2025

ICT285 Database

ASSESSMENT 1

JERVIN ALEJANDRO 32940204

Contents

Question 1: Relational algebra

1. π Quantity (σ JobNo=4746 (SUPPLY))
   - Tthe query restricts (filters) the SUPPLY table to find all records for the job with JobNo 4745. Then, from these filtered records, it projects (displays) only the Quantity attribute.
2. π Weight (σ PartName=”Left-handed screwdriver” (PART))
   - The query starts by restricting the PART table to locate the specific record where the PartName is 'Left-handed screwdriver'. From this single record, it then projects the Weight attribute.
3. π JobName, PartName (σ Quantity < 500 ((SUPPLY * S.JobNo = P.JobNo PROJECT) * S.PNo = P.PNo PART))
   - The query performs a natural join on the SUPPLY, PART, and PROJECT tables to link supply records with their corresponding part and project details. From this combined dataset, it restricts the results to only include rows where the Quantity is less than 500. Finally, it projects the JobName and PartName for these matching records.
4. π SupplierName (σ JobName = “New Academic Building” OR JobName = “Removal of Asbestos” ((SUPPLY * SUPPLY.SNo = SUPPLIER.Sno SUPPLIER) * SUPPLY.JobNo=P.JobNo PROJECT))
   - This query joins the SUPPLY, SUPPLIER, and PROJECT tables to link suppliers to the projects they supplied. It then restricts this combined data to find records

where the JobName is either 'New Academic Building' or 'Removal of Asbestos'. Finally, it projects the SupplierName to list all suppliers involved in at least one of these two projects.

5. π SupplierName (π SNo (σ JobName = “New Academic Building” (SUPPLY * SUPPLY.JobNo=P.JobNo Project)))

   INTERSECT

   π SupplierName (π SNo (σ JobName = “Removal of Asbestos” ((SUPPLY * SUPPLY.SNo = SUPPLIER.Sno SUPPLIER) * SUPPLY.JobNo = P.JobNo PROJECT))

   - This query is solved in two parts. First, it finds the set of all supplier numbers (SNo) who supplied the 'New Academic Building' project. Second, it finds the set of all supplier numbers for the 'Removal of Asbestos' project.
     The INTERSECT operator is then used to find only the supplier numbers that appear in both sets. Finally, this result is joined with the SUPPLIER table to retrieve and project the corresponding SupplierName.

6. π SupplierName (σ PartName = “Fufful Valve” ((SUPPLY * SUPPLY.PNo = P.PNo PART) * SUPPLY.SNo = SUPPLIER.SNo SUPPLIER))

   - The query joins the SUPPLY, PART, and SUPPLIER tables to create a combined view of which supplier supplied which part. It then restricts this data to find all records where the PartName is 'Fufful Valve'. Finally, it projects the SupplierName from these filtered records.

7. π JobName (σ SupplierName = “Insightly Co#” OR City = “Dubai” ((SUPPLY * SUPPLY.JobNo = P.JobNo PROJECT) * SUPPLY.SNo = SUPPLIER.SNo SUPPLIER))
   - The query joins the SUPPLY, SUPPLIER, and PROJECT tables to connect suppliers to their projects. From this combined information, it restricts the results to include only records where the SupplierName is 'Insightly Co#' or the supplier's City is 'Dubai'. Finally, it projects the JobName to list the names of all projects supplied by these suppliers.
8. π SNo, SupplierName, City ((σ StartYear >= 2020 AND Country = “Singapore” (PROJECT)) * PROJECT.JobNo=SUPPLY.JobNo SUPPLY) * SUPPLY.SNo = SUPPLIER.SNo SUPPLIER)
   - The query begins by restricting the PROJECT table to find all projects that started in the year 2020 and are located in 'Singapore'. The result is then joined with the SUPPLY table (on JobNo) and subsequently with the SUPPLIER table (on SNo) to find the suppliers who worked on these specific projects. Finally, it projects the full details (SNo, SupplierName, City) of those suppliers.
9. σ StartYear = 2020 (PROJECT) →→ PROJECT2020

   FilterProject * FilterProject.JobNo = S.JobNo SUPPLY →→ SuppliesFor2020Projects

   π PNo(SuppliesFor2020Projects) →→ UsedPartNumbers2020

   PART LEFT JOIN PART.PNo=UsedPartNumbers2020.PNo UsedPartNumbers2020 →→ AllPartsWithUsageInfo

   σ UsedPartNumbers2020.PNo Is NULL (AllPartsWithUsageInfo) →→ UnusedParts

π PartName (UnusedParts) →→ Solution

- This query identifies parts not used in 2020 by first determining which parts *were* used and then finding everything not on that list. It begins by creating a list of all part numbers (PNo) that were supplied to projects starting in 2020. Next, it performs a LEFT JOIN from the complete PART table to this list of "used parts". The crucial step is then to filter this result, keeping only the rows where the part number from the "used parts" list is NULL. A NULL value signifies that a part in the main PART table had no match, meaning it was unused. Finally, the query projects the PartName from these unmatched rows.

10. σ StartYear = 2020 (PROJECT) →→ PROJECT2020

FilterProject * FilterProject.JobNo = S.JobNo SUPPLY →→ SuppliesFor2020Projects

π PNo, JobNo (SuppliesFor2020Projects) →→ PartProjectPairs2020

π JobNo (Projects2020) →→ RequiredProjectsID

PartProjectPairs2020 / RequiredProjectsID →→ PartNumbers

PartNumbers * PartNumbers.PNo=PART.PNo PART →→ PartsWIthDetails

σ PartName (PartsWithDetails) →→ Solution

- This query uses relational division (/) to find parts supplied to *all* 2020 projects. First, it creates the divisor: a list of all JobNos for projects that started in 2020. Second, it creates the dividend: a list of all part-project pairs (PNo, JobNo) from the SUPPLY table. The division operation then finds all PNos that are associated

with every single JobNo in the divisor list. Finally, the resulting part numbers are joined with the PART table to project their corresponding PartName.

Question 2: SQL – SELECT queries

1. SELECT *

   FROM tutorials.WORK

   WHERE DESCRIPTION LIKE '%Signed%';

   - This query scans the WORK table. It uses the WHERE clause with the LIKE operator to filter for records where the DESCRIPTION column contains the substring 'Signed'. The SELECT * then retrieves and displays all columns for every matching work of art.

2. SELECT Nationality, COUNT (*) AS "Number Of Artists"

   FROM tutorials.ARTIST

   GROUP BY Nationality

   HAVING COUNT(*) > 1;

   - This query first groups all records in the ARTIST table by Nationality. For each group, it calculates the number of artists using COUNT(*). The HAVING clause then filters these groups, keeping only those where the artist count is greater than one. The final result displays each of these nationalities and the corresponding number of artists.

3. SELECT MEDIUM, COUNT(*) AS "Number of Works"

   FROM tutorials.WORK

   GROUP BY Medium

   ORDER BY "Number of Works" DESC;

   - The query groups the rows in the WORK table based on the Medium column. It then uses COUNT(*) to calculate the total number of artworks for each medium. Finally, the ORDER BY clause sorts the results in descending order, displaying the mediums with the most works first.

4. SELECT C.FirstName AS "Customer Name", C.LastName AS "Customer LastName", A.FirstName AS "Artist Name", A.Lastname AS "Artist LastName"

   FROM tutorials.CUSTOMER C

   JOIN tutorials.CUSTOMER_ARTIST_INT I

   ON C.CustomerID = I.CustomerID

   JOIN tutorials.ARTIST A

   ON I.ArtistID = A.ArtistID

   ORDER BY C.LastName, A.LastName;

   - This query links customers to the artists they have an interest in. It achieves this by joining the CUSTOMER and ARTIST tables through
     the CUSTOMER_ARTIST_INT linking table. The SELECT statement retrieves the

names of the customer and the artist for each interest record. The results are sorted first by the customer's last name and then by the artist's last name.

5. SELECT FirstName ||''|| LastName AS "Full Name", Email

   FROM tutorials.CUSTOMER

   WHERE Street IS NULL;

   - This query retrieves the full name and email address for customers who do not have a recorded street address. It scans the CUSTOMER table and uses the WHERE clause to filter for records where the Street column has a NULL value.

6. SELECT W.WorkID, W.Title, A.FirstName ||''|| A.LastName AS "Artist Name", T.SalesPrice

   FROM tutorials.WORK W

   JOIN tutorials.ARTIST A ON W.ArtistID = A.ArtistID

   JOIN tutorials.TRANS T ON W.WorkID = T.WorkID

   WHERE T.SalesPrice >

   (SELECT AVG(SalesPrice) FROM tutorials.Trans);

   - This query identifies artworks that sold for a price higher than the overall average sales price. It first calculates the average of all sales prices from the TRANS table using a subquery. Then, it joins the WORK, ARTIST, and TRANS tables and filters the results, keeping only those transactions where the SalesPrice is greater than

the pre-calculated average. The final output lists the work's ID, title, artist's full name, and its selling price.

7. SELECT C.FirstName ||''|| C.LastName AS "Full Name"

   FROM tutorials.CUSTOMER C

   LEFT JOIN tutorials.TRANS T ON C.CustomerID = T.CustomerID

   WHERE T.TransactionID IS NULL;

   - This query finds all customers who have never purchased any artwork. It uses a LEFT JOIN to connect the CUSTOMER table with the TRANS table on CustomerID. This ensures all customers are included in the result, regardless of whether they have a matching transaction. The WHERE clause then filters for rows where T.TransactionID is NULL, which effectively isolates the customers who have no transactions.

8. SELECT A.FirstName ||''|| A.LastName AS "Full Name", COUNT(I.CustomerID) AS "Number of Admirers"

   FROM tutorials.ARTIST A

   JOIN tutorials.CUSTOMER_ARTIST_INT I ON A.ArtistID = I.ArtistID

   GROUP BY A.ArtistID, A.FirstName, A.LastName

   ORDER BY "Number of Admirers" DESC;

   - This query determines which artist has the most customer interest. It joins the ARTIST and CUSTOMER_ARTIST_INT tables, then groups the results by artist.

For each artist, it counts the number of interested customers using COUNT(I.CustomerID). The results are then sorted in descending order by this count, placing the artist with the highest number of admirers at the top of the list.

9. SELECT A.FirstName ||''|| A.LastName AS "Full Name", SUM(T.SalesPrice) AS "Total Sales"

   FROM tutorials.ARTIST A

   JOIN tutorials.WORK W ON A.ArtistID = W.ArtistID

   JOIN tutorials.TRANS T ON W.WorkID = T.WorkID

   GROUP BY A.ArtistID, A.FirstName, A.LastName

   ORDER BY "Total Sales" DESC;

   - This query calculates the total dollar amount of sales for each artist. It joins the ARTIST, WORK, and TRANS tables to link artists to the sales transactions of their work. The results are grouped by artist, and the SUM() function is used to calculate the total SalesPrice for each one. The final list is sorted in descending order to show the artists with the highest total sales first.

10. SELECT C.FirstName ||''|| C.LastName AS "Customer Name"

    FROM tutorials.CUSTOMER C

    JOIN tutorials.CUSTOMER_ARTIST_INT I ON C.CustomerID = I.CustomerID

    JOIN tutorials.ARTIST A ON I.ArtistID = A.ArtistID

```
WHERE A.Nationality = 'United States'

GROUP BY C.FirstName, C.LastName

HAVING COUNT(A.ArtistID) = (SELECT COUNT(ArtistID) FROM tutorials.ARTIST

WHERE Nationality = 'United States');
```

- This query identifies customers who are interested in every artist from the 'United States'. It joins the customer and artist tables via the interest table, filtering to only consider interests in US artists. It then groups the results by customer and counts the number of US artists each customer is interested in. The HAVING clause compares this count to the total number of US artists in the database (determined by a subquery). Only customers whose interest count matches the total number of US artists are included in the final result.

Question 3: Further SQL

1.

```
INSERT INTO CUSTOMER(CustomerID, LastName, FirstName, Street, City, State,
ZipPostalCode, Country, Email)

VALUES (1054, 'Bloggs', 'Joseph', '13 Murdoch Street', 'Hyden', 'WA', '6359', 'Australia',
'JoBloggs@someisp.com');


INSERT INTO ARTIST (ArtistID, LastName, FirstName, Nationality, DateOfBirth,
DateDeceased)

VALUES (20, 'Symbol', 'Gallic', 'French', 1972, NULL);


INSERT INTO WORK

VALUES (602, 'Gorillas in the Mist', 'Unique', 'Watercolour on Ppaer', '45 * 35cm -
Signed', 20);


INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice, CustomerID,
WorkID)

VALUES (256, '27/07/20', 4600, 1054, 602);
```

2.

```
CREATE TABLE MATCH (
    MatchID VARCHAR(5),
    Date DATE,
    Result CHAR(1),
    HomeTeamID VARCHAR(5),
    AwayTeamID VARCHAR(5),
    CONSTRAINT MatchPK PRIMARY KEY (MatchID),
    CONSTRAINT MatchResultCHK CHECK (Result IN ('W', 'L', 'D')),
    CONSTRAINT HomeTeamFK FOREIGN KEY (HomeTeamID) REFERENCES
Team(TeamID),
    CONSTRAINT AwayTeamFK FOREIGN KEY (AwayTeamID) REFERENCES
Team(TeamID)
);
```

3.

```
ALTER TABLE MATCH
    ADD (
      HomeTeamScore NUMBER(3),
      AwayTeamScore NUMBER(3),
      CONSTRAINT HomeScoreCHK CHECK (HomeTeamScore >= 0),
      CONSTRAINT AwayScoreCHK CHECK (AwayTeamScore >= 0)
    );
```

Question 4: Normalisation

The core problem with the current database design is data redundancy, where the same information is duplicated unnecessarily across the system. This single flaw is the source of several significant risks. Specifically, it introduces data anomalies, errors that can occur when staff add, update, or delete records. These constant risks of data corruption make the system inefficient to operate, difficult to maintain, and fundamentally unreliable for accurate reporting. The cause of these anomalies is a poorly structured table that violates the principles of database normalisation. The following anomalies are:

Insertion Anomalies

This problem arises because the table structure requires that all data fields in a row be filled. Consequently, if a piece of information is missing, the system will reject the entire entry, preventing valid data from being stored.

For example, a new doctor's details cannot be entered because the system requires a Patient ID and Consult Date for every new row. Since a new doctor has no patient history, their information cannot be recorded.

Deletion Anomalies

The current design is vulnerable to unintentional data loss because unrelated information is stored together in a single row, deleting one piece of data can unintentionally erase other critical information contained in that same row.

For example, if a patient has only one consultation on record, deleting that single entry would also erase all of their personal details, as there would be no other record of their existence in the database.

Update Anomalies

The system's design makes it difficult to keep information current. To change one detail, users must find and edit every record where that detail appears. If even one record is missed, the data becomes inconsistent. This flaw makes routine updates both inefficient and unsafe.

For example, to change the fee for a "Standard" consultation, every single record of that service must be located and updated individually. This manual process is not only time-consuming but also creates a high risk of data inconsistency, which could lead to significant billing errors if any record is missed.

Solution

The recommended solution is to normalise the database structure. The current Patient-Treatment table will be replaced by a set of four linked tables:

- Patient
- Doctor
- TreatmentItem
- Consultation

This normalised structure will store each piece of information in a single location, thereby eliminating data redundancy and preventing the associated errors. The result will be a more efficient, reliable, and maintainable database.

The following database design resolves the previously identified flaws. By achieving Third Normal Form (3NF), it eliminates the data anomalies and ensures data integrity.

Patient Table

Column Name	Data Type	Key	Description
PatientID	Number	**PK**	Unique identifier for each patient
PatientName	Varchar		The full name of the patient
PatientDOB	Date		The patient's date of birth

Doctor Table

Column Name	Data Type	Key	Description
ProviderNumber	Varchar	**PK**	Unique identifier for each doctor.
DoctorName	Varchar		The full name of the doctor

TreatmentItem Table

Column Name	Data Type	Key	Description
ItemNumber	Varchar	**PK**	Unique identifier for each service item
ItemDescription	Varchar		A description of the service
Fee	Currency		The standard fee for the service

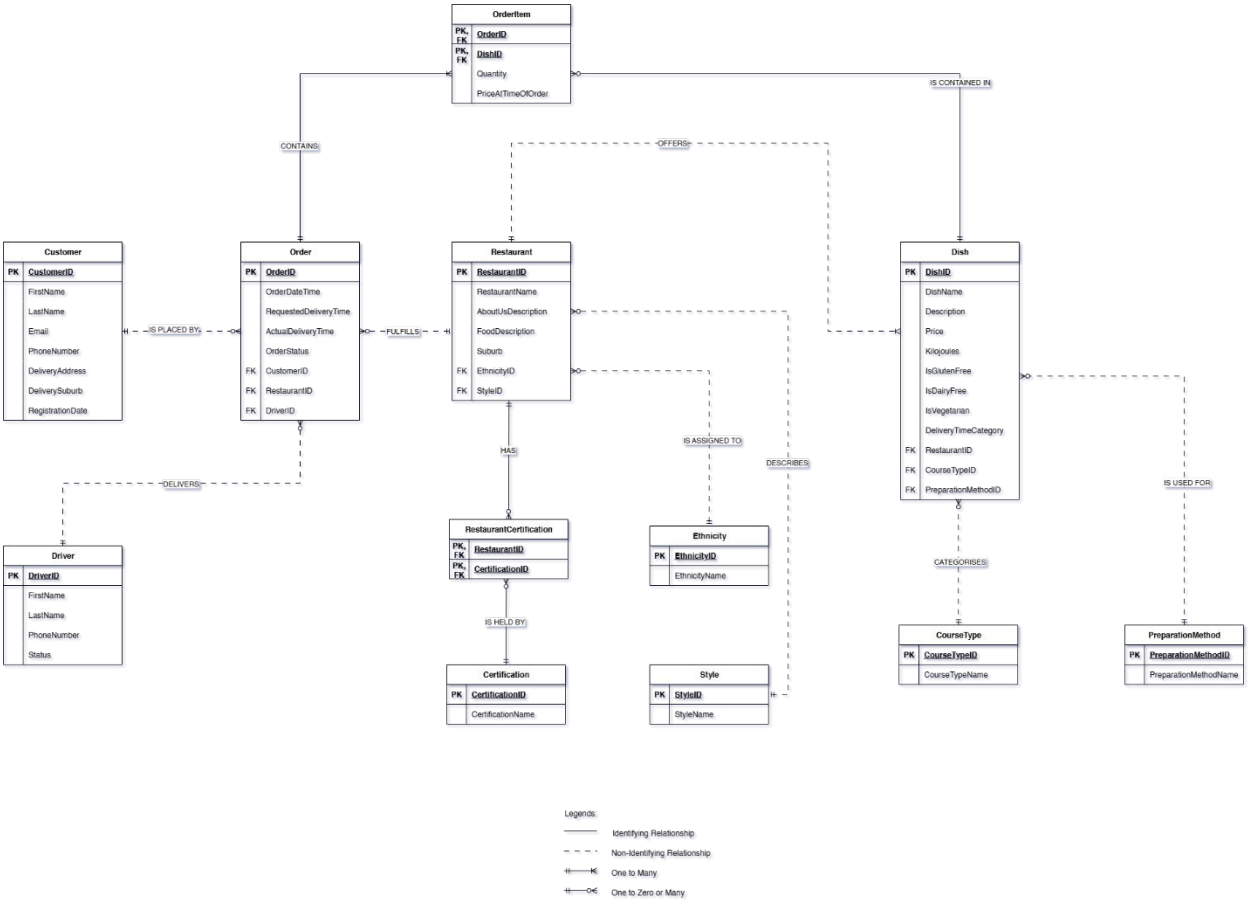
Consultation Table

Column Name	Data Type	Key	Description
PatientID	Number	**PK, FK**	Identifies the patient (references Patient)
ConsultDate	DateTime	**PK**	The date and time of the consultation
ItemNumber	Varchar	**PK, FK**	Identifies the service provided (references TreatmentItem)

ProviderNumber	Varchar	**FK**	Identifies the doctor who provided the service (references Doctor)

In this solution, a doctor's information can be added directly to the DOCTOR table. This action is now independent and does not require associated patient or consultation records, resolving a key limitation of the previous structure. Additionally, the new design isolates data, preventing accidental information loss. For instance, deleting a specific consultation record only removes the entry from the CONSULTATION table. The patient's primary record, containing their name and date of birth, is unaffected as it resides securely in the PATIENT table. This ensures transactional records can be managed without risk to core patient data. Finally, the new design eliminates the risk of data inconsistency when information is updated. Previously, changing a detail like a doctor's name required modifying multiple records, which could lead to errors. Now, such changes are made only once in a central location. This single update is automatically reflected across all associated consultation records, ensuring data integrity and accuracy throughout the system.

Question 5: Conceptual Design



Assumptions

- The database's financial functionality is limited to calculating order totals based on dish prices. In accordance with the project scope, it does not manage payments, transactions, or financial accounts.
- The system tracks driver availability with a simple status indicator. A driver is marked as either "Free" or "On Delivery" to make it easy to see at a glance who is ready for the next assignment.

- An "OrderStatus" field has been added to the ORDER table to track each order through the delivery process. This field will use status indicators such as 'Placed', 'Preparing', 'Delivering', and 'Completed'.
- The system relies on a direct text match between the customer and restaurant suburb fields to enforce the local ordering rule.
- To keep the database organised and efficient, each key record is assigned a unique, system-generated reference number . This method ensures that information can be retrieved quickly and reliably, preventing data errors.
- The customer's email address serves as a unique identifier within the system. This design choice enables it to function as a login credential and prevents the creation of duplicate accounts.
- Dietary requirements, such as 'Vegetarian' or 'Gluten Free', are implemented as dedicated 'Yes/No' fields in the DISH table. This approach is highly efficient for managing the current, predefined set of dietary needs.