

SQL Handout

English edition

Contents

1	Querying basics	3
1.1	SELECT and FROM	3
1.2	Choosing columns	3
1.3	Filtering with WHERE	4
2	Filtering & logic	5
2.1	AND, OR, NOT	5
2.2	LIKE, IN and BETWEEN	5
2.3	NULL	6
3	Sorting & limiting	7
3.1	ORDER BY and LIMIT	7
4	Aggregates & grouping	9
4.1	Aggregate functions	9
4.2	GROUP BY and HAVING	9
5	Joins	11
5.1	Keys & relationships	11
5.2	INNER JOIN	11
5.3	Joining with grouping	12
5.4	LEFT JOIN	12
6	Modifying data	13
6.1	INSERT	13
6.2	UPDATE	13
6.3	DELETE	14
7	Defining tables	15
7.1	CREATE TABLE & data types	15
7.2	Keys & constraints	15
7.3	ALTER TABLE	16
8	Database design	17
8.1	Relationships & ER diagrams	17
8.2	Normalisation	18

1 Querying basics

1.1 ■ SELECT and FROM

A database keeps data in tables . A query reads data with SELECT. SELECT * returns every column; FROM names the table.

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT * FROM student;
```

table: student

id	name	score
1	'Mei'	88
2	'Sam'	71
3	'Ana'	95

a column (one field)

a row (one record)

01-table-anatomy

A table stores records: each row is one record, each column is one field

- Each row is one record . SQL keywords are written in UPPERCASE by habit.
- A statement ends with a semicolon ;.

1.2 ■ Choosing columns

List the columns you want, separated by commas. Use AS to rename a column in the result (an alias).

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT name, score AS mark FROM student;
```

A selected column can also be a calculation – pair it with AS to name the new column:

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
SELECT name, score + 5 AS bonus FROM student;
```

- SELECT DISTINCT col removes duplicate values from the result.

1.3 ■ Filtering with WHERE

WHERE keeps only the rows that match a condition . Compare with =, <> (not equal), <, >, <=, >=. Put text in single quotes.

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71), (3, 'Ana',
↵ 95);
SELECT name, score FROM student WHERE score >= 80;
```

- Read the parts in this logical order: FROM → WHERE → SELECT. (The database may do the work differently, but the result is the same.)

Common mistakes

- Put text values in single quotes: WHERE name = 'Ann'; column names have no quotes.
- SELECT * returns every column – name only the columns you need.
- WHERE filters rows; it comes after FROM.

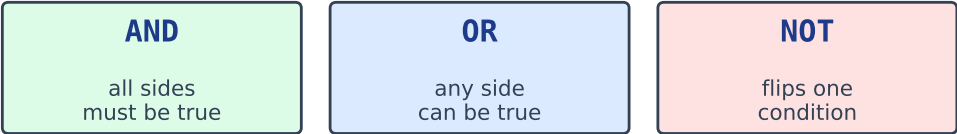
2 Filtering & logic

2.1 ■ AND, OR, NOT

Combine conditions with AND, OR, NOT. AND needs all sides true; OR needs any side true; NOT reverses one. Use brackets to set the precedence .

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
  ↪ (1, 'Mei', '11A', 88), (2, 'Sam', '11B', 71), (3, 'Ana', '11A', 95);
SELECT name FROM student WHERE form = '11A' AND score >= 90;
```

Combine WHERE conditions carefully



Exam tip: use brackets for precedence; AND is stricter than OR

AND needs all true; OR needs any; NOT flips one

2.2 ■ LIKE, IN and BETWEEN

LIKE matches a text pattern : % stands for any text and _ for one character – these are wildcards .

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
  ↪ (1, 'Mei', '11A', 88), (2, 'Sam', '11B', 71), (3, 'Ana', '11A', 95);
SELECT name FROM student WHERE name LIKE 'M%';           -- starts with M
```

Pattern	Matches
'M%'	starts with M
'%a'	ends with a
'%an%'	contains "an"
'M_i'	M, then exactly one character, then i

IN matches a set of values; BETWEEN matches a range (both ends included).

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
  ↪ (1, 'Mei', '11A', 88), (2, 'Sam', '11B', 71), (3, 'Ana', '11A', 95);
SELECT name, score FROM student
WHERE form IN ('11A', '11C') AND score BETWEEN 80 AND 100;
```

2.3 ■ NULL

NULL marks a missing value – nothing was stored in that cell. NULL is not 0 and not an empty string. A comparison with = or <> never matches it; test with IS NULL or IS NOT NULL.

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', NULL), (3, 'Ana',
  ↪ 95);
SELECT name FROM student WHERE score IS NULL;
```

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', NULL), (3, 'Ana',
  ↪ 95);
SELECT name, score FROM student WHERE score IS NOT NULL;
```

- WHERE score = NULL returns no rows at all – not even Sam’s row.
- A NULL in arithmetic gives NULL: score + 5 stays NULL for Sam.

Common mistakes

- Test for an empty value with IS NULL, never = NULL.
- In LIKE, % matches any text and _ matches one character: 'A%' means ”starts with A”.
- IN (1, 2, 3) is shorter than many ORs; BETWEEN a AND b includes both ends.

3 Sorting & limiting

3.1 ■ ORDER BY and LIMIT

ORDER BY sorts the result rows. Add DESC for descending (high to low); the default is ascending (low to high).

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71), (3, 'Ana', 95);
SELECT name, score FROM student ORDER BY score DESC;
```

3.1.1 Sort by more than one column

List several columns. A tie in the first column is broken by the next.

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 88), (3, 'Ana', 95);
SELECT name, score FROM student ORDER BY score DESC, name ASC;
```

3.1.2 LIMIT (top-N)

LIMIT n keeps only the first n rows – pair it with ORDER BY for a top-N list.

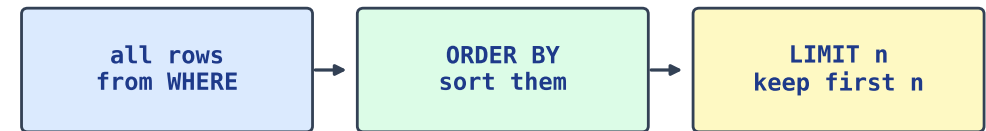
```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71), (3, 'Ana', 95);
SELECT name, score FROM student ORDER BY score DESC LIMIT 2;
```

- ORDER BY can also sort by an alias or a calculation: ORDER BY avg_score DESC.
- Text sorts alphabetically: ORDER BY name runs A → Z.

Common mistakes

- ORDER BY sorts ascending by default; add DESC for descending.
- ORDER BY comes near the end, after WHERE and GROUP BY.
- LIMIT caps how many rows come back, but only after sorting.

Sort first, then cut



Exam tip: ORDER BY before LIMIT; ASC is default; DESC reverses the order

ORDER BY sorts; LIMIT keeps the first n after sorting

4 Aggregates & grouping

4.1 ■ Aggregate functions

An aggregate function turns many rows into one summary value. Wrap an average in `ROUND(x, 2)` to tidy it.

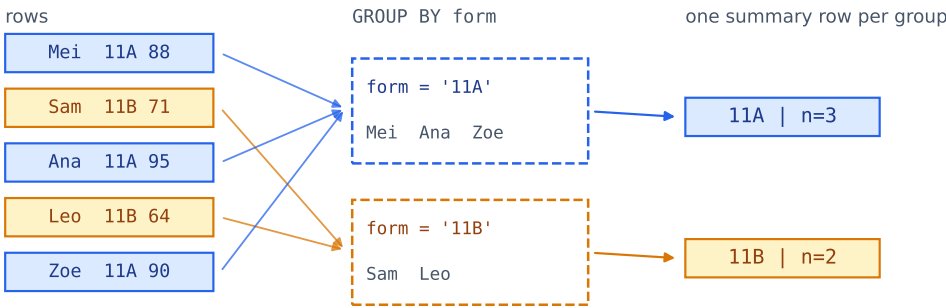
Function	Gives
<code>COUNT(*)</code>	how many rows
<code>SUM(col)</code>	the total
<code>AVG(col)</code>	the mean average
<code>MIN(col) / MAX(col)</code>	the smallest / largest value

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
  ↪ (1, 'Mei', '11A', 88), (2, 'Sam', '11B', 71), (3, 'Ana', '11A', 95);
SELECT COUNT(*), ROUND(AVG(score), 2), MAX(score) FROM student;
```

4.2 ■ GROUP BY and HAVING

`GROUP BY` makes one summary row per group . `HAVING` filters those groups – it is like `WHERE`, but it runs after grouping.

```
CREATE TABLE student (id INTEGER, name TEXT, form TEXT, score INTEGER);
INSERT INTO student VALUES
  ↪ (1, 'Mei', '11A', 88), (2, 'Sam', '11B', 71), (3, 'Ana', '11A', 95);
SELECT form, COUNT(*) AS n, ROUND(AVG(score), 2) AS avg_score
FROM student
GROUP BY form
HAVING COUNT(*) > 1;
```



SQL · `GROUP BY` collapses rows into groups; `HAVING` filters the groups

GROUP BY collects rows into one bucket per value; each bucket becomes one summary row

Write SQL clauses in their required syntax order. The table below is a logical model for reasoning about a query; the engine may optimise its physical execution order:

Step	Clause
1	FROM (and any JOIN)
2	WHERE – filter rows
3	GROUP BY – form groups
4	HAVING – filter groups
5	SELECT – compute the output columns
6	ORDER BY, then LIMIT

Common mistakes

- For predictable grouped results, group each selected non-aggregate column (or use a known functional dependency supported by the database). SQLite permits a bare column outside `GROUP BY`, but it may come from an arbitrary row in the group; a successful query does not guarantee the value you intended.
- Filter rows with `WHERE` (before grouping) and filter groups with `HAVING` (after).
- `COUNT(*)` counts rows; `COUNT(col)` skips NULLs in that column.

5 Joins

5.1 ■ Keys & relationships

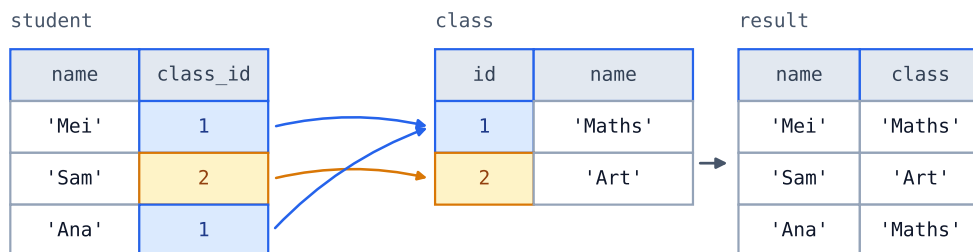
A primary key uniquely names each row in a table. A foreign key in one table points to the primary key of another – that builds a relationship between them.

```
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT, class_id
↳ INTEGER REFERENCES class(id));
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2);
SELECT * FROM student;
```

5.2 ■ INNER JOIN

A join combines rows from two tables. `INNER JOIN ... ON ...` keeps rows where the keys match. A short alias (`s`, `c`) keeps the query readable.

```
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT, class_id
↳ INTEGER REFERENCES class(id));
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2);
SELECT s.name, c.name AS class
FROM student s INNER JOIN class c ON s.class_id = c.id;
```



ON s.class_id = c.id

05-join

INNER JOIN matches each foreign key to a primary key and combines the matched rows

5.3 ■ Joining with grouping

Join first, then `GROUP BY` to summarise across the joined rows.

```
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT, class_id
↳ INTEGER REFERENCES class(id));
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', 2), (3, 'Ana', 1);
SELECT c.name AS class, COUNT(*) AS n
FROM student s INNER JOIN class c ON s.class_id = c.id
GROUP BY c.id, c.name;
```

Group by the key `c.id` as well as the name: two classes that happen to share a name would otherwise be merged into one row.

5.4 ■ LEFT JOIN

An `INNER JOIN` keeps only matched rows. A `LEFT JOIN` keeps **every** row of the left (first) table; where there is no match, the right table's columns come back as `NULL`.

```
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT, class_id
↳ INTEGER REFERENCES class(id));
INSERT INTO class VALUES (1, 'Maths'), (2, 'Art');
INSERT INTO student VALUES (1, 'Mei', 1), (2, 'Sam', NULL);
SELECT s.name, c.name AS class
FROM student s LEFT JOIN class c ON s.class_id = c.id;
```

- Sam has no class, but the row still appears – with class as `NULL`.
- To find only the unmatched rows, add `WHERE c.id IS NULL`.

Common mistakes

- A join with no `ON` condition pairs every row with every row (a cross join).
- Match the foreign key to the primary key: `ON orders.customer_id = customers.id`.
- An `INNER JOIN` drops rows that have no match on the other side; use a `LEFT JOIN` to keep them.

6 Modifying data

6.1 ■ INSERT

INSERT INTO ... VALUES ... adds new rows . Name the columns, then give the values in the same order. (Each block below ends with a SELECT so you can see the result.)

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student (id, name, score) VALUES (1, 'Mei', 88);
INSERT INTO student VALUES (2, 'Sam', 71);
SELECT * FROM student;
```

- INSERT INTO t VALUES (...), (...); adds several rows in one statement.



Exam tip: INSERT INTO table VALUES (...); column order must match

INSERT adds a new row to a table

6.2 ■ UPDATE

UPDATE ... SET ... WHERE ... changes existing rows. **Always** add WHERE, or every row changes.

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
UPDATE student SET score = 75 WHERE name = 'Sam';
SELECT * FROM student;
```

6.3 ■ DELETE

DELETE FROM ... WHERE ... removes rows. Without WHERE it empties the whole table .

```
CREATE TABLE student (id INTEGER, name TEXT, score INTEGER);
INSERT INTO student VALUES (1, 'Mei', 88), (2, 'Sam', 71);
DELETE FROM student WHERE score < 80;
SELECT * FROM student;
```

Common mistakes

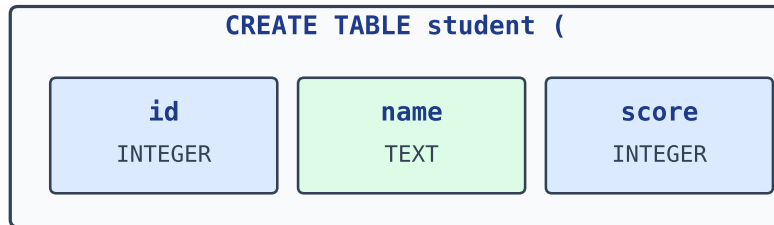
- UPDATE and DELETE **without** a WHERE change every row – always add the WHERE.
- In INSERT, the values must line up with the column list in order and type.
- Test a risky DELETE first as a SELECT with the same WHERE.

7 Defining tables

7.1 ■ CREATE TABLE & data types

CREATE TABLE defines a table's schema : the column names and their data types . The main SQLite types are INTEGER, TEXT, and REAL (a floating-point number: fast, but not always exact, so $0.1 + 0.2 = 0.3$ is false).

```
CREATE TABLE student (id INTEGER, name TEXT, score REAL);
INSERT INTO student VALUES (1, 'Mei', 88.5);
SELECT * FROM student;
```



Exam tip: name columns and types first; PRIMARY KEY marks the unique id

CREATE TABLE names columns and their types

7.2 ■ Keys & constraints

A constraint is a rule on a column: PRIMARY KEY (a unique id), NOT NULL (must have a value), UNIQUE, and DEFAULT (a fallback value).

```
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  score INTEGER DEFAULT 0
);
INSERT INTO student (id, name) VALUES (1, 'Mei');
SELECT * FROM student;
```

Declare a foreign key with REFERENCES. In SQLite, enable enforcement for each connection with `PRAGMA foreign_keys = ON;` before starting a transaction. A declaration alone does not guarantee enforcement.

```
PRAGMA foreign_keys = ON;
CREATE TABLE class (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  class_id INTEGER REFERENCES class(id)
);
INSERT INTO class VALUES (1, 'Maths');
INSERT INTO student VALUES (1, 'Mei', 1);
SELECT s.name, c.name AS class
FROM student s INNER JOIN class c ON s.class_id = c.id;
```

- The long form is FOREIGN KEY (class_id) REFERENCES class(id) on its own line.

7.3 ■ ALTER TABLE

ALTER TABLE ... ADD COLUMN ... changes the schema of a table that already exists. A DEFAULT fills the new column in the old rows.

```
CREATE TABLE student (id INTEGER, name TEXT);
INSERT INTO student VALUES (1, 'Mei');
ALTER TABLE student ADD COLUMN score INTEGER DEFAULT 0;
SELECT * FROM student;
```

- ALTER TABLE student RENAME TO pupil; renames the whole table.
- DROP TABLE student; deletes the table completely – structure **and** data.

Common mistakes

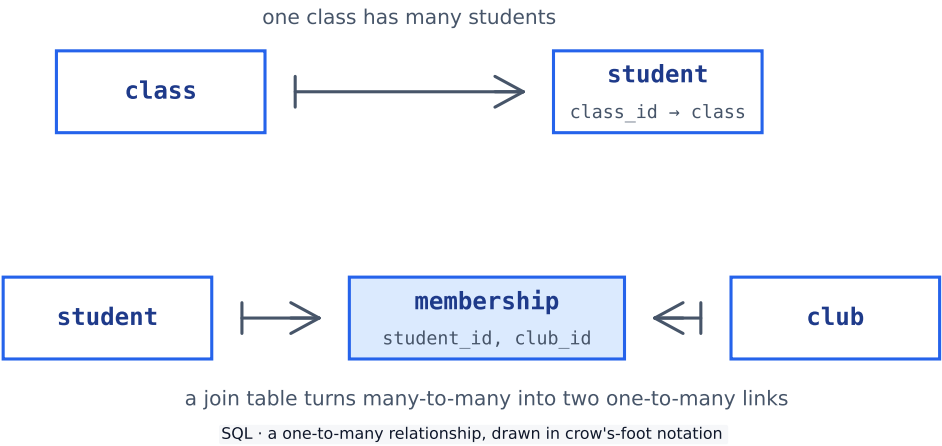
- Declare a suitable type for each column. SQLite permits omitted type names and normally uses flexible typing; a type declaration alone does not validate every stored value.
- Design a primary key to be unique and non-NULL. In ordinary SQLite tables, some primary-key declarations still permit NULL; use NOT NULL explicitly for a text key. INTEGER PRIMARY KEY has special row-id behavior: inserting NULL allocates an integer id.
- With foreign-key enforcement enabled, a non-NULL foreign-key value must match a referenced key. NULL is allowed unless the column also has NOT NULL.

8 Database design

8.1 ■ Relationships & ER diagrams

Tables connect through relationships. One-to-many is the most common: one class has many students. Many-to-many needs a join table in the middle. An entity-relationship diagram (ER diagram) draws each entity as a box and each relationship as a line.

Relationship	Example
one-to-one	a person and their passport
one-to-many	a class and its students
many-to-many	students and clubs



The “many” side carries the foreign key; a many-to-many link needs a join table

The join table holds one row per link, and each of its columns is a foreign key. Its primary key is the pair (student_id, club_id), a composite key, so the same student cannot join the same club twice. Both columns are NOT NULL, so each membership

names a student and a club. Here Mei is in two clubs, and Chess has two members:

```
CREATE TABLE student (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE club (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE membership (
  student_id INTEGER NOT NULL REFERENCES student(id),
  club_id INTEGER NOT NULL REFERENCES club(id),
  PRIMARY KEY (student_id, club_id)
);
INSERT INTO student VALUES (1, 'Mei'), (2, 'Sam');
INSERT INTO club VALUES (1, 'Chess'), (2, 'Art');
INSERT INTO membership VALUES (1, 1), (1, 2), (2, 1);
SELECT s.name, c.name AS club
FROM membership m
INNER JOIN student s ON m.student_id = s.id
INNER JOIN club c ON m.club_id = c.id;
```

8.2 ■ Normalisation

Normalisation organises tables to avoid redundancy (the same data repeated) and the update mistakes it causes. The first three normal forms:

- 1NF: every cell holds one atomic value – no lists inside a cell.
- 2NF: no column depends on only part of a composite key.
- 3NF: no column depends on another non-key column.

Unnormalised (bad)	Normalised (better)
student(name, club1, club2)	student(name) + membership(student, club)

Common mistakes

- Split repeating groups into their own table (normalisation) instead of many similar columns.
- Each table should describe ONE kind of thing.
- Link tables with a foreign key that points to another table’s primary key.