



CAIN'S JAWBONE

RELATIONAL DATABASE

Daisy Kopycienski

CMPT308

May 2022

EXECUTIVE SUMMARY

Cain's Jawbone is a murder mystery puzzle book from 1934 where the 100 pages are printed out of order. The book is accompanied by a competition where the reader's job is to arrange the 100 pages in the correct order and solve the 6 murders (6 murderer/victim pairs) in the story. The pages are printed with "incorrect" page numbers from 1-100 to identify which page is which.

I was contacted to build a relational database for a wealthy potential solver of the book. They were having trouble keeping track of which characters and events happened on which pages, as well as additional details such as poem segments, puzzles, and the murders. I was tasked to create tables to organize and keep track of the data relating to past and current competitions, solvers, and book details (pages, characters, murders etc.). To create a working database, I was presented with a handful of pages from the book, which I used as sample data.







TABLE OF CONTENTS

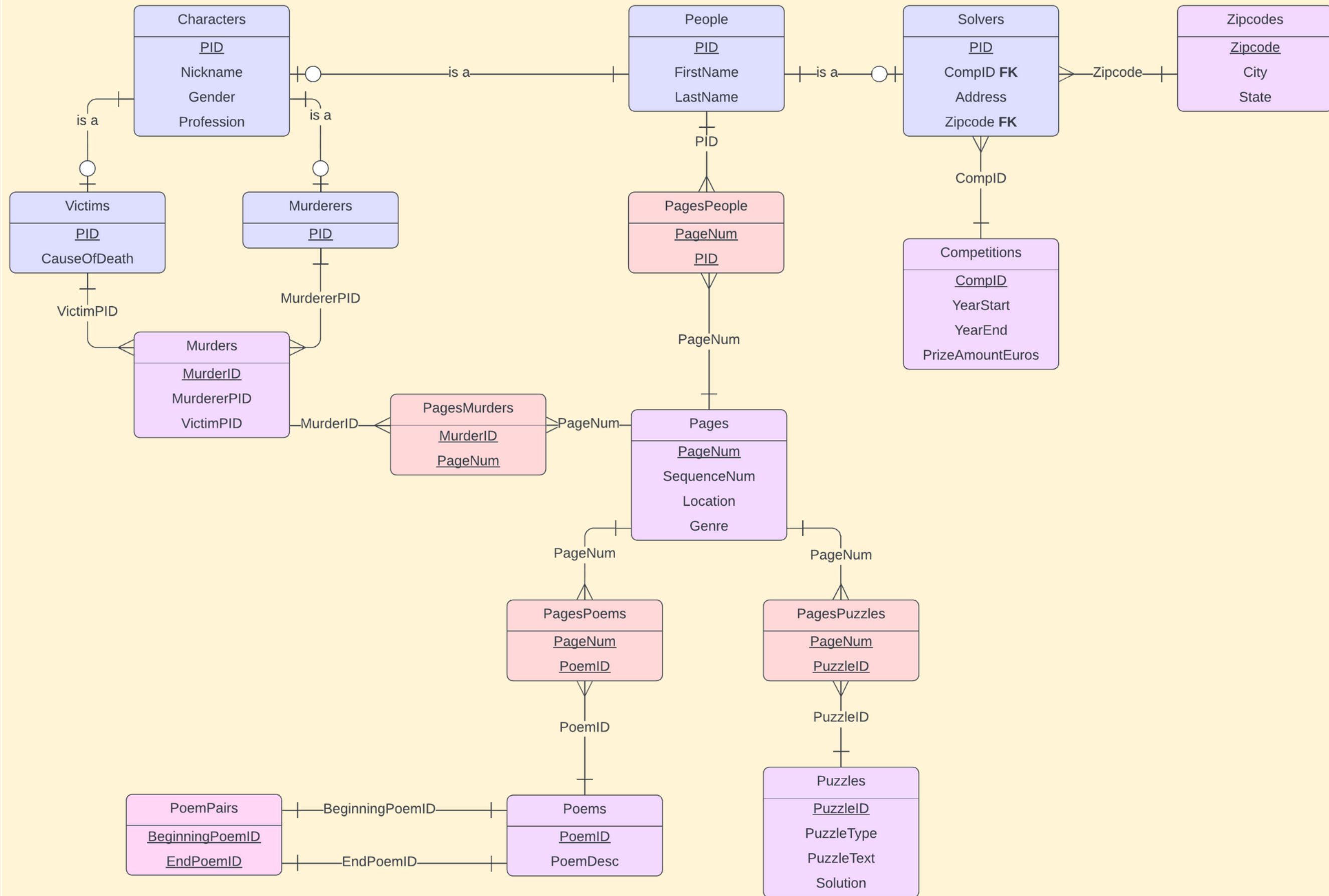
Entity Relationship Diagram:	4
Tables:	5
Views:	22
Reports:	32
Stored Procedures:	36
Triggers:	41
Security:	44
Implementation Notes:	46
Known Problems:	47
Future Enhancements:	47

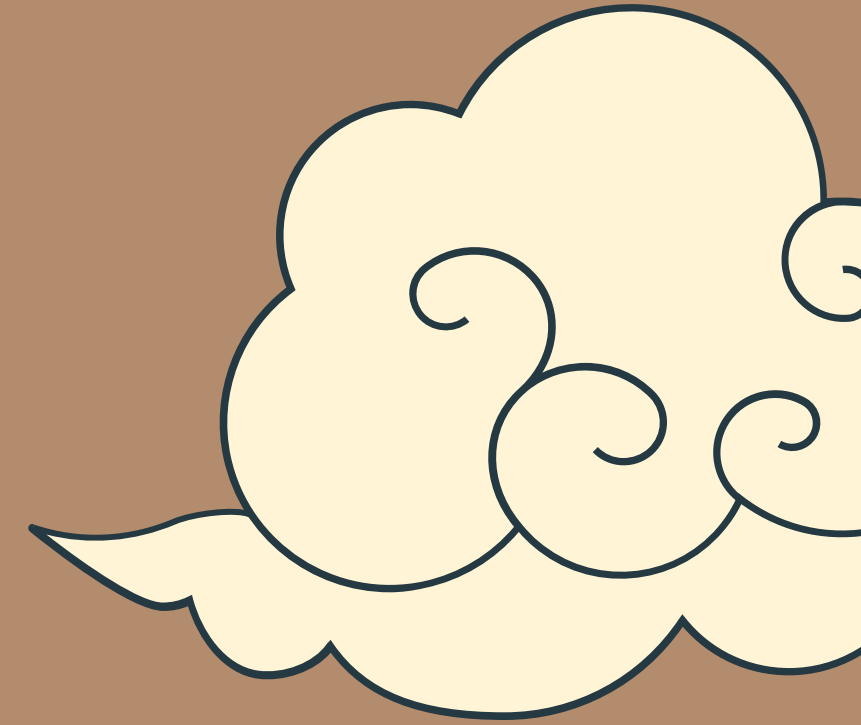
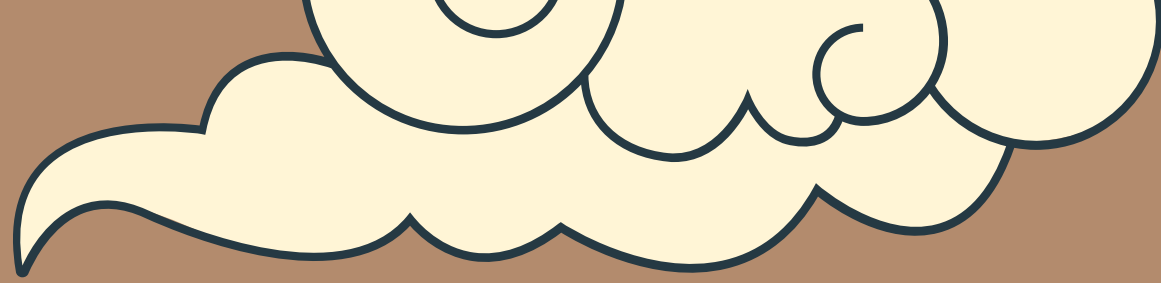


ENTITY-RELATIONSHIP DIAGRAM

Cain's Jawbone

Daisy Kopyciensi | May 5, 2022 CMPT308





TABLES:



People

This table stores the full name of every person in the database (both real and fictional)

Create Statement:

```
CREATE TABLE People (  
  PID int NOT NULL,  
  FirstName text,  
  LastName text,  
  PRIMARY KEY(PID)  
);
```

Functional Dependency:

PID → FirstName, LastName

TestData:

pid [PK] integer	firstname text	lastname text
1	Samantha	Turner
2	William	Kennedy
3	Charles	Day
4	Francis	Ferdinand
5	John	Finnemore
6	Henry	Peterson
7	Alan	Labouseur
8	William	Smith
9	Katherine	Snyder
10	Trinder	Adams
11	Olivia	Adams

Characters

This table is a sub-type of the People table and stores data specific to the fictional characters in the book

TestData:

Create Statement:

```
CREATE TABLE Characters (  
  PID int NOT NULL references People(PID),  
  Gender char(1) CHECK (Gender in ('M', 'F')),  
  Nickname text,  
  Profession text,  
  PRIMARY KEY(PID)  
);
```

pid [PK] integer	gender character (1)	nickname text	profession text
3	M	Charlie	Professor's Assistant
4	F	[null]	Doctor
5	M	[null]	Poet
8	M	Bills	Race Car Driver
9	F	Kate	Jockey
10	F	[null]	Elementary School Teacher
11	F	Olive	Scientist

Functional Dependency:

PID → Gender, Nickname, Profession

Solvers

This table is a sub-type of the People table and store data specific to the real people who solved the puzzle and won a competition





Create Statement:

```
CREATE TABLE Solvers (  
  PID int NOT NULL references People(PID),  
  CompID int NOT NULL references Competitions(CompID),  
  Address text,  
  Zipcode numeric(5) references Zipcodes(Zipcode),  
  PRIMARY KEY(PID)  
);
```

Functional Dependency:

PID → CompID, Address, ZipCode

TestData:

pid [PK] integer 	compid integer 	address text 	zipcode numeric (5) 
1	1	3731 Saratoga Place	90706
2	1	1200 Bel Air Road	90077
5	2	45 Reade Place	12602
7	3	3399 North Road	12601

Zipcodes

This table relates to the zipcodes in the Solver table and stores the city and state data that accomponies them


Create Statement:

```
CREATE TABLE Zipcodes (  
  Zipcode numeric(5) NOT NULL,  
  City text,  
  State text,  
  PRIMARY KEY(Zipcode)  
);
```

Functional Dependency:

Zipcode → City, State

TestData:

zipcode [PK] numeric (5) 	city text 	state text 
90706	Santa Fe Springs	California
90077	Los Angeles	California
12602	Poughkeepsie	New York
12601	Poughkeepsie	New York

Murderers

This is a sub-type of the People table and the Characters table and keeps track of the characters which are also murderers


Create Statement:

```
CREATE TABLE Murderers (  
  PID int NOT NULL references People(PID),  
  PRIMARY KEY(PID)  
);
```

Functional Dependency:

PID →

TestData:

pid [PK] integer 
3
4
9
11

Victims

This is a sub-type of the People table and the Characters table and keeps track of the characters which are also murder victims



Create Statement:

```
CREATE TABLE Victims (  
  PID int NOT NULL references People(PID),  
  CauseOfDeath text,  
  PRIMARY KEY(PID)  
);
```

Functional Dependency:

PID → CauseOfDeath

TestData:

pid [PK] integer 	causeofdeath text 
4	Poison
8	Firearm Homicide
10	Poison

Murders

This is a strong entity which records murderer/victim pairs and identifies them with a unique number

Create Statement:

```
CREATE TABLE Murders (  
  MurderID int NOT NULL references Murders(MurderID),  
  VictimPID int NOT NULL references Victims(PID) UNIQUE,  
  MurdererPID int NOT NULL references Murderers(PID),  
  PRIMARY KEY(MurderID)  
);
```

Functional Dependency:

MurderID → VictimPID, MurdererPID

TestData:

murderid [PK] integer	victimpid integer	murdererpid integer
1	10	11
2	8	4
3	4	9

PagesMurders

This table connects the murders with the page or pages they occurred in



Create Statement:

```
CREATE TABLE PagesMurders (  
  PageNum int NOT NULL references Pages(PageNum),  
  MurderID int NOT NULL references Murders(MurderID),  
  PRIMARY KEY(PageNum, MurderID)  
);
```

Functional Dependency:

MurderID, PageNum →

TestData:

pagenum [PK] integer 	murderid [PK] integer 
23	1
34	1
74	2
23	3

Competitions

This is a strong entity that records the competition details that occur for solving the book

Create Statement:

```
CREATE TABLE Competitions (  
  CompID int NOT NULL,  
  YearStart numeric(4) CHECK (YearStart <= YearEnd),  
  YearEnd numeric(4) CHECK (YearEnd >= YearStart),  
  PrizeAmountEuros numeric(6,2),  
  PRIMARY KEY(CompID)  
);
```

TestData:

compid [PK] integer	yearstart numeric (4)	yearend numeric (4)	prizeamounteuros numeric (6,2)
1	1934	1935	15.00
2	2019	2020	1000.00
3	2021	2023	250.00

Functional Dependency:

CompID → YearStart, YearEnd, PrizeAmountEuros

Pages

This is a strong entity to record additional information on each page. Every out of order page is marked with a unique number 1-100 which serves as a natural primary key (PageNum). The sequence number is used to denote the "correct" page number in the sequence 1-100.

Create Statement:

```
CREATE TABLE Pages (  
  PageNum int NOT NULL CHECK (PageNum <= 100),  
  SequenceNum int CHECK (SequenceNum <= 100) UNIQUE,  
  Location text,  
  Genre text,  
  PRIMARY KEY(PageNum)  
);
```

Functional Dependency:

PageNum → SequenceNum, Location, Genre

TestData:

pagenum [PK] integer	sequencenum integer	location text	genre text
23	4	Peebles University	Murder
34	5	Peebles University	Murder
74	99	Staircase	Murder
1	88	Bedroom	Romance
99	51	[null]	Narration
98	89	Hallway	Romance

PagesPeople

This table connects the people in the book to what pages they appear in

Create Statement:

```
CREATE TABLE PagesPeople (  
  PageNum int NOT NULL references Pages(PageNum),  
  PID int NOT NULL references People(PID),  
  PRIMARY KEY(PageNum, PID)  
);
```

Functional Dependency:

PageNum, PID →

TestData:

pagenum [PK] integer	pid [PK] integer
23	10
23	11
34	10
34	11
74	8
74	4
23	4
23	9
74	3
1	3

Poems

This is a strong entity which assigns a unique key to each poem and contains a brief content description.


Create Statement:

```
CREATE TABLE Poems (  
  PoemID int NOT NULL,  
  PoemDesc text,  
  PRIMARY KEY(PoemID)  
);
```

Functional Dependency:

PoemID → PoemDesc

TestData:

poemid [PK] integer 	poemdesc text 
1	Rose
2	Viking ship
3	Flower garden
4	Ocean



PagesPoems

This table connects the poems with page numbers

Create Statement:

```
CREATE TABLE PagesPoems (  
  PageNum int NOT NULL references Pages(PageNum),  
  PoemID int NOT NULL references Poems(PoemID),  
  PRIMARY KEY(PageNum, PoemID)  
);
```

TestData:

pagenum [PK] integer 	poemid [PK] integer 
1	1
98	3
23	2
34	4

Functional Dependency:

PageNum, PoemID →



PoemPairs

Each poem in the book is split into two halves and spread over two separated pages. This table connects the first half of a poem with its end pair

Create Statement:

```
CREATE TABLE PoemPairs (  
  BeginningPoemID int NOT NULL references Poems(PoemID),  
  EndPoemID int NOT NULL references Poems(PoemID),  
  PRIMARY KEY(BeginningPoemID, EndPoemID)  
);
```

TestData:

beginningpoemid [PK] integer 	endpoemid [PK] integer 
3	1
2	4

Functional Dependency:

BeginningPoemID, EndPoemID →

Puzzles

This is a strong entity to record all of the puzzles within the book with their type and solutions. If a solution has not yet been found, a '?' character will be inserted by default.

Create Statement:

```
CREATE TABLE Puzzles (  
  PuzzleID int NOT NULL,  
  PuzzleType text NOT NULL,  
  PuzzleText text NOT NULL,  
  Solution text DEFAULT '?',  
  PRIMARY KEY(PuzzleID)  
);
```

TestData:

puzzleid [PK] integer	puzzletype text	puzzletext text	solution text
1	Spoonerism	The Lord is a shoving leopard	The Lord is a loving shepherd
2	Riddle	What has many words but never speaks?	a book
3	Riddle	What question can you never answer yes to?	?

Functional Dependency:

PuzzleID → PuzzleType, PuzzleText, Solution

PagesPuzzles

This table connects the puzzles to its page or pages, as sometimes the puzzle is on one page and its solution is on another



Create Statement:

```
CREATE TABLE PagesPuzzles (  
  PageNum int NOT NULL references Pages(PageNum),  
  PuzzleID int NOT NULL references Puzzles(PuzzleID),  
  PRIMARY KEY(PageNum, PuzzleID)  
);
```

Functional Dependency:

PageNum, PuzzleID →

TestData:

pagenum [PK] integer 	puzzleid [PK] integer 
74	1
99	2
1	3
23	2

VIEWS



View 1: MurderDetails

This view provides additional character data for each murder/victim pair

```
CREATE VIEW MurderDetails
AS
Select ms.MurderID, ms.victimPID as VPID, p1.firstname as
VFirstName, p1.lastname as VLastName, c1.nickname as VNickname,
c1.gender as VGender, c1.profession as VProfession, v.causeofdeath,
ms.murdererpid as MPID, p2.firstname as MFirstName, p2.lastname as
MLastName, c2.nickname as MNickName, c2.gender as MGender,
c2.profession as MProfession
FROM
Murders ms inner join victims v on ms.victimpid = v.pid
        inner join murderers m on ms.murdererpid = m.pid
        inner join characters c1 on ms.victimpid = c1.pid
        inner join characters c2 on ms.murdererpid = c2.pid
        inner join people p1 on ms.victimpid = p1.pid
        inner join people p2 on ms.murdererpid = p2.pid
ORDER BY ms.murderid ASC;
```

View 1: MurderDetails

Query 1: All Information (the v stands for victim and the m stands for murderer)

```
select *  
from murderDetails
```




murderid integer	vpid integer	vfirstname text	vlastname text	vnickname text	vgender character (1)	vprofession text	causeofdeath text
1	10	Trinder	Adams	[null]	F	Elementary School Teacher	Poison
2	8	William	Smith	Bills	M	Race Car Driver	Firearm Homicide
3	4	Francis	Ferdinand	[null]	F	Doctor	Poison

mpid integer	mfirstname text	mlastname text	mnickname text	mgender character (1)	mprofession text
11	Olivia	Adams	Olive	F	Scientist
4	Francis	Ferdinand	[null]	F	Doctor
9	Katherine	Snyder	Kate	F	Jockey

View 1: MurderDetails

Query 2: Names of victim and murderer for every murder in which both characters are female


```
select murderID, vfirstName, vlastname, mfirstname, mlastname
from murderDetails
where vgender = 'F' and mgender = 'F'
```

murderid integer 	vfirstName text 	vlastname text 	mfirstname text 	mlastname text 
1	Trinder	Adams	Olivia	Adams
3	Francis	Ferdinand	Katherine	Snyder

View 1: MurderDetails

Query 3: Full names for each victim/murderer pair with associated IDs. With a fully completed database, this query would return the 6 pairs of information necessary to solve the murder section of the competition.

```
select murderID, vfirstName, vlastname, mfirstname, mlastname  
from murderDetails
```

murderid integer 	vfirstName text 	vlastname text 	mfirstname text 	mlastname text 
1	Trinder	Adams	Olivia	Adams
2	William	Smith	Francis	Ferdinand
3	Francis	Ferdinand	Katherine	Snyder

View 2: PageDetails

This view provides all data connecting characters to pages they are on

Create view PageDetails

AS

```
select p.pageNum as PageNum, p.sequenceNum, p.location, p.genre, ppl.PID,
```

```
ppl.firstName, ppl.lastName
```

```
from PagesPeople pppl right outer join Pages p on pppl.pageNum = p.pagenum
```

```
left outer join People ppl on pppl.pid = ppl.pid
```

View 2: PageDetails

Query 1: All Information

```
select *  
from PageDetails
```

pagenum integer	sequencenum integer	location text	genre text	pid integer	firstname text	lastname text
23	4	Peebles University	Murder	10	Trinder	Adams
23	4	Peebles University	Murder	11	Olivia	Adams
34	5	Peebles University	Murder	10	Trinder	Adams
34	5	Peebles University	Murder	11	Olivia	Adams
74	99	Staircase	Murder	8	William	Smith
74	99	Staircase	Murder	4	Francis	Ferdinand
23	4	Peebles University	Murder	4	Francis	Ferdinand
23	4	Peebles University	Murder	9	Katherine	Snyder
74	99	Staircase	Murder	3	Charles	Day
1	88	Bedroom	Romance	3	Charles	Day
98	89	Hallway	Romance	[null]	[null]	[null]
99	51	[null]	Narration	[null]	[null]	[null]

View 2: PageDetails

Query 2: Sequence numbers from 1-100 with associated page number. With a fully completed database, this query will output the correct order of pages needed to submit for the competition.




```
select distinct sequencenum, pagenum  
from pageDetails  
order by sequenceNum ASC
```

sequencenum integer	pagenum integer
4	23
5	34
51	99
88	1
89	98
99	74

View 2: PageDetails

Query 3: Character's name and PID who is present in the highest number of pages



```
select Distinct pd.pid, pd.firstName, pd.lastname
from pageDetails pd
where pid in (select pd.pid
              from pageDetails pd
              group by pd.pid
              order by count(pd.pid) DESC
              limit 1)
;
```

pid integer		firstname text		lastname text	
	4	Francis		Ferdinand	

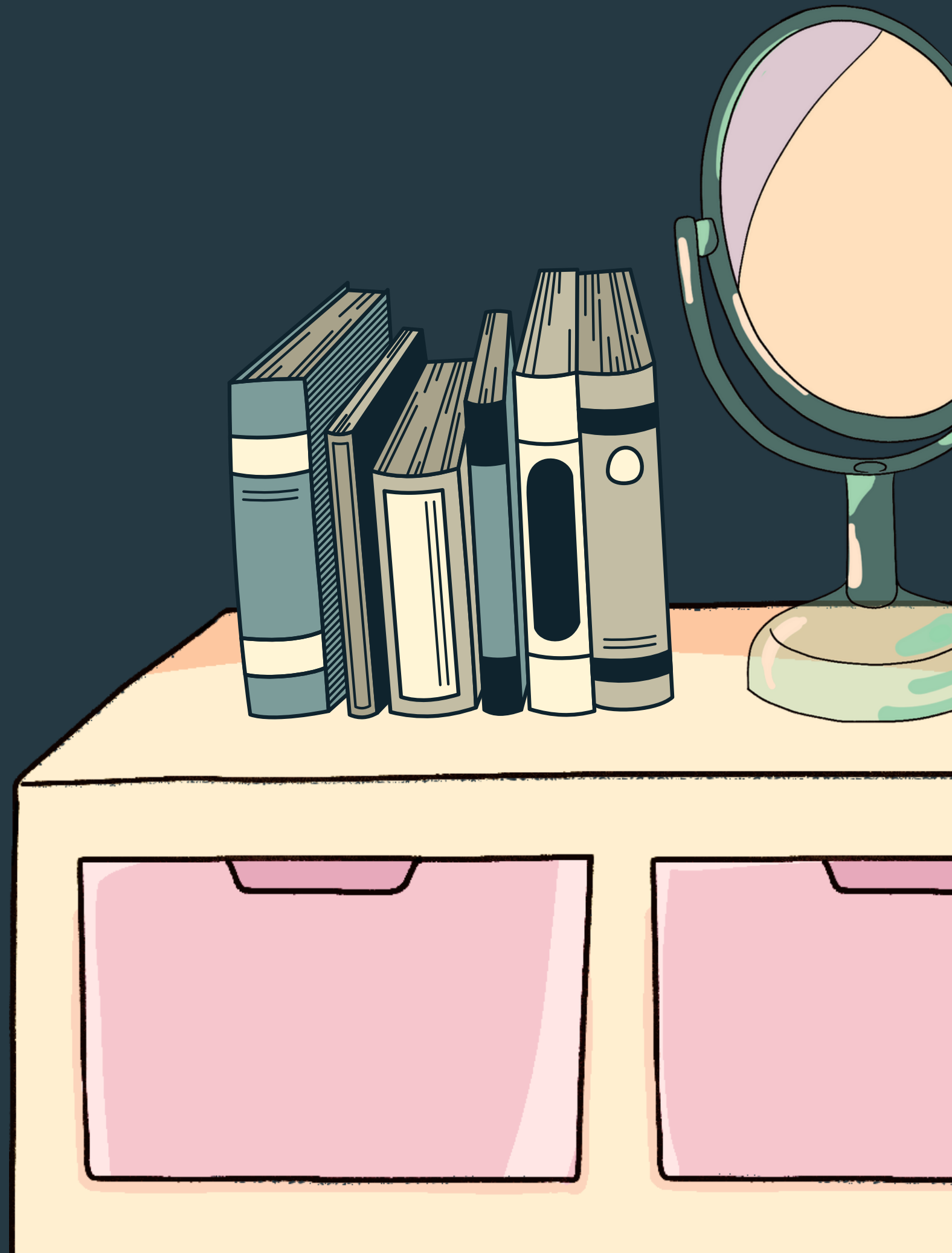
View 2: PageDetails

Query 4: All page numbers and locations which have no characters present

```
select pd.pageNum, pd.location  
from pageDetails pd  
where pd.PID is NULL
```

pagenum integer		location text	
	98	Hallway	
	99	[null]	



REPORTS



Report 1

Query: Returns page numbers and locations of murders using both of the views and the genre column



```
select distinct pageNum, location
from pageDetails pd left outer join murderDetails md1 on pd.pid = md1.Vpid
                    left outer join murderDetails md2 on pd.pid = md2.Mpid
where genre = 'Murder'
```

pagenum integer		location text	
23		Peebles University	
34		Peebles University	
74		Staircase	

Report 2

Query: Returns puzzle IDs and solutions of solved puzzles that are sharing pages with characters who have been to a university



```
select puzzleID, solution
from Puzzles
where Solution != '?' and
    puzzleID in (select puzzleID
                from PagesPuzzles
                where pageNum in (select pageNum
                                from pageDetails
                                where location like '%University'));
```

puzzleid [PK] integer 	solution text 
2	a book

Report 3

Query: Returns number of characters present at location and name of location in which there is a beginning half of a poem on the same page and more than one character at the location

```
select location, count(pid)
from pageDetails
where pagenum in (select pageNum
                  from pagespoems
                  where poemID in (select poemID
                                   from poems
                                   where poemID in (select beginningPoemID
                                                    from poemPairs)
                                   )
                  )
Group by location
having count(distinct pid) >= 2;
```

location text		count bigint	
Peebles University			4

STORED PROCEDURES




Stored Procedure 1

Returns every page a character is on. Input: PID for the character.

```
create or replace function PagesFor(int, REFCURSOR) returns refcursor as
$$
declare
  personID int := $1;
  resultset REFCURSOR := $2;
begin
  open resultset for
    select pageNum
    from PagesPeople
    where personID = PID;
  return resultset;
end;
$$
language plpgsql;
```

```
-- Example for person 11
select PagesFor(011, 'results');
Fetch all from results;
```

pagenum	
integer	
	23
	34

Valid Input Checks

The next section shows the stored procedure functions to ensure the inserts and updates done by my client and their assistant adhere to the rules of the puzzle book.

RULES:

1. Each sequence number from 1-100 can only be used exactly once, as this denotes the correct order of the pages and there can only be one page in one spot (Unique constraint).
2. It is impossible for one person to be a victim of more than one murder (Unique constraint).
3. A person can only be on a page if they are a character.
4. A poem can not be a beginning half and an end half, they have to be one or the other.
5. A poem can not be in more than one pair, because a poem only has one match.
6. There can not be more than 6 entries in the Murders table, because there are only 6 murders throughout the book.

Valid Input Checks

Checks to make sure that the person being inserted into the PagesPeople table is a Character (ensures that rule 3 is being followed).

```
CREATE OR REPLACE FUNCTION PersonOnPagelsCharacter()  
RETURNS TRIGGER AS $$  
BEGIN  
  IF new.pid not in(select ch.pid  
    from Characters ch  
    where (new.pid = ch.pid)  
  )THEN RAISE EXCEPTION 'Person needs to be a character';  
  END IF;  
  RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

Trigger results on slide 42

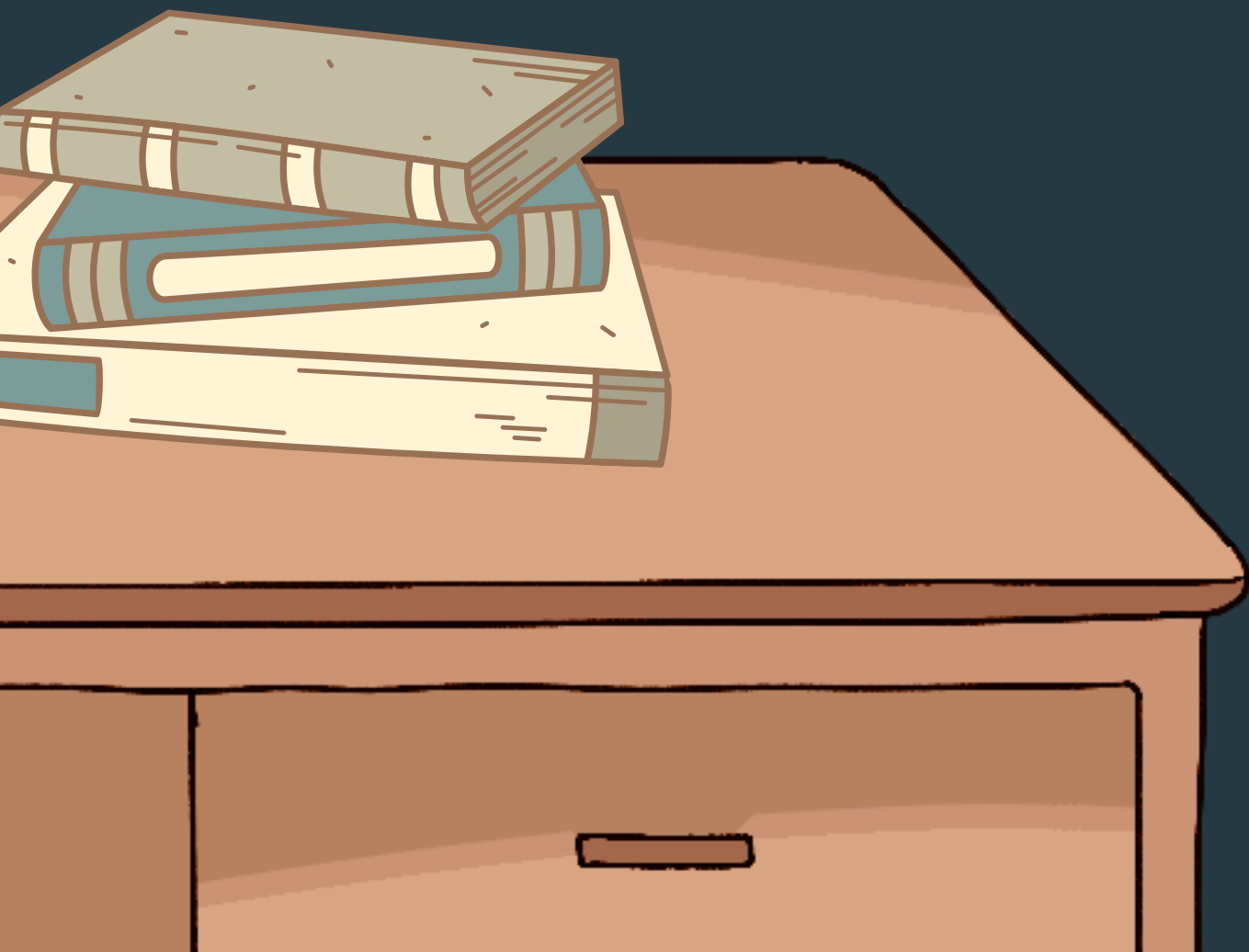
Valid Input Checks

Checks that there are not more than 6 entries in the Murders table, because there are only 6 murders throughout the book. (ensures that rule 6 is being followed).

```
CREATE OR REPLACE FUNCTION OnlySixMurders()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF (select count(MurderID)  
        from Murders  
        ) > 6  
    THEN RAISE EXCEPTION 'There are already 6 murders, can not add another.';  
    END IF;  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

Trigger results on slide 43

TRIGGERS



Trigger 1

```
CREATE OR REPLACE TRIGGER PersonOnPageIsCharacter  
BEFORE INSERT ON PagesPeople  
FOR EACH ROW  
EXECUTE PROCEDURE PersonOnPageIsCharacter();
```

```
insert into PagesPeople(pageNum, PID)  
values  
(23, 007);
```

Messages

```
ERROR:  Person needs to be a character  
CONTEXT:  PL/pgSQL function persononpageischaracter() line 6 at RAISE  
SQL state: P0001
```

Trigger 2

```
CREATE OR REPLACE TRIGGER OnlySixMurders  
AFTER INSERT ON Murders  
FOR EACH ROW  
EXECUTE PROCEDURE OnlySixMurders();
```

```
insert into Murders(MurderID, VictimPID, MurdererPID)  
values  
(007, 003, 011);
```

Messages

```
ERROR: There are already 6 murders, can not add another.  
CONTEXT: PL/pgSQL function onlysicsmurders() line 6 at RAISE  
SQL state: P0001
```

SECURITY

Admin: This role grants access to all aspects of the database (only for the database architect)

Client: This role grants access to the client who hired me, they do not get to edit the schema of the database, but can query, insert, and delete rows

Assistant: This role grants limited access to the client's assistant who is helping read through the pages and enter the data into the database. They do not need to query from it.



SECURITY

Admin:

```
create role admin;  
grant all  
on all tables in schema public  
to admin;
```

Client:

```
create role client;  
grant select, insert, update  
on all tables in schema public  
to client;
```

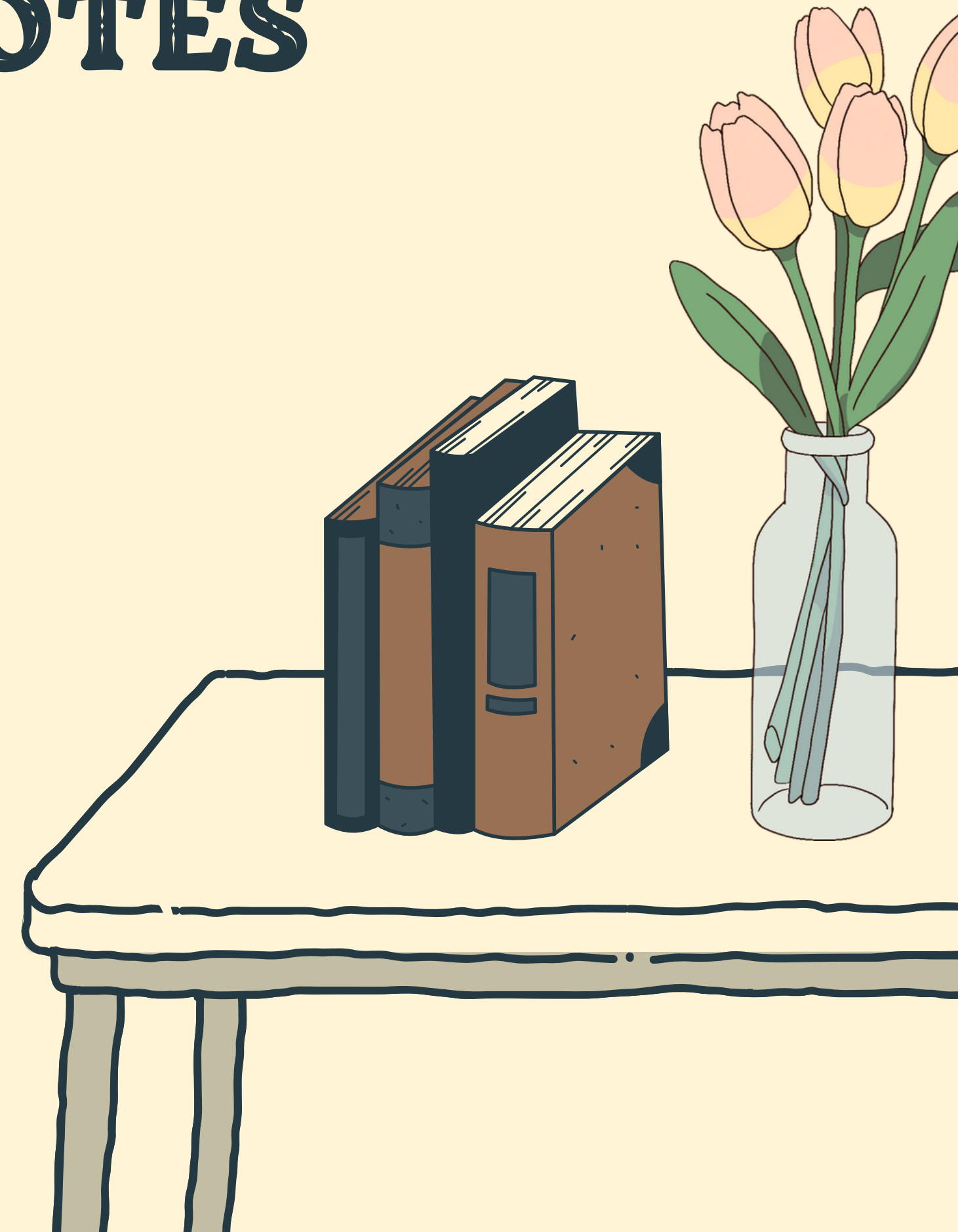
Assistant:

```
create role assistant;  
grant insert, update  
on all tables in schema public  
to assistant;
```



IMPLEMENTATION NOTES

- With a larger number of pages and sample data, the queries could return more interesting and complex information.
- In order for the database to achieve its goal of assisting my client with the puzzle, the entered data must be complete and accurate. Something the database is unable to check for and therefore, there is risk for human error.



KNOWN PROBLEMS

1. It would be difficult to assign a page with its correct sequence number without knowing the page before it and its sequence number. Even though this SequenceNum is a necessary data point, it seems arbitrary at this time.
2. As it is, without the trigger check, any person could be added to a page. This could be an issue as only characters should be on pages.
3. As we only used a small number of page data for the sample queries, the detail and information gained was not too extensive.
4. Some people suspect that one of the characters that appears in a large number of pages is actually a dog and not a person at all, but in our database would be saved in the People table which could cause problems.
5. The victims table is denoted as having a 1 to many relationship with the murders table, even though one victim can only be in one murder.

FUTURE ENHANCEMENTS

- To solve problem 1, it could be helpful to implement new columns in the Pages table that record the previous pageNum and the following pageNum. This would help chain the known pages together in their correct sequence.
- With a larger supply of example pages we may discover that there are more shared features other than just poems and puzzles. We could create more strong entities to record these details and more Pages___ tables to connect them with the page numbers.
- To address problem 4, we could implement an additional table, possibly a sub-type of characters, for pets/animals.
- Additional stored procedures and triggers could be implemented to ensure the rules around the Poems are followed (a poem can only be a beginning or an end, not both, and a poem can only be apart of one pair).

