Library Database Report

Architect & Developer: Jonathan Wilson

1. Database Design

The library database represents a management system for a public library. The system helps the library staff manage their resources, including books, magazines, digital media, and other materials. Additionally, it provides efficient access to library member's information and facilitates borrowing and tracking of library materials. The library management system ensures data integrity and minimizes redundancy for both library staff and members.

1.1. Project Scope and Overview

This project has several parts.

- **Design** – This section covers the aspects of the design of the library database. The ER diagram helps us to to understand the various entities along with their attributes and to facilitate the visualization of the relationships between the various entities. The relational mappings show us the mappings between the entities and their associated keys. This mapping will aid us in understanding how we might construct primary and foreign keys.
- **Database Implementation** – This section demonstrates how the database was created and initialized with sample data.
- **Querying and Manipulation** – This section shows the functional requirements and the solution implementations to those requirements.
- **Extended Features** – Finally we explore two extended features that we can incorporate into the existing database through slight modifications.

Note that assumptions that were made are clarified in each section of this report. If you have any concerns please contact the architect and developer listed above or see the code for more details. For the sake of brevity and simplicity only the essentials are explained leaving out details that can easily be derived from the explanations and examples that will be presented.

1.2. ER Diagram

Figure 1 shows the ER diagram for the library database. The following will explore the ER diagram in depth by looking at each entity and relation:

- **Material** – In this library database design the Material is the central asset. It has a key material_id that uniquely identifies a Material object consisting of title, publication_date, genre_id, and catalog_id. Material can contain any kind of library object or asset such as books, magazines, etc. The genre_id and catalog_id are foreign keys that help us tie a particular material to a genre and catalog. Some materials might not have a genre or catalog yet due to various reasons but can still be logged into the database and changed at a later date.

- **Genre** – Material and genre are related through the Categorized As relationship linked via the genre_id. A particular genre might be Fiction or History that contains a description of what the genre is about. Many materials can also have the same genre but each individual material will only have one genre.
- **Catalog** – likewise the Catalog entity is linked with Material via the catalog_id and has the Cataloged In relationship. A particular material can be cataloged as a type of thing such as magazine or book and a designated location. Many materials can also have the same catalog but each individual material will only have one catalog.
- **Borrow** – this entity represents a particular event or transaction. A material can be borrowed by a particular member and a staff member has to process this transaction. A member may borrow many materials but there is only one record tied to each material for a particular date, member and material. Borrow has been designated as a weak entity as without a material, member and a specific borrow date it can not uniquely be identified. borrow_id has been designated as a surrogate key and primary key but each borrow record must maintain a uniqueness constraint on material_id, member_id and borrow_id. This method was done to simplify joins and other query functions. The identifying relationships are Borrowed and Borrowed By tied to Material and Member respectively. Lastly, is the Staff entity, identified by the foreign key staff_id, which represents a library staff member who does the actual processing of the transaction. Other attributes are used for keeping track of the material. If the return_date is NULL then it is assumed that the material has not been returned yet. If the return_date is NULL and the due_date is less than the current date then a material is marked overdue. See extended features.
- **Member** – this entity was explained a bit above. Members are those who use materials. They are uniquely identified by a member_id (such as a library card number) and are actual people with a name, contact_info, and join_date.
- **Staff -** this entity was also explained above. Staff are library workers who take care of transactions among other various duties. They have attributes such as name, hire_date, job_title, contact_info, and have a unique identifier staff_id.
- **Authorship** – Next we move to the upper right of the diagram. Authorship represents an entity that explains who authored what material. An Authorship is linked to Material via a Authored relationship. Authorship is uniquely identified by the surrogate key authorship_id but depends on the uniqueness constraint of the foreign keys author_id and material_id as it is also a weak entity which requires the existence of a material and author to make any sense. Similar to Borrow. There might be multiple authors associated with the same authorship but I choose to record each as separate records instead of creating a multi-valued author_id attribute.
- **Author** – is a person who contributes to an authorship. A author has a name, birth-date, nationality, and an identifier author_id.

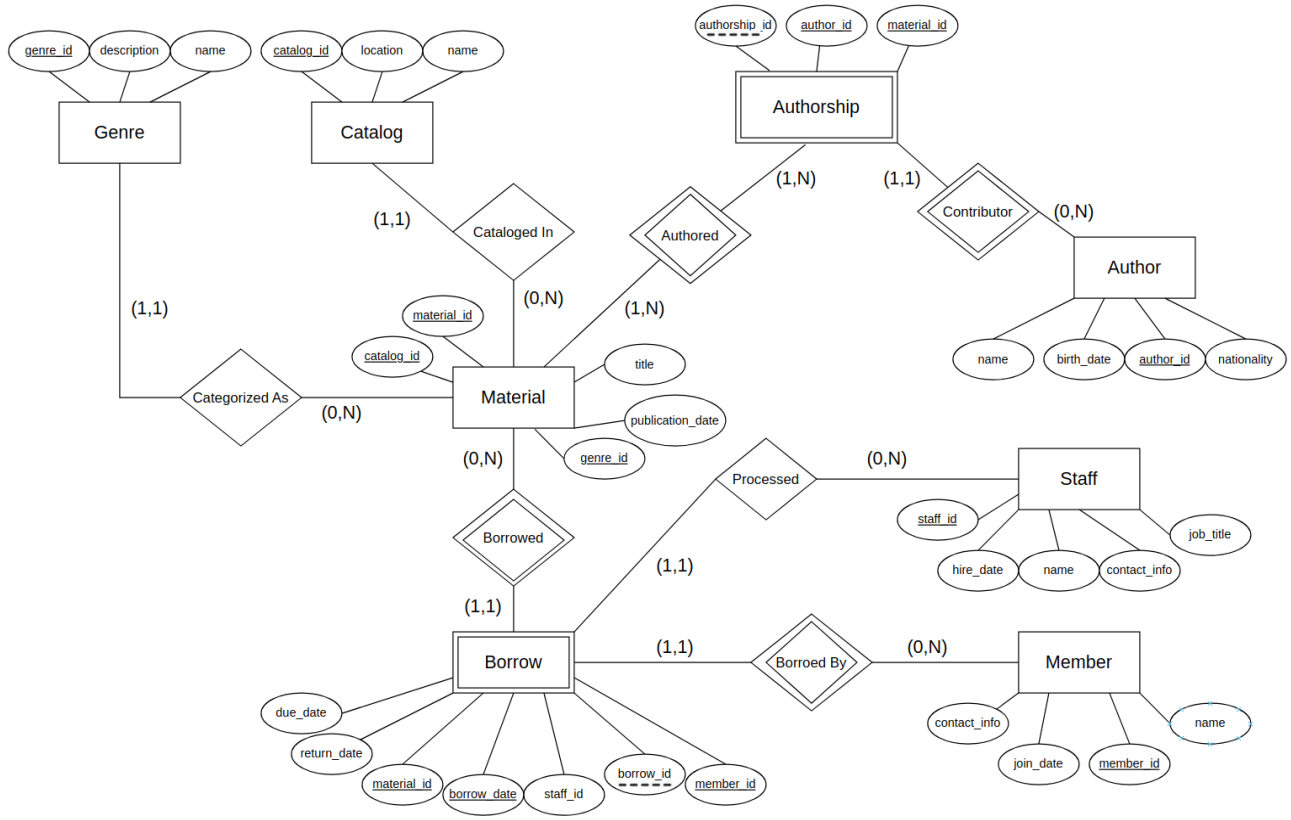


Figure 1: Entity Relation Diagram

1.3. Relational Schema Mapping

Much of the mapping was explained in the previous section.

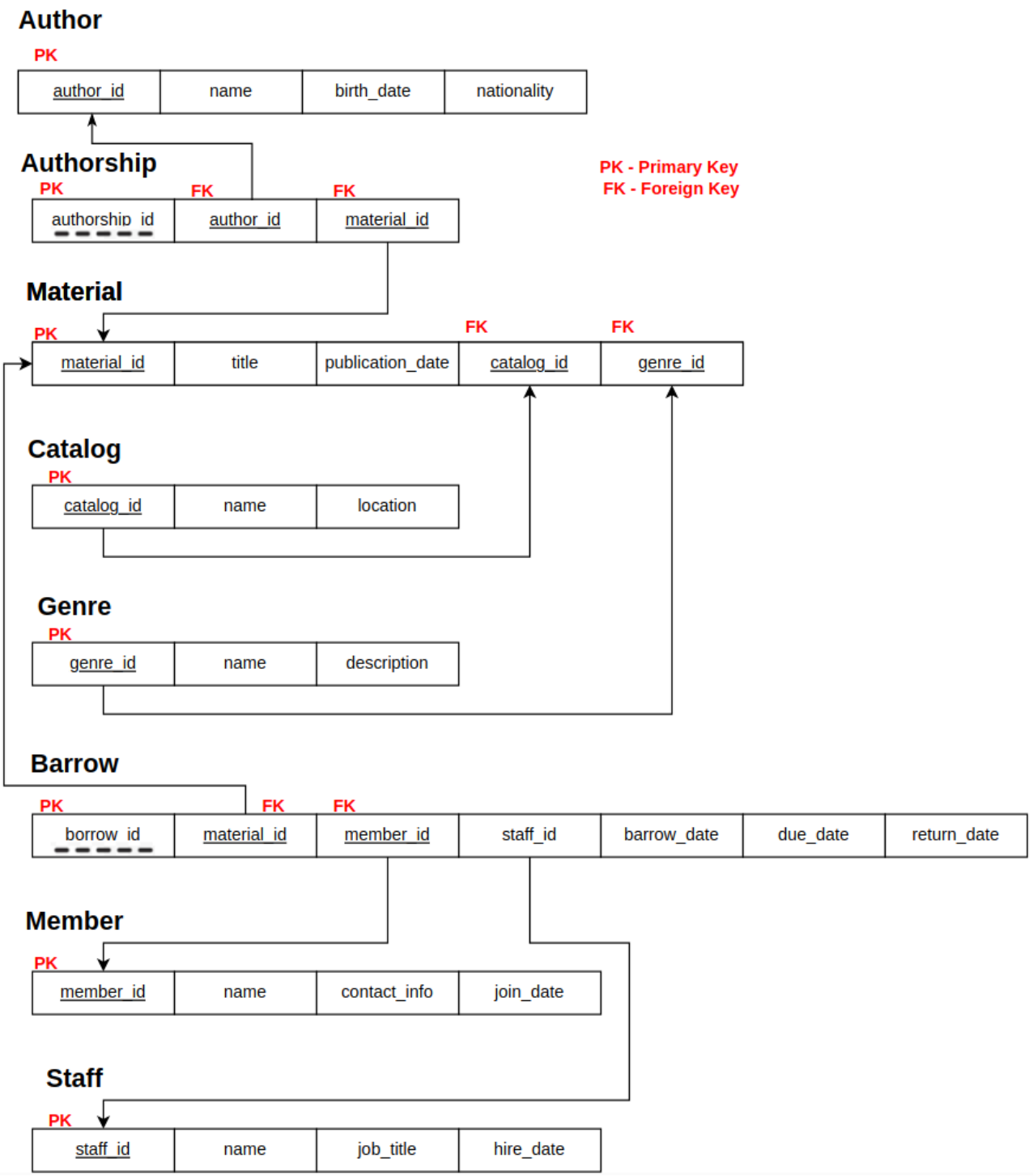


Figure 2: Relational Mapping

2. Database Implementation

This section explains the implementation of the database schema. Figure 3 shows a sample part of the code for the creation of the Borrow table. Its attributes, data types, primary key, foreign keys and constraints are all listed below. The creation of the actual database itself is also shown in the upper left. Sample initialization data was inserted using the Copy From constructs from the csv files that were provided. On the right you can see the output of the sample data for Borrow. The rest of

the tables were created in a similar fashion and omitted to avoid redundancy. See code for more details.

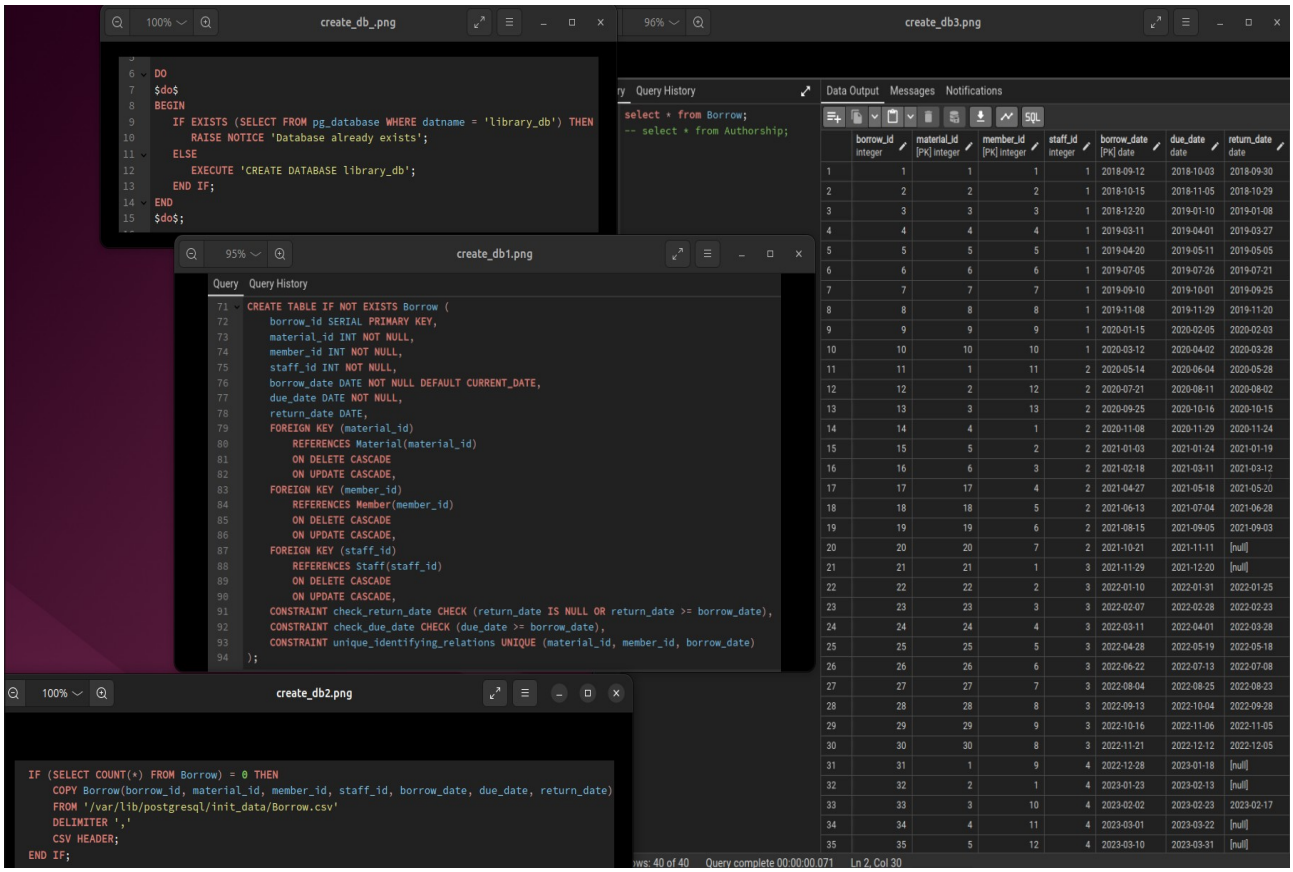


Figure 3: DB Creation

3. Querying and Manipulation

The following are common activities (queries, views, and stored procedures) to perform tasks, such as searching, updating, inserting and deleting library records from the database. These activities are part of the requirements for the database. To aid in readability the code that was run will be text highlighted with all other commented out in green. Some descriptions are omitted as heading and or query were self explanatory or there was no need to elaborate on assumptions made.

3.1. Listing what materials are currently available in the library

In Figure 4 a simple select query can be run against the Material table to obtain the desired results. A view called available_materials has also been created for convenience (see code). The assumptions here are that a material is considered unavailable if it has been borrowed and not yet returned. Not yet returned here is interpreted as a material is contained in the Borrow table and the return_date is Null.

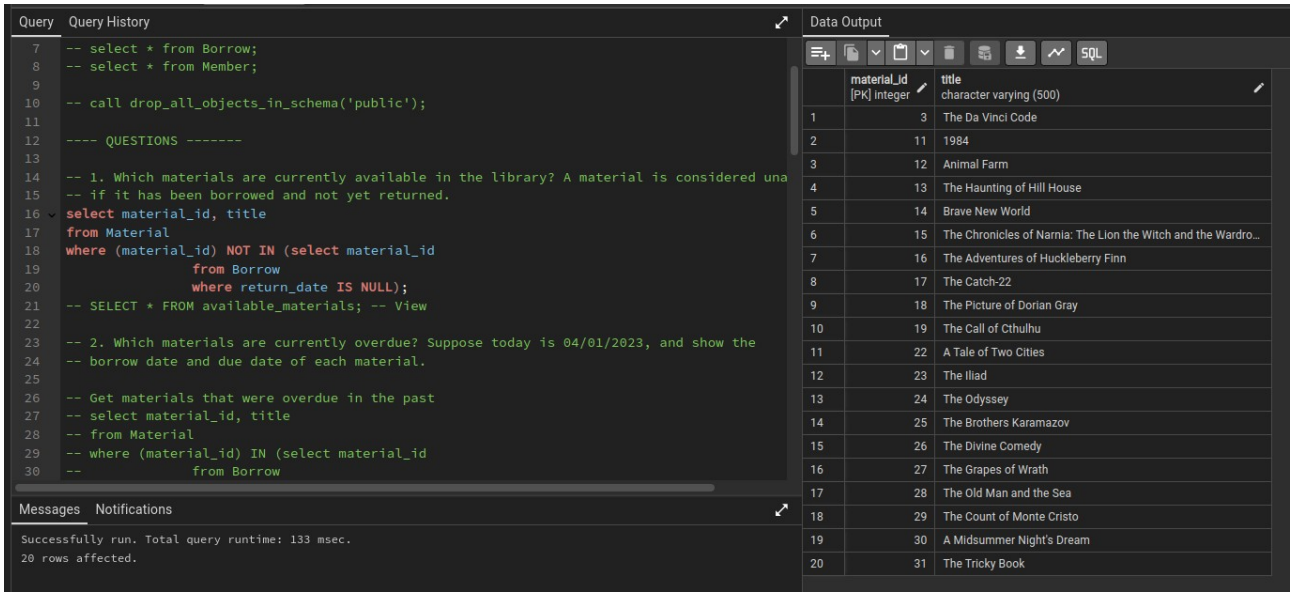


Figure 4: available materials

3.2. Listing which materials are currently overdue or were overdue in the past.

This section explores two different activities namely what materials were overdue on an arbitrary date and which materials are currently overdue.

3.2.1. Materials overdue based on determined date

For Figure 5 we have set a date variable today_date that is then referenced in the following select query. This query ignores any date beyond the determined date and selects materials that meet the criteria for being overdue. This method seems to be less error prone and easier than using the between construct or doing string manipulation.

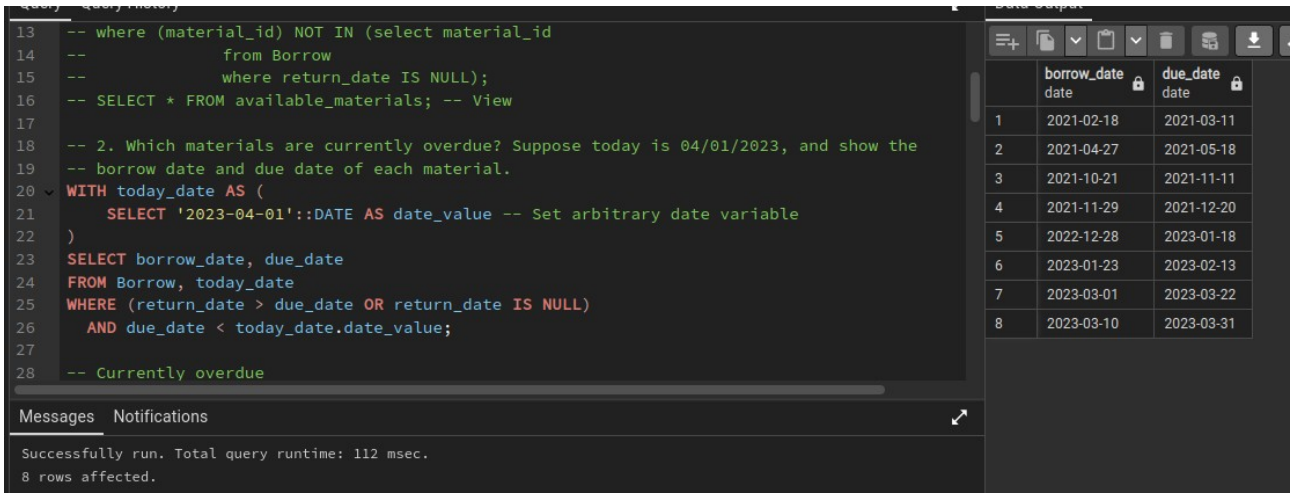


Figure 5

3.2.2. Materials currently overdue

For this activity we demonstrate the usage of a View. The query for the view is listed above in green. See code for how this view was implemented.

```
Query  Query History
19  -- borrow date and due date of each material.
20  -- WITH today_date AS (
21  --     SELECT '2023-04-01'::DATE AS date_value -- Set arbitrary date variable
22  -- )
23  -- SELECT borrow_date, due_date
24  -- FROM Borrow, today_date
25  -- WHERE (return_date > due_date OR return_date IS NULL)
26  --   AND due_date < today_date.date_value;
27
28  -- Currently overdue
29  -- SELECT borrow_date, due_date
30  -- FROM Borrow
31  -- WHERE return_date IS NULL;
32  SELECT * FROM currently_overdue; -- View
33
```

Data Output

	borrow_date date	due_date date
1	2021-10-21	2021-11-11
2	2021-11-29	2021-12-20
3	2022-12-28	2023-01-18
4	2023-01-23	2023-02-13
5	2023-03-01	2023-03-22
6	2023-03-10	2023-03-31
7	2023-03-15	2023-04-05
8	2023-03-25	2023-04-15
9	2023-03-30	2023-04-20
10	2023-03-26	2023-04-16
11	2023-03-28	2023-04-18

Messages Notifications

```
Successfully run. Total query runtime: 105 msec.
11 rows affected.
```

Figure 6: Materials currently overdue

3.3. Showing top 10 most borrowed materials in the library

```
Query  Query History
32  -- SELECT * FROM currently_overdue; -- View
33
34  -- 3. What are the top 10 most borrowed materials in the library? Show the title of each
35  -- material and order them based on their available counts.
36  select m.title, count(m.title)
37  from Borrow as b, Material as m
38  where b.material_id = m.material_id
39  group by b.material_id, m.title
40  limit 10;
41  -- select * from show_top_10_materials; --View
```

Data Output

	title character varying (500)	count bigint
1	The Catcher in the Rye	3
2	To Kill a Mockingbird	3
3	The Da Vinci Code	3
4	The Hobbit	3
5	The Shining	3
6	Pride and Prejudice	3
7	The Great Gatsby	2
8	Moby Dick	2
9	Crime and Punishment	2
10	The Hitchhiker's Guide to the Galaxy	2

Messages Notifications

```
Successfully run. Total query runtime: 152 msec.
10 rows affected.
```

Figure 7: Top 10 Materials

3.4. Showing how many materials an author has written

In this case we show the author Lucas Piki.

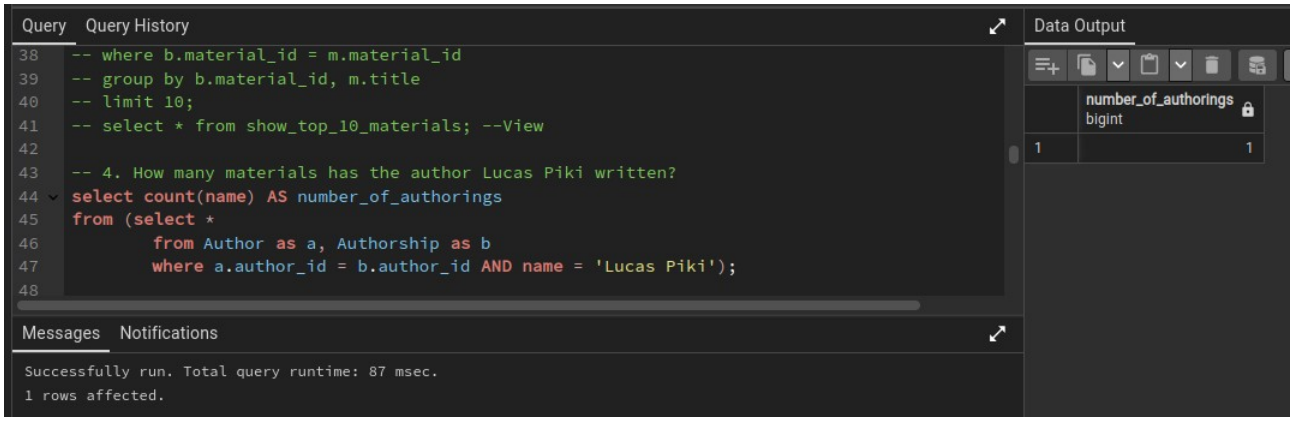
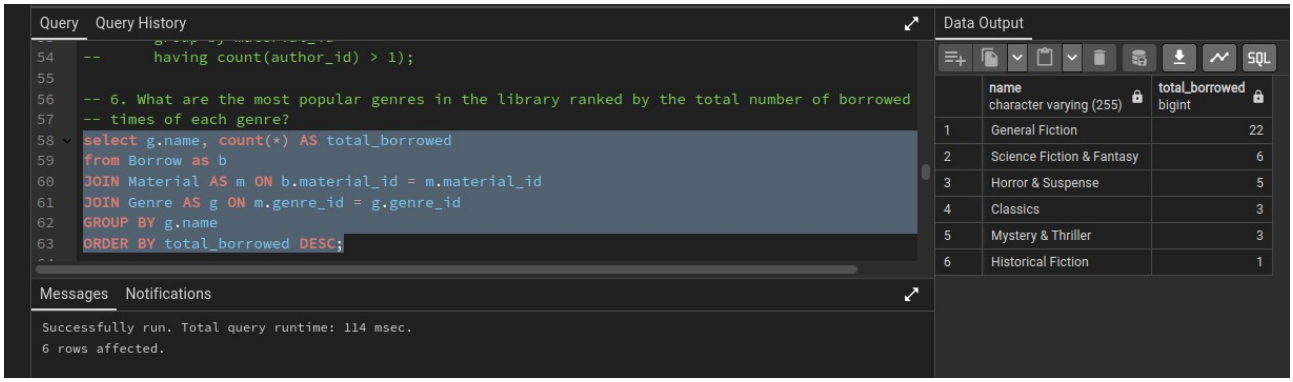


Figure 8

3.5. Materials were written by two or more authors

Figure 9

3.6. Most popular genres in the library ranked by the total number of borrowed times of each genre



3.7. Number of materials borrowed from 09/2020-10/2020

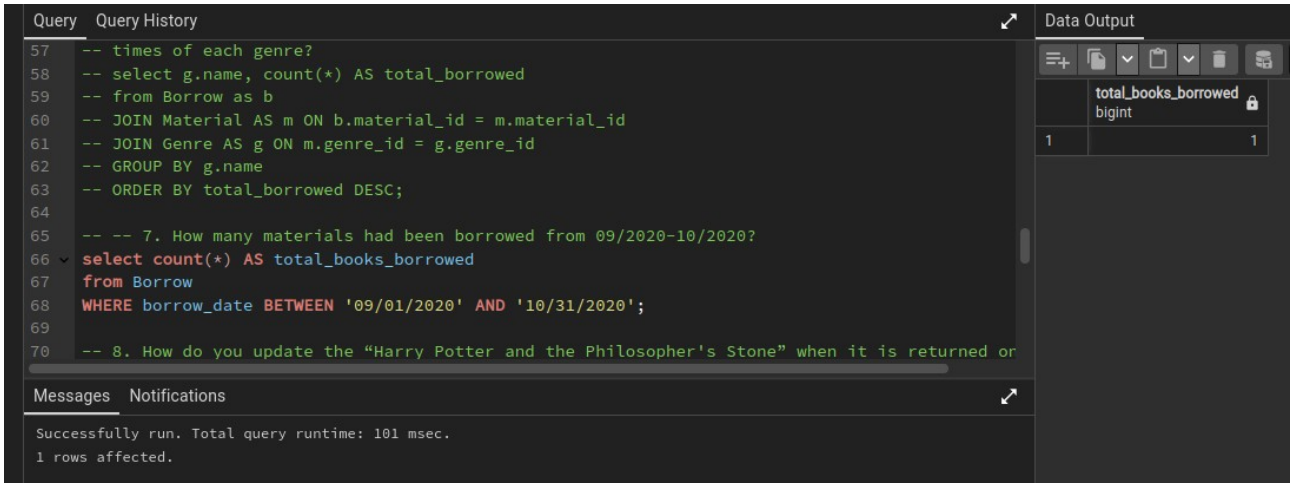


Figure 10

3.8. Update return date

For this activity we want to update a return date in the Borrow table given material title. The first query is what we see before the update is executed and the following query and output show what happens after the update is completed. Since this is such a useful activity I created a stored procedure called update_return_date that takes a title and date returned and executes the query below. See code for the procedure implementation. In Figure 11 you will see material_id 20. This id is associated with the title below. We use this in the select query for convenience to demonstrate that the update did what we wanted.

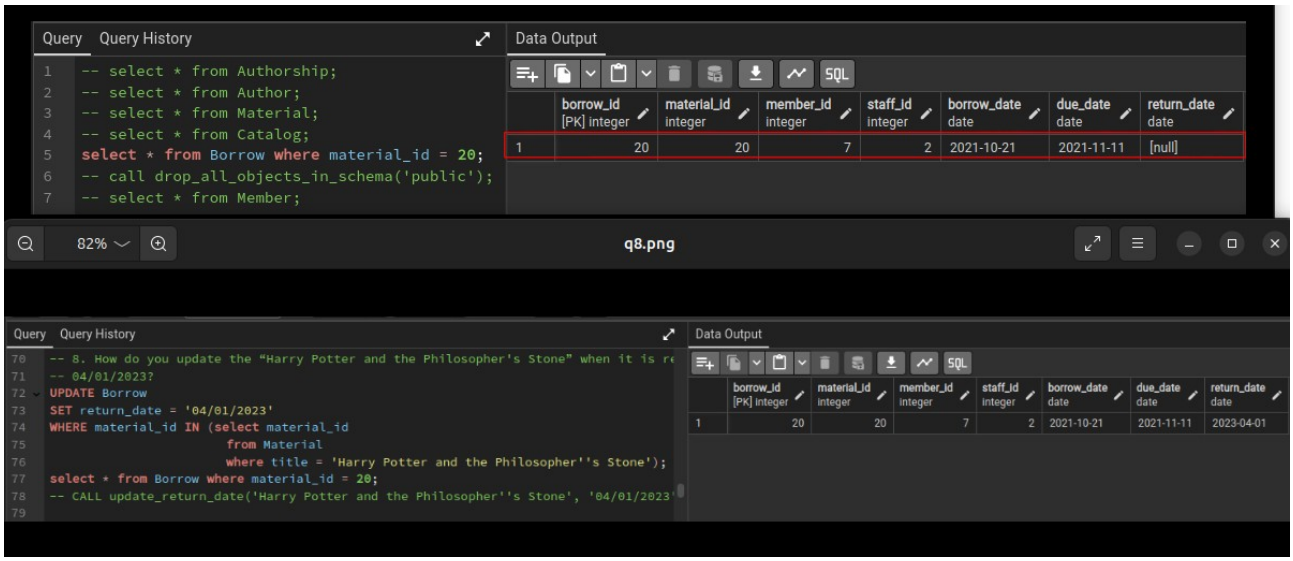


Figure 11: Update return date

3.9. Deleting a member and all their associated records

Figure 12 show the before and after action of deleting a member form the Member table. The first part shows the Member table with Emily Miller. Here member_id is 5 so we use that to find any associated records in Borrow. In Borrow we see that she has 3 records so these will need to be checked post deletion to ensure they are deleted. The last two parts show the delete command removing Emily Miller from the Member table and her associated records in Borrow. Because the Borrow and Member schema have foreign key constraint with an on delete cascade action we can delete records and have their associated records deleted with them.

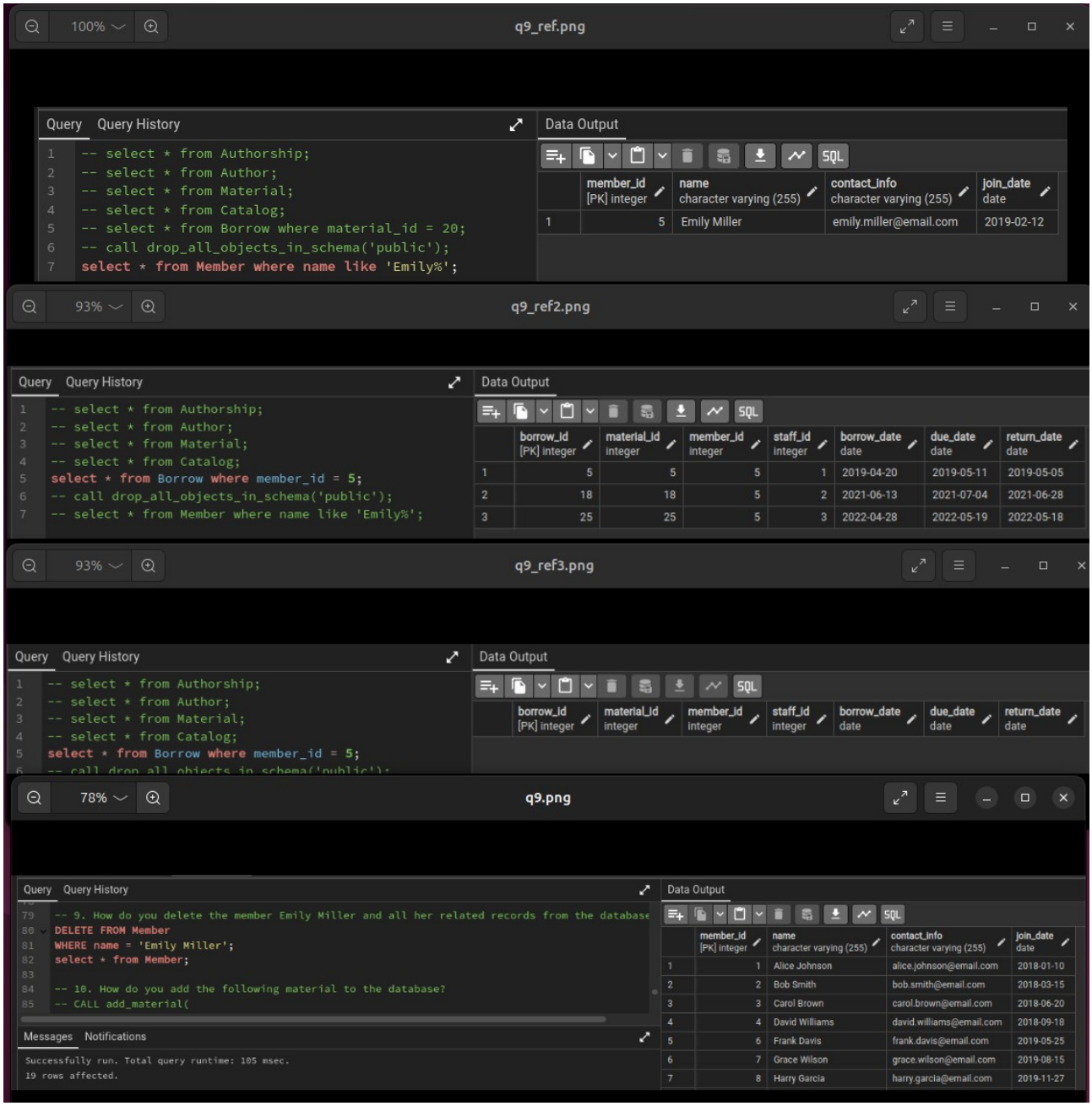


Figure 12: Deleting a member

3.10. Adding new materials

This part is much more complected and will be hard to show all the code via a screen shot. I'll do my best to explain the process and show what is essential. For more details on how this works please refer to the code and in particular the stored procedure for add_material. Refer to Figure 13 for the essential code implementation details and Figure 14 for execution and output details.

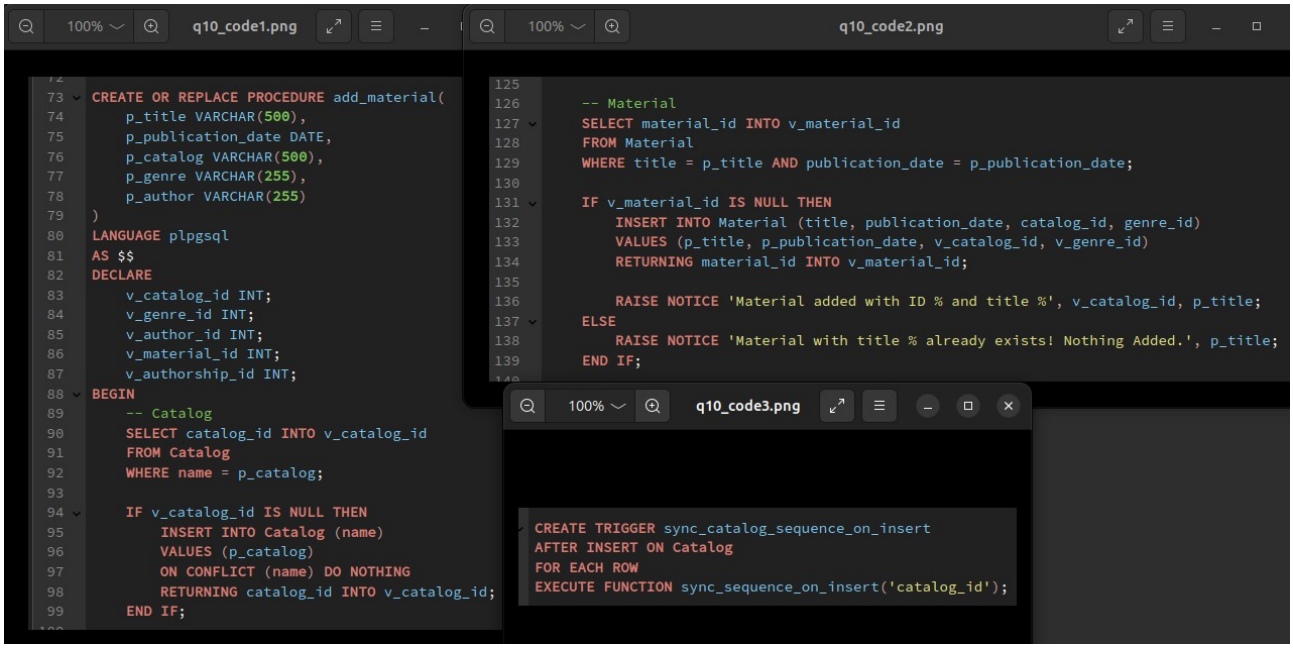


Figure 13: Add material code

To the left we have procedure creation which takes parameters (indicated as p_*) that were designated in the requirements. Anything with v_* are to store variables during processing. The code only shows the insert for catalog but much of the other tables were done in the same way. Basically if a new genre is entered then it will update the Genre table along with its keys if not then it will do nothing. The Material table is also shown in the upper right as does a similar process. Triggers were created to synchronize the key sequences for insertions. There might be a better way of doing this potentially with indexes but this is what solution I found that seems to work. Essentially for every insertion for said table it finds the max of the current key (an integer) and increments the next key value by one. There may be issues with this implantation (potentially spurious results for non-normalized schema) but for now it will suffice.

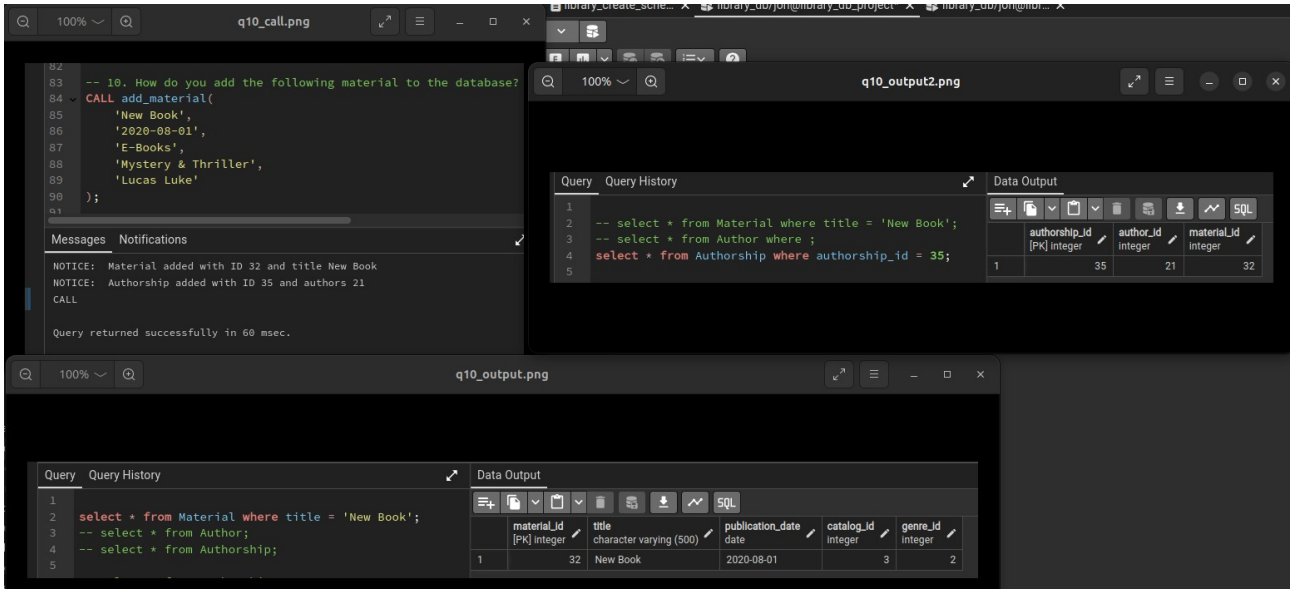


Figure 14: Add material execution and output

Figure 14 shows the execution of the procedure and the associated outputs. The NOTICE logs show what new or existing IDs for the keys and the new title name for convenience. As you can see the Materiel and Authorship tables have been updated correctly.

4. Design of Extended Features

For the design of the extended features a Membership table with the following attributes and constraints was created:

- **status –** STRING{active, deactivated} default is active
- **overdue_occurrences** – INT (no greater than 3) default 0
- **fee_paid** – BOOL{True, False, Null}default is Null
- Foreign Key INT (**member_id**) initialized from current members acts as primary key

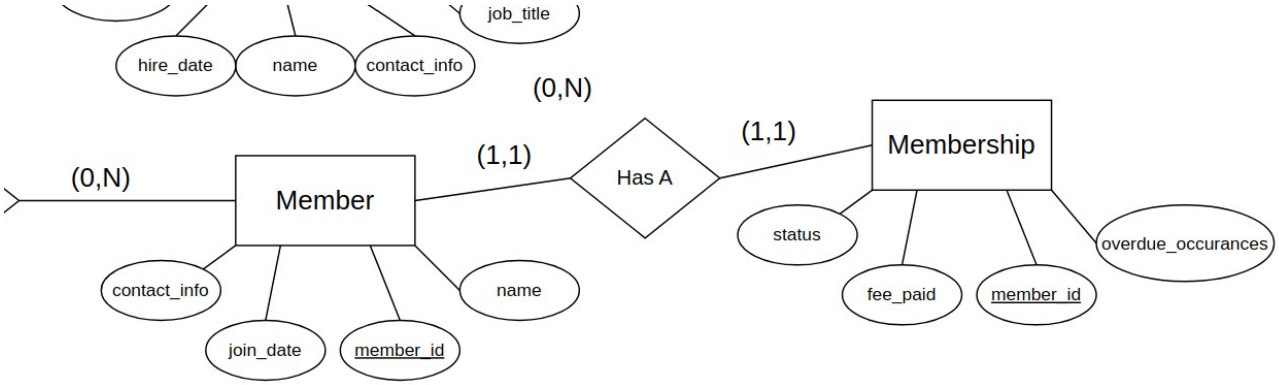


Figure 15: ERD for Extended Features

The following figure shows the creation of the table and populating with the default members:

79% extra_create_membershi...

```
-- Membership
CREATE TABLE IF NOT EXISTS Membership (
    member_id INT PRIMARY KEY,
    status VARCHAR(12) DEFAULT 'active' CHECK (status IN ('active', 'deactivated')),
    overdue_occurrences INT DEFAULT 0 CHECK (overdue_occurrences BETWEEN 0 AND 3),
    fee_paid BOOLEAN DEFAULT NULL,
    FOREIGN KEY (member_id) REFERENCES Member(member_id)
);
```

100% extra_default_membe...

	member_id [PK] integer	status character varying (12)	overdue_occurrences integer	fee_paid boolean
1	1	active	0	[null]
2	2	active	0	[null]
3	3	active	0	[null]
4	4	active	0	[null]
5	5	active	0	[null]
6	6	active	0	[null]
7	7	active	0	[null]
8	8	active	0	[null]
9	9	active	0	[null]
10	10	active	0	[null]
11	11	active	0	[null]
12	12	active	0	[null]
13	13	active	0	[null]
14	14	active	0	[null]
15	15	active	0	[null]
16	16	active	0	[null]
17	17	active	0	[null]
18	18	active	0	[null]
19	19	active	0	[null]
20	20	active	0	[null]

Figure 16: Membership Creation

4.1. Daily alerts on overdue material

There are more professional ways of doing this but if you want a quick and easy way to setup an alert system you can just use stored procedures, a bash script (or Python, Perl, whatever) and a cron job. Later when you know what ecosystem you are working then you can determine what kind of logging you would like to establish.

- Since both of the extended features are related a stored procedure called process_overdue_materials was created. Focusing on the alerting feature of the procedure it checks the Borrow table to see if it meets the criteria for an overdue item. If the criteria are met then the member_id and material_id are returned as a temporary table. This table can then be passed down stream as a data object (JSON, XML, etc. or what ever the scripting language requires or API will allow) to a software program (script) that then sends it to some kind of email. Since Staff table already has contact info we can inform a lead staff member using their current contact info. This part of the feature has been omitted.
- Create a script (in Python or Bash) that:
    - Makes a connection to the DB (perhaps establishing a session or client server relation)
    - The script will then run the stored procedure noted above on a predetermine cadence and gather the data (member_id, material_id)
    - If the results are not empty then have the results sent to selected email as an alert. Another query could be run to fetch the appropriate email from the Staff table and be used as a variable that could be passed in a query string (string interpolation in Python for example).
    - Then a call to a function that actually sends the alert message or payload.
- Lastly, create a cron job to run the script daily (maybe early morning). Note that currently the script is limited to being run once daily. If you run it multiple times it will keep incrementing current offenders until they hit 3. This is something I could work on and fix later but don’t have time. Just be aware of this limitation. The procedure should only run once per day anyway.

```sql
-- Extra Features
CREATE OR REPLACE FUNCTION process_overdue_materials()
RETURNS TABLE(member_id INT, material_id INT) AS $$
BEGIN
    RETURN QUERY
    SELECT b.member_id, b.material_id
    FROM Borrow as b
    WHERE b.return_date IS NULL
      AND b.due_date < CURRENT_DATE;

    UPDATE Membership
    SET overdue_occurrences =
        CASE
            WHEN fee_paid = TRUE THEN 0
            WHEN overdue_occurrences < 3 THEN overdue_occurrences + 1
            ELSE overdue_occurrences
        END,
        -- Set active or deactivated status
        status =
        CASE
            WHEN fee_paid = TRUE THEN 'active'
            WHEN overdue_occurrences + 1 >= 3 THEN 'deactivated'
            ELSE status
        END,
        -- Reactivate member as needed
        fee_paid =
            CASE
                WHEN fee_paid = TRUE THEN NULL
                ELSE fee_paid
            END
    WHERE Membership.member_id IN (
        SELECT b.member_id
        FROM Borrow as b
        WHERE b.return_date IS NULL
          AND b.due_date < CURRENT_DATE
    );
END;
$$ LANGUAGE plpgsql;
```

Figure 17: Extra Feature Procedure

```
1  -- select * from Membership;
2  SELECT * FROM process_overdue_materials();
```

	member_id integer	material_id integer
1	7	20
2	1	21
3	9	1
4	1	2
5	11	4
6	12	5
7	13	6
8	17	7
9	8	8
10	9	9
11	20	10

Figure 18: Currently overdue

4.2. Auto deactivate members based on number of overdue occurrence and reactivate after fee is paid.

This extended feature will use the same procedure used above. The key part of this feature is doing conditional checks an altering the Membership table as needed. These conditions include:

- Increment the overdue_occurrences column in the Membership_Status table when an overdue event happens in Borrow. Any members that have overdue_occurrences => 3 have their status set to deactivated.
- If a member pays the fee the fee_paid column will be set to True and their status is set to active again. The default is Null. Null values are not check. fee_paid set to False when members are deactivated.

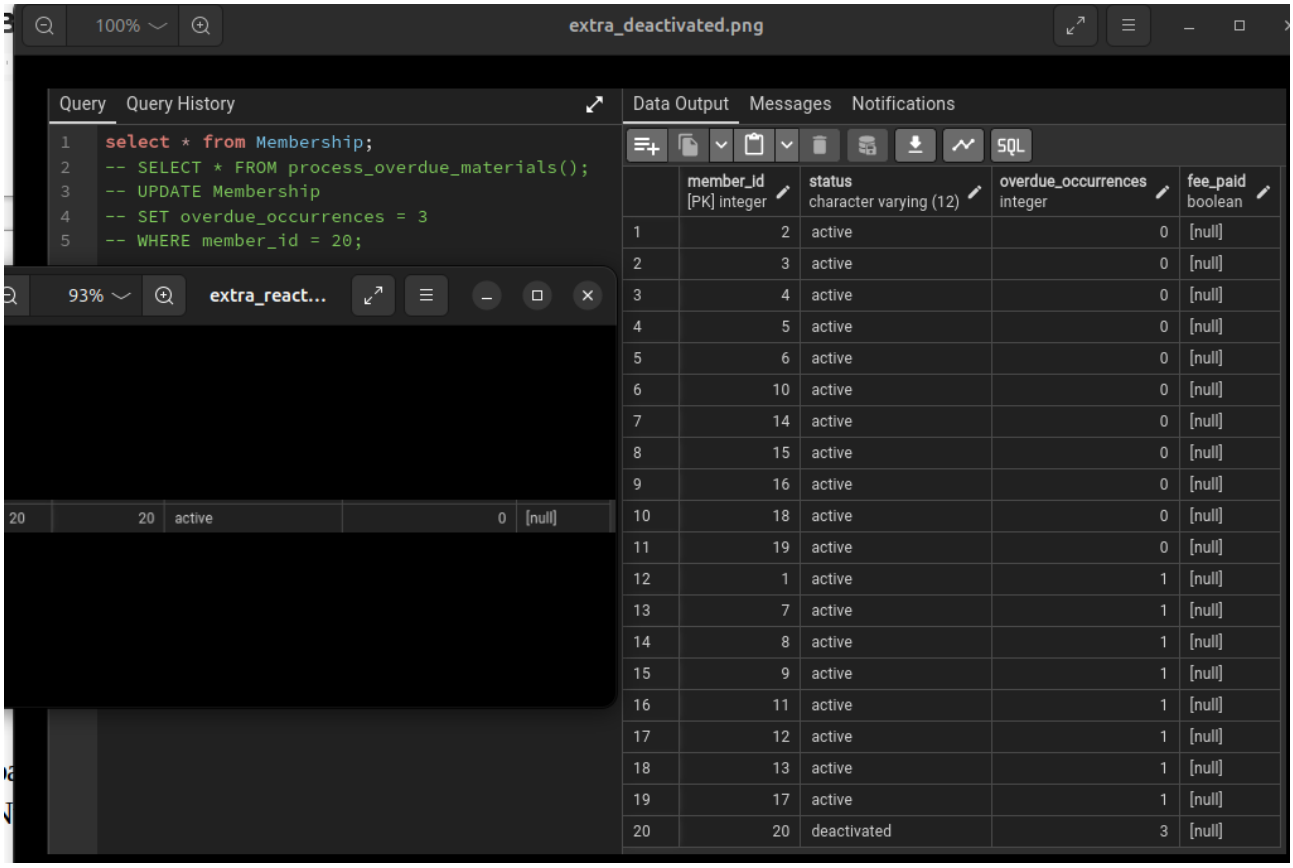


Figure 19: Extra Feature Output

In Figure 19 to the right is the table for Membership after the procedure is run once with the exception that I altered the table to have member_id 20 set to have overdue occurrences to 3 which made this member deactivated. When I set their fee_paid to True the member's status was then set to active again. See the middle left picture for this.