FIELD NOTES ISSUE 01

JOINS *without* SURPRISES.

A visual companion for seeing row matches, missing keys, and fanout before your totals betray you.

Taught by Tsumugi • drawn from executable rows


SQL
FIELD NOTES

Want more collectible plates? Vote for the instructor fan pack.

CASE FILE 01 KEY CARDINALITY

A duplicate lookup key changed the result.

A duplicated customer key made two of four orders match twice, producing six input rows for `SUM()`.

Predict the row pairs before you touch the aggregate.

02 RELATIONS BETWEEN STAGES

A query builds a new row set.

SQL does not save and update one value at a time like a Python loop. Each logical stage accepts a relation and produces another relation.

01
FROM + JOIN
build candidate row pairs

→

02
WHERE
discard pairs that fail

→

03
GROUP + HAVING
form groups, then filter them

→

04
SELECT + DISTINCT
project the final row shape

→

05
ORDER BY
arrange the final rows

VARIABLE INSTINCT

```
total = total + order.amount
```

One mutable value changes over time.

RELATIONAL INSTINCT

```
SELECT customer_id, SUM(total)
FROM joined_rows
GROUP BY customer_id;
```

Define the input relation and output grain before applying the aggregate.

Tsumugi's check

Say what one output row means after every major clause.

03 BEFORE THE JOIN

Read the key constraints before the JOIN.

A **primary key** identifies one row. A **foreign key** points at that identity. The foreign key may repeat because one order can own many fills.

PARENT TABLE

orders

order_id	PK	unique · not null
created_at		one order fact

1 many

CHILD TABLE

fills

fill_id	PK	one fill identity
order_id	FK	may repeat

01

Primary key

`orders.order_id` is the identity of an order. It cannot be NULL or duplicated.

02

Foreign key

`fills.order_id` says which order a fill belongs to. Several fills may point to the same order.

03

Join condition

`o.order_id = f.order_id` compares values. It does not require both columns to be primary keys.

Think in sets of rows, not variables being updated.

A query describes a result table. Each clause changes which rows or columns qualify for that result; it does not mutate a saved value like a Python assignment.

Forecast

Write “one row per…” beside both tables before the ON clause.

04 THE PAIRING PROMISE

A JOIN produces one row for each matching pair.

For every row on one side, SQL finds all rows on the other side that satisfy the `ON` condition. Each match becomes one output row.

```
SELECT o.id, c.name, o.total FROM orders o JOIN customers c ON c.id = o.customer_id;
```

LEFT INPUT

orders

ID	CUSTOMER_ID	TOTAL
101	1	$120
102	2	$80
103	1	$40
104	3	$260

LOOKUP

customers

ID	NAME
1	Aiko
2	Ben
3	Chie
4	Dev
5	Emi

1 Order `101` carries `customer_id = 1`.

2 The lookup contains exactly one `id = 1`.

3 One match means one joined row. The database repeats this search for every order.

JOIN OUTPUT

Four matches become four rows

4 rows

ORDER_ID	NAME	TOTAL
101	Aiko	$120
102	Ben	$80
103	Aiko	$40
104	Chie	$260

05 CHOOSE THE PRESERVED SIDE

JOIN type decides which non-matches survive.

Skip the Venn diagram. Ask which row pairs qualify, then decide which unmatched input rows must still appear.

INNER

Matched pairs only

Unmatched rows disappear from both inputs.

Use when the relationship must exist.

LEFT

Preserve the left

Every left row survives. Missing right values become synthetic NULLs.

Use for “all customers, even zero orders.”

FULL

Preserve both

Keep unmatched rows from each input after the matched pairs.

Use for reconciliation.

CROSS

Every possible pair

`m` rows × `n` rows = `m` × `n` outputs.

Use to manufacture an expected grid.

RIGHT JOIN preserves the table written second.

Swapping the inputs produces the equivalent LEFT JOIN and often makes the preserved side easier to see.

06 THE MISSING MATCH

LEFT JOIN keeps the left row alive.

When no fill matches, SQL still emits the order. Every requested column from the missing right side appears as **NULL in the result**.

```
SELECT o.order_id, f.fill_id FROM orders o LEFT JOIN fills f ON f.order_id = o.order_id;
```

order 101
two matching fills

→ 101 · F1 | 101 · F2

order 102
zero matching fills

→ 102 · NULL

LEFT JOIN OUTPUT

The NULL is a placeholder in the result

3 rows

ORDER_ID	FILL_ID	WHAT HAPPENED
101	F1	first match
101	F2	second match
102	NULL	left row preserved

The NULL fields are constructed in the JOIN output.

Order 102 has no stored fill row. The outer join still emits the order and fills the requested right-side columns with NULL.

Production scar
NULL padding belongs to the result. Do not imagine a secret blank row in the right table.

07 THE SAME CONDITION, A DIFFERENT QUESTION

ON chooses matches. WHERE chooses survivors.

A right-side filter inside ON still preserves the left row. Move it to WHERE, and the synthetic NULL row can fail the filter.

KEEP EVERY CUSTOMER

```
LEFT JOIN orders o
  ON o.customer_id = c.id
 AND o.created_at >=
     DATE '2026-07-01'
```

The date is part of the matching rule. Customers without a recent order remain.

DROP THE NON-MATCHES

```
LEFT JOIN orders o
  ON o.customer_id = c.id
WHERE o.created_at >=
      DATE '2026-07-01'
```

The output is filtered after NULL padding. Missing orders cannot satisfy the date predicate.

Cold forecast If Dee has no orders, which version still emits a Dee row? **The ON version.**

Try the break

Move one right-side predicate. Predict the result before you run it.

08 COUNT THE RIGHT THING

COUNT(*) sees output rows. COUNT(f.id) sees matches.

After a LEFT JOIN, the unmatched customer still owns one result row. **COUNT(*)** counts it; **COUNT(f.id)** ignores its NULL right-side value.

Alice

`Alice · NULL`

COUNT(*)	COUNT(f.id)
1	0

Bob

`Bob · 123`

`Bob · 456`

COUNT(*)	COUNT(f.id)
2	2

Choose a counted column that proves the event happened.

For "number of orders," count a non-null order key. For "number of joined rows," count *. Those are different questions.

Tsumugi's check

Name the event your counted column proves.

09 PROVING ABSENCE

Use row identity to prove absence.

For stale orders with no fills at all, test a non-null match key or write the intent directly with **NOT EXISTS**.

ROBUST

LEFT JOIN anti-join

```
LEFT JOIN fills f
  ON f.order_id = o.order_id
WHERE f.order_id IS NULL
```

The joined key can only be NULL when no matching fill row survived.

CLEAR INTENT

Correlated NOT EXISTS

```
WHERE NOT EXISTS (
  SELECT 1 FROM fills f
  WHERE f.order_id = o.order_id
)
```

Reads as the business rule: retain the order when no related fill exists.

DATA-DEPENDENT

Nullable attribute

```
LEFT JOIN fills f
  ON f.order_id = o.order_id
WHERE f.filled_at IS NULL
```

Also accepts a real fill row whose timestamp happens to be NULL.

A wider data contract can break a currently correct query.

Your original query passed because every current fill row had a timestamp. The production note exposed the assumption without rejecting a semantically correct result.

CASE FILE 02 · PROVE ABSENCE

10 THE FANOUT INCIDENT

Duplicate lookup keys multiply matching pairs.

Duplicate Aiko
Run the import twice

ORDERS · UNCHANGED

Four facts

ID	CUSTOMER_ID	TOTAL
101	1	$120
102	2	$80
103	1	$40
104	3	$260

CUSTOMERS · TOGGLE THIS

Lookup key grain

ID	NAME	STATE
1	Aiko	original copy
2	Ben	unique
3	Chie	unique
4	Dev	unique
5	Emi	unique
1	Aiko	imported twice

JOINED ROWS
6
+2 from one bad key

SUM(O.TOTAL)
$660
+$160 overcount

Orders 101 and 103 each find two Aiko rows. Both facts are repeated before the aggregate runs.

WHAT THE DATABASE ACTUALLY RETURNS

Two Aiko orders multiply into four result rows

KEY PROMISE BROKEN

ORDER_ID	NAME	TOTAL	MATCHED CUSTOMER ROW
101	Aiko	$120	aiko-a
101	Aiko	$120	duplicate id = 1
102	Ben	$80	ben
103	Aiko	$40	aiko-a
103	Aiko	$40	duplicate id = 1
104	Chie	$260	chie

A JOIN returns every matching pair.

A primary key or unique constraint is what limits a lookup key to one match. `DISTINCT` may hide a display symptom after multiplication has already changed an aggregate.

DISTINCT edits the final projection.

It can collapse identical display rows. It cannot undo a total already multiplied by the join.

PAIR COUNT

6 rows

two Aiko facts repeated

AGGREGATE

$660

the damage is already counted

FINAL DISTINCT

one display total

still the wrong $660

```
-- DISTINCT can collapse identical projected customer rows.
SELECT DISTINCT c.id, c.name
FROM orders o
JOIN customers c ON c.id = o.customer_id;

-- It cannot repair an aggregate fed multiplied rows.
SELECT SUM(o.total) AS total
FROM orders o
JOIN customers c ON c.id = o.customer_id;
```

Use DISTINCT only when duplicate output rows are the intended equivalence rule.

`DISTINCT` on a single aggregate row is a no-op: the input rows already produced the wrong $660. If you cannot explain why two projected rows mean the same result, do not use it as a repair reflex.

Repair the relationship at its earliest valid stage.

Choose among a uniqueness constraint, an explicit survivor rule, or pre-aggregation according to the data contract.

ENFORCE

Reject the duplicate

```
PRIMARY KEY (customer_id)
```

Correct when the lookup promises one row per customer.

DEDUPLICATE

Choose a survivor

```
QUALIFY ROW_NUMBER()
  OVER (...) = 1
```

Correct only when the survivor rule is explicit and auditable.

PRE-AGGREGATE

Match the target grain

```
GROUP BY order_id
```

Reduce many fill rows to one order summary before the order-level join.

raw fills
many / order

→

fill totals
one / order

→

orders joined
one / order

Tsumugi's check

State the survivor rule before you write ROW_NUMBER().

Ten forecasts before the answer key.

Define one result row and estimate the row count before writing SQL.

01

Grain contract

State what one row means in accounts and invoices.

Predict: which key may repeat?

02

Duplicated lookup

Invoices are 10→125, 20→80, 10→45, 30→300, 50→60. The lookup contains account 10 twice and no account 50.

Predict: joined rows and `SUM(total)`.

03

Choose the JOIN

Match: only accounts with invoices; every account; dates present in either cash table; every account × date cell.

Predict: INNER, LEFT, FULL, or CROSS.

04

The filter moved

Return members with no login since July 1.

Predict: ON or WHERE for the date?

05

Aggregate first

Find overfilled orders without multiplying order quantities.

Predict: the grain of fill_totals.

06

Count the missing

An account has no invoice match after a LEFT JOIN.

Predict: `COUNT(*)`, `COUNT(invoice_id)`, and their difference.

13B ONE MORE CORE · THREE STRETCH

07

Exists, never refunded

Paid at least once and refunded never.

Predict: one positive and one negative test.

08 · STRETCH

Role + compound match

Join assets twice, then match deposits and trades by user plus date.

Predict: why both predicates are required, though neither guarantees one-to-one matching.

09 · STRETCH

Expected rows

Build a date grid, then find the missing candles.

Predict: CROSS first, anti-join second.

10 · STRETCH

Reconcile and classify

FULL OUTER daily cash flow plus half-open fee tiers.

Predict: the boundary at exactly 100.

The eight-question JOIN preflight.

Run this before every production join. If one answer is vague, the query is not ready for an aggregate.

01 **What does one row represent in each input?**

02 **Which key should be unique?**

03 **How many right rows can match each left row?**

04 **Which non-matches must survive?**

05 **Is the right-side filter part of the match or a final filter?**

06 **What is the output grain after the JOIN?**

07 **Did aggregation happen before a many-sided JOIN?**

08 **Can a duplicate-key probe prove the result?**

RED FLAGS

unexplained row growth

DISTINCT as reflex

SUM before grain proof

Keep this page

Use the checklist until you can answer it without prompting.

APPENDIX A SHARP-PENCIL REVIEW

Answers, with the reason each query holds.

Several SQL shapes can be correct. The invariant is what matters: output grain, match key, and treatment of missing rows.

01 State the grain, then count matches

```
SELECT a.account_id,
       COUNT(i.invoice_id) AS invoice_matches
FROM accounts a
LEFT JOIN invoices i ON i.account_id = a.account_id
GROUP BY a.account_id
ORDER BY a.account_id;
```

`accounts` has one row per account and `invoices` has one per invoice. The `GROUP BY` sets the result grain to one row per account. Counting the non-null invoice key preserves Dee with zero matches.

02 Predict the fanout, then prove its cause

```
SELECT COUNT(*) AS joined_rows,
       SUM(i.total) AS joined_total
FROM invoices i
JOIN account_import a
  ON a.account_id = i.account_id;
```

```
SELECT account_id, COUNT(*) AS rows_per_key
FROM account_import
GROUP BY account_id
HAVING COUNT(*) > 1;
```

The repeated `account_id = 10` pairs both Ari invoices with two lookup rows. Four matching invoices become six joined rows and the joined total rises from 550 to 720. The second query identifies the duplicated lookup key.

03 Choose what survives

```
-- Matching account/invoice pairs only: 4 rows
SELECT COUNT(*)
FROM accounts a JOIN invoices i
  ON i.account_id = a.account_id;

-- Keep every account: 5 rows
SELECT COUNT(*)
FROM accounts a LEFT JOIN invoices i
  ON i.account_id = a.account_id;
```

```
-- Build a deliberate 2 x 2 grid: 4 rows
SELECT *
FROM (VALUES (DATE '2026-01-10'), (DATE '2026-01-11')) d(flow_date)
CROSS JOIN (VALUES (10), (20)) a(account_id);
```

INNER keeps matches. LEFT preserves the account side. CROSS creates every possible pair. FULL OUTER is reserved for reconciliation when unmatched rows from either input must survive.

04 The filter moved

```
SELECT m.member_id, m.name
FROM members m
LEFT JOIN logins l
  ON l.member_id = m.member_id
 AND l.logged_at >= DATE '2026-07-01'
WHERE l.login_id IS NULL;
```

The date belongs in `ON` because it defines a recent-login match. Testing the non-null login key proves that no qualifying login row exists.

05 Aggregate before joining

```
WITH fill_totals AS (
  SELECT order_id, SUM(quantity) AS filled_quantity
  FROM fills
  GROUP BY order_id
)
SELECT o.order_id, o.quantity, f.filled_quantity
FROM orders o
JOIN fill_totals f ON f.order_id = o.order_id
WHERE f.filled_quantity > o.quantity;
```

`fill_totals` is one row per order. The later join therefore cannot repeat an order-level quantity once per raw fill.

06 Count the missing

```
SELECT a.account_id,
       COUNT(*) AS joined_rows,
       COUNT(i.invoice_id) AS matched_invoices,
       COUNT(*) - COUNT(i.invoice_id) AS missing_invoices
FROM accounts a
LEFT JOIN invoices i ON i.account_id = a.account_id
GROUP BY a.account_id;
```

The unmatched account owns one output row, zero non-null invoice keys, and therefore one missing relationship.

07 Paid at least once, refunded never

```
SELECT u.user_id, u.name
FROM users u
WHERE EXISTS (
  SELECT 1 FROM payments p
  WHERE p.user_id = u.user_id AND p.status = 'completed'
)
AND NOT EXISTS (
  SELECT 1 FROM refunds r WHERE r.user_id = u.user_id
);
```

Membership tests keep each user once regardless of how many payment rows match. `EXISTS` checks for a completed payment, while `NOT EXISTS` checks for any refund.

08 Role aliases and a compound match

```
SELECT t.trade_id, base.name, quote.name
FROM trades t
JOIN assets base  ON base.symbol = t.base_asset
JOIN assets quote ON quote.symbol = t.quote_asset
ORDER BY t.trade_id;
```

```
SELECT DISTINCT d.user_id
FROM deposits d
JOIN trades t
  ON t.user_id = d.user_id
 AND t.executed_at::DATE = d.completed_at::DATE
WHERE d.status = 'completed'
ORDER BY d.user_id;
```

Aliases describe two roles played by the same table. The second relationship needs both user and calendar day; remove either and unrelated facts can pair. Both predicates are required, but neither alone promises one-to-one cardinality.

09 Find missing expected rows

```
WITH expected AS (
  SELECT p.pair_id, c.trade_date
  FROM pairs p
  CROSS JOIN (
    SELECT DATE '2026-01-01' + CAST(i AS INTEGER) AS trade_date
    FROM range(0, 3) t(i)
  ) c
  WHERE c.trade_date >= p.listed_at
)
SELECT e.pair_id, e.trade_date
FROM expected e
LEFT JOIN candles c
  ON c.pair_id = e.pair_id
 AND c.trade_date = e.trade_date
WHERE c.pair_id IS NULL
ORDER BY e.pair_id, e.trade_date;
```

The generated calendar and pairs create the rows that should exist. Any expected row left unmatched by the anti-join is a missing candle.

10 Reconcile and classify

```
WITH dep AS (
  SELECT completed_at AS flow_date,
         SUM(usd_value) AS deposits_usd
  FROM cash_deposits
  WHERE status = 'completed'
  GROUP BY completed_at
), wd AS (
  SELECT completed_at AS flow_date,
         SUM(usd_value) AS withdrawals_usd
  FROM cash_withdrawals
  WHERE status = 'completed'
  GROUP BY completed_at
)
SELECT COALESCE(dep.flow_date, wd.flow_date) AS flow_date,
       COALESCE(dep.deposits_usd, 0)
         - COALESCE(wd.withdrawals_usd, 0) AS net_flow_usd
FROM dep
FULL OUTER JOIN wd ON wd.flow_date = dep.flow_date
ORDER BY flow_date;
```

```
SELECT v.user_id, v.volume, f.tier_name
FROM user_volume v
JOIN fee_tiers f
  ON v.volume >= f.min_volume
 AND (v.volume < f.max_volume OR f.max_volume IS NULL)
ORDER BY v.user_id;
```

FULL OUTER JOIN preserves a date found on either side. The half-open range assigns the exact boundary of 100 to the next tier once, not twice.

Final JOIN checklist

Name both grains. Forecast match counts. Choose the preserved side. Separate match predicates from final filters. Aggregate only after the output grain is proven.

1. Predict the row count.

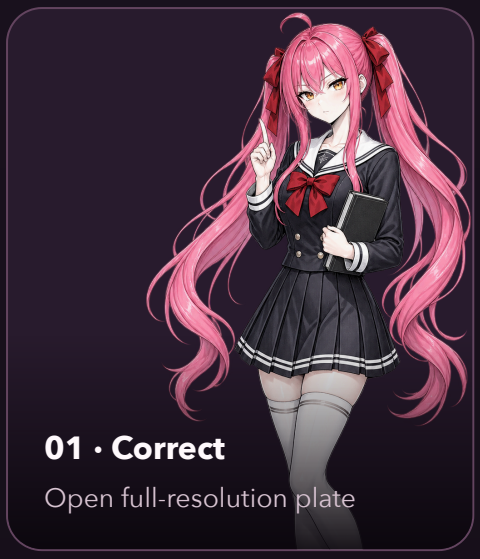
2. Run the JOIN.

3. Explain every surprise.

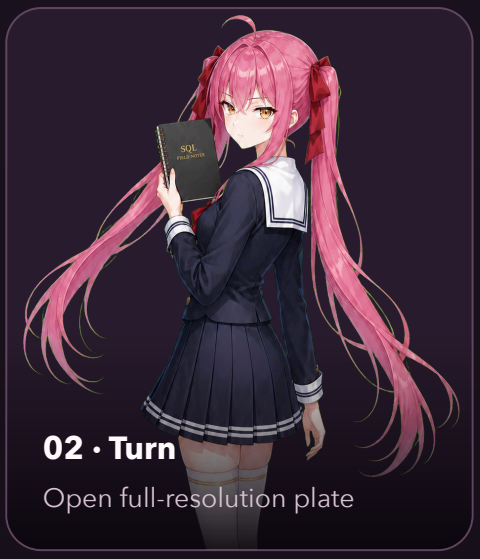
AFTER CLASS COLLECTIBLE PLATES

Five field-note keepsakes.

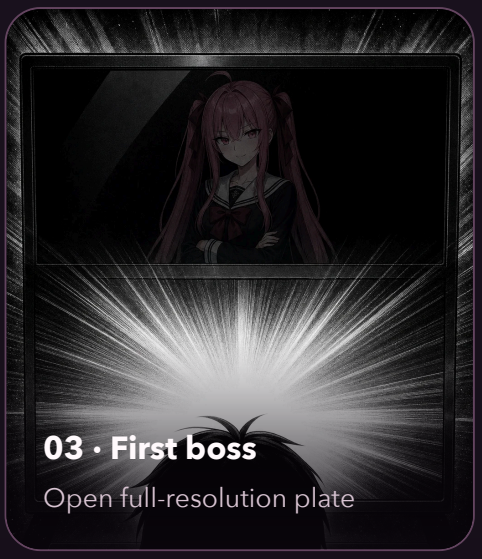
The educational guide stays broadly shareable. These mild, non-spoiler plates are optional after-class material.



01 · Correct
Open full-resolution plate



02 · Turn
Open full-resolution plate



03 · First boss
Open full-resolution plate

04 · Last call
Open full-resolution plate

05 · Faculty room
Open full-resolution plate

Collectible rule Every teaching point also appears in text, tables, or SQL.

NEXT DROP

WHAT SHOULD DROP NEXT?

Practice calendar or instructor fan pack?

Both follow-up ideas are optional. The complete 224-problem browser course remains free.

PRACTICE

Seasonal SQL calendar

A daily SQL forecast with its answer and postmortem, sized for a short practice session.

Vote: SQL calendar

COLLECTIBLE

Instructor fashion pack

High-resolution plates, expression studies, and field-note variants, published separately from the course.

Vote: fan pack

Each link records one anonymous aggregate vote containing only the event name and fixed choice, excluding account data, identifiers, and query text.