

Relix

Problem-Solving Manual

What kind of problem is this, and which parts of the language solve it — a recipe book for readers who know the language.

Generated 1 October 2026 · revision e87dc7b0e95d

Contents

About this manual	5
1. The method	6
Grain first	6
Find the quantifier	7
Decompose into views	9
Check your answer	10
2. Recipes: selection and summary	11
Which rows, and which columns	11
How many, how much, per what	14
3. Recipes: ranking	17
The most, the top N, the latest	17
4. Recipes: existence and absence	21
Has at least one	21
None, never, missing	23
5. Recipes: every and only	26
Every: related to all of a set, or all rows pass a test	26
6. Recipes: comparison	30
What changed, what differs	30
7. Recipes: graphs	33
Things that belong together	33
Reachable, how far, and by what route	35
When the rule is recursive	38

8. Recipes: time	42
As of, during, overlapping	42
In a row, per visit, gaps	45
9. Recipes: choosing and solving	48
The best combination within limits	48
What value makes this true	50
Enough cases to cover	52
10. Recipes: reshaping	55
As columns, as a list, as a tree	55
11. Recipes: explaining	58
Why is this row here, where did it come from	58
12. Recipes: sampling	61
A random, representative subset	61
13. Recipes: iteration	64
Iterate until it settles	64
14. Case studies	68
Fraud rings	68
Family history	70
Inventory allocation	72
SLA breaches	74
Reconciling two systems	76
Test planning	77
15. Engineering the solution	80
Engineering the solution	80
16. When Relix is not the tool	83

When Relix is not the tool	83
--------------------------------------	----

17. Recipe finder 85

The recipe coordinate	85
---------------------------------	----

Recipe finder	86
-------------------------	----

About this manual

This manual is for readers who already know the language and want to know which parts of it to reach for. The reference answers *what does this operator do?*; this manual answers the question that comes first — *what kind of problem is this?* — and then hands you a recipe.

Every problem in it is classified the same way, by three questions asked before any code is written:

1. **What is the grain?** Finish the sentence "*one row per* ____".
2. **What is the quantifier?** The word in the question that decides the logic — *which, how many, the most, any, none, every, connected...*
3. **What makes the data difficult?** Nested, dirty, NULL-heavy, unbounded, federated, or out of step in time.

The answers give a problem its *coordinate*, and the coordinate points at a recipe.

Each recipe carries its coordinate as one line under the title, in a **fixed format** that a reader reads and the recipe finder is generated from.

1. The method

The four moves that turn a question into a query, in order.

Chapter	What it teaches
Grain first	Decide "one row per ___" before choosing an operator
Find the quantifier	The word in the question that fixes the logic
Decompose into views	Build the answer as a pipeline of named relations
Check your answer	The checklist that tells <i>runs</i> from <i>right</i>

Grain first

Most wrong queries are wrong about the grain before they are wrong about anything else. So before you choose an operator, finish one sentence — "one row per ___" — and write the answer's heading down. That sentence is the grain, and it is the single most useful thing to decide first, because everything else follows from it and almost nothing corrects it.

The grain decides the first operator

The grain is not a detail you settle at the end; it is the shape of the answer, and the shape of the answer chooses the operator that produces it.

- *One row per customer* is a summary keyed on the customer — a **γ**.
- *One row per customer and month* is the same **γ** with a **derived key**; the extra key is the extra word in the grain.
- *One row per pair* — this order with that shipment, this record with its duplicate — is a **join**.
- *One row per group of connected things* — a household, a fraud ring — is **CLUSTER**, because the group is not a column any row already carries.

Read those the other way and the method appears: say the grain out loud, and the first operator is the one whose output has that grain. A question you cannot phrase as "one row per ___" is usually two questions, which is the subject of **decompose into views**.

Sketch the answer before you write it

Once the grain is decided, sketch the answer as a tiny inline table — the columns it has, and three or four rows you can work out by hand. This costs a minute and pays twice. It forces the grain to be concrete: if you cannot say what one row looks like, you have not decided the grain yet. And it becomes the test — the known answer you check the finished query against, which is where **checking your answer** begins.

How a grain goes wrong

Grain rarely goes wrong loudly. It goes wrong quietly, and the query still returns rows — the wrong ones. Three failures account for most of it:

- **A join fans rows out.** Join to a table with more than one matching row and every left row multiplies. *One row per order* silently becomes *one row per order line*, and a **SUM**

over it double-counts. This is the commonest grain bug, and the reason to reach for a semi-join ($\bowtie$) when you mean to *filter* rather than *combine*.

- **A γ is keyed on too much.** An extra column in the grouping list splits every group; the answer has more rows than there are things, and each aggregate is computed over a sliver.
- **A set operation removes duplicates the question needed.** $\cup$, $\cap$ and $-$ are set operations — they deduplicate. If the grain was a bag (a row *per occurrence*), a set operation quietly collapses the occurrences the question was counting.

Check the grain you got

The grain you *intended* and the grain you *got* are different claims, and the gap between them is where the bug lives. Two checks close it:

- **Count against distinct.** `COUNT` of the answer should equal `COUNT` of `6` over the key columns. If the total is larger, a join fanned out; if smaller, a set operation deduplicated. Either way the grain is not what you said.
- **Ask the engine what it can prove.** `relix.keys` lists the candidate keys the engine has established for a relation — the column sets that identify a row. If the grain you wrote down is not among them, the engine cannot see the uniqueness you are assuming, and neither, probably, can the data.

Decide the grain, sketch its answer, and these checks turn a query that *runs* into a query that is *right*. The rest of the method — the quantifier, the decomposition, the final checklist — all assume the grain is settled first.

Find the quantifier

Somewhere in almost every question is a single word that decides the logic. Find that word and the family of operators follows — the grain told you the *shape* of the answer, and the quantifier tells you the *rule* that fills it. Learning to hear it is most of what this manual teaches.

The question says...	Class	Reach for
"which ..."	Selection	σ , π , joins
"how many / how much"	Summary	γ , ROLLING, DOWNSAMPLE
"the most / top N / latest"	Ranking	TOP, WINDOW RANK, ARGMAX
"any / at least one"	Existence	$\bowtie$
"none / never / missing"	Absence	$\triangleright$, $-$
"every / all / only"	Universal	$\forall$, $\div$
"what changed / what differs"	Comparison	$-$, Δ , $\bowtie$
"connected / reachable / via / ancestors"	Graph	CLOSURE, PATH, CLUSTER, TRACE, FIX
"as of / during / overlapping"	Temporal alignment	AS-OF join, interval join
"in a row / per visit / gaps"	Sequence	SESSIONIZE, WINDOW LAG/LEAD

The question says...	Class	Reach for
"the best combination within limits"	Optimization	OPTIMIZE
"what value makes this true"	Equation	SOLVE
"enough cases to cover"	Generation	COVER, generators
"as a tree / as columns / flattened"	Reshaping	PIVOT/UNPIVOT, TREE, μ , COLLECT
"why / where did this come from"	Explanation	WHY
"a random sample / a representative subset"	Sampling	SAMPLE, SAMPLE ... ROWS
"converge / iterate until stable / simulate rounds"	Iteration	ITERATE

The **Class** column is the registry every recipe's **coordinate** draws its class from: this table, and nowhere else, is the list of classes a reader is taught and a recipe may claim.

Questions that hide their quantifier

The table works when the word is on the surface. Often it is not — the quantifier is buried in a domain noun, and the skill is translating the noun back into the word:

- "*loyal customers*" is usually **every** — customers who ordered in *every* period, not merely *some*.
- "*orphaned records*" is **none** — records with *no* matching parent.
- "*duplicates*" is **graph**, not selection — two records are the same when something links them, and *linked* runs in chains, so it is **CLUSTER**, not a join.
- "*the current price*" is **temporal alignment** — the price *as of* now.
- "*churn*" is **absence** — customers present last month and *not* this one.

When a question resists the table, restate it as plainly as you can and the word tends to surface. "*Which suppliers are fully certified?*" becomes "*suppliers that hold **every** required certificate*" — and *every* is in the table.

Two quantifiers, two views

A real question often carries more than one quantifier, and the mistake is to look for the single operator that does both. There isn't one. "*Suppliers that hold every certificate **and** have never delivered late*" is a **universal** ($\forall/\div$) and an **absence** ($\triangleright$), and the answer is the two computed separately and combined — one view each, intersected. Hearing two words is the signal to write two views, which is what **decompose into views** is for.

Difficulty is a modifier, not a class

Three questions fix a problem's coordinate: the grain, the quantifier, and what makes the data difficult. The third — nested, dirty, NULL-heavy, unbounded, federated, out of step in time — is deliberately *not* in the table above. It is a **modifier** on a recipe, not a class of its own: a *summary* over *federated* sources is still a γ , run across two connections; a *ranking* over *dirty* data is still **TOP**, with the junk filtered first. Find the quantifier to choose the recipe, then let the difficulty tell you which of the recipe's pitfalls apply.

Decompose into views

Build a solution as a pipeline of named relations, one idea to a name. A view for the candidates, a view for the ones that qualify, a view for the answer — each a single step you can read, run and check on its own. This is the habit that turns a hard question into a sequence of easy ones.

It costs nothing

The reason to decompose freely is that it is free. A view is not a materialised table the engine computes and stores; the optimiser **expands every view before it rewrites**, so the rules that push a selection down, prune a partition or fold a scan into a database see straight through the view boundaries. Ten named views and one long expression optimise to the same plan. You pay nothing for the names, and you get back a solution you can follow — so name everything worth naming.

One view per quantifier

The natural seams are the quantifiers. A question with one quantifier is one view; a question with two is two views, combined at the end. "*Suppliers fully certified and never late*" becomes `FullyCertified`, `NeverLate`, and their intersection — `Approved`. Each view answers one plain question, and the last line says how the answers fit together. When you heard two words in *find the quantifier*, this is where they become two names.

Name a view for what its rows are

Name a view after *what its rows are*, not after what produced them. `Qualified`, `Rings`, `BigSpenders`, `Breaches` — a reader learns the solution from the names alone, because each one is a noun the domain already uses. `JoinedThenFiltered` names the machinery and teaches nothing; `Approved` names the meaning and teaches the whole step. The names are the outline of the argument, so make them read like one.

Inspect each stage as you build it

Because each view is a real relation, you can look at it before you build the next. Run `query { Stage }` over a few sample rows and check the grain is what you said. Ask `:schema Stage` for the columns and types it produced. Ask `:tree Stage` for the operator tree, or `:opt/:explain` for what the optimiser and planner made of it. Building blind and debugging a hundred-line expression at the end is the slow path; building a view at a time, and checking each, is the fast one.

Generate, then dispose

Some problems have no obvious first relation — there is nothing to filter, because the answer has to be *produced*. The move there is **generate and test**: a cross product or a generator *proposes* every candidate, and a σ *disposes* of the ones that do not hold. A test matrix is `Method × Currency × Tier` narrowed by a constraint; a search is a range of values narrowed to those that satisfy an equation. Propose with $\times$, dispose with σ — and thin the survivors with `COVER` when there are too many.

When a view wants recursion

A pipeline of views walks *forward*: each reads the ones before it. When a step needs to read *itself* — every ancestor, all the parts of the parts, everything reachable — another view cannot express it, and you reach for recursion instead: `CLOSURE` for the two-column case, `FIX` for the general one, `ITERATE` when each round *replaces* the last rather than adding to it. That is the boundary of the technique: decompose until a step would have to refer to its own output, and let a recursive operator take it from there.

Check your answer

A query that returns rows is not a query that is right. It ran, it produced a table, and the table looks plausible — and none of that is evidence. The same short checklist catches most of the ways a plausible answer is wrong, and every recipe in this manual ends with the checks that apply to it. Run it before you trust a result, and especially before someone else does.

The checklist

1. **Tiny data with a known answer.** Write inline tables small enough to work the answer out by hand, and include the awkward rows on purpose: an empty group, a NULL, a duplicate, a boundary. A query that is right on ten carefully chosen rows is far more trustworthy than one that looks right on ten thousand.
2. **The grain.** Is it one row per what you said in *grain first*? `COUNT` the answer against `COUNT` of δ over the key. If they differ, a join fanned out or a set operation deduplicated — the answer is at the wrong grain, whatever else it gets right.
3. **Bag or set.** Did a `u/n/-/δ` remove duplicates the question needed to keep, or did a bag keep duplicates it wanted gone? Counting questions are bags; membership questions are sets. Using the wrong one is invisible until you check the count.
4. **NULL.** What does each predicate do with a NULL? A comparison with NULL is UNKNOWN, not false, and σ keeps only what is *true* — so `$\sigma$  status  $\neq$  "open"` drops a NULL status, and `$\neg$ (amount > 100)` does not keep a missing amount. Decide, per predicate, whether that is what you meant.
5. **Empty inputs.** What does the answer say when an input has no rows? The edge cases are where the surprises live: $\exists$ by an empty relation is satisfied by *everyone*, $\forall$ over an entity with no rows says nothing about it, and an inner join drops a key with no match. Feed an empty input and confirm the answer is the one you want.
6. **What actually ran.** `:explain` shows the plan — the operators, and which parts ran inside a database rather than in the engine. A query can be right and still do far too much work; the plan is where you find out, and the subject of *engineering the solution*.
7. **Why a row is there.** When a row is surprising — present when you expected it gone, or gone when you expected it present — `WHY` reifies its lineage as data, naming the exact source rows that produced it. Trace a row you *understand* first, then the one that puzzles you.

Not every check applies to every query, which is why each recipe names the ones that matter to it. But the discipline is the same everywhere: a result is a claim, and a claim is worth checking before you act on it.

2. Recipes: selection and summary

Recipe	For
Which rows, and which columns	Filtering rows and choosing columns, keeping the grain
How many, how much, per what	Totals and counts per group, running totals, time buckets

Which rows, and which columns

Grain: one row per input row · **Class:** Selection · **Signals:** which, that match, filter, only the ones · **Operators:** σ , π , $\bowtie$, $\ltimes$

The problem

"Which paid orders are over £100, and can I see just the order number and the amount with a surcharge added? And which orders come from a customer in the north?"

How to recognise it

The question says **which, that match, filter, or only the ones that ...** It asks for a subset of rows you already have, or a subset of their columns, and — this is the tell — **the grain does not change**. One order in, at most one order out. If the answer would have *fewer things than one per input row* it is a summary (γ); if it could have *more*, a join is fanning out and that is the mistake this recipe exists to avoid.

Two operators do almost all of it:

- **σ (select)** keeps the rows a condition matches.
- **π (project)** chooses the columns to keep, and derives new ones.

A third question — *which rows are also in another table?* — is still selection, not a summary, so it must keep each row **once**. That is a semi-join ($\ltimes$), not a plain join.

The data

```
Customers := [
| customer | region |
|-----|-----|
| Ann      | North  |
| Bo       | South  |
| Cy       | North  |
];

Orders := [
| order_id | customer | amount | status |
|-----|-----|-----|-----|
| 1         | Ann      | 120    | paid   |
| 2         | Ann      | 40     | paid   |
| 3         | Bo       | 200    | open   |
| 4         | Cy       | 90     | paid   |
| 5         | Dee      | 60     | paid   |
];
```

Dee has an order but is not a customer; Bo's big order is not paid.

Recipe 1: keep the rows, then the columns

σ takes a condition — combine tests with $\wedge$ and $\vee$. Keep the paid orders over £100:

```
query {  $\sigma$  amount > 100  $\wedge$  status = "paid" (Orders) };
```

order_id	customer	amount	status
1	Ann	120	paid

(1 row)

π chooses and **derives** columns; `expr  $\rightarrow$  name` names a derived one. Name a view after what its rows are, and build on it:

```
HighValue := {  $\sigma$  amount > 100 (Orders) };  
query {  $\pi$  order_id, customer, amount (HighValue) };  
query {  $\pi$  order_id, amount, amount * 1.1  $\rightarrow$  with_surcharge (HighValue) };
```

order_id	customer	amount
1	Ann	120
3	Bo	200

(2 rows)

order_id	amount	with_surcharge
1	120	132
3	200	220

(2 rows)

Recipe 2: keep the rows that are also in another table

Which orders come from a northern customer? This is a filter on `Orders` — one row per order — so reach for a **semi-join**, which keeps a left row when a match exists and never duplicates it. A semi-join takes a condition, like every join but the natural one:

```
North := {  $\sigma$  region = "North" (Customers) };  
query { Orders  $\ltimes$  Orders.customer = North.customer North };
```

order_id	customer	amount	status
1	Ann	120	paid
2	Ann	40	paid
4	Cy	90	paid

(3 rows)

A plain join answers a *different* question — it brings the customer's columns along, so it is the right tool when you want them:

```
query {  $\pi$  order_id, customer, region (Orders  $\bowtie$  North) };
```

order_id	customer	region
1	Ann	North
2	Ann	North
4	Cy	North

(3 rows)

Variations

- **Membership against a fixed list** rather than a table: build the list as an inline relation and semi-join to it, or use σ region = "North" $\vee$ region = "West" (...) when it is two or three values.

- **The rows *not* in the other table** is the Absence class — an anti-join ($\triangleright$), covered there.
- **A derived flag instead of a filter:** $\pi \dots, \text{amount} > 100 \rightarrow \text{is_big}(\text{Orders})$ keeps every row and labels it, when the caller wants to see what was excluded.

Pitfalls

- **A join to filter can fan out.** If the other table has more than one matching row, a plain join multiplies the left rows — turning "one row per order" into "one row per order *per match*". A semi-join cannot: it is the safe way to *filter*. With two contacts for Ann, the join doubles her orders and the semi-join does not:

```

Contacts := [
| customer | channel |
|-----|-----|
| Ann      | email  |
| Ann      | phone  |
| Cy       | email  |
];
query {  $\pi$  order_id, customer, amount (Orders  $\bowtie$  Contacts) };
query { Orders  $\bowtie$  Orders.customer = Contacts.customer Contacts };

```

order_id	customer	amount
1	Ann	120
1	Ann	120
2	Ann	40
2	Ann	40
4	Cy	90

(5 rows)

order_id	customer	amount	status
1	Ann	120	paid
2	Ann	40	paid
4	Cy	90	paid

(3 rows)

- **NULL is not a value.** $\sigma \text{ status} = \text{"paid"}$ drops a row whose status is NULL, because $\text{NULL} = \text{"paid"}$ is UNKNOWN, not false. If a missing status should count, say so with Nz or a second $\vee \text{IsNull}(\text{status})$.
- **$\neq$ also drops NULLs.** $\sigma \text{ status} \neq \text{"open"}$ does **not** keep a NULL status — $\text{NULL} \neq \text{"open"}$ is UNKNOWN too. "Not open" and "missing" are different questions.
- **π can rename away a name you still need later.** A derived column shadows an input column of the same name; give it a fresh name unless you mean to replace it.

Check it

- **The grain is one row per input row.** COUNT of the answer must be $\leq \text{COUNT}$ of the input. If it is larger, a join fanned out — that is the bug.
- Put a **NULL** in the column you filter on and decide, on tiny data, whether it should survive. σ will drop it; that is often right and sometimes not.
- Put a row in the input whose key is **absent** from the reference table (Dee) and confirm the semi-join drops it.

Related

- **Every: related to all of a set** — when the filter is *matches all of*, not *matches any of*.
- **Has at least one** and **None, never, missing** — $\times$ is *at least one*, $\triangleright$ is *none*.
- **How many, how much, per what** — when the answer has fewer rows than the input, it is a γ .
- Reference pages (docs/reference): `selection`, `projection`, `semi-join`, `natural join`.

How many, how much, per what

Grain: one row per group · **Class:** Summary · **Signals:** how many, how much, total, average, per, running total, per interval · **Operators:** γ , `ROLLING`, `DOWNSAMPLE`

The problem

"How many orders and how much money per region? What is each region's running total through the morning? And how much came in per five-minute window?"

How to recognise it

The question says **how many, how much, total, average**, or carries a **per** — *per region, per day, per customer*. The answer has **fewer rows than the input**: many rows go in, one row per group comes out. That collapse is the signature of a summary, and the group is the grain.

Three shapes, and the *per what* tells them apart:

- **Per a key** — one row per region, per customer. That is γ .
- **A running value, keeping every row** — a running total, a moving average. That keeps the grain and adds a column: `ROLLING`.
- **Per a time interval** — per hour, per five minutes. That is `DOWNSAMPLE`.

The data

Order events through one morning. An inline table's columns are STRING or NUMBER, so the timestamp is converted to a real `TIMESTAMP` first — `DOWNSAMPLE` needs one to align its windows.

```
Orders := [
| at                | region | amount |
|-----|-----|-----|
| 2026-06-01T09:02:00Z | East   | 100    |
| 2026-06-01T09:05:00Z | East   | 120    |
| 2026-06-01T09:11:00Z | West   | 80     |
| 2026-06-01T09:13:00Z | East   | 90     |
| 2026-06-01T09:14:00Z | West   | 95     |
];
```

```
Typed := {  $\pi$  to_timestamp(at)  $\rightarrow$  at, region, amount (Orders) };
```

Recipe 1: one row per group (γ)

γ lists the grouping keys, then the aggregates, each named. `COUNT(*)` counts rows; `SUM`/`AVG`/`MIN`/`MAX` skip NULLs.

```
query { γ region, COUNT(*) → orders, SUM(amount) → total, AVG(amount) → mean (Typed) };
```

region	orders	total	mean
East	3	310	103.3333333333
West	2	175	87.5

(2 rows)

The keys are the grain. γ region gives one row per region; add a key and the grain gets finer — one row per region *and* something else. Decide "*one row per* ___" before you write the γ , and the keys follow.

Recipe 2: a running total, keeping every row (ROLLING)

A running total is not a collapse — every order stays, and each carries the total so far. ROLLING ... OVER ALL ROWS is the cumulative frame; SORT orders it and PER restarts it per group.

```
query { ROLLING SUM(amount) OVER ALL ROWS SORT at ASC PER region AS running (Typed) };
```

at	region	amount	running
2026-06-01T09:02:00Z	East	100	100
2026-06-01T09:05:00Z	East	120	220
2026-06-01T09:13:00Z	East	90	310
2026-06-01T09:11:00Z	West	80	80
2026-06-01T09:14:00Z	West	95	175

(5 rows)

OVER n ROWS is the other frame — a trailing window of n rows, for a moving average.

Recipe 3: per time interval (DOWNSAMPLE)

How much per five minutes? is a group whose key is a **time bucket**. DOWNSAMPLE floors each timestamp to the start of its window and consolidates within it.

```
query { DOWNSAMPLE at BY '5m' USING SUM (Typed) };
```

bucket	sum_amount
2026-06-01T09:00:00Z	100
2026-06-01T09:05:00Z	120
2026-06-01T09:10:00Z	265

(3 rows)

AVG/SUM keep the numeric columns only — there is no average of a region name — so region is dropped unless it is a grouping key. Add PER to keep it and bucket within each region:

```
query { DOWNSAMPLE at BY '5m' USING SUM PER region (Typed) };
```

region	bucket	sum_amount
East	2026-06-01T09:00:00Z	100
East	2026-06-01T09:05:00Z	120
West	2026-06-01T09:10:00Z	175
East	2026-06-01T09:10:00Z	90

(4 rows)

Variations

- **MIN/MAX in DOWNSAMPLE** consolidate every scalar column, not just numbers, because they compare rather than compute — the alphabetically first host, the earliest time.
- **Having** — to keep only the groups whose aggregate passes a test, put a `σ` above the `γ`: `σ total > 200 (…)`.
- **The row at the maximum**, not the maximum value, is a Ranking question (`ARGMAX`), not a summary.

Pitfalls

- **The grain is the keys, and it is easy to key on too much.** An extra column in the `γ` list silently splits every group. If the answer has more rows than you expected, count the distinct keys.
- **COUNT(*) and COUNT(col) differ on NULLs.** `COUNT(*)` counts rows; `COUNT(col)` counts rows where `col` is not NULL. When some rows have a missing value, they are different numbers and only one answers the question.
- **AVG ignores NULLs, it does not treat them as zero.** The average of `10`, `NULL` is `10`, not `5`. If a missing value should count as zero, `Nz(col, 0)` it first.
- **A DOWNSAMPLE bucket with no rows produces no row.** The series has gaps where nothing happened, so a `ROLLING ... OVER 3 ROWS` over the buckets spans present buckets, not consecutive intervals.
- **ROLLING needs a total order to be deterministic.** Two rows with equal `SORT` keys can fall either way in the frame; add a tie-breaker key when it matters.

Check it

- **The grain is one row per group.** `COUNT` of a `γ`'s output equals `COUNT` of `δ` of its key columns over the input — if not, a key is finer than you think.
- The totals must **reconcile**: `SUM` of the per-group totals equals the grand `SUM(amount)` over the whole input.
- Put a **NULL** amount in and watch `COUNT(*)`, `COUNT(amount)` and `AVG` diverge on tiny data.

Related

- **Which rows, and which columns** — when the answer keeps the input's grain instead of collapsing it.
- **The most, the top N, the latest** — *the most / top N* wants the row, not the total (`TOP`, `ARGMAX`).
- **In a row, per visit, gaps** — *gaps between events* is `SESSIONIZE`, a cousin of the time bucket.
- Reference pages (docs/reference): `group`, `ROLLING`, `DOWNSAMPLE`.

3. Recipes: ranking

Recipe	For
The most, the top N, the latest	The top rows, the latest per key, and ties

The most, the top N, the latest

Grain: one row per top thing (or per key, for the latest) · **Class:** Ranking · **Signals:** the most, top N, highest, latest, best per, runner-up · **Operators:** TOP, WINDOW
RANK/ROW_NUMBER/DENSE_RANK, ARGMAX

The problem

"What are the two best-selling products? The best seller in each region? The latest price for each product? And when two products tie for best, do I want one of them or both?"

How to recognise it

The question says **the most, the highest, top N, latest, best per, Nth**. It is about position in an order, not a total — so the answer is a **row**, carrying its columns, not an aggregated number. That is what separates ranking from summary: *how much did the top region sell* is a γ ; *which region sold the most* is a rank.

The choice of operator is really a choice about what you want back, and about ties:

- **The top row(s), with all their columns** → TOP ... PER.
- **One value from the top row** (its name, its id) → ARGMAX.
- **Every row, labelled with its rank**, so you decide what to keep → WINDOW RANK and its family.

The data

```
Sales := [  
| region | product | units |  
|-----|-----|-----|  
| North  | yo-yo   | 30    |  
| North  | kite    | 30    |  
| North  | drum    | 10    |  
| South  | atlas   | 20    |  
| South  | globe   | 25    |  
];
```

North has a **tie**: yo-yo and kite both sold 30. That tie is the interesting part.

Recipe 1: the top rows (TOP)

TOP n <sort> keeps the top n rows; PER restarts the count per group. It returns whole rows.

```
query { TOP 2 units DESC (Sales) };  
query { TOP 1 units DESC PER region (Sales) };
```

region	product	units
North	yo-yo	30
North	kite	30

```
(2 rows)
region  product  units
-----
North   yo-yo     30
South   globe     25
(2 rows)
```

Note that `TOP 1 PER region` returns **one** North row even though two are tied — it keeps a fixed number of rows, and breaks the tie for you. Whether that is what you want is Recipe 3.

Recipe 2: one value from the top row (ARGMAX)

When you want a single field of the top row — its name, its id — and one row per group, `ARGMAX(rank, yield)` reads "*rank by the first, return the second*". It lives inside a `γ`, so it composes with other aggregates.

```
query { γ region, ARGMAX(units, product) → top_product, MAX(units) → best (Sales) };
```

```
region  top_product  best
-----
North   yo-yo         30
South   globe         25
(2 rows)
```

`MAX(units)` gives the number; `ARGMAX(units, product)` gives *which product* achieved it — the row-with-the-maximum lookup SQL needs a window or a self-join for.

Recipe 3: rank every row, then decide (WINDOW RANK)

`TOP 1` decides the tie for you. When you want **both** tied leaders, rank every row and filter — `RANK()` gives tied rows the same rank, so `rnk ≤ 1` keeps them all:

```
Ranked := { WINDOW RANK() SORT units DESC PER region AS rnk (Sales) };
query { σ rnk ≤ 1 (Ranked) };
```

```
region  product  units  rnk
-----
North   yo-yo     30     1
North   kite      30     1
South   globe     25     1
(3 rows)
```

The three ranking functions differ only on ties, and the difference is the whole point of choosing between them:

- `ROW_NUMBER()` — 1, 2, 3 — never ties; breaks a tie arbitrarily.
- `RANK()` — 1, 1, 3 — ties share a rank, then a gap.
- `DENSE_RANK()` — 1, 1, 2 — ties share a rank, no gap. Use it for "*the Nth distinct value*": `dr = 2` is the runner-up **tier**, even when several share the top.

Recipe 4: the latest per key (ROW_NUMBER)

Latest per key is a ranking in disguise: number each key's rows newest-first, keep number 1, drop the number. This is the SQL `ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...) = 1` idiom.

```
PriceHistory := [
| product | at          | price |
```

```
|-----|-----|-----|
| kite   | 2026-06-01 | 8   |
| kite   | 2026-06-03 | 9   |
| kite   | 2026-06-02 | 7   |
| drum   | 2026-06-05 | 20  |
];
```

```
Latest := { WINDOW ROW_NUMBER() SORT at DESC PER product AS rn (PriceHistory) };
query {  $\pi$  product, at, price ( $\sigma$  rn = 1 (Latest)) };
```

product	at	price
kite	2026-06-03	9
drum	2026-06-05	20

(2 rows)

Use `ROW_NUMBER` here, not `RANK`: you want exactly one row per key even if two share a timestamp, so an arbitrary tie-break is the right behaviour.

Variations

- **Top N% rather than top N** — `WINDOW NTILE(n)` or `PERCENT_RANK`.
- **The smallest** — `TOP ... ASC, ARGMIN`, or a `SORT ... ASC`.
- **Ranking a JDBC-backed table** pushes down to PostgreSQL/H2 as a SQL window; `TOP` and the windows fold into the scan.

Pitfalls

- **TOP N and `RANK()` $\leq N$ disagree on ties.** `TOP 1` returns exactly one row; `RANK() = 1` returns every row tied for first. This is the single most common ranking bug — decide "one, or all the tied ones?" before you choose.
- **`RANK` skips, `DENSE_RANK` does not.** After two rows tied at 1, `RANK` gives the next row 3 and `DENSE_RANK` gives it 2. "The 2nd-highest distinct value" needs `DENSE_RANK`; `RANK = 2` would find nothing when two share first.
- **A window is blocking and adds a column; `TOP` filters.** `$\sigma$  rn  $\leq 3$` over a window runs after ranking every row — it is a filter on the output, not a shortcut through it. `TOP` is the one that keeps only the top rows.
- **`SORT` must be a total order for a stable answer.** Equal sort keys let a row fall either way; add a tie-breaker (`SORT units DESC, product ASC`) when the choice must be repeatable.

Check it

- Put a **tie** in the data (as North has) and run both `TOP 1 PER` and `RANK()  $\leq 1$` . If they return different row counts — and they should — you have proven which question each one answers.
- The grain of `TOP n PER k` is *at most n rows per k*: `COUNT` per key must be $\leq n$.
- For *latest per key*, `COUNT` of the answer equals the number of distinct keys.

Related

- **How many, how much, per what** — when you want the total, not the row that holds it.

- Which rows, and which columns — TOP is a cousin of σ that keeps a ranked subset.
- In a row, per visit, gaps — WINDOW LAG/LEAD compare a row with its neighbour.
- Reference pages (docs/reference): top, window-ranking, argmax.

4. Recipes: existence and absence

Recipe	For
Has at least one	Keeping an entity when a matching row exists
None, never, missing	Keeping an entity with no match; what a reference list lacks

Has at least one

Grain: one row per entity · **Class:** Existence · **Signals:** any, at least one, has, ever, some, with a · **Operators:** ⋈

The problem

"Which customers have placed at least one order? Which have ever placed one over £100?"

How to recognise it

The question says **any, at least one, has, ever, some, or with a** It keeps an entity when a matching row **exists** on the other side, and the tell is that the answer is *one row per entity* — the same customer, once, no matter how many orders they placed. If a plain join would return the customer once per order, existence wants them once: that is a **semi-join** (⋈), not a join.

A semi-join is the join you reach for when you want to *filter by existence* rather than *bring the other table's columns in*.

The data

```
Customers := [
| customer |
|-----|
| Ann      |
| Bo       |
| Cy       |
| Dee      |
];

Orders := [
| order_id | customer | amount |
|-----|-----|-----|
| 1         | Ann      | 120    |
| 2         | Ann      | 40     |
| 3         | Bo       | 200    |
| 4         | Cy       | 30     |
];
```

Ann has two orders, Cy has one small one, Dee has none.

Recipe: keep the entity when a match exists (⋈)

A semi-join takes a condition and keeps each left row at most once — Ann does not appear twice for her two orders:

```
query { Customers ⋈ Customers.customer = Orders.customer Orders };
```

```
customer
-----
Ann
Bo
Cy
(3 rows)
```

At least one that also passes a test puts the test in the join condition. Which customers have ever ordered over £100?

```
query { Customers ⋈ Customers.customer = Orders.customer ∧ Orders.amount > 100 Orders };
```

```
customer
-----
Ann
Bo
(2 rows)
```

The condition is *exists an order that is both this customer's and over £100* — one qualifying order is enough to keep the customer.

Variations

- **At least N**, not at least one, is a Summary with a filter: $\sigma n \geq 2$ (γ customer, $\text{COUNT}(*)$ $\rightarrow n$ (Orders)), then semi-join back if you need the customer's other columns.
- **The entities *without* a match** is the **Absence** class — an anti-join ($\triangleright$). Existence and absence are the two halves of the same question.
- **Existence across three tables** chains: semi-join to the second, then the result to the third.

Pitfalls

- **A join is not a semi-join.** A plain join to test existence duplicates the entity once per match and drags the other table's columns along. To *filter by existence*, $\ltimes$ is the operator — it keeps each entity once and adds nothing.
- **"At least one over £100" is not "every order over £100".** The semi-join keeps a customer with *one* qualifying order even if their others are small. If you mean *all* their orders qualify, that is the **Universal** class ($\forall$).
- **NULL never matches.** A row whose join key is NULL matches nothing, so it is treated as *no match exists* — usually right, but decide it.

Check it

- **The grain is one row per entity.** **COUNT** of the answer must be $\leq$ the number of distinct entities, and equal to **COUNT** of δ on the entity key. If it is larger, a join fanned out where a semi-join was meant.
- Put an entity with **two** matches (Ann) and confirm it appears **once**.
- Put an entity with **no** match (Dee) and confirm it is absent — then check the **Absence** recipe returns exactly Dee.

Related

- **None, never, missing** — the other half: $\triangleright$ keeps the entities with no match.

- Which rows, and which columns — semi-join as a membership filter.
- Every: related to all of a set — every, not any.
- Reference pages (docs/reference): semi-join, anti-join.

None, never, missing

Grain: one row per entity · **Class:** Absence · **Signals:** none, never, no, without, missing, not in, orphaned · **Operators:** ▷, −

The problem

"Which customers have never placed an order? Which catalogue items are not stocked at all?"

How to recognise it

The question says **none, never, no, without, missing, not in**, or **orphaned**. It keeps an entity when **no** matching row exists — the mirror of **existence**. Two operators express it, and which one you want depends on what you already have:

- ▷ (**anti-join**) — keep the left rows that have **no** match on the right, on a condition. Keeps the left row's columns.
- − (**difference**) — the rows in one relation that are not in another, when both are lists of the *same shape* (a reference list against an actual list).

The trap that defines this class is NULL: *absence* and *unknown* are different, and a careless test conflates them.

The data

```
Customers := [
| customer |
|-----|
| Ann      |
| Bo       |
| Cy       |
| Dee      |
];

Orders := [
| order_id | customer | amount |
|-----|-----|-----|
| 1         | Ann      | 120    |
| 2         | Ann      | 40     |
| 3         | Bo       | 200    |
| 4         | Cy       | 30     |
];
```

Dee has never ordered.

Recipe 1: never (anti-join)

▷ keeps the left rows with **no** matching right row. Which customers have never ordered?

```
query { Customers ▷ Customers.customer = Orders.customer Orders };
```

```
customer
```

```
Dee
(1 row)
```

"None over £100" is not the same as "never ordered". *No order over £100* means everyone except the customers who have **at least one** such order — so build the positive set and take it away. Ann (120) and Bo (200) qualify as big spenders; the rest do not:

```
BigSpenders := { ⋈ (π customer (σ amount > 100 (Orders))) };
query { Customers - BigSpenders };
```

```
customer
-----
Cy
Dee
(2 rows)
```

Note the shape: *none* is *everyone minus the ones with at least one*. Reaching straight for $\sigma \text{ amount} \leq 100$ would be wrong — it would keep Cy's small order but say nothing about Dee, and it would wrongly admit a customer who has *both* a large and a small order.

Recipe 2: missing from a reference list (–)

When you have two lists of the same kind of thing — what *should* be there and what *is* — the entries missing from the actual list are the set difference. Both sides must be projected to the same heading first.

```
Catalogue := [
| sku |
|-----|
| A   |
| B   |
| C   |
| D   |
];

Stocked := [
| sku | qty |
|-----|-----|
| A   | 5   |
| C   | 0   |
| D   | 2   |
];

query { π sku (Catalogue) - π sku (Stocked) };
```

```
sku
---
B
(1 row)
```

Missing is not the same as present-but-empty. C is stocked with a quantity of zero — it *has* a row, so it is not in the difference. "Not stocked" (B) and "out of stock" (C) are different questions:

```
query { π sku (σ qty = 0 (Stocked)) };
```

```
sku
---
C
(1 row)
```

Variations

- **Unavailable = missing u out of stock** — the union of the two sets above, when the business cares about "cannot ship" rather than the reason.
- **Present in neither list** or **symmetric difference** (in one or the other but not both) is the **Comparison** class (Δ).
- **Anti-join keeps columns; difference does not need a key match.** Use $\triangleright$ when you want the left row's other columns; use $-$ when you have two clean lists.

Pitfalls

- $\triangleright$ **and** $\sigma \dots \neq$ **are different.** An anti-join asks *does any matching row exist*; a σ with $\neq$ tests one row at a time and cannot see "none across the group". *No order over £100* needs the anti-join / difference shape, not $\sigma \text{ amount} \leq 100$.
- **NULL is not absence.** A NULL join key matches nothing, so an anti-join **keeps** a left row whose key is NULL (it has "no match"). If NULL means "not yet known" rather than "definitely none", filter the NULLs first and decide deliberately.
- $-$ **needs identical headings.** Project both sides to the same columns; a stray column on one side makes the difference empty or wrong.
- $-$ **is set difference — it removes duplicates.** If the question is about counts (a bag), $-$ is the wrong tool.

Check it

- **Existence and absence must partition the whole.** The customers from the **existence** recipe plus the customers from Recipe 1 here should be every customer, with no overlap. **COUNT** them and check they sum to the total.
- Put a row with a **NULL** key in the left relation and watch the anti-join keep it — then decide whether that is what you meant.
- Put an entity in the reference list that is **present but zero** (C) and confirm the difference does *not* flag it.

Related

- **Has at least one** — the mirror image; $\times$ keeps the entities with a match.
- **What changed, what differs** — Δ for *in one or the other but not both*.
- **Every: related to all of a set** — *only from a set* is built from an anti-join too.
- Reference pages ([docs/reference](#)): [anti-join](#), [difference](#).

5. Recipes: every and only

Recipe	For
Every: related to all of a set, or all rows pass a test	Related to <i>all</i> of a set, or every row passing

Every: related to all of a set, or all rows pass a test

Grain: one row per entity · **Class:** Universal · **Signals:** every, all, always, only, never failed ·

Operators: $\div$, $\forall$, $\triangleright$

The problem

"Which suppliers hold every certification we require, and have delivered on time every time?"

How to recognise it

The question says **every**, **all**, **always**, **only** or **never failed**. It is about a set of things an entity has, or a history an entity has built up, and it keeps the entity only when the whole set passes.

The word every covers three different questions, and telling them apart is most of the work:

The question	What it asks	Reach for
"holds every required certificate"	the entity's set contains a given set	$\div$
"delivered on time every time"	each of the entity's rows passes a test	$\forall$
"holds only approved certificates"	nothing in the entity's set is outside a given set	$-$ and $\triangleright$

Exactly these is the first and the third together.

The data

A procurement team keeps the certifications each supplier holds, the ones it requires, and a delivery log. Delta holds no certificates and has never delivered.

```
Certifications := [
| supplier | cert   |
|-----|-----|
| Acme     | IS09001 |
| Acme     | HACCP   |
| Birch    | IS09001 |
| Cobalt   | IS09001 |
| Cobalt   | HACCP   |
| Cobalt   | Organic |
];

Required := [
| cert   |
|-----|
| IS09001 |
| HACCP   |
];
```

```
Suppliers := [
| supplier |
|-----|
| Acme     |
| Birch    |
| Cobalt    |
| Delta    |
];

Deliveries := [
| supplier | delivery | status |
|-----|-----|-----|
| Acme     | 1        | on time |
| Acme     | 2        | on time |
| Birch    | 3        | on time |
| Birch    | 4        | late    |
| Cobalt   | 5        | on time |
];
```

Recipe 1: contains a set (÷)

Divide the pairs by the set that must be covered. The result keeps the columns of the left relation that are not in the right, so here one row per supplier.

```
FullyCertified := { Certifications ÷ Required };
query { FullyCertified };
```

```
supplier
-----
Acme
Cobalt
(2 rows)
```

Cobalt's extra `Organic` does not matter: division asks whether the set is **covered**, not whether it **matches**.

Recipe 2: every row passes (∀)

Group by the entity and keep the group only when every row satisfies the condition.

```
AlwaysOnTime := { ∀ supplier : status = "on time" (Deliveries) };
query { AlwaysOnTime };
```

```
supplier
-----
Acme
Cobalt
(2 rows)
```

Recipe 3: only from a set (– and ▷)

"Only" is the negation turned round: an entity qualifies when **no** row falls outside the set. Find the rows that fall outside with an anti-join, then take their owners away from everyone.

```
Unapproved := { Certifications ▷ Certifications.cert = Required.cert Required };
OnlyApproved := { π supplier (Certifications) – π supplier (Unapproved) };
query { OnlyApproved };
```

```
supplier
-----
Acme
```

Birch
(2 rows)

Exactly the required set is both at once:

```
ExactlyRequired := { FullyCertified n OnlyApproved };  
query { ExactlyRequired };
```

supplier

Acme
(1 row)

Putting it together

The question at the top had two **every*s*, so it is two views and an intersection:

```
Approved := { FullyCertified n AlwaysOnTime };  
query { Approved };
```

supplier

Acme
Cobalt
(2 rows)

Pitfalls

- **An entity with no rows is not in the answer.** Delta has never delivered, so it has no group, so $\forall$ says nothing about it. It is not "on time every time" vacuously. If the business rule says a supplier with no deliveries passes (or explicitly fails), start from `Suppliers` and say so: $\pi \text{ supplier } (\text{Suppliers}) - \pi \text{ supplier } (\sigma \text{ status} \neq \text{"on time"} (\text{Deliveries}))$ gives *never late*, which includes Delta.
- **Dividing by an empty set is satisfied by everyone.** If `Required` is empty, every supplier in `Certifications` covers it. That is the correct logic and usually the wrong business answer, so check the divisor is not empty when it comes from a query rather than a fixed list.
- **NULL fails $\forall$.** A row whose condition is UNKNOWN disqualifies its group, so a delivery with a missing status disqualifies its supplier. Decide whether that is what you mean, and use `Nz` or `IsNull` to say otherwise.
- **Division needs declared headings.** A schema-on-read source (JSON, HTTP, MongoDB) must be projected into named columns first.

Check it

- Try an entity with **no** rows, one with **extra** rows, and one with a single failing row. The three variants disagree about exactly these cases, and that is how you tell which one you wrote.
- The grain is one row per entity. `COUNT` the answer against `COUNT` of its δ ; they should be equal.

Related

- [Things that belong together](#)
- [Has at least one](#) and [None, never, missing](#) — $\ltimes$ is *at least one*, $\triangleright$ is *none*

- Reference pages (docs/reference): division, for-all, anti-join

6. Recipes: comparison

Recipe	For
What changed, what differs	Reconciling two snapshots, or two systems

What changed, what differs

Grain: one row per thing that differs · **Class:** Comparison · **Signals:** what changed, what differs, reconcile, added, removed, drift, mismatch · **Operators:** $-$, Δ , $\bowtie$

The problem

"What changed between yesterday's price list and today's? And when I reconcile the two, which items were added, which were removed, and which had their price changed?"

How to recognise it

The question compares **two** relations of the same kind of thing and asks what is **different** — *what changed, what was added or removed, where do these two systems disagree, has this drifted*. There are two depths of answer, and the tell is whether you need to see the values side by side:

- **Just the rows that differ**, no alignment — Δ (symmetric difference): the rows in one relation or the other but not both.
- **Aligned by a key, so you can see old and new together** and label each as added / removed / changed — a full outer join ($\bowtie$) with a null test.

The data

Yesterday's and today's price list — one changed price, one removed item, one added.

```
Yesterday := [  
| sku | price |  
|-----|  
| A   | 10    |  
| B   | 20    |  
| C   | 30    |  
];
```

```
Today := [  
| sku | price |  
|-----|  
| A   | 10    |  
| B   | 25    |  
| D   | 40    |  
];
```

A is unchanged, B changed 20 → 25, C was removed, D was added.

Recipe 1: just the discrepancies (Δ)

When the two relations already share the same columns and you only want the rows that disagree, symmetric difference is the whole answer:

```
query { Yesterday  $\Delta$  Today };
```

sku	price
B	20
C	30
B	25
D	40

(4 rows)

Δ is $(R - S) \cup (S - R)$ — everything in exactly one side. Note that a *changed* row shows up as **two** rows: B's old 20 and its new 25, unaligned. That is the limit of Δ — it tells you B is involved, not that 20 became 25.

Recipe 2: reconcile by key (⋈)

To see old and new **side by side** and classify each item, align the two on their key with a full outer join. Rename the value column on each side first, so both survive the join:

```
Y := { p (price → price_y) (Yesterday) };
T := { p (price → price_t) (Today) };
Recon := { Y ⋈ Y.sku = T.sku T };
query { Recon };
```

sku	price_y	sku_r	price_t
A	10	A	10
B	20	B	25
C	30	NULL	NULL
NULL	NULL	D	40

(4 rows)

A full outer join keeps every key from both sides, filling the missing side with NULL. Now the discrepancies are exactly the rows where the two prices disagree **or** one side is NULL — and because a comparison with NULL is UNKNOWN, the NULL sides need an explicit null test:

```
query {
  π Nz(sku, sku_r) → sku, price_y, price_t (
    σ price_y ≠ price_t v price_y = NULL v price_t = NULL (Recon))
};
```

sku	price_y	price_t
B	20	25
C	30	NULL
D	NULL	40

(3 rows)

$Nz(sku, sku_r)$ collapses the two key columns into one — for a removed row the key is on the left, for an added row on the right. The row now reads as a diff: a value on both sides that disagree is a **change**, a left-only value is a **removal**, a right-only value is an **addition**.

Variations

- **Only additions**, or only removals: a one-directional difference — $\pi sku (Today) - \pi sku (Yesterday)$ is the added keys, and the reverse is the removed keys. That is the **Absence** shape.
- **A status column** instead of reading the NULLs by eye: derive it, e.g. $price_y = NULL \rightarrow was_added$, and project it.

- **Reconciling across two databases** is this recipe with each side a connection — the full outer join and the null tests are identical; only the sources change. That is the *reconciling two systems* case study (Part III).

Pitfalls

- **Δ shows a change as two rows.** For *what changed*, that is often enough; for *what did it change from and to*, you need the keyed join. Choosing Δ when you needed the alignment is the common mistake.
- **A full outer join renames the shared key.** The right side's `sku` comes back as `sku_r`; the unmatched rows have NULL on one side, so `Nz(sku, sku_r)` (or a null test) is how you recover a single key. Do not project `sku` alone — it is NULL for every added row.
- **NULL breaks the disagreement test.** `price_y  $\neq$  price_t` is UNKNOWN when either side is NULL, so it silently misses added and removed rows. The `v ... = NULL` arms are not optional.
- **Δ and $\ominus$ need union-compatible headings.** Same columns, same order; project both sides first if they differ.

Check it

- **Every key falls into exactly one bucket.** `unchanged + changed + added + removed` should account for every distinct key across both sides, with no overlap.
- Reverse the operands of Δ and confirm the answer is unchanged — symmetric difference is symmetric, which is what distinguishes it from $\ominus$.
- Put a row that is **identical** on both sides (A) and confirm it appears in neither the Δ nor the filtered reconciliation.

Related

- **None, never, missing** — a one-directional difference is *added* or *removed* alone.
- **Which rows, and which columns** — the null tests here are ordinary σ .
- Reference pages ([docs/reference](#)): [symmetric-difference](#), [difference](#), [full-outer-join](#), [null test](#).

7. Recipes: graphs

Recipe	For
Things that belong together	Undirected groups — duplicates, households, rings
Reachable, how far, and by what route	Directed reachability, distance, and the optimal route
When the rule is recursive	General recursion when the two-column form will not do

Things that belong together

Grain: one row per member, labelled with its group · **Class:** Graph · **Signals:** connected, linked, same as, belong together, duplicates, rings, households · **Operators:** $\bowtie$, $\cup$, CLUSTER

The problem

"Our customer table has duplicates. Two records are the same customer if they share an email address or a phone number. Which records belong together?"

How to recognise it

The question says **same as**, **linked**, **belong together**, **duplicates**, **rings** or **households**, and the rule for "together" is about **pairs**: two records are together if *something* connects them. What makes it a graph problem rather than a join is the chain: if A is linked to B and B to C, then A and C are together even though nothing connects them directly. A join finds the links; only a traversal finds the groups.

This is a two-step recipe, and it is the same two steps whatever the domain:

1. **Links.** Write the rule for "these two are connected" as a join, and keep one row per connected pair.
2. **Groups.** Hand the pairs to CLUSTER, which follows every chain and labels each member with its group.

The data

```
Customers := [
| id | name       | email       | phone       |
|----|-----|-----|-----|
| 1 | Ann Lee   | ann@x.com  | 555-0101 |
| 2 | A. Lee   | ann@x.com  | 555-0199 |
| 3 | Annie Lee | annie@y.com | 555-0199 |
| 4 | Bob Ray  | bob@z.com  | 555-0300 |
| 5 | Robert Ray | rray@z.com | 555-0300 |
| 6 | Cy Tan   | cy@w.com   | 555-0400 |
];
```

Records 1 and 2 share an email, 2 and 3 share a phone, 4 and 5 share a phone, and 6 shares nothing.

Step 1: the links

Join the table to a renamed copy of itself. `Customers.id < Other.id` keeps each pair once, and stops a record linking to itself.

```

Other := { p Other (Customers) };

Links := {
  π Customers.id → a, Other.id → b (
    Customers ⋈ Customers.id < Other.id
      ∧ (Customers.email = Other.email ∨ Customers.phone = Other.phone)
    Other)
};
query { Links };

```

a	b
1	2
2	3
4	5

(3 rows)

Step 2: the groups

`CLUSTER` only sees records that appear in a link, so record 6 would disappear. Linking every record to itself keeps the ones that match nothing as groups of one.

```

SelfLinks := { π id → a, id → b (Customers) };

Entities := { CLUSTER a, b AS entity_id (Links ∪ SelfLinks) };
query { Entities };

```

a	entity_id
1	1
2	1
3	1
4	2
5	2
6	3

(6 rows)

Records 1 and 3 are in the same entity although they share neither an email nor a phone. That is the chain through record 2, and it is the reason to use `CLUSTER` rather than the join alone.

Using the answer

Put the names back, and count how many records each entity has:

```

Grouped := {
  τ entity_id, id (
    π entity_id, id, name (Entities ⋈ Entities.a = Customers.id Customers))
};
query { Grouped };

```

entity_id	id	name
1	1	Ann Lee
1	2	A. Lee
1	3	Annie Lee
2	4	Bob Ray
2	5	Robert Ray
3	6	Cy Tan

(6 rows)

```
EntitySizes := { γ entity_id, COUNT(a) → records (Entities) };
query { σ records > 1 (EntitySizes) };
```

entity_id	records
1	3
2	2

(2 rows)

Pitfalls

- **Chains merge more than you expect.** One shared value is enough to join two groups. A family sharing a landline, or a placeholder such as n/a or 000-0000 in the phone column, links everyone who has it into one giant group. Remove junk values before building the links: $\sigma \text{ phone} \neq \text{"000-0000"} \text{ (Customers)}$. Look at the size of the largest group first; it is where this goes wrong.
- **NULL links nothing.** $\text{NULL} = \text{NULL}$ is UNKNOWN, so two records with no email are not linked by it. That is almost always what you want.
- **Records outside every link vanish** unless you add the self-links in step 2.
- **The group label is a label, not an identity.** Groups are numbered by their smallest member, so adding a record can renumber every group after it. Do not store `entity_id` as a key; recompute it, or key an entity by its smallest member's `id`.
- **CLUSTER ignores direction.** "A reports to B" is a direction; for "who is above whom" use CLOSURE, not CLUSTER.
- **A pair of records is evidence, not proof.** Two records sharing a phone may be two people. Keep the rule that made each link visible (for example as a `reason` column in Links) so that a reviewer can see why a group formed.

Check it

- The grain is one row per record: $\text{COUNT}(a)$ over `Entities` should equal the number of customers.
- Put a chain in your test data (A-B-C with no A-C link) and check A and C land together. Put a lone record in and check it survives.
- Check the largest group by hand before trusting the rest.

Related

- **Every: related to all of a set**
- **Reachable, how far, and by what route** — reachable from (CLOSURE), shortest route (PATH)
- Reference pages (docs/reference): CLUSTER, theta join, rename

Reachable, how far, and by what route

Grain: one row per reachable pair · **Class:** Graph · **Signals:** reachable, can get to, connected to, within N hops, shortest route, cheapest route, via · **Operators:** CLOSURE, PATH, TRACE

The problem

"From each airport, where can I get to with any number of connections? Which are within two hops? And what is the cheapest itinerary between two cities, and which airports does it pass through?"

How to recognise it

The question is about following a **directed edge** repeatedly — *reachable, can get to, within N hops, the route via, the cheapest way*. What makes it a graph problem rather than a join is the chain: a single join finds the direct edges, but *reachable* means any number of hops. Three operators answer three sharpenings of the question, and the tell is **how much you need to know about the connection**:

- **Just whether you can get there** — **CLOSURE** (every reachable pair, any distance).
- **Within a hop budget, and how far** — **PATH** (a hop window, stamping each pair with its shortest distance).
- **The best weighted route, and its stops** — **TRACE** (the cheapest/highest route as an ordered node sequence).

For *undirected* "who belongs together", see [Things that belong together](#) (**CLUSTER**); this recipe is directed.

The data

Direct flights with a fare. **Edges** is the two-column projection the closure operators want.

```
Flights := [
| origin | dest | cost |
|-----|-----|-----|
| JFK    | ORD  | 200  |
| ORD    | LAX  | 150  |
| JFK    | LAX  | 500  |
| ORD    | DEN  | 100  |
| DEN    | LAX  | 80   |
];
```

```
Edges := {  $\pi$  origin, dest (Flights) };
```

Recipe 1: everything reachable (CLOSURE)

CLOSURE from, to follows the edge to a fixpoint and returns every pair connected by one or more hops — the direct edges plus every derived one.

```
query {  $\tau$  origin, dest (CLOSURE origin, dest (Edges)) };
```

```
origin  dest
-----  ---
DEN     LAX
JFK     DEN
JFK     LAX
JFK     ORD
ORD     DEN
ORD     LAX
(6 rows)
```

"Can I get from JFK to anywhere?" is then a plain σ on the result — and the optimiser folds that σ **into** the closure as a single-source search, so scoping to one origin is an ordinary selection, not new syntax:

```
query { σ origin = "JFK" (CLOSURE origin, dest (Edges)) };
```

origin	dest
JFK	ORD
JFK	LAX
JFK	DEN

(3 rows)

RCLOSURE additionally pairs each node with itself, for when "stay put" is a valid route.

Recipe 2: within a hop budget, with distance (PATH)

When the question caps the distance — *within two hops* — and wants to know **how far**, **P**ATH takes a hop window and stamps each pair with its shortest distance:

```
query { τ hops, dest (σ origin = "JFK" (PATH origin, dest HOPS 1 TO 2 AS hops (Edges))) };
```

origin	dest	hops
JFK	LAX	1
JFK	ORD	1
JFK	DEN	2

(3 rows)

The distance is the **shortest** path length, so (from, to) is a candidate key — a pair never appears at two depths. As with **C**LOSURE, a σ on an endpoint seeds the traversal at that node.

Recipe 3: the optimal route, and its stops (TRACE)

Reachability says *whether*; **T**RACE says *how, and how much*. Give it a weight column and **M**INIMIZE/**M**AXIMIZE, and it returns the best route for every reachable pair as an ordered array of nodes:

```
query { τ origin, dest (TRACE origin, dest VIA cost MINIMIZE AS route (Flights)) };
```

origin	dest	cost	route
DEN	LAX	80	[DEN, LAX]
JFK	DEN	300	[JFK, ORD, DEN]
JFK	LAX	350	[JFK, ORD, LAX]
JFK	ORD	200	[JFK, ORD]
ORD	DEN	100	[ORD, DEN]
ORD	LAX	150	[ORD, LAX]

(6 rows)

TRACE uses the whole **F**lights relation (it needs the weight), not the two-column **E**des. Explode the route into rows with μ route (...) when you want one stop per row.

Variations

- **Undirected traversal** — union the reversed edges into the input first, or use the ↔ separator (CLOSURE from ↔ to, PATH from ↔ to), which reads each edge both ways.
- **Weighted reachability without the route** — --provenance --semiring tropical gives shortest-path costs through **C**LOSURE without materialising the paths.
- **A rule the two-column form cannot express** — extra columns, a filter or a join each round — is **general recursion** (**F**IX).

Pitfalls

- **CLOSURE and PATH are two-column, and directed.** Project to exactly the edge columns first (Edges above), and remember the edge direction — `from → to` is not the same graph as `to → from`. Use `↔` for undirected.
- **CLOSURE has no distance; PATH has no route; TRACE has both cost and route.** Reaching for CLOSURE when you needed the hop count, or PATH when you needed the actual stops, is the common miss — pick by how much you need to know.
- **Scope with a σ , not by hand.** The start node is deliberately not part of the syntax: `o origin = c` above the operator is folded into a single-source search. A predicate on the *distance* or an inequality stays as a residual σ (correct, just not pushed).
- **Dense graphs are large, and these block.** All-pairs reachability can be quadratic; the operators buffer and never push down, and `--max-fixpoint-rounds` guards a runaway.
- **TRACE weights should be whole units for an exact total** — scale money to cents, since floating-point sums can differ in their last bits.

Check it

- Put a **cycle** in the edges and confirm CLOSURE still terminates (a derived pair is never re-added) — closure over cyclic data is exactly where a hand-rolled join loops forever.
- Put a pair reachable **two ways** (JFK→LAX direct and via ORD) and confirm CLOSURE lists it once, PATH reports the *shortest* distance, and TRACE keeps the *cheapest* route.
- Check a node just outside the hop window is excluded by PATH and included by CLOSURE.

Related

- **Things that belong together** — undirected connectivity with CLUSTER, the other half of the graph class.
- **When the rule is recursive** — FIX, the general form these specialise.
- **As columns, as a list, as a tree** — TREE folds the same parent edges into nested documents; `μ` explodes a TRACE route.
- Reference pages (`docs/reference`): `closure`, `path`, `trace`.

When the rule is recursive

Grain: one row per derived pair · **Class:** Graph · **Signals:** ancestors, explosion, all descendants, parts of parts, keep expanding, until nothing new · **Operators:** FIX

The problem

"Our bill of materials stores 'assembly contains part' as direct edges. Give me every component of a bike at any depth — the full parts explosion."

How to recognise it

The question needs a rule applied **to its own output, repeatedly** — *all descendants, parts of parts, ancestors, keep expanding until nothing new appears*. It is the same "follow the chain" shape as **reachability**, but reach for **FIX** when the two-column **CLOSURE** cannot say it: the step needs **extra columns**, a **filter**, or a **join against another relation** each round. **FIX** is the general recursion operator — SQL's **WITH RECURSIVE**, one Datalog rule.

You give it a **base** (where to start) and a **step** (how to derive more from what you have so far); it repeats the step until the answer stops growing.

The data

```
Contains := [  
| assembly | part |  
|-----|-----|  
| Bike     | Frame |  
| Bike     | Wheel |  
| Wheel    | Rim   |  
| Wheel    | Spoke |  
| Wheel    | Tyre  |  
| Tyre     | Tube  |  
];
```

A bike contains a wheel, a wheel contains a tyre, a tyre contains a tube — three levels deep.

Recipe: base and step to a fixpoint (FIX)

FIX name (base, step) binds name inside the step to *everything derived so far*. The step joins what you have to the next edge and re-labels the result to stay union-compatible with the base:

```
Explosion := { FIX BOM (  
  Contains,  
  π assembly, sub → part (BOM ⋈ ρ Edge(part, sub) (Contains))  
) };  
query { τ assembly, part (Explosion) };
```

assembly	part
Bike	Frame
Bike	Rim
Bike	Spoke
Bike	Tube
Bike	Tyre
Bike	Wheel
Tyre	Tube
Wheel	Rim
Wheel	Spoke
Wheel	Tube
Wheel	Tyre

(11 rows)

Read the step: renaming **Contains** to **Edge(part, sub)** makes its first column share the name **part**, so **BOM ⋈ Edge** links each known part to what *it* contains; the projection relabels the deeper **sub** back to **part**. "*Everything to order for a bike*" is then a selection — folded into the recursion by the optimiser, since **assembly** is carried through unchanged:

```
query { π part (σ assembly = "Bike" (Explosion)) };
```

```
part
-----
Frame
Wheel
Rim
Spoke
Tyre
Tube
(6 rows)
```

Variations

- **Two-column reachability is CLOSURE.** This exact query is plain reachability, so CLOSURE assembly, `part (Contains)` is shorter and faster — FIX earns its keep only when the step needs the extra columns, filter, or join. See [reachability](#).
- **Carry a value down** — accumulate a depth, a running cost, or a path. That is a step that *computes*, which is exactly what FIX can do and CLOSURE cannot (see the termination pitfall).
- **Bound a recursion you are unsure of** — `--max-fixpoint-rounds` caps the depth, `--max-materialized-rows` caps the memory.

Pitfalls

- **The step must be monotone and linear.** Allowed: $\sigma \pi \rho \delta \mu$ and the joins, $\cup$, n . Not allowed: difference, division, aggregation, outer joins — and exactly **one** mention of the bound name. These are what guarantee a least fixpoint.
- **A step that carries values through terminates; one that computes may not.** The bill of materials finishes because every value it emits already appears in the input — there are finitely many rows to derive. A step like $\pi \ n + 1 \rightarrow m \ (N)$ invents a new value every round and never stops. That is not a bug to avoid — counting, accumulating cost, numbering rounds are all worth doing — but bound them, because whether such a query finishes cannot be decided without running it.
- **Cyclic data is fine.** Going round a cycle re-derives a row already held, so the round adds nothing and the loop closes — under set semantics. It is *computed values*, not cycles, that run away.
- **Reach for CLOSURE/PATH/TRACE first.** For binary reachability they are simpler and use a faster adjacency loop; FIX is the general, heavier form.

Check it

- Put a **three-level** nesting (Bike → Wheel → Tyre → Tube) in and confirm the deepest part reaches the top assembly — one level would pass a plain join too, so depth is the test that proves the recursion.
- Put a **cycle** in the data and confirm the query still terminates.
- If you wrote a step that computes, run it on tiny data **with a round cap** first, and reason about why it stops before removing the cap.

Related

- [Reachable, how far, and by what route](#) — CLOSURE/PATH/TRACE, the specialised two-column forms FIX generalises.

- **As columns, as a list, as a tree** — TREE folds the same parent edges into nested documents.
- **How many, how much, per what** — an aggregation is *not* allowed inside a FIX step; do it after.
- Reference pages (docs/reference): fix, closure.

8. Recipes: time

Recipe	For
As of, during, overlapping	The value as of a moment; overlapping periods
In a row, per visit, gaps	Sessions, gaps, and each row's neighbour in order

As of, during, overlapping

Grain: one row per event (as-of), or one per overlapping pair (interval) · **Class:** Temporal alignment · **Signals:** as of, at the time, in effect, during, while, overlapping, conflicts with · **Operators:** ASOF, IJOIN

The problem

"What exchange rate was in effect at the moment of each order? And which room reservations overlap a maintenance window?"

How to recognise it

The question aligns rows **in time** rather than on an exact key. Two shapes, and the tell is whether each side is a *point* or a *period*:

- **A point matched to the nearest point in time** — *as of, at the time, in effect, the last known* The timestamps rarely line up exactly, so an equi-join fails and you want the **nearest** row. That is an **AS-OF join** (ASOF).
- **A period overlapping a period** — *during, while, overlapping, conflicts with, back-to-back.* Each row carries an interval [start, end) and you match by a geometric relationship. That is an **interval join** (IJOIN).

Both need real temporal values. An inline table holds text, so convert the columns with `to_timestamp` first — text sorts chronologically for ISO-8601, but only a real `TIMESTAMP` has a *distance* for a tolerance to measure.

Recipe 1: the value in effect at a moment (ASOF)

Each order should carry the exchange rate that was current *at or before* it. An AS-OF join snaps each left row to the nearest right row in the direction the inequality names — `>=` is backward (*at or before*, the "as of" default).

```
RatesRaw := [
| at                               | rate |
|-----|-----|
| 2026-01-02T09:00:00Z | 0.80 |
| 2026-01-02T12:00:00Z | 0.82 |
];

OrdersRaw := [
| at                               | usd |
|-----|-----|
| 2026-01-02T10:30:00Z | 100 |
| 2026-01-02T13:00:00Z | 200 |
| 2026-01-02T08:00:00Z | 50  |
];
```

```
Rates := { π to_timestamp(at) → at, rate (RatesRaw) };
Orders := { π to_timestamp(at) → at, usd (OrdersRaw) };
```

```
query { Orders ASOF Orders.at >= Rates.at Rates };
```

at	usd	at_r	rate
2026-01-02T10:30:00Z	100	2026-01-02T09:00:00Z	0.8
2026-01-02T13:00:00Z	200	2026-01-02T12:00:00Z	0.82
2026-01-02T08:00:00Z	50	NULL	NULL

(3 rows)

By default an order with no prior rate keeps NULLs (left-outer). The 08:00 order has no rate before it. Add `INNER` to drop such rows, and now the conversion pays off:

```
query { π at, usd, rate, usd * rate → gbp (Orders ASOF INNER Orders.at >= Rates.at Rates) };
```

at	usd	rate	gbp
2026-01-02T10:30:00Z	100	0.8	80
2026-01-02T13:00:00Z	200	0.82	164

(2 rows)

`WITHIN DURATION 'PT30M'` adds a staleness bound — reject a match older than the tolerance — and `<=` flips the direction to *the next* rate at or after the order.

Recipe 2: periods that overlap (IJOIN)

Which reservations clash with a maintenance window? Each side is an interval, and "clash" is *any shared time* — Allen's `INTERSECTS`. An interval join names the endpoints of each side and applies the geometric predicate for you:

```
MaintRaw := [
| room | start                | finish                |
|-----|-----|-----|
| 101   | 2026-03-04T00:00:00Z | 2026-03-06T00:00:00Z |
];

ResRaw := [
| res | room | checkin                | checkout                |
|-----|-----|-----|-----|
| R1   | 101   | 2026-03-05T14:00:00Z | 2026-03-08T11:00:00Z |
| R2   | 101   | 2026-03-06T14:00:00Z | 2026-03-09T11:00:00Z |
| R3   | 102   | 2026-03-05T14:00:00Z | 2026-03-08T11:00:00Z |
];

Maint := { π room, to_timestamp(start) → start, to_timestamp(finish) → finish (MaintRaw) };
Res := { π res, room, to_timestamp(checkin) → checkin, to_timestamp(checkout) → checkout (ResRaw) };

query {
  σ room = room_r (
    Res IJOIN INTERSECTS (Res.checkin, Res.checkout, Maint.start, Maint.finish) Maint)
};
```

res	room	checkin	checkout	room_r	start	finish
R1	101	2026-03-05T14:00:00Z	2026-03-08T11:00:00Z	101	2026-03-04T00:00:00Z	2026-03-06T00:00:00Z

(1 row)

The interval join matches on **endpoints only**, so the same-room condition is a `σ` on top — and the right side's `room` arrives disambiguated as `room_r`.

Variations

- **The thirteen Allen relations** name every way two intervals can relate: `OVERLAPS`, `DURING`, `CONTAINS`, `MEETS` (touching, no gap and no overlap), `PRECEDES` (a gap between), `EQUALS`, and their converses. `INTERSECTS` is the convenience superset for *any* shared time.
- `TIES(FIRST|LAST)` on an AS-OF join decides which of two same-instant right rows to take.
- **The gap between events** — how long until the next one — is a Sequence question (`WINDOW LEAD`), a cousin of this class.

Pitfalls

- **Text is not a timestamp.** ISO-8601 text *orders* correctly, so an AS-OF without a tolerance may look right, but it has no *distance*: a `WITHIN` over text is rejected, and any duration arithmetic is wrong. Convert with `to_timestamp` first.
- **Intervals are half-open** [`start`, `end`). A reservation ending exactly when a window starts does **not** overlap it — that is `MEETS`, not `INTERSECTS`. Whether a clean handoff counts as a conflict is a business rule; naming the relation puts it in the script rather than hiding it in a comparison.
- **AS-OF is left-outer by default.** An event with no prior row keeps NULLs, not no row. Use `INNER` when an unmatched event should disappear, and remember a downstream arithmetic on the NULL side yields NULL.
- **IJOIN matches on endpoints only.** A partition key (same room, same sensor) is a `σ` on the result, and it matters for performance too — filter to the partition before the join on large data.
- **The inequality direction is the match direction.** `>=` is backward (as-of), `<=` is forward (the next one). Getting it backwards silently returns the wrong neighbour.

Check it

- Put an event **before any** right-side row (the 08:00 order) and confirm the default keeps it with NULLs and `INNER` drops it.
- Put a boundary case — an interval that **touches** but does not overlap — and confirm `INTERSECTS` excludes it while `MEETS` finds it.
- On tiny data, work out the nearest match by hand and check `at_r` names the row you expected.

Related

- **In a row, per visit, gaps** — the gap to the *next* event is `WINDOW LEAD`, over one relation rather than two.
- **How many, how much, per what** — `DOWNSAMPLE` buckets one relation by time; this class aligns two.
- Reference pages (`docs/reference`): `asof-join`, `interval-join`, `to_timestamp`, `duration literal`.

In a row, per visit, gaps

Grain: one row per event (labelled), or one per session · **Class:** Sequence · **Signals:** in a row, per visit, consecutive, gaps, since last, streak, the next one · **Operators:** SESSIONIZE, WINDOW LAG/LEAD

The problem

"Group these page views into visits that end after 30 minutes idle, and count each visit. And by how much did each daily meter reading change from the one before?"

How to recognise it

The question is about a row's **position in an ordered stream** and its **neighbours** — in a row, consecutive, per visit, gaps between, since last time, streak, the next one. It is not about a key you can group on directly; the grouping *emerges* from the order. Two operators cover it:

- **Segment a stream where there is an idle gap** — clicks into visits, pings into bursts. That is SESSIONIZE, which walks the order and hands each row a session id.
- **Compare a row with its neighbour** — the change since the previous row, the gap to the next. That is an offset window, WINDOW LAG (look back) or LEAD (look forward).

Both add a column and keep every row, so they compose with the rest of the query.

Recipe 1: split a stream into sessions (SESSIONIZE)

A *visit* is a run of events with no long gap. SESSIONIZE orders each partition, and starts a new session whenever the gap to the previous row exceeds the threshold.

```
EventsRaw := [
  | user | ts
  |-----|-----|
  | 1 | 2026-01-01T10:00:00Z |
  | 1 | 2026-01-01T10:20:00Z |
  | 1 | 2026-01-01T11:30:00Z |
  | 2 | 2026-01-01T09:00:00Z |
];

Events := { π user, to_timestamp(ts) → ts (EventsRaw) };
Sessions := { SESSIONIZE ts GAP DURATION 'PT30M' PER user AS session (Events) };
query { Sessions };
```

user	ts	session
1	2026-01-01T10:00:00Z	1
1	2026-01-01T10:20:00Z	1
1	2026-01-01T11:30:00Z	2
2	2026-01-01T09:00:00Z	1

(4 rows)

The session id is only useful once you aggregate over it — *how many visits, how long each* is an ordinary γ keyed on the emergent (user, session) grain:

```
query { γ user, session, COUNT(ts) → events, MIN(ts) → started, MAX(ts) → ended (Sessions) };
```

user	session	events	started	ended
1	1	2	2026-01-01T10:00:00Z	2026-01-01T10:20:00Z
1	2	1	2026-01-01T11:30:00Z	2026-01-01T11:30:00Z

```

      2      1      1  2026-01-01T09:00:00Z  2026-01-01T09:00:00Z
(3 rows)

```

Recipe 2: compare a row with its neighbour (LAG/LEAD)

Change since last looks **back** one row: `LAG(reading)` brings the previous row's value alongside the current one, so a subtraction gives the delta. The first row of a partition has no predecessor, so its `LAG` is `NULL`.

```

Meter := [
| day          | reading |
|-----|-----|
| 2026-01-01 | 100    |
| 2026-01-02 | 130    |
| 2026-01-03 | 125    |
];

```

```

WithPrev := { WINDOW LAG(reading) SORT day ASC AS prev (Meter) };
query { π day, reading, reading - prev → change (WithPrev) };

```

day	reading	change
2026-01-01	100	NULL
2026-01-02	130	30
2026-01-03	125	-5

(3 rows)

`LEAD` is the mirror — it looks **forward** to the next row, for *time until the next event or the value that follows*:

```

query { WINDOW LEAD(reading) SORT day ASC AS next_reading (Meter) };

```

day	reading	next_reading
2026-01-01	100	130
2026-01-02	130	125
2026-01-03	125	NULL

(3 rows)

Variations

- **Your own session boundary.** `SESSIONIZE`'s rule is a fixed *gap > threshold*. For a boundary on something else ("a new session when the page changes"), compute the neighbour with `LAG` and build the running id yourself.
- `LAG(expr, n, default)` looks back `n` rows and substitutes `default` for the missing edge, so `LAG(reading, 1, reading)` reports a first-row change of zero instead of `NULL`.
- **Runs of a repeated value** (a streak) is a sessionization on a derived order — a gap-and-island cousin.

Pitfalls

- **Order must be a total order.** `SESSIONIZE` and the offset windows both depend on the `SORT`; two rows with equal keys can fall either way, changing which is "previous". Add a tie-breaker when the neighbour must be well-defined.
- **The gap is strict, and edges are NULL.** A gap *exactly* equal to the threshold stays in the same session; the first row of each partition has no `LAG` and the last has no `LEAD`, and arithmetic on that `NULL` is `NULL`. Decide what the edge should say.

- **PER resets everything.** Both operators restart per partition — session ids restart at 1, and LAG is NULL at each partition's first row, not just the global first. Forgetting PER bleeds one entity's stream into the next.
- **These are blocking.** SESSIONIZE and the windows buffer and sort each partition; over a provably-unbounded input they are rejected.

Check it

- Put a gap **exactly** at the threshold and one just over it, and confirm only the second opens a new session.
- Check the session grain: SUM of each session's events equals the total row count per user.
- Put a single-row partition (user 2) in and confirm its LAG/LEAD are NULL and its session is 1.

Related

- **As of, during, overlapping** — aligning two relations in time, where this class walks *one*.
- **How many, how much, per what** — the per-session counts are an ordinary γ ; ROLLING is the windowed aggregate cousin.
- Reference pages (docs/reference): sessionize, window-offset.

9. Recipes: choosing and solving

Recipe	For
The best combination within limits	The best subset under a budget
What value makes this true	Filling the one value that satisfies an equation
Enough cases to cover	A covering test suite over a parameter space

The best combination within limits

Grain: one row per chosen item · **Class:** Optimization · **Signals:** the best, the most/least within, under a budget, maximise, minimise, subject to, without exceeding · **Operators:** OPTIMIZE

The problem

"We have 50 person-days this quarter and a backlog of features, each with a business value and a build cost. Which subset delivers the most value without going over budget?"

How to recognise it

The question asks for the **best combination** subject to **limits** — *maximise value under a budget, the cheapest set that covers everything, pick the most within a cap*. The tell is that no single row is the answer and no sort finds it: the value of a choice depends on the *other* choices, because they compete for the same limited resource. That is optimisation, and OPTIMIZE states it directly — an objective to maximise or minimise, and one or more SUBJECT TO constraints.

This is a knapsack, and it is exactly the shape SQL cannot express: *the most valuable subset whose costs sum to at most 50* is not a filter, a rank, or an aggregate.

The data

```
Backlog := [
| feature | value | cost |
|-----|-----|-----|
| SearchRevamp | 60 | 10 |
| MobileSync | 100 | 20 |
| Dashboards | 120 | 30 |
| DarkMode | 40 | 5 |
];
```

Recipe: maximise a total under a budget (OPTIMIZE)

State the objective and the constraint; OPTIMIZE returns the **chosen input rows**, schema unchanged — it is a filter that picks the optimal subset.

```
query { OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(cost) <= 50 (Backlog) };
```

feature	value	cost
SearchRevamp	60	10
Dashboards	120	30
DarkMode	40	5

(3 rows)

SUM(value) is what to maximise; SUM(cost) <= 50 is the limit. The solver evaluates every feasible subset — you describe *what optimal means*, not how to search.

A separate problem per group is PER. Give each team its own budget in one statement:

```
TeamBacklog := [
| team      | feature      | value | cost |
|-----|-----|-----|-----|
| search    | SearchRevamp | 60    | 10   |
| search    | Dashboards   | 120   | 30   |
| mobile    | MobileSync   | 100   | 20   |
| mobile    | DarkMode     | 40    | 5    |
];
```

```
query { OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(cost) <= 30 PER team (TeamBacklog) };
```

team	feature	value	cost
search	Dashboards	120	30
mobile	MobileSync	100	20
mobile	DarkMode	40	5

(3 rows)

Variations

- **Minimise instead** — MINIMIZE SUM(cost) SUBJECT TO SUM(coverage) >= 10 is the set-cover shape: the cheapest set that meets a requirement.
- **Several constraints** — chain them with AND: SUBJECT TO SUM(cost) <= 50 AND SUM(risk) <= 3.
- **Continuous allocation** rather than a yes/no subset — OPTIMIZE ALLOCATE (0.0, 1.0) ... -> weight assigns each row a fractional weight (a portfolio), returning every row with an allocation column.

Pitfalls

- **A constraint is required.** OPTIMIZE MAXIMIZE SUM(value) with no SUBJECT TO is an error — unbounded "maximise" has no answer. There is always a limit; name it.
- **Ties are the solver's choice.** When two subsets reach the same optimum, which one comes back is not specified. If you need a particular winner, break the tie in the objective (MAXIMIZE SUM(value) - SUM(cost)).
- **An infeasible group yields no rows, silently.** If a group's constraints cannot be met, it is logged and skipped rather than failing the query — so a missing group is not necessarily a bug in your data. Check for groups you expected but did not get.
- **OPTIMIZE blocks and never pushes down.** It buffers its input and runs the solver in-engine; it needs the mathematical-programming solver installed (ojAlgo ships with Relix).
- **Objective and constraints are linear SUM forms.** A non-linear goal is out of scope for this operator.

Check it

- Work the tiny case by hand: enumerate the feasible subsets of four items and confirm the returned one is genuinely the best. Optimisation answers are easy to *believe* and hard to *eyeball*, so a known-answer case matters most here.
- Check the constraint holds: `SUM(cost)` over the returned rows must be within the budget.
- Make a group **infeasible** (every item over budget) and confirm it drops out rather than erroring.

Related

- **What value makes this true** — `SOLVE` inverts one equation; `OPTIMIZE` searches a space of choices.
- **How many, how much, per what** — a plain `MAX/MIN` finds an extreme value; `OPTIMIZE` finds the best *combination* under constraints.
- **The most, the top N, the latest** — `TOP` ranks independent rows; optimisation is for when the rows compete.
- Reference pages (docs/reference): `optimize`, `solve`.

What value makes this true

Grain: one row per input row · **Class:** Equation · **Signals:** fill in the blank, complete, what value, given ... find, goal-seek, back out · **Operators:** `SOLVE`

The problem

"An invoice import has gaps: some rows know the quantity and unit price but not the line total; others know the total and one of the two factors. The relationship is always `line_total = qty × unit_price`. Fill in whichever value is missing on each row."

How to recognise it

The question gives a **fixed relationship** and asks for the **one value that makes it hold** — *fill in the blank, complete the row, given these, find that, back out the rate*. It is a spreadsheet goal-seek: rearrange one equation to solve for its one unknown. The tell is that *which* value is unknown may vary from row to row, and the relationship stays the same.

`SOLVE` is this and only this. It is **not** `OPTIMIZE` — there is no search, no choice, no constraint; it inverts one arithmetic equation per row. (It does not even use the mathematical-programming solver — it is a deterministic tree-walk, and streams.)

The data

A blank cell is NULL.

```
Invoice := [
| item | qty | unit_price | line_total |
|-----|-----|-----|-----|
| Widget | 3 | 5.00 | 15.00 |
| Gadget |  | 12.00 | 60.00 |
| Sprocket | 4 |  | 10.00 |
```

```
];
```

Each row is missing a different one of the three participating columns.

Recipe: fill the one blank (SOLVE)

Write the equation; SOLVE decides *per row* which value is missing and rearranges to find it.

```
query { SOLVE line_total = qty * unit_price (Invoice) };
```

item	qty	unit_price	line_total
Widget	3	5	15
Gadget	5	12	60
Sprocket	4	2.5	10

(3 rows)

Widget multiplies; Gadget and Sprocket divide. The direction is chosen per row from whichever column is NULL — one statement fills a different hole in each.

Variations

- **Any invertible arithmetic** — `SOLVE area = width * height`, `SOLVE total = principal + interest`. The equation may use `+` `-` `*` `÷` and unary minus, each column appearing at most once per side.
- **Compute a column when nothing is missing** is not this recipe — that is a plain π with arithmetic ($\pi \dots$, `qty * unit_price → line_total`). `SOLVE` earns its place only when the unknown varies.

Pitfalls

- **Exactly one blank, or nothing happens.** A row with **no** NULL among the participating columns, or with **two or more**, passes through unchanged — `SOLVE` acts only when there is precisely one hole. It does not *check* a fully-populated row, and it cannot solve for two unknowns.

```
Check := [
| item      | qty | unit_price | line_total |
|-----|-----|-----|-----|
| Full      | 2   | 4.00       | 8.00       |
| TwoBlank  | 3   |            |            |
];
query { SOLVE line_total = qty * unit_price (Check) };
```

item	qty	unit_price	line_total
Full	2	4	8
TwoBlank	3	NULL	NULL

(2 rows)

- **It does not validate.** A `Full` row whose values are inconsistent (`2 × 4 ≠ 9`) is *not* corrected or flagged — there is no blank to fill. Checking a relationship is a σ (`line_total ≠ qty * unit_price`), a different job.
- **Only invertible arithmetic.** A column appearing twice, or a non-arithmetic function, cannot be inverted deterministically and is rejected.
- **Division by zero, or a NULL factor that should have been the unknown,** gives NULL — check the inputs the inversion divides by.

Check it

- For each row, plug the filled value back into the equation by hand on tiny data — a goal-seek is only right if the completed row satisfies the relationship.
- Put a **fully-populated** row and a **two-blank** row in and confirm both pass through untouched — that is the boundary of what SOLVE does.
- If you also need to *catch* rows that violate the relationship, add a σ ; SOLVE fills gaps, it does not audit.

Related

- **The best combination within limits** — OPTIMIZE chooses values under constraints; SOLVE inverts a single fixed equation.
- **Which rows, and which columns** — computing a column when nothing is missing is a π ; validating a relationship is a σ .
- Reference pages (docs/reference): solve, optimize.

Enough cases to cover

Grain: one row per generated case · **Class:** Generation · **Signals:** enough cases, all combinations, every pair, test matrix, cover, generate then filter · **Operators:** COVER, $\times$ (generate)

The problem

"We test across three browsers, three operating systems and two plans. The full matrix is 18 combinations, and Safari only runs on macOS. Give me the smallest test suite that still exercises every pair of factor values — and prove it is complete."

How to recognise it

The question is about **producing** rows rather than filtering existing ones — *enough cases, every combination, all pairs, a test matrix, generate and then narrow*. The tell is a combinatorial space that explodes and a budget that cannot afford all of it. Two moves make the class:

- **Generate the candidate space**, then constrain it — a cross product ($\times$) proposes every combination, and a σ removes the invalid ones. *Generator proposes, selection disposes*.
- **Thin it to just enough** — COVER t keeps a small subset in which every combination of t values still appears at least once. $t = 2$ is all-pairs, the usual sweet spot.

The data

Three factors — the full space is $3 \times 3 \times 2 = 18$.

```
Browser := [  
| browser |  
|-----|  
| Chrome  |  
| Firefox |  
| Safari  |
```

```
];

OS := [
| os      |
|-----|
| Windows|
| macOS  |
| Linux   |
];

Plan := [
| plan |
|-----|
| Free |
| Pro  |
];
```

Recipe 1: generate, then constrain ($\times$ and σ)

The cross product is the generator: it proposes every combination. A σ then removes the ones that cannot happen — Safari runs only on macOS:

```
Full := { Browser  $\times$  OS  $\times$  Plan };
Valid := {  $\sigma$  browser  $\neq$  "Safari"  $\vee$  os = "macOS" (Full) };
query {  $\gamma$  COUNT(*)  $\rightarrow$  valid_combos (Valid) };
```

```
valid_combos
-----
14
(1 row)
```

Recipe 2: thin to a covering suite (COVER)

COVER 2 keeps a subset in which **every pair** of factor values still appears together at least once. Crucially, it derives the pairs it must cover from the *constrained* input — so the invalid combinations are never demanded, and the constraint comes for free:

```
Suite := { COVER 2 (Valid) };
query {  $\tau$  browser, os (Suite) };
```

browser	os	plan
Chrome	Linux	Free
Chrome	Windows	Free
Chrome	macOS	Pro
Firefox	Linux	Pro
Firefox	Windows	Pro
Firefox	macOS	Free
Safari	macOS	Free
Safari	macOS	Pro

(8 rows)

Eight cases instead of the full fourteen, yet every browser-os, browser-plan and os-plan pair is present somewhere. The τ only sorts for reading; COVER emits in selection order.

Prove it is complete, in-language. Coverage is verifiable with a difference — every pair in the valid space minus every pair in the suite is empty when the design is complete, so no external oracle is needed:

```
query { ( $\pi$  browser, os (Valid)) - ( $\pi$  browser, os (Suite)) };
```

browser	os

(0 rows)

```
query { σ browser = "Safari" (Suite) };
```

browser	os	plan
Safari	macOS	Free
Safari	macOS	Pro

(2 rows)

Variations

- **COVER 1** keeps every value once (the cheapest); **COVER t** at $t = \text{the number of columns}$ is every distinct row (equivalent to δ). t interpolates between them.
- **Bias which candidates win** — order the input first: **COVER 2** (τ priority DESC (Valid)), since the greedy search breaks ties toward earlier rows.
- **An unbounded generator** (a numeric range, the naturals) must be bounded before a blocking operator like **COVER**, γ or δ consumes it — a σ or TOP below it. A cross product of finite tables, as here, is already bounded.

Pitfalls

- **Strength is required and is not a filter.** **COVER** (...) with no t is an error. **COVER 2** does not *reduce* the semantics — it selects rows; it never invents a combination that was not in the input.
- **Greedy, not minimal.** **COVER** finds a near-minimal suite, not a provably minimal one — it may be a row or two above the theoretical optimum. That is the right trade-off for the cost of an exact solver.
- **Constrain before you cover, not after.** Filtering the *suite* after **COVER** can break coverage. Put the σ below **COVER** so the coverage universe is the valid space — then the constraint is respected by construction.
- **A blocking generator over an unbounded input is rejected.** Bound it first.

Check it

- **Verify coverage in-language** with the difference above — do not trust the row count alone. An empty difference for every factor pair is the proof.
- Add a **constraint** and confirm the excluded combinations appear in neither the valid space nor the suite (no Safari on Windows).
- Compare the suite size to the full matrix — a covering suite should be markedly smaller, or **COVER** is not earning its place.

Related

- **Which rows, and which columns** — the σ that constrains the generated space is ordinary selection.
- **What changed, what differs** — the coverage proof is a set difference.
- Reference pages (docs/reference): **cover**, **cross**, **distinct**, **difference**.

10. Recipes: reshaping

Recipe	For
As columns, as a list, as a tree	Pivoting, nesting, unnesting, folding to a tree

As columns, as a list, as a tree

Grain: changes on purpose — per group (nest/pivot), per element (unnest), per root (tree) ·

Class: Reshaping · **Signals:** as columns, as a list, gather, flatten, as a tree, wide, nested ·

Operators: PIVOT/UNPIVOT, COLLECT, μ , TREE

The problem

"Turn the long quarterly sales table into one column per quarter. Gather each customer's order ids into a list. Flatten a stored list back into rows. And fold an org chart's parent pointers into a nested tree."

How to recognise it

The question is about **shape**, not selection or summary — *as columns, as a list, flatten, as a tree, wide instead of long*. The tell is that the **grain changes on purpose**: reshaping deliberately turns one row-per-thing into a different one-row-per-something. Which operator depends on the target shape:

- **Rows → columns** (long to wide) — PIVOT (and UNPIVOT back).
- **Many rows → one list** — COLLECT, inside a γ . Its inverse, **list → many rows**, is μ (unnest).
- **Parent pointers → a nested tree** — TREE, the recursive generalisation of COLLECT.

Recipe 1: rows to columns (PIVOT)

PIVOT value BY key turns each distinct value of the key column into an output column. A group with no row for a key gets NULL there.

```
QuarterlySales := [
| region | quarter | revenue |
|-----|-----|-----|
| East   | q1       | 100     |
| East   | q2       | 120     |
| West   | q1       | 80      |
];

query { PIVOT revenue BY quarter PER region (QuarterlySales) };

region  q1  q2
-----
East    100 120
West    80  NULL
(2 rows)
```

The column headers come from the **data**, so the output schema is *open* — downstream operators that need named columns may need a p. UNPIVOT is the reverse, folding listed columns back into rows.

Recipe 2: many rows to a list (COLLECT), and back (μ)

`COLLECT`, inside a `γ`, gathers a group's values into an array instead of reducing them to one number:

```
Orders := [
| customer | order_id |
|-----|-----|
| Ann      | 1         |
| Ann      | 2         |
| Bo       | 3         |
];

Gathered := { γ customer, COLLECT(order_id) → order_ids (Orders) };
query { Gathered };
```

customer	order_ids
Ann	[1, 2]
Bo	[3]

(2 rows)

`μ` (unnest) is the exact inverse — it explodes an array into one row per element, and `WITH ORDINALITY` records each element's position:

```
query { μ order_ids WITH ORDINALITY pos (Gathered) };
```

customer	order_ids	pos
Ann	1	1
Ann	2	2
Bo	3	1

(3 rows)

`μ` is also how you flatten a source that already stores an array (a JSON field of line items, say) into flat rows.

Recipe 3: parent pointers to a tree (TREE)

An adjacency table — each row pointing at its parent — folds into a forest of nested documents in one pass. `TREE key BY parentKey` follows the edge to any depth; a row whose parent is absent (or `NULL`) is a root.

```
Employees := [
| id | manager_id | name |
|----|-----|-----|
| 1  | 0          | Ada  |
| 2  | 1          | Bob  |
| 3  | 1          | Cara |
| 4  | 2          | Dan  |
];

query { TREE id BY manager_id ORDER id ASC AS reports (Employees) };
```

id	manager_id	name	reports
1	0	Ada	[{id: 2, manager_id: 1, name:...

(1 row)

One row per root, each carrying its whole subtree in the added column — the recursive version of `COLLECT`, which nests a single level. From there it is ordinary nested data: `μ`

`reports` walks one level, `π reports` reads the column.

Variations

- **Aggregate before pivoting.** If a group has several rows for one key, `PIVOT` picks one non-deterministically — `γ-sum first (PIVOT total BY key PER g (γ ...))` when you want the total.
- **COLLECT of a struct** — `COLLECT({order_id, amount})` bundles whole objects, not just one field.
- **TREE for any hierarchy** — org charts, bills of materials, threaded comments, a query's own plan (`TREE(relix.plan)`).

Pitfalls

- **PIVOT's schema is open and data-dependent.** The columns and their order come from the runtime key values, so a downstream query cannot assume a name exists; rename or test defensively.
- **COLLECT keeps NULLs; the other aggregates skip them.** `COLLECT(bonus)` yields `[10, NULL]`, one element per row — matching `array_agg`. If you did not want the NULLs, filter them first.
- **A terminal elides nested values.** A table view shows an array or struct truncated at the column width; use `--format json` (or project into the nesting) to see a `COLLECT` or `TREE` result whole.
- **TREE needs a well-formed forest.** A duplicate key or a cycle is a runtime error, not a silently-dropped row — which is a feature (it refuses to loop forever), but clean the adjacency first.
- **Order within a nested/exploded result is not guaranteed** unless you ask: `ORDER` on `TREE`, a `τ` below `COLLECT`, and remember `μ`'s ordinal restarts per input row.

Check it

- **The grain changed on purpose — state the new one.** After `COLLECT` it is one row per group; after `μ` one row per element; after `TREE` one row per root. Count against that.
- `COLLECT` then `μ` should **round-trip** back to the original rows (the optimiser even collapses the pair). Try it on tiny data.
- Put a group with a **missing** pivot cell (West's q2) and confirm it is NULL, not absent.
- Give `TREE` a **second root** and confirm you get two output rows — it is a forest, not a single tree.

Related

- **How many, how much, per what** — `γ` with a scalar aggregate collapses a group to a number; `COLLECT` keeps the list.
- **Connected, reachable, via** — `CLOSURE/PATH` traverse the same parent edges `TREE` folds, but produce reachability, not nested structure.
- Reference pages (`docs/reference`): `pivot`, `unpivot`, `collect`, `unnest`, `tree`.

11. Recipes: explaining

Recipe	For
Why is this row here, where did it come from	Reifying a result's lineage and reading it

Why is this row here, where did it come from

Grain: one row per result (after WHY), then one per contributing source tuple (after μ) · **Class:** Explanation · **Signals:** why, where from, which inputs, how derived, trace back, provenance, justify · **Operators:** WHY (ω), μ

The problem

"This result row looks wrong — which source rows produced it, and how did they combine? I want to trace a surprising result back to the exact inputs behind it."

How to recognise it

The question is about a **result you already have**, not a new one — *why is this here, where did this come from, which inputs does it depend on, justify this number*. It is a debugging and audit question, and the answer is the row's **lineage**: the source tuples that produced it and the way they combined. WHY reifies that lineage as an ordinary nested column, so the rest of the algebra can slice it like any other data.

The tell that you want WHY and not a rerun: you are not trying to change the result, you are trying to *account for* it.

The data

```
Orders := [
| oid | cid | region | amount |
|-----|-----|-----|-----|
| 100 | 7  | west  | 40    |
| 101 | 7  | west  | 25    |
| 102 | 9  | east  | 10    |
];

Customers := [
| cid | name |
|-----|-----|
| 7   | Ada  |
| 9   | Grace |
];
```

Recipe: reify lineage, then read it (WHY, μ)

WHY (R) emits every result row of R unchanged, plus one provenance column recording which source rows produced it. Wrap the query whose results you want to explain:

```
Aug := { WHY ( $\pi$  region, amount (Orders  $\bowtie$  Orders.cid = Customers.cid Customers)) };
query { Aug };
```

region	amount	provenance
west	40	[{coefficient: 1, variables: ...
west	25	[{coefficient: 1, variables: ...

```
east      10  [{coefficient: 1, variables: ...
(3 rows)
```

provenance is ordinary nested data — an array of *derivations* (the ways the row could arise), each an array of the source tuples that combined. So the ordinary operators read it: `μ` explodes a derivation array, and dotted paths read its fields. To name the exact source rows behind every *west* result, flatten one level to the derivations, lift the variables out, flatten again, and select:

```
PerDeriv := { μ provenance (Aug) };
Lifted   := { π region, amount, provenance.variables → variables (PerDeriv) };
PerVar   := { μ variables (Lifted) };
query {
  π region, amount, variables.relation → source, variables.ordinal → ordinal
  (σ region = "west" (PerVar))
};
```

region	amount	source	ordinal
west	40	Customers	1
west	40	Orders	1
west	25	Customers	1
west	25	Orders	2

(4 rows)

Each result names the two tuples that combined to produce it, because a join is a *joint* derivation — one derivation, several variables. A result reachable two ways (a union of a duplicated value) would instead show two derivations. Nothing in that query is special to provenance: `π`, `μ` and `σ` do to provenance exactly what they do to any nested column, which is the whole point of reifying lineage as data.

Variations

- **The side-channel** — `--provenance --semiring lineage` reports the same lineage as a one-shot dump rather than a queryable column, and also offers the cheap semirings (reachability, path count, shortest/cheapest path).
- `:why` **in the REPL** is sugar for query `{ WHY(...) }`.
- **Plan, not data** — to explain *how the engine ran* a query rather than where a row came from, read `relix.plan` (EXPLAIN-as-data), the reflective sibling of `WHY`.

Pitfalls

- **Lineage stops at the aggregation wall.** A `γ`, `τ`, `δ`, difference, or outer join below `WHY` is read as an opaque base relation — its output rows become a fresh source, and lineage is not threaded *through* the grouping. Deep-through-aggregation lineage is not available; put `WHY` below the aggregation if you need the inputs to it.
- **WHY is a hard optimiser barrier and blocks.** No rewrite crosses it (that is deliberate — a merged duplicate or an inlined view would change the reified lineage), it buffers, and it never pushes down. Use it to explain, not in a hot path.
- **A terminal elides the nested column.** `provenance` is a nested array; read it with `--format json` or take it apart with `μ/π` — a table view truncates it.
- **μ takes a column, not a path.** To explode `provenance.variables`, lift it into a column with `π ... → variables` first, then `μ variables`.

- `provenance` **is a reserved output name**. A clash with an existing input column is a validation error.

Check it

- Trace a row you **already understand** first — confirm `WHY` names the source tuples you expect — before trusting it on the surprising one.
- Count the derivations: a row produced by a single join path has **one** derivation with several variables; a row reachable two ways has **several** derivations. The exploded form is the one to count over.
- If a row's lineage looks impossibly shallow, check for an **aggregation** below `WHY` — the wall is the usual reason.

Related

- [What changed, what differs](#) and every other recipe — `WHY` is the tool the *Check it* step reaches for when a result is surprising.
- [As columns, as a list, as a tree](#) — reading `provenance` uses the same `μ/path` navigation as any nested column.
- Reference pages ([docs/reference](#)): `why`, `unnest`, `provenance`.

12. Recipes: sampling

Recipe	For
A random, representative subset	A reproducible random sample

A random, representative subset

Grain: one row per sampled row · **Class:** Sampling · **Signals:** a random sample, a representative subset, *N at random*, a preview, reproducible sample · **Operators:** `SAMPLE ...`
`ROWS`, `SAMPLE p`

The problem

"Give me 3 random events for a test fixture — the same 3 every time so the test is stable. Or roughly 40% of the rows as a quick preview. And let me sample within one region."

How to recognise it

The question asks for a **random** or **representative** subset, not a filtered or ranked one — a random sample, *N at random*, a representative subset, a preview, a spot check. The tell is that **which** rows come back is not determined by their values: any row is as eligible as any other. That rules out `σ` (which picks by a condition) and `TOP` (which picks by rank).

Two forms, and the tell is whether you want an **exact count** or a **proportion**:

- **Exactly N rows** — `SAMPLE n ROWS` (reservoir sampling): returns exactly `n` (or all rows if there are fewer).
- **A percentage** — `SAMPLE p` (Bernoulli): each row is kept with probability `p`, so the count varies around `p × |input|`.

`SEED` makes either one **reproducible** — the same seed and input always pick the same rows — which is what turns a random sample into a stable test fixture.

The data

```
Events := [  
| id | region |  
|----|-----|  
| 1 | EMEA |  
| 2 | EMEA |  
| 3 | APAC |  
| 4 | EMEA |  
| 5 | APAC |  
| 6 | EMEA |  
| 7 | APAC |  
| 8 | EMEA |  
| 9 | APAC |  
| 10 | EMEA |  
];
```

Recipe 1: exactly N random rows (`SAMPLE n ROWS`)

`SAMPLE n ROWS` returns exactly `n` rows drawn uniformly. Pin the draw with `SEED` so a fixture or benchmark replays the same rows:

```
query { SAMPLE 3 ROWS SEED 7 (Events) };
```

id	region
4	EMEA
2	EMEA
3	APAC

(3 rows)

Ask for more than the relation holds and you get all of it, not an error:

```
query { SAMPLE 50 ROWS SEED 7 (Events) };
```

id	region
1	EMEA
2	EMEA
3	APAC
4	EMEA
5	APAC
6	EMEA
7	APAC
8	EMEA
9	APAC
10	EMEA

(10 rows)

Recipe 2: a percentage (SAMPLE p)

`SAMPLE p` keeps each row with probability `p` — a streaming sample whose size varies. Use it for a rough preview of a large input where the exact count does not matter:

```
query { SAMPLE 0.4 SEED 42 (Events) };
```

id	region
3	APAC
4	EMEA
7	APAC
8	EMEA

(4 rows)

Recipe 3: sample within a stratum

To sample *within* a subset — a stratum — filter first, then sample the result. The sample is drawn from just that subset:

```
query { SAMPLE 2 ROWS SEED 1 (σ region = "APAC" (Events)) };
```

id	region
7	APAC
9	APAC

(2 rows)

For a sample of each stratum, run this per region (or union the per-region samples).

Variations

- **Bernoulli reproducibly** — `SAMPLE p SEED k` is the streaming, percentage cousin of the seeded reservoir.

- **A deterministic prefix**, not a random sample, is λ (LIMIT / TOP n) — the first n rows in input order, not a uniform draw.
- **Without a seed**, both forms are non-deterministic — the rows differ run to run, which is what you want for a genuine spot check but not for a fixture.

Pitfalls

- **A sample is not a filter and not a prefix.** SAMPLE picks at random; σ picks by a condition; λ /TOP picks a deterministic slice. Using TOP 100 as a "sample" biases toward whatever the input order favours.
- **No seed means no reproducibility.** For a test fixture, a benchmark, or anything you need to replay, always SEED it — otherwise the "sample" changes under you.
- **The reservoir count is a cap, not a promise of exactly N.** SAMPLE n ROWS returns all rows when the input has fewer than n ; it never pads.
- SAMPLE p **gives a proportion, not a count.** If you need *exactly* k rows, use the ROWS form — the Bernoulli form's size varies around $p \times |\text{input}|$.
- SAMPLE **buffers/streams and does not push down**, so the draw happens in-engine over what the source returns.

Check it

- Run the seeded query **twice** and confirm the rows are identical — that is the reproducibility guarantee, and the reason to seed a fixture.
- Ask for **more rows than exist** and confirm you get all of them, not an error.
- For SAMPLE p , run it over a larger input and confirm the count sits *around* $p \times |\text{input}|$, not exactly at it — that is the difference from the ROWS form.

Related

- **Which rows, and which columns** — σ picks by a condition; sampling picks at random.
- **The most, the top N, the latest** — TOP is a *deterministic* slice by rank, the thing a sample is often mistaken for.
- Reference pages (docs/reference): sample-reservoir, sample-bernoulli, limit.

13. Recipes: iteration

Recipe	For
Iterate until it settles	Converging, or simulating rounds, replacing state each time

Iterate until it settles

Grain: one row per thing that settles (a page, a cell) · **Class:** Iteration · **Signals:** converge, iterate until stable, simulate rounds, settle, numeric fixpoint, generations · **Operators:** `ITERATE`

The problem

"Rank these pages by a score that is refined until it stops moving. Or run a cellular automaton forward a few generations, where cells are born and die each round."

How to recognise it

The question describes a computation that **settles** rather than **accumulates** — *converge, until stable, simulate N rounds, until the numbers stop moving, generations*. Each round starts from what the last round produced and **replaces** it; the answer is the final round.

This is the tell that separates it from **recursion** (`FIX`): `FIX` only ever *adds* rows and terminates on its own when nothing new appears. `ITERATE` *replaces* the whole state each round, so rows can come and go — which is exactly what a score being refined, or a cell dying, needs, and what `FIX` cannot express.

`ITERATE name (base, step)` binds `name` to the previous round's whole output; the step's output replaces it. Say when to stop:

- `ROUNDS n` — apply the step exactly `n` times.
- `UNTIL STABLE MAX n ROUNDS` — stop when a round changes nothing.
- `UNTIL c, ... WITHIN  $\epsilon$  PER k, ... MAX n ROUNDS` — stop when no value in columns `c` moved by more than ϵ , pairing rows across rounds by key `k`.

The two `UNTIL` forms **require** a round cap and do not return a result they did not reach.

Recipe 1: converge to a fixpoint (UNTIL ... WITHIN)

PageRank refines every page's score by passing it along outgoing links, round after round, until the scores stop moving. A page's rank is the share of a random surfer's time spent on it.

```
Links := [
| src | dst |
|-----|-----|
| A   | B   |
| A   | C   |
| B   | C   |
| C   | A   |
| D   | C   |
];
```

```
Pages := {  $\delta$  ( $\pi$  src  $\rightarrow$  page (Links)  $\cup$   $\pi$  dst  $\rightarrow$  page (Links)) };
```

```

OutDegree := { γ src, COUNT(*) → out (Links) };
Weighted  := { Links ⋈ OutDegree };

-- 0.0375 is the jump share: (1 - 0.85) / 4 pages
Rank := { ITERATE R (
  π page, 0.25 → rank (Pages),
  π page, 0.0375 + 0.85 * Coalesce(passed, 0) → rank (
    Pages ⋈ Pages.page = In.dst ρ In(dst, passed) (
      γ dst, SUM(rank / out) → passed (ρ From(src, rank) (R) ⋈ Weighted)))
) UNTIL rank WITHIN 0.0001 PER page MAX 100 ROUNDS };
query { τ rank DESC (Rank) };

```

page	rank
C	0.394199878685
A	0.37249131329
B	0.19580880811
D	0.0375

(4 rows)

The step reads the previous round's (page, rank) as R, passes each page's rank along its links (rank / out per link), sums what arrives at each destination, and keeps a page nothing links to via the outer join. UNTIL rank WITHIN 0.0001 PER page stops when no page's rank moved by more than the tolerance between rounds.

A scalar numeric fixpoint — Newton's method for a square root, $x \rightarrow (x + n/x) / 2$ — is the same shape with one row and one UNTIL column.

Recipe 2: simulate rounds (a state machine)

Conway's Life is the canonical state machine: each generation *replaces* the last, and a cell's fate depends on its live neighbours. ROUNDS 4 runs the glider forward four generations.

```

Offsets := [
| dx | dy |
|----|----|
| -1 | -1 |
| -1 | 0  |
| -1 | 1  |
| 0  | -1 |
| 0  | 1  |
| 1  | -1 |
| 1  | 0  |
| 1  | 1  |
];

Glider := [
| x | y |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 0 | 2 |
| 1 | 2 |
| 2 | 2 |
];

Later := { ITERATE Board (
  Glider,
  π x, y (σ n = 3 (ρ C(x, y, n) (γ nx, ny, COUNT(*) → n (π x + dx → nx, y + dy → ny (Board × Offsets))))))
  u π x, y (σ n = 2 (ρ C(x, y, n) (γ nx, ny, COUNT(*) → n (π x + dx → nx, y + dy → ny (Board × Offsets)))))) ⋈ Board)
} ROUNDS 4 };
query { τ x, y (Later) };

```

x	y
—	—
1	3

```
2 1
2 3
3 2
3 3
(5 rows)
```

The step counts live neighbours (`Board × Offsets`, grouped), keeps cells with exactly three (a birth) and cells with exactly two that were already alive (survival). Rows are born and die each round — the whole board is replaced — which is why this is `ITERATE`, not `FIX`. Swapping `ROUNDS 4` for `UNTIL STABLE MAX 50 ROUNDS` would settle a still life, report a blinker as a period-2 cycle, and report the glider as not settling (it moves forever).

Variations

- **The three stop clauses** answer three questions: *run N rounds* (`ROUNDS`), *has it stopped changing?* (`UNTIL STABLE`), *have the numbers converged?* (`UNTIL ... WITHIN`).
- **A rule written once** — the step may call a relation-valued function, so `ITERATE G (Start, generation(G)) ROUNDS 5` applies a def-ined rule each round.
- **Label propagation, smoothing, other scores** are Recipe 1's shape with a different step.

Pitfalls

- `ITERATE` **replaces**; `FIX` **accumulates**. If the answer only ever grows (reachability, a parts explosion), that is **recursion** (`FIX`), which needs no round cap. Reach for `ITERATE` when state is *replaced* each round.
- **An `UNTIL` test needs a round cap and may not be reached**. The `UNTIL` forms require `MAX n ROUNDS` and report two different misses: a **cycle** (a round reproduced an earlier one — it can never settle, and the error names the period) or the **cap reached** without converging. Neither returns a partial result.
- **The step must be deterministic**. A step calling `Rand()`, `NOW()`, an unseeded sample, or a user function is rejected — a round that could answer differently makes a convergence test meaningless.
- `UNTIL ... WITHIN` **needs declared `NUMBER` columns and a key that identifies one row per round**. Two rows sharing a `PER` key in a round is an error; the base's schema must be declared (not schema-on-read).
- **A selection is not pushed into an `ITERATE`**. Filtering PageRank to one page would change the other pages' values, so the optimiser leaves a σ above it.

Check it

- Give an `UNTIL ... WITHIN` iteration a **tiny** input whose fixpoint you can compute by hand, and confirm it converges to it — and that the ranks (or shares) **sum** to what they should.
- Feed a known **cycle** (a Life blinker) to `UNTIL STABLE` and confirm it reports the period rather than looping.
- Confirm `ROUNDS 0` returns the base unchanged, and that a state-machine step actually *loses* rows some round (proving it replaces rather than accumulates).

Related

- **When the rule is recursive** — `FIX`, the *accumulating* sibling; the Pitfalls above are mostly about telling the two apart.
- **How many, how much, per what** — each round's step is ordinary algebra, γ and joins included (unlike a `FIX` step).
- Reference pages (`docs/reference`): `iterate`, `fix`.

14. Case studies

Problems that cross several classes, each classified out loud before any code.

Case study	Classes it crosses
Fraud rings	Existence + graph + explanation
Family history	Graph + time
Inventory allocation	Summary + optimization
SLA breaches	Temporal alignment + sequence
Reconciling two systems	Comparison + federation
Test planning	Generation

Fraud rings

Existence + Graph + Explanation

The problem

"Money is being cycled between accounts to launder it. Find the rings — groups of accounts that pass money round in a loop — flag every account within a few hops of a known-bad one, and, for anything we escalate, show exactly which transfers made it suspicious."

Classifying it

Three questions, three classes — say them out loud before writing anything:

- "accounts that pass money round in a loop" — a **graph** question. A ring is a cycle: an account reachable from itself. **CLOSURE**, then keep the self-pairs.
- "within a few hops of a known-bad account" — still graph, but **bounded** and distance-aware: **PATH** with a hop window, scoped to the flagged account.
- "show which transfers made it suspicious" — an **explanation** question: **WHY** reifies the lineage of a flagged result.

The data

```
Accounts := [  
| account | holder | flagged |  
|-----|-----|-----|  
| 1001    | Ada   | false |  
| 1002    | Bo    | false |  
| 1003    | Cy    | true  |  
| 1004    | Dee   | false |  
| 2001    | Eve   | false |  
| 2002    | Fin   | false |  
];
```

```
Transfers := [  
| src | dst | amount |  
|-----|-----|-----|  
| 1001 | 1002 | 500    |  
| 1002 | 1003 | 480    |  
| 1003 | 1001 | 460    |  
| 1001 | 1004 | 200    |  
| 2001 | 2002 | 300    |  
];
```

```
Edges := {  $\pi$  src, dst (Transfers) };
```

1001 $\rightarrow$ 1002 $\rightarrow$ 1003 $\rightarrow$ 1001 is a ring; 1004 is paid out of it; 2001 $\rightarrow$ 2002 is an ordinary one-way transfer.

Stage 1: the rings (graph — a cycle)

An account is in a ring when it can reach **itself**. Compute reachability, then keep the self-pairs:

```
Reach      := { CLOSURE src, dst (Edges) };  
RingMembers := {  $\delta$  ( $\pi$  src  $\rightarrow$  account ( $\sigma$  src = dst (Reach))) };  
query { RingMembers };
```

```
account  
-----  
  1001  
  1002  
  1003  
(3 rows)
```

Stage 2: proximity to a flagged account (graph — bounded)

Which accounts are within three transfer hops of the flagged 1003? A σ on the endpoint scopes the traversal, and PATH stamps each with its distance:

```
query {  
   $\tau$  gen, account (  
     $\pi$  dst  $\rightarrow$  account, depth  $\rightarrow$  gen (  
       $\sigma$  src = 1003 (PATH src, dst HOPS 1 TO 3 AS depth (Edges))))  
};
```

```
account  gen  
-----  
  1001    1  
  1002    2  
  1004    2  
  1003    3  
(4 rows)
```

Put the holders back with an ordinary **existence** semi-join to `RingMembers` (each account once):

```
query { π account, holder (Accounts ⋈ Accounts.account = RingMembers.account RingMembers) };
```

account	holder
1001	Ada
1002	Bo
1003	Cy

(3 rows)

Stage 3: why is this escalated (explanation)

For a transfer into a flagged account, `WHY` names the exact source rows behind it — the audit trail an escalation needs:

```
Flagged := { π account (σ flagged = true (Accounts)) };
query { WHY (π src, dst, amount (Transfers ⋈ Transfers.dst = Flagged.account Flagged)) };
```

src	dst	amount	provenance
1002	1003	480	[{coefficient: 1, variables: ...

(1 row)

What this shows

The three classes compose without any special machinery: the graph stage produces a relation of ring members, the existence stage filters accounts by it, and the explanation stage reifies lineage as a column the other operators can read. Each stage is named after what its rows *are* — `RingMembers`, `Flagged` — so the pipeline reads as the investigation does.

A caution the graph stage earns: one shared value merges a ring. A placeholder account, or a legitimate hub everyone transacts with, will pull unrelated accounts into one giant "ring" — look at the largest component first, exactly as the **connected-groups** recipe warns. And a ring is evidence, not proof; keep the `WHY` trail so a human decides.

Recipes drawn on

- **Reachable, how far, and by what route** — `CLOSURE`, `PATH`.
- **Has at least one** — the semi-join back to accounts.
- **Why is this row here** — `WHY`.
- **Things that belong together** — the merge caution.

Family history

Graph + Time

The problem

"From a family tree stored as parent links, list a person's ancestors, say how many generations back each one is, and show when each was born."

Classifying it

- "a person's ancestors ... how many generations back" — a **graph** question over a parent-pointer relation: reachability gives the ancestors, and a bounded traversal gives the generation distance.
- "when each was born" — a **time** attribute, joined back once the ancestors are known. (In a real tree the source is a GEDCOM file, read through the engine's **GEDCOM connector**; the shape below is the same, with the individuals and links as inline stand-ins.)

A parent-pointer tree is the same shape as the fraud graph and the bill of materials — this is "follow the edge to any depth", and the operator is **CLOSURE** (or **FIX** when the step needs more than two columns; see **recursion**).

The data

```
People := [  
| id | name | born |  
|----|-----|-----|  
| 1  | Ada  | 1900 |  
| 2  | Bob  | 1925 |  
| 3  | Cara | 1952 |  
| 4  | Dan  | 1978 |  
| 5  | Cy   | 1955 |  
];
```

```
ParentOf := [  
| child | parent |  
|-----|-----|  
| 2      | 1      |  
| 3      | 2      |  
| 4      | 3      |  
| 5      | 2      |  
];
```

Dan (4) descends from Cara (3), Bob (2), Ada (1); Cy (5) is Bob's other child — Dan's uncle.

Stage 1: the ancestors (graph — reachability)

CLOSURE child, parent follows the parent edge to any depth. Scope it to Dan:

```
Ancestors := { CLOSURE child, parent (ParentOf) };  
query {  $\pi$  parent  $\rightarrow$  ancestor ( $\sigma$  child = 4 (Ancestors)) };
```

```
ancestor  
-----  
      3  
      2  
      1  
(3 rows)
```

Stage 2: how many generations back (graph — distance)

PATH stamps each ancestor with its distance, so "grandparent" is generation 2:

```
query {  
   $\tau$  gen, ancestor (  
     $\pi$  parent  $\rightarrow$  ancestor, gen (  
       $\sigma$  child = 4 (PATH child, parent HOPS 1 TO 5 AS gen (ParentOf))))  
};
```

ancestor	gen
3	1
2	2
1	3

(3 rows)

Stage 3: with birth years (time)

Join the ancestor set back to `People` — an ordinary **existence** semi-join keeps each ancestor once and brings the temporal column along:

```
DanAncestors := { ⋈ (π parent → id (σ child = 4 (Ancestors))) };
query { τ born, name (People ⋈ People.id = DanAncestors.id DanAncestors) };
```

id	name	born
1	Ada	1900
2	Bob	1925
3	Cara	1952

(3 rows)

What this shows

The graph is the whole problem; time is a column you pick up at the end. Note that the sibling line (Cy) never appears — reachability up the parent edge is exactly *ancestors*, not *relatives*. For *descendants*, run the closure the other way (`CLOSURE parent, child`); for *cousins and the rest*, `CLUSTER` the undirected "related-to" graph, which is the **connected-groups** recipe.

Two edges from a real GEDCOM would change nothing here: the connector produces the same `People` and `ParentOf` relations, and dates arrive as real `DATE/TIMESTAMP` values, so "ancestors born before 1930" is a σ on the temporal column.

Recipes drawn on

- **Reachable, how far, and by what route** — `CLOSURE`, `PATH`.
- **When the rule is recursive** — `FIX`, when a step needs more than two columns.
- **Has at least one** — the join back for birth years.

Inventory allocation

Summary + Optimization

The problem

"Each region has more orders than it can ship this week. Show the demand per region, then choose the orders to fulfil that deliver the most value without exceeding each region's shipping capacity."

Classifying it

- "the demand per region" — a **summary**: one row per region, γ over the orders.
- "choose the orders ... most value ... without exceeding capacity" — an **optimization**: a per-region knapsack, `OPTIMIZE` with a capacity constraint.

The tell that the second half is optimization, not ranking: the orders **compete** for a shared capacity, so no per-order sort finds the answer — the value of shipping one order depends on which others you shipped.

The data

```
Orders := [
| order | region | value | units |
|-----|-----|-----|-----|
| 01    | North  | 100   | 30    |
| 02    | North  | 120   | 40    |
| 03    | North  | 60    | 20    |
| 04    | South  | 90    | 25    |
| 05    | South  | 50    | 15    |
];
```

Each region can ship **50 units** this week.

Stage 1: the demand (summary)

```
query { γ region, COUNT(*) → orders, SUM(units) → demanded, SUM(value) → value (Orders) };
```

region	orders	demanded	value
North	3	90	280
South	2	40	140

(2 rows)

Stage 2: what to ship (optimization)

OPTIMIZE picks the best subset per region under the capacity constraint — PER region solves an independent knapsack for each:

```
Fulfil := { OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(units) <= 50 PER region (Orders) };
query { τ region, order (Fulfil) };
```

order	region	value	units
01	North	100	30
03	North	60	20
04	South	90	25
05	South	50	15

(4 rows)

Stage 3: what it comes to (summary again)

The chosen set is an ordinary relation, so summarise it the same way:

```
query { γ region, SUM(value) → shipped_value, SUM(units) → shipped_units (Fulfil) };
```

region	shipped_value	shipped_units
North	160	50
South	140	40

(2 rows)

What this shows

Summary brackets the optimization on both sides: γ *frames* the problem (what is demanded, where the pressure is) and γ *reports* the answer (what got shipped). The optimiser sits in the middle doing the one thing a γ cannot — choosing under a constraint.

Note North's result is where ranking would have gone wrong: TOP 1 value PER region picks O2 (value 120), but O1 + O3 together deliver 160 within the same 50 units. Competing for a budget is the signature of optimization, and the reason **ranking** is the wrong class here.

South is infeasible-free — it fits. Watch for a region whose every order exceeds capacity: OPTIMIZE drops such a group silently rather than erroring, so a region missing from Fulfil is a result, not a bug.

Recipes drawn on

- **How many, how much, per what** — the framing and the reporting γ .
- **The best combination within limits** — OPTIMIZE ... PER.
- **The most, the top N, the latest** — the trap this problem is not.

SLA breaches

Temporal alignment + Sequence

The problem

"Support tickets have an event log — opened, resolved, and status changes in between. Which tickets breached the 8-hour resolution SLA, and how long did each one spend between status changes?"

Classifying it

- "how long from opened to resolved ... breached the SLA" — a **time** question: pair each ticket's open event with its resolve event and measure the span. Pairing two events of the same ticket is a self-join; measuring the span is temporal arithmetic (TIMESTAMP – TIMESTAMP → DURATION).
- "how long between status changes" — a **sequence** question: compare each event with the previous one in the ticket's ordered log, WINDOW LAG.

Both need real timestamps, so the log is converted with to_timestamp first.

The data

```
Events := [
| ticket | at                | status |
|-----|-----|-----|
| T1      | 2026-06-01T09:00:00Z | open   |
| T1      | 2026-06-01T11:00:00Z | resolved |
| T2      | 2026-06-01T09:00:00Z | open   |
| T2      | 2026-06-02T15:00:00Z | resolved |
| T3      | 2026-06-01T09:00:00Z | open   |
];

Typed := {  $\pi$  ticket, to_timestamp(at) → at, status (Events) };
```

T1 resolved in 2 hours, T2 took 30, T3 is still open.

Stage 1: time to resolve, and the breaches (time)

Split the log into the open and resolved events, join them per ticket, and subtract:

```
Opened  := {  $\pi$  ticket, at → opened  ( $\sigma$  status = "open"      (Typed)) };
Resolved := {  $\pi$  ticket, at → resolved ( $\sigma$  status = "resolved" (Typed)) };
```

```
Elapsed := { π ticket, resolved - opened → took (Opened ⋈ Resolved) };
query { Elapsed };
```

ticket	took
T1	PT2H
T2	PT30H

(2 rows)

The breaches are a σ on the duration — comparing DURATIONs directly:

```
query { σ took > DURATION 'PT8H' (Elapsed) };
```

ticket	took
T2	PT30H

(1 row)

Stage 2: time in each status (sequence)

How long did each event's status hold before the next change? is the gap to the previous row in the ticket's ordered log — WINDOW LAG:

```
query {
  π ticket, at, at - prev → since_prev (
    WINDOW LAG(at) SORT at ASC PER ticket AS prev (Typed))
};
```

ticket	at	since_prev
T1	2026-06-01T09:00:00Z	NULL
T1	2026-06-01T11:00:00Z	PT2H
T2	2026-06-01T09:00:00Z	NULL
T2	2026-06-02T15:00:00Z	PT30H
T3	2026-06-01T09:00:00Z	NULL

(5 rows)

What this shows

The join-and-subtract in Stage 1 is the general shape for "*duration between two related events*", and the WINDOW LAG in Stage 2 is "*duration between consecutive events*" — two different questions that both reduce to temporal subtraction, and the reason time is its own axis in this manual.

T3 is the interesting omission. The inner join in Stage 1 silently drops it, which is correct for "*time to resolve*" (it has none) but **wrong** for "*still open past SLA*" — that is a different question, about the absence of a resolve event more than 8 hours after the open. Answering it needs a reference "now" and an **absence** shape (an open with no matching resolve), and deciding it deliberately is exactly the *Check it* discipline: what does the answer say about an input with no closing row?

Recipes drawn on

- **As of, during, overlapping** — temporal arithmetic and DURATION comparison.
- **In a row, per visit, gaps** — WINDOW LAG.
- **None, never, missing** — the still-open-past-SLA question.

Reconciling two systems

Comparison + Federation

The problem

"Our ledger and the bank statement should agree. Show every reference where the two disagree — a different amount, or present in one and missing from the other."

Classifying it

- "where the two disagree ... missing from the other" — a **comparison**: align the two sources by key and keep the rows that differ. A full outer join keeps every reference from both sides, and a null test catches the missing ones.
- **Federation** is the difficulty axis, not a class of its own: the two sides are *different systems* — our database and the bank's export. In the engine that is two connections, and the reconciliation query is identical to the one below; the inline tables here stand in for the two live sources so the example runs.

The data

```
Ours := [
| ref | amount |
|-----|-----|
| A   | 100    |
| B   | 200    |
| C   | 300    |
];
```

```
Bank := [
| ref | amount |
|-----|-----|
| A   | 100    |
| B   | 250    |
| D   | 400    |
];
```

A agrees; B differs (200 vs 250); C is only in our ledger; D is only on the statement.

Stage 1: align by key (comparison)

Rename the amount on each side so both survive the join, then full-outer-join on the reference:

```
O      := { p (amount → ours) (Ours) };
Bk     := { p (amount → bank) (Bank) };
Recon := { O ⋈ O.ref = Bk.ref Bk };
```

The right key comes back as `ref_r`, and an unmatched side is NULL.

Stage 2: keep the discrepancies

The disagreements are the rows where the amounts differ **or** one side is NULL — and because a comparison with NULL is UNKNOWN, the NULL sides need an explicit null test:

```
query {
  π Nz(ref, ref_r) → ref, ours, bank (
    σ ours ≠ bank ∨ ours = NULL ∨ bank = NULL (Recon))
};
```

ref	ours	bank
B	200	250
C	300	NULL
D	NULL	400

(3 rows)

`Nz(ref, ref_r)` collapses the two key columns into one — for a ledger-only row the key is on the left, for a bank-only row on the right.

Federation: the same query across two systems

Nothing about Stage 1 or 2 changes when the two sides are live systems. Declare a connection for each and read the ledger tables through them:

```
connection erp    from postgres("...");
connection bankfeed from mysql("...");

O := { p (amount → ours) (erp.ledger) };
Bk := { p (amount → bank) (bankfeed.statement) };
```

The full outer join and the null tests are identical. What the planner does differ: each side's scan (and any σ/π above it) pushes down into its own database, and the join itself runs in the engine, because it spans two connections. `--explain` shows exactly which part ran where — the subject of [Part IV](#).

What this shows

Reconciliation is the comparison recipe with the difficulty axis turned up: the operators are the same (`⋈`, a null test, `Nz`), and *federation* only changes where the work runs, not what the query says. That separation — one query, many backends — is the point of a relational front end over a pile of systems.

The trap is the one comparison always carries: `ours ≠ bank` is UNKNOWN when either side is NULL, so it silently misses the added and removed references. The `v ... = NULL` arms are not optional — they are what turn a *difference* into a *reconciliation*.

Recipes drawn on

- **What changed, what differs** — `⋈`, the null tests, `Nz`.
- **None, never, missing** — a one-directional view (only-in-ours, only-in-bank).

Test planning

Generation

The problem

"We test payments across three payment methods, three currencies and two amount tiers. The full matrix is too many combinations to run, and bank transfers only work in USD. Give me the smallest suite that still exercises every pair of settings, and prove it covers them."

Classifying it

This is a single-class case study on purpose — it shows a class at full depth rather than crossing several. It is **generation**: *produce* a set of test cases rather than filter existing ones. The tells are "*the full matrix*" (a combinatorial space), "*too many to run*" (a budget), "*every pair*" (a coverage strength), and "*bank transfers only in USD*" (a constraint on the space). Two moves: a cross product generates the space and a σ constrains it, then **COVER** thins it to a pairwise-covering suite.

The data

```
Method := [
| method |
|-----|
| card   |
| paypal |
| bank   |
];

Currency := [
| currency |
|-----|
| USD      |
| EUR      |
| GBP      |
];

Tier := [
| tier |
|-----|
| small |
| large |
];
```

The full matrix is $3 \times 3 \times 2 = 18$.

Stage 1: generate, then constrain

The cross product proposes every combination; a σ removes the ones that cannot happen (bank is USD-only):

```
Valid := {  $\sigma$  method  $\neq$  "bank"  $\vee$  currency = "USD" (Method  $\times$  Currency  $\times$  Tier) };
query {  $\gamma$  COUNT(*)  $\rightarrow$  full (Method  $\times$  Currency  $\times$  Tier) };
query {  $\gamma$  COUNT(*)  $\rightarrow$  valid (Valid) };
```

```
full
----
  18
(1 row)
valid
----
  14
(1 row)
```

Stage 2: thin to a covering suite

COVER 2 keeps a subset in which every **pair** of settings still appears together at least once. It derives the pairs it must cover from the *constrained* set, so the impossible combinations are never demanded:

```
query {  $\gamma$  COUNT(*)  $\rightarrow$  chosen (COVER 2 (Valid)) };
```

```
chosen
-----
      8
(1 row)
```

Stage 3: prove it covers (comparison)

Coverage is not a matter of trust: every pair in the valid space minus every pair in the suite is empty when the design is complete. The proof is an ordinary set difference:

```
query { (π method, currency (Valid)) - (π method, currency (COVER 2 (Valid))) };

method  currency
-----
(0 rows)
```

What this shows

Generation is "*describe the space, then thin it*", and the thinning is honest because the proof is in the language itself — a difference of projected pairs, not an external oracle. The constraint is respected **by construction**: because the σ sits *below* COVER, the coverage universe is the valid space, so no thinning step can smuggle a bank-in-EUR case back in.

The one number to watch is Stage 2's chosen: a covering suite should be markedly smaller than the full matrix (8 vs 18 here). If it is not, either the space is already small or COVER is not earning its place — reach for it when the combinations explode, not when there are a dozen.

Recipes drawn on

- Enough cases to cover — COVER, the cross product, the coverage proof.
- What changed, what differs — the set difference that proves completeness.

15. Engineering the solution

Making a correct query run well, and moving a solution into a program.

Chapter	Covers
Engineering the solution	Reading a plan, what stops a pushdown, bounding a generator, budgets, working offline, moving into Java

Engineering the solution

The recipes and case studies get the *answer* right. This part is about making a correct query *run well* — reading what the engine decided, keeping the work where it belongs, and moving a finished solution into a program. None of it changes what a query means; it changes what it costs.

Reading a plan

A query says *what*; the plan says *how*. `--explain` (or the `relsx.plan` relation, the same information as rows) prints the rewrites the optimiser applied and the physical plan it will run. Read it before assuming anything about cost.

The most important thing a plan tells you over a database connection is **where each part runs**. This query, over a Postgres connection, folds entirely into one SQL statement:

```
τ region (σ status = "paid" ∧ amount > 100 (sales.orders))
```

```
PushedScan [jdbc/sales] SELECT "id", "amount", "status", "region" FROM "orders"  
WHERE (("status" = 'paid') AND ("amount" > 100)) ORDER BY ("region") ...
```

The selection and the sort became a `WHERE` and an `ORDER BY`; the database does the work and the engine receives only the rows it asked for. A `PushedScan` line is the plan telling you a whole sub-tree left the engine.

What stops a pushdown

A pushdown is not guaranteed, and the plan is where you find out it did not happen. The recurring reasons:

- **A hard barrier.** `WHY` reifies lineage and no rewrite crosses it, so nothing above it pushes. It appears in the plan as a bare `Why` with the work stacked in-engine beneath.
- **Cross-connection work.** A join whose two sides are *different* connections cannot push — no single database can see both — so each side's scan pushes and the join runs in the engine. That is federation, and it is the `reconciliation` case study's whole point.
- **The backend cannot express it.** A MongoDB source pushes σ , μ and cheap π/λ into an aggregation pipeline, but evaluates τ and γ in the engine over the rows that come back. A σ immediately above a window function stays in-engine, because SQL `WHERE` cannot reference a window column at the same level.
- **A collation mismatch.** A string comparison pushes only when the engine can trust the column compares as it does; on a case-insensitive database it is kept in-engine unless the connection declares otherwise.

When a plan shows work in the engine you expected in the database, it is one of these — not a mystery. The `pushdown` reference page is the exhaustive account.

Bounding a generator

A generator relation can be endless — the naturals, a range with no top. An operator that must see all its input before producing any output — γ , δ , a sort, `COVER`, `ITERATE`, a cumulative `ROLLING` — cannot run over an endless input, and the engine refuses it at plan time rather than looping forever.

The fix is to put the bound **below** the blocking operator: a σ that the generator can satisfy, or a `λ/TOP` that caps the stream. *Generator proposes, selection disposes* — and the selection has to come first. A `TOP n` over a generator is the rescue case the planner recognises: it makes the stream finite, so a blocking operator above it is safe again.

Materialization budgets

Some operators must hold rows in memory: a join's build side, a spool shared between two readers, everything a `FIX` or `ITERATE` has derived so far. The engine shares sub-expressions and spills work it can recompute for free, so most plans stay lean without help. When one does not — a recursion that runs away, a product that explodes — two caps turn a hang into a clear error:

- `--max-fixpoint-rounds` bounds how many rounds one `FIX/ITERATE` may iterate. Pick it from how *deep* you expect to go — a hierarchy's levels, a graph's diameter — not from the size of the answer.
- `--max-materialized-rows` bounds how many rows one operator may hold. Pick it when you do *not* know the shape of a terminating answer, since it bounds memory rather than depth.

Both are off unless asked for; setting both on a query you are unsure of is reasonable.

Working offline

Analysing a query against a database connection normally introspects the live catalog for each table's columns. That needs the database reachable, which is the wrong dependency for writing and checking a query at your desk or in CI. Two ways to cut it:

- **Declare the schema.** A binding-form source states a table's columns inline (`source Orders from shop { table: "orders", schema: { id: NUMBER, amount: NUMBER } }`), so analysis, optimisation and plan inspection all run with nothing connected. The query contacts the database only when it actually executes.
- **Snapshot the catalog.** Capture the schema once and analyse against the snapshot, so a whole script type-checks and plans offline and reconnects only to run.

Either way, *reading a plan, checking a pushdown, and catching a wrong column name* are things you do without touching the database — which is what makes them cheap enough to do on every change.

Moving a solution into Java

A `.rel` script is one front end; a program is another. When a solution graduates from "a query I run" to "a step inside an application", it moves onto the **embedding API** — the same engine, reached from Java, described in the programming guide.

The shape carries over directly. Building a query is the algebra as method calls, one per operator; `optimized()` returns the rewritten form to inspect; `render()`, `explain()` and `schema()` are terminals that execute nothing; `stream()` / `toList()` run it. A value that varies — an order id, a threshold — is bound by building the query with it (`select(eq(attr("id"), num(orderId)), rel("Orders"))`), not by pasting it into text. What was a `.rel` script and a CLI flag becomes a `Rel` session and a method call, and the query you verified against the engine is the query the program runs.

The programming guide is where that surface is taught, and its Java examples compile and run as part of the build — so, like this manual's `rel` blocks, they are real.

16. When Relix is not the tool

Where to stop — the problems it does not do, and the ones it does that you might think it does not.

Chapter	Covers
When Relix is not the tool	Row-at-a-time thinking forced into sets, recursion where an operator would do, and what is out of range

When Relix is not the tool

The most useful thing a manual can do is say where it stops. Two kinds of problem belong here: the ones where you are holding Relix the wrong way round, and the ones that are genuinely outside what it does. Recognising either saves more time than any recipe.

Row-at-a-time thinking, forced into sets

The commonest way to misuse a relational engine is to write a loop in it. *"For each order, look at the previous order and compute the gap"* is a sentence about one row at a time, and translated literally it becomes a pile of self-joins keyed on *"the row before this one"* — awkward to write, quadratic to run, and wrong at the edges.

Every such question has a set-shaped answer that is shorter and faster:

- *"compare each row with its neighbour"* is a window (`WINDOW LAG/LEAD`), not a self-join — see [sequence](#).
- *"a running total"* is `ROLLING`, not a triangular self-join — see [summary](#).
- *"the top N per group"* is `TOP ... PER`, not a correlated count — see [ranking](#).

The tell that you have slipped into row-at-a-time thinking is a query that mentions *the previous, the next, so far, or running* and answers it with a join back to the same table. Stop and name the set operation instead: the grain does not change, so it is a window; the answer is one row per group, so it is a γ . Thinking in relations is the whole skill [Part I](#) is trying to build, and this is where it pays off.

Recursion where an operator would do

The second self-inflicted case is reaching for the general tool when a named one exists. *"Everything reachable from here"* can be written as a `FIX`, and *"the parts of the parts"* can be written as a hand-rolled loop — but both are `CLOSURE`, which is shorter, faster (an adjacency-indexed loop rather than the general fixpoint), and says what you mean. `PATH` gives you the distance, `TRACE` the optimal route, `CLUSTER` the undirected groups. Writing a `FIX` where one of these fits is not wrong, just needless work you will maintain forever.

`FIX` earns its keep only when the step needs more than two columns, a filter, or a join each round; `ITERATE` when state is *replaced* rather than accumulated. Reach past the named graph operators only when the shape genuinely does not fit them — [reachability](#) and [recursion](#) draw the line.

Genuinely outside the range

Some problems are not a matter of holding the tool right — Relix simply does not do them.

- **Anything that writes.** Relix reads sources and derives relations from them; it does not insert, update, or delete. A query computes an answer, it does not change the world. "Update every overdue invoice" is not a query — compute the set of overdue invoices with Relix, and hand it to whatever owns the write.
- **Text search and fuzzy matching.** `LIKE` matches a simple pattern, and that is the whole of it. Ranked relevance search, full-text indexing, spelling-tolerant or semantic matching, embeddings — these belong to a search engine, and Relix has nothing to add to them. Join *on* their results if you must, but do not try to compute them here.
- **General-purpose and stateful computation with side effects.** Relix has no I/O, no mutable variables, no calling out to services mid-query. A computation that must talk to the network, keep a mutable cache, or interleave with the outside world is a program, not a relation — write it in a language, and use the `embedding API` to let it *call* Relix for the relational parts.

What is *not* outside the range

One thing worth stating plainly, because it is a natural assumption and it is wrong:

iterative numeric methods are in scope. Convergence — PageRank, label propagation, relaxation, a numeric fixpoint — is not a reason to leave the engine. `ITERATE` runs a computation round by round until it settles, so *"iterate until the numbers stop moving"* is a query like any other. See `iteration`. The line is *writing* and *side effects*, not *iteration*.

The honest default

When a problem is outside the range, the right move is not to force it — it is to do the relational part in Relix and hand the rest to the tool that owns it. Compute the set, export it, and let the writer write, the search engine search, the program run. A front end that knows its own edges is more useful than one that pretends it has none.

17. Recipe finder

Index	By
Recipe finder	Question word, operator, and class

The recipe coordinate

Every recipe opens with a **coordinate**: a single blockquote line, directly under the title, that says what kind of problem the recipe solves. It is written for the reader — computing a coordinate is what [Part I](#) teaches, and a recipe showing its own is the method worked through — and it is machine-readable, so the [recipe finder](#) is generated from it rather than kept by hand, and a build guard can check that every class of problem has a recipe.

Why a line, not front-matter

The coordinate is a line of the page, not YAML front-matter above it, for two reasons.

- **It is content.** The method chapters teach the reader to classify a problem before writing a query; the coordinate at the top of each recipe is that classification, shown. Hidden in front-matter it would stop teaching.
- **It renders.** These manuals render through one markdown grammar — fenced code, `##` sub-headings, blockquotes, lists, tables, indented code — shared by the site and the PDF. A blockquote is part of that grammar; a `---` front-matter block is not, and would be published as a horizontal rule and a run of text. And one artifact that both the reader and the finder read cannot drift, where a second copy in front-matter could.

The format

Directly under the `#` title, one blockquote line, its fields separated by `:`:

```
> **Grain:** one row per entity · **Class:** Universal · **Signals:** every, all, only · **Operators:** ÷, ∇, ▷
```

Field	Axis	Value
Grain	grain	Free text finishing "one row per ...".
Class	quantifier	Exactly one class name from the table in quantifier.md .
Signals	quantifier	The words in a question that point here, comma-separated. Feeds the finder's index by question word, and names the same words the recipe's <i>How to recognise it</i> section expands.
Operators	—	The headline operators the recipe uses, comma-separated. Feeds the index by operator.
Difficulties	difficulty	<i>Optional.</i> One or more of nested, dirty, NULL, unbounded, federated, time, comma-separated. Present only when handling that difficulty is what the recipe is <i>about</i> — every recipe has a NULL pitfall, so mentioning NULL is not enough to earn the tag.

A recipe has **one** Class. A problem that genuinely needs two is two recipes, each linking the other under *Related* — the same rule the backlog applies to an issue that wants two `area:` labels.

The class registry

The set of valid Class names is the Class column of the quantifier table in [quantifier.md](#), and nowhere else, so the classes a reader is taught and the classes a recipe may claim are one list. A coordinate whose Class is not in that table is an error, not a new class.

What reads it

- **The recipe finder** (Part V, generated): invert **Signals** for the index by question word, invert **Operators** for the index by operator, and group by **Class**.
- **The completeness guard** (ships with the manual, in *relix-core*): every class in the registry is covered by at least one recipe or still marked *planned* in the index; every recipe's Class is a registry class; and every coordinate parses — Grain, Class, Signals and Operators all present.

The parse

The coordinate is the first blockquote after the # title. Fields are ·-separated; each is ****Label:**** value; the multi-value fields (Signals, Operators, Difficulties) are , -separated; surrounding whitespace is trimmed; the Class match is case-insensitive. The labels are exactly Grain, Class, Signals, Operators and the optional Difficulties.

Recipe finder

Generated from each recipe's coordinate by `_generate_finder.py` — do not edit by hand. Three ways in: by the word in the question, by the operator, and by the class of problem.

By question word

The words that signal each recipe, alphabetised.

Signal	Recipe
a preview	A random, representative subset
a random sample	A random, representative subset
a representative subset	A random, representative subset
added	What changed, what differs
all	Every: related to all of a set, or all rows pass a test
all combinations	Enough cases to cover
all descendants	When the rule is recursive
always	Every: related to all of a set, or all rows pass a test
ancestors	When the rule is recursive
any	Has at least one
as a list	As columns, as a list, as a tree
as a tree	As columns, as a list, as a tree
as columns	As columns, as a list, as a tree
as of	As of, during, overlapping

Signal	Recipe
at least one	Has at least one
at the time	As of, during, overlapping
average	How many, how much, per what
back out	What value makes this true
belong together	Things that belong together
best per	The most, the top N, the latest
can get to	Reachable, how far, and by what route
cheapest route	Reachable, how far, and by what route
complete	What value makes this true
conflicts with	As of, during, overlapping
connected	Things that belong together
connected to	Reachable, how far, and by what route
consecutive	In a row, per visit, gaps
converge	Iterate until it settles
cover	Enough cases to cover
drift	What changed, what differs
duplicates	Things that belong together
during	As of, during, overlapping
enough cases	Enough cases to cover
ever	Has at least one
every	Every: related to all of a set, or all rows pass a test
every pair	Enough cases to cover
explosion	When the rule is recursive
fill in the blank	What value makes this true
filter	Which rows, and which columns
flatten	As columns, as a list, as a tree
gaps	In a row, per visit, gaps
gather	As columns, as a list, as a tree
generate then filter	Enough cases to cover
generations	Iterate until it settles
given ... find	What value makes this true
goal-seek	What value makes this true
has	Has at least one

Signal	Recipe
highest	The most, the top N, the latest
households	Things that belong together
how derived	Why is this row here, where did it come from
how many	How many, how much, per what
how much	How many, how much, per what
in a row	In a row, per visit, gaps
in effect	As of, during, overlapping
iterate until stable	Iterate until it settles
justify	Why is this row here, where did it come from
keep expanding	When the rule is recursive
latest	The most, the top N, the latest
linked	Things that belong together
maximise	The best combination within limits
minimise	The best combination within limits
mismatch	What changed, what differs
missing	None, never, missing
n at random	A random, representative subset
nested	As columns, as a list, as a tree
never	None, never, missing
never failed	Every: related to all of a set, or all rows pass a test
no	None, never, missing
none	None, never, missing
not in	None, never, missing
numeric fixpoint	Iterate until it settles
only	Every: related to all of a set, or all rows pass a test
only the ones	Which rows, and which columns
orphaned	None, never, missing
overlapping	As of, during, overlapping
parts of parts	When the rule is recursive
per	How many, how much, per what
per interval	How many, how much, per what
per visit	In a row, per visit, gaps
provenance	Why is this row here, where did it come from

Signal	Recipe
reachable	Reachable, how far, and by what route
reconcile	What changed, what differs
removed	What changed, what differs
reproducible sample	A random, representative subset
rings	Things that belong together
runner-up	The most, the top N, the latest
running total	How many, how much, per what
same as	Things that belong together
settle	Iterate until it settles
shortest route	Reachable, how far, and by what route
simulate rounds	Iterate until it settles
since last	In a row, per visit, gaps
some	Has at least one
streak	In a row, per visit, gaps
subject to	The best combination within limits
test matrix	Enough cases to cover
that match	Which rows, and which columns
the best	The best combination within limits
the most	The most, the top N, the latest
the most/least within	The best combination within limits
the next one	In a row, per visit, gaps
top n	The most, the top N, the latest
total	How many, how much, per what
trace back	Why is this row here, where did it come from
under a budget	The best combination within limits
until nothing new	When the rule is recursive
via	Reachable, how far, and by what route
what changed	What changed, what differs
what differs	What changed, what differs
what value	What value makes this true
where from	Why is this row here, where did it come from
which	Which rows, and which columns
which inputs	Why is this row here, where did it come from

Signal	Recipe
while	As of, during, overlapping
why	Why is this row here, where did it come from
wide	As columns, as a list, as a tree
with a	Has at least one
within n hops	Reachable, how far, and by what route
without	None, never, missing
without exceeding	The best combination within limits

By operator

Operator	Recipe
ARGMAX	The most, the top N, the latest
ASOF	As of, during, overlapping
CLOSURE	Reachable, how far, and by what route
CLUSTER	Things that belong together
COLLECT	As columns, as a list, as a tree
COVER	Enough cases to cover
DOWNSAMPLE	How many, how much, per what
FIX	When the rule is recursive
IJOIN	As of, during, overlapping
ITERATE	Iterate until it settles
OPTIMIZE	The best combination within limits
PATH	Reachable, how far, and by what route
PIVOT/UNPIVOT	As columns, as a list, as a tree
ROLLING	How many, how much, per what
SAMPLE p	A random, representative subset
SAMPLE ... ROWS	A random, representative subset
SESSIONIZE	In a row, per visit, gaps
SOLVE	What value makes this true
TOP	The most, the top N, the latest
TRACE	Reachable, how far, and by what route
TREE	As columns, as a list, as a tree
WHY (ω)	Why is this row here, where did it come from
WINDOW LAG/LEAD	In a row, per visit, gaps

Operator	Recipe
WINDOW RANK/ROW_NUMBER/DENSE_RANK	The most, the top N, the latest
× (generate)	Enough cases to cover
÷	Every: related to all of a set, or all rows pass a test
Υ	How many, how much, per what
μ	As columns, as a list, as a tree · Why is this row here, where did it come from
π	Which rows, and which columns
σ	Which rows, and which columns
∀	Every: related to all of a set, or all rows pass a test
Δ	What changed, what differs
—	None, never, missing · What changed, what differs
U	Things that belong together
⋈	Which rows, and which columns
⋈	Has at least one · Which rows, and which columns
▷	Every: related to all of a set, or all rows pass a test · None, never, missing
⋈	What changed, what differs
⋈	Things that belong together

By class

Class	Recipe
Absence	None, never, missing
Comparison	What changed, what differs
Equation	What value makes this true
Existence	Has at least one
Explanation	Why is this row here, where did it come from
Generation	Enough cases to cover
Graph	Reachable, how far, and by what route · Things that belong together · When the rule is recursive
Iteration	Iterate until it settles
Optimization	The best combination within limits
Ranking	The most, the top N, the latest
Reshaping	As columns, as a list, as a tree
Sampling	A random, representative subset

Class	Recipe
Selection	Which rows, and which columns
Sequence	In a row, per visit, gaps
Summary	How many, how much, per what
Temporal alignment	As of, during, overlapping
Universal	Every: related to all of a set, or all rows pass a test