SQL in a Unit 4 context

SQL behind a data exchange

A client might send JSON to an API, but the back end may validate the request and use SQL to read or update a relational database.

```
mobile app → HTTPS/JSON → API validation → parameterised SQL → database
mobile app ← JSON response ← result transformation ← SQL result
```

> **Separation of concerns:** SQL manages relational data. The API controls authentication, authorisation, validation, error handling and the representation returned to the client.

Use a small relational schema

```
VENUE(
    venue_id PRIMARY KEY,
    venue_name,
    suburb,
    accessible
)
```

EVENT(event_id PRIMARY KEY, venue_id FOREIGN KEY REFERENCES VENUE(venue_id), title, event_date, capacity, status)

BOOKING(booking_id PRIMARY KEY, event_id FOREIGN KEY REFERENCES EVENT(event_id), attendee_email, places, booking_status)

Keys connect records without repeating all venue and event data in every booking.

Trace a relationship

Explain how a BOOKING record is connected to a VENUE.

Check your answer

BOOKING.event_id links to EVENT.event_id. That EVENT record contains venue_id, which links to VENUE.venue_id. The relationship therefore passes through EVENT.

SELECT, WHERE and ORDER BY

```
SELECT event_id, title, event_date
FROM EVENT
WHERE status = 'open'
  AND event_date >= '2026-08-05'
ORDER BY event_date ASC, title ASC;
```

- `SELECT` chooses columns.
- `FROM` identifies the table.
- `WHERE` filters rows before grouping.
- `ORDER BY` sorts the result; `ASC` is ascending and `DESC` descending.

Text and date literals use quotes. Numeric and Boolean syntax depends on the database system and schema.

Write a query

Develop a query that returns the title and capacity of cancelled events, largest capacity first.

Check your answer

```
SELECT title, capacity
FROM EVENT
WHERE status = 'cancelled'
ORDER BY capacity DESC;
```

Conditions and IN

```
SELECT event_id, title
FROM EVENT
WHERE status IN ('open', 'full')
  AND capacity >= 20;
```

`IN` tests membership of a set. Combine conditions with `AND`, `OR` and parentheses so the intended logic is unambiguous.

> **NULL needs special handling:** use `IS NULL` or `IS NOT NULL`, not `= NULL`.

Aggregate functions and GROUP BY

Aggregate functions summarise rows: `COUNT`, `MIN`, `MAX`, `AVG` and `SUM`.

```
SELECT event_id, SUM(places) AS booked_places
FROM BOOKING
WHERE booking_status = 'confirmed'
GROUP BY event_id;
```

Each output row represents one event group. Non-aggregated selected columns normally need to appear in `GROUP BY`.

Count by status

Develop a query that counts bookings for each `booking_status`.

Check your answer

```
SELECT booking_status, COUNT(*) AS booking_count
FROM BOOKING
GROUP BY booking_status
ORDER BY booking_status;
```

WHERE and HAVING do different jobs

- `WHERE` filters individual rows before groups are formed.
- `HAVING` filters groups after aggregation.

```
SELECT event_id, SUM(places) AS booked_places
FROM BOOKING
WHERE booking_status = 'confirmed'
GROUP BY event_id
HAVING SUM(places) >= 20;
```

The WHERE clause excludes non-confirmed rows. HAVING keeps only event groups whose confirmed places total at least 20.

INNER JOIN related tables

```
SELECT EVENT.title, VENUE.venue_name, VENUE.suburb
FROM EVENT
INNER JOIN VENUE
    ON EVENT.venue_id = VENUE.venue_id
WHERE EVENT.status = 'open'
ORDER BY EVENT.title;
```

An inner join returns only rows with a matching relationship. Qualify column names with table names when ambiguity is possible.

Join three tables

Develop a query returning booking ID, event title and venue name for confirmed bookings.

View a sample solution

```
SELECT BOOKING.booking_id, EVENT.title, VENUE.venue_name
FROM BOOKING
INNER JOIN EVENT
    ON BOOKING.event_id = EVENT.event_id
INNER JOIN VENUE
    ON EVENT.venue_id = VENUE.venue_id
WHERE BOOKING.booking_status = 'confirmed';
```

Subqueries

A subquery supplies a result to another query.

```
SELECT title, capacity
FROM EVENT
WHERE capacity > (
    SELECT AVG(capacity)
    FROM EVENT
);
```

The inner query calculates the average capacity. The outer query returns events above it.

Use a join when combining related columns and a subquery when one result naturally supplies a condition. Either may sometimes express the same goal; clarity and database behaviour matter.

Filter by a related set

Develop a subquery that returns event titles at accessible venues.

Check your answer

```
SELECT title
FROM EVENT
WHERE venue_id IN (
    SELECT venue_id
```

```
    FROM VENUE
    WHERE accessible = TRUE
);
```

The Boolean literal may differ across database systems.

CREATE and constraints

```
CREATE TABLE BOOKING (
    booking_id INTEGER PRIMARY KEY,
    event_id INTEGER NOT NULL,
    attendee_email VARCHAR(254) NOT NULL,
    places INTEGER NOT NULL CHECK (places > 0),
    booking_status VARCHAR(20) NOT NULL,
    FOREIGN KEY (event_id) REFERENCES EVENT(event_id)
);
```

Types and supported constraint syntax vary by database system. Constraints provide a final integrity layer but do not replace clear UI validation and server-side checks.

INSERT and UPDATE

```
INSERT INTO BOOKING
    (booking_id, event_id, attendee_email, places, booking_status)
VALUES
    (501, 42, 'sam@example.org', 2, 'confirmed');
```

```
UPDATE BOOKING
SET booking_status = 'cancelled'
WHERE booking_id = 501;
```

> **Check UPDATE scope:** omitting or miswriting WHERE may change every row. Preview the target with an equivalent SELECT and use transactions or backups appropriate to the system.

Update a specific event

Develop an UPDATE that marks event 42 as full.

Check your answer

```
UPDATE EVENT
SET status = 'full'
WHERE event_id = 42;
```

ALTER and DROP

```
ALTER TABLE EVENT
ADD registration_url VARCHAR(500);
```

`ALTER` changes an existing structure. `DROP` removes a database object, such as a table, and can cause irreversible data loss.

Before structural change, consider existing data, defaults, nullability, dependent queries and exchange contracts. Adding a required field without a value for old records can break the system.

Secure application queries

Never build SQL by directly joining untrusted input into a query string.

```
# Unsafe idea
query = "SELECT * FROM BOOKING WHERE attendee_email = '" + input + "'"
```

Safe pattern

query = "SELECT booking_id, event_id, places FROM BOOKING WHERE attendee_email = ?"
ExecuteWithParameters(query, [input])

- Use parameterised queries or prepared statements.
- Grant the application database account only needed permissions.
- Authorise the user before querying protected records.
- Select only fields the response needs.
- Avoid exposing database errors or schema details to clients.

Parameters protect values, not arbitrary table names or SQL keywords. Dynamic structure requires a controlled allow-list.

Independent practice: event availability

Using the VENUE, EVENT and BOOKING schema:

1. **Develop** a query returning open event titles, venue names and dates in accessible venues, earliest first.
2. **Develop** a query returning each event ID and confirmed booked-place total, but only totals of 10 or more.
3. **Develop** a subquery returning events whose capacity is greater than the maximum capacity of events at venue 3.
4. **Explain** why attendee email should not be returned by a public events endpoint.

View a sample solution

1.
```
SELECT EVENT.title, VENUE.venue_name, EVENT.event_date
FROM EVENT
INNER JOIN VENUE ON EVENT.venue_id = VENUE.venue_id
WHERE EVENT.status = 'open'
  AND VENUE.accessible = TRUE
```

```
ORDER BY EVENT.event_date ASC;
```

2.
```
SELECT event_id, SUM(places) AS booked_places
FROM BOOKING
WHERE booking_status = 'confirmed'
GROUP BY event_id
HAVING SUM(places) >= 10;
```

3.
```
SELECT event_id, title, capacity
FROM EVENT
WHERE capacity > (
    SELECT MAX(capacity)
    FROM EVENT
    WHERE venue_id = 3
);
```

4. The public endpoint needs event availability, not attendee personal information. Excluding the email minimises disclosure and enforces separation between public and authorised booking data.

Revision #5
Created 2026-10-06 00:40:40 UTC by Mr Napper
Updated 2026-10-06 01:32:37 UTC by Mr Napper