**Задание**

Из задания к курсовому проекту возьмем описание одной из таблиц – «… экологических катастрофах (предприятие, тип катастрофы, дата, количество пострадавших людей, последствия, сведения об участии ЭС в устранении ЭК …». На основе этого описание создадим таблицу eco\_catastrophy как секционированную таблицу с предложением *PARTITION BY*, указав метод разбиения (в нашем случае *RANGE*) и список столбцов, которые будут образовывать ключ разбиения (в нашем случае это один столбец – *data\_catastrophe*).

**Создание партиции по диапазону:**

CREATE TABLE eco\_catastrophy (

"id" SERIAL NOT NULL,

id\_company int4 NOT NULL,

id\_type\_catastrophe int4 NOT NULL,

data\_catastrophe TIMESTAMP NOT NULL,

number\_victims int4 NOT NULL,

effects TEXT,

detriment DECIMAL,

id\_department int4[]

) PARTITION BY RANGE (data\_catastrophe);

**Создание секций:**

В определении каждой секции должны задаваться границы, соответствующие методу и ключу разбиения родительской таблицы. Секции, создаваемые таким образом, во всех отношениях являются обычными таблицами PostgreSQL. Указание границ, при котором множество значений новой секции пересекается со множеством значений в одной или нескольких существующих секциях, будет ошибочным. В нашем примере каждая секция должна содержать данные за один квартал.

CREATE TABLE eco\_catastrophy\_2020\_3kv PARTITION OF eco\_catastrophy

FOR VALUES FROM ('2020-07-01') TO ('2020-10-01');

CREATE TABLE eco\_catastrophy\_2020\_4kv PARTITION OF eco\_catastrophy

FOR VALUES FROM ('2020-10-01') TO ('2021-01-01');

CREATE TABLE eco\_catastrophy\_2021\_1kv PARTITION OF eco\_catastrophy

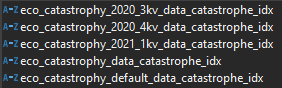
FOR VALUES FROM ('2021-01-01') TO ('2021-04-01');

CREATE TABLE eco\_catastrophy\_default PARTITION OF eco\_catastrophy DEFAULT;

**Создание индекса:**

Создадим в секционируемой таблице индекс по ключевому столбцу – *data\_catastrophe*. При этом автоматически будет создан соответствующий индекс в каждой секции и все секции, которые вы будете создавать или присоединять позднее, тоже будут содержать такой индекс. Индекс по ключу, строго говоря, создавать не обязательно, но в большинстве случаев он будет полезен

CREATE INDEX ON eco\_catastrophy (data\_catastrophe);



**Добавление данных:**

INSERT INTO eco\_catastrophy (id\_company, id\_type\_catastrophe, data\_catastrophe, number\_victims, effects, detriment, id\_department)

SELECT

trunc(random() \* 10),

trunc(random() \* 1000),

now() - '3 month'::interval \* random(),

trunc(random() \* 10000),

md5(random()::text),

(random() \* 100000)::decimal,

mas\_if((trunc(random() \* 10))::int4)

FROM

generate\_series(1, 100000);

Запускаем данную команду несколько раз, с разными параметрами интервала дат (*now() - '8 month'::interval \* random()*, *now() - '6 month'::interval \* random()*, *now() - '4 month'::interval \* random()*, *now() - '2 month'::interval \* random()*, *now() + '2 month'::interval \* random()*) и получаем 500 000 рандомных записей. Функция *mas\_if(in\_cnt)* – это пользовательская функция, которая на вход принимает размер массива (*in\_cnt*), целое число, а на выходе возвращает массив рандомных целых чисел.

**Результат добавления данных:**

SELECT

(SELECT count(\*) FROM eco\_catastrophy\_2020\_3kv) as eco\_catastrophy\_2020\_3kv,

(SELECT count(\*) FROM eco\_catastrophy\_2020\_4kv) as eco\_catastrophy\_2020\_4kv,

(SELECT count(\*) FROM eco\_catastrophy\_2021\_1kv) as eco\_catastrophy\_2021\_1kv,

(SELECT count(\*) FROM eco\_catastrophy\_default) as eco\_catastrophy\_default,

(SELECT count(\*) FROM ONLY eco\_catastrophy) as eco\_catastrophy;



**Создаем новую таблицу вне структуры секций и загружаем в нее данные из секции eco\_catastrophy\_default:**

CREATE TABLE eco\_catastrophy\_2021\_2kv

(LIKE eco\_catastrophy INCLUDING DEFAULTS INCLUDING CONSTRAINTS);

ALTER TABLE eco\_catastrophy\_2021\_2kv ADD CONSTRAINT catastrophy\_2021\_2kv

CHECK (data\_catastrophe >= DATE '2021-04-01' AND

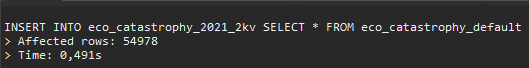
data\_catastrophe < DATE '2021-07-01');

INSERT INTO eco\_catastrophy\_2021\_2kv SELECT \* FROM eco\_catastrophy\_default;

С помощью предложение LIKE определяем таблицу, из которой в новую таблицу будут автоматически скопированы все имена столбцов, их типы данных и их ограничения на NULL. Дополняем командами что и как будем копировать: INCLUDING DEFAULTS – скопировать выражения значений по умолчанию в определениях копируемых столбцов, без этого указания выражения по умолчанию не копируются, и INCLUDING CONSTRAINTS – скопировать ограничения-проверки.

Создаем ограничение CHECK в присоединяемой таблице, соответствующее ожидаемому ограничению секции. Благодаря этому система сможет обойтись без сканирования, необходимого для проверки неявного ограничения секции.

Копируем все данные, которые хранятся в дефолтной секции в новую таблицу, которая пока еще не является секцией нашей партиции – *eco\_catastrophy*.



**Удаляем все данные из дефолтной секции:**

DELETE FROM eco\_catastrophy\_default;



**Делаем таблицу eco\_catastrophy\_2021\_2kv секцией:**

ALTER TABLE eco\_catastrophy ATTACH PARTITION eco\_catastrophy\_2021\_2kv

FOR VALUES FROM ('2021-04-01') TO ('2021-07-01');

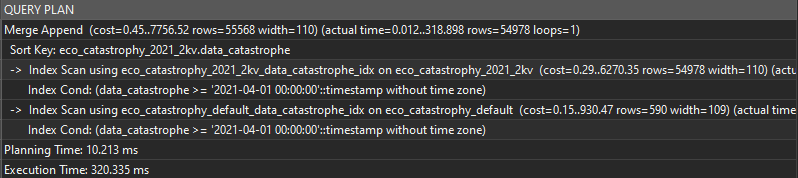


**Результат добавление новой секции eco\_catastrophy\_2021\_2kv:**

EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy

WHERE data\_catastrophe>='2021-04-01' ORDER BY data\_catastrophe



Исходя из построенного плана видим, что поиск идет по новой добавленной секции и когда мы делали операцию добавления новой секции, создался новый индекс (*eco\_catastrophy\_2021\_2kv\_data\_catastrophe\_idx*) по полю data\_*catastrophe*

**Создание партиции по списку:**

CREATE TABLE eco\_catastrophy\_list (

"id" SERIAL NOT NULL,

id\_company int4 NOT NULL,

id\_type\_catastrophe int4 NOT NULL,

data\_catastrophe TIMESTAMP NOT NULL,

number\_victims int4 NOT NULL,

effects TEXT,

detriment DECIMAL,

id\_department int4[]

) PARTITION BY LIST ("id\_company");

**Создание секций:**

CREATE TABLE eco\_catastrophy\_list\_0 PARTITION OF eco\_catastrophy\_list

FOR VALUES IN (0);

CREATE TABLE eco\_catastrophy\_list\_1\_3 PARTITION OF eco\_catastrophy\_list

FOR VALUES IN (1,3,5);

CREATE TABLE eco\_catastrophy\_list\_7\_9 PARTITION OF eco\_catastrophy\_list

FOR VALUES IN (7,9);

CREATE TABLE eco\_catastrophy\_list\_default PARTITION OF eco\_catastrophy\_list DEFAULT;

CREATE TABLE eco\_catastrophy\_list\_2\_4\_6\_8 PARTITION OF eco\_catastrophy\_list

FOR VALUES IN (2,4,6,8)

PARTITION BY RANGE (data\_catastrophe);

**Создание вложенных секций:**

CREATE TABLE eco\_catastrophy\_list\_2\_4\_6\_8\_2020 PARTITION OF

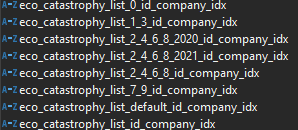
eco\_catastrophy\_list\_2\_4\_6\_8 FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');

CREATE TABLE eco\_catastrophy\_list\_2\_4\_6\_8\_2021 PARTITION OF

eco\_catastrophy\_list\_2\_4\_6\_8 FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');

**Создание индекса:**

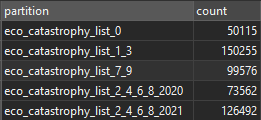
CREATE INDEX ON eco\_catastrophy\_list ("id\_company");



**Результат скопированных данных из eco\_catastrophy:**

SELECT tableoid::regclass AS partition, count(\*) FROM eco\_catastrophy\_list

GROUP BY tableoid;



**Создание партиции по хешу:**

CREATE TABLE eco\_catastrophy\_hash (

"id" SERIAL PRIMARY KEY NOT NULL,

id\_company int4 NOT NULL,

id\_type\_catastrophe int4 NOT NULL,

data\_catastrophe TIMESTAMP NOT NULL,

number\_victims int4 NOT NULL,

effects TEXT,

detriment DECIMAL,

id\_department int4[]

) PARTITION BY HASH ("id");

**Создание секций:**

CREATE TABLE eco\_catastrophy\_hash0 PARTITION OF eco\_catastrophy\_hash

FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE eco\_catastrophy\_hash1 PARTITION OF eco\_catastrophy\_hash

FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE eco\_catastrophy\_hash2 PARTITION OF eco\_catastrophy\_hash

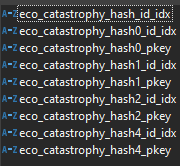
FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE eco\_catastrophy\_hash4 PARTITION OF eco\_catastrophy\_hash

FOR VALUES WITH (MODULUS 4, REMAINDER 3);

**Создание индекса:**

CREATE INDEX ON eco\_catastrophy\_hash ("id");



**Заполнение с автоматической раскладкой по секциям:**

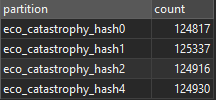
INSERT INTO eco\_catastrophy\_hash

SELECT \* FROM eco\_catastrophy ON CONFLICT ("id") DO NOTHING;

**Распределение строк по секциям происходит равномерно:**

SELECT tableoid::regclass AS partition, count(\*) FROM eco\_catastrophy\_hash

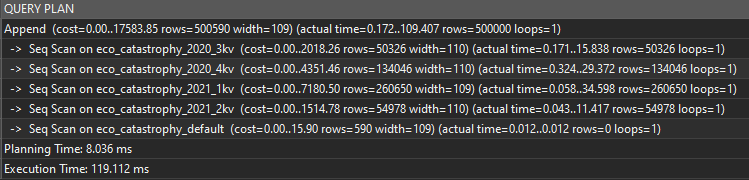
GROUP BY tableoid;



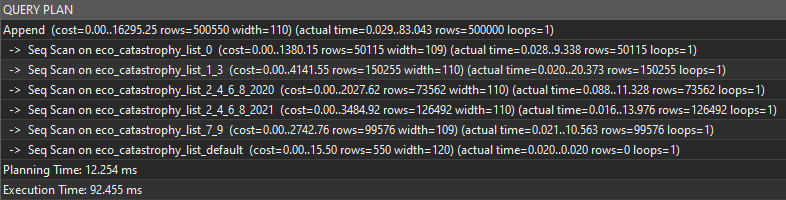
**Сравнение и анализ партиций и обычной таблицы:**

Создадим обычную таблицу (*eco\_catastrophy\_test*) с такими же полями как у партиций и добавим индекс по *id* (*CREATE INDEX ON eco\_catastrophy\_test ("id")* и заполним данными скопировав из партиции *eco\_catastrophy*).

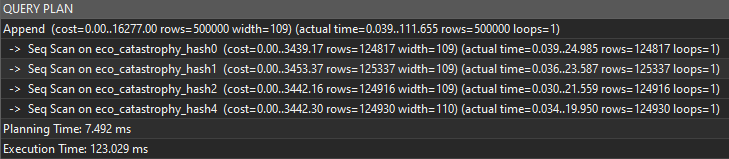
EXPLAIN (ANALYZE) SELECT \* FROM eco\_catastrophy;



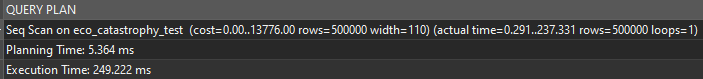
EXPLAIN (ANALYZE) SELECT \* FROM eco\_catastrophy\_list;



EXPLAIN (ANALYZE) SELECT \* FROM eco\_catastrophy\_hash;



EXPLAIN (ANALYZE) SELECT \* FROM eco\_catastrophy\_test;



Из результатов видим что по затраченному времени на запрос (Execution Time) показатель почти у всех на одном уровне (не считая *eco\_catastrophy\_test*), быстрее справилась партиция разбитая по списку (*eco\_catastrophy\_list*) – 92.455 ms, а медленнее обычная таблица (*eco\_catastrophy\_test*) – 249.222 ms, что почти в 2 раза, хотя количество записей одинаково.

Общее количество строк (rows), полученных при последовательном сканировании (Seq Scan) у всех одинаково – 500000. У партиций данный параметр можно посмотреть по каждой секции.

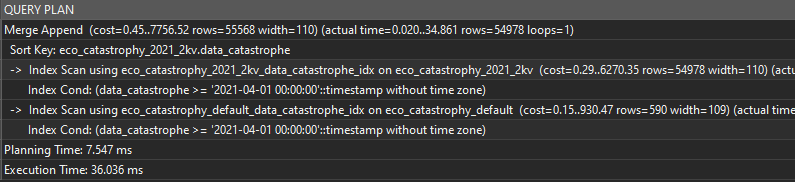
Оценка затратности операции (cost) у партиций почти одинаковы (eco\_catastrophy – 17583.85, *eco\_catastrophy\_list* – 16295.25, *eco\_catastrophy\_hash* – 16277.00) а у обычной таблицы (*eco\_catastrophy\_test*) – 13766.00.

Добавим к запросу условие на дату и сортировку. Сортировка, для партиции – *eco\_catastrophy* будет по полю *data\_catastrophe* (поле, по которому разбита партиция на секции), для *eco\_catastrophy\_list* будет по полю *id\_company* (поле, по которому разбита партиция по секциям), для *eco\_catastrophy\_hash* будет по полю *id* (поле, по которому разбита партиция по секциям) и для *eco\_catastrophy\_test* будет по полю *id*.

EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy

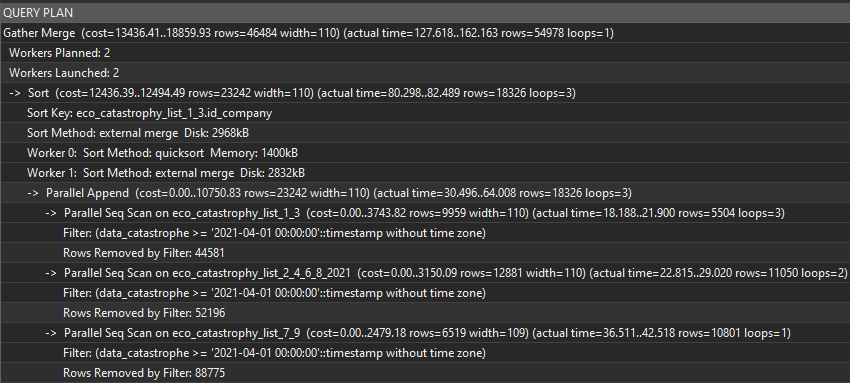
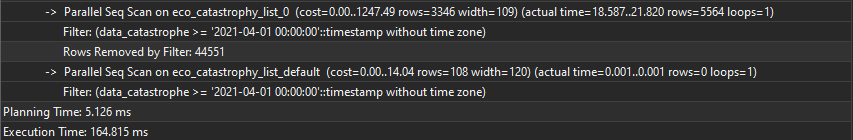
WHERE data\_catastrophe>='2021-04-01' ORDER BY data\_catastrophe;



EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy\_list

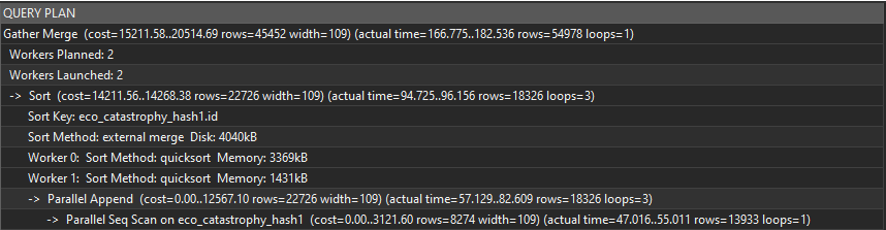
WHERE data\_catastrophe>='2021-04-01' ORDER BY id\_company;

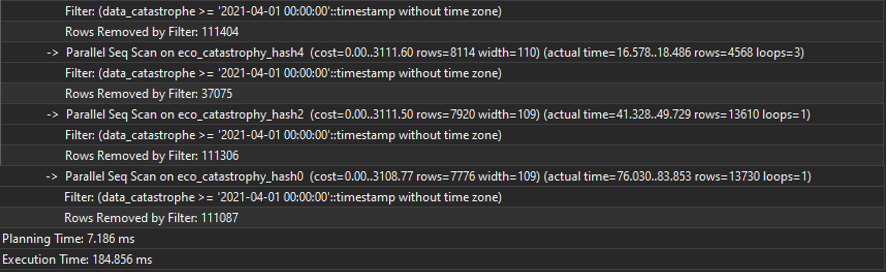
 

EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy\_hash

WHERE data\_catastrophe>='2021-04-01' ORDER BY "id";

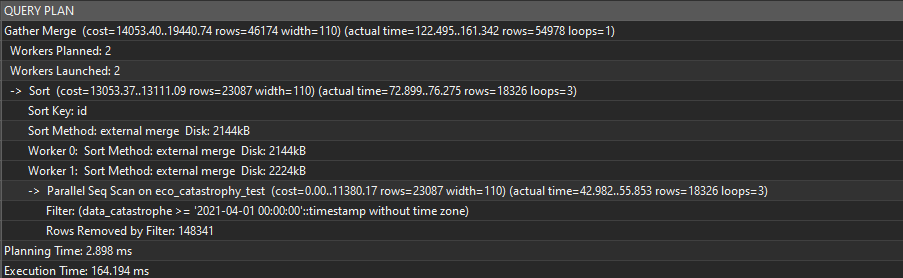




EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy\_test

WHERE data\_catastrophe>='2021-04-01' ORDER BY "id";



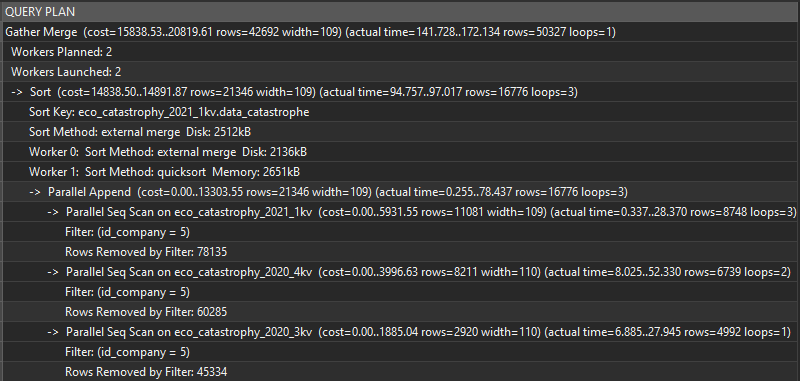
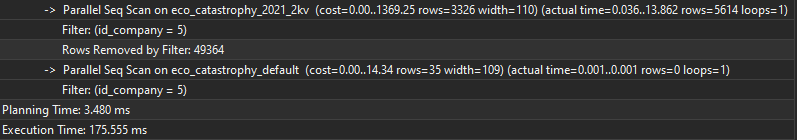
Из результатов видно, что первая партиция (*eco\_catastrophy*) разбитая по диапазонам дат справилась лучше всех – по затраченному времени (Execution Time) на запрос – 36.036 ms, оценка затратности операции (cost) – 7756.52. Это очевидно потом-что условие поиска и сортировка происходила по ключу разбиения партиции на секции, что привело к тому что поиск нужных записей проводился в 2-х секциях – *eco\_catastrophy\_20201\_2kv* (это секция с данными за период от 01.04.2021 до 01.07.2021, как раз попадающая под условие поиска *data\_catastrophe>='2021-04-01'*) и *eco\_catastrophy\_default* (это секция где хранятся данные которые не попали в остальные секции по разным причинам, но на данный момент она пуста). Также стоит отметить, что, последовательное сканирование (Seq Scan) сменилось на Index Scan — используется индекс *eco\_catastrophy\_20201\_2kv\_data\_catastrophe\_idx* (для секции *eco\_catastrophy\_20201\_2kv* и *eco\_catastrophy\_default\_data\_catastrophe\_idx* для секции *eco\_catastrophy\_default*) для условий *WHERE* и читает таблицу при отборе строк.

Вторая (*eco\_catastrophy\_list*) и третья (*eco\_catastrophy\_hash*) партиции справилась хуже, им пришлось сканировать все секции в поисках нужных записей, т.к. ключом разбиение у второй является поле *id\_company*, а у третей поле *id*, а не поле дат (data\_catastrophe) как у первой партиции. Но стоит отметить, что, у второй партиции (*eco\_catastrophy\_list*) подсекция *eco\_catastrophy\_list\_2\_4\_6\_8\_2020* не была задействована в сканировании. Напомню, что мы секцию *eco\_catastrophy\_list\_2\_4\_6\_8* разбили на 2 подсекции – *eco\_catastrophy\_list\_2\_4\_6\_8\_2020* (подсекция с данными у которых поле *data\_catastrophe* лежит в пределах 2020 года) и *eco\_catastrophy\_list\_2\_4\_6\_8\_2021* (подсекция с данными у которых поле *data\_catastrophe* лежит в пределах 2021 года). Поэтому т.к. у нас условие найти данные, у которых *data\_catastrophe>='2021-04-01'*, секция *eco\_catastrophy\_list\_2\_4\_6\_8\_2020* с данными за 2020 год отбрасывается. Потом партиции сортируются: вторая (*eco\_catastrophy\_list*) по *id\_company*, третья (*eco\_catastrophy\_hash*) по *id*. Используется последовательное сканирование секций, но в параллельном режиме (Parallel Seq Scan).

Обычная таблица справилась чуть лучше. По затраченному времени (Execution Time) на запрос показатель равен 164.194 ms, что наравне со второй партицией (*eco\_catastrophy\_list*) – 164.815 ms. Оценка затратности операции (cost) – 19440.74, что хуже, чем у первой (7756.52) и второй (18859.93) партиций, но лучше, чем у третей (20514.69) партиции.

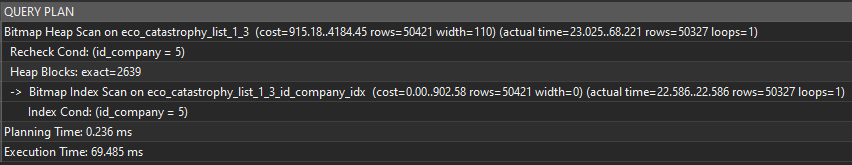
EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy WHERE id\_company=5 ORDER BY data\_catastrophe;

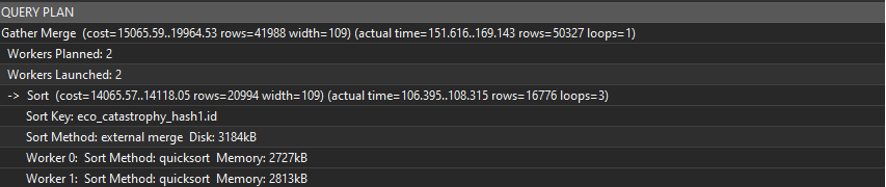
EXPLAIN (ANALYZE)

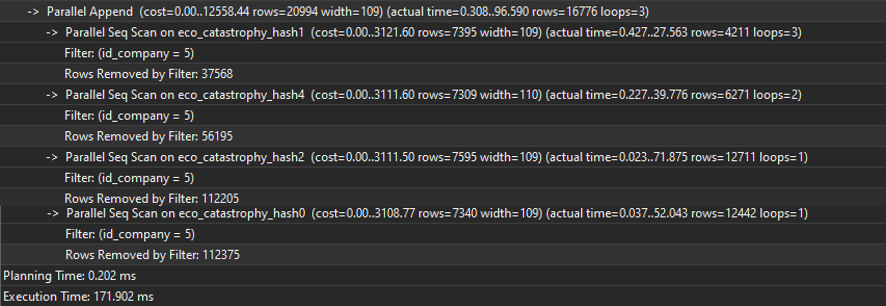
SELECT \* FROM eco\_catastrophy\_list WHERE id\_company=5 ORDER BY id\_company;



EXPLAIN (ANALYZE)

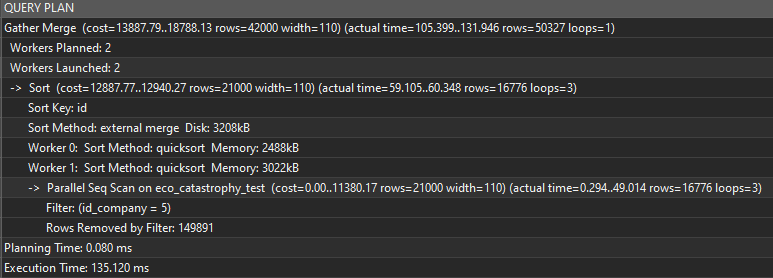
SELECT \* FROM eco\_catastrophy\_hash WHERE id\_company=5 ORDER BY "id";





EXPLAIN (ANALYZE)

SELECT \* FROM eco\_catastrophy\_test WHERE id\_company=5 ORDER BY "id";



Поменяв условие поиска записей в *WHERE* (теперь ищем все записи у которых поле *id\_company* равно пяти), видно что вторая партиция (*eco\_catastrophy\_list*) разбитая по списку справилось лучше всех, по затраченному времени (Execution Time) на запрос – 69.485 ms, оценка затратности операции (cost) – 4184.45. Это потому, что условие поиска и сортировка происходила по ключу разбиения её секций. Поэтому поиск происходит только в одной секции – *eco\_catastrophy\_list\_1\_3*, где хранятся записи с *id\_company* равным 1,3 и 5 что удовлетворяет нашему условию *WHERE id\_company=5*. Также стоит отметить, что, последовательное сканирование (Seq Scan) сменилось на Bitmap Index Scan — используется индекс *eco\_catastrophy\_list\_1\_3\_id\_company\_idx* для определения нужных нам записей, а затем PostgreSQL лезет в саму таблицу (Bitmap Heap Scan), чтобы убедиться, что эти записи на самом деле существуют.

Первая партиция (eco\_*catastrophy*) которую разбивали по диапазонам дат и третья партиция (*eco\_catastrophy\_hash*) разбитая по хешу, справилась хуже, т.к. поиск проводился по всем секциям партиций. Результаты первой партиции (*eco\_catastrophy*) - по затраченному времени (Execution Time) на запрос – 175.555 ms, оценка затратности операции (cost) – 20819.61. Результаты третьей партиции (*eco\_catastrophy\_hash*) - по затраченному времени (Execution Time) на запрос – 171.902 ms, оценка затратности операции (cost) – 19964.53. У обеих партиций используется последовательное сканирование секций, но в параллельном режиме (Parallel Seq Scan).

Показатели обычной таблицы по затраченному времени (Execution Time) на запрос – 135.120 ms, оценка затратности операции (cost) – 18788.13.

Добавим еще индексов (по ключевым полям) к партициям и к обычной таблице, и проверим на других запросах, как будет происходит их построение и какие из индексов будут участвовать в построении запроса, а какие нет.

CREATE INDEX ON eco\_catastrophy ("id\_company");

CREATE INDEX ON eco\_catastrophy ("id");

CREATE INDEX ON eco\_catastrophy\_list ("data\_catastrophe");

CREATE INDEX ON eco\_catastrophy\_list ("id");

CREATE INDEX ON eco\_catastrophy\_hash ("id\_company");

CREATE INDEX ON eco\_catastrophy\_hash ("data\_catastrophe");

CREATE INDEX ON eco\_catastrophy\_test ("id\_company");

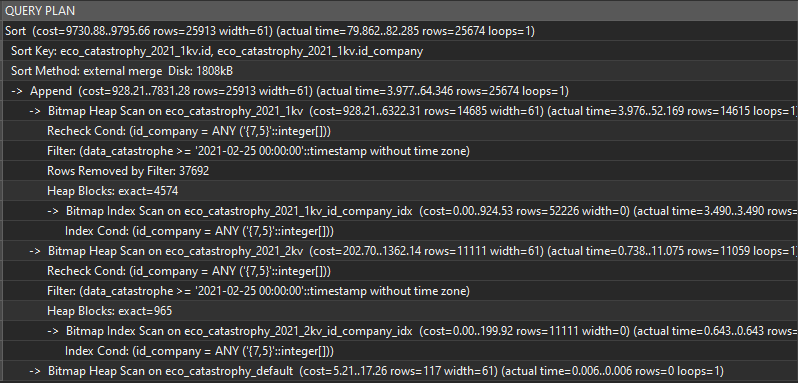
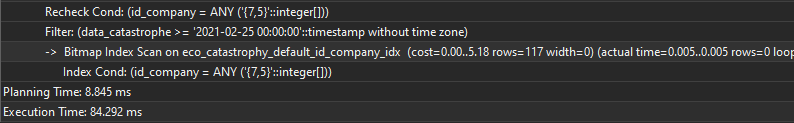
CREATE INDEX ON eco\_catastrophy\_test ("data\_catastrophe");

EXPLAIN (ANALYZE) SELECT

"id", id\_company, data\_catastrophe::date, number\_victims, effects, detriment

FROM eco\_catastrophy WHERE data\_catastrophe>='2021-02-25' AND

id\_company IN(7,5) ORDER BY "id", id\_company;

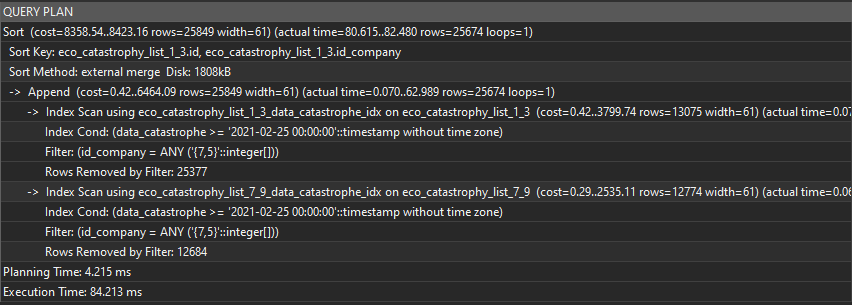
 

EXPLAIN (ANALYZE)SELECT

"id", id\_company, data\_catastrophe::date, number\_victims, effects, detriment

FROM eco\_catastrophy\_list WHERE data\_catastrophe>='2021-02-25' AND

id\_company IN(7,5) ORDER BY "id", id\_company;

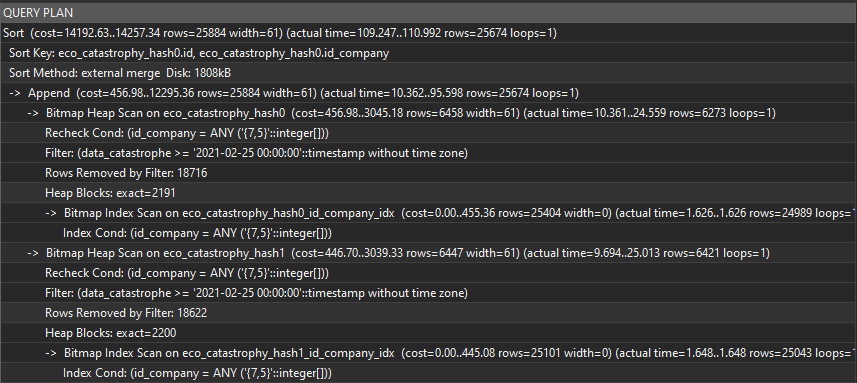


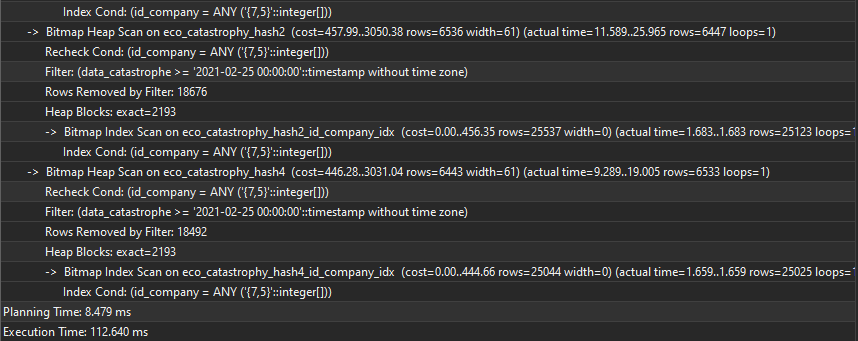
EXPLAIN (ANALYZE) SELECT

"id", id\_company, data\_catastrophe::date, number\_victims, effects, detriment

FROM eco\_catastrophy\_hash WHERE data\_catastrophe>='2021-02-25' AND

id\_company IN(7,5) ORDER BY "id", id\_company;



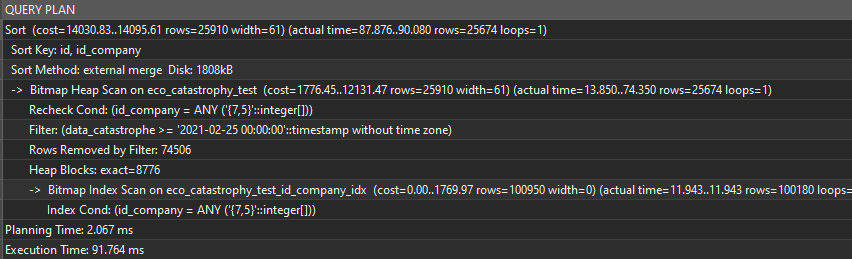


EXPLAIN (ANALYZE) SELECT

"id", id\_company, data\_catastrophe::date, number\_victims, effects, detriment

FROM eco\_catastrophy\_test WHERE data\_catastrophe>='2021-02-25' AND

id\_company IN(7,5) ORDER BY "id", id\_company;



Из полученных результатов видим, что первая партиция (разбитая по диапазону) и вторая партиция (разбитая по спискам) справилась лучше всех.

Результаты первой партиции (eco\_*catastrophy*) - по затраченному времени (Execution Time) на запрос – 84.292 ms, оценка затратности операции (cost) – 9795.66. Результаты второй партиции (*eco\_catastrophy\_list*) - по затраченному времени (Execution Time) на запрос – 84.213 ms, оценка затратности операции (cost) – 8423.16. В первой партиции поиск записей происходит с помощью Bitmap Index Scan — который использует индексы: *eco\_catastrophy\_2021\_1kv\_id\_company\_idx* для секции *eco\_catastrophy\_2021\_1kv*, *eco\_catastrophy\_2021\_2kv\_id\_company\_idx* для секции *eco\_catastrophy\_2021\_2kv* и *eco\_catastrophy\_default\_id\_company\_idx* для секции *eco\_catastrophy\_default*, чтобы определить нужные нам записи, а затем PostgreSQL лезет в саму таблицу: (Bitmap Heap Scan), чтобы убедиться, что эти записи на самом деле существуют. Следует отметить, что это новый индекс, который мы создали для этой партиции по полю *id\_company*. Во второй партиции поиск записей происходит с помощью Index Scan — который использует индексы: *eco\_catastrophy\_list\_1\_3\_data\_catastrophe\_idx* для секции *eco\_catastrophy\_list\_1\_3*, *eco\_catastrophy\_list\_7\_9\_data\_catastrophe\_idx* для секции *eco\_catastrophy\_list\_7\_9*, чтобы определить нужные нам записи при условий *WHERE*, читает таблицу при отборе строк. Следует отметить, что это новый индекс, который мы создали для этой партиции по полю *data\_catastrophe*.

Третья партиция (*eco\_catastrophy\_hash*) справилась хуже всех. Результаты: по затраченному времени (Execution Time) на запрос – 112.640 ms, оценка затратности операции (cost) – 14257.34. Поиск записей происходит по всем секциям с помощью Bitmap Index Scan — который использует новый индекс, который мы создали для этой партиции по полю *id\_company*.

Результаты таблицы *eco\_catastrophy\_test*: по затраченному времени (Execution Time) на запрос – 91.764 ms, оценка затратности операции (cost) – 14095.61. Обычная таблица по затраченному времени и по оценке затратности операций уступает первой и второй партициям, но лучше чем партиция разбитая по хешу но не на много. Также стоит отметить, что по сравнению с предыдущими запросами, последовательное сканирование (Seq Scan) или последовательное сканирование в параллельном режиме (Parallel Seq Scan) сменилось на Bitmap Index Scan — используется новый индекс *eco\_catastrophy\_test\_id\_company\_idx* для определения нужных нам записей, а затем PostgreSQL лезет в саму таблицу (Bitmap Heap Scan), чтобы убедиться, что эти записи на самом деле существуют.

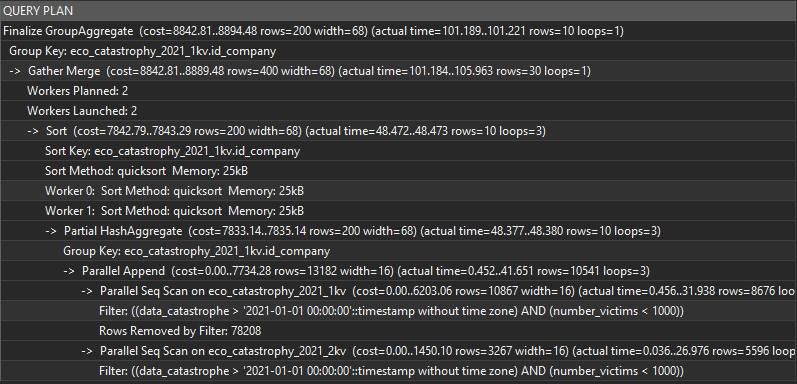
EXPLAIN (ANALYZE)

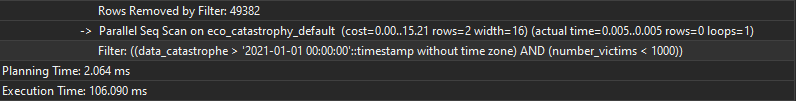
SELECT id\_company, max(detriment), min(detriment)

FROM eco\_catastrophy WHERE data\_catastrophe>'2021-01-01'

AND number\_victims<1000

GROUP BY id\_company ORDER BY id\_company;





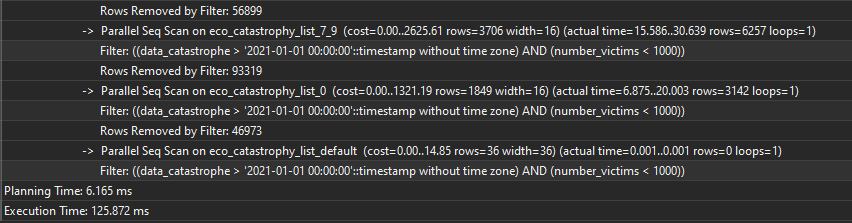
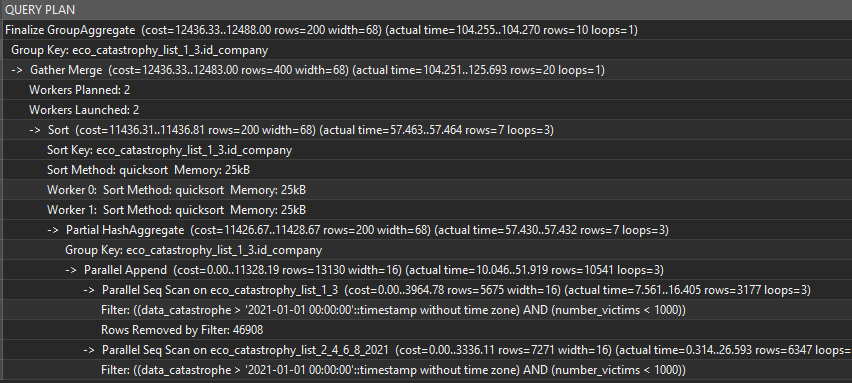
EXPLAIN (ANALYZE)

SELECT id\_company, max(detriment), min(detriment)

FROM eco\_catastrophy\_list WHERE data\_catastrophe>'2021-01-01'

AND number\_victims<1000

GROUP BY id\_company ORDER BY id\_company;



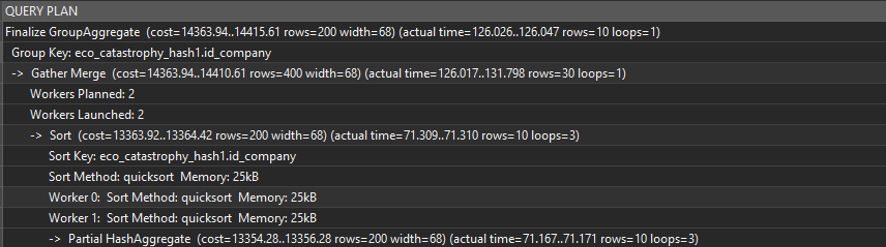
EXPLAIN (ANALYZE)

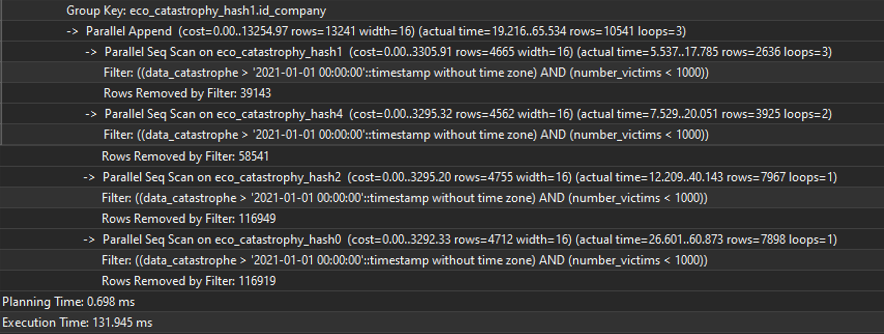
SELECT id\_company, max(detriment), min(detriment)

FROM eco\_catastrophy\_hash WHERE data\_catastrophe>'2021-01-01'

AND number\_victims<1000

GROUP BY id\_company ORDER BY id\_company;





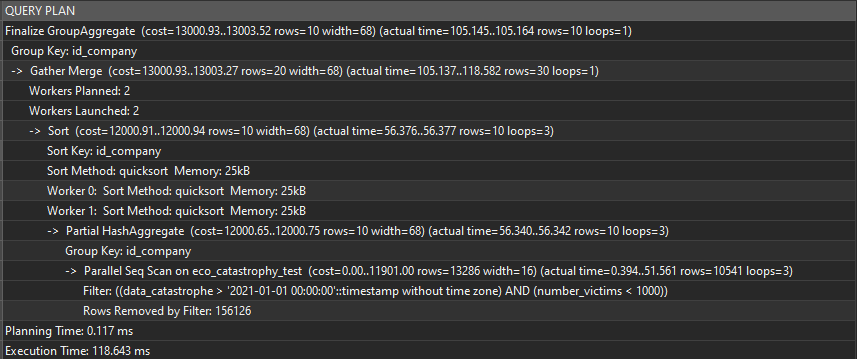
EXPLAIN (ANALYZE)

SELECT id\_company, max(detriment), min(detriment)

FROM eco\_catastrophy\_test WHERE data\_catastrophe>'2021-01-01'

AND number\_victims<1000

GROUP BY id\_company ORDER BY id\_company;



Из полученных результатов видим что первая партиция (разбитая по диапазонам) производит поиск только в секциях где хранятся данные за 2021 год – это секция *eco\_catastrophy\_2021\_1kv и секция eco\_catastrophy\_2021\_2kv*, а также секция отвечающая за данные которые не попали ни в одну из секций – это секция *eco\_catastrophy\_default*. Результаты первой партиции (*eco\_catastrophy*) - по затраченному времени (Execution Time) на запрос – 106.090 ms, оценка затратности операции (cost) – 8894.48. Следует отметить, что используется последовательное сканирование секций в параллельном режиме (Parallel Seq Scan).

Вторая партиция (разбитая по спискам) производит последовательное сканирование секций в параллельном режиме (Parallel Seq Scan) по всем секциям кроме одной – *eco\_catastrophy\_list\_2\_4\_6\_8\_2020*. Напомню, что мы секцию *eco\_catastrophy\_list\_2\_4\_6\_8* разбили на 2 подсекции – *eco\_catastrophy\_list\_2\_4\_6\_8\_2020* (подсекция с данными у которых поле *data\_catastrophe* лежит в пределах 2020 года) и *eco\_catastrophy\_list\_2\_4\_6\_8\_2021* (подсекция с данными у которых поле *data\_catastrophe* лежит в пределах 2021 года). Поэтому т.к. у нас условие найти данные, у которых *data\_catastrophe>'2021-01-01'*, секция *eco\_catastrophy\_list\_2\_4\_6\_8\_2020* с данными за 2020 год отбрасывается, что нам на пользу. Результаты: по затраченному времени (Execution Time) на запрос – 125.872 ms, оценка затратности операции (cost) – 12488.00.

Третья партиция также производит последовательное сканирование секций в параллельном режиме (Parallel Seq Scan) по всем секциям. Результаты: по затраченному времени (Execution Time) на запрос – 131.945 ms, оценка затратности операции (cost) – 14415.61.

Результаты *eco\_catastrophy\_test*: по затраченному времени (Execution Time) на запрос – 118.643 ms, оценка затратности операции (cost) – 13003.52. Обычная таблица по затраченному времени и оценке затратности операций опять уступает партициям (первой (*eco\_catastrophy*) – разбитая по диапазонам дат и второй (*eco\_catastrophy\_list*) – разбитая по спискам), но чуть лучше показывает результат по сравнению с третей партицией. Из этого следует что таблицы, которые хранят большое количество данных лучше преобразовывать в партиции.