DAT2000

DATABASE 2

Mandatory Assignment

USN RINGERIKE H24

25.10.2024

Group 7

129605 Mia Trollstøl

209660 Kenneth Andreas Hansen

265931 Kristian Martin Tvenning

263423 Jonas El Hbabi Helling

265954 Lucas Leon Svaet Holter

Index

Part 1. Physical database design

1.

The file organization offered by PostgreSQL is heap files. Most DBMS uses heap files with indexes. The reason is that with sequential files it creates too much activity on the hard disk, that creates problems with concurrency control.

The advantages of this file organization are flexibility and fast insertions since there is no need for reordering. Heap also handles concurrent reads important to concurrency control. The disadvantages are that it can become fragmented over time with wasted space and slower access. It needs regular maintenance VACUUM to reclaim unused space.

2.

The default index structure provided by PostgreSQL is B+tree. The advantages of B+ tree structure is that it automatically reorganizes itself, so reorganization of the entire file is not required to maintain performance.

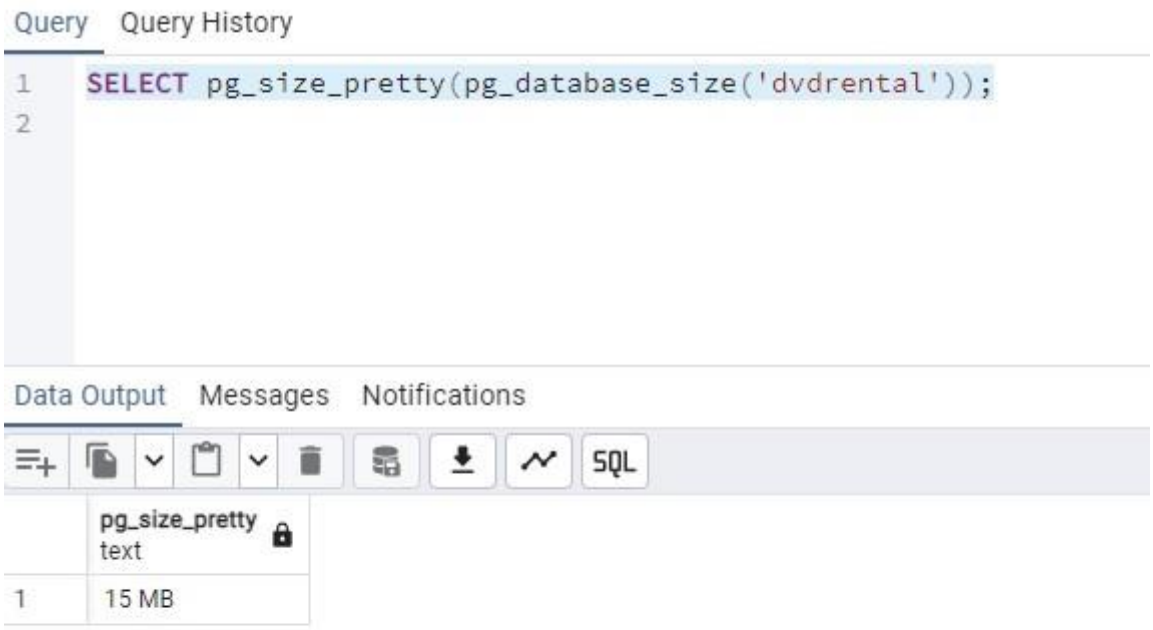
The disadvantages are extra insertion and deletion overhead. The process of updating indexes involves extra overhead. Space overhead increases storage and maintenance cost.

3.

3.1.

SQL query to determine the size of the whole dvdrental database.

```
SELECT pg_size_pretty(pg_database_size('dvdrental'));
```



3.2.

SQL query to determine the size of each table in the dvdrental database.

The query used to determine the size of each table in the dvdrental database sorted by size limit 15 since there are 15 tables in the database.

SELECT relname AS "relation", pg_size_pretty (pg_total_relation_size (C .oid)) AS "total_size"

FROM pg_class C

LEFT JOIN pg_namespace N ON (N.oid = C .relnamespace)

WHERE nspname NOT IN ('pg_catalog', 'information_schema') AND C .relkind <> 'i' AND

nspname !~ '^pg_toast'

ORDER BY pg_total_relation_size (C .oid) DESC

LIMIT 15;

Query Query History

```
1  SELECT
2   relname AS "relation", pg_size_pretty (pg_total_relation_size (C .oid)) AS "total_size" FROM
3  pg_class C
4  LEFT JOIN pg_namespace N ON (N.oid = C .relnamespace)
5  WHERE nspname NOT IN ('pg_catalog', 'information_schema') AND C .relkind <> 'i' AND
6  nspname !~ '^pg_toast'
7  ORDER BY pg_total_relation_size (C .oid) DESC
8  LIMIT 15;
```

Data Output Messages Notifications

SQL

	relation name	total_size text
1	rental	2352 kB
2	payment	1816 kB
3	film	936 kB
4	film_actor	488 kB
5	inventory	440 kB
6	customer	208 kB
7	address	152 kB
8	film_category	112 kB
9	city	112 kB
10	actor	72 kB
11	store	40 kB
12	staff	32 kB
13	language	24 kB
14	country	24 kB
15	category	24 kB

3.3.

SQL query to determine the size of indexes.

Example from table customer.

SELECT indexname, indexdef, pg_size_pretty(pg_indexes_size('customer'))

FROM pg_indexes

WHERE tablename = 'customer';

Query Query History

```
1 SELECT indexname, indexdef, pg_size_pretty(pg_indexes_size('customer'))
2 FROM pg_indexes
3 WHERE tablename = 'customer';
4
```

Data Output Messages Notifications

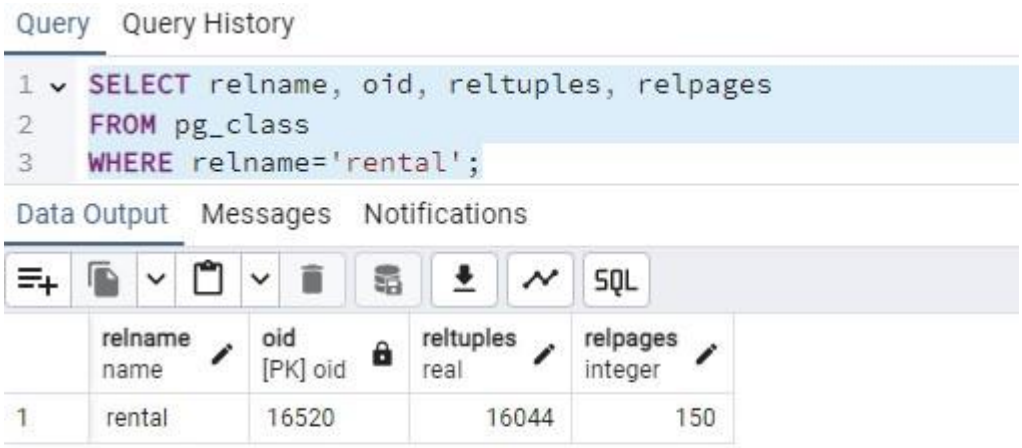
	indexname name	indexdef text	pg_size_pretty text
1	customer_pkey	CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (customer_...	112 kB
2	idx_fk_address_id	CREATE INDEX idx_fk_address_id ON public.customer USING btree (address_id)	112 kB
3	idx_fk_store_id	CREATE INDEX idx_fk_store_id ON public.customer USING btree (store_id)	112 kB
4	idx_last_name	CREATE INDEX idx_last_name ON public.customer USING btree (last_name)	112 kB

3.4.

SQL query to determine the number of tuples in each table.

Example from rental table.

```
SELECT relname, oid, reltuples, relpages
FROM pg_class
WHERE relname='rental';
```



3.5.

SQL query to determine the number of pages to store tuples in each relation.

Example from film:

SELECT CTID, * from film;

53,6 = 54 pages

Query Query History

```
1  select CTID, * from film;
```

Data Output Messages Notifications

	ctid tid	film_id [PK] integer	title character
982	(52,6)	982	Women D
983	(52,7)	983	Won Dare
984	(52,8)	984	Wonderfu
985	(52,9)	985	Wonderla
986	(52,10)	986	Wonka Se
987	(52,11)	987	Words Hu
988	(52,12)	988	Worker Ta
989	(52,13)	989	Working M
990	(52,14)	990	World Lea
991	(52,15)	991	Worst Bai
992	(52,16)	992	Wrath Mil
993	(52,17)	993	Wrong Be
994	(52,18)	994	Wyoming
995	(53,1)	995	Yentl Idah
996	(53,2)	996	Young La
997	(53,3)	997	Youth Kic
998	(53,4)	998	Zhivago C
999	(53,5)	999	Zoolande
1000	(53,6)	1000	Zorro Ark

Part 2. Denormalization

- **Analyze the logical data for `dvdrental` database and suggest areas that require denormalization. Make any assumptions to justify your case.**

  We assume the dvdrental store workers are permanently employed and that its customers do not frequently change their address. Therefore, it could be wise to denormalize the address table, given that this database is used for the dvdrental store, and they likely have no interest in seeing where their selves live. For salary payments it is assumed that a third-party salary solution has this covered. However, for "undelivered DVD" fees it would be swifter to have the customer address in the customer table.

1. **You may check first the 7 views that currently exist in the database.**
   a. Actor_info: shows the actor_id, first_name, last_name and a column of lists of films the actors have starred in and that film's category; customer list: shows the ID, name and other personal info about the customer, as well as a "notes" column; film_list, nicer_but_slower_film_list, sales_by_film_category, sales_by_store and staff_list is self-explanatory.

2. **If you determined an area for denormalization, would you create regular views or materialized views? And why?**
   a. How often would one get new films? Let's consider the `film_list`. There is no need to update this more often than the store gets new films. Therefore, the `film_list` can be a materialized view, to retrieve the results faster. This is handy whether one wants to know what films is in the inventory or if the customer is wondering "Do you have {film}?" without waiting too much and avoiding unnecessary overhead. If the view is re-materialized each week (say Thursday to Friday night, before the weekend), then this would also be the case for the `actor_info` view (to keep consistency in the film-actor relation). Sales data would not be interesting without a range of dates to compare with, so this is also not interesting to keep as a regular view.

3. **Propose a list of new views**
   Proposal for new Views: Rental history, Films_Available

- 1) Discuss for each of these views the reasons for creating it and how it will be useful:
- **Films_Available**: Helps staff respond to customer inquiries about film availability. Both in terms of what the customer will be available to rent, but also to see if it might be feasible to order more copies of a film that often is

unavailable due to extensive rentals and thus not being available to be rented, like those who are popular films through time like Titanic and Pulp Fiction.
Rental history: it can be useful in customer service such as if the customer asks for recommendations based on previous history or want to rent the movie, they rented last year but can't remember the name etc. Furthermore, it can also be useful for business analysis (data cube).

- Provide the SQL code for creating each of these views that you proposed:

SQL code for each view:

Films_available:

```
-- Films available
CREATE VIEW films_available AS
SELECT film.film_id, film.title,
COUNT(inventory.inventory_id) AS total_copies,
COUNT(inventory.inventory_id) - COUNT(rental.rental_id)
AS available_copies
FROM inventory
JOIN film ON inventory.film_id = film.film_id
LEFT JOIN rental
ON inventory.inventory_id = rental.inventory_id
AND rental.return_date IS NULL
GROUP BY film.film_id, film.title;
```

Rental_history:

```
CREATE VIEW rental_history AS
SELECT * FROM rental;
```

(Assuming the GUI prompts the manager to enter customer ID, where it is later queried "SELECT * FROM rental WHERE customer_ID = {number}"

Part 3. Database administration and security

1.

Database security is the mechanisms that protects the database against unauthorized access. There are three main objectives in database security. The first is secrecy. Information should not be disclosed to unauthorized users. The second is integrity. Only authorized users should have the opportunity to modify data. The third is availability. If you are an authorized user, you should not be denied access.

PostgreSQL implements several security measures to protect the database.

Authorization is the process of granting a right or privilege to control access to the database. It determines if the user is who it claims to be. PostgreSQL uses roles to control access. A role can be a user or a group of users. GRANT and REVOKE are used to assign or remove privileges on parts of or the whole database.

Access control ensures only authorized users can access and manipulate data. It includes authentication and controls what the users can do. The approach in PostgreSQL is called Discretionary Access Control (DAC). Access control is gained by granting and revoking privileges. (GRANT and REVOKE). The privileges SELECT, INSERT, UPDATE and DELETE dictate what the users can do. The users may be assigned different forms of authorization parts of the database. The privileges allow the user to create or access what they need to accomplish their job.

Data integrity involves maintaining the consistency of the data in the database. Integrity prevents data from becoming invalid. Invalid data will give invalid or incorrect results. Only authorized users should be able to modify data. To enforce data integrity PostgreSQL uses constraints such as primary keys, foreign key and other rules. Transaction integrity is ensured by the ACID properties. (Atomicity, consistency, isolation, durability). The reliable processing of transactions ensures data integrity.

Encryption is encoding of data by a special algorithm that makes the data unreadable without the decryption key. Encryption transforms text into non-readable cipher text. To read it, the text must be transformed back to readable text.

There are two categories of cryptography systems. With The Symmetric key cryptography, both the sender and the receiver use the same digital key to encrypt and decrypt the message. The public key cryptography system uses pairs of keys. Each pair consists of a public key which is widely distributed and known to others, and a private key that is kept secret by the owner. The sender encrypts the message with public key of the receiver, then the receiver decrypts the message with its private key. PostgreSQL supports Secure sockets layer (SSL)/Transport Layer security (TLS) to secure data transmission between clients and the database server. This protects the data during transit.

PostgreSQL uses authorization, access control, data integrity and encryption to protect databases from unauthorized access and threats. This ensures that confidential and accurate data is available to authorised users.

2.

2.1

Create a role for staff member 1.

CREATE ROLE staff_1 WITH LOGIN PASSWORD ‘staff1password’;

Grant necessary privileges to do their task.

GRANT SELECT, INSERT, UPDATE, DELETE ON inventory TO staff_1;

GRANT SELECT, INSERT, UPDATE, DELETE ON rental TO staff_1;

GRANT SELECT, INSERT, UPDATE, DELETE ON payment TO staff_1;

Create a role for staff member 2.

CREATE ROLE staff_2 WITH LOGIN PASSWORD 'staff2password';

Grant necessary privileges to do their task.

GRANT SELECT, INSERT, UPDATE, DELETE ON inventory TO staff_2;

GRANT SELECT, INSERT, UPDATE, DELETE ON rental TO staff_2;

GRANT SELECT, INSERT, UPDATE, DELETE ON payment TO staff_2;

2.2.

```
CREATE VIEW Customer_list_staff1 AS
SELECT *
FROM Customer
WHERE store_id = 1;

CREATE VIEW Customer_list_staff2 AS
SELECT *
FROM Customer
WHERE store_id = 2;
```

2.3.

Provide read, insert, update, and delete privileges to staff member 1 on the view customer_list_staff1.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON Customer_list_staff1 TO staff _1
```

2.4.

Provide read, insert, update, and delete privileges to staff member 2 on the view customer_list_staff2.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON Customer_list_staff2 TO staff _2
```

2.5.

Provide read privilege to staff member 1 on the view customer_list_staff2.

```
GRANT SELECT ON Customer_list_staff2 TO staff _1
```

2.6.

Provide read privilege to staff member 2 on the view customer_list_staff1.

```
GRANT SELECT ON Customer_list_staff1 TO staff _2
```

Part 4. Stored procedures and triggers

1.

“film_in_stock” is a procedure/function used to check whether or not copies of certain movies are available at a specific store. The procedure takes film_id and store_id as inputs/arguments, and then returns the corresponding list of films as inventory_id. Each copy of the same movie will have a different inventory_id. Therefore, you will see the “SETOF” statement take part of the return-statement which ensures that there can be multiple rows returned. This is done by running the query on the Inventory table.

2.1

When renting a dvd, the customer might find it difficult to decide exactly which title to rent since there are many movies to choose from. Therefore, we want to make it easier for the employee to help customers with this type of decision making. We have concluded two ways to implement a trigger to achieve this. (1) By implementing a film popularity table, followed by a trigger for automatically keeping count of how many times a movie has been rented, and (2) by using a materialized view which is updated on insert at the rental table. The use of a trigger function can quickly provide the information needed for ranging titles popularity without needing to iterate the rental table.

The table method:

Creating the table to track how many times a title is rented:

```
CREATE TABLE film_popularity(
    film_id INT PRIMARY KEY,
    rent_count INT DEFAULT 0
);
```

film_id column identifies the title, and rent_count column keeps track of how many times a title has been rented.

2.2

Creating the trigger function:

```
8  CREATE OR REPLACE FUNCTION update_film_popularity()
9  RETURNS TRIGGER
10 LANGUAGE 'plpgsql'
11 AS $BODY$
12 BEGIN
13     IF EXISTS (SELECT 1 FROM film_popularity WHERE film_id = NEW.film_id)
14         THEN
15         UPDATE film_popularity
16         SET rent_count = rent_count + 1
17         WHERE film_id = NEW.film_id;
18     ELSE
19         INSERT INTO film_popularity(film_id, rent_count)
20         VALUES (NEW.film_id, 1)
21     END IF;
22     RETURN NEW;
23 END;
24 $BODY$;
```

update_film_popularity() is a trigger function that updates the rent_count row value of a movie title in the film_popularity table each time the title is rented.

2.3

- **Line 8-10:** Creating function and naming it “update_film_popularity()”, and specifying that that the function should respond to triggering events such as INSERT and UPDATE.
- **Line 10-12:** Setting up for the functionality by defining PostgreSQL as language and beginning the functions body and logic.
- **Line 13-17:** IF EXISTS test to check if the NEW.film_id (the rented title in process) exists in a row (SELECT 1) in the film_popularity table. If the test is true, there is a match, and the corresponding title's row in the film_popularity table is updated by adding 1 to the rent_count column of that title.
- **Line 18-20:** The ELSE statement is only executed whenever the IF returns false. This is a result of the NEW.title not being found in the film_popularity table. Therefore, a new row is inserted to the film_popularity table containing the NEW.title and a default value of 1 for the rent_count column. Practically this means that this is the first time a title has been rented since the implementation of the new film_popularity table.

- Line 21-24: Ending the if test, returning the new row, ending the logic, and lastly ending the body of the function.

Creating the trigger:

```
26 CREATE TRIGGER track_film_popularity
27 AFTER INSERT ON rental
28 FOR EACH ROW
29 EXECUTE FUNCTION update_film_popularity
```

When creating the trigger track_film_popularity, we specify that the trigger should act after an insert on- and operate on each row in the rental table. The function update_film_popularity is executed when the trigger is fired.

The materialized view method:

```
36 CREATE MATERIALIZED VIEW film_popularity AS
37 SELECT film_id, COUNT(*) AS rent_count
38 FROM rental
39 GROUP BY film_id;
```

We start by creating the materialized view, which implies that the view has a stored value which can be updated on insert. The view is populated with data upon initiation, and is identified by film_id, which allows for a count of how many times a title has been rented (rent_count).

Creating the trigger function:

```
42 CREATE OR REPLACE FUNCTION update_film_popularity();
43 RETURNS TRIGGER
44 LANGUAGE 'plpgsql'
45 AS $BODY$
46 BEGIN
47     REFRESH MATERIALIZED VIEW film_popularity;
48     RETURN NEW;
49 END;
50 $BODY$
```

This trigger function differs from the other trigger function by continuously updating the materialized view. This way, the view will always stay up do date with the rental table.

Key differences between the table method and the materialized view method:

- Both methods are executed on insert, and ensures that the data is easily accessible, however, there are a few key differences to evaluate. Using the table method will increase one title incrementally in a new table when the title is rented, whereas the view will fully update itself on insert. Updating the view frequently has a higher cost than reading a number from a column in a table. Still, there can be faster read speeds from a materialized view, which depends on the complexity of the underlying queries. To conclude, the suggested method would be the use of a materialized view based on its performance capabilities to ensure quick accessible data for the salesperson/employee.

Now that we have established a way for the employee to confidently recommend titles to costumers, we want to further assist with the sales conversation even after the customer has decided upon a title to rent. Therefore, notifying the employee about the additional sales questions before the transaction can be carried out, will likely improve the quality of sales. The sales questions should regard other products such as popcorn, where the employee is notified: ex. “Have you remembered to ask if the customer wants popcorn and a drink”.

Creating the trigger function:

```
60  CREATE OR REPLACE FUNCTION sales_reminder()
61  RETURNS TRIGGER
62  LANGUAGE 'plpgsql'
63  AS $BODY$
64  BEGIN
65      RAISE EXCEPTION 'Have you asked about popcorn and drinks?';
66      RETURN NEW;
67  END;
68  $BODY$
```

Trigger nr. 2

In this trigger function, the employee must confirm that they have asked the questions for additional sales. This is accomplished by raising an exception, which compared to raising a notice will halt the process of inserting a new row into the rental table. This forces the employee to interact with the program before the process can continue.

Creating the trigger:

```
70  CREATE TRIGGER trigger_sales_reminder
71  BEFORE INSERT ON rental
72  FOR EACH ROW
73  EXECUTE FUNCTION sales_reminder();
```

As mentioned, the process cannot continue without the employee's acknowledgement of the exception. This halting is a result of the BEFORE INSERT statement, which implies that the trigger should fire before insert on the rental table.

```
107 -- Function to check if the customer has any overdue rentals
108 CREATE OR REPLACE FUNCTION check_overdue_rentals()
109 RETURNS TRIGGER AS $$
110 DECLARE
111     overdue_count INT;
112 BEGIN
113     -- Query the rental table to count overdue rentals
114     -- Overdue means more than 7 days late and a "return_date" value of NULL
115     SELECT COUNT(*)
116     INTO overdue_count
117     FROM rental
118     WHERE customer_id = NEW.customer_id
119       AND return_date IS NULL
120       AND EXTRACT(DAY FROM CURRENT_DATE - rental_date) > 7;
121
122     -- If any overdue rentals exist, stop the rental process
123     IF overdue_count > 0 THEN
124         RAISE EXCEPTION 'Please return already rented film, which is currently overdue.';
125     END IF;
126
127     -- If no overdue rentals, proceed with the new rental
128     RETURN NEW;
129 END;
130 $$ LANGUAGE plpgsql;
131
132 CREATE TRIGGER stop_film_rent
133 BEFORE INSERT ON rental
134 FOR EACH ROW
135 EXECUTE FUNCTION check_overdue_rentals();
```

2.4

```
CREATE OR REPLACE FUNCTION stop_film_rent()
RETURNS TRIGGER AS $$
BEGIN
 IF EXISTS (
    SELECT 1
    FROM rental
    WHERE customer_id = NEW.customer_id
    AND return_date IS NULL
    AND CURRENT_TIMESTAMP - rental_date > INTERVAL '7 days'
  ) THEN
    RAISE EXCEPTION 'Please return already rented film, which is currently overdue.';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tr_stop_film_rent
BEFORE INSERT ON rental
FOR EACH ROW
EXECUTE FUNCTION stop_film_rent();
```