

SQL foundations Read. Predict. Query. Explain.

A detailed classroom reference and write-first practice handout. Main datasets: Rick & Morty and IMDb Movies. Northshore Coffee is an optional bridge into the official Lab 2.

How we will use the class

First half: the instructor demonstrates, then you try T1-T6. Second half: work on the official Lab 2 using Northshore Coffee. Keep the pair-practice questions on the website for extra time rather than crowding out the lab.

For a 75-minute session

0-5: recap and schemas. 5-15: SELECT and expressions, T1-T2. 15-27: WHERE and LIKE, T3-T4. 27-37: logic and NULL, T5-T6. 37-70: official Lab 2. 70-75: exit ticket. For a different class length, preserve the half-teaching / half-lab split.

The repeatable routine

1. Name the database and table. 2. Predict the output on paper. 3. Write and run one SQL statement. 4. Check one row, a boundary, or a counterexample. 5. Explain what the result does and does not establish. Partners alternate writer and checker.

What to keep open

Today's activity: learnwithabdou.com/courses/mis3220/today SQL playground:
learnwithabdou.com/courses/mis3220/sql-playground

T1-T4 use Rick & Morty; switch once to IMDb Movies for T5-T6. Select Northshore Coffee only for the optional bridge or official lab. Database and table names are different things: Rick & Morty -> scripts; IMDb Movies -> movies.

Print guide

Pages 2-9: schema and detailed notes. Pages 10-11: write-first questions. Page 12: optional Coffee bridge and submission reminders. No graded Lab 2 question text or answer key is reproduced here.



Two databases. Two meanings of a row.

Rick & Morty: scripts contains 1,905 dialogue records from 11 selected episode titles. An episode number can recur in another season. IMDb Movies: movies contains 1,000 film records in a historical teaching snapshot. These are separate databases; today does not use JOINS.

scripts - 1905 rows	Type	movies - 1000 rows	Type
id [PK]	INTEGER	id [PK]	INTEGER
lineIndex	INTEGER	poster	TEXT
seasonNumber	INTEGER	title	TEXT
episodeNumber	INTEGER	year	INTEGER
episodeName	TEXT	rating_cert	TEXT
characterName	TEXT	runtime	INTEGER
dialogue	TEXT	genre	TEXT
dialogueLength	INTEGER	imdb_score	REAL
wordCount	INTEGER	overview	TEXT
		meta_score	INTEGER
		director	TEXT
		star1	TEXT
		star2	TEXT
		star3	TEXT
		star4	TEXT
		votes	INTEGER
		gross	INTEGER

Read the labels carefully

scripts.id is a record identifier; lineIndex is its episode-local line position. characterName repeats across dialogue rows. wordCount and dialogueLength are stored counts. movies.id is a local row key, not an external IMDb title identifier. runtime is minutes; genre can be a combined text label. Missing gross or meta_score values are unknown, not zero.

Quick schema shows the loaded database's actual names. A table called characters is not present in this Rick & Morty dataset. Film fields belong to movies, not scripts.

1. SELECT, FROM and the shape of a result

One script row is one dialogue record, not one character or episode. Repeated speaker names are expected. First identify the table and what one row means; then choose the columns that answer the question.

SELECT chooses output columns and their order. FROM identifies the source table. SELECT * is useful for a first inspection; explicit columns make the intended answer clearer. Selecting fewer columns does not itself remove rows.

AS changes an output heading for this query only. It neither renames the stored column nor updates its values. FROM scripts AS s gives the table a temporary query nickname; s.characterName still refers to the original column.

Write text values in straight single quotes, identifiers normally unquoted, and aliases with spaces in double quotes. Commas separate SELECT expressions; do not put a comma immediately before FROM. Finish each statement with a semicolon.

For these single-table queries, reason through FROM, then WHERE, then SELECT, DISTINCT, ORDER BY and LIMIT. This is a useful logical model, not a promise about the engine's physical execution plan.

Choose a readable sample

```
SELECT id, characterName, episodeName
FROM scripts
ORDER BY id
LIMIT 3;
```

Give the output and table temporary names

```
SELECT s.id, s.characterName AS speaker,
       s.episodeName AS "Episode title"
FROM scripts AS s
ORDER BY s.id
LIMIT 3;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

2. Calculations, text and DISTINCT

A SELECT expression is calculated for each surviving row. You can return the original value beside a hypothetical value to check the arithmetic. +, -, *, / and % mean addition, subtraction, multiplication, division and integer remainder. Parentheses control arithmetic order.

SQLite divides two integers using integer division: 5 / 2 returns 2. Use 5 / 2.0 for 2.5. For movie hours, runtime / 60.0 avoids truncation. These SELECT calculations do not change stored data.

Use || to concatenate text in SQLite, including an explicit separator such as ' / '. + is arithmetic, not text concatenation. An expression involving NULL usually remains NULL; an absent value is not automatically an empty string.

DISTINCT removes duplicate output rows. DISTINCT characterName returns different speaker labels; DISTINCT characterName, episodeNumber returns distinct pairs. Adding id usually prevents duplicate removal because the id is unique.

Use output aliases in ORDER BY. For WHERE, repeat the original expression or use the source column. SQLite permits some aliases in WHERE as an extension, but relying on that is confusing and not portable.

A hypothetical edit, without changing stored words

```
SELECT id, wordCount,
       wordCount + 2 AS words_after_intro,
       characterName || ' / ' || episodeName AS label
FROM scripts ORDER BY id LIMIT 3;
```

Different speaker labels

```
SELECT DISTINCT characterName
FROM scripts
ORDER BY characterName;
```

Check integer division directly

```
SELECT 5 / 2 AS integer_result,
       5 / 2.0 AS decimal_result,
       17 % 5 AS remainder;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

3. WHERE: exact values, lists and boundaries

WHERE retains a row only when its condition is true. You may filter on a column that you do not display. Filtering rows and choosing columns are separate decisions.

= means equal; <> and != mean not equal. > and < exclude the boundary; >= and <= include it. Translate "more than" differently from "at least", and "under" differently from "at most". Write numeric thresholds without quotes.

IN is an exact-value list for one expression: `characterName IN ('Rick', 'Morty')` means Rick OR Morty. It is not pattern matching. Every OR condition must be a complete comparison: `characterName = 'Rick' OR 'Morty'` is not a reliable way to ask this question.

BETWEEN 8 AND 12 includes both 8 and 12. Its equivalent is `wordCount >= 8 AND wordCount <= 12`. NOT BETWEEN excludes known values inside that range, but it does not automatically include missing values.

This database contains selected episodes, not every episode. An empty result does not prove the SQL is broken: check the values that actually exist. Do not invent a table called `characters` or a column called `season`.

Both a speaker and a word-count boundary

```
SELECT id, characterName, wordCount
FROM scripts
WHERE characterName = 'Summer' AND wordCount >= 10
ORDER BY id;
```

Two exact speakers in an inclusive range

```
SELECT id, characterName, wordCount
FROM scripts
WHERE characterName IN ('Beth', 'Jerry')
  AND wordCount BETWEEN 8 AND 12
ORDER BY id;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

4. LIKE: match the text, not its meaning

Use = for an exact stored value and LIKE for a text pattern. `characterName = 'Rick%'` looks for the literal percent sign; it does not mean "starts with Rick".

LIKE `'Rick%'` matches a prefix. LIKE `'%Rick'` matches a suffix. LIKE `'%Rick%'` matches Rick anywhere. `%` matches zero or more characters, so Rick itself also matches `Rick%`.

`_` matches exactly one character. LIKE `'R____'` matches four-character values beginning with R. The underscore is a wildcard only inside a LIKE pattern; underscores in names such as `imdb_score` are ordinary identifier characters.

The classroom SQLite defaults make LIKE case-insensitive for ASCII letters (Rick and rick), while ordinary text equality here uses the default case-sensitive comparison. Do not assume those rules across every language, collation or database.

LIKE does not understand synonyms, whole words or genre categories. A movie genre field can contain a combined label such as `Crime, Drama`: `= 'Drama'` and LIKE `'%Drama%'` ask different questions.

To put an apostrophe in SQL text, double it: `'Schindler's List'`. To search for a literal wildcard, choose an escape character: LIKE `'%!_%' ESCAPE '!'` finds an underscore. These are optional reference details.

Find episode labels containing a word

```
SELECT DISTINCT episodeName
FROM scripts
WHERE episodeName LIKE '%Dog%'
ORDER BY episodeName;
```

Four-character speaker labels beginning with R

```
SELECT DISTINCT characterName
FROM scripts
WHERE characterName LIKE 'R____'
ORDER BY characterName;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

5. AND, OR, NOT and parentheses

AND requires both tests to be true. OR requires at least one. NOT reverses true/false, but NOT unknown remains unknown when NULL is involved. Predict a specific row before running the full query.

AND binds more tightly than OR. `year >= 2000 AND runtime <= 100 OR imdb_score >= 8.5` means (recent AND short) OR highly rated: it can admit older highly rated movies.

If every movie must be recent, write `year >= 2000 AND (runtime <= 100 OR imdb_score >= 8.5)`. Parentheses express the business rule; they are not decorative.

Use the real IMDb Movies snapshot here: one movies row is one film. runtime is in minutes; imdb_score is the stored score. This is a historical teaching snapshot, not today's ratings or a complete census of films.

Validate meaning, not just syntax. A query can run successfully and still include the wrong rows. Explain one qualifying row and invent a boundary case that should fail.

A recent short-or-highly-rated watchlist

```
SELECT id, title, year, runtime, imdb_score
FROM movies
WHERE year >= 2000
      AND (runtime <= 100 OR imdb_score >= 8.5)
ORDER BY id;
```

Known titles outside a pair of years

```
SELECT id, title, year
FROM movies
WHERE year NOT IN (1999, 2000)
ORDER BY id LIMIT 5;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

6. NULL: unknown is not zero

NULL represents missing or unknown data. It differs from 0, an empty string "", and the literal text 'NULL'. A blank-looking result cell is not enough evidence: use a SQL test.

Use IS NULL and IS NOT NULL. `gross = NULL` and `gross <> NULL` evaluate to unknown, so WHERE keeps no rows from those tests. They can run with no syntax error while answering the wrong question.

A missing gross value does not mean a film earned no money. A missing meta_score does not mean a score of zero. Report the missingness instead of inventing a value.

Be explicit when a rule must include missing values: `meta_score < 60 OR meta_score IS NULL`. NOT IN is especially tricky if its list contains NULL; avoid that construction in today's work.

COALESCE can supply a display fallback, but replacing missing numbers with zero changes their meaning. It is a later tool, not a substitute for deciding what the missing value means.

Find an evidence gap

```
SELECT id, title, year, meta_score
FROM movies
WHERE meta_score IS NULL AND year >= 2010
ORDER BY id;
```

Keep only recorded box-office values

```
SELECT id, title, gross
FROM movies
WHERE gross IS NOT NULL
ORDER BY id LIMIT 5;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

7. Predictable previews and debugging

ORDER BY controls display order: ASC (the default) is smallest first; DESC is largest first. Add a unique id as a final tie-breaker when you need repeatable selection among equal scores. This is a short preview of next week, not an extra Lab 2 requirement.

LIMIT 5 returns at most five rows after filtering and ordering. Without ORDER BY, the chosen rows are unspecified; that does not mean SQLite randomly samples them. OFFSET skips rows in that ordered sequence. DISTINCT does not guarantee sorting.

The playground displays up to 200 result rows and labels a truncated result. A displayed count of 200 is not necessarily the total number of matching records. Narrow the question or use an instructor-provided COUNT(*) check; aggregation is not today's required topic.

Use Quick schema or the database reference to confirm names. Autocomplete helps with table and column names; it does not write the reasoning for you. Run one SQL statement at a time. The sqlite3 shell command .schema is not SQL and will not run here.

If a query fails, check the selected database, FROM table, spelling, commas and quotes. If it runs but seems wrong, check boundaries, parentheses, NULL and patterns. Start simple and add one condition at a time.

Save SQL as text before leaving. A screenshot is useful for discussion but does not satisfy the official Lab 2 requirement to submit runnable query text in the Word report. SELECT does not edit the source data.

The five highest stored scores, with a stable tie-break

```
SELECT id, title, imdb_score
FROM movies
ORDER BY imdb_score DESC, id
LIMIT 5;
```

A second page of a stable title list

```
SELECT id, title
FROM movies
ORDER BY id
LIMIT 5 OFFSET 5;
```

Predict one result before running. These are demonstration examples; use the numbered T questions for your own attempt.

Your turn: T1-T3

Name: _____ Partner: _____

Write the SQL or a prediction below before running it. Then record a result and explain the reasoning. These are ungraded foundation exercises, not the official lab questions.

T1. Choose the evidence / Rick & Morty

Return id, lineIndex and episodeName (label it episode) for the first four records by id. Before running, predict the number of rows and columns.

Explain: Why do id and lineIndex start differently? Is this a list of four different episodes?

T2. Make a useful display label / Rick & Morty

For the first four records by id, return id, wordCount, wordCount * 3 AS triple_words, and characterName || ' / ' || episodeName AS label. Explain what remains stored afterward.

Explain: Check one multiplication by hand. Does the expression alter wordCount?

T3. Find concise lines in two episodes / Rick & Morty

Across the included seasons, find records from episodeNumber 2 or 4 whose wordCount is between 8 and 12, inclusive. Show id, episodeNumber, characterName and wordCount, ordered by id.

Explain: Would counts of 8 and 12 pass? Would an episode 3 record with 10 words pass?

Your turn: T4-T6

Name: _____ Partner: _____

Write the SQL or a prediction below before running it. Then record a result and explain the reasoning. These are ungraded foundation exercises, not the official lab questions.

T4. Search episode names, then check length / Rick & Morty

Find records where episodeName contains Rick and wordCount is at least 25. Return id, episodeName and wordCount, ordered by id.

Explain: Why does episodeName = '%Rick%' ask a different question?

T5. A watchlist with two alternatives / IMDb Movies

Find films released from 2010 onward that either last no more than 95 minutes OR have imdb_score at least 8.5. Show id, title, year, runtime and imdb_score, ordered by id.

Explain: Would a 1994 film with a score of 9.3 qualify? Explain where the parentheses belong.

T6. Find missing evidence / IMDb Movies

Find films from 2010 onward with a NULL gross value and an imdb_score of at least 8.0. Return id, title, year, imdb_score and gross, ordered by id.

Explain: Does this establish that these films earned zero money? Why would = NULL fail silently?

Northshore Coffee: use only when ready

The official Lab 2 uses shops, menu_items and employees. The orders table is in the database but is reserved for later weeks. A menu row may represent a particular size; repeated item names are not automatically duplicate mistakes.

shops - 8 rows

id [PK], name, city, address, phone, open_date, has_drive_thru

menu_items - 45 rows

id [PK], name, category, size, price, calories, is_seasonal, description

employees - 25 rows

id [PK], first_name, last_name, email, phone, shop_id, position, hire_date, hourly_wage

orders - 204 rows (later weeks)

id [PK], shop_id, employee_id, order_date, total_amount, payment_method, tip_amount

C1. A different menu question

Show id, name, category and calories for seasonal items with no calorie count on file. Order by id. Use menu_items.

Explain: Why is an unknown calorie count not a zero-calorie promise?

C2. A different location question

Show id, name, city and has_drive_thru for shops in Tofte or Beaver Bay. Order by id and explain the 1/0 flag without assuming all shops in a city have the same value.

Explain: Which part selects rows, and which part decides the displayed columns?

For the second half of class

Open Lab 2 on the website and follow the official Canvas handout: Q1-Q15, Check the AI, your client transfer and your own reflections. Each student submits independently. Use the Word template, paste SQL as text plus its result and a one-line answer, and save Lab02_LastName.docx. The handout states September 15 at 2:00 PM; confirm the current Canvas window. Screenshots of SQL do not replace runnable query text.

Exit ticket

Explain = versus LIKE; repair the idea behind = NULL; describe one boundary or parentheses mistake you can now catch.

Sources and scope

Aligned to MIS 3220 Week 2 notes (Sept 8 and 10), Pre-Lab 2, and the original Lab 2 handout. Original examples were checked against the actual classroom database files and browser SQL engine. SQLite references: sqlite.org/lang_select.html and sqlite.org/lang_expr.html. ORDER BY is a preview; JOINS and aggregation are outside today's required scope.