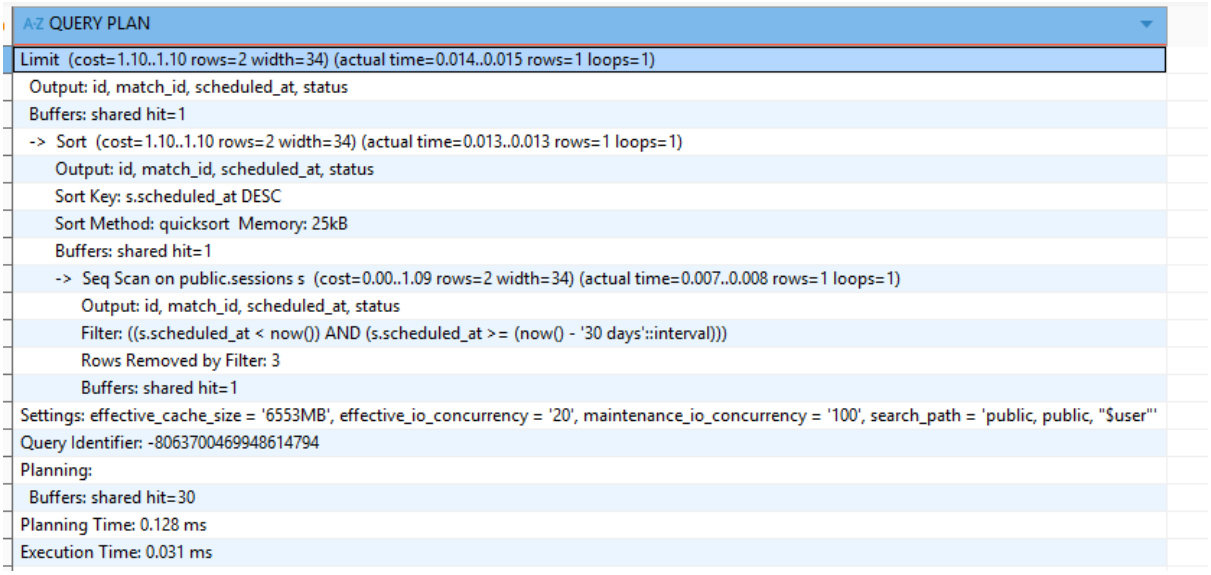
Задание 1

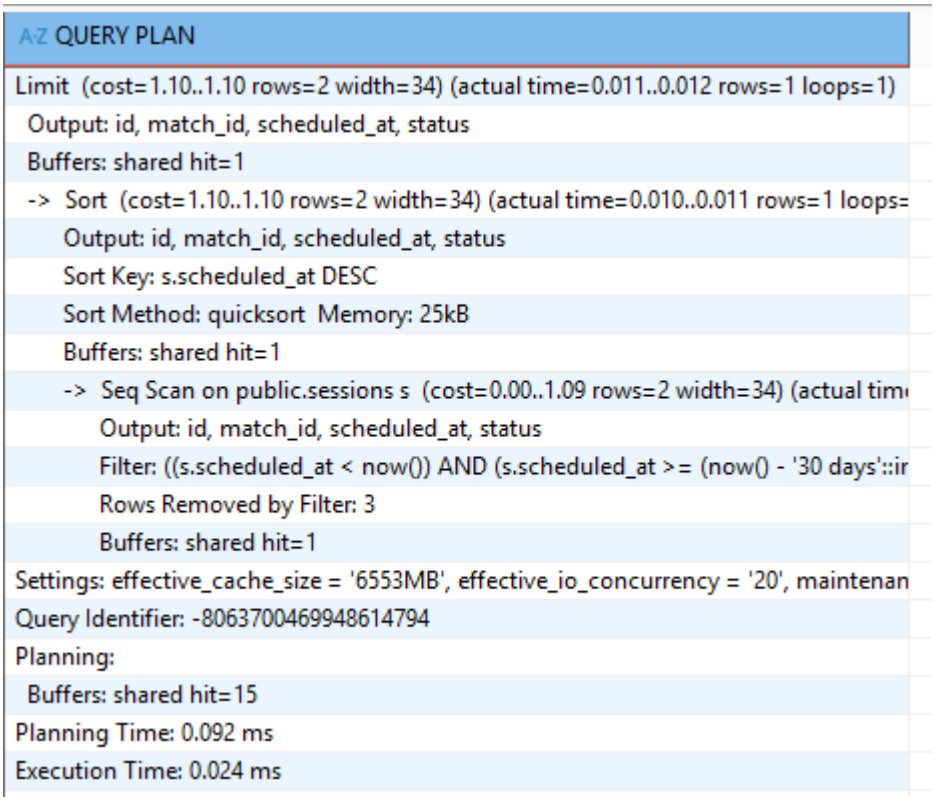
1) Запрос на сессии за последние 30 дней

До индексирования:



QUERY PLAN
Limit (cost=1.10..1.10 rows=2 width=34) (actual time=0.014..0.015 rows=1 loops=1)
Output: id, match_id, scheduled_at, status
Buffers: shared hit=1
-> Sort (cost=1.10..1.10 rows=2 width=34) (actual time=0.013..0.013 rows=1 loops=1)
Output: id, match_id, scheduled_at, status
Sort Key: s.scheduled_at DESC
Sort Method: quicksort Memory: 25kB
Buffers: shared hit=1
-> Seq Scan on public.sessions s (cost=0.00..1.09 rows=2 width=34) (actual time=0.007..0.008 rows=1 loops=1)
Output: id, match_id, scheduled_at, status
Filter: ((s.scheduled_at < now()) AND (s.scheduled_at >= (now() - '30 days'::interval)))
Rows Removed by Filter: 3
Buffers: shared hit=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenance_io_concurrency = '100', search_path = 'public, public, "$user"'
Query Identifier: -8063700469948614794
Planning:
Buffers: shared hit=30
Planning Time: 0.128 ms
Execution Time: 0.031 ms

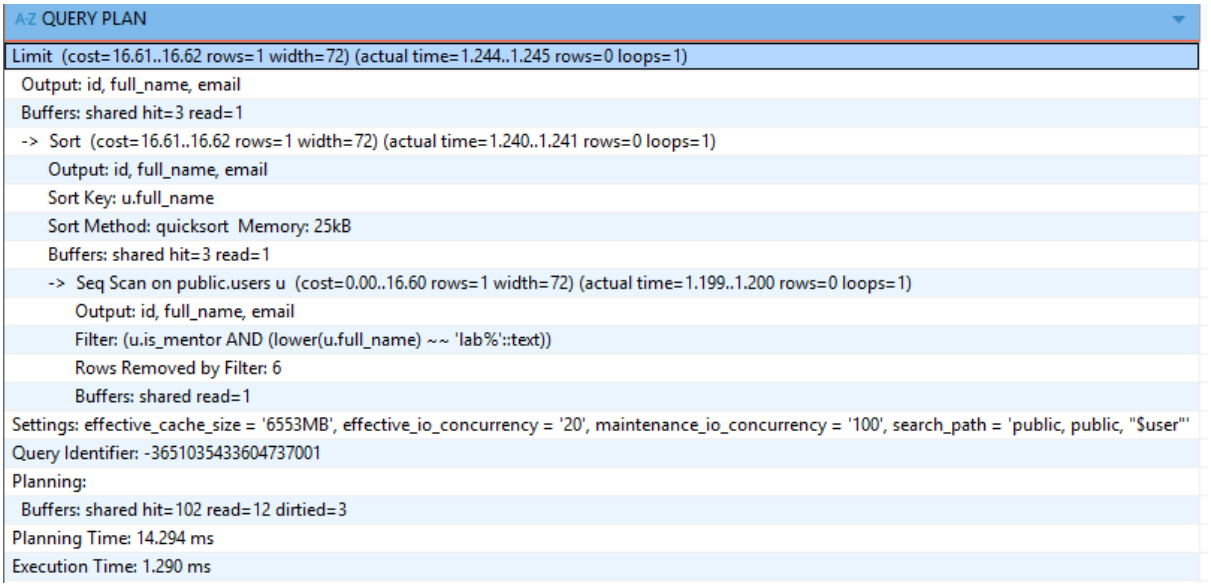
BRIN индексирование (т к запрос по промежутку времени, предполагаются большие объемы упорядоченных данных):



QUERY PLAN
Limit (cost=1.10..1.10 rows=2 width=34) (actual time=0.011..0.012 rows=1 loops=1)
Output: id, match_id, scheduled_at, status
Buffers: shared hit=1
-> Sort (cost=1.10..1.10 rows=2 width=34) (actual time=0.010..0.011 rows=1 loops=
Output: id, match_id, scheduled_at, status
Sort Key: s.scheduled_at DESC
Sort Method: quicksort Memory: 25kB
Buffers: shared hit=1
-> Seq Scan on public.sessions s (cost=0.00..1.09 rows=2 width=34) (actual tim
Output: id, match_id, scheduled_at, status
Filter: ((s.scheduled_at < now()) AND (s.scheduled_at >= (now() - '30 days'::ir
Rows Removed by Filter: 3
Buffers: shared hit=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenan
Query Identifier: -8063700469948614794
Planning:
Buffers: shared hit=15
Planning Time: 0.092 ms
Execution Time: 0.024 ms

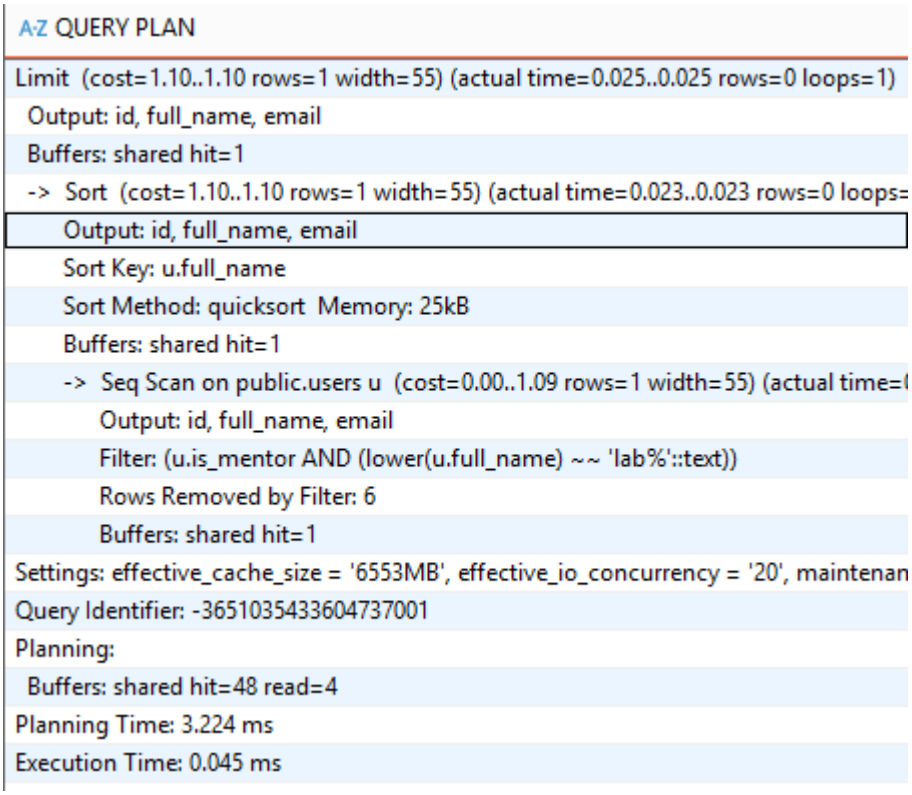
2) Извлекает из таблицы users всех менторов, у которых имя начинается с lab и сортирует по возрастанию

До индексирования:



QUERY PLAN
Limit (cost=16.61..16.62 rows=1 width=72) (actual time=1.244..1.245 rows=0 loops=1)
Output: id, full_name, email
Buffers: shared hit=3 read=1
-> Sort (cost=16.61..16.62 rows=1 width=72) (actual time=1.240..1.241 rows=0 loops=1)
Output: id, full_name, email
Sort Key: u.full_name
Sort Method: quicksort Memory: 25kB
Buffers: shared hit=3 read=1
-> Seq Scan on public.users u (cost=0.00..16.60 rows=1 width=72) (actual time=1.199..1.200 rows=0 loops=1)
Output: id, full_name, email
Filter: (u.is_mentor AND (lower(u.full_name) ~~ 'lab%'::text))
Rows Removed by Filter: 6
Buffers: shared read=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenance_io_concurrency = '100', search_path = 'public, public, "$user"'
Query Identifier: -3651035433604737001
Planning:
Buffers: shared hit=102 read=12 dirtied=3
Planning Time: 14.294 ms
Execution Time: 1.290 ms

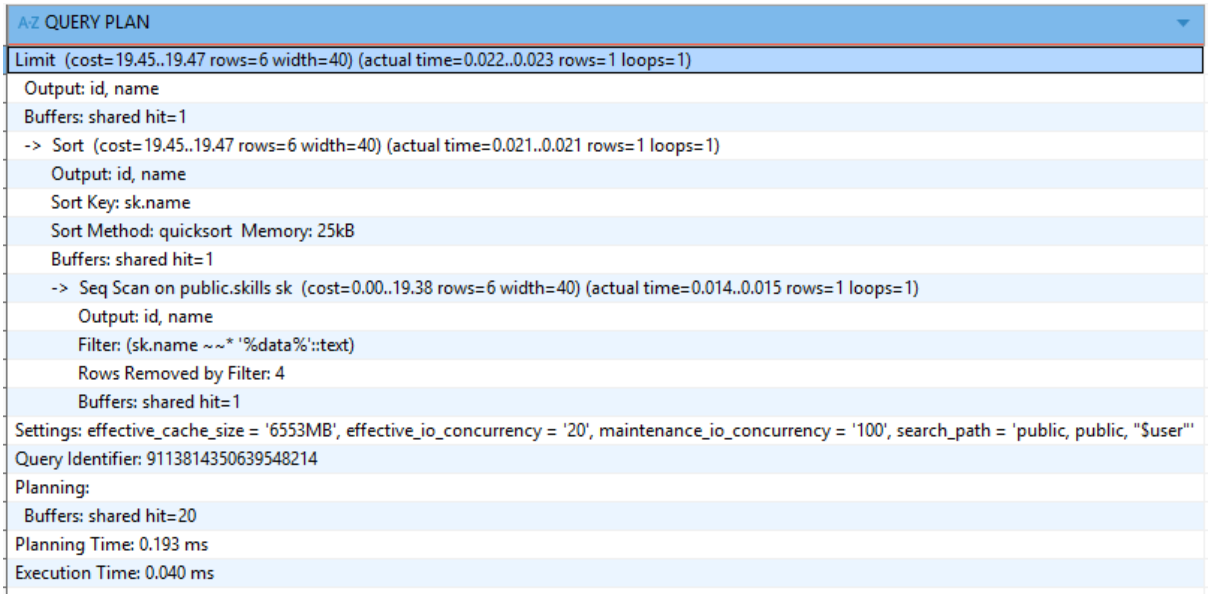
B-tree (т е стандартное) индексирование функции с надстройкой под текстовый поиск:



QUERY PLAN
Limit (cost=1.10..1.10 rows=1 width=55) (actual time=0.025..0.025 rows=0 loops=1)
Output: id, full_name, email
Buffers: shared hit=1
-> Sort (cost=1.10..1.10 rows=1 width=55) (actual time=0.023..0.023 rows=0 loops=
Output: id, full_name, email
Sort Key: u.full_name
Sort Method: quicksort Memory: 25kB
Buffers: shared hit=1
-> Seq Scan on public.users u (cost=0.00..1.09 rows=1 width=55) (actual time=
Output: id, full_name, email
Filter: (u.is_mentor AND (lower(u.full_name) ~~ 'lab%'::text))
Rows Removed by Filter: 6
Buffers: shared hit=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenan
Query Identifier: -3651035433604737001
Planning:
Buffers: shared hit=48 read=4
Planning Time: 3.224 ms
Execution Time: 0.045 ms

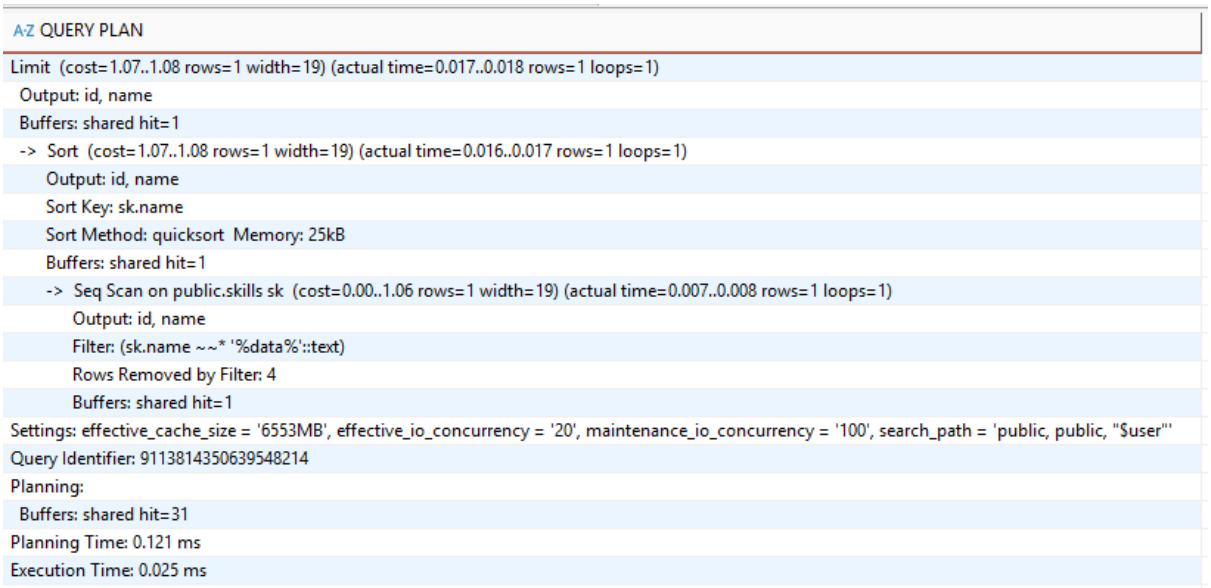
3) Из таблицы skills извлекает все скилы, где встречается data в любом регистре

До индексирования:



```
QUERY PLAN
Limit  (cost=19.45..19.47 rows=6 width=40) (actual time=0.022..0.023 rows=1 loops=1)
  Output: id, name
  Buffers: shared hit=1
  ->  Sort  (cost=19.45..19.47 rows=6 width=40) (actual time=0.021..0.021 rows=1 loops=1)
        Output: id, name
        Sort Key: sk.name
        Sort Method: quicksort  Memory: 25kB
        Buffers: shared hit=1
        ->  Seq Scan on public.skills sk  (cost=0.00..19.38 rows=6 width=40) (actual time=0.014..0.015 rows=1 loops=1)
              Output: id, name
              Filter: (sk.name ~~* '%data%'::text)
              Rows Removed by Filter: 4
              Buffers: shared hit=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenance_io_concurrency = '100', search_path = 'public, public, "$user"'
Query Identifier: 9113814350639548214
Planning:
  Buffers: shared hit=20
Planning Time: 0.193 ms
Execution Time: 0.040 ms
```

GIN индексирование под LIKE:

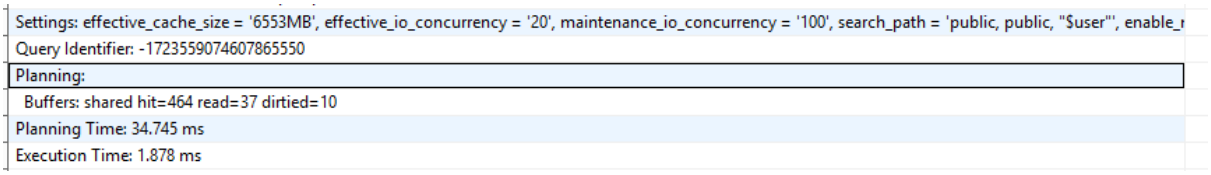


```
QUERY PLAN
Limit  (cost=1.07..1.08 rows=1 width=19) (actual time=0.017..0.018 rows=1 loops=1)
  Output: id, name
  Buffers: shared hit=1
  ->  Sort  (cost=1.07..1.08 rows=1 width=19) (actual time=0.016..0.017 rows=1 loops=1)
        Output: id, name
        Sort Key: sk.name
        Sort Method: quicksort  Memory: 25kB
        Buffers: shared hit=1
        ->  Seq Scan on public.skills sk  (cost=0.00..1.06 rows=1 width=19) (actual time=0.007..0.008 rows=1 loops=1)
              Output: id, name
              Filter: (sk.name ~~* '%data%'::text)
              Rows Removed by Filter: 4
              Buffers: shared hit=1
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenance_io_concurrency = '100', search_path = 'public, public, "$user"'
Query Identifier: 9113814350639548214
Planning:
  Buffers: shared hit=31
Planning Time: 0.121 ms
Execution Time: 0.025 ms
```

Вывод: все три вида индексов юзабельны и значительно ускоряют исполнение запросов

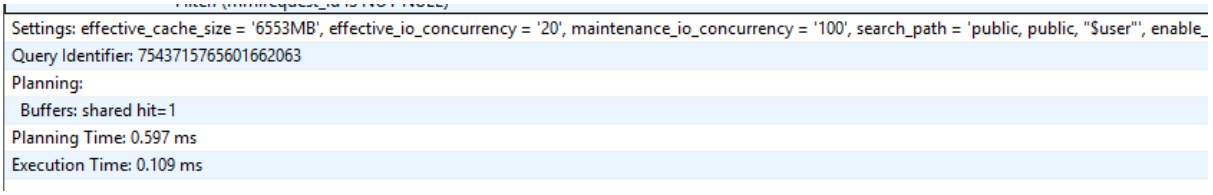
Задание 2

1) топ менторов по скилу с рейтингами (hash join only)

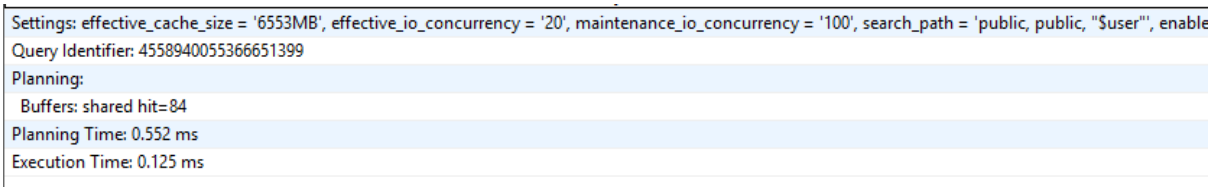


```
Settings: effective_cache_size = '6553MB', effective_io_concurrency = '20', maintenance_io_concurrency = '100', search_path = 'public, public, "$user"', enable_
Query Identifier: -1723559074607865550
Planning:
  Buffers: shared hit=464 read=37 dirtied=10
Planning Time: 34.745 ms
Execution Time: 1.878 ms
```

2) статистика по матчам за последние 30 дней (hash join only)

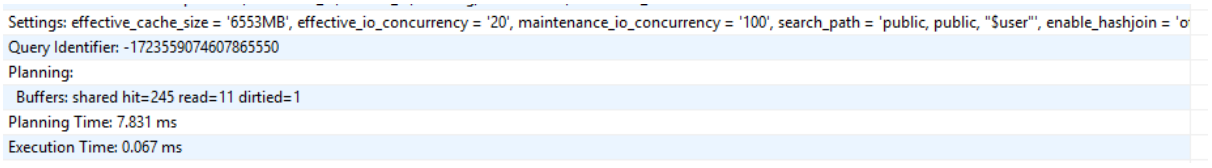


3) статистика спроса/предложения + ранжировка навыков (hash join only)

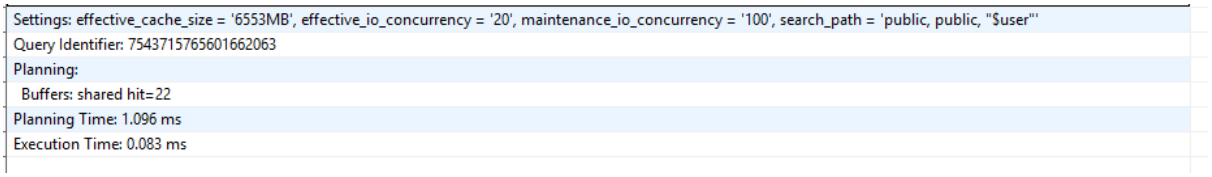


После индексирования + отключение hash join и включение merge join и nested loop

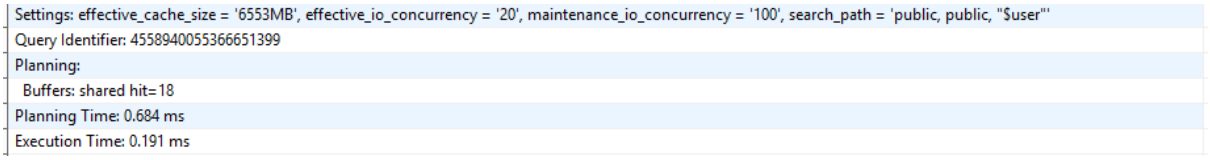
1)



2)



3)



Вывод: индексы значительно ускоряют доступ к данным, hash join самый эффективный как до, так и после индексирования в большинстве случаев, в большинстве случаев планировщик справится с выбором правильных тактик намного лучше человека.

Задание 3

1) Dirty read невозможен принципиально, но работает он так: если одна транзакция читает данные, измененные другой транзакцией, но незакоммиченные, но вдруг вторая транзакция откатывается, то первая транзакция прочитала по сути неверные данные. В postgres уровень изоляции uncommitted такое не позволяет принципиально, так что невозможно выполнить напрямую по определению, но реально воспроизвести похожий эффект через костыли.
2) Non-repeatable read

Что за аномалия: функция в рамках одной транзакции вернула разные значения.
Как воспроизвести: создается две сессии, в одной функция вызывается в первый раз, затем слип, затем во второй раз. После выполнения функции в первый раз в другой сессии вызывается коммит нового значения. Таким образом в первой сессии в выводе будут два значения, до изменений и после, и все это в рамках одной транзакции.
Пример результата:
1-й вызов

```
SET search_path = lab3, public;

UPDATE tx_demo SET val = 100 WHERE id = 1;

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT now() AS t1_time_1, val AS t1_first_read
FROM tx_demo
WHERE id = 1;

SELECT pg_sleep(8);

SELECT now() AS t1_time_2, val AS t1_second_read
FROM tx_demo
WHERE id = 1;

COMMIT;
```

Connected (8 queries)

Run | Explain | Analyze

1: SET | 2: UPDATE 1 | 3: BEGIN | 4: SET | 5: SELECT 1 | 6: SELECT 1 | 7: SELECT 1 | 8: COMMIT

#	t1_time_1	t1_first_read
1	2025-12-19 20:07:32.949034+00	100

2-й вызов

```
SET search_path = lab3, public;

UPDATE tx_demo SET val = 100 WHERE id = 1;

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT now() AS t1_time_1, val AS t1_first_read
FROM tx_demo
WHERE id = 1;

SELECT pg_sleep(8);

SELECT now() AS t1_time_2, val AS t1_second_read
FROM tx_demo
WHERE id = 1;

COMMIT;
```

Connected (8 queries)

Run | Explain | Analyze

1: SET | 2: UPDATE 1 | 3: BEGIN | 4: SET | 5: SELECT 1 | 6: SELECT 1 | 7: SELECT 1 | 8: COMMIT

#	t1_time_2	t1_second_read
1	2025-12-19 20:07:32.949034+00	110

Как исправляется: добавлением уровня изоляции REPEATABLE READ. По сути это означает, что все запросы в рамках одной транзакции используют одну и ту же версию данных, куда новые коммиты не входят.
Результат после изменений:
1-й

```
SET search_path = lab3, public;

UPDATE tx_demo SET val = 100 WHERE id = 1;

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT now() AS t1_time_1, val AS t1_first_read
FROM tx_demo
WHERE id = 1;

SELECT pg_sleep(8);

SELECT now() AS t1_time_2, val AS t1_second_read
FROM tx_demo
WHERE id = 1;

COMMIT;
```

Connected (8 queries)

Run | Explain | Analyze

1: SET | 2: UPDATE 1 | 3: BEGIN | 4: SET | 5: SELECT 1 | 6: SELECT 1 | 7: SELECT 1 | 8: COMMIT

#	t1_time_1	t1_first_read
1	2025-12-19 20:19:03.552281+00	100

2-й

```
SET search_path = lab3, public;

UPDATE tx_demo SET val = 100 WHERE id = 1;

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT now() AS t1_time_1, val AS t1_first_read
FROM tx_demo
WHERE id = 1;

SELECT pg_sleep(8);

SELECT now() AS t1_time_2, val AS t1_second_read
FROM tx_demo
WHERE id = 1;

COMMIT;
```

Connected (8 queries)

Run | Explain | Analyze

1: SET | 2: UPDATE 1 | 3: BEGIN | 4: SET | 5: SELECT 1 | 6: SELECT 1 | 7: SELECT 1 | 8: COMMIT

#	t1_time_2	t1_second_read
1	2025-12-19 20:19:03.552281+00	100

3) Phantom read. По сути аномалия аналогична предыдущей, но не значение изменяется, а добавляются новые эл-ты, удовлетворяющие условию.

1-й

```
SET search_path = lab3, public;

TRUNCATE tx_phantom;
INSERT INTO tx_phantom(grp)
SELECT 'open' FROM generate_series(1, 5);

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT count(*) AS t1_first_count
FROM tx_phantom
WHERE grp = 'open';

SELECT pg_sleep(16);

SELECT count(*) AS t1_second_count
FROM tx_phantom
WHERE grp = 'open';

COMMIT;
```

Connected (9 queries)

Run | Explain | Analyze

1: SET | 2: TRUNCATE | 3: INSERT 5 | 4: BEGIN | 5: SET | 6: SELECT 1 | 7: SELECT 1 | 8: SELECT 1 | 9: COMMIT

#	t1_first_count
1	5

2-й

```sql
SET search_path = lab3, public;

TRUNCATE tx_phantom;
INSERT INTO tx_phantom(grp)
SELECT 'open' FROM generate_series(1, 5);

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT count(*) AS t1_first_count
FROM tx_phantom
WHERE grp = 'open';

SELECT pg_sleep(16);

SELECT count(*) AS t1_second_count
FROM tx_phantom
WHERE grp = 'open';

COMMIT;
```

Connected (9 queries)

Run | Explain | Analyze

1: SET | 2: TRUNCATE | 3: INSERT 5 | 4: BEGIN | 5: SET | 6: SELECT 1 | 7: SELECT 1 | 8: SELECT 1 | 9: COMMIT

#	t1_second_count
1	8

Как исправляется: добавлением уровня изоляции REPEATABLE READ, но можно например использовать и SERIALIZABLE, но интерпретатор может поругаться.
Результат:
1-й

```sql
SET search_path = lab3, public;

TRUNCATE tx_phantom;
INSERT INTO tx_phantom(grp)
SELECT 'open' FROM generate_series(1, 5);

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT count(*) AS t1_first_count
FROM tx_phantom
WHERE grp = 'open';

SELECT pg_sleep(16);

SELECT count(*) AS t1_second_count
FROM tx_phantom
WHERE grp = 'open';

COMMIT;
```

Connected (9 queries)

Run | Explain | Analyze

1: SET | 2: TRUNCATE | 3: INSERT 5 | 4: BEGIN | 5: SET | 6: SELECT 1 | 7: SELECT 1 | 8: SELECT 1 | 9: COMMIT

#	t1_first_count
1	5

2-й

```
SET search_path = lab3, public;

TRUNCATE tx_phantom;
INSERT INTO tx_phantom(grp)
SELECT 'open' FROM generate_series(1, 5);

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT count(*) AS t1_first_count
FROM tx_phantom
WHERE grp = 'open';

SELECT pg_sleep(16);

SELECT count(*) AS t1_second_count
FROM tx_phantom
WHERE grp = 'open';

COMMIT;
```

Connected (9 queries)

Run | Explain | Analyze

1: SET | 2: TRUNCATE | 3: INSERT 5 | 4: BEGIN | 5: SET | 6: SELECT 1 | 7: SELECT 1 | 8: SELECT 1 | 9: COMMIT

#	t1_second_count
1	5

4) Lost update

Суть заключается в том,что две транзакции параллельно читают, считают и записывают на основе пересекающихся значений, и одна транзакция может перетереть другую (то есть обе использовали старые значения и обе посчитали в итоге неправильно)

1-й

```
SET search_path = lab3, public;

UPDATE tx_demo SET val = 100 WHERE id = 1;

BEGIN;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

CREATE TEMP TABLE t1_read(val int) ON COMMIT DROP;
INSERT INTO t1_read(val)
SELECT val FROM tx_demo WHERE id = 1;

SELECT pg_sleep(12);

UPDATE tx_demo
SET val = (SELECT val FROM t1_read) + 10
WHERE id = 1;

COMMIT;

SELECT val AS final_after_t1 FROM tx_demo WHERE id = 1;
```

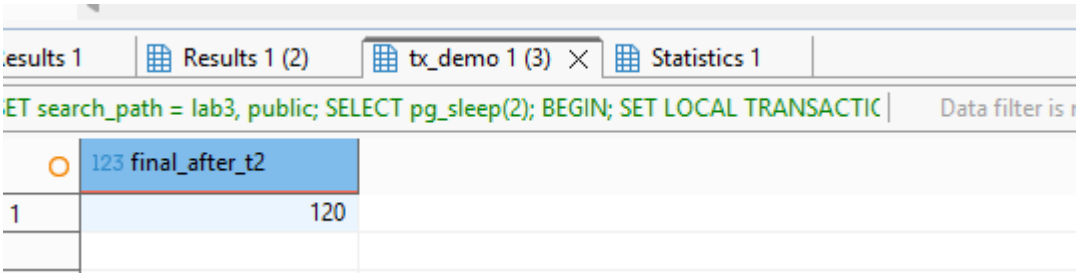
Connected (10 queries)

Run | Explain | Analyze

1: SET | 2: UPDATE 1 | 3: BEGIN | 4: SET | 5: CREATE | 6: INSERT 1 | 7: SELECT 1 | 8: UPDATE 1 | 9: COMMIT | 10: SELECT 1

#	final_after_t1
1	110

2-й



Как исправить: изменить логику запросов. Либо делать атомарный апдейт или сделать лок на область данных. В данном случае используется второй вариант.

1-й

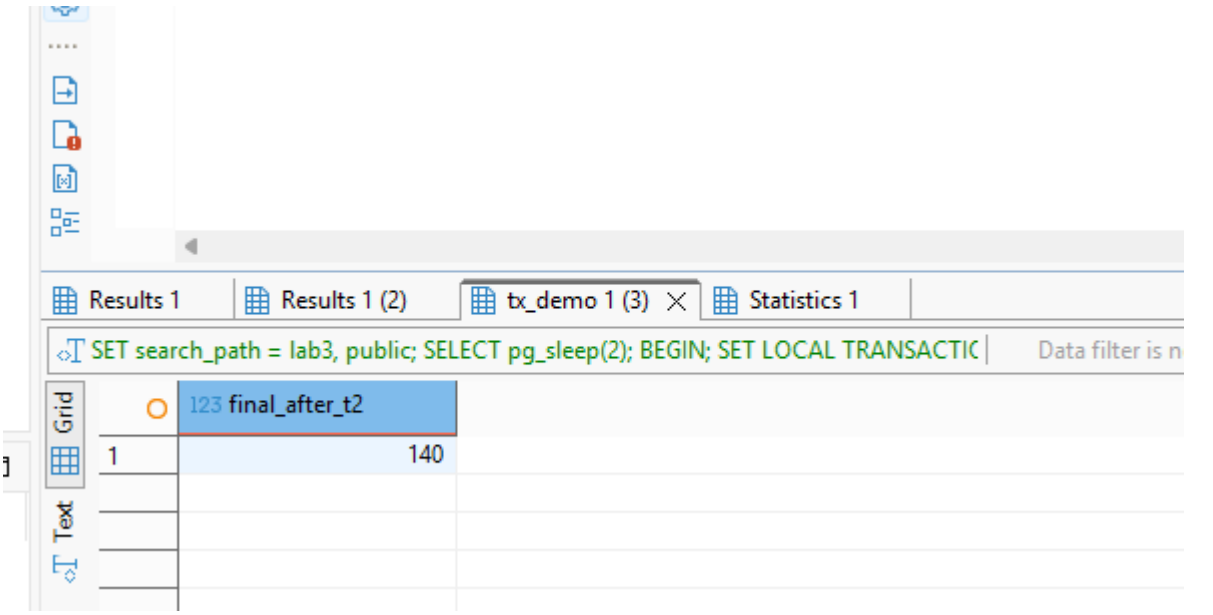
```sql
BEGIN;

SELECT val
FROM tx_demo
WHERE id = 1
FOR UPDATE;

SELECT pg_sleep(15);

UPDATE tx_demo
SET val = val + 10
WHERE id = 1;

COMMIT;
```

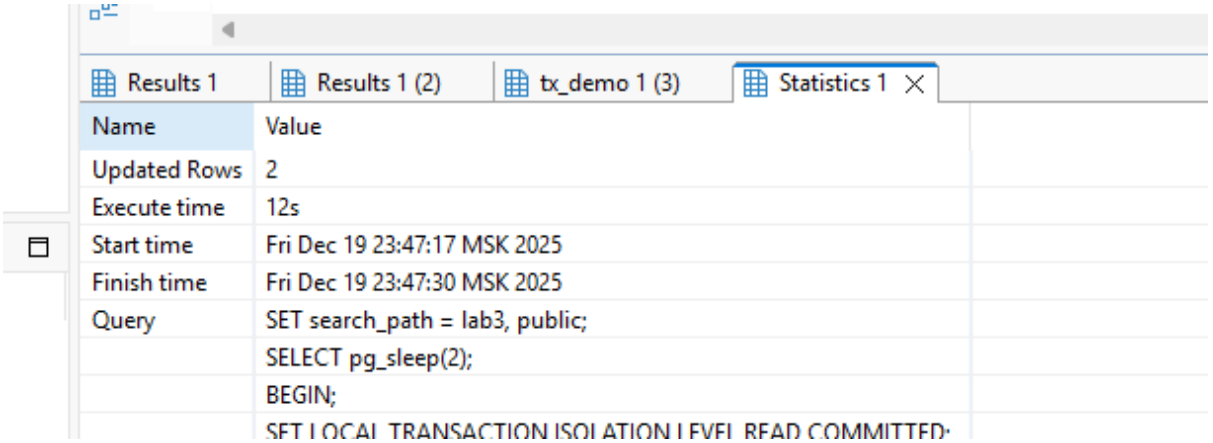
Connected (5 queries)

Run | Explain | Analyze

1: BEGIN | 2: SELECT 1 | 3: SELECT 1 | 4: UPDATE 1 | 5: COMMIT

#	val
1	120





По времени исполнения видно, что одна транзакция ждала другую, и уже после ее завершения выполнилась, получив корректное значение.