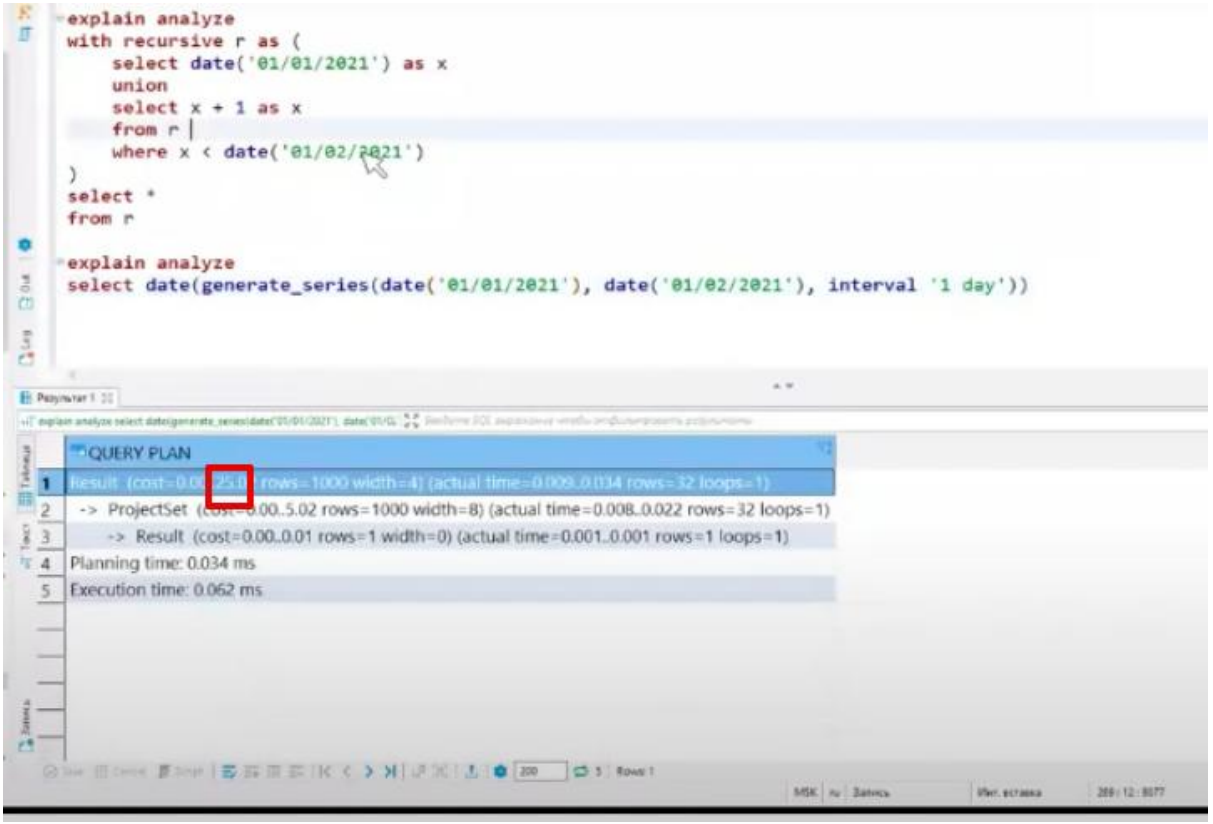
Функция **generate_series** генерирует ряд значений, она работает как с числами, так и с датами.
Задали число/дату, откуда начинаем, где заканчиваем и соответственно с каким шагом.



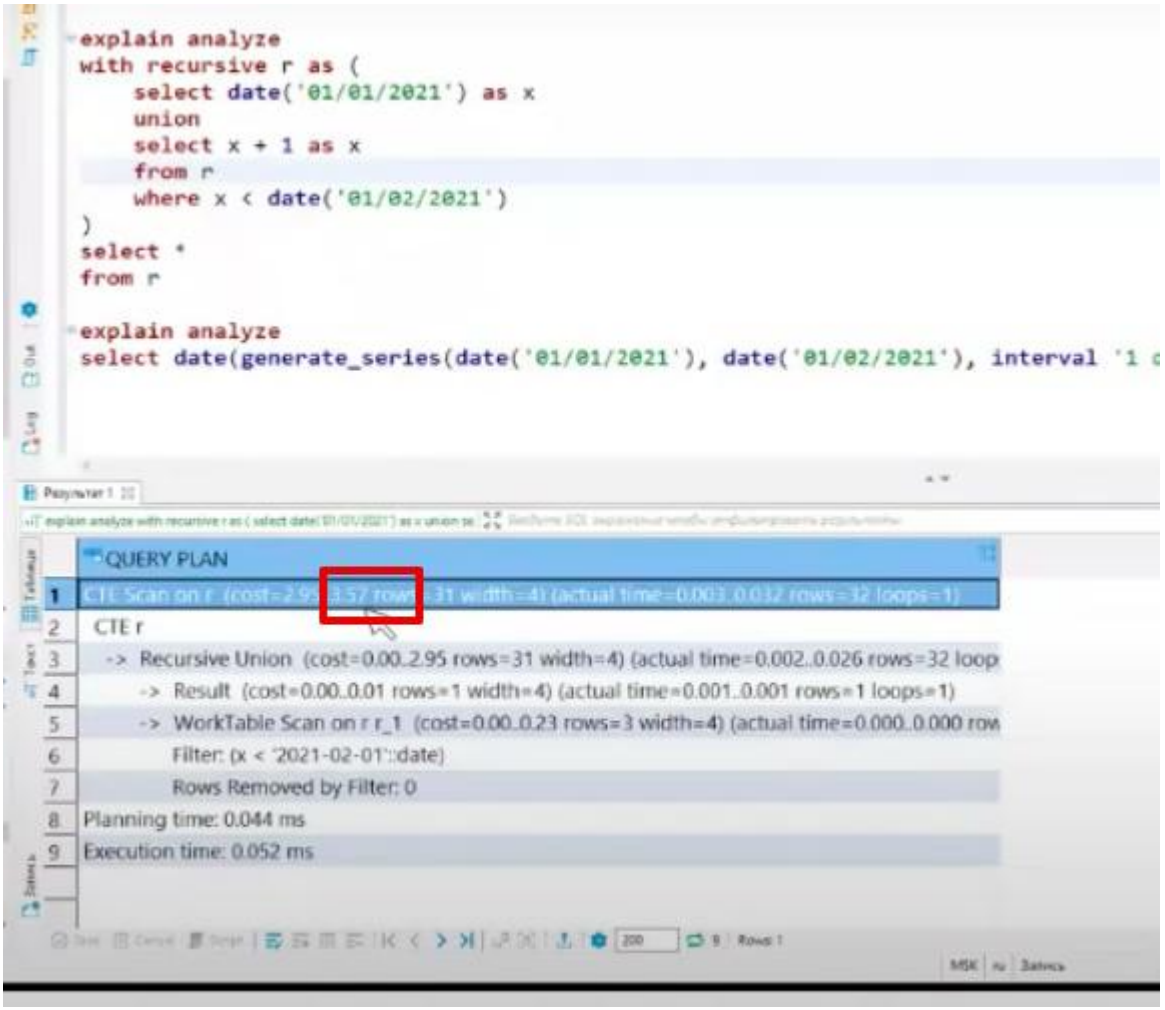
То же самое мы можем сделать с **рекурсией** - формируем точно такой же календарик:



Но мы понимаем, что **generate_series** работает не во всех СУБД, преимущественно только в Postgre и через **generate_series** у нас “скушало” 25 ресурсов,



а через **рекурсию** - 3,57.



Универсальное решение или специфичное: как узнать?

Generate_series - это не универсальное решение (не чистый SQL), это специфика Postgre. **Рекурсия** - это будет классический SQL. Надо помнить, что универсальные решения (классический SQL) используются во всех СУБД.
Определить, универсальное это решение или специфическое, можно только в документации (там указывается, например, что применимо к Postgre, плюс всегда можно посмотреть, что относится к нативному SQL).

Лирическо-практическое отступление.
В свое время на Хабре провели очень интересное исследование, когда взяли разработчика-программиста и стоматолога (из профессии, где примерно такой же уровень квалификации, и примерно такой же уровень зарплаты). Стоматологу во время работы надо держать в голове порядка 100-150 операций. Программисту - 1000-1500 операций. Есть ли возможность держать в голове такой объем? Конечно нет. Вы должны уметь работать с документацией. Умение работать с документацией - это 50% того, что вы трудоустроитесь.

Следует учитывать, что иногда простой лаконичный запрос может потреблять дикое количество ресурсов, а сложный - типа рекурсии, окон и пр. может привести практически к отсутствию затрат. Но все это приходит с опытом и пониманием, когда проверяется все самостоятельно. Вначале пишется простой запрос, после начинается рефакторинг: вы сделали основу, понимаете, что в рекурсии делать проверки некорректно - это будет трудозатратно. Соответственно, в рекурсии оставляете только формирование календаря, а далее уже вне рекурсии начинаете работать с данными.

Запросы для самостоятельной работы

Есть исходные данные:

```
create table test (
        date_event timestamp,
        field varchar(50),
        old_value varchar(50),
        new_value varchar(50)
)
insert into test (date_event, field, old_value, new_value)
values
('2017-08-05', 'val', 'ABC', '800'),
('2017-07-26', 'pin', '', '10-AA'),
('2017-07-21', 'pin', '300-L', ''),
('2017-07-26', 'con', 'CC800', 'null'),
('2017-08-11', 'pin', 'EKH', 'ABC-500'),
('2017-08-16', 'val', '990055', '100')
select * from test order by date(date_event)
```

В данной таблице хранят информацию по изменению "статуса" для каждого типа поля (field).

То есть, есть поле pin, на 21.07.2017 было изменено значение, соответственно новое (new_value) стало '' (пустая строка) и старое (old_value), записалось как '300-L'.

Далее 26.07.2017 изменили значение с '' (пустая строка) на '10-AA'. И так по разным полям в разные даты были какие-то изменения значений.

Задача: составить запрос таким образом, что бы в новой результирующей таблице был календарь изменения значений для каждого поля.

Всего три столбца: дата, поле, текущий статус.

То есть для каждого поля будет отображение каждого дня с отображением текущего статуса. К примеру, для поля pin на 21.07.2017 статус будет '' (пустая строка), на 22.07.2017 - '' (пустая строка). и т.д. до 26.07.2017, где статус станет '10-AA'

Решение должно быть универсальным для любого SQL, не только под PostgreSQL ;)

(маленькая таблица исходные данные, большая - фрагмент нужного результата)

```
with recursive r(a, b, c) as (
    select temp_t.i, temp_t.field, t.new_value
    from
            (select min(date(t.date_event)) as i, f.field
            from test t, (select distinct field from test) as f
            group by f.field) as temp_t
    left join test t on temp_t.i = t.date_event and temp_t.field = t.field
    union all
    select a + 1, b,
        case
                when t.new_value is null then c
                else t.new_value
                end
    from r
    left join test t on t.date_event = a + 1 and b = t.field
    where a < (select max(date(date_event)) from test)
)
SELECT *
FROM r
order by b, a
```

```
select
	gs::date as change_date,
	fields.field as field_name,
	case
		when (
			select new_value
			from test t
			where t.field = fields.field and t.date_event = gs::date) is not null
			then (
				select new_value
				from test t
				where t.field = fields.field and t.date_event = gs::date)
		else (
			select new_value
			from test t
			where t.field = fields.field and t.date_event < gs::date
			order by date_event desc
			limit 1)
	end as field_value
from
	generate_series((select min(date(date_event)) from test), (select
max(date(date_event)) from test), interval '1 day') as gs,
	(select distinct field from test) as fields
order by
	field_name, change_date
select
	distinct field, gs, first_value(new_value) over (partition by value_partition)
```

```
from
	(select
		t2.*,
		t3.new_value,
		sum(case when t3.new_value is null then 0 else 1 end) over (order by t2.field, t2.gs) as value_partition
	from
		(select
			field,
			generate_series((select min(date_event)::date from test), (select max(date_event)::date from test), interval '1 day')::date as gs
		from test) as t2
	left join test t3 on t2.field = t3.field and t2.gs = t3.date_event::date) t4
order by
	field, gs
with recursive r as (
	--стартовая часть рекурсии
		select
		min(t.date_event) as c_date
		  ,max(t.date_event) as max_date
	from test t
	union
	-- рекурсивная часть
	select
	    r.c_date+ INTERVAL '1 day' as c_date
	   ,r.max_date
	from r
```

```
        where r.c_date < r.max_date
 ),
t as (select t.field
                , t.new_value
                , t.date_event
                , case when lead(t.date_event) over (partition by t.field order by t.date_event)
is null
                        then max(t.date_event) over ()
                        else lead(t.date_event) over (partition by t.field order by
t.date_event)- INTERVAL '1 day'
                  end
                        as next_date
                , min (t.date_event) over () as min_date
                , max(t.date_event) over () as max_date
from (
select t1.date_event
                ,t1.field
                ,t1.new_value
                ,t1.old_value
from test t1
union all
select distinct min (t2.date_event) over () as date_event --первые стартовые даты
                ,t2.field
                ,null as new_value
                ,null as old_value
from test t2) t
)
```

```
select r.c_date, t.field , t.new_value

from r

join t on r.c_date between t.date_event and t.next_date

order by t.field,r.c_date
```