Exercise walk-through

8.3.1

Joining two tables, aggregates with GROUP BY, and choosing SQL data types

A-Level Computer Science

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

8.3.1 Joining two tables, aggregates with GROUP BY, and choosing SQL data types

---

Name: _____ Class: _____ Date: _____ **Total: 38 marks**

KEY WORDS inner join • data type • query • join • foreign key

Objective

Choose SQL field types and build queries from their input data to their output. Sheet 8.3 introduces DDL and single-table DML; this sheet develops two-table joins and grouped results.

You must be able to:

- create a table definition with attributes of appropriate data types: `CHARACTER`, `VARCHAR` `BOOLEAN`, `INTEGER`, `REAL`, `DATE`, `TIME`
- create a database, add a **primary key** and add a **foreign key** , and change

a table definition with `ALTER TABLE`

- write an SQL script to query data held in at most **two** tables, using `INNER JOIN`
- use `SUM`, `COUNT`, `AVG`, `GROUP BY` and `ORDER BY` in a query

1 Key knowledge and worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

The two tables used throughout hold members and their current loans:

```
MEMBER(MemberID, Name, JoinDate, Active)
LOAN(LoanID, MemberID, BookTitle, LoanDate, Fine)
```

`MemberID` is the primary key of `MEMBER` and a foreign key in `LOAN`.

Choosing the data type

A `CREATE TABLE` question gives a mark for the types as well as for the structure, and the type has to suit what the field actually holds.

Type	Use it for	The trap
`INTEGER`	a count, an ID, a whole number	not for a phone number or a postcode: leading zeros are lost and no arithmetic is ever done on them
`REAL`	a measurement, a price, an average	
`VARCHAR(n)`	text of varying length, up to n characters	the usual right answer for a name or a title
`CHARACTER` (`CHAR(n)`)	text of a **fixed** length, such as a two-letter code	wastes space on anything variable
`BOOLEAN`	a field with exactly two states – yes/no, true/false	use this when the attribute represents true or false
`DATE`	a calendar date	stores dates for date comparisons and calculations; `VARCHAR` stores characters
`TIME`	a time of day	

Building the structure

```
CREATE DATABASE Library;

CREATE TABLE MEMBER (
  MemberID INTEGER PRIMARY KEY,
  Name VARCHAR(50),
  JoinDate DATE,
  Active BOOLEAN
);

CREATE TABLE LOAN (
  LoanID INTEGER PRIMARY KEY,
  MemberID INTEGER,
  BookTitle VARCHAR(100),
  LoanDate DATE,
  Fine REAL,
  FOREIGN KEY (MemberID) REFERENCES MEMBER(MemberID)
);
```

- The **foreign key** line names the field in **this** table, then the table and field it points at. Both halves are needed.
- To add a key to a table that already exists, use `ALTER TABLE`:

```
ALTER TABLE LOAN ADD FOREIGN KEY (MemberID) REFERENCES MEMBER(MemberID);
```

`ALTER TABLE` also adds a column: `ALTER TABLE MEMBER ADD Email VARCHAR(100);`

Worked example: trace matching rows

MEMBER	
MemberID	Name
1	Mei
2	Lin
3	Lin

LOAN			
LoanID	MemberID	BookTitle	Fine
1	1	Orbit	0
2	1	River	2
3	2	Sky	4

An **INNER JOIN** pairs rows when its `ON` condition is true. Here use the foreign key MemberID in LOAN and the primary key MemberID in MEMBER. Not every SQL join must compare keys, but this design uses the key relationship.

```
SELECT M.Name, L.BookTitle
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID
WHERE L.Fine > 0;
```

First match each loan to its member. Then keep rows whose Fine exceeds zero. Finally display Name and BookTitle. An **alias**, such as M for MEMBER, shortens a table name; qualifying fields also removes ambiguity when both tables contain MemberID.

Matched rows before WHERE			
MemberID	Name	BookTitle	Fine
1	Mei	Orbit	0
1	Mei	River	2
2	Lin	Sky	4

Result after WHERE Fine $>$ 0	
Name	BookTitle
Mei	River
Lin	Sky

Worked example: group by the member's identity

An **aggregate function** combines values: `COUNT` counts, `SUM` totals, and `AVG` finds the mean. This query counts current loans per member who has at least one loan:

```
SELECT M.MemberID, M.Name, COUNT(L.LoanID) AS NumLoans
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID
GROUP BY M.MemberID, M.Name
ORDER BY NumLoans DESC;
```

GROUP BY MemberID, Name		
MemberID	Name	NumLoans
1	Mei	2
2	Lin	1

Member 3 has no matching loan.
An INNER JOIN omits that member.

Group by the identifier as well as the name: two different members can both be called Lin. Each selected field outside an aggregate appears in `GROUP BY`. Member 3 has no matching loan, so this INNER JOIN does not include that member. `SELECT COUNT(*) FROM LOAN;` instead gives one total, 3, for the whole loan table.

To compute the mean fine on the displayed three loans, add 0, 2 and 4 and divide by 3: `AVG(Fine)` returns 2. `SUM(Fine)` returns 6. These functions use values, whereas COUNT counts rows/non-null values.

Write the clauses in this order: `SELECT`, `FROM` with `INNER JOIN ... ON`, `WHERE`, `GROUP BY`, `ORDER BY`. `WHERE` filters individual rows before grouping. A comma-separated two-table FROM with no matching condition produces every pairing; an explicit INNER JOIN missing ON may instead be rejected as invalid SQL.

Exercise 2.1

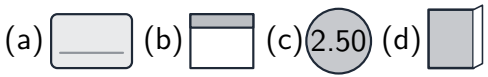
2.1 Give an appropriate SQL data type for each of these fields. [4]

(a) `Active` – whether a member's card is still valid

(b) `JoinDate` – the date a member joined

(c) `Fine` – an amount of money owed, such as 2.50

(d) `BookTitle` – the title of a book



Exercise 2.1

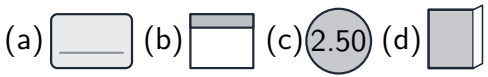
2.1 Give an appropriate SQL data type for each of these fields. [4]

(a) `Active` – whether a member's card is still valid

(b) `JoinDate` – the date a member joined

(c) `Fine` – an amount of money owed, such as 2.50

(d) `BookTitle` – the title of a book



2.1

(a) `BOOLEAN` – it has exactly two states

(b) `DATE`

(c) `REAL` – it has a fractional part (also accepted: `DECIMAL(6,2)`)

(d) `VARCHAR(100)` – text of varying length; any sensible n is accepted

Exercises 2.2·2.3

2.2 Write the SQL statement that creates a database called `Library`. [1]

`Library`

2.3 Write the SQL statement that adds `MemberID` in `LOAN` as a foreign key referring to `MEMBER`, given that both tables already exist. [2]

`MEMBER`

MemberID	Name

`LOAN`

LoanID	MemberID

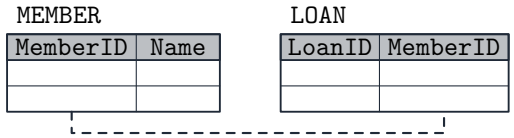
Exercises 2.2·2.3

2.2 Write the SQL statement that creates a database called `Library`. [1]

2.2

- `CREATE DATABASE Library;`

2.3 Write the SQL statement that adds `MemberID` in `LOAN` as a foreign key referring to `MEMBER`, given that both tables already exist. [2]



2.3

- `ALTER TABLE LOAN`
- `ADD FOREIGN KEY (MemberID) REFERENCES MEMBER(MemberID);`

Exercises 2.4·2.5

2.4 Write an SQL query that gives the **total number of loans** in the `LOAN` table. [2]

`LOAN`

LoanID	MemberID	BookTitle

2.5 State the order in which these clauses must be written in a query: `GROUP BY`, `SELECT`, `WHERE`, `FROM`, `ORDER BY`. [2]

`GROUP BY` `SELECT` `WHERE` `FROM` `ORDER BY`

Exercises 2.4·2.5

2.4 Write an SQL query that gives the **total number of loans** in the `LOAN` table. [2]

`LOAN`

LoanID	MemberID	BookTitle

2.4

- `SELECT COUNT(*)` – or `COUNT(LoanID)`
- `FROM LOAN;`
- no `GROUP BY`, because the question wants one number for the whole table

2.5 State the order in which these clauses must be written in a query: `GROUP BY`, `SELECT`, `WHERE`, `FROM`, `ORDER BY`. [2]

`GROUP BY` `SELECT` `WHERE` `FROM` `ORDER BY`

2.5

- `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`
- an `INNER JOIN ... ON` goes with the `FROM`, between `FROM` and `WHERE`

Exercise 2.6

2.6 Use the worked MEMBER and LOAN data. List the MemberID and BookTitle pairs that remain when `Fine >= 4`. State the COUNT of those rows and their SUM of Fine. [3]

LOAN

LoanID	MemberID	BookTitle	Fine
1	1	Orbit	0
2	1	River	2
3	2	Sky	4

Exercise 2.6

2.6 Use the worked MEMBER and LOAN data. List the MemberID and BookTitle pairs that remain when `Fine >= 4`. State the COUNT of those rows and their SUM of Fine. [3]

LOAN

LoanID	MemberID	BookTitle	Fine
1	1	Orbit	0
2	1	River	2
3	2	Sky	4

2.6 The remaining pair is `(2, Sky)`. COUNT is 1. SUM of Fine is 4.

Exercise 3.1

3.1 Write SQL (DDL) to create the table `LOAN` with the fields `LoanID`, `MemberID`, `BookTitle`, `LoanDate` and `Fine`.

`LoanID` is the primary key and `MemberID` is a foreign key referring to `MEMBER`. Choose an appropriate data type for every field. [5]

`LOAN`

LoanID	MemberID	BookTitle	LoanDate	Fine

Exercise 3.1

3.1 Write SQL (DDL) to create the table `LOAN` with the fields `LoanID`, `MemberID`, `BookTitle`, `LoanDate` and `Fine`.

`LoanID` is the primary key and `MemberID` is a foreign key referring to `MEMBER`. Choose an appropriate data type for every field. [5]

`LOAN`

LoanID	MemberID	BookTitle	LoanDate	Fine

3.1

- `CREATE TABLE LOAN (` with the closing `);`

Full solution on next slide

Full solution 3.1

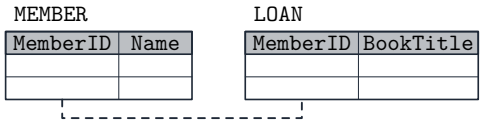
3.1

- `CREATE TABLE LOAN (` with the closing `);`
- `LoanID INTEGER PRIMARY KEY`
- `MemberID INTEGER`, `LoanDate DATE`
- `BookTitle VARCHAR(100)`, `Fine REAL`
- `FOREIGN KEY (MemberID) REFERENCES MEMBER(MemberID)`

```
CREATE TABLE LOAN (
  LoanID INTEGER PRIMARY KEY,
  MemberID INTEGER,
  BookTitle VARCHAR(100),
  LoanDate DATE,
  Fine REAL,
  FOREIGN KEY (MemberID) REFERENCES MEMBER(MemberID)
);
```

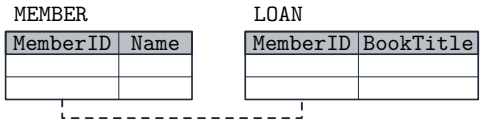
Exercise 3.2

3.2 Write an SQL query that lists the **name** of each member together with the **titles** of the books they currently have on loan. [4]



Exercise 3.2

3.2 Write an SQL query that lists the **name** of each member together with the **titles** of the books they currently have on loan. [4]



3.2

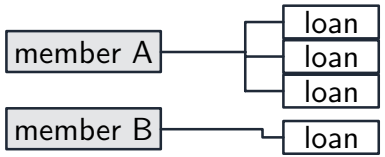
- `SELECT M.Name, L.BookTitle`
- `FROM MEMBER M INNER JOIN LOAN L`
- `ON M.MemberID = L.MemberID;`

```
SELECT M.Name, L.BookTitle
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID;
```

- A comma join with a matching WHERE condition is also valid; mark equivalent syntax by the same logical steps.

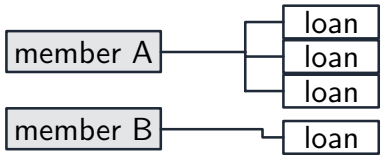
Exercise 3.3

3.3 Write an SQL query that gives, for each member, their **MemberID**, **name** and **number of current loans** (include members with at least one loan), with the busiest member first. [4]



Exercise 3.3

3.3 Write an SQL query that gives, for each member, their **MemberID**, **name** and **number of current loans** (include members with at least one loan), with the busiest member first. [4]



3.3

- `SELECT M.MemberID, M.Name, COUNT(L.LoanID)`

Full solution on next slide

Full solution 3.3

3.3

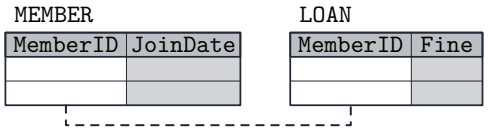
- `SELECT M.MemberID, M.Name, COUNT(L.LoanID)`
- the join with its `ON` clause
- `GROUP BY M.MemberID, M.Name`
- `ORDER BY` the count `DESC`

```
SELECT M.MemberID, M.Name, COUNT(L.LoanID) AS NumLoans
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID
GROUP BY M.MemberID, M.Name
ORDER BY NumLoans DESC;
```

- MemberID separates different members with the same name. Both selected non-aggregate fields occur in GROUP BY.

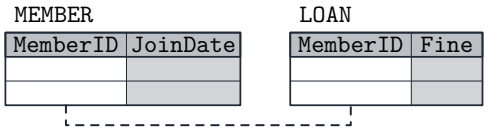
Exercise 3.4

3.4 Write an SQL query that gives the **average fine** owed by members who joined before 1 January 2024. [3]



Exercise 3.4

3.4 Write an SQL query that gives the **average fine** owed by members who joined before 1 January 2024. [3]



3.4

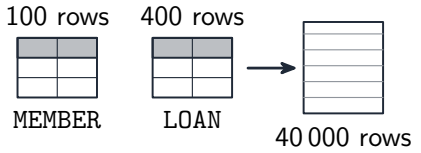
- `SELECT AVG(L.Fine)`
- the join with its `ON` clause
- `WHERE M.JoinDate < '2024-01-01';`

```
SELECT AVG(L.Fine)
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID
WHERE M.JoinDate < '2024-01-01';
```

Exercise 3.5

3.5 A student writes this query and is surprised that it returns 40,000 rows, when `MEMBER` holds 100 rows and `LOAN` holds 400. Every loan references an existing member.

```
SELECT M.Name, L.BookTitle
FROM MEMBER M, LOAN L;
```

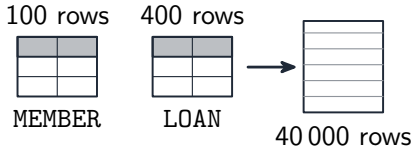


Explain what is wrong and write the corrected query. [3]

Exercise 3.5

3.5 A student writes this query and is surprised that it returns 40,000 rows, when `MEMBER` holds 100 rows and `LOAN` holds 400. Every loan references an existing member.

```
SELECT M.Name, L.BookTitle
FROM MEMBER M, LOAN L;
```



Explain what is wrong and write the corrected query. [3]

3.5

- The two-table FROM has no matching condition, so it forms a Cartesian product
- every member is therefore paired with every loan, giving $100 \times 400 = 40\,000$ rows
- the corrected query uses an `INNER JOIN` with a matching condition, keeping the 400 pairings where the foreign key matches the primary key

```
SELECT M.Name, L.BookTitle
FROM MEMBER M INNER JOIN LOAN L
  ON M.MemberID = L.MemberID;
```

Exercise 3.6

3.6 Table `Result(ClassID, Mark)` holds examination marks. Write SQL to output each class ID and its average mark, highest average first. Explain what `ON DELETE CASCADE` would do if these rows referenced a `Class` table and a class were deleted. [5]

ClassID | Mark

Exercise 3.6

3.6 Table `Result(ClassID, Mark)` holds examination marks. Write SQL to output each class ID and its average mark, highest average first. Explain what `ON DELETE CASCADE` would do if these rows referenced a `Class` table and a class were deleted. [5]

ClassID | Mark

3.6

```
SELECT ClassID, AVG(Mark)
FROM Result
GROUP BY ClassID
ORDER BY AVG(Mark) DESC;
```

- Correct SELECT/FROM and AVG.
- GROUP BY ClassID and descending ordering.
- Referencing Result rows for the deleted class would also be deleted.