Topic 9: Databases

IGCSE Computer Science 0478 | SenpaiCorner | Mr. Wei

Learning Objectives

- Define a single-table (flat-file) database; distinguish **fields** (columns) from **records** (rows) and individual data items.
- Assign appropriate data types — text/alphanumeric, character, Boolean, integer, real, date/time — to database fields.
- Select and justify a **primary key** field that uniquely identifies every record.
- Read, write, complete, and trace SQL queries using `SELECT`, `FROM`, `WHERE`, `ORDER BY` (ASCENDING / DESCENDING), `AND`, `OR`, `SUM`, and `COUNT`.

9.1 Single-Table Databases

A **database** is an organised collection of data. For IGCSE 0478 you are only required to understand and design **single-table databases** (also called *flat-file databases*).

Fields, Records, and Data Items

- **Record (Row):** All the data relating to one entity (e.g. one student). Each row in the table is one complete record.
- **Field (Column):** A single category of data stored for every entity (e.g. the `FirstName` column). Each column is one field.
- **Data Item:** The individual value stored in a single cell — the intersection of one record and one field.

StudentID	FirstName	Age	Grade
S001	Alice	16	10
S002	Bob	15	9
S003	Carol	17	11

Fields (Columns)

Record (Row)

Data Item

Validation in Databases

Just as in algorithm design, a database must **validate** data on entry to ensure it is sensible:

- **Length check** — ensures a text field is not too short or too long (e.g. a password must be 8–20 characters).

- **Range check** — ensures a numeric field falls within acceptable bounds (e.g. age must be between 0 and 120).
- **Presence check** — ensures a required field is not left blank.

Validation vs. Verification

Validation checks that data is *sensible* (correct type or range). It does *not* guarantee the data is *accurate* — a student's age of 99 would pass a range check if the upper bound were set too high.

9.2 Basic Data Types

Every field in a database must be assigned a **data type**, which controls what kind of data can be stored in that column.

Data Type	Description	Example Use
Text / Alphanumeric	String of letters, numbers, and symbols	Name, address, phone number
Character	A single letter, digit, or symbol	Gender (`'M'/'F'`), Grade (`'A'`)
Boolean	One of two states (True/False or Yes/No)	`Paid = TRUE`, `In_Stock = FALSE`
Integer	Whole number — no decimal part	Age, number of items in stock
Real	Number with a fractional or decimal part	Price (`19.99`), weight (`4.5 kg`)
Date/Time	Date or time in a structured format	Date of Birth (`YYYY-MM-DD`)

Phone Numbers and Leading Zeros — Always Use Text

Always assign **Text/Alphanumeric** to phone numbers or ID codes that begin with a zero (e.g. `"077123456"`). If you use **Integer**, the database will automatically strip the leading zero, permanently corrupting the data.

Choosing Integer for Any Field That Contains Digits

A common error is assigning **Integer** to every field that contains numbers. Ask: *will arithmetic ever be performed on this field?* If not — and especially if the value may contain leading zeros or be purely a label — use **Text/Alphanumeric** instead.

9.3 The Primary Key

A **primary key** is a field (or minimal set of fields) that **uniquely identifies every record** in a table. No two records may share the same primary key value, and the field must never be empty (null).

Identifying a Suitable Primary Key

- **Good choices:** `StudentID`, `PassportNumber`, `Barcode` — values that are structurally guaranteed to be unique per entity and assigned by the system.
- **Poor choices:** `FirstName`, `Age`, `Grade` — values that can be shared by multiple records.

A Name Is Never a Safe Primary Key

`LastName` is not a valid primary key even if no two students currently share a surname — a new student could be added with the same surname at any time. Only choose a field that is *structurally* guaranteed unique for every possible entity (e.g. a system-assigned ID number).

Justifying Your Primary Key in the Exam

State *two* things: (1) the field you chose, and (2) *why* it is unique. Example: "`StudentID` is the primary key because it is assigned uniquely to each student and no two students share the same ID." One-word answers (*"StudentID"*) rarely earn full marks.

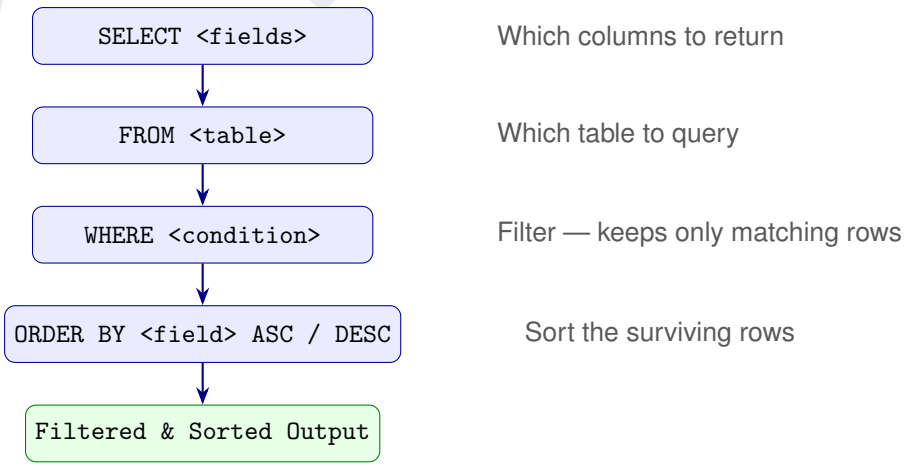
9.4 Structured Query Language (SQL)

SQL is the standard language for querying and retrieving data from a database. For IGCSE 0478 you must be able to read, write, complete, and trace the output of SQL queries on **single tables only**.

SQL Keywords

- `SELECT` — specifies which fields (columns) to return. Use `SELECT *` to return all fields.
- `FROM` — specifies the table to query.
- `WHERE` — applies a filter; only records satisfying the condition are returned.
- `ORDER BY ASCENDING` — sorts results smallest to largest (A–Z, 1–10).
- `ORDER BY DESCENDING` — sorts results largest to smallest (Z–A, 10–1).
- `AND` / `OR` — combines multiple conditions within a `WHERE` clause.
- `SUM(field)` — aggregate function: returns the numeric total of a field across matching records.
- `COUNT(field)` — aggregate function: returns *how many* records matched the criteria.

SQL Query Execution Order — Flow Diagram



Always Trace SQL in This Exact Order

(1) Apply `WHERE` to filter records. (2) Apply `ORDER BY` to sort survivors. (3) Apply `SELECT` to choose columns.
Never apply `SELECT` first — in a trace question you may need columns not in the final output (e.g. a column used only in the `WHERE` clause).

SQL Examples

Example 1 — Basic extraction and sorting.
Find the name and age of every student older than 15, sorted from oldest to youngest.

```
SELECT Name, Age
FROM STUDENTS
WHERE Age > 15
ORDER BY DESCENDING Age;
```

Example 2 — Combining conditions with AND / OR.
Extract all fields for students in Grade 10, or who have scored more than 80 points.

```
SELECT *
FROM STUDENTS
WHERE Grade = 10 OR Points > 80;
```

Tracing SQL Output Step by Step

Query: `SELECT Name, Age FROM STUDENTS WHERE Age > 15 ORDER BY DESCENDING Age;`

StudentID	Name	Age	Grade
S001	Alice	16	10
S002	Bob	15	9
S003	Carol	17	11
S004	Dave	14	9

Step 1 — Apply WHERE Age >15: Alice (16) and Carol (17) survive. Bob (15) is not >15; Dave (14) fails.
Step 2 — Apply ORDER BY DESCENDING Age: Carol (17) first, then Alice (16).
Step 3 — Apply SELECT Name, Age: Only the Name and Age columns appear.

Output:

Name	Age
Carol	17
Alice	16

Example 3 — Aggregate functions SUM and COUNT.
Count students in Grade 11; find total points scored by Grade 9 students.

```
SELECT COUNT(StudentID)
FROM STUDENTS
WHERE Grade = 11;

SELECT SUM(Points)
FROM STUDENTS
WHERE Grade = 9;
```

COUNT Query — Worked Mark-Scheme Response

Q: What does `SELECT COUNT(StudentID) FROM STUDENTS WHERE Grade = 11;` return from the table above?

Working: Apply `WHERE Grade = 11`: only Carol (S003) qualifies. `COUNT(StudentID)` counts one matching record.
Output: A single cell containing the value **1**.

Mark-point guidance: 1 mark for correctly filtering to one record via `WHERE`; 1 mark for stating `COUNT` returns the number of matching records (not the StudentID value).

Confusing COUNT and SUM With the Underlying Field Values

`COUNT(StudentID)` returns *how many* records matched — not a StudentID value. Similarly, `SUM(Points)` returns a single numeric total — not a list of individual point values. Writing a list of records as the output of a `COUNT` or `SUM` query scores zero marks.