



GROUP ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT069-3-3-DBS

Database Security

APD3F2202CS(DA), APU3F2202CS(CYB), APD3F2022IT(ISS)

NAME OF LECTURER: KULOTHUNKAN PALASUNDRAM

HAND OUT DATE: 15th JULY 2022

HAND IN DATE: 25th September 2022

WEIGHTAGE: 60%

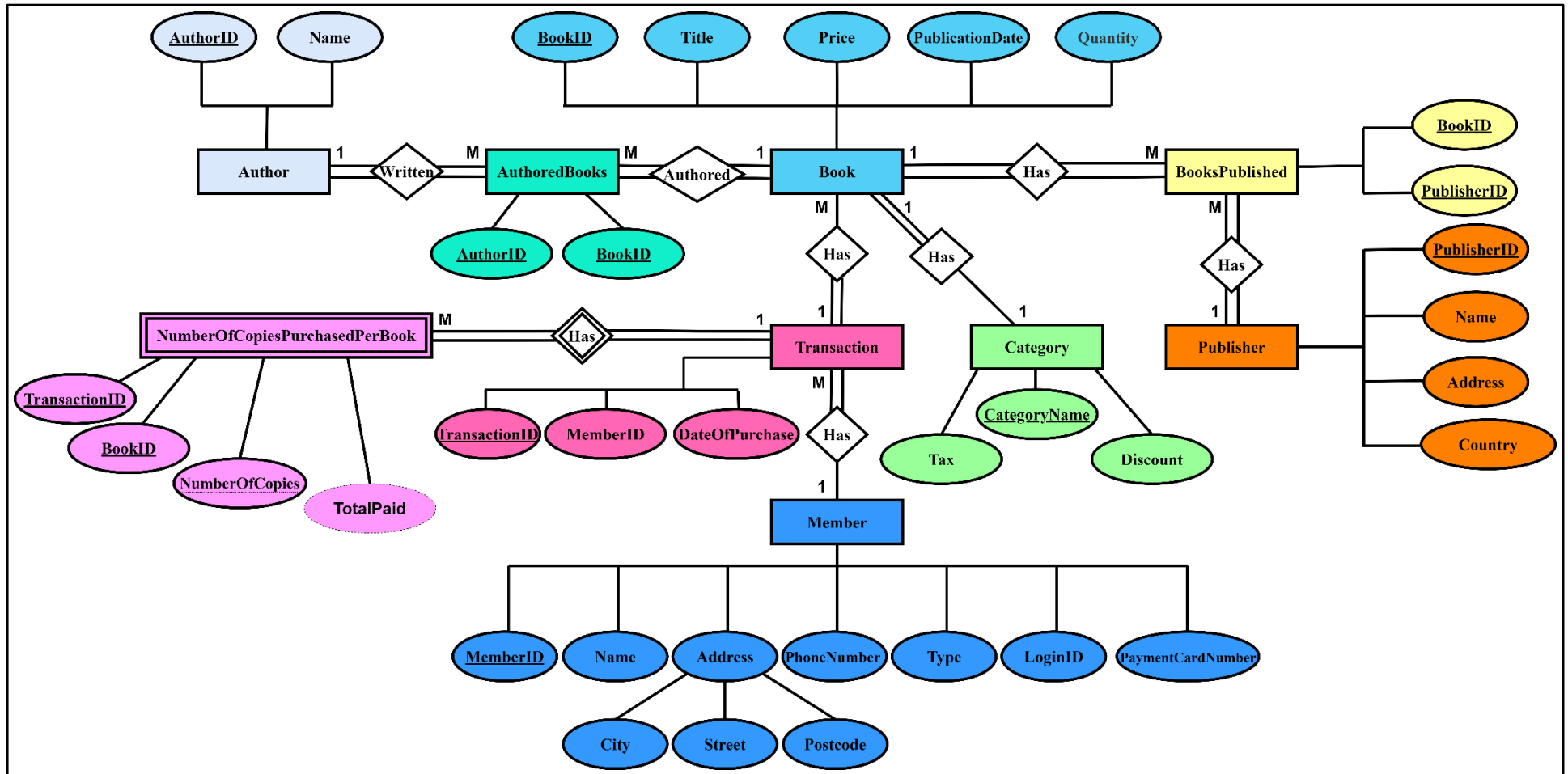
STUDENT NAME	TP-NUMBER	INTAKE
Mozhar Alhosni	TP058272	APD3F2202CS(CYB)
Tarun Aitha	TP058015	APD3F2022CS(DA)
Khirthiga A/P Chandrasekharan	TP058844	APD3F2022CS(CYB)
Haroon Gilani	TP058413	APD3F2022IT(ISS)

Table of Contents

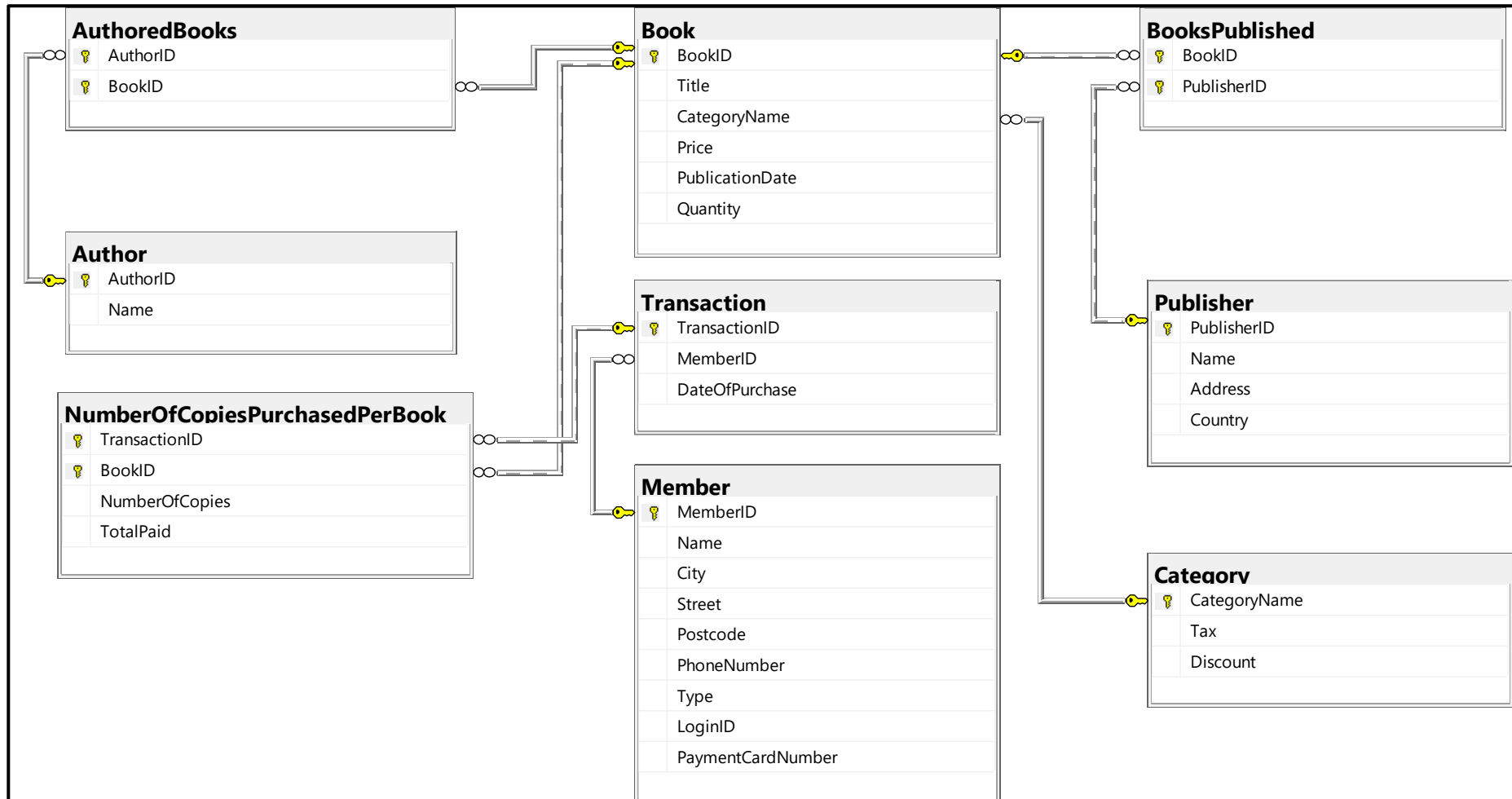
1.0 Entity Relationship Diagram.....	4
1.1 SQL Server Database Diagram	5
2.0 CIA – Confidentiality, Integrity, and Availability	6
2.1 Confidentiality	6
2.1.1 SQL-Logins and Users	6
2.1.2 Roles and Permissions	7
2.1.3 Row-Level Security	10
2.1.4 Encryption	12
2.2 Integrity	15
2.2.1 Inferential integrity checks in DDL Statements of	15
2.2.2 Auditing	16
2.2.3 Triggers	18
2.2.4 Views	19
2.3 Availability.....	20
2.3.1 Full Backups	21
2.3.2 Differential Backups	21
2.3.3 Transaction Log Backups.....	22
2.3.4 Tail Log Backups	22
3.0 Backup and Restore Strategy	22
4.0 Key Learnings	24
4.1 Mozhar Alhosni – TP058272	24

4.2 Tarun Aitha (TP058015)	25
4.3 Khirthiga A/P Chandrasekharan (TP058844)	25
4.4 Haroon Gilani (TP058413)	25
References	26

1.0 Entity Relationship Diagram



1.1 SQL Server Database Diagram



2.0 CIA – Confidentiality, Integrity, and Availability

2.1 Confidentiality

To achieve confidentiality for the database of APU's BookStore Private Limited (APUBSPLDB), a plethora of SQL syntax and features were used that, with their accompanying semantics and functionalities, respectively, provided access to data within the database to only authorized users/entities.

2.1.1 SQL-Logins and Users

First, five SQL-logins were created with difficult, randomly generated passwords, and their (the SQL-logins) default database was set to "APUBSPLDB":

```
-- Start: Create Logins --  
CREATE LOGIN [PeterCarter] WITH PASSWORD='iFT3n7a1Z1INQj6rPqyJxmdvXtqomm8M', DEFAULT_DATABASE=APUBSPLDB;  
CREATE LOGIN [RajRaj] WITH PASSWORD='2j1S3M80MwS9RKzAy5oig1rEAEEx29BuK', DEFAULT_DATABASE=APUBSPLDB;  
CREATE LOGIN [AdamMuhammad] WITH PASSWORD='oke4FufIT2wY09A1Ttn2Dozh8wZeTF24', DEFAULT_DATABASE=APUBSPLDB;  
CREATE LOGIN [IanStirk] WITH PASSWORD='cmYJZGky5M536cN031U55gSpDZ20HZQP', DEFAULT_DATABASE=APUBSPLDB;  
CREATE LOGIN [DaudSalim] WITH PASSWORD='fX4kcpnec14HjnaIHp61t89k7ZFfxo0v', DEFAULT_DATABASE=APUBSPLDB;  
-- End: Create Logins --
```

Firstly, setting difficult, randomly generated passwords for the logins prevents threat actors from penetrating the database in case they launched a brute force attack against these SQL-logins; for example, a threat actor may run the following Nmap command that uses the Lua-script named "ms-sql-brute" to brute force passwords of SQL-logins:

"nmap -p 1433 --script ms-sql-brute IPAddressOfTarget"

Secondly, setting the default database to "APUBSPLDB" for the SQL-logins serves two purposes: it follows the principal of least privilege since it disallows authorized users/entities to have access to assets that they do not require to complete their work, and it also disallows threat actors from laterally moving within the SQL Server instance environment, in case they were able to crack/brute force the passwords of the SQL-logins (this will be more coherent when users are created and granted permissions that only provide them access to the specified database in the upcoming section, because the semantics of the "DEFAULT_DATABASE" option only lands the users on the specified database on login, however, it does not disallow them from changing to other databases within the SQL server).

However, this setting of passwords in plaintext introduces a risk of threat actors gaining access to the backend and reading the SQL queries, thus yielding having passwords useless. One approach to solving this problem/risk can be that instead of supplying the plaintext password, the SQL server allows supplying the hash of the password (thus, the end user would provide his password, it gets hashed, and the hash is used instead), or another approach is encrypting the database and its SQL commands files to prevent exposure of plaintext passwords (this will be performed when encryption is mentioned in the upcoming sections).

After creating these five SQL-logins, they are enabled so that the users of the database instance can log in to it:

```
-- Start: Enable Logins
ALTER LOGIN [PeterCarter] ENABLE;
ALTER LOGIN [RajRaj] ENABLE;
ALTER LOGIN [AdamMuhammad] ENABLE;
ALTER LOGIN [IanStirk] ENABLE;
ALTER LOGIN [DaudSalim] ENABLE;
-- End: Enable Logins
```

After creating the SQL-logins, each is mapped/contained to a user:

```
-- Start: Create Users --
CREATE USER [PeterCarter] FOR LOGIN [PeterCarter];
CREATE USER [RajRaj] FOR LOGIN [RajRaj];
CREATE USER [AdamMuhammad] FOR LOGIN [AdamMuhammad];
CREATE USER [IanStirk] FOR LOGIN [IanStirk];
CREATE USER [DaudSalim] FOR LOGIN [DaudSalim];
-- End: Create Users --
```

SQL-logins entitles/grants the principal that it is created for access to the SQL Server's instance and databases within it, while *users* grant principals login entry into a single database. Thus, to follow the principle of least privilege, the five logins are initially assigned a default database, and now, with users created for them, they will be granted permissions according to their work/functions by adding them to user-defined roles. Granting permissions on the user level can be more granular than on the login level.

2.1.2 Roles and Permissions

Assigning users' roles is of utmost importance for the confidentiality of the database since it allows for the best of following for the principal of least privileges. Roles are created and granted their relevant permissions, and subsequently, users that belong to that role are added to it so that they inherit its permissions.

According to the business rules given by APU BookStore Private Limited, there are four main roles: A member's role, a book store administrator's role, a database administrator's role, and a management role. After creating the member's role, it is granted several permissions as per the business rules:

```
-- Start: Role for MembersRole --
CREATE ROLE [MembersRole];
GRANT SELECT ON [Member] TO MembersRole;
GRANT SELECT ON [Book] TO MembersRole;
GRANT UPDATE ON [Member](
    [Name],
    [City],
    [Street],
    [Postcode],
    [PhoneNumber],
    [PaymentCardNumberEncrypted]
) TO MembersRole;

GRANT SELECT ON dbo.UndeletedTransactionsView TO [MembersRole];
GRANT UPDATE, DELETE ON [Transaction] TO [MembersRole];
GRANT SELECT, UPDATE, DELETE ON [NumberOfCopiesPurchasedPerBook] TO [MembersRole];
-- End: Role for Members --
```

To achieve confidentiality, the member's role is only assigned to the permissions it requires, thus, this implies that any other permission that was not explicitly allowed for it will be denied, by default. Therefore, for example, if any user that belongs to the member's role tried to use any SQL statement on the "Book" table/relation other than "SELECT", the action/operation will fail, as that role does not have that permission assigned to it explicitly.

One important detail regarding the permissions of the member's role is that they are granted "SELECT" on the view named "UndeletedTransactionsView" instead of the table/relation "Transaction", this is so due to the ability of members to choose to delete their transactions, however, they are not deleted but instead, a bit column called "IsDeletedByUser" is set to 1 to denote they are no longer to be visible for the user. Another important detail that greatly enhances the confidentiality of the database is that when the member role is granted the ability to update their records within the "Member" table, they are only granted permission on specific columns, this is so to prevent them to alter columns that are incremented automatically, such as the member ID. This specific feature/permission can also be viewed under the umbrella of achieving *integrity*. Once the role is created, the relevant users now can be added to it:

```
-- Start: Assign User IanStirk to MembersRole --
ALTER ROLE [MembersRole] ADD MEMBER [IanStirk];
-- End: Assign User IanStirk to MembersRole --

-- Start: Assign User DaudSalim to MembersRole --
ALTER ROLE [MembersRole] ADD MEMBER [DaudSalim];
-- End: Assign User DaudSalim to MembersRole --
```

Now, whenever these two users/members log in (via their respective SQL logins), they will only be able to perform what they inherited from the "MembersRole", thus, this keeps the other data from different tables within the database confidential, disallowing unauthorized access to it from users having the "MembersRole".

The second role, according to the business rules, is the book store administrator's role:

```
-- Start: Role for Book Store Administrators --
CREATE ROLE BookStoreAdministratorsRole;
GRANT SELECT, UPDATE, INSERT, DELETE ON [Book] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [Publisher] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [Author] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [AuthoredBooks] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [Publisher] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [BooksPublished] TO [BookStoreAdministratorsRole];
GRANT SELECT, UPDATE, INSERT, DELETE ON [Category] TO [BookStoreAdministratorsRole];
GRANT SELECT, INSERT, UPDATE ON [Member] TO [BookStoreAdministratorsRole];
GRANT SELECT ON [Transaction] TO [BookStoreAdministratorsRole];
-- End: Role for Book Store Administrators --
```

Users inheriting from this role will be able to query all the tables, and update, add, and delete records within them (except for two tables). After creating the role, the relevant user is added to it:

```
-- Start: Assign User AdamMuhammad to [BookStoreAdministratorsRole] --
ALTER ROLE [BookStoreAdministratorsRole] ADD MEMBER [AdamMuhammad];
-- End: Assign User AdamMuhammad to [BookStoreAdministratorsRole] --
```

This maintains confidentiality by disallowing any other member that does not inherit the

permissions of this role from carrying out operations they ought not to perform.

The third role, according to the business rules, is the database administrator's role:

```

--- Start: Role for Database Administrators ---
CREATE ROLE DatabaseAdministratorsRole;
GRANT ALTER ON DATABASE::[APUBSPLDB] TO [DatabaseAdministratorsRole]; -- only ALTER needed
--- End: Role for Database Administrators ---

```

Users inheriting from this role will be granted the “ALTER” role on the entire database of APUBSPLDB, thus, this provides them with access to all of the DDL T-SQL statements only. One pitfall here is to instead assign this role “CONTROL” on the database instead of “ALTER”, however, this violates the principle of least privilege and compromises the confidentiality (and integrity) of the database at the highest possible level. Granting “CONTROL” provides users with more than only DDL T-SQL statements, as shown below (a few of the granted permissions that users of this role are not supposed to possess, highlighted with an arrow pointing towards them):

The screenshot shows a SQL query window with the following code:

```

1 USE APUBSPLDB;
2
3 CREATE USER testUser WITHOUT LOGIN;
4 GRANT CONTROL ON DATABASE::APUBSPLDB TO testUser;
5
6 EXECUTE AS USER = 'testUser';
7 SELECT * FROM sys.fn_my_permissions('APUBSPLDB', 'database');
8 revert;
9 DROP USER testUser;

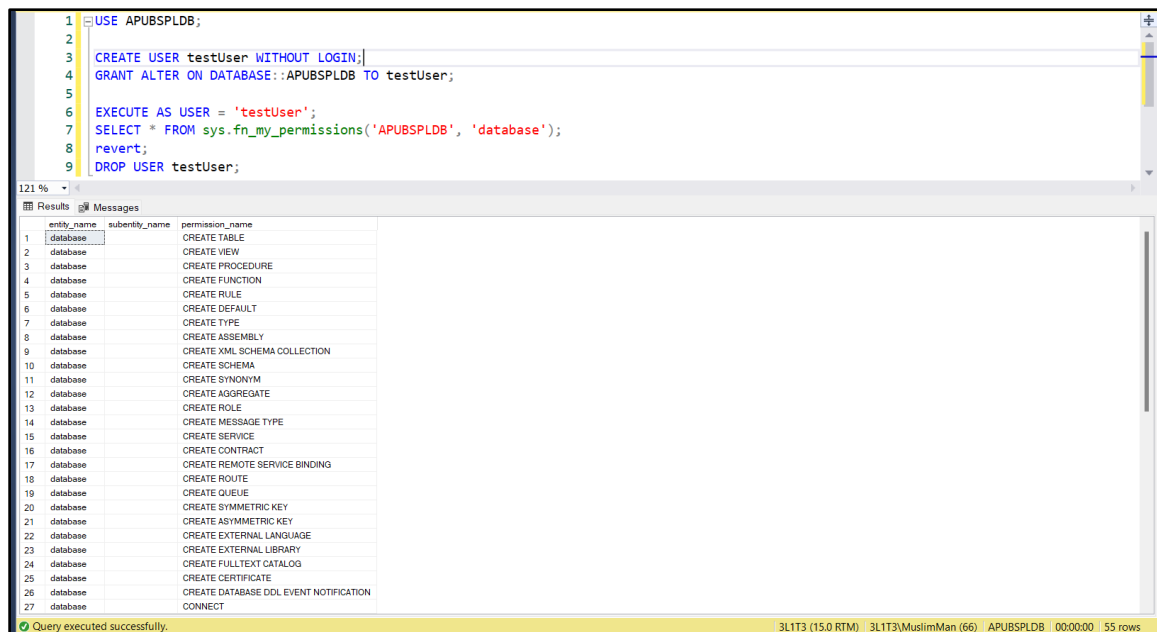
```

Below the query window, the 'Results' pane displays a table with 3 columns: entity_name, subentity_name, and permission_name. The table lists various permissions granted to the user. Red arrows point to the following permissions:

- VIEW DATABASE STATE
- VIEW DEFINITION
- TAKE OWNERSHIP
- ALTER
- ALTER ANY MASK
- UNMASK
- EXECUTE ANY EXTERNAL SCRIPT
- ADMINISTER DATABASE BULK OP...
- ALTER ANY SENSITIVITY CLASSIFIC...
- VIEW ANY SENSITIVITY CLASSIFICA...
- CONTROL

The status bar at the bottom indicates: Query executed successfully. 3L1T3 (15.0 RTM) 3L1T3\MuslimMan (66) APUBSPLDB 00:00:00 81 rows

Thus, to avoid this great (rather, complete) compromise of confidentiality, members of the database administrator's role are granted “CONTROL”; below its permissions are shown, noting that they are all only DDL T-SQL statements:



```

1 USE APUBSPLDB;
2
3 CREATE USER testUser WITHOUT LOGIN;
4 GRANT ALTER ON DATABASE::APUBSPLDB TO testUser;
5
6 EXECUTE AS USER = 'testUser';
7 SELECT * FROM sys.fn_my_permissions('APUBSPLDB', 'database');
8 revert;
9 DROP USER testUser;

```

entity_name	subentity_name	permission_name
database		CREATE TABLE
database		CREATE VIEW
database		CREATE PROCEDURE
database		CREATE FUNCTION
database		CREATE RULE
database		CREATE DEFAULT
database		CREATE TYPE
database		CREATE ASSEMBLY
database		CREATE XML SCHEMA COLLECTION
database		CREATE SCHEMA
database		CREATE SYNONYM
database		CREATE AGGREGATE
database		CREATE ROLE
database		CREATE MESSAGE TYPE
database		CREATE SERVICE
database		CREATE CONTRACT
database		CREATE REMOTE SERVICE BINDING
database		CREATE ROUTE
database		CREATE QUEUE
database		CREATE SYMMETRIC KEY
database		CREATE ASYMMETRIC KEY
database		CREATE EXTERNAL LANGUAGE
database		CREATE EXTERNAL LIBRARY
database		CREATE FULLTEXT CATALOG
database		CREATE CERTIFICATE
database		CREATE DATABASE DDL EVENT NOTIFICATION
database		CONNECT

Query executed successfully. 3L1T3 (15.0 RTM) 3L1T3\MuslimMan (66) APUBSPLDB 00:00:00 55 rows

Once this role is created, the relevant user is added to it:

```

-- Start: Assign User RajRaj to DatabaseAdministratorsRole --
ALTER ROLE DatabaseAdministratorsRole ADD MEMBER RajRaj;
-- End: Assign User RajRaj to DatabaseAdministratorsRole --

```

The fourth and last role according to the business rules is the management role:

```

--- Start: Role for Management ---
CREATE ROLE ManagementRole;
GRANT SELECT ON DATABASE::[APUBSPLDB] TO [ManagementRole];
--- End: Role for Management ---

```

It provides its members the permission to only query all tables within the APUBSPLDB database, and any other permission is denied, by default.

Creating logins, users, and roles, and adding users to their appropriate roles is of extraordinary importance when it comes to achieving confidentiality within the database of APU's Bookstore Private Limited; as it does not only manage all activities of users and allows them their required permissions, but it will also be beneficial when managing *Row-Level Security* (to achieve -greater- confidentiality) and *auditing* (to achieve integrity).

2.1.3 Row-Level Security

Although users have been added to their appropriate roles that grant them specific permissions, the confidentiality of the database is still not achieved, due to one user belonging to the "MembersRole" being able to query the records of other users within the table of "Member", for example (same applies for the records within the "Transaction" table). This breaches the confidentiality of APUBSPLDB, as all data is not protected from unauthorized access. To overcome this issue, row-level security will be implemented on the "Member" and "Transaction" tables, so that whenever a user with permission to query them does query them, the result returned will display that user's data only and not that of the others. With row-level

security implemented, the confidentiality of the data of all members and their transactions is protected from being viewed or manipulated by other members.

Implementing row-level security is a multi-step process: first, a security predicate function that performs the desired behavior is created, and secondly, that function is used as the filter predicate for the security policy that will be applied to a table. Such is the case for the “Member” table, where the security policy will be applied to it, which uses a security predicate:

```
-- Start: Row Level Security to allow members only see thier reocrds
CREATE FUNCTION [dbo].[functionSecurityPredicateAllowMembersOnlyTheirRecords]
    (@MemberName AS VARCHAR(90))
RETURNS TABLE
WITH SCHEMABINDING
AS
    RETURN SELECT 1 AS [functionSecurityPredicateAllowMembersOnlyTheirRecordsResult]
    WHERE @MemberName = USER_NAME() OR USER_NAME() = 'dbo';
-- End: Row Level Security to allow members only see thier reocrds

-- Start: Create Security Policy to allow members only see their records
CREATE SECURITY POLICY [SecurityPolicyAllowMembersOnlyTheirRecords]
ADD FILTER PREDICATE
    [dbo].[functionSecurityPredicateAllowMembersOnlyTheirRecords](REPLACE([Name], ' ', ''))
ON [dbo].[Member]
-- End: Create Security Policy to allow members only see their records
```

With this security policy implemented to/on the “Member” table, members querying the table will only be able to retrieve their records but not the records of other members. However, there is a big limitation in the implementation of the security predicate, which is that since user names stored within records are not unique, this could result in the first record being returned and the other records being ignored (however, for the sake of APUBSPLDB, there will not be duplicate user names).

The same procedure is carried out for the “Transaction” table, however, instead of using the member’s name as the function parameter, it uses the member Id, and, it queries the Member table for a member Id that is identical to the parameter’s one, where the name matches the current user’s name:

```
-- Start: Row Level Security to allow members only see thier reocrds
CREATE FUNCTION [dbo].[functionSecurityPredicateAllowMembersOnlyTheirTransactions]
    (@MemberID AS INT)
RETURNS TABLE
WITH SCHEMABINDING
AS
    RETURN SELECT 1 AS [functionSecurityPredicateAllowMembersOnlyTheirTransactionsResult]
    WHERE @MemberID = (SELECT [Member].[MemberID] from [dbo].[Member] WHERE
        REPLACE([Member].[Name], ' ', '') = USER_NAME() OR USER_NAME() = 'dbo';
-- End: Row Level Security to allow members only see thier reocrds

-- Start: Create Security Policy to allow members only see their records
CREATE SECURITY POLICY [SecurityPolicyAllowMembersOnlyTheirTransactions]
ADD FILTER PREDICATE
    [dbo].[functionSecurityPredicateAllowMembersOnlyTheirTransactions](MemberID)
ON [dbo].[Transaction]
-- End: Create Security Policy to allow members only see their records
```

With this security policy implemented on the “Transaction” table, members querying for their transaction records will not be able to view the records of other members. This increases the confidentiality of the data within the “Transaction” table specifically, and the entire database as a whole.

Although row-level security is now implemented, the confidentiality of the database is still not complete/fulfilled, with the reason being that the database files can be read in plaintext if a threat actor manages to take over the backend where it is served from, additionally, the “Member” table has a column that stores the payment card number of a member, encrypting this value is mandatory for the case of confidentiality, otherwise, a malicious threat actor may be able to read the payment card number and use it for illegal transactions.

2.1.4 Encryption

The last measure of defense that is taken to heighten the confidentiality of APUBSPLDB is encryption. Encryption is the art of concealing messages from unauthorized/unwanted parties. For APUBSPLDB, the column that stores the payment card number of members must be encrypted, so that it is protected from unauthorized users/entities viewing it, and possibly abusing it.

When it comes to encrypting data, SQL-Server encrypts it with a hierarchical encryption and key management infrastructure, thus, each layer encrypts the layer that is below it by using a combination of certificates, asymmetric keys, and symmetric keys (Microsoft, 2021).

With the encryption being hierarchical, SQL Server by default creates a Service Master Key that can encrypt individual Databases’ Master Keys, however, for APUBSPLDB, the Master Key (which is a symmetric key behind the scenes) created is protected/encrypted by a difficult, randomly-generated password (Master Keys can also be protected by the Service Master Key, using the following syntax “CREATE MASTER KEY”, in which “ENCRYPTION BY PASSWORD” is omitted):

```
-- Start: Create Master Key
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'PZmG4yKx69KBgwmGXq18nQ1m21xsAlYn';
-- End: Create Master Key

-- Start: Create Certificate that is protected by the MasterKey
CREATE CERTIFICATE APUBSPLDBCertificateMasterKeyProtected
WITH SUBJECT = 'APUBSPLDB Certificate (MasterKey Protected) - Self Signed';
-- End: Create Certificate that is protected by the MasterKey

-- Start: Create Certificate that is protected by a Password
CREATE CERTIFICATE APUBSPLDBCertificatePasswordProtected
ENCRYPTION BY PASSWORD = 'Tu2QyKbNawTaDUKcxaXXAQFR67334Y'
WITH SUBJECT = 'APUBSPLDB Certificate (Password Protected) - Self Signed';
-- End: Create Certificate that is protected by a Password
```

Once the Master Key is created, certificates can also be created, and they can be either protected by the Master Key or a password. As shown in the figure above, two certificates were created, the former is protected by the Master Key, while the latter is protected by a password (these two certificates will be used subsequently to encrypt the payment card numbers).

In addition to creating the two certificates, two asymmetric keys are also created, however, unlike the certificates, the encryption algorithm can be specified, and in this case, the cryptographically strongest algorithm available has been chosen:

```
-- Start: Create an Asymmetric Key that is Protected by the Master Key
CREATE ASYMMETRIC KEY APUBSPLDBAsymmetricKeyMasterKeyProtected
WITH ALGORITHM = RSA_2048
-- End: Create an Asymmetric Key that is Protected by the Master Key

-- Start: Create an Asymmetric Key that is Protected with a Password
CREATE ASYMMETRIC KEY APUBSPLDBAsymmetricKeyPasswordProtected
WITH ALGORITHM = RSA_2048
ENCRYPTION BY PASSWORD = 'xdCE4TAM7YZ99xTf27cUSXW74WGJOMK';
-- End: Create an Asymmetric Key that is Protected with a Password
```

Symmetric keys can also be used, however, they are only recommended over asymmetric ones in cases where performance is favored over stronger security. Now that certificates and asymmetric keys have been created, encrypting the entire database and specific columns becomes possible (or rather, it becomes possible with more options, since “ENCRYPTBYPASSPHRASE()” does not require any keys or certificates).

Now, to achieve greater confidentiality of APUBSPLDB, it will be encrypted entirely, by first creating a Database Encryption Key that is encrypted by a previously created Master Key-Protected certificate, and then setting the encryption state to on:

```
-- Start: Create Database Encryption Key and enable TDE
CREATE DATABASE ENCRYPTION KEY WITH ALGORITHM = AES_256
ENCRYPTION BY SERVER CERTIFICATE APUBSPLDBCertificateMasterKeyProtected;
ALTER DATABASE APUBSPLDB SET ENCRYPTION ON;
-- End: Create Database Encryption Key and enable TDE
```

This protects the data while “at rest”.

Afterward, since there are five users within the “Member” table, five different encryption procedures available from SQL Server will be applied to them.

For the first member, their payment card number is encrypted using the “ENCRYPTBYPASSPHRASE” function, with the encryption key/phrase being a difficult, randomly-generated string: “xdCE4TAM7YZ99xTf27cUSXW74WGds3JOMK”. It is very important to note that the PaymentCardNumber column value must be converted to “NVARCHAR” for the encryption to succeed since the data type of that column originally is “VARBINARY”. Decrypting the encrypted payment card number is achieved by using the “DECRYPTBYPASSPHRASE” function:

```
-- Start: Column Level Encryption for PaymentCardNumber in Member Table Using EncryptByPassphrase
UPDATE [Member] set PaymentCardNumberEncrypted =
ENCRYPTBYPASSPHRASE('xdCE4TAM7YZ99xTf27cUSXW74WGds3JOMK', CONVERT(nvarchar, PaymentCardNumberEncrypted)) where Member.MemberID = 1;
UPDATE [Member] set PaymentCardNumberEncrypted =
DECRYPTBYPASSPHRASE('xdCE4TAM7YZ99xTf27cUSXW74WGds3JOMK', CONVERT(nvarchar, PaymentCardNumberEncrypted)) from Member where MemberID = 1;
-- End: Column Level Encryption for PaymentCardNumber in Member Table Using EncryptByPassphrase
```

For the second member, their payment card number is encrypted with the previously created MasterKey-Protected Certificate using the “ENCRYPTBYCERT” function.

Decrypting the encrypted payment card number is achieved by using the “DECRYPTBYCERT” function:

```
-- Start: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYCERT
UPDATE [Member] SET PaymentCardNumberEncrypted =
    ENCRYPTBYCERT(CERT_ID('APUBSPLDBCertificateMasterKeyProtected'), PaymentCardNumberEncrypted)
FROM Member WHERE MemberID = 2;
SELECT CONVERT(varchar, DECRYPTBYCERT(CERT_ID('APUBSPLDBCertificateMasterKeyProtected'), PaymentCardNumberEncrypted))
FROM Member WHERE MemberID = 2;
-- End: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYCERT
```

For the third member, their payment card number is encrypted with the previously created Password-Protected Certificate using the “ENCRYPTBYCERT” function. Decrypting the encrypted payment card number is achieved by using the “DECRYPTBYCERT”, however, one important parameter is the third one, in which the password that was used to protect the certificate when creating it is provided:

```
-- Start: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
UPDATE [Member] SET PaymentCardNumberEncrypted =
    ENCRYPTBYCERT(CERT_ID('APUBSPLDBCertificatePasswordProtected'), PaymentCardNumberEncrypted) WHERE Member.MemberID = 3;
SELECT CONVERT(varchar, DECRYPTBYCERT(CERT_ID('APUBSPLDBCertificatePasswordProtected'),
    PaymentCardNumberEncrypted, N'Tu2QyKbNawTaDUKcXaXQFR67334Y'))
FROM Member WHERE MemberID = 3;
-- End: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
```

For the fourth member, their payment card number is encrypted with the previously created MasterKey-Protected Asymmetric Key using the “ENCRYPTBYASYMKEY” function. Decrypting the encrypted payment card number is achieved by using the “DECRYPTBYASYMKEY”:

```
-- Start: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
UPDATE [Member] SET PaymentCardNumberEncrypted =
    ENCRYPTBYASYMKEY(AsymKey_ID('APUBSPLDBAsymmetricKeyMasterKeyProtected'), PaymentCardNumberEncrypted) WHERE Member.MemberID = 4;
SELECT CONVERT(varchar, DECRYPTBYASYMKEY(AsymKey_ID('APUBSPLDBAsymmetricKeyMasterKeyProtected'),
    PaymentCardNumberEncrypted))
FROM Member WHERE MemberID = 4;
-- End: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
```

For the fifth member, their payment card number is encrypted with the previously created Password-Protected Asymmetric key using the function “ENCRYPTBYASYMKEY”. Decrypting the encrypted payment card number is achieved by using the “DECRYPTBYASYMKEY”, however, one important parameter is the third one, in which the password that was used to protect the asymmetric key when creating it is provided:

```
-- Start: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
UPDATE [Member] SET PaymentCardNumberEncrypted =
    ENCRYPTBYASYMKEY(AsymKey_ID('APUBSPLDBAsymmetricKeyPasswordProtected'), PaymentCardNumberEncrypted) WHERE Member.MemberID = 5;
SELECT CONVERT(varchar, DECRYPTBYASYMKEY(AsymKey_ID('APUBSPLDBAsymmetricKeyPasswordProtected'),
    PaymentCardNumberEncrypted, N'xdCE4TAM7YZ99xTf27cUSXW74WGJOMK'))
FROM Member WHERE MemberID = 5;
-- End: Column Level Encryption for PaymentCardNumber in Member Table Using ENCRYPTBYAsymmetricKey
```

With encryption being performed on all of the members’ payment card numbers, the confidentiality of APUBSPLDB has been greatly heightened. However, it still can be improved upon, for example, Always Encrypted can be also implemented; this feature disallows any user, even the “sa” account from viewing plaintext data, as the data will always be encrypted, and it is only decrypted on the client side.

2.2 Integrity

To ensure the integrity of APU's Book Store Private Limited database (APUBSPLDB), a myriad of SQL commands and conditions were added to the DDL statements, such that no user will be able to submit/provide false/invalid data. Integrity within the CIA triad emphasizes that at all times, data must be protected from any actions/situations that may render it corrupted/ingenuine. For example, allowing a member to set the total paid per transaction instead of calculating it automatically with triggers might allow them to provide misleading data, such as 0 instead of 100. Thus, to assure the integrity of data within APUBSPLDB, many security measures were taken.

2.2.1 Inferential integrity checks in DDL Statements of

```
CREATE TABLE [Member]
(
    [MemberID]          INT IDENTITY(1,1),
    [Name]              VARCHAR(90) NOT NULL,
    [City]              VARCHAR(30) NOT NULL,
    [Street]            VARCHAR(45) NOT NULL,
    [Postcode]          CHAR(5) NULL,
    [PhoneNumber]        VARCHAR(11) NOT NULL,
    [Type]              VARCHAR(20) NOT NULL CHECK([Type] = 'Staff' OR ([Type] = 'Student')),
    [LoginID]           VARCHAR(30) NULL, -- This is Null because it gets set by a trigger --
    [PaymentCardNumberEncrypted] VARBINARY(1000) NOT NULL

    CONSTRAINT [PK_Member_MemberID] PRIMARY KEY ([MemberID])
);
```

The DDL code for the member table created in the database is shown above. The column “Name” has the condition “NOT NULL”, so the value entered cannot be left blank by the user. The other variables in the database which are assigned NULL can be left with empty values by the user since they will be assigned via triggers. As for the “Type” variable the conditions are set to be NOT NULL and also there is a condition set that checks whether the input is “Staff” or “Student”. This allows only these 2 values can be entered in the variable because according to the business rules there are only two types of people in the database which are either “Staff” or “Student”.

The “PaymentCardNumberEncrypted” is set as VARBINARY where the users cannot leave the variable empty while entering the data in it as conditions state “NOT NULL”. The “MemberID” is set as the primary key in this table with the value auto-generated starting from 1. By setting values as NULL it will not allow the variables to be empty, where the user must always enter a value.

2.2.2 Auditing

The database administrator is responsible for managing the database and has full access to it, including database audits. Database auditing is the process of keeping an eye on the database to keep track of the actions taken by database users. Auditing the database aids in maintaining its integrity since the database administrator can watch and guard all transactions occurring. If a member can bypass security mechanisms put in place, the database administrator will be able to pinpoint where in time integrity was lost and correct it.

The amount of Server group requirements are used to set up the auditing for the APUBSPLB database. The code for the creation of the Server Audit is shown below:

```
-- Start: Create Server Audit and enable it
CREATE SERVER AUDIT [ServerAudit-APUBSPLDB] TO FILE (
    FILEPATH = 'M:\APUBSPLDB-Audits\',
    MAXSIZE = 1 GB,
    MAX_ROLLOVER_FILES = 5,
    RESERVE_DISK_SPACE = OFF)
WITH
(
    QUEUE_DELAY = 1000,
    ON_FAILURE = FAIL_OPERATION
)
ALTER SERVER AUDIT [ServerAudit-APUBSPLDB] WITH (STATE = ON);
-- End: Create Server Audit and enable it
```

The server auditing logs will be produced in the user's specified location, where the database administrator can subsequently examine and review the database logs (using SQL Server). The server auditing file's maximum size is set to 5GB, and reserve disc space is not enabled. The maximum number of rollover files is 5, therefore 6 files with a total size of 5GB will be produced, and any additional log entries will cause those files to be overwritten. As a result, the database administrator will have more logs to review for a longer time and maintain the integrity of the data within the database very extensively.

```
-- Start: Create Server Audit Specification
CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB];
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(SUCCESSFUL_LOGIN_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(LOGIN_CHANGE_PASSWORD_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(FAILED_LOGIN_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(LOGOUT_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(DATABASE_CHANGE_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(DATABASE_PERMISSION_CHANGE_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD
(DATABASE_OBJECT_PERMISSION_CHANGE_GROUP);
-- End: Create Server Audit Specification
```

The specifications for the server audit are shown in the above figure. Below, each action group is listed and described:

- “SUCCESSFUL_LOGIN_GROUP”

This group allows the database administrator to see the successful logins into the SQL server.

- “LOGIN_CHANGE_PASSWORD_GROUP”

The group allows the database administrator to track the password changes and resets of the SQL server logins.

- “FAILED_LOGIN_GROUP”

This group allows the database administrator to track the failed logins into the SQL server.

- “LOGOUT_GROUP”

This group allows the database administrator to track the users whenever they log out from the SQL server.

- “DATABASE_CHANGE_GROUP”

This group allows tracking of the creation, modification, and deletion of the database.

- “DATABASE_PERMISSION_CHANGE_GROUP”

This group allows tracking the permissions changes in the SQL server itself.

- “DATABASE_OBJECT_OWNERSHIP_CHANGE_GROUP”

This group allows tracking of the changes of the ownership in the database objects including certificates, symmetric and asymmetric keys, and schemas.

- DATABASE_OBJECT_PERMISSION_CHANGE_GROUP”

This group allows to track the attempts to grant, deny or revoke permissions on the database objects including certificates, database master keys, encryption keys, and asymmetric, assemblies and schemes in the database.

All the above groups have been added to the APU’S bookstore and will be stored as logs in a file so that the database administrator can view them while auditing the database.

```
-- Start: Create Database Audit Specification --  
USE APUBSPLDB;  
CREATE DATABASE AUDIT SPECIFICATION [DatabaseAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB];  
  
ALTER DATABASE AUDIT SPECIFICATION [DatabaseAuditSpecification-APUBSPLDB] FOR SERVER AUDIT [ServerAudit-APUBSPLDB] ADD  
  (SELECT, UPDATE, INSERT, DELETE, EXECUTE, RECEIVE, REFERENCES ON DATABASE::APUBSPLDB BY [public]);  
-- End: Create Database Audit Specification --
```

The above SQL code is for the database auditing logs where it will track all the logs whenever anyone tries to SELECT, UPDATE, INSERT, DELETE, EXECUTE, and RECEIVE in the database. In this way, the database administrator can track, and view, the activities at the server level and the database level.

2.2.3 Triggers

Triggers are set of SQL statements that are automatically executed when there is any change in the database. It is triggered as a response to certain events such as INSERT, UPDATE, or DELETE. They greatly contribute to the integrity of the data within the database, as they take care of inserting (or performing other actions) values that must be calculated automatically and not handled over for the user to do so.

```

/***** Object: Trigger [dbo].[SetLoginID] Script Date: 8/23/2022 2:12:50 PM *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE TRIGGER [SetLoginID] ON [Member] AFTER INSERT AS
BEGIN
    SET NOCOUNT ON
    UPDATE [Member] SET [LoginID] = CONCAT(REPLACE([Name], ' ', ''), CAST(RAND()*14307+214 AS INT), '@apubspl.com')
END

```

The above trigger in the database will trigger when the data is inserted into the [Member] table which creates a random member ID [SetLoginID] where it will take the name entered by the user and attach with a random number and set as @apubspl.com as the [SetLoginID] in the member table. Thus, this trigger maintains the integrity of the user

```

/***** Object: Trigger [dbo].[SetTotalPaid] Script Date: 8/23/2022 4:10:54 PM *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE TRIGGER [dbo].[SetTotalPaid] ON [dbo].[NumberOfCopiesPurchasedPerBook] AFTER INSERT AS BEGIN
SET NOCOUNT ON
UPDATE [NumberOfCopiesPurchasedPerBook] SET [TotalPaid] =
((SELECT [Price] FROM [Book] WHERE [BookID] = NumberOfCopiesPurchasedPerBook.BookID) *
((SELECT [Price] FROM [Book] WHERE [BookID] = NumberOfCopiesPurchasedPerBook.BookID) *
((SELECT CAST([Discount] AS FLOAT) FROM [Category] WHERE [CategoryName] = (SELECT CategoryName FROM Book WHERE BookID = NumberOfCopiesPurchasedPerBook.BookID)) / CAST(100 AS FLOAT)
+ (SELECT [Price] FROM [Book] WHERE [BookID] = NumberOfCopiesPurchasedPerBook.BookID) *
((SELECT CAST([Tax] AS FLOAT) FROM [Category] WHERE [CategoryName] = (SELECT CategoryName FROM Book WHERE BookID = NumberOfCopiesPurchasedPerBook.BookID)) / CAST(100 AS FLOAT));
END

```

The above figure is the trigger for the column [SetTotalPaid] column in the table [NumberOfCopiesPurchasedPerBook]. This statement will trigger when a value is inserted into the database and set the value of the column [SetTotalPaid] as the amount where the user needs to pay by calculating the discount and the tax to the number of books and the total number of books purchased by the user. Thus, this trigger maintains the integrity of the total amount paid by a customer, while taking care of the tax and discount, based on the category of the book purchased.

```

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE TRIGGER dbo.SetIsUserDeletedToTrue
ON dbo.[Transaction]
INSTEAD OF DELETE AS BEGIN
    SET NOCOUNT ON;
    UPDATE [Transaction]
        SET [IsDeletedByUser] = 1;
END
GO

```

The above figure is the trigger for the column [SetIsUserDeletedToTrue] on the transaction table. This statement will trigger when the user deletes his/her transaction from the transaction table with a setting as 1 when it is deleted, by default the value is set as 0.

A trigger will set the values automatically for the user consuming time for that column in the database.

2.2.4 Views

A view is a type of virtual table where it is set based on the rules set to the table in the database. The rules can be set by the database administrator which contains certain rows and columns that only the user can view and the rest of the data can be hidden from the view of the user.

```
/****** Script for SelectTopNRows command from SSMS *****/  
SELECT [TransactionID]  
      , [MemberID]  
      , [DateOfPurchase]  
FROM [APUBSPLDB].[dbo].[UndeletedTransactionsView]
```

The above SQL code is the view table created for the “Transaction” table where the columns “TransactionID”, “MemberID”, and “DataOfPurchase” are set to view the user, and more hidden columns are hidden from the view of the user.

2.3 Availability

Availability within the CIA triad emphasizes that data should be readily available for users that require it, regardless of any external circumstances acting upon it (such as corruption, unauthorized changes, failed hardware, network attacks, or any other undesirable events) that may hinder its reachability. For example, availability security measures ensure that even if a threat actor compromises the entire SQL-Server instance and encrypts it, the database administrator shall be able to restore the data in the shortest period, so that it becomes available back again to those who require it. One real-world case scenario was when WannaCry spread throughout the globe, some organizations (such as the German Deutsche Bahn) *had full backups ready to be used* and recovered from the virus extremely quickly, while others (such as the English National Health Service) *did not have full backups ready to be used*, thus, they suffered losses in millions of Pounds sterling.

To achieve (great) availability of the data within APUBSPLDB, backups will be utilized. However, before performing actual backups, within the SQL-Server spectrum, the recovery model must be decided first, out of three: simple, full, or bulk-logged. For APUBSPLDB, the full recovery model has been chosen.

The simple recovery model -the most basic of recovery models- was not chosen for APUBSPLDB since it does not support using transaction log backups (this however does result in less disk space usage, since the transaction logs will be deleted for completed transactions automatically) nor does it support point-in-time and page restores, as only the restoration of the secondary read-only file is supported (Jayaram, 2018a). Moreover, the simple recovery model was not chosen due to the restriction of being able to perform full, differential, and file level backups only (Jayaram, 2018a).

On the other hand, the full recovery model was chosen so that the greatest level of availability can be met. In the full recovery model, unlike in the simple model, the log records of completed transactions are not removed from the transaction log file; the transaction log records remain in the transaction log until a log backup is performed. The log records are written to transaction log backup, while the completed transaction log records are deleted from the transaction log when a transaction log backup is performed against databases that use the full recovery mode. Therefore, all DDL and DML statements/transactions performed are fully recorded in the transaction log file (Jayaram, 2018a).

In addition, the full recovery model has been chosen since it supports point-in-time restores, is suitable for supporting mission-critical applications (such as APU's Book Store PL), and facilitates the recovery of all the data with zero or minimal data loss (Jayaram, 2018a).

The full recovery model supports all six types of backups within SQL-Server, which start from full, differential, transaction log, copy-only, file and/or file-group, and partial (Jayaram, 2018a).

Now that the recovery model has been chosen, actual backups can be created; the most common types of backups available in SQL Server are full backups, differential backups, transaction log backups, tail log backups, copy-only backups, file backups, and partial backups (Jayaram, 2018). For APUBSPLDB, only full, differential, transaction log, and tail log backups will be created, providing it with a great level of availability that allows it to recover rapidly.

2.3.1 Full Backups

Full backups back up every object within the database, it is a complete copy that stores all the objects of the database, hence it is the foundation of any other subsequent type of backup, as a full backup must be performed at least once before other types of backups can be carried out (Jayaram, 2018).

For APUBSPLDB, a full backup is created and is also mirrored, while preserving the existing media header and backup sets on the media volume (this is the default behavior, thus, “WITH NOFORMAT” can be omitted, however, it is added for clarity):

```
BACKUP DATABASE [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBFullBackup.BAK'
MIRROR TO DISK = 'C:\APUBSPLDBFullBackupMirror.BAK'
WITH NOFORMAT
```

Full backups can be also split into multiple files, as the backup may be extremely big and must be split into chunks:

```
BACKUP DATABASE [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBFullBackup1.BAK'
TO DISK = 'M:\APUBSPLDBFullBackup2.BAK'
TO DISK = 'M:\APUBSPLDBFullBackup3.BAK'
TO DISK = 'M:\APUBSPLDBFullBackup4.BAK'
WITH NOFORMAT
```

Backups can also be encrypted, adding another layer of protection that increases the overall confidentiality of the database:

```
BACKUP DATABASE [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBTailLogBackup.log'
WITH ENCRYPTION (
    ALGORITHM = AES_256,
    SERVER_CERTIFICATE = APUBSPLDBCertificateMasterKeyProtected;
);
```

2.3.2 Differential Backups

Differential backups serve as a superset of the latest full backups and contain all the changes that have been carried out since the latest full backups (Jayaram, 2018). Since

differential backups don't backup every object within the database, they can be created quicker than full backups (Jayaram, 2018). For APUBSPLDB, a differential backup is created:

```
BACKUP DATABASE [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBDifferentialBackup.BAK'
WITH DIFFERENTIAL
```

2.3.3 Transaction Log Backups

A transaction log backup backs up the transaction logs of a database, transaction log files store a series of logs that provide the history of every modification of data in a database, most importantly, transaction log backups contain within them all logs records that have not been included in the last transaction log backup (Jayaram, 2018). Transaction log backups are incremental (unlike the cumulative differential backups), as they allow a database to be recovered to a specific point in time (Jayaram, 2018):

```
BACKUP LOG [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBLogBackup.TRN'
WITH NOFORMAT
```

2.3.4 Tail Log Backups

The first procedure/action that must be taken in the event of a failure for a database that is operating in the full or bulk-logged recovery models is creating a tail log backup of the live transaction log; this is an intermediate step that must be performed before starting the procedure of restoring (Jayaram, 2018). In case APUBSPLDB fails, the following syntax describes the tail log backup restoration command that will be used before starting the restoration of the database:

```
BACKUP LOG [APUBSPLDB]
TO DISK = 'M:\APUBSPLDBTailLogBackup.log'
WITH CONTINUE_AFTER_ERROR;
```

With four types of backups created for APUBSPLDB, its availability has been greatly heightened: any undesirable event occurring which may prevent access to the data within the database can be easily overcome by utilizing one or more of the created backups.

3.0 Backup and Restore Strategy

Successfully backing up APUBSPLDB is pivotal to a successful disaster recovery (DR) plan (all failing under availability security measures), and determining the frequency of data backup depends on the frequent data changes, and most importantly, the predefined policies and Service Level Agreement (SLA) that was dictated by APU Book Store Private Limited (Jayaram, 2018a). The backup destination of all APUBSPLDB backups will be local storage on hard disk drives, in addition to the backups being saved to a cloud storage provider.

While planning the backup strategy, Recovery-Point-Objective (RTO) and Recovery-

Time-Objective (RPO) were vital parameters to be considered (Jayaram, 2018a). The former is the point in time that data can be restored, while the latter is the amount of time taken to perform the restoration (Jayaram, 2018a). Given the SLA provided by APU Book Store Private limited, the backup strategy is as follows:

- A full backup is performed once a day during off-peak hours (12:00 A.M until 7:00 A.M). At this time, the library will not be open, thus, utilizing this time is best for a full backup.
- A differential backup is taken every day at 5:00 P.M. So that in case an undesired event happened in the middle of the day, all the transactions that were not recorded for the upcoming 12:00 A.M backup can be restored and then backed up with the next 12:00 A.M full backup.
- A transaction backup must be taken every 45 minutes. This is so because the interval at which members and staff come and go is very short. Thus, since no data must be lost, a transaction backup must be taken every 45 minutes.

4.0 Key Learnings

4.1 Mozhar Alhosni – TP058272

As a college student specializing in Cybersecurity, comparing my knowledge and comprehension of databases and their security techniques and procedures before and after this assignment is identical to comparing the sun with the moon. Not only I have applied theoretical security concepts to databases, but I also gained an immense amount of practical knowledge. I can safely say that now if a company (or anyone) tasks me with the mission of protecting their SQL Server databases, *I would be able to accept and manage that request very professionally.*

Firstly, I have learned about users and logins, how to create manage, and delete them, in addition to how to grant them permissions. Creating logins and mapping them to users allows great auditing of transactions/events within the SQL Server instance and the databases contained within it. Moreover, I learned that managing permissions in SQL Server is of extreme importance, as neglecting it will result in a full compromise of the CIA triad. Moreover, I learned that creating roles and assigning them permissions is better than assigning permissions to individual discreet users (unless it is required for specific cases). This way, any other user that belongs to a specific group of users can be added to the relevant role and inherit its permissions, instead of assigning the user the same roles as the others, as this produces the risk of granting a user wrongly more permissions than they require, thus violating the principle of least privilege. After learning all about users, roles, and permissions, I now greatly understand how to achieve confidentiality of the database by utilizing all three concepts and merging them.

Secondly, I have learned about database auditing and the plethora of options that SQL Server provides; I learned how to create a Server Audit, a Server Audit Specification, and a Database Audit Specification. Maintaining audits and watching them greatly contributes to the overall integrity of data within the database, in addition to referential integrity checks.

Thirdly, I have learned a great deal about encryption and hashing within SQL Server databases and that it is built upon a complex hierarchical structure that allows for a great number of possibilities. Encrypting and hashing data is directly proportional to the confidentiality level of data within a database, thus, these concepts are of utmost importance.

Fourthly, I have learned about database backups, their different types, how to choose the relevant backup type, and how to decide on the backup and recovery model. Backups contribute greatly to the availability of data within the database.

Many challenges were faced during this assignment, and all of them were overcome with research, patience, trial and error, and faith that true and beneficial knowledge can only be attained and mastered through difficult times and experiences.

4.2 Tarun Aitha (TP058015)

Because of this assignment, I've picked up a lot more information than simply the fundamentals of database security measures that can be implemented. During the process of constructing this database, I stumbled into several previously unknown actions and features, such as logins, triggers, views, server audits, database auditing, and auditing for the database itself. In addition to that, I gained knowledge about server audits, database auditing, and the availability and integrity of servers. I am now aware of the significance of backing up databases and the procedures for restoring them if an issue occurs in the future. My overall grasp of difficult database concepts, such as server specification groups, database specification groups, and the tasks they play, has increased as a result of my participation in a collaborative project.

4.3 Khirthiga A/P Chandrasekharan (TP058844)

As a direct result of working on this assignment, I have a deeper comprehension of a wide variety of topics that are connected to database security. I was surprised to learn how important data security is when it comes to protecting sensitive digital information that may be accessed via database logins. In addition to this, I am conscious of the importance of The database is responsible for maintaining not just the three pillars of database security—availability, confidentiality, and integrity—but also logins, triggers, views, server audits, database auditing, and some frequently used processes such as DDL and DML. My understanding of DDL statements, which create, amend, and destroy database objects like tables and indexes, has increased. I've gained a deeper understanding of the group of programming languages that are used particularly for making changes to databases by using DML. As a consequence of this, participating in the project as part of a group has been of great assistance to me in expanding my understanding of databases, security solutions, and their different responsibilities.

4.4 Haroon Gilani (TP058413)

As one of the many things I've learned, the principles of security features that can be applied in a database have been brought to my attention via the course of this assignment. The most important things that I picked up while constructing this database were how to maintain the database's confidentiality, integrity, and availability through the use of logins, triggers, views, server auditing, database auditing, and certain fundamental operations such as DDL and DML. In the case that there is a problem, I have realized how important it is to back up a database, and I also understand how to recover a database. My understanding of more sophisticated database concepts, such as server specification groups, database specification groups, and the function that each serves within a database, has typically improved.

References

- Jayaram, P. (2018a, March 1). *An overview of the process of SQL Server backup-and-restore*. SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More. <https://www.sqlshack.com/overview-sql-server-backup-restore-process/>
- Jayaram, P. (2018b, April 10). *Understanding SQL Server database recovery models*. SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More. <https://www.sqlshack.com/understanding-database-recovery-models/>
- Jayaram, P. (2018c, April 19). *Understanding SQL Server Backup Types*. SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More. <https://www.sqlshack.com/understanding-sql-server-backup-types/>
- Microsoft. (2021, June 12). *Encryption hierarchy*. Microsoft Docs. <https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/encryption-hierarchy?view=sql-server-ver16>