A VISUAL REFERENCE · SQLite

SQLite, one change at a time

From a first SELECT to relational queries, transactions, indexes, JSON and full-text search. Complete source. A small change. Its actual result.

The Org file is the canonical source. Read the HTML guide beside your work, or download its PDF and source for offline reading.

> **Last changes: 2026-10-31.** Introduced runnable SQLite examples of queries, joins, window functions, transactions, indexes, JSON and full-text search, with verified results and downloadable databases.

96
worked examples

16
topic families

1
canonical Org source

READING THE TOUR

Three ways to see a change

SQL program

Read the query beside its real result; expand the complete schema and fixture.

Actual rows

Inspect typed SQLite output, NULL values, query plans and expected constraint failures.

One change

Follow a comparison, run the standalone SQL or explore its downloadable database.

Read from the beginning or jump to a stable example ID. Each example includes its complete input and captured result. Copy link preserves the address of the guide you are reading, including its current domain and path.

The starting atlas supplies twenty-four relational questions. This guide retains all twenty-four topics, develops them into complete independent programs, and adds schema design, data changes, transactions, indexes, JSON, full-text search and practical report patterns.

Every example starts with a new database. The concise view shows the statements to study; expand the complete program for its schema and fixture. No example depends on having run an earlier one. Run a downloaded SQL file in a fresh working directory with `sqlite3 -header -box :memory: < SQL-01-001.sql`. The WAL example needs a fresh file database instead of `:memory:`. The VACUUM INTO example writes `snapshot.sqlite3` in its working directory.

Results are actual output from the native sqlite3 CLI. The guide renders its JSON rows as tables. `NULL` means an absent SQL value; italic *empty text* is an empty string. A failed constraint is labelled as an expected diagnostic. Only nonempty result sets produce JSON arrays in this CLI format. Downloads include the standalone SQL, captured results and a data snapshot after the example. Downloads use rollback-journal mode to remain self-contained; the WAL example still executes in WAL mode and reports that native result. If a failed program leaves a transaction open, connection close rolls it back before the database is saved.

Starting database

The shared fixture has eight synthetic observations. Values total 100. Site D has no lookup label, the seventh calendar day has no observation, and notes contain NULL. The site column is deliberately not a declared foreign key, so lookup gaps remain available for join examples. Dedicated constraint examples create actual foreign-key relationships with enforcement enabled.

id	site	value	day	note
1	A	12	2026-01-01	NULL
2	B	20	2026-01-02	checked
3	A	18	2026-01-03	NULL
4	B	8	2026-01-04	recheck
5	C	10	2026-01-05	NULL
6	A	6	2026-01-06	checked
7	C	10	2026-01-08	archived
8	D	16	2026-01-09	NULL

Lookup labels: A = River; B = Hill; C = Coast. Full-text examples use a separate four-document synthetic corpus in their complete source. All dates and input values are fixed. Query-plan wording depends on the SQLite version; these examples show actual chosen plans rather than timing claims. Full preparation requires SQLite 3.45 or newer with JSON and FTS5.

References and provenance

- Starting atlas: ROOT SQLite, pinned commit 26039d49fe2e6399552114160e7c7850fd6c4fb9
- SELECT and relational queries
- Window functions and frames
- Common table expressions and recursion
- STRICT tables
- Transactions and savepoints
- EXPLAIN QUERY PLAN
- JSON functions and JSONB
- FTS5 queries and ranking
- Native command-line shell

The atlas supplies the starting questions. This guide uses a new, explicitly listed fixture and independent explanations, comparisons and added examples. All twenty-four original topics are mapped in ATLAS_PANEL properties and `sources/atlas-provenance.json`. No poster builder or external dataset is needed. SQLite and its installed extensions remain external dependencies.

01 · A first SELECT

SQL-01-001

Select stored columns

A projection chooses which fields answer the question. An explicit ORDER BY defines the returned row order.

SQL to study

```
SELECT id,site,value FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	site	value
1	A	12.0
2	B	20.0
3	A	18.0
4	B	8.0
5	C	10.0
6	A	6.0
7	C	10.0
8	D	16.0

SQLite · exit 0

Clause or feature	Meaning
`SELECT`	Choose the output columns.
`FROM sample`	Read the shared observation table.
`ORDER BY id`	Return a deterministic order.

SQL-01-002

Filter individual observations

WHERE tests individual rows before grouping or window calculations. Here the cutoff includes values equal to sixteen.

Compare with SQL-01-001 · Select stored columns.

SQL to study

```
SELECT id,site,value
FROM sample WHERE value>=16 ORDER BY id;
```

Actual SQLite result

Result 1 · 3 rows

id	site	value
2	B	20.0
3	A	18.0
8	D	16.0

SQLite · exit 0

Clause or feature	Meaning
`WHERE`	Retain only rows whose condition is true.
`>=16`	Include the threshold itself.

SQL-01-003

Find the top three

Sort the largest values first, then use the primary key as a tie-breaker. LIMIT selects a deterministic top set.

SQL to study

```
SELECT id,value FROM sample
ORDER BY value DESC,id LIMIT 3;
```

Actual SQLite result

Result 1 · 3 rows

id	value
2	20.0
3	18.0
8	16.0

SQLite · exit 0

Clause or feature	Meaning
`DESC`	Sort values from largest to smallest.
`id`	Break any ties consistently.
`LIMIT 3`	Return at most three rows.

SQL-01-004

Return distinct values

DISTINCT removes duplicate site values. Ordering is a separate operation and remains explicit.

SQL to study

```
SELECT DISTINCT site FROM sample ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site
A
B
C
D

SQLite · exit 0

Clause or feature	Meaning
`DISTINCT`	Return each selected value once.
`ORDER BY site`	Sort the unique results.

SQL-01-005

Calculate a derived column

Expressions can calculate new columns without changing stored observations. AS supplies a useful result name.

SQL to study

```
SELECT id,value*2 AS doubled
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	doubled
1	24.0
2	40.0
3	36.0
4	16.0
5	20.0
6	12.0
7	20.0
8	32.0

SQLite · exit 0

Clause or feature	Meaning
`value*2`	Calculate from each stored value.
`AS doubled`	Name the result column.

SQL-01-006

Classify with CASE

CASE turns an explicit rule into a derived label. Every row receives either high or low.

SQL to study

```
SELECT id,CASE WHEN value>=16 THEN 'high'
               ELSE 'low' END AS class
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	class
1	low
2	high
3	high
4	low
5	low
6	low
7	low
8	high

SQLite · exit 0

Clause or feature	Meaning
`CASE WHEN`	Evaluate a classification condition.
`ELSE`	Provide the alternative class.

02 · Filtering, NULL and ordering

SQL-02-001

Display a value for NULL

COALESCE substitutes a display value for an unknown note. The stored NULL remains unchanged.

SQL to study

```
SELECT id,coalesce(note,'(missing)') AS note
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	note
1	(missing)
2	checked
3	(missing)
4	recheck
5	(missing)
6	checked
7	archived
8	(missing)

SQLite · exit 0

Clause or feature	Meaning
`coalesce`	Return the first non-NULL argument.
`NULL`	Represent an absent value, not an empty string.

SQL-02-002

Test NULL explicitly

Equality with NULL produces unknown. IS NULL is the predicate for absent values.

Compare with SQL-02-001 · Display a value for NULL.

SQL to study

```
SELECT id,note FROM sample
WHERE note IS NULL ORDER BY id;
```

Actual SQLite result

Result 1 · 4 rows

id	note
1	*NULL*
3	*NULL*
5	*NULL*
8	*NULL*

SQLite · exit 0

Clause or feature	Meaning
`IS NULL`	Test whether a value is absent.
`WHERE`	Unknown conditions do not pass the filter.

SQL-02-003

Combine range and membership filters

BETWEEN includes both endpoints. IN expresses a finite membership test without repeating equality comparisons.

SQL to study

```
SELECT id,site,value FROM sample
WHERE value BETWEEN 10 AND 18
  AND site IN ('A','C') ORDER BY id;
```

Actual SQLite result

Result 1 · 4 rows

id	site	value
1	A	12.0
3	A	18.0
5	C	10.0
7	C	10.0

SQLite · exit 0

Clause or feature	Meaning
`BETWEEN`	Use an inclusive numeric interval.
`IN`	Accept either named site.
`AND`	Require both conditions.

SQL-02-004

Match and normalize text

LIKE supports percent as a variable-length wildcard. This example also exposes lowercasing as a distinct calculation.

SQL to study

```
SELECT id,lower(note) AS note FROM sample
WHERE note LIKE 'check%' ORDER BY id;
```

Actual SQLite result

Result 1 · 2 rows

id	note
2	checked
6	checked

SQLite · exit 0

Clause or feature	Meaning
`LIKE check%`	Match the chosen prefix.
`lower`	Normalize the displayed text; it does not update storage.

SQL-02-005

Place NULL values last

NULLS LAST makes the missing-value placement explicit. A primary-key tie-breaker fixes the order among equal notes.

SQL to study

```
SELECT id,note FROM sample
ORDER BY note NULLS LAST,id;
```

Actual SQLite result

Result 1 · 8 rows

id	note
7	archived
2	checked
6	checked
4	recheck
1	*NULL*
3	*NULL*
5	*NULL*
8	*NULL*

SQLite · exit 0

Clause or feature	Meaning
`NULLS LAST`	Place absent notes after actual strings.
`id`	Stabilize ties within each note value.

SQL-02-006

Continue with keyset pagination

A known key gives the starting point for the next page. This example returns the next three primary keys after four.

SQL to study

```
SELECT id,site FROM sample
WHERE id>4 ORDER BY id LIMIT 3;
```

Actual SQLite result

Result 1 · 3 rows

id	site
5	C
6	A
7	C

SQLite · exit 0

Clause or feature	Meaning
`id>4`	Continue after the last seen key.
`ORDER BY id`	Use the same key for filtering and ordering.
`LIMIT`	Bound the page size.

03 · Expressions, text and dates

SQL-03-001

Distinguish integer and real arithmetic

Operand types affect division. Cast deliberately when a fractional result is required.

SQL to study

```
SELECT 5/2 AS integer_division,
       5.0/2 AS real_division,
       CAST(5 AS REAL)/2 AS cast_division;
```

Actual SQLite result

Result 1 · 1 rows

integer_division	real_division	cast_division
2	2.5	2.5

SQLite · exit 0

Clause or feature	Meaning
`Integer division`	Two integer operands produce an integer quotient.
`CAST AS REAL`	Choose real arithmetic explicitly.

SQL-03-002

Format a consistent identifier

printf applies a fixed-width numeric format; lower changes the site label independently.

SQL to study

```
SELECT printf('S%02d',id) AS label,lower(site) AS site
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

label	site
S01	a
S02	b
S03	a
S04	b
S05	c
S06	a
S07	c
S08	d

SQLite · exit 0

Clause or feature	Meaning
`printf S%02d`	Pad integer identifiers with a leading zero.
`lower`	Convert ASCII site letters to lowercase.

SQL-03-003

Clean and join text

Text functions can clean an input and build a display label while preserving the original stored values.

SQL to study

```
SELECT trim('  River  ') AS cleaned,
       'site-'||lower('A') AS slug,
       replace('north_bank','_',' ') AS title;
```

Actual SQLite result

Result 1 · 1 rows

cleaned	slug	title
River	site-a	north bank

SQLite · exit 0

Clause or feature	Meaning
`trim`	Remove surrounding spaces.
`||`	Concatenate text values.
`replace`	Substitute an exact substring.

SQL-03-004

Guard a denominator with NULLIF

NULLIF converts a zero denominator into NULL. COALESCE supplies a display fallback only after the calculation.

SQL to study

```
SELECT 12.0/nullif(0,0) AS unknown_ratio,
       coalesce(12.0/nullif(0,0),0) AS display_ratio,
       round(12.0/5,2) AS rounded;
```

Actual SQLite result

Result 1 · 1 rows

unknown_ratio	display_ratio	rounded
NULL	0	2.4

SQLite · exit 0

Clause or feature	Meaning
`nullif(a,b)`	Return NULL when the arguments are equal.
`Display fallback`	Zero here is a presentation choice, not the unknown mathematical result.

SQL-03-005

Measure elapsed days

ISO date text and julianday provide a reproducible elapsed-day calculation. The reference date is fixed rather than taken from the clock.

SQL to study

```
SELECT id,CAST(julianday(day)-julianday('2026-01-01')
              AS INTEGER) AS days
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	days
1	0
2	1
3	2
4	3
5	4
6	5
7	7
8	8

SQLite · exit 0

Clause or feature	Meaning
`julianday`	Map a date to a numeric day count.
`Fixed origin`	Keep this calculation reproducible.

SQL-03-006

Calculate a calendar boundary

Date modifiers are applied from left to right. Compute the last day of the chosen month without relying on its length.

SQL to study

```
SELECT date('2026-02-12','start of month','+1 month','-1 day')
       AS month_end,
       strftime('%Y-%m','2026-02-12') AS month;
```

Actual SQLite result

Result 1 · 1 rows

month_end	month
2026-02-28	2026-02

SQLite · exit 0

Clause or feature	Meaning
`start of month`	Move to day one.
`+1 month then -1 day`	Move to the last day of the original month.
`strftime`	Choose a date display format.

04 · Aggregates and grouped questions

SQL-04-001

Summarize the entire table

Aggregates collapse all observations into one summary row. COUNT(*) counts rows regardless of missing notes.

SQL to study

```
SELECT count(*) AS n,round(avg(value),2) AS mean,
       min(value) AS minimum,max(value) AS maximum
FROM sample;
```

Actual SQLite result

Result 1 · 1 rows

n	mean	minimum	maximum
8	12.5	6.0	20.0

SQLite · exit 0

Clause or feature	Meaning
`count(*)`	Count observation rows.
`avg / min / max`	Compute summary statistics over values.

SQL-04-002

Group observations by site

GROUP BY changes the unit of the result from an observation to a site. Each output row is a group summary.

SQL to study

```
SELECT site,count(*) AS n,sum(value) AS total
FROM sample GROUP BY site ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site	n	total
A	3	36.0
B	2	28.0
C	2	20.0
D	1	16.0

SQLite · exit 0

Clause or feature	Meaning
`GROUP BY site`	Form one group per site.
`ORDER BY`	Define the order of the grouped results.

SQL-04-003

Filter groups with HAVING

WHERE filters input rows; HAVING filters completed groups. Here only group totals above twenty-five survive.

Compare with SQL-04-002 · Group observations by site.

SQL to study

```
SELECT site,sum(value) AS total FROM sample
GROUP BY site HAVING sum(value)>25 ORDER BY site;
```

Actual SQLite result

Result 1 · 2 rows

site	total
A	36.0
B	28.0

SQLite · exit 0

Clause or feature	Meaning
`HAVING`	Test an aggregate after grouping.
`sum(value)>25`	Keep only groups that meet the total threshold.

SQL-04-004

Count missing and present values

COUNT(column) ignores NULL. FILTER restricts one aggregate without removing rows from the other aggregates.

SQL to study

```
SELECT site,count(*) AS n,count(note) AS noted,
       count(*) FILTER (WHERE note IS NULL) AS missing
FROM sample GROUP BY site ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site	n	noted	missing
A	3	1	2
B	2	2	0
C	2	1	1
D	1	0	1

SQLite · exit 0

Clause or feature	Meaning
`count(note)`	Count only non-NULL notes.
`FILTER`	Apply a condition to one aggregate.

SQL-04-005

Calculate a weighted mean

The weights are explicit inputs, not observation frequencies inferred from the row count.

SQL to study

```
WITH weights(id,w) AS (VALUES(1,1),(2,2),(3,1))
SELECT sum(s.value*w.w)/sum(w.w) AS weighted_mean
FROM sample AS s JOIN weights AS w ON w.id=s.id;
```

Actual SQLite result

Result 1 · 1 rows

weighted_mean
17.5

SQLite · exit 0

Clause or feature	Meaning
`Weighted numerator`	Multiply every value by its weight.
`Weight total`	Divide by the sum of weights, not the number of rows.

SQL-04-006

Make an ordered list per group

Ordering inside the aggregate determines the list order. An outer ORDER BY independently orders the groups.

SQL to study

```
SELECT site,group_concat(id,',' ORDER BY id) AS ids
FROM sample GROUP BY site ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site	ids
A	1,3,6
B	2,4
C	5,7
D	8

SQLite · exit 0

Clause or feature	Meaning
`group_concat`	Combine values with a chosen separator.
`Aggregate ORDER BY`	Specify list order explicitly; supported since SQLite 3.44.

05 · Joins and missing relationships

SQL-05-001

Match a lookup with an inner join

Only observations with a matching lookup row survive. Site D deliberately has no lookup in this fixture.

SQL to study

```
SELECT s.id,t.label FROM sample AS s
JOIN site AS t ON t.id=s.site ORDER BY s.id;
```

Actual SQLite result

Result 1 · 7 rows

id	label
1	River
2	Hill
3	River
4	Hill
5	Coast
6	River
7	Coast

SQLite · exit 0

Clause or feature	Meaning
`JOIN ON`	Require a matching site key.
`Aliases`	Distinguish observation columns from lookup columns.

SQL-05-002

Keep observations with missing metadata

A LEFT JOIN preserves every observation. The unmatched lookup fields become NULL.

Compare with SQL-05-001 · Match a lookup with an inner join.

SQL to study

```
SELECT s.id,t.label FROM sample AS s
LEFT JOIN site AS t ON t.id=s.site ORDER BY s.id;
```

Actual SQLite result

Result 1 · 8 rows

id	label
1	River
2	Hill
3	River
4	Hill
5	Coast
6	River
7	Coast
8	*NULL*

SQLite · exit 0

Clause or feature	Meaning
`LEFT JOIN`	Preserve the rows from the left side.
`NULL label`	Expose the missing lookup for D.

SQL-05-003

Find missing relationships

NOT EXISTS expresses an anti-join. It reports observations whose site key has no lookup row.

SQL to study

```
SELECT s.id,s.site FROM sample AS s
WHERE NOT EXISTS(SELECT 1 FROM site AS t WHERE t.id=s.site)
ORDER BY s.id;
```

Actual SQLite result

Result 1 · 1 rows

id	site
8	D

SQLite · exit 0

Clause or feature	Meaning
`NOT EXISTS`	Keep a row only when the correlated lookup is empty.
`Correlation`	Match the lookup to the current observation.

SQL-05-004

Compare an ON filter with a WHERE filter

A right-side condition in ON controls matching. The same condition in WHERE removes the unmatched rows after the join.

Compare with SQL-05-002 · Keep observations with missing metadata.

SQL to study

```
SELECT 'ON' AS placement,count(*) AS n
FROM sample AS s LEFT JOIN site AS t
 ON t.id=s.site AND t.label='River'
UNION ALL
SELECT 'WHERE',count(*)
FROM sample AS s LEFT JOIN site AS t ON t.id=s.site
WHERE t.label='River'
ORDER BY placement;
```

Actual SQLite result

Result 1 · 2 rows

placement	n
ON	8
WHERE	3

SQLite · exit 0

Clause or feature	Meaning
`ON condition`	Preserve unmatched left rows.
`WHERE condition`	Require a matching River label after the join.

SQL-05-005

See join multiplicity

A one-to-many relationship can return an observation more than once. This is expected join behaviour, not an accidental duplicate removal problem.

SQL to study

```
CREATE TABLE tag(sample_id INTEGER,tag TEXT);
INSERT INTO tag VALUES(1,'red'),(1,'checked'),(2,'blue');
SELECT s.id,t.tag FROM sample AS s
JOIN tag AS t ON t.sample_id=s.id ORDER BY s.id,t.tag;
```

Actual SQLite result

Result 1 · 3 rows

id	tag
1	checked
1	red
2	blue

SQLite · exit 0

Clause or feature	Meaning
`One-to-many`	One observation can have several tags.
`Result unit`	The output now represents an observation-tag pair.

SQL-05-006

Join a table to itself

Two aliases let one table play two roles. Date arithmetic finds observations exactly one day apart.

SQL to study

```
SELECT a.id AS first_id,b.id AS next_id
FROM sample AS a JOIN sample AS b
 ON b.day=date(a.day,'+1 day') ORDER BY a.id,b.id;
```

Actual SQLite result

Result 1 · 6 rows

first_id	next_id
1	2
2	3
3	4
4	5
5	6
7	8

SQLite · exit 0

Clause or feature	Meaning
`Two aliases`	Refer to different rows of the same table.
`Date relation`	Require a one-day gap, not merely consecutive IDs.

06 · Subqueries, CTEs and sets

SQL-06-001

Compare with a correlated group mean

The subquery depends on the outer row site. It compares each observation with its own group rather than the global mean.

SQL to study

```
SELECT s.id,s.site,s.value FROM sample AS s
WHERE s.value>(SELECT avg(t.value) FROM sample AS t
               WHERE t.site=s.site) ORDER BY s.id;
```

Actual SQLite result

Result 1 · 2 rows

id	site	value
2	B	20.0
3	A	18.0

SQLite · exit 0

Clause or feature	Meaning
`Correlated subquery`	Re-use the outer row site as a condition.
`Group mean`	Compare within each site.

SQL-06-002

Name an intermediate result

A common table expression gives a query stage an inspectable name. The final query chooses its own output order.

SQL to study

```
WITH totals AS (
 SELECT site,sum(value) AS total FROM sample GROUP BY site
)
SELECT site,total FROM totals ORDER BY total DESC,site;
```

Actual SQLite result

Result 1 · 4 rows

site	total
A	36.0
B	28.0
C	20.0
D	16.0

SQLite · exit 0

Clause or feature	Meaning
`WITH totals AS`	Name the grouped intermediate result.
`Final SELECT`	Select and order the intermediate rows.

SQL-06-003

Use EXISTS as a membership question

EXISTS asks whether any qualifying observation is present. It does not multiply lookup rows by the number of matches.

SQL to study

```
SELECT t.id,t.label FROM site AS t
WHERE EXISTS(SELECT 1 FROM sample AS s
             WHERE s.site=t.id AND s.value>15) ORDER BY t.id;
```

Actual SQLite result

Result 1 · 2 rows

id	label
A	River
B	Hill

SQLite · exit 0

Clause or feature	Meaning
`EXISTS`	Test whether the subquery has a row.
`One lookup row`	Return each qualifying site only once.

SQL-06-004

Expose the NOT IN and NULL trap

A NULL in the right-hand set makes unmatched NOT IN comparisons unknown. NOT EXISTS tests actual equality matches instead.

Compare with SQL-05-003 · Find missing relationships.

SQL to study

```
SELECT 'NOT IN' AS method,count(*) AS n FROM sample
WHERE site NOT IN(SELECT id FROM site UNION ALL SELECT NULL)
UNION ALL
SELECT 'NOT EXISTS',count(*) FROM sample AS s
WHERE NOT EXISTS(SELECT 1 FROM site AS t WHERE t.id=s.site)
ORDER BY method;
```

Actual SQLite result

Result 1 · 2 rows

method	n
NOT EXISTS	1
NOT IN	0

SQLite · exit 0

Clause or feature	Meaning
`NULL in a set`	An unmatched NOT IN test becomes unknown.
`NOT EXISTS`	Avoid that NULL-dependent membership result.

SQL-06-005

Compare UNION and UNION ALL

UNION removes duplicate result rows. UNION ALL retains them; both queries explicitly sort their output.

SQL to study

```
SELECT site FROM sample UNION SELECT id FROM site ORDER BY site;
SELECT site FROM sample UNION ALL SELECT id FROM site ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site
A
B
C
D

Result 2 · 11 rows

site
A
A
A
A
B
B
B
C
C
C
D

SQLite · exit 0

Clause or feature	Meaning
`UNION`	Apply set semantics to the selected rows.
`UNION ALL`	Preserve repeated rows.

SQL-06-006

Intersect and subtract sets

INTERSECT finds site keys shared by both tables. EXCEPT finds observed keys that are absent from the lookup.

SQL to study

```
SELECT site FROM sample INTERSECT SELECT id FROM site ORDER BY site;
SELECT site FROM sample EXCEPT SELECT id FROM site ORDER BY site;
```

Actual SQLite result

Result 1 · 3 rows

site
A
B
C

Result 2 · 1 rows

site
D

SQLite · exit 0

Clause or feature	Meaning
`INTERSECT`	Keep rows present on both sides.
`EXCEPT`	Keep left rows absent on the right.

07 · Window functions

SQL-07-001

Number rows within each site

ROW_NUMBER preserves observations while assigning positions within each site. A tie-breaker makes equal values deterministic.

SQL to study

```
SELECT id,site,row_number() OVER(
 PARTITION BY site ORDER BY value DESC,id) AS position
FROM sample ORDER BY site,position;
```

Actual SQLite result

Result 1 · 8 rows

id	site	position
3	A	1
1	A	2
6	A	3
2	B	1
4	B	2
5	C	1
7	C	2
8	D	1

SQLite · exit 0

Clause or feature	Meaning
`PARTITION BY`	Restart numbering for each site.
`Window ORDER BY`	Choose the order used for numbering.
`Outer ORDER BY`	Choose the returned row order.

SQL-07-002

Calculate a running total

An explicit ROWS frame makes the accumulation boundary clear. Window aggregation retains one result per observation.

SQL to study

```
SELECT id,sum(value) OVER(ORDER BY id
 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cumulative
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	cumulative
1	12.0
2	32.0
3	50.0
4	58.0
5	68.0
6	74.0
7	84.0
8	100.0

SQLite · exit 0

Clause or feature	Meaning
`ROWS frame`	Use physical rows in the window ordering.
`UNBOUNDED PRECEDING`	Start with the first row.

SQL-07-003

Measure change from the preceding row

LAG accesses a previous ordered row. The first difference is unknown, so its result is NULL.

SQL to study

```
SELECT id,value-lag(value) OVER(ORDER BY id) AS change
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	change
1	*NULL*
2	8.0
3	-2.0
4	-10.0
5	2.0
6	-4.0
7	4.0
8	6.0

SQLite · exit 0

Clause or feature	Meaning
`lag(value)`	Read the previous row value.
`First row`	No previous row exists.

SQL-07-004

Calculate a three-row moving mean

The first windows contain fewer than three rows. The unit is ordered observations rather than calendar days.

SQL to study

```
SELECT id,round(avg(value) OVER(ORDER BY id
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW),2) AS mean3
FROM sample ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	mean3
1	12.0
2	16.0
3	16.67
4	15.33
5	12.0
6	8.0
7	8.67
8	10.67

SQLite · exit 0

Clause or feature	Meaning
`2 PRECEDING`	Include up to two earlier observations.
`ROWS not days`	Missing calendar dates do not create rows.

SQL-07-005

Compare rank and dense rank

Tied values share a rank. RANK leaves a gap after a tie; DENSE_RANK advances without a gap.

SQL to study

```
SELECT id,value,rank() OVER(ORDER BY value DESC) AS r,
 dense_rank() OVER(ORDER BY value DESC) AS dense_r
FROM sample ORDER BY value DESC,id;
```

Actual SQLite result

Result 1 · 8 rows

id	value	r	dense_r
2	20.0	1	1
3	18.0	2	2
8	16.0	3	3
1	12.0	4	4
5	10.0	5	5
7	10.0	5	5
4	8.0	7	6
6	6.0	8	7

SQLite · exit 0

Clause or feature	Meaning
`Ties`	Do not add id to the rank window: doing so would break value ties.
`Outer id order`	Use id only to stabilize the displayed tied rows.

SQL-07-006

Give LAST_VALUE a complete frame

LAST_VALUE uses the end of its window frame. An explicit full-partition frame retrieves the final observation for every row.

SQL to study

```
SELECT id,first_value(value) OVER w AS first,
 last_value(value) OVER w AS last FROM sample
WINDOW w AS (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING) ORDER BY id;
```

Actual SQLite result

Result 1 · 8 rows

id	first	last
1	12.0	16.0
2	12.0	16.0
3	12.0	16.0
4	12.0	16.0
5	12.0	16.0
6	12.0	16.0
7	12.0	16.0
8	12.0	16.0

SQLite · exit 0

Clause or feature	Meaning
`Named window`	Reuse the same definition for both functions.
`UNBOUNDED FOLLOWING`	Extend the frame to the final row.

08 · Recursive queries

SQL-08-001

Generate a finite sequence

A recursive CTE combines an anchor row with a terminating recursive step.

SQL to study

```
WITH RECURSIVE seq(n) AS (
 VALUES(1) UNION ALL SELECT n+1 FROM seq WHERE n<5
) SELECT n,n*n AS square FROM seq ORDER BY n;
```

Actual SQLite result

Result 1 · 5 rows

n	square
1	1
2	4
3	9
4	16
5	25

SQLite · exit 0

Clause or feature	Meaning
`Anchor`	Start at one.
`WHERE n<5`	Bound recursion explicitly.
`UNION ALL`	Keep every generated step.

SQL-08-002

Generate a calendar with missing days

A recursive date series supplies the missing seventh day. A LEFT JOIN then counts observations on every calendar day.

SQL to study

```
WITH RECURSIVE days(day) AS (
 VALUES('2026-01-01') UNION ALL
 SELECT date(day,'+1 day') FROM days WHERE day<'2026-01-09'
) SELECT d.day,count(s.id) AS n FROM days AS d
LEFT JOIN sample AS s ON s.day=d.day
GROUP BY d.day ORDER BY d.day;
```

Actual SQLite result

Result 1 · 9 rows

day	n
2026-01-01	1
2026-01-02	1
2026-01-03	1
2026-01-04	1
2026-01-05	1
2026-01-06	1
2026-01-07	0
2026-01-08	1
2026-01-09	1

SQLite · exit 0

Clause or feature	Meaning
`Date series`	Create rows for dates without observations.
`count(s.id)`	Count matches, not the preserved calendar row.

SQL-08-003

Walk a parent-child hierarchy

Recursion follows child links from one root and carries the depth. The final path order gives a stable readable hierarchy.

SQL to study

```
CREATE TABLE node(id INTEGER PRIMARY KEY,parent INTEGER,name TEXT);
INSERT INTO node VALUES(1,NULL,'Archive'),(2,1,'Field'),
 (3,1,'Lab'),(4,2,'North'),(5,2,'South');
WITH RECURSIVE tree(id,path,depth) AS (
 SELECT id,name,0 FROM node WHERE parent IS NULL
 UNION ALL SELECT n.id,t.path||'/'||n.name,t.depth+1
 FROM node AS n JOIN tree AS t ON n.parent=t.id
) SELECT path,depth FROM tree ORDER BY path;
```

Actual SQLite result

Result 1 · 5 rows

path	depth
Archive	0
Archive/Field	1
Archive/Field/North	2
Archive/Field/South	2
Archive/Lab	1

SQLite · exit 0

Clause or feature	Meaning
`Carried columns`	Maintain a path and depth through recursive steps.
`Child join`	Extend only from the current parent.

SQL-08-004

Prevent cycles in a graph walk

The visited path prevents a node from being repeated within one walk. Delimiters keep node one distinct from node eleven.

SQL to study

```
CREATE TABLE edge(a INTEGER,b INTEGER);
INSERT INTO edge VALUES(1,2),(2,3),(3,1),(2,4);
WITH RECURSIVE walk(id,path,depth) AS (
 VALUES(1,',1,',0) UNION ALL
 SELECT e.b,w.path||e.b||',',w.depth+1
 FROM edge AS e JOIN walk AS w ON e.a=w.id
 WHERE instr(w.path,','||e.b||',')=0 AND w.depth<8
) SELECT id,depth FROM walk ORDER BY depth,id;
```

Actual SQLite result

Result 1 · 4 rows

id	depth
1	0
2	1
3	2
4	2

SQLite · exit 0

Clause or feature	Meaning
`Visited path`	Reject a node already present in this path.
`Depth bound`	Add a separate finite upper bound.

SQL-08-005

Carry an accumulated calculation

A recursive CTE can carry both a counter and a running product. This example calculates factorials through six.

SQL to study

```
WITH RECURSIVE fact(n,value) AS (
 VALUES(0,1) UNION ALL
 SELECT n+1,value*(n+1) FROM fact WHERE n<6
) SELECT n,value FROM fact ORDER BY n;
```

Actual SQLite result

Result 1 · 7 rows

n	value
0	1
1	1
2	2
3	6
4	24
5	120
6	720

SQLite · exit 0

Clause or feature	Meaning
`State pair`	Carry the current n and factorial value.
`Termination`	Stop before the integer product grows large.

SQL-08-006

Expand a bill of materials

Quantities multiply along each relationship. Leaf totals combine all paths that reach the same final part.

SQL to study

```
CREATE TABLE component(parent TEXT,child TEXT,qty INTEGER);
INSERT INTO component VALUES('kit','case',1),('kit','pack',2),('pack','bolt',4);
WITH RECURSIVE bom(part,qty) AS (
 VALUES('kit',1) UNION ALL
 SELECT c.child,b.qty*c.qty FROM component AS c
 JOIN bom AS b ON c.parent=b.part
) SELECT part,sum(qty) AS needed FROM bom AS b
WHERE NOT EXISTS(SELECT 1 FROM component AS c WHERE c.parent=b.part)
GROUP BY part ORDER BY part;
```

Actual SQLite result

Result 1 · 2 rows

part	needed
bolt	8
case	1

SQLite · exit 0

Clause or feature	Meaning
`Multiply quantities`	Two packs of four bolts require eight bolts.
`Leaf filter`	Keep only parts with no children.

09 · Schema, types and constraints

SQL-09-001

Reject a violated CHECK constraint

A CHECK constraint validates the stored row. This intentionally invalid insert produces a captured native diagnostic and an expected failure exit.

SQL to study

```
CREATE TABLE quantity(id INTEGER PRIMARY KEY,n INTEGER NOT NULL CHECK(n>=0));
INSERT INTO quantity VALUES(1,-1);
```

Actual SQLite result

Expected diagnostic

```
Error near line 21 of SQL-09-001.sql: CHECK constraint failed: n>=0
```

SQLite · expected exit 1

Clause or feature	Meaning
`NOT NULL`	Require a known quantity.
`CHECK(n>=0)`	Reject negative values.
`Expected failure`	The invalid row is not stored.

SQL-09-002

Enforce a foreign key

Foreign-key enforcement is enabled in every standalone fixture. This deliberately missing parent key is rejected.

SQL to study

```
CREATE TABLE parent(id INTEGER PRIMARY KEY);
CREATE TABLE child(id INTEGER PRIMARY KEY,parent_id INTEGER REFERENCES parent(id));
INSERT INTO child VALUES(1,99);
```

Actual SQLite result

Expected diagnostic

```
Error near line 22 of SQL-09-002.sql: FOREIGN KEY constraint failed
```

SQLite · expected exit 1

Clause or feature	Meaning
`REFERENCES`	Declare the relationship in the schema.
`PRAGMA foreign_keys=ON`	Enable enforcement for this connection.

SQL-09-003

Compare STRICT storage with ANY

STRICT tables check declared types. A numeric string can be losslessly converted to INTEGER, while ANY preserves the original text.

SQL to study

```
CREATE TABLE typed(n INTEGER,tag ANY) STRICT;
INSERT INTO typed VALUES('000123','000123');
SELECT n,typeof(n) AS n_type,tag,typeof(tag) AS tag_type FROM typed;
```

Actual SQLite result

Result 1 · 1 rows

n	n_type	tag	tag_type
123	integer	000123	text

SQLite · exit 0

Clause or feature	Meaning
`STRICT`	Use the table-level strict type rules.
`ANY`	Keep the supplied storage type without numeric coercion.

SQL-09-004

Store a generated column

The schema defines the total calculation once. Integer cents keep this example exact.

SQL to study

```
CREATE TABLE line(qty INTEGER,unit_cents INTEGER,
 total_cents INTEGER GENERATED ALWAYS AS (qty*unit_cents) STORED);
INSERT INTO line(qty,unit_cents) VALUES(3,120),(2,125);
SELECT qty,unit_cents,total_cents FROM line ORDER BY total_cents;
```

Actual SQLite result

Result 1 · 2 rows

qty	unit_cents	total_cents
2	125	250
3	120	360

SQLite · exit 0

Clause or feature	Meaning
`GENERATED ALWAYS`	Calculate the value from other columns.
`STORED`	Persist the generated result.
`Integer cents`	Avoid a binary floating-point money example.

SQL-09-005

Use a composite key without rowid

WITHOUT ROWID uses the declared primary key as the table key. The two-part key identifies each coordinate entry.

SQL to study

```
CREATE TABLE coordinate(a TEXT,b INTEGER,value TEXT,
 PRIMARY KEY(a,b)) WITHOUT ROWID;
INSERT INTO coordinate VALUES('A',1,'north'),('A',2,'south'),('B',1,'east');
SELECT a,b,value FROM coordinate ORDER BY a,b;
```

Actual SQLite result

Result 1 · 3 rows

a	b	value
A	1	north
A	2	south
B	1	east

SQLite · exit 0

Clause or feature	Meaning
`Composite primary key`	Identify a row with both a and b.
`WITHOUT ROWID`	Use that key as the underlying table key.

SQL-09-006

See how UNIQUE treats NULL

A UNIQUE constraint permits multiple NULL values. COUNT(column) and COUNT(DISTINCT column) ignore those unknown values.

SQL to study

```
CREATE TABLE code(id INTEGER PRIMARY KEY,label TEXT UNIQUE);
INSERT INTO code VALUES(1,NULL),(2,NULL),(3,'X');
SELECT count(*) AS n,count(label) AS known,
 count(DISTINCT label) AS distinct_known FROM code;
```

Actual SQLite result

Result 1 · 1 rows

n	known	distinct_known
3	1	1

SQLite · exit 0

Clause or feature	Meaning
`UNIQUE and NULL`	Two unknown labels do not conflict.
`Counting NULL`	Only the actual X label contributes to known counts.

10 · Writing data, views and triggers

SQL-10-001

Insert a row with RETURNING

RETURNING exposes the row written by the statement. A single-row insert makes its result order unambiguous.

SQL to study

```
INSERT INTO sample VALUES(9,'A',14,'2026-01-10','new')
RETURNING id,site,value;
```

Actual SQLite result

Result 1 · 1 rows

id	site	value
9	A	14.0

SQLite · exit 0

Clause or feature	Meaning
`INSERT VALUES`	Supply the new observation.
`RETURNING`	Return selected fields of the inserted row.

SQL-10-002

Update one row with RETURNING

A primary-key predicate makes the scope of the update explicit. RETURNING shows the updated value.

SQL to study

```
UPDATE sample SET value=value+2 WHERE id=2
RETURNING id,value;
```

Actual SQLite result

Result 1 · 1 rows

id	value
2	22.0

SQLite · exit 0

Clause or feature	Meaning
`WHERE id=2`	Limit the change to one known row.
`value+2`	Calculate from its previous value.

SQL-10-003

Delete one row with RETURNING

The returned fields describe the row that was removed. Other observations remain in the downloadable database.

SQL to study

```
DELETE FROM sample WHERE id=8 RETURNING id,site;
```

Actual SQLite result

Result 1 · 1 rows

id	site
8	D

SQLite · exit 0

Clause or feature	Meaning
`DELETE WHERE`	Choose a finite set to remove.
`RETURNING`	Inspect the removed row fields.

SQL-10-004

Handle a conflict with UPSERT

A primary-key conflict updates the existing lookup, while a new key is inserted. A final SELECT defines the display order.

SQL to study

```
INSERT INTO site(id,label) VALUES('A','River East'),('D','Desert')
ON CONFLICT(id) DO UPDATE SET label=excluded.label;
SELECT id,label FROM site ORDER BY id;
```

Actual SQLite result

Result 1 · 4 rows

id	label
A	River East
B	Hill
C	Coast
D	Desert

SQLite · exit 0

Clause or feature	Meaning
`ON CONFLICT(id)`	Handle the declared unique key.
`excluded.label`	Read the value proposed by the conflicting insert.

SQL-10-005

Reuse a view definition

A view stores a query definition rather than a separate result table. Reading it reflects the current underlying data.

SQL to study

```
CREATE VIEW site_total AS
 SELECT site,sum(value) AS total FROM sample GROUP BY site;
SELECT site,total FROM site_total ORDER BY site;
```

Actual SQLite result

Result 1 · 4 rows

site	total
A	36.0
B	28.0
C	20.0
D	16.0

SQLite · exit 0

Clause or feature	Meaning
`CREATE VIEW`	Name a reusable query.
`Not a stored snapshot`	The view reads the current sample rows.

SQL-10-006

Record changes with a trigger

An AFTER UPDATE trigger records old and new values. The audit table is ordinary stored data.

SQL to study

```
CREATE TABLE audit(id INTEGER PRIMARY KEY,sample_id INTEGER,
 old_value REAL,new_value REAL);
CREATE TRIGGER sample_audit AFTER UPDATE OF value ON sample
BEGIN
 INSERT INTO audit(sample_id,old_value,new_value)
 VALUES(old.id,old.value,new.value);
END;
UPDATE sample SET value=13 WHERE id=1;
SELECT sample_id,old_value,new_value FROM audit ORDER BY id;
```

Actual SQLite result

Result 1 · 1 rows

sample_id	old_value	new_value
1	12.0	13.0

SQLite · exit 0

Clause or feature	Meaning
`AFTER UPDATE OF value`	Run after a value update.
`old / new`	Read the row before and after the update.

11 · Transactions and savepoints

SQL-11-001

Inspect and roll back a change

The first result sees the uncommitted write on this connection. ROLLBACK restores the original stored value.

SQL to study

```
BEGIN;
UPDATE sample SET value=99 WHERE id=1;
SELECT 'staged' AS phase,value FROM sample WHERE id=1;
ROLLBACK;
SELECT 'after' AS phase,value FROM sample WHERE id=1;
```

Actual SQLite result

Result 1 · 1 rows

phase	value
staged	99.0

Result 2 · 1 rows

phase	value
after	12.0

SQLite · exit 0

Clause or feature	Meaning
`BEGIN`	Start an explicit transaction.
`Read your write`	This connection sees its pending update.
`ROLLBACK`	Undo the transaction.

SQL-11-002

Commit a group of writes

COMMIT makes the inserted observation part of the database state. The summary reads the committed result.

Compare with SQL-11-001 · Inspect and roll back a change.

SQL to study

```
BEGIN;
INSERT INTO sample VALUES(9,'A',14,'2026-01-10','new');
COMMIT;
SELECT count(*) AS n,sum(value) AS total FROM sample;
```

Actual SQLite result

Result 1 · 1 rows

n	total
9	114.0

SQLite · exit 0

Clause or feature	Meaning
`COMMIT`	Finish the transaction successfully.
`Final summary`	Check the persisted row count and total.

SQL-11-003

Roll back to a savepoint

A savepoint gives a partial rollback target inside a transaction. The earlier update survives; the later one is undone.

SQL to study

```
BEGIN;
UPDATE sample SET value=14 WHERE id=1;
SAVEPOINT later;
UPDATE sample SET value=0 WHERE id=2;
ROLLBACK TO later;
RELEASE later; COMMIT;
SELECT id,value FROM sample WHERE id IN(1,2) ORDER BY id;
```

Actual SQLite result

Result 1 · 2 rows

id	value
1	14.0
2	20.0

SQLite · exit 0

Clause or feature	Meaning
`SAVEPOINT`	Mark an intermediate transaction state.
`ROLLBACK TO`	Undo later work without ending the outer transaction.
`RELEASE`	Remove the savepoint marker.

SQL-11-004

Defer a foreign-key check until commit

A deferred relationship permits the child to be inserted before its parent within one transaction. Both exist when the transaction commits.

SQL to study

```
CREATE TABLE parent(id INTEGER PRIMARY KEY);
CREATE TABLE child(name TEXT,parent_id INTEGER REFERENCES parent(id)
 DEFERRABLE INITIALLY DEFERRED);
BEGIN;
INSERT INTO child VALUES('pending',42);
INSERT INTO parent VALUES(42);
COMMIT;
SELECT c.name,p.id AS parent_id FROM child AS c
JOIN parent AS p ON p.id=c.parent_id;
```

Actual SQLite result

Result 1 · 1 rows

name	parent_id
pending	42

SQLite · exit 0

Clause or feature	Meaning
`INITIALLY DEFERRED`	Check the relationship at transaction commit.
`Commit condition`	The missing parent must exist by the end.

SQL-11-005

Begin a write transaction immediately

BEGIN IMMEDIATE requests the write transaction at the start. This one-connection example shows the statement sequence, not a concurrent-lock benchmark.

SQL to study

```
BEGIN IMMEDIATE;
UPDATE sample SET value=21 WHERE id=1;
COMMIT;
SELECT id,value FROM sample WHERE id=1;
```

Actual SQLite result

Result 1 · 1 rows

id	value
1	21.0

SQLite · exit 0

Clause or feature	Meaning
`BEGIN IMMEDIATE`	Request the write transaction up front.
`Single connection`	No contention or timeout measurement is claimed here.

SQL-11-006

Select write-ahead logging

WAL is a journal mode of a file-backed database. The native run uses a fresh file and closes the connection before saving its database download.

SQL to study

```
PRAGMA journal_mode=WAL;
BEGIN; UPDATE sample SET value=value+1 WHERE id=1; COMMIT;
SELECT id,value FROM sample WHERE id=1;
```

Actual SQLite result

Result 1 · 1 rows

journal_mode
wal

Result 2 · 1 rows

id	value
1	13.0

SQLite · exit 0

Clause or feature	Meaning
`journal_mode=WAL`	Use write-ahead logging for this file.
`One writer`	WAL does not provide multiple simultaneous writers.
`Portable download`	The exported data snapshot uses rollback-journal mode so it opens as one file.

12 · Indexes and query plans

SQL-12-001

Read a table-scan plan

EXPLAIN QUERY PLAN reports the chosen access strategy. Its numeric fields and wording may change between SQLite versions.

SQL to study

```
EXPLAIN QUERY PLAN
SELECT id,site,value FROM sample WHERE site='A';
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
2	0	216	SCAN sample

SQLite · exit 0

Clause or feature	Meaning
`SCAN`	Visit the table without a selective site index.
`Plan version`	Read this as a diagnostic, not a stable application interface.

SQL-12-002

Add a selective index

Creating an index changes the available access paths. The captured plan identifies the named index used for the equality predicate.

Compare with SQL-12-001 · Read a table-scan plan.

SQL to study

```
CREATE INDEX ix_site ON sample(site);
EXPLAIN QUERY PLAN
SELECT id,site,value FROM sample WHERE site='A';
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
3	0	62	SEARCH sample USING INDEX ix_site (site=?)

SQLite · exit 0

Clause or feature	Meaning
`CREATE INDEX`	Build a search structure on site.
`SEARCH`	Use a restricted index lookup.
`ix_site`	The chosen index is named in the native plan.

SQL-12-003

Use a covering index

The selected columns and filter can be satisfied from this index. The plan explicitly identifies a covering access path.

Compare with SQL-12-002 · Add a selective index.

SQL to study

```
CREATE INDEX ix_site_value ON sample(site,value);
EXPLAIN QUERY PLAN
SELECT site,value FROM sample WHERE site='A' ORDER BY value;
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
3	0	55	SEARCH sample USING COVERING INDEX ix_site_value (site=?)

SQLite · exit 0

Clause or feature	Meaning
`Composite key`	Store site followed by value.
`COVERING`	The query does not need additional table fields.

SQL-12-004

Use a composite index prefix and range

Equality on the leading column and a range on the next column fit the index ordering.

SQL to study

```
CREATE INDEX ix_site_day ON sample(site,day);
EXPLAIN QUERY PLAN
SELECT id,day FROM sample
WHERE site='A' AND day>='2026-01-03' ORDER BY day;
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
3	0	47	SEARCH sample USING COVERING INDEX ix_site_day (site=? AND day>?)

SQLite · exit 0

Clause or feature	Meaning
Leading equality	Fix site before traversing the day range.
Range and order	The second index column supplies the ordered range.

SQL-12-005

Index a subset of rows

A partial index stores only observations whose note is NULL. The query repeats the condition needed to use that subset.

SQL to study

```
CREATE INDEX ix_missing ON sample(day) WHERE note IS NULL;
EXPLAIN QUERY PLAN
SELECT id,day FROM sample WHERE note IS NULL
 AND day>='2026-01-03' ORDER BY day;
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
4	0	194	SEARCH sample USING INDEX ix_missing (day>?)

SQLite · exit 0

Clause or feature	Meaning
`Partial index`	Index only rows satisfying its WHERE clause.
`Matching condition`	The query implies that same condition.

SQL-12-006

Index an expression

An expression index supports a query that uses the same expression. This example indexes normalized lookup labels.

SQL to study

```
CREATE INDEX ix_lower_label ON site(lower(label));
EXPLAIN QUERY PLAN
SELECT id,label FROM site WHERE lower(label)='river';
```

Actual SQLite result

Result 1 · 1 rows

id	parent	notused	detail
3	0	61	SEARCH site USING INDEX ix_lower_label (<expr>=?)

SQLite · exit 0

Clause or feature	Meaning
`Expression key`	Store the lowercased label as the index key.
`Matching expression`	Use lower(label) in the query predicate.

13 · JSON documents

SQL-13-001

Extract typed JSON values

JSON extraction and JSON type inspection answer different questions. JSON true and JSON null retain their JSON type names.

SQL to study

```
WITH doc(body) AS (VALUES('{"site":"A","count":3,"checked":true,"missing":null}'))
SELECT json_extract(body,'$.site') AS site,
 json_extract(body,'$.count') AS n,
 json_type(body,'$.checked') AS checked_type,
 json_type(body,'$.missing') AS missing_type FROM doc;
```

Actual SQLite result

Result 1 · 1 rows

site	n	checked_type	missing_type
A	3	true	null

SQLite · exit 0

Clause or feature	Meaning
`json_extract`	Convert selected scalar values to SQL values.
`json_type`	Describe a value using JSON type names.

SQL-13-002

Compare JSON text and SQL scalars

The arrow operators choose different result representations. JSON null becomes SQL NULL under the scalar extraction operator.

SQL to study

```
WITH doc(body) AS (VALUES('{"n":4,"empty":null}'))
SELECT body->'$.n' AS json_value,body->>'$.n' AS sql_value,
 body->>'$.empty' AS empty,typeof(body->'$.n') AS json_type,
 typeof(body->>'$.n') AS sql_type
FROM doc;
```

Actual SQLite result

Result 1 · 1 rows

json_value	sql_value	empty	json_type	sql_type
4	4	*NULL*	text	integer

SQLite · exit 0

Clause or feature	Meaning
->	Return a JSON text representation.
->>	Return a corresponding SQL scalar value.

SQL-13-003

Expand a JSON array into rows

json_each is a table-valued function. Array keys become positions, while value and type describe each element.

SQL to study

```
SELECT CAST(key AS INTEGER) AS position,value,type
FROM json_each('["A","B",null]') ORDER BY position;
```

Actual SQLite result

Result 1 · 3 rows

position	value	type
0	A	text
1	B	text
2	*NULL*	null

SQLite · exit 0

Clause or feature	Meaning
`json_each`	Return one row per immediate child.
`Array keys`	JSON arrays use zero-based indexes.

SQL-13-004

Modify and merge JSON objects

json_set edits selected paths. Merge Patch treats a null-valued object member in the patch as a removal request.

SQL to study

```
SELECT json_set('{"site":"A","count":3}',
 '$.count',4,'$.checked',json('true')) AS updated,
 json_patch('{"site":"A","keep":1}',
            '{"site":"B","keep":null}') AS patched;
```

Actual SQLite result

Result 1 · 1 rows

updated	patched
{"site":"A","count":4,"checked":true}	{"site":"B"}

SQLite · exit 0

Clause or feature	Meaning
`json(true)`	Supply a JSON Boolean rather than quoted text.
`json_patch`	Apply JSON Merge Patch object semantics.

SQL-13-005

Aggregate ordered values into JSON

An aggregate can produce a JSON array directly from query rows. Ordering inside the aggregate fixes the element order.

SQL to study

```
SELECT json_group_array(site ORDER BY site) AS sites
FROM (SELECT DISTINCT site FROM sample);
```

Actual SQLite result

Result 1 · 1 rows

sites
["A","B","C","D"]

SQLite · exit 0

Clause or feature	Meaning
`json_group_array`	Build a JSON array from selected values.
`Ordered aggregate`	Specify the array element order.

SQL-13-006

Store SQLite JSONB

JSONB is SQLite internal binary JSON representation, available since 3.45. It is a BLOB, not the JSONB format of another database system.

SQL to study

```
SELECT typeof(jsonb('{"site":"A","n":3}')) AS storage,
 json(jsonb('{"site":"A","n":3}')) AS text_json,
 json_extract(jsonb('{"site":"A","n":3}'),'$.n') AS n;
```

Actual SQLite result

Result 1 · 1 rows

storage	text_json	n
blob	{"site":"A","n":3}	3

SQLite · exit 0

Clause or feature	Meaning
`jsonb`	Produce SQLite binary JSON.
`json`	Convert it back to portable JSON text.
`Compatibility`	Requires SQLite 3.45 or newer.

14 · Full-text search

SQL-14-001

Search terms with FTS5

An FTS5 virtual table indexes text for MATCH queries. The first example includes its complete corpus creation in the reading view.

SQL to study

```
CREATE VIRTUAL TABLE docs USING fts5(title,body);
INSERT INTO docs(rowid,title,body) VALUES
 (1,'River survey','pottery fragments near the river bank'),
 (2,'Hill archive','bronze tools from the hill settlement'),
 (3,'Coastal notes','pottery and shells from the coastal survey'),
 (4,'River laboratory','pottery analysis in the laboratory');
SELECT rowid,title FROM docs WHERE docs MATCH 'pottery' ORDER BY rowid;
```

Actual SQLite result

Result 1 · 3 rows

rowid	title
1	River survey
3	Coastal notes
4	River laboratory

SQLite · exit 0

Clause or feature	Meaning
`USING fts5`	Create the full-text virtual table.
`MATCH`	Apply full-text query syntax.
`Fixture`	The remaining chapter examples use this same four-document corpus.

SQL-14-002

Search an exact phrase

A quoted FTS phrase requires the tokens in the stated order. It is different from asking for either word anywhere.

Compare with SQL-14-001 · Search terms with FTS5.

SQL to study

```
SELECT rowid,title FROM docs
WHERE docs MATCH '"river bank"' ORDER BY rowid;
```

Actual SQLite result

Result 1 · 1 rows

rowid	title
1	River survey

SQLite · exit 0

Clause or feature	Meaning
`Quoted phrase`	Require adjacent matching tokens in order.
`SQL quoting`	Single quotes contain the FTS query string.

SQL-14-003

Combine a prefix and an exclusion

A star applies prefix matching to the preceding token. NOT removes documents matching the second term.

SQL to study

```
SELECT rowid,title FROM docs
WHERE docs MATCH 'pot* NOT laboratory' ORDER BY rowid;
```

Actual SQLite result

Result 1 · 2 rows

rowid	title
1	River survey
3	Coastal notes

SQLite · exit 0

Clause or feature	Meaning
`pot*`	Match indexed tokens beginning with pot.
`NOT laboratory`	Exclude that term from either indexed column.

SQL-14-004

Order matches by BM25

FTS5 BM25 ranks the matching documents. Better matches have smaller values; rowid breaks ties without displaying unstable score decimals.

Compare with SQL-14-001 · Search terms with FTS5.

SQL to study

```
SELECT rowid,title FROM docs WHERE docs MATCH 'pottery'
ORDER BY bm25(docs),rowid;
```

Actual SQLite result

Result 1 · 3 rows

rowid	title
4	River laboratory
1	River survey
3	Coastal notes

SQLite · exit 0

Clause or feature	Meaning
`bm25(docs)`	Use the native full-text ranking function.
`Ascending score`	Smaller is better in SQLite FTS5.
`rowid`	Stabilize equal scores.

SQL-14-005

Highlight matching tokens

The native highlight function marks matched text. The guide displays those markers as text, without interpreting them as page markup.

SQL to study

```
SELECT rowid,highlight(docs,1,'[',']') AS excerpt
FROM docs WHERE docs MATCH 'pottery' ORDER BY rowid;
```

Actual SQLite result

Result 1 · 3 rows

rowid	excerpt
1	[pottery] fragments near the river bank
3	[pottery] and shells from the coastal survey
4	[pottery] analysis in the laboratory

SQLite · exit 0

Clause or feature	Meaning
`Column 1`	Choose body, the second indexed column.
`Markers`	Surround each matching token with square brackets.

SQL-14-006

Restrict a full-text query by column

Column filters give title and body different roles. Both named conditions must match the same document.

SQL to study

```
SELECT rowid,title FROM docs
WHERE docs MATCH 'title:river AND body:analysis' ORDER BY rowid;
```

Actual SQLite result

Result 1 · 1 rows

rowid	title
4	River laboratory

SQLite · exit 0

Clause or feature	Meaning
`title:river`	Require a title token.
`body:analysis`	Require a body token in the same document.

15 · Inspection and database copies

SQL-15-001

Inspect the schema catalog

sqlite_schema exposes stored schema objects. Filter object types deliberately and define the result order.

SQL to study

```
SELECT name,type FROM sqlite_schema
WHERE type='table' ORDER BY name;
```

Actual SQLite result

Result 1 · 2 rows

name	type
sample	table
site	table

SQLite · exit 0

Clause or feature	Meaning
`sqlite_schema`	Read schema metadata as rows.
`type filter`	Exclude automatically created index objects.

SQL-15-002

Query column metadata as a table

Table-valued PRAGMAs let metadata participate in ordinary SELECT statements. A rowid-alias primary key is reported differently from an explicit NOT NULL column.

SQL to study

```
SELECT name,type,"notnull" AS required
FROM pragma_table_info('sample') ORDER BY cid;
```

Actual SQLite result

Result 1 · 5 rows

name	type	required
id	INTEGER	0
site	TEXT	1
value	REAL	1
day	TEXT	1
note	TEXT	0

SQLite · exit 0

Clause or feature	Meaning
`pragma_table_info`	Expose column metadata as a table-valued function.
`required flag`	Report the declared NOT NULL property.

SQL-15-003

Check declared foreign-key relationships

The table-valued foreign-key check returns violations. The starting fixture deliberately leaves sample.site unconstrained, so missing lookup D is not a declared foreign-key violation.

Compare with SQL-05-003 · Find missing relationships.

SQL to study

```
SELECT count(*) AS violations FROM pragma_foreign_key_check;
```

Actual SQLite result

Result 1 · 1 rows

violations
0

SQLite · exit 0

Clause or feature	Meaning
`Declared constraints`	Only schema REFERENCES clauses are checked.
`Missing metadata`	An unconstrained lookup gap is a separate data-quality question.

SQL-15-004

Run a database consistency check

quick_check inspects database consistency. It does not validate every application-level interpretation of the data.

SQL to study

```
PRAGMA quick_check;
```

Actual SQLite result

Result 1 · 1 rows

quick_check
ok

SQLite · exit 0

Clause or feature	Meaning
quick_check	Run the native consistency diagnostic.
Scope	Database consistency and domain correctness are different checks.

SQL-15-005

Inspect affected-row counts

changes() reports the directly affected rows of the most recent data-changing statement on this connection.

SQL to study

```
UPDATE sample SET note='none' WHERE id=-1;
SELECT changes() AS affected;
INSERT INTO sample VALUES(9,'A',14,'2026-01-10','new');
SELECT changes() AS affected;
```

Actual SQLite result

Result 1 · 1 rows

affected
0

Result 2 · 1 rows

affected
1

SQLite · exit 0

Clause or feature	Meaning
`changes()`	Inspect the previous statement direct row count.
`Zero-match update`	A valid UPDATE can affect no rows.

SQL-15-006

Create and verify a database copy

VACUUM INTO writes a separate database file. Attaching that file makes the copied row count directly inspectable in this fresh working directory.

SQL to study

```
VACUUM INTO 'snapshot.sqlite3';
ATTACH DATABASE 'snapshot.sqlite3' AS snapshot;
SELECT (SELECT count(*) FROM main.sample) AS original_rows,
       (SELECT count(*) FROM snapshot.sample) AS copied_rows;
DETACH DATABASE snapshot;
```

Actual SQLite result

Result 1 · 1 rows

original_rows	copied_rows
8	8

SQLite · exit 0

Clause or feature	Meaning
`VACUUM INTO`	Create a compact copy at a new filename.
`ATTACH`	Read a second database through a schema name.
`Fresh directory`	The target file must not already contain a database.

16 · Practical query patterns

SQL-16-001

Pivot a small categorical summary

Conditional aggregation makes a fixed set of categories into columns. The category list is explicit.

SQL to study

```
SELECT sum(CASE WHEN site='A' THEN value ELSE 0 END) AS a,
 sum(CASE WHEN site='B' THEN value ELSE 0 END) AS b,
 sum(CASE WHEN site='C' THEN value ELSE 0 END) AS c,
 sum(CASE WHEN site='D' THEN value ELSE 0 END) AS d FROM sample;
```

Actual SQLite result

Result 1 · 1 rows

a	b	c	d
36.0	28.0	20.0	16.0

SQLite · exit 0

Clause or feature	Meaning
`Conditional aggregate`	Contribute only the selected category value.
`Fixed columns`	Add another expression for a new category.

SQL-16-002

Find islands of consecutive dates

Subtracting an ordered row number from each distinct day gives a constant key within a run of consecutive dates.

SQL to study

```
WITH distinct_days AS (SELECT DISTINCT day FROM sample),
 keyed AS (SELECT day,julianday(day)-row_number() OVER(ORDER BY day) AS k
           FROM distinct_days)
SELECT min(day) AS start,max(day) AS end,count(*) AS days
FROM keyed GROUP BY k ORDER BY start;
```

Actual SQLite result

Result 1 · 2 rows

start	end	days
2026-01-01	2026-01-06	6
2026-01-08	2026-01-09	2

SQLite · exit 0

Clause or feature	Meaning
`Distinct dates`	Remove duplicate dates before numbering.
`Island key`	A one-day increase and one-row increase cancel.

SQL-16-003

Return the top two within each group

A CTE allows the window result to be filtered in an outer query. Ties are resolved by the primary key.

Compare with SQL-07-001 · Number rows within each site.

SQL to study

```
WITH ranked AS (SELECT id,site,value,row_number() OVER(
 PARTITION BY site ORDER BY value DESC,id) AS position FROM sample)
SELECT id,site,value FROM ranked WHERE position<=2
ORDER BY site,position;
```

Actual SQLite result

Result 1 · 7 rows

id	site	value
3	A	18.0
1	A	12.0
2	B	20.0
4	B	8.0
5	C	10.0
7	C	10.0
8	D	16.0

SQLite · exit 0

Clause or feature	Meaning
`Outer filter`	Filter a window result in a subsequent query stage.
`At most two`	Sites with fewer observations return fewer rows.

SQL-16-004

Deduplicate an explicit ingestion key

This separate incoming table defines batch plus code as its business key. Keep the earliest row for that key; do not infer that equal observation values are duplicates.

SQL to study

```
CREATE TABLE incoming(id INTEGER PRIMARY KEY,batch TEXT,code TEXT,value INTEGER);
INSERT INTO incoming VALUES(1,'b1','A',12),(2,'b1','A',12),
 (3,'b1','B',20),(4,'b2','A',18);
DELETE FROM incoming WHERE id IN(
 SELECT id FROM (SELECT id,row_number() OVER(
  PARTITION BY batch,code ORDER BY id) AS position FROM incoming)
 WHERE position>1);
SELECT id,batch,code,value FROM incoming ORDER BY id;
```

Actual SQLite result

Result 1 · 3 rows

id	batch	code	value
1	b1	A	12
3	b1	B	20
4	b2	A	18

SQLite · exit 0

Clause or feature	Meaning
`Business key`	Deduplication depends on the declared meaning of batch and code.
`Earliest id`	Choose the retained row deterministically.

SQL-16-005

Calculate a calendar-based rolling total

Generate missing dates before using a ROWS window. Zero is an explicit modelling choice for a day with no observation.

Compare with SQL-07-004 · Calculate a three-row moving mean.

SQL to study

```
WITH RECURSIVE days(day) AS (
 VALUES('2026-01-01') UNION ALL
 SELECT date(day,'+1 day') FROM days WHERE day<'2026-01-09'
), filled AS (SELECT d.day,coalesce(s.value,0) AS value
 FROM days AS d LEFT JOIN sample AS s ON s.day=d.day)
SELECT day,value,sum(value) OVER(ORDER BY day
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS total3
FROM filled ORDER BY day;
```

Actual SQLite result

Result 1 · 9 rows

day	value	total3
2026-01-01	12.0	12.0
2026-01-02	20.0	32.0
2026-01-03	18.0	50.0
2026-01-04	8.0	46.0
2026-01-05	10.0	36.0
2026-01-06	6.0	24.0
2026-01-07	0	16.0
2026-01-08	10.0	16.0
2026-01-09	16.0	26.0

SQLite · exit 0

Clause or feature	Meaning
`Calendar first`	Insert the missing seventh day before windowing.
`Zero fill`	Treat no observation as zero only for this stated model.
`One row per day`	This fixture has at most one observation on each day.

SQL-16-006

Compose a joined summary report

A grouped CTE, lookup join and scalar total combine into one report. The missing lookup remains visible rather than disappearing.

SQL to study

```
WITH totals AS (SELECT site,count(*) AS n,sum(value) AS total
 FROM sample GROUP BY site)
SELECT t.site,coalesce(s.label,'(unmapped)') AS label,t.n,t.total,
 round(t.total/(SELECT sum(value) FROM sample),3) AS share
FROM totals AS t LEFT JOIN site AS s ON s.id=t.site
ORDER BY t.total DESC,t.site;
```

Actual SQLite result

Result 1 · 4 rows

site	label	n	total	share
A	River	3	36.0	0.36
B	Hill	2	28.0	0.28
C	Coast	2	20.0	0.2
D	(unmapped)	1	16.0	0.16

SQLite · exit 0

Clause or feature	Meaning
`Grouped CTE`	Set the report unit to one site.
`LEFT JOIN`	Retain the unlabelled site.
`Scalar denominator`	Use the total over all observations.

Example and command index

- SQL-01-001 · Select stored columns
- SQL-01-002 · Filter individual observations
- SQL-01-003 · Find the top three
- SQL-01-004 · Return distinct values
- SQL-01-005 · Calculate a derived column
- SQL-01-006 · Classify with CASE
- SQL-02-001 · Display a value for NULL
- SQL-02-002 · Test NULL explicitly
- SQL-02-003 · Combine range and membership filters
- SQL-02-004 · Match and normalize text
- SQL-02-005 · Place NULL values last
- SQL-02-006 · Continue with keyset pagination
- SQL-03-001 · Distinguish integer and real arithmetic
- SQL-03-002 · Format a consistent identifier
- SQL-03-003 · Clean and join text
- SQL-03-004 · Guard a denominator with NULLIF
- SQL-03-005 · Measure elapsed days
- SQL-03-006 · Calculate a calendar boundary
- SQL-04-001 · Summarize the entire table
- SQL-04-002 · Group observations by site
- SQL-04-003 · Filter groups with HAVING
- SQL-04-004 · Count missing and present values
- SQL-04-005 · Calculate a weighted mean
- SQL-04-006 · Make an ordered list per group
- SQL-05-001 · Match a lookup with an inner join
- SQL-05-002 · Keep observations with missing metadata
- SQL-05-003 · Find missing relationships
- SQL-05-004 · Compare an ON filter with a WHERE filter
- SQL-05-005 · See join multiplicity
- SQL-05-006 · Join a table to itself
- SQL-06-001 · Compare with a correlated group mean
- SQL-06-002 · Name an intermediate result
- SQL-06-003 · Use EXISTS as a membership question
- SQL-06-004 · Expose the NOT IN and NULL trap
- SQL-06-005 · Compare UNION and UNION ALL
- SQL-06-006 · Intersect and subtract sets
- SQL-07-001 · Number rows within each site
- SQL-07-002 · Calculate a running total
- SQL-07-003 · Measure change from the preceding row
- SQL-07-004 · Calculate a three-row moving mean
- SQL-07-005 · Compare rank and dense rank
- SQL-07-006 · Give LAST_VALUE a complete frame
- SQL-08-001 · Generate a finite sequence
- SQL-08-002 · Generate a calendar with missing days
- SQL-08-003 · Walk a parent-child hierarchy
- SQL-08-004 · Prevent cycles in a graph walk
- SQL-08-005 · Carry an accumulated calculation
- SQL-08-006 · Expand a bill of materials
- SQL-09-001 · Reject a violated CHECK constraint
- SQL-09-002 · Enforce a foreign key
- SQL-09-003 · Compare STRICT storage with ANY
- SQL-09-004 · Store a generated column
- SQL-09-005 · Use a composite key without rowid
- SQL-09-006 · See how UNIQUE treats NULL
- SQL-10-001 · Insert a row with RETURNING
- SQL-10-002 · Update one row with RETURNING
- SQL-10-003 · Delete one row with RETURNING
- SQL-10-004 · Handle a conflict with UPSERT
- SQL-10-005 · Reuse a view definition
- SQL-10-006 · Record changes with a trigger
- SQL-11-001 · Inspect and roll back a change
- SQL-11-002 · Commit a group of writes
- SQL-11-003 · Roll back to a savepoint
- SQL-11-004 · Defer a foreign-key check until commit
- SQL-11-005 · Begin a write transaction immediately
- SQL-11-006 · Select write-ahead logging

- SQL-12-001 · Read a table-scan plan
- SQL-12-002 · Add a selective index
- SQL-12-003 · Use a covering index
- SQL-12-004 · Use a composite index prefix and range
- SQL-12-005 · Index a subset of rows
- SQL-12-006 · Index an expression
- SQL-13-001 · Extract typed JSON values
- SQL-13-002 · Compare JSON text and SQL scalars
- SQL-13-003 · Expand a JSON array into rows
- SQL-13-004 · Modify and merge JSON objects
- SQL-13-005 · Aggregate ordered values into JSON
- SQL-13-006 · Store SQLite JSONB
- SQL-14-001 · Search terms with FTS5
- SQL-14-002 · Search an exact phrase
- SQL-14-003 · Combine a prefix and an exclusion
- SQL-14-004 · Order matches by BM25
- SQL-14-005 · Highlight matching tokens
- SQL-14-006 · Restrict a full-text query by column
- SQL-15-001 · Inspect the schema catalog
- SQL-15-002 · Query column metadata as a table
- SQL-15-003 · Check declared foreign-key relationships
- SQL-15-004 · Run a database consistency check
- SQL-15-005 · Inspect affected-row counts
- SQL-15-006 · Create and verify a database copy
- SQL-16-001 · Pivot a small categorical summary
- SQL-16-002 · Find islands of consecutive dates
- SQL-16-003 · Return the top two within each group
- SQL-16-004 · Deduplicate an explicit ingestion key
- SQL-16-005 · Calculate a calendar-based rolling total
- SQL-16-006 · Compose a joined summary report

Content: CC0 1.0. Structure and publishing framework: CC BY 4.0 by Lukas C. Bossert. Repository source · Publishing framework source · License scope and third-party notices.