Design of the database

Andrii Yanechko

March 2022

1 Program Description

This is a database for ordering a ticket on the events.

2 Design database for CDP program

Main entities of the database are:

- User;
- Event;
- Ticket.

2.1 User entity

User entity, illustrated on a figure 2.1, represents user in the database and have several fields:

Name	**Type**	**Description**	**Constraints**
`id`	*integer*	unique identifier of the user	**Primary Key**
`name`	*text*	name of the user	**Unique**
`email`	*text*	email of the user	**Unique**
`created_date`	*text*	date of instance creation in the UTC	**N\A**
`updated_date`	*text*	date of the last update in the UTC	

Indexes for user entity:

Name	**Type**
`name`	*B-tree*
`email`	*B-tree*

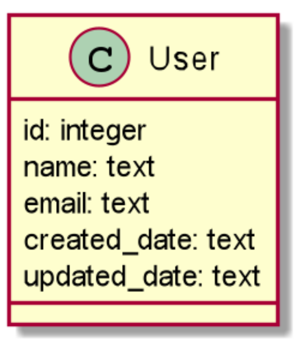


Figure 1: User representation in the database

2.2 Event entity

Event entity, illustrated on a figure 2 , represents event in the database and have several fields:

<table>
<tr><th>Name</th><th>Type</th><th>Description</th><th>Constraints</th></tr>
<tr><td>id</td><td>integer</td><td>unique identifier of the event</td><td>Primary Key</td></tr>
<tr><td>title</td><td>text</td><td>title of the event</td><td rowspan="2">Unique</td></tr>
<tr><td>date</td><td>text</td><td>start date of the event in the UTC</td></tr>
<tr><td>created_date</td><td>text</td><td>date of instance creation in the UTC</td><td rowspan="2">N\A</td></tr>
<tr><td>updated_date</td><td>text</td><td>date of the last update in the UTC</td></tr>
</table>

Indexes for event entity:

Name	Type
title	*B-tree*
date	*B-tree*

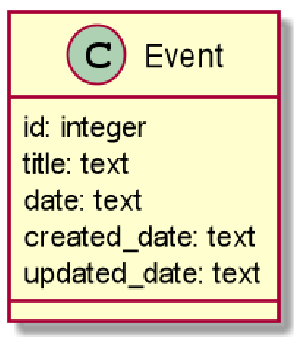


Figure 2: Event representation in the database

2.3 Ticket entity

Ticket entity, illustrated on a figure 3 , represents ticket in the database and have several fields:

<table>
<tr><th>Name</th><th>Type</th><th>Description</th><th colspan="2">Constraints</th></tr>
<tr><td>id</td><td>integer</td><td>unique identifier of the ticket</td><td colspan="2">Primary Key</td></tr>
<tr><td>user_id</td><td>integer</td><td>id of the user which has ordered this ticket</td><td colspan="2">Secondary Key</td></tr>
<tr><td>event_id</td><td>integer</td><td>id of the event on which ticket is booked</td><td>Secondary Key</td><td rowspan="2">Unique</td></tr>
<tr><td>place</td><td>integer</td><td>number of place of the ticket</td><td>N\A</td></tr>
<tr><td>category</td><td>string</td><td>category of the ticket</td><td colspan="2" rowspan="3">N\A</td></tr>
<tr><td>created_date</td><td>string</td><td>date of instance creation in the UTC</td></tr>
<tr><td>updated_date</td><td>string</td><td>date of the last update in the UTC</td></tr>
</table>

Indexes for ticket entity:

Name	Type
`event_id`	*B-tree*
`user_id`	*B-tree*
`category`	*B-tree*

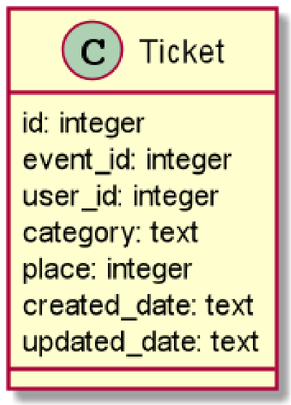


Figure 3: Ticket representation in the database

3 Implementation of the database design in the PostgresSQL

3.1 User table

```
 CREATE TABLE public."user"
(
id integer NOT NULL,
name character varying(50) NOT NULL,
email character varying(50) NOT NULL,
created_date character varying(50),
updated_date character varying(50),
PRIMARY KEY (id),
CONSTRAINT name_unique UNIQUE (name),
CONSTRAINT email_unique UNIQUE (email)
);

ALTER TABLE IF EXISTS public."user"
OWNER to postgres;
```

3.2 Event table

```
 CREATE TABLE public.event
(
id integer NOT NULL,
title character varying(50) NOT NULL,
date character varying(50) NOT NULL,
created_date character varying(50),
updated_date character varying(50),
PRIMARY KEY (id),
CONSTRAINT title_date UNIQUE (title, date)
);

ALTER TABLE IF EXISTS public.event
OWNER to postgres;
```

3.3 Ticket table

```
 CREATE TABLE public.ticket
(
id integer NOT NULL,
user_id integer NOT NULL,
event_id integer NOT NULL,
place integer NOT NULL,
category character varying(30) NOT NULL,
created_date character varying(50),
updated_date character varying,
PRIMARY KEY (id),
CONSTRAINT unique_event_id_place UNIQUE (event_id, place),
CONSTRAINT foreign_key_user_id FOREIGN KEY (user_id)
REFERENCES public."user" (id) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
NOT VALID,
CONSTRAINT foreign_key_event_id FOREIGN KEY (event_id)
REFERENCES public.event (id) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
NOT VALID
);

ALTER TABLE IF EXISTS public.ticket
OWNER to postgres;
```

3.4 Database entity relations

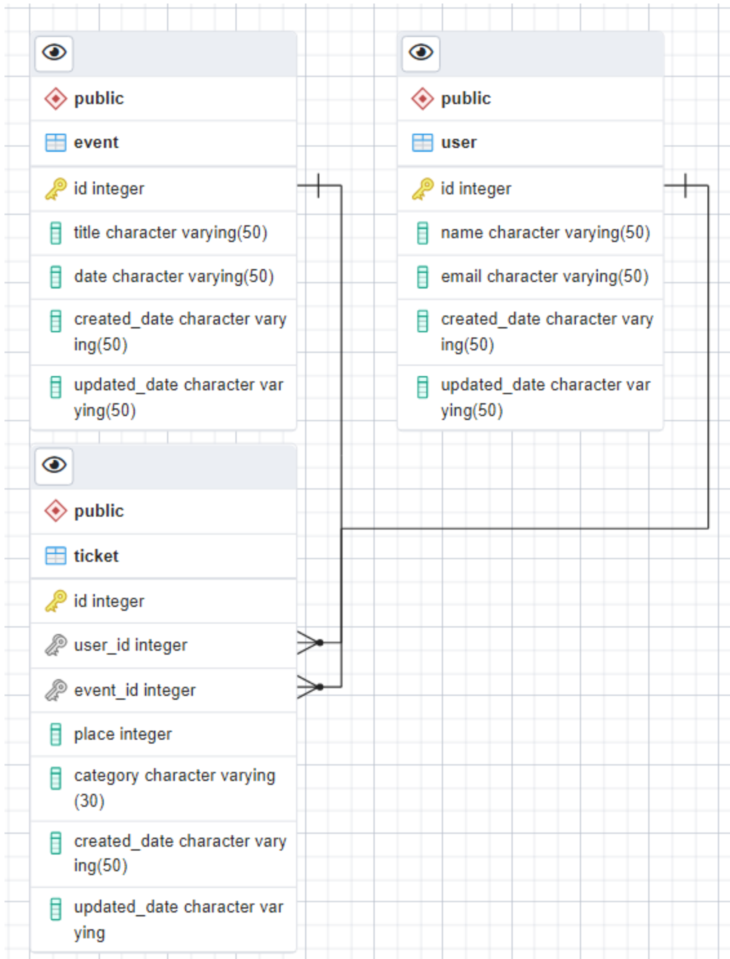


Figure 4: Entities relation in the database

3.5 Trigger for update time on UPDATE

Trigger function looks like:

```
    CREATE OR REPLACE FUNCTION public.set_update_time()
RETURNS trigger
LANGUAGE 'plpgsql'
VOLATILE
COST 100
AS $BODY$
```

```
BEGIN
new.updated_time = CURRENT_TIME(2);
RETURN new;
END;
$BODY$;
```

Trigger looks like:

```
CREATE TRIGGER set_update_time_on_update
BEFORE UPDATE OF id, name, email, created_date, updated_date
ON public."user"
FOR EACH ROW
EXECUTE FUNCTION public.set_update_time();
```

3.6 Trigger for name validation

Function to validate name:

```
CREATE FUNCTION public.name_validation()
RETURNS trigger
LANGUAGE 'plpgsql'
VOLATILE NOT LEAKPROOF
AS $BODY$
BEGIN
if NEW.name LIKE '%%' THEN
RAISE EXCEPTION 'name field contains  character';
ELSEIF NEW.name LIKE '%#%' THEN
RAISE EXCEPTION 'name field contains # character';
ELSEIF NEW.name LIKE '%$%' THEN
RAISE EXCEPTION 'name field contains # character';
END IF;
RETURN NEW;
END;
$BODY$;
```

Trigger that invokes BEFORE each update or insert on the `user.name` field:

```
ALTER FUNCTION public.name_validation()
OWNER TO postgres;

CREATE TRIGGER validate_name
BEFORE INSERT OR UPDATE OF name
ON public."user"
FOR EACH ROW
EXECUTE FUNCTION public.name_validation();
```

Trigger in action:

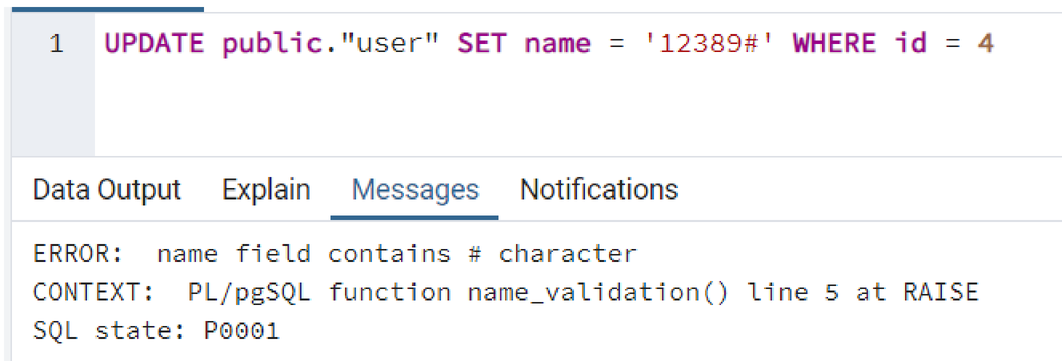


Figure 5: Trigger raises an exception when user name contains #

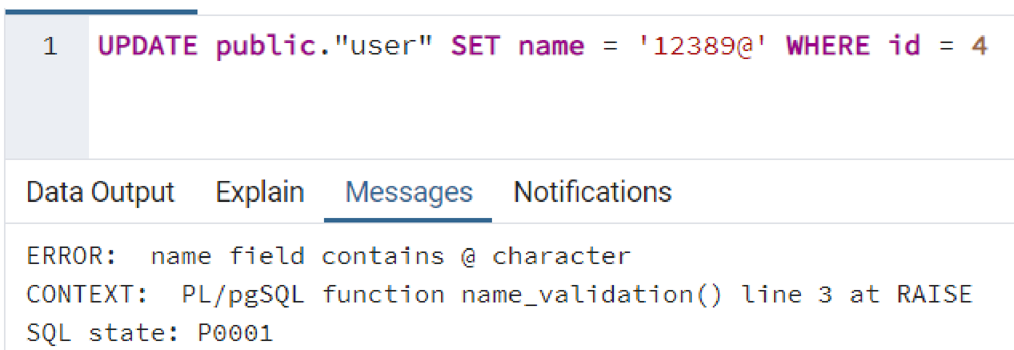


Figure 6: Trigger raises an exception when user name contains

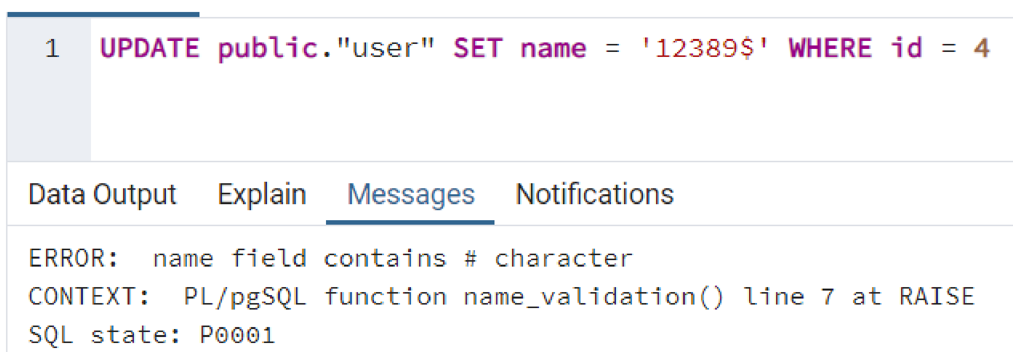


Figure 7: Trigger raises an exception when user name contains $

3.7 Function to return user's average place

```
 CREATE OR REPLACE FUNCTION public.avarage_user_ticket_place(IN "user"
"user")
RETURNS integer
LANGUAGE 'sql'
IMMUTABLE
PARALLEL UNSAFE
COST 100


RETURN (SELECT AVG(ticket.place) FROM ticket WHERE ticket.user_id =
"user".id);
```

3.8 Create a function that will return an average user name length for the event by event's name

```
 CREATE FUNCTION public.avarage_users_name_length_for_event(IN event_name
character varying)
RETURNS numeric
LANGUAGE 'sql'


RETURN (SELECT AVG(length) FROM (SELECT LENGTH("name") FROM (
SELECT DISTINCT "user".name as "name" FROM "event"
INNER JOIN ticket ON ticket.event_id = "event".id
INNER JOIN "user" ON "user".id = ticket.user_id
WHERE "event".title LIKE event_name
) as select_user_names) as select_length);

ALTER FUNCTION public.avarage_users_name_length_for_event(character
varying)
OWNER TO postgres;
```

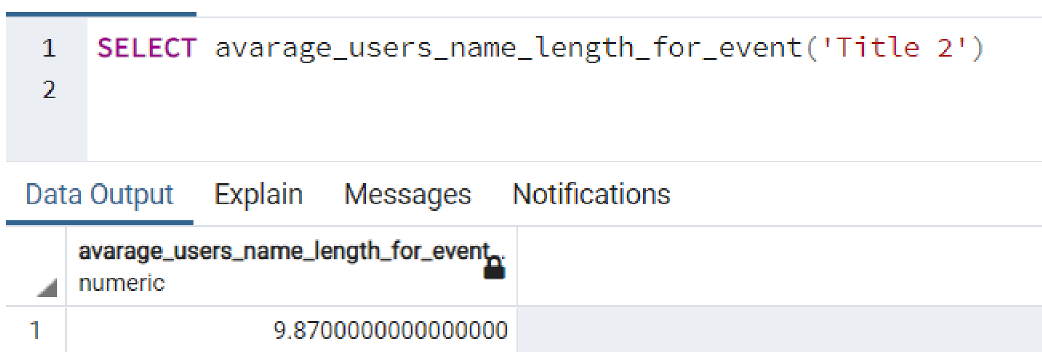


Figure 8: Select average user name for the event by event's name

3.9 Create a function that will return users that have more than 5 tickets

```
 CREATE FUNCTION public.select_users_which_have_more_then_5_tickets()
RETURNS SETOF "user"
LANGUAGE 'plpgsql'
IMMUTABLE
AS $BODY$
BEGIN
RETURN QUERY (
SELECT "user".* FROM ticket
INNER JOIN "user" on "user".id = ticket.user_id
WHERE ticket.id IS NOT NULL
GROUP BY "user".id
HAVING COUNT(*) > 5
);
END;
$BODY$;

ALTER FUNCTION public.select_users_which_have_more_then_5_tickets()
OWNER TO postgres;
```

4 PostgresSQL index comparation

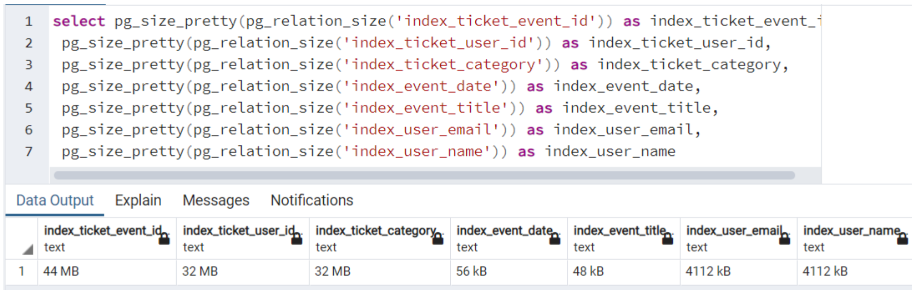
4.1 Index comparation

So, I've investigated 2 types of indexes **B-tree** and **hash**. I don't investigate 2 other indexes **GIN** and **GIST**, because they are not supporting a `character varying`, that is actully 95% of my indexes, so data for the investigation won't be accurate in that case.

4.1.1 Size of indexes

Query that I used to measure indexes size:

```
   select pg_size_pretty(pg_relation_size('index_ticket_event_id'))
as index_ticket_event_id,
pg_size_pretty(pg_relation_size('index_ticket_user_id')) as index_ticket_user_id,
pg_size_pretty(pg_relation_size('index_ticket_category')) as index_ticket_category,
pg_size_pretty(pg_relation_size('index_event_date')) as index_event_date,
pg_size_pretty(pg_relation_size('index_event_title')) as index_event_title,
pg_size_pretty(pg_relation_size('index_user_email')) as index_user_email,
pg_size_pretty(pg_relation_size('index_user_name')) as index_user_name
```

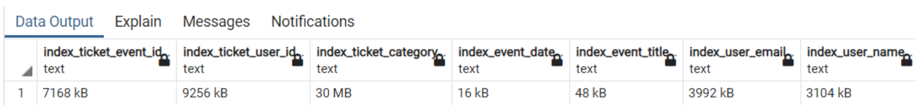


```sql
select pg_size_pretty(pg_relation_size('index_ticket_event_id')) as index_ticket_event_i
 pg_size_pretty(pg_relation_size('index_ticket_user_id')) as index_ticket_user_id,
 pg_size_pretty(pg_relation_size('index_ticket_category')) as index_ticket_category,
 pg_size_pretty(pg_relation_size('index_event_date')) as index_event_date,
 pg_size_pretty(pg_relation_size('index_event_title')) as index_event_title,
 pg_size_pretty(pg_relation_size('index_user_email')) as index_user_email,
 pg_size_pretty(pg_relation_size('index_user_name')) as index_user_name
```

Data Output Explain Messages Notifications

	index_ticket_event_id text	index_ticket_user_id text	index_ticket_category text	index_event_date text	index_event_title text	index_user_email text	index_user_name text
1	44 MB	32 MB	32 MB	56 kB	48 kB	4112 kB	4112 kB

Figure 9: Represents how much size each index takes using hash indexes



Data Output Explain Messages Notifications

	index_ticket_event_id text	index_ticket_user_id text	index_ticket_category text	index_event_date text	index_event_title text	index_user_email text	index_user_name text
1	7168 kB	9256 kB	30 MB	16 kB	48 kB	3992 kB	3104 kB

Figure 10: Represents how much size each index takes using B-tree indexes

4.1.2 Speed of queries

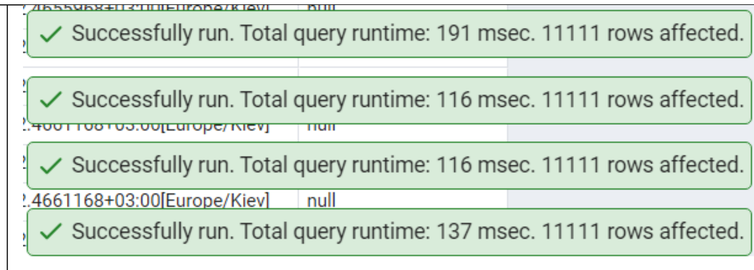
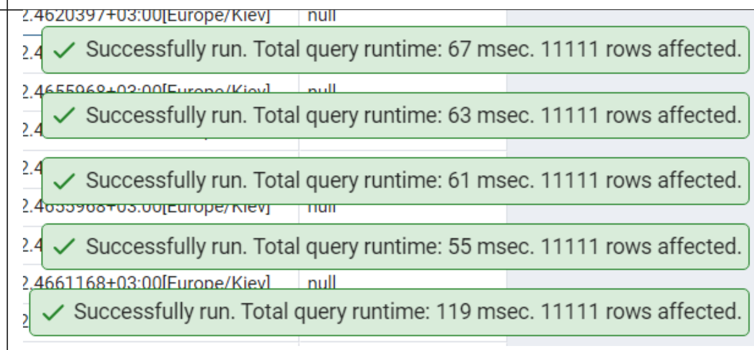
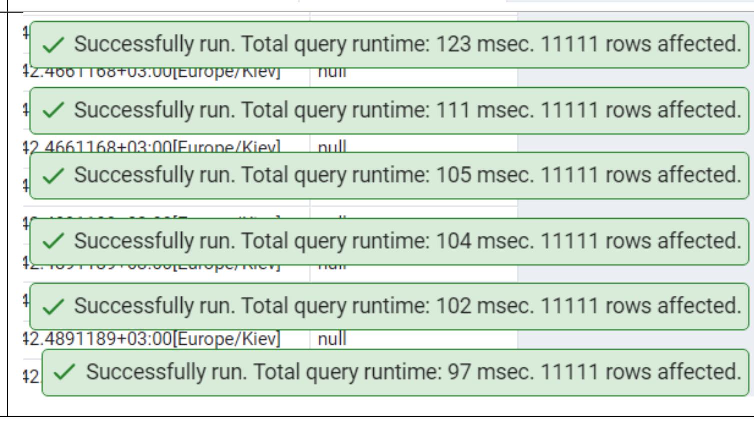
Query for selecting user by name (exact match):

```
SELECT * FROM public."user" WHERE "name" LIKE 'name 1050'
```

B-tree	Successfully run. Total query runtime: 111 msec. 1000 rows affected. Successfully run. Total query runtime: 106 msec. 1000 rows affected. Successfully run. Total query runtime: 104 msec. 1000 rows affected. Successfully run. Total query runtime: 110 msec. 1000 rows affected.
hash	Successfully run. Total query runtime: 57 msec. 1000 rows affected. Successfully run. Total query runtime: 58 msec. 1000 rows affected. Successfully run. Total query runtime: 64 msec. 1000 rows affected. Successfully run. Total query runtime: 64 msec. 1000 rows affected. Successfully run. Total query runtime: 55 msec. 1000 rows affected.
no index	Successfully run. Total query runtime: 107 msec. 1000 rows affected. Successfully run. Total query runtime: 108 msec. 1000 rows affected. Successfully run. Total query runtime: 104 msec. 1000 rows affected. Successfully run. Total query runtime: 112 msec. 1000 rows affected. Successfully run. Total query runtime: 108 msec. 1000 rows affected.

Query for selecting user by name (contains):

```
SELECT * FROM public."user" WHERE name LIKE "%name 5%"
```

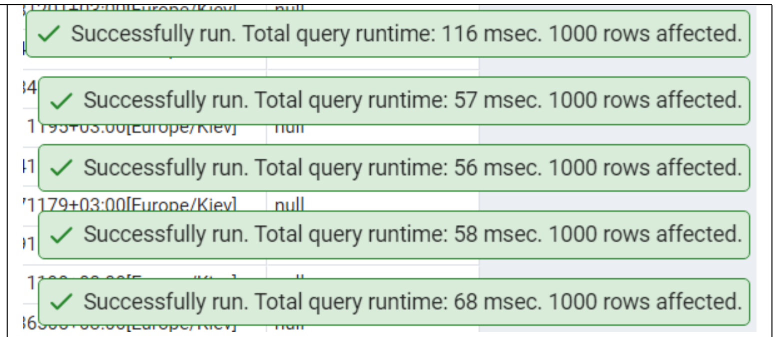
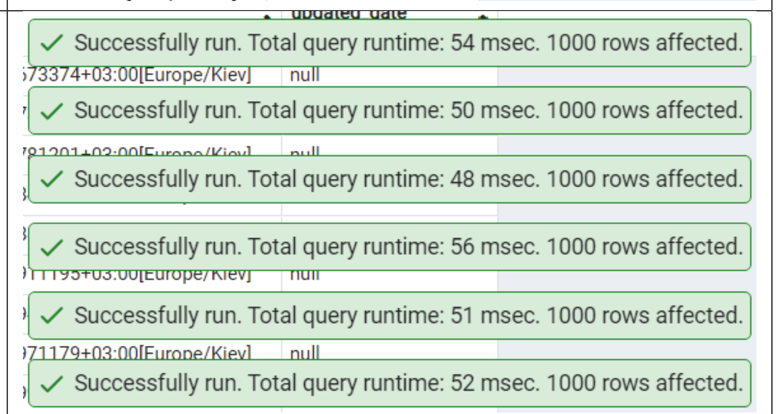
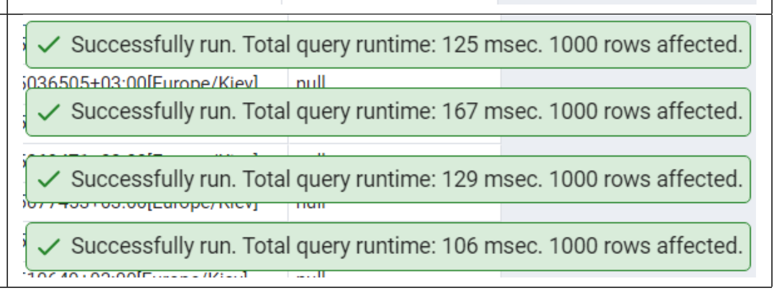
<table>
<tr><td>B-tree</td><td>
</td></tr>
<tr><td>hash</td><td>
</td></tr>
<tr><td>no index</td><td>
</td></tr>
</table>

Query for selecting user by email (contains):
`SELECT * FROM public."user" WHERE email LIKE "%17@%"`

B-tree	Successfully run. Total query runtime: 111 msec. 1000 rows affected. Successfully run. Total query runtime: 106 msec. 1000 rows affected. Successfully run. Total query runtime: 104 msec. 1000 rows affected. Successfully run. Total query runtime: 110 msec. 1000 rows affected.
hash	Successfully run. Total query runtime: 57 msec. 1000 rows affected. Successfully run. Total query runtime: 58 msec. 1000 rows affected. Successfully run. Total query runtime: 64 msec. 1000 rows affected. Successfully run. Total query runtime: 64 msec. 1000 rows affected. Successfully run. Total query runtime: 55 msec. 1000 rows affected.
no index	Successfully run. Total query runtime: 107 msec. 1000 rows affected. Successfully run. Total query runtime: 108 msec. 1000 rows affected. Successfully run. Total query runtime: 104 msec. 1000 rows affected. Successfully run. Total query runtime: 112 msec. 1000 rows affected. Successfully run. Total query runtime: 108 msec. 1000 rows affected.

Query by email from users that have at least one ticket (contains):

```
SELECT public."user".* FROM public."user"
WHERE (SELECT COUNT(*)FROM public.ticket WHERE public.ticket.id = public."user".id)
> 0
AND public."user".email LIKE "%77@%"
```

B-tree	
hash	
no index	

4.2 My conclusion

So as I can see **B-tree** index is not very useful in my case, I guess it's because my indexes are almost refers `character varying` types and that's why **B-tree** is not very efficient.

Taking look on the size which indexes are taking, **hash** is on the figure and on the **b-tree** figure, **hash** index takes more size than **b-tree**, but taking a look at the efficient comparing tables **hash** is more-more efficient in case of `character varying` indexes.