

Relix

Language Reference Manual

The complete reference for the Relix relational-algebra language — one entry per keyword, operator and built-in function.

Generated 1 October 2026 · revision e87dc7b0e95d

Contents

About this manual	8
1. Getting started	9
Getting started (your first script)	9
Glossary (the vocabulary)	12
2. Language & scripts	16
source (data source declaration)	17
source ... from http (HTTP/JSON API source)	19
connection (shared database connection)	24
connection ... from gedcom (GEDCOM genealogy source)	26
connection ... from log (web server access log source)	30
relate (schema relationship declaration)	35
Inline Table (markdown / csv literal)	38
:= and query (views and result queries)	40
Delimited Identifier (`name`)	41
Comments (-- and / ... /)	43
Operator spellings (Unicode and ASCII)	44
Grammar (EBNF)	48
namespace (script namespace declaration)	62
import (reuse symbols from another file)	63
def (scalar function definition)	64
def ... : RELATION (table-valued function)	65
Relation catalog (relix.relationships)	68
Introspection standard library (relix.unused / relix.deps / relix.impact / relix.find / relix.schema / relix.cycles / relix.funcs)	70
Observability feed (relix.events / relix.rules)	73
Component inventory (relix.version)	77
Candidate keys (relix.keys)	78
Reserved namespace (relix.catalog)	80

3. Core operators 83

Selection (σ / SELECT)	83
Projection (π / PROJECT)	85
Rename (ρ / RENAME)	87
Distinct (δ / DISTINCT)	90
Sort (τ / SORT / ORDER [BY])	92
Limit (λ / LIMIT)	94

4. Predicates 97

Comparison Operators ($= \neq < \leq > \geq$)	97
Logical AND ($\wedge$ / AND)	99
Logical OR ($\vee$ / OR)	100
Logical NOT ($\neg$ / NOT)	101
Null Test ($\perp$ / NULL)	102
Set Membership ($\in$ / IN, $\notin$ / NOT IN)	105
Pattern Match (LIKE / NOT LIKE)	107

5. Joins 110

Natural Join ($\bowtie$ / JOIN)	110
Theta Join ($\Join$ / $><$)	112
Left Outer Join ($\Join$ / $ ><$ / LJOIN)	115
Right Outer Join ($\Join$ / $>< $ / RJOIN)	117
Full Outer Join ($\Join$ / $ >< $ / FJOIN)	119
Semi Join ($\bowtie$ / SEMI)	121
Anti Join ($\Join$ / ANTI)	122
AS-OF Join (ASOF)	124
Interval Join (IJOIN)	129
Lateral Join (LATERAL)	136

6. Set operations 141

Union ($\cup$ / UNION)	141
Union All ($\cup$ / UALL)	143

Outer-union / heterogeneous merge ($\sqcup$ / OUNION)	144
Intersection ($\cap$ / INTER / INTERSECT)	146
Set Difference ($-$ / DIFF / MINUS / EXCEPT)	148
Symmetric Difference (Δ / SYMDIFF)	149
Cartesian Product ($\times$ / CROSS)	151
Relational Division ($\div$ / DIV)	152
Composition ($\circ$ / COMPOSE)	154

7. Grouping & aggregation 158

Group / Aggregation (γ / GROUP [BY])	158
SUM (aggregate)	161
AVG (aggregate)	162
COUNT (aggregate)	164
MIN (aggregate)	165
MAX (aggregate)	167
COLLECT (aggregate / NEST)	168
ARGMAX (aggregate)	170
ARGMIN (aggregate)	171
Window — sliding / cumulative aggregate (ROLLING)	173
Window — ranking functions (WINDOW ROW_NUMBER / RANK / DENSE_RANK / PERCENT_RANK / NTILE)	176
Window — offset functions (WINDOW LAG / LEAD / FIRST_VALUE / LAST_VALUE)	179

8. Nested data (NF²) 183

Struct Construction ($\{ \dots \}$)	183
Array Construction ($[\dots]$)	185
Unnest (μ / UNNEST)	187
WITH ORDINALITY (unnest index clause)	189
Unpivot (UNPIVOT)	191
Pivot (PIVOT)	193
Tree (TREE)	195

9. Literals & temporal types 199

DATE literal	199
TIME literal	200
TIMESTAMP literal	201
DURATION literal	202
Truth relation literals (UNIT / EMPTY, a.k.a. DEE / DUM)	203

10. Built-in functions 207

Len (string length)	207
UCase (upper-case)	208
LCase (lower-case)	208
Trim (strip surrounding whitespace)	209
LTrim (strip leading whitespace)	209
RTrim (strip trailing whitespace)	210
Left (leftmost characters)	211
Right (rightmost characters)	212
Mid (substring)	212
InStr (find substring position)	213
Chr (character from code)	214
Asc (code of first character)	215
Replace (substitute text)	216
Abs (absolute value)	217
Int (floor to integer)	217
Fix (truncate toward zero)	218
Ceil (round up)	219
Round (round to nearest)	220
Sgn (sign)	221
Sqr (square root)	222
Log (natural logarithm)	222
Exp (exponential)	223
Power (raise to a power)	224
Sin (sine)	225
Cos (cosine)	225

Tan (tangent)	226
Atn (arctangent)	226
Rand (random number)	227
NOW (current timestamp)	228
CURRENT_DATE (today)	229
CURRENT_TIME (current clock time)	230
YEAR (extract year)	232
MONTH (extract month)	232
DAY (extract day of month)	233
HOUR (extract hour)	234
MINUTE (extract minute)	235
SECOND (extract second)	235
DATE_TRUNC (truncate timestamp to a unit)	236
MINUTES (measure a duration in minutes)	238
SECONDS (measure a duration in seconds)	239
DAYS (measure a duration in days)	239
to_date (parse a string to a DATE)	240
to_time (parse a string to a TIME)	241
to_timestamp (parse a string to a TIMESTAMP)	241
IIf (inline if / conditional value)	242
Nz (null-to-value)	243
Coalesce (first non-null)	244
IsNull (test for missing value)	245
IsNumeric (test if value looks numeric)	246
CStr (convert to string)	247
CInt (convert to integer)	247
CDBl (convert to number)	248
Entries (read an object's fields as rows)	249

11. Advanced operators 251

Universal Quantification ($\forall$ / FORALL)	252
Top-K Per Group (TOP ... PER)	254

Transitive Closure (CLOSURE / RCLOSURE, or postfix + / *)	256
Connected Components (CLUSTER)	259
Bounded Path Reachability (PATH)	262
Optimal-Path Extraction (TRACE)	264
General Recursion (FIX)	268
Iteration (ITERATE)	273
Goal-Seek (SOLVE)	276
Declarative Optimisation (OPTIMIZE)	278
Combinatorial Covering (COVER)	281
Bernoulli Sampling (SAMPLE p)	283
Reservoir Sampling (SAMPLE n ROWS)	285
DOWNSAMPLE	288
Gap-and-Island / Sessionization (SESSIONIZE)	291
Why (WHY / ω)	294
Query optimizer (rewrite rules)	299
Pushdown (what the backend computes)	323

About this manual

This is the complete reference for the Relix relational-algebra language — one page per keyword, operator and built-in function. The pages are arranged below in a **learning order**: start with [getting started](#), which runs a first script end to end, then work down through the statements of a `.relix` file and the core operators toward joins, set algebra, aggregation, nested data and the advanced operators. The [glossary](#) defines the vocabulary the rest of the manual assumes.

Every operator has both a **Unicode** form (e.g. `σ`, `⋈`, `⋈`) and an **ASCII** keyword form (e.g. `SELECT`, `JOIN`, `UNION`); the two are interchangeable and produce identical results. Each page gives the syntax, a plain-English description, real-world examples, and "See Also" links to related pages. The harder operators include a fully worked example with sample data and the expected result.

- > New to relational algebra? Read the sections in order. Already fluent? Jump
- > straight to the operator you need from the tables below.

1. Getting started

New here? This is the whole language in one page: bring data in, filter and combine it, name the interesting steps, ask for a result. Everything below is detail on one of those four moves.

Page	What it does
Getting started	Your first script — declare data, name a result, run it
Glossary	The vocabulary: relation, tuple, bag vs set, materialisation, pushdown

Getting started (your first script)

Syntax

```
source Name from csv("...") { ... }; -- 1. bring data in
Name := { <expression> };           -- 2. name a result
query { <expression> };              -- 3. ask for one back
```

Description

A Relix script is a text file holding three kinds of statement: declarations that bring data in, assignments that give a name to a result, and `query` statements that say which results you actually want back. Nothing is written anywhere — a script reads data and produces answers.

You never say *how* to compute an answer. You write the shape of the result you want and the engine decides the order of work: which filters to push down into the database, which join algorithm to use, what to evaluate once and reuse.

Every operator has two spellings, a mathematical one and a keyword one — σ and `SELECT`, $\bowtie$ and `JOIN`, $\cup$ and `UNION`. They parse to exactly the same thing, so use whichever you can type. This manual leads with the symbols because they are shorter to read.

Technical Description

The whole script is analysed before any of it runs: imports are resolved, every relation and function name is registered, and each expression's output columns and types are inferred and checked against the operators using them. A mistyped column is an analysis error naming the line, not a failure halfway through a result set.

An assignment (`:=`) declares a *view*, not a copy of the rows. It names an expression, and the optimizer folds that expression into whatever reads it — so splitting a long query into named steps costs nothing at run time and is the normal way to write one.

Examples

The same two relations read from CSV files rather than written inline:

```
source Customers from csv("./data/customers.csv") {
  header: true,
  schema: { customer_id: NUMBER, name: STRING, city: STRING }
};
```

```

source Orders from csv("./data/orders.csv") {
  header: true,
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER }
};

query { γ name, SUM(amount) → total (Customers ⋈ Orders) };

```

Reading from a database instead — only the declaration changes, never the query:

```

source Orders from database {
  url: "${DB_URL}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER }
};

query { σ amount ≥ 100 (Orders) };

```

The same expression typed in ASCII, for a keyboard without the glyphs:

```

Orders := [
| order_id | customer_id | amount |
|-----|-----|-----|
| 100      | 1           | 120    |
];

query { SELECT amount >= 100 (Orders) };

```

Worked Example

A first script, built up one statement at a time. The data is written inline so that the whole thing runs with no database and no files — copy any block into a script and run it.

These two relations recur throughout the manual: three customers, four orders, and two deliberate awkward rows — **Cara** has never ordered, and **order 103** belongs to customer 4, who is not in Customers.

```

Customers := [
| customer_id | name  | city  |
|-----|-----|-----|
| 1          | Alice | London |
| 2          | Bob   | Berlin |
| 3          | Cara  | Lisbon |
];

Orders := [
| order_id | customer_id | amount |
|-----|-----|-----|
| 100      | 1           | 120    |
| 101      | 1           | 80     |
| 102      | 2           | 45     |
| 103      | 4           | 60     |
];

```

An inline table is a relation like any other, so the shortest possible query asks for one back unchanged:

```

query { Orders };

```

order_id	customer_id	amount
100	1	120
101	1	80
102	2	45

```
103      4      60
(4 rows)
```

Keep some rows, then keep some columns. `σ` (SELECT) filters rows by a condition; `π` (PROJECT) picks the columns to keep, and renames one on the way out with `→`. Read the expression from the inside out — the relation is in the brackets, and each operator wraps the one before it:

```
query { π order_id, amount → value (σ amount ≥ 80 (Orders)) };
```

```
order_id  value
-----
100      120
101       80
(2 rows)
```

Bring the two relations together. `⋈` (JOIN) matches rows on the columns the two sides share — here `customer_id`, which is why the result carries it only once. Alice appears twice because she has two orders: a join multiplies rows, it does not look values up. Cara and order 103 both disappear, having nothing on the other side to match:

```
query { Customers ⋈ Orders };
```

```
customer_id  name  city  order_id  amount
-----
1  Alice  London  100      120
1  Alice  London  101       80
2  Bob   Berlin  102       45
(3 rows)
```

Summarise, and name the step. `γ` (GROUP) collapses rows that share a key into one row per key, with an aggregate over each group. `:=` names the result so the next expression can read it — that name is a view, not a stored copy, so the engine still runs this as a single query:

```
CustomerTotals := { γ name, SUM(amount) → total (Customers ⋈ Orders) };
```

```
query { τ total DESC (CustomerTotals) };
```

```
name  total
-----
Alice  200
Bob    45
(2 rows)
```

That is the whole shape of a Relix script: declare data, filter and combine it, name the interesting steps, and ask for what you want.

Limitations

A script only reads. There is no statement that writes rows back to a source, creates a table, or changes one.

See Also

[glossary](#), [inline-table](#), [source](#), [assignment & query](#), [select](#), [project](#), [group](#), [natural-join](#)

Notes

Where to go next: read chapter 1 for the rest of the statements a script can hold (database sources, imports, function definitions), or jump to the operator you need from the tables in the index. Every operator page carries a worked example that runs on inline data, so any of them can be tried without a database.

Glossary (the vocabulary)

Syntax

R, S	stand for whole relations (or expressions producing one)
a, b, c	stand for attributes – the columns of a relation
$\sigma \pi \rho \gamma \tau \lambda \delta \mu$	the unary operators: one relation in, one relation out
$\bowtie \ltimes \ltimes \ltimes \ltimes \ltimes \ltimes \ltimes \ltimes$	the joins: two relations in, one out
$\cup \cap \setminus - \times \div \circ \Delta$	the set operations
$\wedge \vee \neg$	the connectives that build a condition

Description

The terms this manual uses, in the order you meet them. Every one links to the page that covers it properly. If you are here before writing anything, read [getting started](#) first — it uses most of these words in a single page of script.

The data model

- **Relation** — a table: a set of rows that all have the same named columns. It is the only kind of value an operator takes and the only kind it returns, which is why operators nest freely.
- **Tuple**, or **row** — one member of a relation: a value for each column.
- **Attribute**, or **column** — one named, typed field of every row.
- **Heading**, or **schema** — a relation's column names and types. Relix works out the heading of every expression before running it, so an operator that cannot apply to its input is an error at analysis time.
- **Degree** — how many columns a relation has. **Cardinality** — how many rows.
- **Type** — what a column holds: `NUMBER`, `STRING`, `BOOLEAN`, the temporal types `DATE/TIME/TIMESTAMP/DURATION`, a `struct`, an `array`, or `ANY` for data whose shape is only known when it is read.
- **NULL** — a missing value, not a zero or an empty string. Comparing it to anything yields neither true nor false, so `σ` drops the row; test for it with `IS NULL`.
- **Bag** and **set** — a bag may hold the same row twice, a set may not. Relix keeps duplicates unless you ask for them to go: `⋈` deduplicates, `⋈` does not, and `δ` removes them from anything.
- **Key** — a set of columns whose values identify a row uniquely. The engine derives keys where it can and uses them to skip work, such as a `δ` over rows that are already distinct.
- **Nested relation (NF²)** — a relation whose columns may themselves hold structs, arrays, or whole relations. `COLLECT` builds one and `μ` flattens it back.

Writing a script

- **Statement** — one declaration, assignment or query, terminated by `;`.
- **Source** — a **declaration** binding a name to external data: a database table, a CSV or JSON file, an HTTP endpoint.
- **Base relation** — a relation the data comes from directly (a source or an **inline table**), as opposed to one computed from others.
- **View** — a name given to an expression with `:=`. It stores no rows; the optimizer folds the expression into whatever reads it, so naming the steps of a long query is free.
- **Query statement** — `query { ... }`, which marks a result you want back. A script with no query computes nothing.
- **Namespace** and **import** — how one script's **names** are grouped and how another script **reuses** them.
- **Scalar function** — a function over single values, `Len(name)`, either built in or **defined** in the script.
- **Table-valued function (TVF)** — a **function returning a relation**, callable anywhere a relation is expected.
- **Relationship** — a named, bounded link between two relations, declared with `relate` or a source's `references` block. Together they form the **schema graph**, which is what lets a join be resolved from the names alone.
- **Catalog relation** — a read-only `relidx.*` relation describing the script or the engine itself, queried like any other: the **introspection stdlib** and the **observability feed**.

Operators and expressions

- **Operator** — a function from relations to a relation. Each has a symbol and an equivalent keyword (σ = `SELECT`); the two parse identically.
- **Predicate** — a condition evaluated per row, built from **comparisons**, `IN`, `LIKE` and the connectives `^` `v` `→`. It is what goes inside a σ or a join.
- **Selection** (σ) keeps rows; **projection** (π) keeps, reorders, renames and computes columns; **rename** (ρ) renames a relation or its columns.
- **Aggregate** — a function reducing a group of rows to one value (`SUM`, `COUNT`). **Grouping** (γ) splits a relation by key and applies aggregates to each group.
- **Join** — combining two relations by matching rows. **Natural** matches on shared column names, **theta** on an explicit condition, **outer** keeps unmatched rows with NULLs, and **semi/anti** filter one side by whether a match exists rather than combining columns.
- **Set operations** — `u`, `n`, `-` over rows of the same shape; `x` pairs every row with every row; `÷` answers "for *all* of these"; `o` joins and then drops the matched columns.
- **Quantification** ($\forall$) — keeps the groups in which every row satisfies a condition, as opposed to σ , which asks about one row at a time.
- **Recursion** — `CLOSURE` walks a graph edge relation to reachability; `FIX` is the general least-fixpoint form.
- **Window** — an aggregate or ranking computed over the rows *around* each row rather than collapsing them: `ROLLING`, `WINDOW`.

- **Provenance**, or **lineage** — which input rows produced an output row. `ω` reifies it as a queryable column.

How a query runs

- **Analysis** — the pass that resolves names, infers every expression's heading and reports errors, before any data is read. Its output is the **semantic model** the later stages read.
- **Logical plan** — the tree of operators the script describes. **Physical plan** — what the engine will actually execute: join algorithms chosen, scans bound to connectors, work ordered.
- **Optimizer rule** — a rewrite that replaces part of the logical plan with a cheaper equivalent, each with a code (SEL-001, PROJ-004) you can see fire. See [the optimizer](#).
- **Pushdown** — sending work to the system holding the data, so a filter becomes a SQL `WHERE` clause or a Mongo `$match` and the rows never travel.
- **Streaming** and **blocking** — a streaming operator emits a row as soon as it has one; a blocking operator (sort, grouping, the deduplicating set operations) must see all its input first. **Materialisation** is that buffering.
- **Cardinality estimate** — the planner's guess at how many rows an operator will produce, used to choose between plans. It is a guess, and `:explain` shows it as `~N rows`.
- **Shared sub-plan** — one sub-expression read by several parts of a plan and evaluated once, rather than recomputed per reader.
- **Boundedness** — whether a relation is known to be finite. A [generator](#) can be unbounded, and an operator that must see every row cannot run over one.
- **Connector** — the component that reads an external system (JDBC, CSV, JSON, HTTP, MongoDB) and, where it can, executes pushed-down work on the engine's behalf.

Examples

Every part of one small query, named:

```
Orders := [
| order_id | customer_id | amount |
|-----|-----|-----|
| 100      | 1           | 120    |
| 101      | 1           | 80     |
| 102      | 2           | 45     |
];

BigSpenders := { τ total DESC (γ customer_id, SUM(amount) → total (σ amount ≥ 50 (Orders))) };

query { BigSpenders };
```

Reading it from the inside out: `Orders` is a **base relation**, here an **inline table**. `σ amount ≥ 50` is a **selection** whose **predicate** is a comparison. `γ customer_id, SUM(amount) → total` is a **grouping**: `customer_id` is the grouping key, `SUM` the **aggregate**, `total` the output **attribute**. `τ total DESC` **sorts**, and is a **blocking** operator — it cannot emit its first row until it has seen the last. `BigSpenders :=` names the whole expression as a **view**, storing nothing, and `query { ... }` asks for it back.

Had `Orders` been a database source rather than an inline table, the optimizer would have pushed the selection, the grouping and the sort into one SQL statement — **pushdown** — and the engine would have read only the three result rows.

See Also

[getting started](#), [optimizer](#), [repl](#), [assignment & query](#), [relate](#)

Notes

Relix follows the standard relational-algebra vocabulary, so a textbook's "relation, tuple, attribute" mean here exactly what they mean there. Where SQL uses a different word for the same idea — table, row, column — the two are used interchangeably in this manual.

2. Language & scripts

The shape of a `.relix` file — declaring data, naming results, and reusing code.

Page	What it does
source	Declare an external data source (CSV, database, ...)
http source	Read JSON from an HTTP/REST API (headers, auth, body)
connection	Share one database connection across many sources
gedcom source	Read a GEDCOM genealogy file as individuals and families
log source	Read a web server access log as a typed relation
relate	Declare named, bounded relationships between relations (schema graph)
inline-table	Embed small reference data directly in a script
assignment & query	Name a view with <code>:=</code> and mark a result query
delimited identifier	Backtick a name that collides with a reserved word (<code>`order`</code>)
comments	<code>--</code> to end of line, <code>/* ... */</code> for a block
operator spellings	Every operator's Unicode glyph and its ASCII equivalent
grammar (EBNF)	The whole language as one EBNF grammar — every statement, operator and token
namespace	Declare the script's namespace
import	Reuse symbols defined in another file
def	Define a scalar (per-value) function
def ... : RELATION	Define a table-valued (relation-returning) function
relation catalog	<code>relix.relations</code> — every relation the script can name, with its kind, size and whether it ends
introspection stdlib	<code>relix.unused</code> / <code>relix.deps</code> / <code>relix.impact</code> / <code>relix.find</code> / <code>relix.schema</code> / <code>relix.cycles</code> / <code>relix.funcs</code> — the engine's own introspection, shipped as Relix
observability feed	<code>relix.events</code> / <code>relix.rules</code> — the previous run's engine decisions, as queryable rows
component inventory	<code>relix.version</code> — the engine, the facade and every discovered provider, as queryable rows
candidate keys	<code>relix.keys</code> — the collected candidate keys of every relation, one row per key column
reserved namespace	<code>relix.catalog</code> — the <code>relix.*</code> surface describing itself

source (data source declaration)

Syntax

```
source <Name> from database { url: "...", table: "...", schema: { ... } };
source <Name> from csv("<path>") { header: true, schema: { ... } };

source Orders from database {
  url: "${DB_URL}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER }
};
```

Description

A source declaration is how external data enters a script. It names a relation and tells `relix` where the rows come from — a database table or a CSV file — and what columns to expect. Once declared, you use the name anywhere a relation is expected, just like any other table.

Technical Description

`source` binds a relation symbol (`SourceRelationSymbol`) to an external connector. `from database` requires `url`, `table`, and a `schema`; `from csv("path")` takes an optional `header` flag (default `true`) and a `schema`. Column types are `NUMBER`, `STRING`, `BOOLEAN`, `ANY`, and the temporal types `DATE`/`TIME`/`TIMESTAMP`/`DURATION`. A column may also be **nested**: `{ field: TYPE, ... }` declares a struct and `[TYPE]` an array, composing to any depth. A column or field whose name is not a plain identifier, such as `unit-price`, is written between backticks, as it is in a query; a doubled backtick stands for one. `${VAR}` placeholders in URLs and other string values are resolved from the active environment when a query runs. Other source kinds (`json`, `http`, `generator`) exist for their respective connectors. CSV paths resolve relative to the script's directory.

In a CSV file an **empty field is a NULL** — `2,Grace` gives a NULL third column. A row that simply *ends early* is a different thing and is refused, naming the line and the column it ran out at, rather than read as a row of NULLs. A row carrying **more** fields than the schema names is read: a schema may deliberately name fewer columns than the file carries.

Every source kind (`database`, `connection-table`, `csv`, `json`) additionally accepts a `references` block declaring foreign-key relationships from this source's columns to other relations: each `col -> Target.col` entry (composite: `(a, b) -> Target(a, b)`) becomes a named edge of the schema graph, with an implied upper bound of 1 on the target side. Inline tables take the same block as a trailing `references { ... }` clause. See [relate](#) for the full relationship model (names, inverse names, bounds, symmetric self-edges).

Examples

A database table:

```
source Orders from database {
  url: "${DB_URL}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER }
};
```

A nested column — a struct, and an array of structs:

```
source Docs from mg {
  table: "orders",
  schema: {
    oid: NUMBER,
    address: { city: STRING, geo: { lat: NUMBER, lon: NUMBER } },
    items: [{ sku: STRING, qty: NUMBER }]
  }
};
```

The declaration is what types a path through the column: `address.geo.lat` infers as `NUMBER` and `μ items` yields the struct, where an `ANY` column would give `ANY` for both and catch no misspelled field. Declare `ANY` when the shape genuinely varies — schema-on-read is still what a document store often wants.

A table whose column names are not plain identifiers, named between backticks here and in the query that reads them:

```
source Lines from database {
  url: "${DB}", table: "order-lines",
  schema: { lid: NUMBER, `unit-price`: NUMBER, `line total`: NUMBER }
};
query { σ `unit-price` > 10 (Lines) };
```

Backticks name a column, not a relation. A source's own name must be a plain name:

```
source `order-lines` from database { url: "${DB}", table: "order-lines" };
```

A CSV file with a header row:

```
source Responses from csv("./data/responses.csv") {
  header: true,
  schema: { id: NUMBER, region: STRING, score: NUMBER }
};
```

A table with temporal columns:

```
source Events from database {
  url: "${DB}", table: "events",
  schema: { id: NUMBER, occurred_at: TIMESTAMP, day: DATE }
};
```

A table declaring its foreign key, so the engine knows how it joins:

```
source Customers from database { url: "${DB}", table: "customers",
  schema: { customer_id: NUMBER, name: STRING } };
source Orders from database {
  url: "${DB}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER },
  references: { customer_id -> Customers.customer_id }
};
```

Limitations

Nesting is available on a *column* only — a `def`'s parameters and return type are scalar. A CSV source reads a nested-typed column as raw text, since its cells are flat. The schema you declare is what the engine trusts; mismatched real data is coerced or errors per connector. Secrets in URLs should come from the environment via `${VAR}`, not be hard-coded.

Alternatives

A `connection` declaration plus dotted refs (`conn.table`) shares one set of credentials across many tables. Inline tables embed small reference data directly.

See Also

[http source](#), [connection](#), [relate](#), [inline-table](#), [namespace](#), [import](#)

Notes

Use `${VAR}` placeholders (see `relix help env`) to keep connection URLs and credentials out of the script text. The script keeps the placeholder, so nothing that prints it shows the value.

source ... from http (HTTP/JSON API source)

Syntax

```
source <Name> from http {
  url:      "<url>",                // required; may contain ${ENV} and {pathParam}
  method:   GET | POST | QUERY,    // optional, default GET (Relix is read-only)
  headers:  { "<Name>": "<value>" }, // optional static request headers
  auth:     bearer("...") | basic("...", "...")
           | apikey("...", "...") | apikey(query("..."), "..."),
  body:     "<request body>",      // for POST/QUERY reads; required by QUERY
  extract:  json("<jsonpath>"),    // optional; locates the records array
  schema:   { <col>: <TYPE> at "<jsonpath>", ... } // optional; omit for open
};
```

Description

An HTTP source turns a JSON REST/GraphQL endpoint into a relation. Relix issues the request, parses the JSON response, and yields one row per record. It is the network sibling of the local `json("...")` file source.

Relix is **read-only by design**: only `GET`, `QUERY` and `POST` are allowed. `QUERY` (RFC 10008) is the method made for a read whose query travels in the request body: it is safe and idempotent by definition, and a `QUERY` source must declare a `body`. `POST` is used *solely* to carry a read query in the body for the many endpoints that do not accept `QUERY` (the GraphQL/search idiom). The mutating methods (`PUT`/`PATCH`/`DELETE`) and the bodyless `HEAD` are rejected:

```
source Items from http { url: "https://api.example.com/items", method: PUT };
```

By default a source is **open** (schema-on-read): each JSON object becomes a document row you navigate by path at query time (e.g. `user.name`, `items[0].id`). Declare a `schema { ... }` to make it a **closed, typed** relation whose columns are extracted from each record by an `at "$.path"` binding (or the column name).

Technical Description

`url` is the only required field. `${NAME}` placeholders (in the URL, header values, body, and auth values) keep secrets out of the source text. They are resolved when a query runs: from the active environment in the `relix` command and the REPL, and through the resolver its session was built with in a program embedding `relix`. The declaration keeps the placeholder, so nothing that prints the session shows the value, and a placeholder with no value fails the query with an error naming it and the source.

`extract: json("$.path")` locates the records: the value at the path must be a JSON array (one record per element) or a single object (one record). When omitted, the response body itself is treated as the records — a top-level array or object, exactly like the local JSON file source. `JSONPath` roots (`$`, `$.`) and the trailing `[*]` are accepted and normalised.

`auth` shorthands expand to the standard wire form: `bearer(t)` → `Authorization: Bearer t`; `basic(u,p)` → `Authorization: Basic base64(u:p)`; `apikey(name,v)` → the header `name: v`; `apikey(query(name),v)` → the URL parameter `?name=v`. Equivalent raw headers work too.

A header is set in one place. `headers`, `auth` and an `IN` column bound as `header("...")` all set request headers, and naming the same header in two of them is an analysis error naming the source and the header. Names are compared case-insensitively, as HTTP compares them, so `auth: bearer(...)` beside a declared `"authorization"` header is refused too. Neither of the two values is an obvious winner, so `relix` does not pick one.

Some headers cannot be declared. The HTTP client sets `Connection`, `Content-Length`, `Expect`, `Host` and `Upgrade` itself, and declaring one is an analysis error. So is a header name that is not a valid HTTP token, or a value containing a line break. A value that only becomes invalid once a `${...}` placeholder is substituted is reported when the request is built, as an error naming the source and the header.

A request with a body is sent as JSON unless it says otherwise. When a request carries a body (a declared `body`, or the generated GraphQL document below), `relix` adds `Content-Type: application/json`. A `Content-Type` declared in `headers`, in any capitalisation, is sent instead. A request with no body, such as an ordinary `GET`, carries no `Content-Type`.

A redirect is followed. A 301, 302, 307 or 308 answer to a `QUERY` resends the `QUERY` with its body, and a 303 fetches the result with `GET`, as RFC 10008 specifies.

Typed columns coerce the extracted value to the declared type (`NUMBER`, `STRING`, `BOOLEAN`, `ANY`, `DATE/TIME/TIMESTAMP/DURATION`). A missing path navigates to `NULL` rather than erroring.

Examples

Fetch two columns from an open (schema-less) endpoint, navigating the JSON by path:

```
source Users from http { url: "https://jsonplaceholder.typicode.com/users" };
query { π name, email (Users) };
```

Pin named, typed columns and pull a value out of a nested object with an `at` `"$.path"` binding:

```
source Users from http {
  url: "https://jsonplaceholder.typicode.com/users",
  schema: { id: NUMBER, name: STRING, email: STRING, city: STRING at "$.address.city" }
};
```

```
query { τ id (Users) };
```

Locate a records array nested under a key, and page through results:

```
source Products from http {
  url:      "https://dummyjson.com/products",
  extract:  json("$.products[*]"),
  paginate: { limit: query("limit") [default: 5] },
  schema:   { id: NUMBER, title: STRING, price: NUMBER, brand: STRING }
};
query { τ price DESC (σ price > 10 (Products)) };
```

Send an authenticated POST read with a JSON body. The body goes out as `application/json` without a header saying so:

```
source Echo from http {
  url:      "https://httpbin.org/anything",
  method:   POST,
  auth:     bearer("${API_TOKEN}"),
  body:     "{ \"hello\": \"relix\" }",
  extract:  json("$.json")
};
query Echo;
```

Read with `QUERY`, sending a query in a language the endpoint names. The declared `Content-Type` replaces the JSON default:

```
source OpenOrders from http {
  url:      "https://api.example.com/orders",
  method:   QUERY,
  headers:  { "Content-Type": "application/sql" },
  body:     "SELECT id, total FROM orders WHERE status = 'OPEN'",
  extract:  json("$.rows"),
  schema:   { id: NUMBER, total: NUMBER }
};
query { τ total DESC (OpenOrders) };
```

Live Examples

All four examples below are real, public, no-auth endpoints and run as-is with `relix --exec`.

(1) OPEN source — no schema; navigate the JSON by path at query time. The endpoint returns a top-level array of user objects:

```
source Users from http { url: "https://jsonplaceholder.typicode.com/users" };
query { π name, email (λ 2 (Users)) };
```

Result:

name	email
Leanne Graham	Sincere@april.biz
Ervin Howell	Shanna@melissa.tv

(2 rows)

(2) TYPED source — pin named, typed columns; pull a value out of a nested object with an `at "$.path"` binding. Each user object looks like `{"id":1,"name":"Leanne Graham","email":"...", "address":{"city":"Gwenborough",...}}`, so `city` is extracted from `address.city`:

```
source Users from http {
  url: "https://jsonplaceholder.typicode.com/users",
```

```

    schema: {
      id:    NUMBER,
      name:  STRING,
      email: STRING,
      city:  STRING at "$.address.city"
    }
  };
query { τ id (λ 3 (Users)) };

```

Result:

id	name	email	city
1	Leanne Graham	Sincere@april.biz	Gwenborough
2	Ervin Howell	Shanna@melissa.tv	Wisokyburgh
3	Clementine Bauch	Nathan@yesenia.net	McKenziehaven

(3 rows)

(3) **EXTRACT** path + pagination — the records live under a key, not at the root. The response is `{"products": [ ... ], "total": ..., "limit": ...}`, so **extract** locates the array and the pagination default adds `?limit=5` to the request URL. Ordinary σ/τ then filter and sort the fetched rows:

```

source Products from http {
  url:      "https://dummyjson.com/products",
  extract:  json("$.products[*]"),
  paginate: { limit: query("limit") [default: 5] },
  schema:   { id: NUMBER, title: STRING, price: NUMBER, brand: STRING }
};
query { τ price DESC (σ price > 10 (Products)) };

```

Result:

id	title	price	brand
2	Eyeshadow Palette with Mirror	19.99	Glamour Beauty
3	Powder Canister	14.99	Velvet Touch
4	Red Lipstick	12.99	Chic Cosmetics

(3 rows)

(4) **POST** read with a body + auth — `httpbin.org` echoes the request back, so this shows the body and a bearer token reaching the server. The echoed `json` field holds the body we sent:

```

source Echo from http {
  url:      "https://httpbin.org/anything",
  method:   POST,
  auth:     bearer("${API_TOKEN}"),
  headers:  { "Content-Type": "application/json" },
  body:     "{ \"hello\": \"relix\" }",
  extract:  json("$.json")
};
query Echo;

```

This issues **POST** `https://httpbin.org/anything` with header `Authorization: Bearer <API_TOKEN>` and the given body. With no schema declared, `extract: json("$.json")` makes the echoed body the single open document row:

```

hello
_____
relix

```

(1 row)

To prove the token and body reached the server, extract `httpbin's echo` directly — `$.headers.Authorization` is `Bearer <API_TOKEN>` and `$.data` is the sent body.

GraphQL — the request asks for what the query reads

A `POST` source that declares an `extract` path and **no** body has its request generated: `relix` sends a GraphQL document selecting exactly the columns the query reads, and nothing else.

That is projection pushdown, and GraphQL takes a different shape to SQL's. There is no `SELECT` list to render, because a GraphQL selection set *is* the field list — so the fold is simply that the scan is asked for fewer columns and the document asks for those.

```
source Users from http {
  url:      "https://api.example.com/graphql",
  method:   POST,
  extract:  json("$.data.users"),
  schema:   { id: NUMBER, name: STRING, email: STRING, bio: STRING }
};

query { π id, name (Users) };
```

sends, as `Content-Type: application/json`,

```
{"query":"{ users { id name } }"}
```

The `extract` path says where the rows sit and therefore what to select from — `$.data.users` selects users, GraphQL's data envelope being a wrapper rather than a field. A column's at binding nests the same way: at `"$.author.name"` becomes `author { name }`, and two columns under one parent share its selection set.

A declared body is sent verbatim. Generation only ever fills an absence, so a query you wrote is the query that goes — `relix` never rewrites it to something narrower. A `POST` with no body could not have been reaching a GraphQL endpoint before, so no working source changes meaning. Generation is for `POST` alone: the GraphQL-over-HTTP convention is `POST`, and a `QUERY` source always declares its body.

Only the projection folds. A `σ` stays in the engine, because GraphQL has no general predicate language: filtering is whatever arguments each schema happens to define (`where:`, `filter:`, a Relay connection), and there is nothing in the protocol to render a comparison into. A `λ` stays in the engine for the same reason — `first:` is a widely-followed convention rather than part of GraphQL.

An `IN` column is a request parameter and not a field, so it is never selected even when the query reads it back. An **open** source declares no columns at all, and a GraphQL request must name its fields up front, so nothing is generated for one.

Limitations

Predicates are not pushed into request parameters: a `[required]` `IN` column with no `[default: ...]` is an error. Provide a default, or filter the result with `σ` after the fetch. A pagination default is a whole number:

```
source Products from http { url: "https://dummyjson.com/products",
  paginate: { limit: query("limit") [default: 2.5] } };
```

The response is read fully into memory (no streaming cursor). `extract: csv(...)` over HTTP parses but does not execute — use JSON. PUT/PATCH/DELETE/HEAD are not available (Relix never mutates data). A generated GraphQL request carries no arguments — no filter, no page size, no variables — so a source needing them declares its own `body` and gives up the generated projection.

See Also

[source](#), [connection](#), [assignment & query](#)

Notes

Keep tokens and keys out of the script with `${VAR}` (see `relix help env` for the command line). They are resolved before the request is built, and their values are never logged.

connection (shared database connection)

Syntax

```
connection <Name> from <type> { url: "...", ... };

connection analytics from database { url: "${ANALYTICS_DB}" };

-- then reference tables by dotted name:
query { σ amount > 100 (analytics.orders) };
```

Description

A connection declares one named link to a database that many tables can share, instead of repeating the URL and credentials in every source. After declaring it, you refer to its tables by a dotted name (`analytics.orders`), and relix introspects each table's columns for you.

Technical Description

`connection X from <type> { ... }` declares a database connection. The connector type after `from` is the registry dispatch key — `database` is a permanent alias for `jdbc`; other tokens (e.g. `mongodb`) dispatch to the matching connector. JDBC connections accept `url`, `user`, `password`, `dialect` and `collation`, and require `url`. Tables are referenced as binding-form sources (`source T from conn { table, schema }`) or dotted refs (`conn.table`), the latter resolved by introspecting the live catalog. Declared connections also appear, redacted, in the `relix.connections` system catalog (name + dialect only — never the URL/credentials).

A file connection — `csv`, `gedcom`, `log` — names one file or directory, by `path` or by `url`, never both. A relative `path` resolves against the script's directory, the rule a `csv("...")` source follows. A `url` is either `file:` (a path written as a URL) or `https:`, which is fetched. A fetched file is kept under `~/.relix/cache/files` and is fetched at most once per session, so every scan in a session reads the same bytes even if the file changes on the server. A later session asks the server whether the file changed (ETag/Last-Modified) and reuses the cached copy when it did not, or when the server cannot be reached. `sha256: "<hex>"` pins the file's content, and a file whose digest differs is refused. A compressed (gzip) file is read as its content. A program embedding relix can forbid fetching; a connection naming an `https` file is then an error.

Examples

Declare once, query many tables:

```
connection shop from database { url: "${SHOP_DB}" };
query { shop.customers ⋈ shop.orders };
```

Mongo connection (registry dispatch by type token):

```
connection events from mongodb { url: "${MONGO_URL}" };
```

A file served over https, pinned to the content the query was written against:

```
connection sample from log {
  url: "https://example.org/logs/access.log",
  sha256: "7d8f3a1c0b9e2f4d5a6c7b8e9f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f80",
  format: "combined"
};
```

Bind a specific table with an explicit schema:

```
source Orders from shop { table: "orders",
  schema: { id: NUMBER, amount: NUMBER } };
```

A MySQL connection whose string columns are declared with a binary collation, so string comparison may be pushed down as written:

```
connection shop from database { url: "${SHOP_DB}", collation: exact };
```

Limitations

`collation` is a claim about the database that `relix` cannot check, because a collation belongs to each column and nothing the planner reads reports it. Getting it wrong costs correctness rather than speed: declaring `exact` for columns that are not makes a pushed string comparison answer under the database's collation rather than the engine's. Leaving it out is always safe.

Dotted-ref tables rely on the connector being able to introspect the catalog. The system catalog deliberately redacts URL/user/password. Non-JDBC connector types validate their own key sets.

Alternatives

A standalone `source ... from database` declaration when you only need one table and prefer to spell out its schema inline.

See Also

[source](#), [namespace](#), [import](#)

Notes

`collation` says whether this database compares strings the way `relix` does — `exact` (compare character for character) or `database` (whatever the column's collation says). The default is `exact` for every dialect but MySQL, whose default collation is case- and accent-insensitive. It only affects which comparisons may be pushed down and how they are written; the answer `relix` gives is the same either way. See [pushdown](#).

Same-connection joins and filters can be pushed down into the database as one SQL query; cross-connection work is combined in the engine (federation). A `mongodb` connection pushes

σ , μ , and cheap π/λ into an aggregation pipeline; τ and γ are evaluated in the engine over the rows the pipeline returns.

connection ... from gedcom (GEDCOM genealogy source)

Syntax

```
connection <Name> from gedcom { path: "<file.ged>" };

source <Name> from <connection> { table: "individuals" | "families",
                                   schema: { <col>: <TYPE>, ... } };

connection ancestry from gedcom { path: "family.ged" };
```

Description

GEDCOM is the interchange format every genealogy program reads and writes. A `.ged` file holds a whole family tree — people, families, events — and `relix` reads it as two relations: `individuals` and `families`.

It is a **connection** rather than a file source for one reason: a CSV file is one relation and a `.ged` file is several, and a connection is exactly the thing several tables share. You declare the file once and bind each table you want.

Technical Description

`path` names the `.ged` file, or `url` does: a `file:` URL, or an `https` one, which is fetched once per session and cached (see [connection](#) for the rules every file connection shares). A relative path resolves against the script's directory, as a `csv("...")` source's does. A gzip-compressed file is read as its content.

Two tables are available, and a `source ... from <connection>` binds one:

Table	Fields
<code>individuals</code>	<code>id</code> , <code>name</code> , <code>surname</code> , <code>sex</code> , <code>birth</code> , <code>death</code>
<code>families</code>	<code>id</code> , <code>husband</code> , <code>wife</code> , <code>children</code>

`birth` and `death` are events, each with `date`, `text` and `place`. `children` is an array of individual ids. `id`, `husband`, `wife` and each element of `children` are GEDCOM cross-reference ids with the delimiting `@` removed, so `@I1@` is `I1`.

A schema is optional. GEDCOM's shape is fixed by the format rather than by the file, so the connector describes both tables itself and a dotted reference (`ancestry.individuals`) resolves its columns with nothing declared. The heading it gives is the full one in the table above, typed and nested.

A declared schema decides the heading instead. Every column is filled by name, at every level, and a column the file has nothing for is NULL rather than an error. A schema naming only `id` and `name` gets two columns; a nested `birth: { place: STRING }` gets a one-field struct. Declare one to narrow a relation, to rename its columns, or to state what a query depends on — and note that the same rule is what makes a format as extensible as GEDCOM safe to declare a heading over at all: a file cannot widen a relation behind a

query's back.

A date is exposed twice, and the pair is the point. GEDCOM permits approximations and ranges (ABT 1900, BET 1900 AND 1910) as well as exact days, so `date` is a real `DATE` when the value names an exact day and `NULL` otherwise, while `text` always holds what the file said. Both `NULL` means no date was recorded; a `NULL` `date` beside a present `text` means the date is not exact. One column cannot tell those apart, and a genealogist cares which it is.

A family is a hyperedge — two parents and any number of children — so the binary (parent, child) edge the graph operators take is a projection over μ children (Families) rather than something the connector invents. Keeping the family row is what lets a query distinguish full siblings from half siblings, which a binary edge alone cannot.

Examples

Declare the file and bind both tables:

```
connection ancestry from gedcom { path: "family.ged" };

source Individuals from ancestry { table: "individuals",
  schema: { id: STRING, name: STRING, surname: STRING, sex: STRING,
    birth: { date: DATE, text: STRING, place: STRING } } };

source Families from ancestry { table: "families",
  schema: { id: STRING, husband: STRING, wife: STRING, children: [STRING] } };
```

The binary (parent, child) edge every graph operator takes is μ over the `children` array and a projection — a family is a hyperedge, so both parents contribute an edge to each child:

```
connection ancestry from gedcom { path: "family.ged" };

source Families from ancestry { table: "families",
  schema: { id: STRING, husband: STRING, wife: STRING, children: [STRING] } };

Parents := {
   $\pi$  husband  $\rightarrow$  parent, children  $\rightarrow$  child ( $\mu$  children (Families))
   $\cup$   $\pi$  wife  $\rightarrow$  parent, children  $\rightarrow$  child ( $\mu$  children (Families))
};

query { Parents };
```

Nothing declared at all — the connector supplies the heading:

```
connection ancestry from gedcom { path: "family.ged" };
query {  $\pi$  name, sex (ancestry.individuals) };
```

Only the columns a query needs:

```
connection ancestry from gedcom { path: "family.ged" };
source Names from ancestry { table: "individuals",
  schema: { id: STRING, name: STRING } };
query { Names };
```

Worked Example

The edge below is what the projection in the last example produces — one row per parent-child pair, a two-parent family contributing two. It is declared inline here so every query on this page runs as written:

```
Parents := [
| parent | child |
|-----|-----|
| I1      | I3      |
| I2      | I3      |
| I3      | I5      |
| I4      | I5      |
];

Individuals := [
| id | name      |
|----|-----|
| I1 | John Smith |
| I2 | Mary Jones |
| I3 | Alice Smith |
| I4 | Peter Brown |
| I5 | Robert Smith |
];
```

Every ancestor of every person, however many generations back, is one operator over that edge — the query SQL needs a recursive CTE for:

```
query { τ ancestor ASC, descendant ASC
        (p Ancestry(ancestor, descendant) (CLOSURE parent, child (Parents))) };
```

ancestor	descendant
I1	I3
I1	I5
I2	I3
I2	I5
I3	I5
I4	I5

(6 rows)

I1 and I2 reach I5 although neither is a parent of theirs: that is the grandparent relation, and nothing in the query mentions generations.

Naming the people rather than the ids is an ordinary join:

```
query {
  τ ancestor ASC, descendant ASC
  (π a.name → ancestor, d.name → descendant
   (p a (Individuals) ⋈ a.id = anc (
     p Named(anc, desc) (CLOSURE parent, child (Parents))
   ) ⋈ desc = d.id (p d (Individuals))))
};
```

ancestor	descendant
Alice Smith	Robert Smith
John Smith	Alice Smith
John Smith	Robert Smith
Mary Jones	Alice Smith
Mary Jones	Robert Smith
Peter Brown	Robert Smith

(6 rows)

How far apart two people are is `PATH` over the same edge read **undirected**, so a line of descent can be walked in either direction — which is what makes a sibling, a cousin or a spouse reachable at all, none of whom is anybody's ancestor:

```
query { τ parent ASC, child ASC
      (σ parent = "I3" (PATH parent ↔ child HOPS 1 TO 4 AS degrees (Parents))) };
```

parent	child	degrees
--------	-------	---------

I3	I1	1
----	----	---

I3	I2	1
----	----	---

I3	I3	2
----	----	---

I3	I4	2
----	----	---

I3	I5	1
----	----	---

(5 rows)

Both parents are one step away, the other parent of I3's child is two (up through the child and back down), and I3 reaches **itself** at two for the same reason — an undirected walk may return the way it came. A genealogical distance is a question about that graph, so the answer is the graph's rather than the question's.

Limitations

The path is resolved against the working directory rather than the script's, so a script moved to another directory needs an absolute path or a matching working directory.

Tags outside the two tables' fields are skipped rather than carried: notes, sources, repositories and vendor extensions are not exposed. A relation's heading is what decides the answer, and carrying everything a GEDCOM file may contain would mean carrying an open one.

A connection's name must not be a word the expression grammar already uses, or a dotted reference to it will not parse — `tree` is the `TREE` operator, so `connection tree ...` declares fine and `tree.individuals` does not parse:

```
connection tree from gedcom { path: "family.ged" };
query { tree.individuals };
```

Any name that is not an operator keyword is safe.

See Also

- `connection` — the declaration this one is a kind of
- `CLOSURE` — ancestors and descendants over the edge
- `PATH` — degrees of separation, with the distance
- `UNNEST` — the `μ` that turns `children` into rows

Notes

GEDCOM 5.5.1 is the version read. The reader is tolerant by design: a line it cannot place is skipped rather than refused, because a reader that rejects a file every genealogy program accepts is worse than one that ignores what it does not understand.

connection ... from log (web server access log source)

Syntax

```
connection <Name> from log { path: "<file or directory>" | url: "<https or file URL>",
                             format: "common" | "combined" | "<server format string>",
                             onError: "refuse" | "skip" };

source <Name> from <connection> { table: "<file>", schema: { <col>: <TYPE>, ... } };

connection logs from log { path: "/var/log/nginx", format: "combined" };
```

Description

An access log is a record of every request a web server answered, one line each. A log connection reads it as a relation with typed columns: the request time is a `TIMESTAMP`, the status and byte count are numbers, and the request line is split into `method`, `path` and `protocol`.

The typing is the point. Read as text, the time `10/Oct/2026:13:02:11 +0000` is a string no temporal function accepts, so every query would begin by converting it. Read through this connection it is already a timestamp, and the windowing operators (`SESSIONIZE`, `WINDOW`, `ROLLING`, `DOWNSAMPLE`, `TOP`) apply to it directly.

`clf` is another name for the `log` type.

Technical Description

`format` is `common`, `combined`, or a format string pasted from the server's own configuration: Apache's `LogFormat (%h %l %u %t "%r" %>s %b)` or nginx's `log_format ($remote_addr - $remote_user [$time_local] "$request" $status ...)`. It defaults to `common`. A string mixing the two vocabularies is refused, and so is a directive this connector does not read, rather than guessed at.

Apache	nginx	Column	Type
%h	\$remote_addr	host	STRING
%a		address	STRING
%l		ident	STRING
%u	\$remote_user	user	STRING
%t	\$time_local, \$time_iso8601	at	TIMESTAMP
%r	\$request	method, path, protocol	STRING
%m	\$request_method	method	STRING
%U	\$request_uri	path	STRING
%q		query	STRING
%H	\$server_protocol	protocol	STRING
%>s, %s	\$status	status	NUMBER
%b, %B	\$body_bytes_sent	bytes	NUMBER
	\$bytes_sent	bytes_sent	NUMBER

Apache	nginx	Column	Type
%D		duration_us	NUMBER
%T	\$request_time	duration_s	NUMBER
%v	\$host	server	STRING
%{Name}i	\$http_name	the header's name, lower case, - as _	STRING

combined is common followed by "%{Referer}i" "%{User-Agent}i", so it adds referer and agent: User-Agent is read as agent.

- **is NULL** wherever it appears, which is how a log writes "nothing": no user, no referer, no body.

The declared schema decides the heading. Every column is filled by name, and a column the format does not produce is NULL. A dotted reference with nothing declared gets every column the format produces, at the types above. A declared type different from a field's own converts the field's text: at: STRING keeps the time as written.

A directory holds several logs. When path names a directory, table names a file in it, so a live access.log and a rotated access.log.2.gz are two tables of one connection. When path names a file, table names that file. A gzip file is read transparently, recognised by its content rather than its name. url names a file to fetch over https, as for any file connection (see [connection](#)).

A relative path resolves against the script's directory, as a csv("...") source's does.

A line the format does not match is an error naming its line number, because a line dropped without notice makes every count over the log wrong. onError: "skip" leaves such lines out instead, and the number skipped is written to the application log when the file is closed. Blank lines are not records and are passed over.

A log is read a line at a time as the query asks for rows, so a large log is never held in memory whole.

Examples

Read an nginx server's live log and yesterday's rotated copy, with the format pasted from its configuration:

```
connection nginx from log {
    path:    "/var/log/nginx",
    format:  "$remote_addr - $remote_user [$time_local] \"$request\" $status $body_bytes_sent"
};
source Today    from nginx { table: "access.log",
    schema: { host: STRING, at: TIMESTAMP, path: STRING, status: NUMBER } };
source Yesterday from nginx { table: "access.log.1.gz",
    schema: { host: STRING, at: TIMESTAMP, path: STRING, status: NUMBER } };
query { γ status, COUNT(*) → hits (Today ∪ Yesterday) };
```

Worked Example

visits The examples on this page read one small log: 26 requests from three clients over two hours, in combined format. A line looks like this:

```
203.0.113.7 - - [10/Oct/2026:13:02:11 +0000] "GET /index.html HTTP/1.1" 200 5120 "-" "Mozilla/5.0 (X11; Linux x86_64) Firefox/131.0"
```

A visit is a run of requests from one client with no long silence in it. `SESSIONIZE` finds the silences by itself:

```
connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

Sessions := { SESSIONIZE at GAP DURATION 'PT30M' PER host AS session (Access) };
query { τ host ASC, session ASC (γ host, session, COUNT(*) → hits, MIN(at) → started, MAX(at) → ended (Sessions)) };
```

host	session	hits	started	ended
192.0.2.44	1	4	2026-10-10T13:07:01Z	2026-10-10T13:07:09Z
192.0.2.44	2	1	2026-10-10T14:52:00Z	2026-10-10T14:52:00Z
198.51.100.22	1	6	2026-10-10T13:03:40Z	2026-10-10T13:29:14Z
198.51.100.22	2	3	2026-10-10T14:47:31Z	2026-10-10T15:08:26Z
203.0.113.7	1	7	2026-10-10T13:02:11Z	2026-10-10T13:24:57Z
203.0.113.7	2	5	2026-10-10T14:41:02Z	2026-10-10T15:03:09Z

(6 rows)

Worked Example

a scanner The same operator, with a one-minute gap, is a burst detector. A client asking for four missing pages inside a minute is probing for software to attack:

```
connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

Misses := { σ status = 404 (Access) };
Bursts := { SESSIONIZE at GAP DURATION 'PT1M' PER host AS burst (Misses) };
query { σ misses ≥ 4 (γ host, burst, COUNT(*) → misses, MIN(at) → started (Bursts)) };
```

host	burst	misses	started
192.0.2.44	1	4	2026-10-10T13:07:01Z

(1 row)

The same client's single missing page an hour later is a burst of one, and is left out.

Worked Example

where visitors go Each client's previous request, found with `WINDOW LAG`, turns the log into the edges of a navigation graph. `PATH` then answers *what can a visitor reach from the front page, and in how many clicks*:

```
connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

Nav := { WINDOW LAG(path) SORT at ASC PER host AS from_path (Access) };
Steps := { σ from_path IS NOT NULL (Nav) };
Edges := { δ (π from_path → src, path → dst (Steps)) };
query { τ depth ASC, dst ASC (σ src = "/index.html" (PATH src, dst HOPS 1 TO 3 AS depth (Edges))) };
```

src	dst	depth
/index.html	/docs/faq	1
/index.html	/docs/guide	1
/index.html	/static/app.css	1
/index.html	/api/export	2


```

/index.html /api/report      2
/index.html /docs/intro     2
/index.html /index.html     2
/index.html /api/search     3
/index.html /docs/operators  3
(9 rows)

```

In SQL that is a window function feeding a recursive CTE.

Worked Example

think time How long each client paused before each request. The subtraction of two timestamps is a `DURATION`, with no conversion in the query:

```

connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

Gaps := { WINDOW LAG(at) SORT at ASC PER host AS prev_at (Access) };
query { τ at ASC (π at, path, (at - prev_at) → since_previous (σ host = "198.51.100.22" (Gaps))) };

```

at	path	since_previous
2026-10-10T13:03:40Z	/index.html	NULL
2026-10-10T13:03:41Z	/static/app.css	PT1S
2026-10-10T13:06:15Z	/docs/faq	PT2M34S
2026-10-10T13:12:03Z	/api/export	PT5M48S
2026-10-10T13:20:48Z	/api/report	PT8M45S
2026-10-10T13:29:14Z	/index.html	PT8M26S
2026-10-10T14:47:31Z	/docs/guide	PT1H18M17S
2026-10-10T14:58:02Z	/docs/intro	PT10M31S
2026-10-10T15:08:26Z	/docs/operators	PT10M24S

(9 rows)

The first request has no previous one, so its pause is NULL.

Worked Example

traffic over time Requests per quarter-hour, and a moving average over three of them:

```

connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

PerQuarter := { DOWNSAMPLE at BY '15m' USING COUNT (Access) };
query { τ bucket ASC (ROLLING AVG(count) OVER 3 ROWS SORT bucket ASC AS smoothed (PerQuarter)) };

```

bucket	count	smoothed
2026-10-10T13:00:00Z	12	12
2026-10-10T13:15:00Z	5	8.5
2026-10-10T14:30:00Z	2	6.3333333333
2026-10-10T14:45:00Z	5	4
2026-10-10T15:00:00Z	2	3

(5 rows)

Read the buckets before the average. `DOWNSAMPLE` emits no row for a quarter-hour with no requests, so nothing appears between 13:15 and 14:30, and `OVER 3 ROWS` averages three buckets that are *present* rather than three consecutive quarter-hours. The row for 14:30 averages 13:00, 13:15 and 14:30. That is what the query says, and it is not a moving

average over wall-clock time.

Worked Example

per client The heaviest single response each client received, and the bandwidth each has used so far at every request:

```
connection logs from log { path: "access.log", format: "combined" };
source Access from logs { table: "access.log",
  schema: { host: STRING, at: TIMESTAMP, method: STRING, path: STRING,
    status: NUMBER, bytes: NUMBER, referer: STRING, agent: STRING } };

query { τ host ASC (π host, path, bytes (TOP 1 bytes DESC PER host (Access))) };
```

host	path	bytes
192.0.2.44	/wp-login.php	196
198.51.100.22	/api/export	65536
203.0.113.7	/api/export	65536

(3 rows)

```
Running := { ROLLING SUM(bytes) OVER ALL ROWS SORT at ASC PER host AS used (Access) };
query { τ at ASC (π at, path, bytes, used (σ host = "198.51.100.22" (Running))) };
```

at	path	bytes	used
2026-10-10T13:03:40Z	/index.html	5120	5120
2026-10-10T13:03:41Z	/static/app.css	2048	7168
2026-10-10T13:06:15Z	/docs/faq	7168	14336
2026-10-10T13:12:03Z	/api/export	65536	79872
2026-10-10T13:20:48Z	/api/report	32768	112640
2026-10-10T13:29:14Z	/index.html	0	112640
2026-10-10T14:47:31Z	/docs/guide	18432	131072
2026-10-10T14:58:02Z	/docs/intro	9216	140288
2026-10-10T15:08:26Z	/docs/operators	24576	164864

(9 rows)

Limitations

Nothing is pushed down: a log file has no query language, so every operator runs in the engine over the rows the file yields. Only access logs are read. An application log in another shape (logback, logfmt, JSON lines) needs a connector of its own, or conversion to CSV first.

See Also

[connection](#), [SESSIONIZE](#), [WINDOW LAG](#), [ROLLING](#), [DOWNSAMPLE](#), [TOP](#), [csv source](#)

relate (schema relationship declaration)

Syntax

```
```relix
relate "Name" Source.col -> Target.col;
relate "Name" / "Inverse Name" Source.col -> Target.col;
relate symmetric "Name" Rel.col -> Rel.other_col;
relate "Name" Source(col_a, col_b) -> Target(col_a, col_b) [1..50];
```
```

As a ``references:`` block on a source (database, connection-table, csv, json), or a trailing ``references`` clause on an inline table:

```
```relix
source Orders from database { url: "${DB}", table: "orders",
 schema: { order_id: NUMBER, customer_id: NUMBER },
 references: { customer_id -> Customers.customer_id } };

Cities := [
| name | country |
|-----|-----|
| Chicago | US |
] references { country -> Countries.code };
```
```

Description

`relate` tells `relix` how two relations connect — which columns link them, what the relationship is called, and how many rows can match on each side. Declaring relationships builds a *schema graph* the engine can use to work out join paths mechanically: when a request touches columns from several tables, the engine can find the connecting path itself instead of guessing (or making you spell out every join). The `references:` block is the compact form for ordinary foreign keys, written directly on the source that holds the referencing column.

Names matter: with two links from `Issues` into `Users` ("assignee" and "reporter"), a named edge turns a silent wrong guess into an answerable question — "*join users as assignee or reporter?*".

Technical Description

A relationship is a named edge between two endpoints. Each endpoint is a relation plus an ordered column list with numeric multiplicity bounds:

- **Endpoints** — `Rel.col` names one column; `Rel(col_a, col_b)` names several (a composite key is *one* edge; columns pair positionally). In dot form the last dotted segment is the column, so a namespace-qualified relation uses the parenthesised form: `analytics.Orders(order_id)`.
- **Bounds** — `[min..max]` after an endpoint states how many rows of that endpoint's relation may match one row of the other side; `*` means unbounded. Omitted bounds default to `[0..*]`, the unconstrained reading — an omitted bound asserts nothing. `min = 0` signals that an inner join may drop opposite-side rows; a finite `max` bounds fan-out (a selectivity fact the cost model can use).

- **Inverse name** — `relate "Manages" / "Reports To" ...` names the reverse traversal; its presence records that the directions are distinct roles.
- **symmetric** — for self-referential edges only (cross-sells, friend graphs): the relationship has no direction at the data level. Mutually exclusive with an inverse name.
- **references:** — each `col -> Target.col` entry is a foreign-key arrow. The target side gets an implied `max` of 1; the edge's name defaults to the referencing column list (e.g. `"customer_id"`). Accepted on every schema-bearing source kind (database, connection-table, `csv`, `json`) and, as a trailing `references { ... }` clause, on inline tables. For an open `json` source the referencing columns resolve dynamically (schema-on-read); the target side is validated as usual. Views take no `references` — use a standalone `relate` (a view already *is* an expression; its relationships are declared, not embedded).

Every edge is validated at analysis time — both relations must resolve, every column must exist, and column lists must pair up — as hard errors, because an unchecked edge is a confidently wrong structural claim. Edges may target views (`ActiveOrders := { σ status = "active" (Orders) }`) exactly as they target tables, which is how a relationship carries a filter: put the selection in a view, not in the edge. Every edge is walkable in both directions regardless of which side declares it. Declared relationships appear in the IR report's RELATIONSHIPS section.

Examples

An order-processing schema, declared once so the engine knows the join paths:

```
source Customers from database { url: "${DB}", table: "customers",
  schema: { customer_id: NUMBER, name: STRING } };
source Orders from database { url: "${DB}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, status: STRING },
  references: { customer_id -> Customers.customer_id } };
source OrderItems from database { url: "${DB}", table: "order_items",
  schema: { order_id: NUMBER, product_id: NUMBER, qty: NUMBER } };

relate "Order Line Items" Orders.order_id -> OrderItems.order_id [1..50];
```

With this graph, a request touching `Customers.name` and `OrderItems.qty` has exactly one connecting path (`Customers ← Orders ← OrderItems`), so the engine can assemble the two joins itself. The `[1..50]` bound also tells it each order fans out to at most 50 items.

The two-role case — two foreign keys into the same table, disambiguated by name:

```
source Users from database { url: "${DB}", table: "users",
  schema: { id: NUMBER, name: STRING } };
source Issues from database { url: "${DB}", table: "issues",
  schema: { id: NUMBER, assignee_id: NUMBER, reporter_id: NUMBER } };

relate "Assignee" Issues.assignee_id -> Users.id [0..1];
relate "Reporter" Issues.reporter_id -> Users.id [1..1];
```

A request naming a user column now has two *named* paths, so the ambiguity is askable ("assignee or reporter?"). The `[0..1]` on `Assignee` additionally records that unassigned issues exist — an inner join through it would drop them.

An org hierarchy (one self-edge, two role names), a symmetric self-edge, and a composite multi-tenant foreign key (one edge, never two):

```

source Employees from database { url: "${DB}", table: "employees",
  schema: { id: NUMBER, manager_id: NUMBER } };
source Products from database { url: "${DB}", table: "products",
  schema: { id: NUMBER, related_id: NUMBER } };
source WarehouseSlot from database { url: "${DB}", table: "slots",
  schema: { tenant_id: NUMBER, product_id: NUMBER } };
source Product from database { url: "${DB}", table: "product",
  schema: { tenant_id: NUMBER, id: NUMBER } };

relate "Manages" / "Reports To" Employees.id -> Employees.manager_id;
relate symmetric "Cross Sells" Products.id -> Products.related_id;
relate "Slot Product"
  WarehouseSlot(tenant_id, product_id) -> Product(tenant_id, id);

```

A relationship to a *view* — the filter lives in the view, never in the edge:

```

source Customers from database { url: "${DB}", table: "customers",
  schema: { customer_id: NUMBER, name: STRING } };
source Orders from database { url: "${DB}", table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, status: STRING } };

ActiveOrders := { σ status = "active" (Orders) };
relate "Customer Active Orders"
  Customers.customer_id -> ActiveOrders.customer_id;

```

Bounds inheritance across a filtering view

When an edge targets a view that only *filters* a base relation (a σ chain over a single relation), the engine **derives** the derived endpoint's bounds from the base relationship instead of trusting the author. A σ can only remove rows, so `max` is preserved and `min` relaxes to 0: given `Customers -[1..10]→ Orders`, the edge to `ActiveOrders` above is read as `Customers -[0..10]→ ActiveOrders` — a customer with at least one order may have zero *active* ones, but never more than the ten they started with. Any bounds written on such an edge are ignored (with a warning), because an inherited `min > 0` would be a false claim about row preservation.

Inheritance applies only through a filter and only when a base relationship exists to inherit from. Where the view aggregates (γ) or joins rather than filters, or the base relation carries no matching edge, no rule applies and the edge keeps its declared — or `[0..*]` default — bounds.

A derived (view) endpoint also enters automatic path resolution only when the request *nominates* it (names or matches the view); the base relation is the default. Left free, every filtered view would double the parallel paths through its region — so a view is opt-in vocabulary, not a graph-wide multiplier.

Limitations

Relationships describe equijoin structure only; range and temporal relationships (see the as-of and interval joins) are not graph edges. A many-to-many pair through a junction table is declared as two edges (one per foreign key), not one. Bounds are declarations, not enforced constraints — the engine trusts them for planning and explanation but does not verify data against them. Join-path *resolution* over the graph (finding and correcting the join a query should build) is implemented on top of these declarations.

A relationship's name, and its inverse name, must hold more than whitespace. A blank one is a parse error:

```
relate "" Orders.customer_id -> Customers.customer_id;
```

Alternatives

The `references:` block on a `source` declaration for plain foreign keys — same edge, terser, named after the referencing column. Without any declarations, the engine still executes whatever joins you write; the graph adds names, bounds, and the basis for automatic path resolution.

See Also

[source](#), [connection](#), [natural join](#)

Notes

Declared edges are ground truth (origin `DECLARED`). The graph also admits session-learned edges (origin `LEARNED`, e.g. supplied conversationally) and data-inferred suggestions (`INFERRED`); non-declared edges stay visibly tagged in the IR report and are dropped with a warning — never a hard error — if the session's schema moves out from under them.

Inline Table (markdown / csv literal)

Syntax

```
-- Markdown form:
<Name> := [
col1	col2
v1	v2
];

-- CSV form:
<Name> := csv[
    col1, col2
    v1, v2
];
```

Description

An inline table embeds small reference data directly in the script — lookup tables, status-code mappings, region lists — without needing an external file or database. Two formats are available: a markdown table (easy to read) and a csv block (for values containing commas or pipes). Column names come from the header row.

Technical Description

An inline table binds an `InlineRelationSymbol`. In the markdown form, column names come from the header row and all values are strings unless a column contains only numbers (which infers `NUMBER`). In the csv form, the first non-blank line is the header and fields with commas are quoted. Use `csv[ ]` when values contain commas or pipe characters.

An inline table may declare foreign-key relationships with a trailing `references { col -> Target.col }` clause — the inline-table counterpart of the `references:` field on source declarations. Each entry becomes a named edge of the schema graph, validated at analysis time. See [relate](#) for the full relationship model.

Examples

A markdown reference table:

```
Regions := [
region	country
EMEA	United Kingdom
APAC	Australia
];
```

Numeric column infers NUMBER:

```
Prices := [
item	price
Pen	1.99
Book	12.50
];
```

CSV form for values containing commas:

```
Cities := csv[
    name, country
    "London, UK", GB
    "Paris, France", FR
];
```

An inline table that declares how it joins to a live source (each office row references at most one country — the FK arrow points at the referenced side):

```
source Countries from database { url: "${DB}", table: "countries",
    schema: { code: STRING, name: STRING } };

Offices := [
city	country
London	GB
Paris	FR
] references { country -> Countries.code };
```

Limitations

Intended for small, static reference data — not large datasets (use a source for those). Type inference is limited (numbers vs strings); there is no explicit schema clause on inline tables.

Alternatives

`source ... from csv("path")` for external files; `source ... from database` for live tables.

See Also

[source](#), [relate](#), [assignment](#), [natural-join](#)

Notes

Inline tables join cleanly against live sources — a handy way to attach a small mapping (e.g. status codes → labels) to query results.

:= and query (views and result queries)

Syntax

```
<Name> := { <relational expression> };      -- define a named view
query { <relational expression> };          -- an anonymous result query
query <Name>;                               -- emit a named view as a result

CustomerTotals := { γ customer_id, SUM(amount) → total (Orders) };
query { π customer_id, total (CustomerTotals) };
```

Description

The `:=` assignment names an intermediate result (a "view") so you can build a query up in readable steps and reuse a sub-result by name. A `query` statement marks what the script should actually produce — either an inline expression or a previously-named view. Together they let you write a pipeline as a series of named stages ending in one or more results.

Technical Description

`Name := { body }` binds a `QueryRelationSymbol` whose `body()` is a `RelNode`; it is a logical view, inlined by the optimizer/planner at use sites (INLINE-001) so pushdown crosses former view boundaries. `query { expr }` registers a root query (an `ExpressionQueryTarget`, auto-named `query[n]`); `query Name;` registers a `NamedQueryTarget`. The `SemanticModel`'s `rootQueries` drive execution, optimisation, and IR generation.

Examples

Build in stages and query the last:

```
OrderAmounts := { π order_id, amount (Orders) };
CustomerTotals := { γ customer_id, SUM(amount) → total (Orders) };
query CustomerTotals;
```

Anonymous inline result:

```
query { λ 10 (τ amount DESC (Orders)) };
```

Reuse a view in two places:

```
Active := { σ active = true (Customers) };
query { Active ⋈ Orders };
```

Limitations

A view is a logical definition, not a materialised table — it is recomputed (inlined) wherever used. A script with no `query` statement produces no output.

Outside a view, its columns answer to the view's name, not to the relations inside it. When the view reads a schema-on-read source (JSON, HTTP, MongoDB), a reference that still uses an inner relation's name is not refused. It yields NULL instead; see [theta-join](#) for a worked case.

Alternatives

A table-valued function (`def ... : RELATION`) is a view that takes parameters.

See Also

[def-relation](#), [project](#), [group](#)

Notes

Views are inlined before optimisation, so splitting a query into named views costs nothing at runtime and aids readability.

Delimited Identifier (``name``)

Syntax

```
`name` -- any name, wrapped in backticks
`order` -- the relation/column named "order", not the ORDER operator
`my table` -- a name containing spaces
`a``b` -- a name containing a literal backtick (doubled to escape)
```

Description

Relix reserves a fixed set of operator and keyword words — `SELECT`, `SORT`, `ORDER`, `GROUP`, `JOIN`, `UNION`, `IS`, `IN`, and so on (the full set is the lexer keyword table; `relix help lang operators` prints it). Written bare in a relation position, such a word is read as the operator, so a relation or view that happens to be named after one cannot be referenced directly:

```
π name (order) -- ERROR: 'order' is read as the ORDER/τ operator
```

A **delimited identifier** — the name wrapped in backticks — tells the lexer to treat the text as a plain name, never an operator or keyword:

```
π name (`order`) -- OK: 'order' is the relation named "order"
```

Backticks also let a name contain characters an ordinary identifier can't (spaces, punctuation): ``line total`, `2024 q1``. A literal backtick inside the name is written by doubling it: ``ab ```` is the name ```ab```.

This is the same escape hatch SQL provides with double-quotes (ANSI) or backticks (MySQL). Relix uses backticks because single and double quotes are already string literals.

Technical Description

The lexer emits a delimited identifier as an ordinary `IDENTIFIER` token whose value is the decoded text (backticks stripped, doubled backticks collapsed), so every downstream consumer treats it exactly like a normal name — there is no distinct AST node and no semantic difference from an unreserved bare name. An unterminated identifier (end of input or a newline before the closing backtick) or an empty ```` is a parse error, and so is one that holds only whitespace. An empty name:

```
query { π `` (Orders) };
```

A name of only spaces:

```
query { π ` ` (Orders) };
```

A name that reaches the end of the line before its closing backtick:

```
query { π `order
id` (Orders) };
```

The pretty-printer performs the inverse: when it prints a **relation** or **column** reference whose name collides with a reserved word (or is not identifier-shaped), it re-adds the backticks, so a printed tree always parses back to the identical tree. Operator-introduced parameter names (the columns after `AS`, `VIA`, `BY`, `PER`, `HOPS`, a rename column list, a projection or aggregate alias) are read leniently by the parser and are printed bare.

Examples

A view named after the RA operator word `order` — **defined bare** (a name is a plain identifier on the left of `:=`; only the operator words are reserved, and only inside an expression) and **referenced with backticks** downstream:

```
order := { σ status = "open" (RawOrders) };
Open  := { π id, total (`order`) };
query { Open };
```

Join two relations, one named after a reserved word (`select`), and filter on a column whose name also collides (`order`):

<!-- analysis-skip: a relation named with a delimited identifier can only come from a catalog, so this example cannot declare what it references -->

```
Recent := { `select` ⋈ Users };
query { σ `order` = 1 (Events) };
```

A qualified column reference delimits only the reserved segment:

```
σ `order`.total > 0 (Sales)      -- 'order' the table, 'total' an ordinary column
```

Limitations

- The backtick delimiter is fixed; there is no alternate quoting character.
- A name may not span a line — the closing backtick must be on the same line.
- A single delimited name that itself contains a dot (``a.b``) is stored as the bare string `a.b`, indistinguishable from a qualified reference, so it re-renders as two segments (``a`.`b``). Such names are vanishingly rare.
- Backticks are recognised inside relational-algebra expressions (view bodies, `query { ... }`). A **statement-level** name — the left side of `:=` or a `source / def` declaration — is a bare identifier and does not (yet) accept backticks, so a name that collides with a *statement* keyword (`source`, `query`, `import`, `namespace`, ...) cannot be used there. The RA operator words (`order`, `select`, `sort`, ...) are not statement keywords, so they work bare on the left of `:=` and only need backticks where they are *referenced*.

Alternatives

Prefer a non-reserved name when you control the schema — delimiting is for names you don't control (imported schemas, external tables) or that genuinely need unusual characters.

See Also

[assignment & query](#), [source](#), [reference index](#)

Notes

Delimiting a name that does **not** need it is harmless and a no-op on output — ``Users`` parses to exactly the same relation as `Users`.

Comments (-- and / ... /)

Syntax

```
-- a line comment, to the end of the line
/* a block comment,
   which may span lines */

σ amount > 100 (Orders)  -- keep the large orders
```

Description

A comment is text the engine ignores. Relix uses the same two forms SQL does: `--` runs to the end of the line, and `/* ... */` covers everything between the two markers, however many lines that is.

Both forms work anywhere whitespace does — before a statement, between the operators of an expression, at the end of a line. A script and the algebra inside its braces are commented the same way, so there is nothing to remember about where you are when you write one.

`//` is not a comment. It is two division operators, and it is a syntax error wherever it appears, after a statement:

```
query { Orders }; // every order
```

and inside braces:

```
query { Orders // every order
};
```

Technical Description

Comments are removed by the lexer, so they are invisible to every phase above it. They carry no source position into the tree, appear in no rendering of an expression, and cannot be recovered from a parsed script: a view printed back with `:source` or by a session's own report comes back without them.

A `--` is a comment marker before it is an operator, which is the one place the two forms interact with the rest of the grammar. `- -amount` is arithmetic — negation applied twice — while `--amount` opens a comment and swallows the rest of the line. The space is what tells them apart, and this is the rule SQL follows for the same reason.

An unterminated `/*` runs to the end of the input rather than raising: the text after it was never going to be code.

Examples

Explain a filter at the point it is applied:

```
-- Large orders only: everything below assumes they are the interesting ones.
BigOrders := { σ amount > 100 (Orders) };
```

Comment inside the algebra, not only around it:

```
Recent := {
  σ status = "shipped" (      -- shipped, not merely placed
    σ amount > 50 (Orders) -- and worth looking at
  )
};
```

Take a step out of a pipeline without deleting it:

```
Totals := {
  γ customer_id, SUM(amount) → total (
    /* σ region = "east" ( */
      Orders
    /* ) */
  )
};
```

Head a script with what it is for:

```
/*
  Customer revenue by region.
  Reads Orders and Customers; nothing here writes anything.
*/
ByRegion := { γ region, SUM(amount) → revenue (Orders) };
```

Limitations

A comment is discarded, not attached, so it cannot be used to annotate a view for anything that reads the model back. A name and a `--` line above the definition is how a script explains itself to the next reader; the engine's own reports describe the expression, not the prose around it.

There is no nesting: the first `*/` closes a block comment, whatever came before it. A block comment wrapped around a region that already contains one therefore ends early, and the remainder of the outer comment is read as code.

See Also

- [assignment & query](#) — naming a view, which is the other half of making a script readable
- [delimited identifier](#) — the other lexical rule worth knowing before writing a script

Operator spellings (Unicode and ASCII)

Syntax

```
σ amount > 100 (Orders)      -- Unicode
SELECT amount > 100 (Orders) -- ASCII, identical meaning

A ⋈ B
A JOIN B
```

Description

Every operator in Relix has two spellings: a Unicode glyph and an ASCII form. They are accepted in exactly the same places and mean exactly the same thing — the two forms parse to the same tree, so nothing downstream can tell which one you typed.

Use whichever suits where you are typing. The glyphs read like the algebra they come from and are what the engine prints back to you; the ASCII forms need no special keyboard and survive a terminal, an email, or a code review comment intact. A single expression may mix them freely.

Unary operators

| Unicode | ASCII keyword | Operation |
|-----------|---------------|--|
| π | PROJECT | Projection |
| σ | SELECT | Selection |
| ρ | RENAME | Rename |
| γ | GROUP | Aggregation |
| τ | SORT | Sort |
| λ | LIMIT | Limit |
| δ | DISTINCT | Distinct |
| μ | UNNEST | Unnest (array→rows) |
| $\forall$ | FORALL | Universal quantification ($\forall$ keys : P (R)) |

Joins

| Unicode | ASCII | Operation |
|-----------|-----------|------------------|
| $\bowtie$ | JOIN | Natural join |
| $\bowtie$ | $\bowtie$ | Theta join |
| $\bowtie$ | $\bowtie$ | Left outer join |
| $\bowtie$ | $\bowtie$ | Right outer join |
| $\bowtie$ | $\bowtie$ | Full outer join |
| $\ltimes$ | SEMI | Semi-join |
| $\rhd$ | ANTI | Anti-join |

Set operations

| Unicode | ASCII | Operation |
|----------|---------|-----------------------------------|
| $\times$ | CROSS | Cartesian product |
| $\cup$ | UNION | Union |
| $\cup$ | UALL | Union all |
| $\sqcup$ | OUNION | Outer-union (heterogeneous merge) |
| $-$ | DIFF | Set difference |
| Δ | SYMDIFF | Symmetric difference |
| $\cap$ | INTER | Intersection |

| Unicode | ASCII | Operation |
|---------|---------|-------------|
| ÷ | DIV | Division |
| ◦ | COMPOSE | Composition |

Logical and comparison operators

| Unicode | ASCII | Operation |
|---------|-------|------------------|
| ∧ | AND | Logical and |
| ∨ | OR | Logical or |
| ¬ | NOT | Logical not |
| ≠ | != | Not equal |
| ≤ | <= | Less or equal |
| ≥ | >= | Greater or equal |

Special forms

| Unicode | ASCII | Operation |
|---------|--------|--|
| → | -> | Alias arrow (aggregation/projection) |
| ↔ | <-> | Undirected edge, between a graph operator's two endpoint columns |
| ⊥ | NULL | Null check |
| ∈ | IN | Element of |
| ∉ | NOT IN | Not element of (two tokens) |

Nullary relation literals

The two relations with the empty (zero-column) heading, written where a relation goes. They have no glyph: like `TRUE` and `FALSE` they are literals rather than operators.

| Keyword | Alias | Meaning |
|---------|-------|---|
| UNIT | DEE | Empty heading, one (empty) tuple — <code>TRUE</code> ; the identity of <code>x</code> |
| EMPTY | DUM | Empty heading, no tuple — <code>FALSE</code> |

SQL-prior aliases

Extra synonyms for the canonical forms above, for readers arriving from SQL. They parse to the identical tree; the canonical form is what gets printed back.

| SQL-prior form | Canonical form | AST |
|---|--|------------------|
| INTERSECT | INTER / <code>n</code> | IntersectionNode |
| MINUS, EXCEPT | DIFF / <code>-</code> | DifferenceNode |
| <code>x IS NULL</code> / <code>x IS NOT NULL</code> | <code>x = NULL</code> / <code>x ≠ NULL</code> (<code>⊥</code>) | Null predicate |

| SQL-prior form | Canonical form | AST |
|------------------------|---|------------------------------------|
| ORDER, ORDER BY | <code>Sort / τ</code> | SortNode |
| GROUP BY | <code>GROUP / γ</code> | AggregationNode |
| LJOIN / RJJOIN / FJOIN | <code> >< / >< / >< </code> | outer joins |
| COUNT(*) | <code>COUNT(1)</code> | AggregateFunction(COUNT, number 1) |

BY is an optional noise word after ORDER and GROUP. The MINUS keyword is set difference while the arithmetic - character is unchanged, so a - b and A MINUS B never collide.

COUNT(*) is SQL's row-count spelling: a literal argument is never NULL, so counting it counts every row, where COUNT(expr) counts non-NULL values. * is special **only** as the sole argument to COUNT — SUM(*) is a syntax error and COUNT(a * b) is still multiplication.

Technical Description

The choice of spelling is made in the lexer and is gone by the time a tree exists: σ and SELECT produce the same token, so no phase above the lexer — inference, validation, optimisation, planning, execution — can distinguish them. Two scripts differing only in spelling optimise to the same plan and push down the same query.

Printing goes one way only. A tree rendered back to text — by a session's own report, or by the pretty-printer behind it — comes out in the **canonical** form: the glyph for an operator that has one, and the canonical keyword for an alias. A script written in ASCII and printed back therefore comes back in Unicode, and a script written with INTERSECT comes back with ∩. The round trip preserves the tree, not the typing.

The ASCII forms are ordinary keywords, which is why a relation or column whose name collides with one needs backticks — see the delimited identifier page. The glyphs are not keywords and never collide, so σ is always the operator and a relation may be called select as long as it is written `select`.

Examples

The same filter, both ways:

```
Adults := { σ age > 18 (Users) };
```

```
AdultsAscii := { SELECT age > 18 (Users) };
```

A natural join, both ways:

```
Joined := { A ⋈ B };
```

```
JoinedAscii := { A JOIN B };
```

Mixing the two in one expression is allowed, and is what most scripts end up doing — the glyph for the operator, a word for the connective:

```
ActiveAdults := { SELECT age > 18 ∧ active = "yes" (Users) };
```

An outer join written in ASCII, with an explicit condition:

```
Everyone := { Users |><| Users.id = Orders.user_id Orders };
```

The SQL-prior aliases, which read naturally to someone arriving from SQL:

```
ByRegion := { GROUP BY region, SUM(amount) → revenue (Orders) };
```

```
Placed := { σ shipped_at IS NULL (Orders) };
```

Limitations

⌘ is two tokens in its ASCII form (NOT IN) where the glyph is one. This makes no difference to what parses, but it is the one row of the table where the ASCII column is not a single word.

There is no ASCII spelling that is *only* ASCII: every form in the left column of these tables is accepted everywhere, so a script cannot be made to reject glyphs. A team that wants ASCII-only sources enforces that in review, not in the grammar.

Aggregate names (SUM, COUNT, MIN, ...) have no glyph. They are function names rather than operators, so they are written the same way whichever spelling the surrounding expression uses.

See Also

- [grammar \(EBNF\)](#) — every spelling on this page in its place in the whole language, one rule at a time
- [comments](#) — the other lexical rule worth knowing before writing a script, and the one place `//` will surprise you
- [delimited identifier](#) — backticking a name that collides with one of the ASCII keywords above
- [truth relations](#) — the full page for `UNIT` and `EMPTY`

Grammar (EBNF)

Syntax

```
rule      ::= alternative | alternative      -- a production
"SELECT"  -- a keyword or symbol, written literally
name?    name*   name+   ( ... )           -- optional, zero or more, one or more, group
WORD - reserved_word                       -- a WORD that is not a reserved word
```

Description

This page is the whole grammar of a `.relix` script in one place, from the statements at the top of a file down to a single character of a string. The other pages in this manual explain what each construct *means*; this one says exactly what you may *write*, which makes it the page to read when you are generating Relix rather than reading it: a code generator, a language model, or a person writing a parser for an editor.

The grammar is written in the W3C style of EBNF, the notation the XML and XPath specifications use:

| Notation | Meaning |
|---------------------------|--|
| <code>rule ::= ...</code> | defines <code>rule</code> |
| <code>"SELECT"</code> | the text itself; a word matches in any letter case (<code>select</code> , <code>Select</code>), a symbol matches exactly |
| <code>a b</code> | <code>a</code> followed by <code>b</code> |

| Notation | Meaning |
|-------------------------|---|
| <code>a b</code> | a or b |
| <code>a?</code> | a or nothing |
| <code>a*</code> | zero or more a |
| <code>a+</code> | one or more a |
| <code>(...)</code> | grouping |
| <code>a - b</code> | an a that is not also a b |
| <code>UPPER_CASE</code> | a token class, defined in <i>Tokens</i> at the end of the grammar |
| <code>/* ... */</code> | a comment |

Every operator has a Unicode glyph and an ASCII spelling, and the grammar lists both wherever they are accepted, glyph first: `( "σ" | "SELECT" )`. The two spellings parse to the same tree and may be mixed freely in one expression. The [operator spellings](#) page has the same pairs as a table.

A script has two layers, and the grammar keeps them apart because their tokens differ slightly. The **statement layer** is what sits outside braces: `source`, `:=`, `query` and the other statements. The **expression layer** is what sits inside `{ ... }`, the relational algebra itself. For example, a statement-layer string must use double quotes, while an expression-layer string may use either kind.

The grammar is also published as a single plain-text file, [grammar.ebnf.txt](#), for tools that want the rules without the page around them. It holds the blocks below, in order, followed by the rules EBNF cannot state.

Grammar

Scripts and statements

```
/* A script: an optional namespace, then an optional env, then statements.
   Every statement ends in ";". */
script      ::= namespace_decl? env_decl? statement*

namespace_decl ::= "namespace" script_name ";"
env_decl      ::= "env" ( "from" STRING_DQ )? ( "using" STRING_DQ )? ";"

/* "private" hides a declaration from scripts that import this one. */
statement    ::= "private"? ( import_stmt
                             | source_decl
                             | connection_decl
                             | relate_stmt
                             | def_stmt
                             | query_stmt
                             | assignment )

/* Statement-layer names are plain words; backticks are not accepted here. */
script_name  ::= WORD

/* import "lib.relix";           everything the file exports
   import relation Totals from "lib.relix"; one name, of a stated kind
   import { A, B } from "lib.relix";      several names
   import Totals from "lib.relix";        one name */
import_stmt  ::= "import" NONBLANK_STRING_DQ ";"
               | "import" ( "source" | "relation" | "function" ) script_name
               | "from" NONBLANK_STRING_DQ ";"
               | "import" "{" script_name ( "," script_name )* "," "?" "}"
               | "from" NONBLANK_STRING_DQ ";"
               | "import" script_name "from" NONBLANK_STRING_DQ ";"

/* Name := { expression }; a view
   Name := [ | a | b | ... ]; an inline table (see Inline tables) */
assignment  ::= script_name "!=" assignment_body ";"
assignment_body ::= "{" rel_expr "}"
               | inline_table ( "references" references_block )?

/* query Name; or query { expression }; - marks a result to produce.
   The bare form takes one plain name: write query { relix.plan }; for a
   dotted one. */
query_stmt  ::= "query" ( script_name | "{" rel_expr "}" ) ";"

/* def f(x: NUMBER): NUMBER := { x * 2 }; a scalar function
   def recent(n: NUMBER): relation := { λ n (...) }; a table-valued function
   def srcs(E: RELATION(src, w: NUMBER)): relation := { π src (E) };
                                   a relation parameter, which
   only a table-valued function may declare */
def_stmt    ::= "def" script_name
               ( "(" ( scalar_param ( "," scalar_param )* )? ")" ":"
                 scalar_type "!=" "{" operand "}"
                 | "(" ( parameter ( "," parameter )* )? ")" ":"
                 "relation" "!=" "{" rel_expr "}" ) ";"
scalar_param ::= script_name ":" scalar_type
parameter    ::= scalar_param | script_name ":" relation_type
relation_type ::= "RELATION" "(" column_name ( ":" type )? ( "," column_name ( ":" type )? )* ")"

/* relate "places" / "placed by" Orders.customer_id [0..*] -> Customers.customer_id [1..1]; */
relate_stmt  ::= "relate" "symmetric"? NONBLANK_STRING_DQ ( "/" NONBLANK_STRING_DQ )?
               endpoint "->" endpoint ";"
endpoint     ::= relation_columns multiplicity?
multiplicity ::= "[" INTEGER ".." ( INTEGER | "*" ) "]"

/* Rel.col or Rel(col, ...); the relation may itself be dotted. */
relation_columns ::= script_name ( "." script_name )* "(" script_name ( "," script_name )* ")"
               | script_name ( "." script_name )+
```

```
/* { customer_id -> Customers.customer_id, (a, b) -> T(x, y) } */
references_block ::= "{" ( reference "," "?" ) * "}"
reference        ::= ( script_name | "(" script_name ( "," script_name ) * ")" )
                  "->" relation_columns
```

Sources and connections

```
source_decl      ::= "source" script_name "from" source_config ";"
source_config    ::= csv_source
                  | json_source
                  | http_source
                  | database_source
                  | generator_source
                  | connection_table_source

/* Inside a { field: value, ... } block the fields may come in any order,
   commas between them are optional, and a trailing comma is allowed.
   Which fields are required is noted per source; a missing one is an error. */

/* source Orders from csv("orders.csv") { schema: { id: NUMBER } }; schema required */
csv_source       ::= "csv" "(" STRING_DQ ")" "{" ( csv_field ( ","? csv_field )* ","? )? "}"
csv_field        ::= "header" ":" boolean_literal
                  | "schema" ":" schema_block
                  | "references" ":" references_block

/* source Docs from json("docs.json") { records: "$.items" }; the block is optional */
json_source      ::= "json" "(" STRING_DQ ")"
                  ( "{" ( json_field ( ","? json_field )* ","? )? "}" )?
json_field       ::= "records" ":" STRING_DQ
                  | "references" ":" references_block

/* url, table and schema required */
database_source  ::= "database" "{" ( database_field ( ","? database_field )* ","? )? "}"
database_field   ::= "url" ":" STRING_DQ
                  | "table" ":" STRING_DQ
                  | "schema" ":" schema_block
                  | "references" ":" references_block

/* source Nums from generator { name: "Range", lo: "1", hi: "10" };
   name required; every other field is an argument to the generator. */
generator_source ::= "generator" "{" ( generator_field ( ","? generator_field )* ","? )? "}"
generator_field  ::= script_name ":" STRING_DQ

/* A table on a named connection: source T from warehouse { table: "t", schema: {...} };
   table and schema required. */
connection_table_source ::= connection_name "{" ( table_field ( ","? table_field )* ","? )? "}"
connection_name  ::= script_name - ( "http" | "database" | "csv" | "json" | "generator" )
table_field      ::= "table" ":" STRING_DQ
                  | "schema" ":" schema_block
                  | "references" ":" references_block

/* url required */
http_source      ::= "http" "{" ( http_field ( ","? http_field )* ","? )? "}"
http_field       ::= "url" ":" STRING_DQ
                  | "method" ":" ( "GET" | "POST" | "QUERY" )
                  | "headers" ":" "{" ( STRING_DQ ":" STRING_DQ ","? )* "}"
                  | "auth" ":" auth_spec
                  | "body" ":" STRING_DQ
                  | "extract" ":" extract_spec
                  | "paginate" ":" "{" ( paginate_entry ","? )* "}"
                  | "schema" ":" http_schema_block
auth_spec        ::= "bearer" "(" STRING_DQ ")"
                  | "basic" "(" STRING_DQ "," STRING_DQ ")"
                  | "apikey" "(" ( "query" "(" STRING_DQ ")" | STRING_DQ ) "," STRING_DQ ")"
extract_spec     ::= "json" "(" STRING_DQ ")"
                  | "csv" "(" ( "header" ":" boolean_literal )? ")"
paginate_entry   ::= script_name ":" "query" "(" STRING_DQ ")"
                  ( "[" "default" ":" INTEGER "]" )?

/* connection wh from jdbc { url: "${WH_URL}", user: "app" };
   "database" is an alias for "jdbc", which accepts only the keys url, user,
   password, dialect and collation, and requires url. Other connector types
```

```

(mongodb, gedcom, ...) accept any keys and check them themselves. */
connection_decl ::= "connection" script_name "from" connector_type
                  "{" ( script_name ":" ( STRING_DQ | WORD ) ","? )* "}" ";";
connector_type  ::= WORD

/* A declared heading: { id: NUMBER, addr: { city: STRING }, tags: [STRING] } */
schema_block    ::= "{" ( column_spec ","? )* "}"
http_schema_block ::= "{" ( http_column_spec ","? )* "}"
column_spec     ::= column_name ":" type column_binding? column_modifier?
/* An HTTP column may be an input ("in": a request parameter) or an output ("out", the default). */
http_column_spec ::= column_name ":" ( "in" | "out" )? type column_binding? column_modifier?
column_name      ::= WORD | DELIMITED_IDENTIFIER
column_binding   ::= "as" ( "query" | "path" | "header" ) "(" STRING_DQ ")"
                  | "at" STRING_DQ
column_modifier  ::= "[" ( "required" | "default" ":" STRING_DQ ) "]"

/* A column may be nested – one of a relation parameter's too; a scalar def parameter
   or return type may not. */
type             ::= scalar_type
                  | "{" ( column_name ":" type ","? )+ "}"
                  | "[" type "]"
scalar_type      ::= "NUMBER" | "STRING" | "BOOLEAN" | "ANY"
                  | "DATE" | "TIME" | "TIMESTAMP" | "DURATION"
boolean_literal  ::= "true" | "false"

```

Inline tables

/* An inline table is read as raw text between the brackets, line by line, not as tokens. Two layouts:

```

Markdown:  Cities := [
              | city | country |
              |-----|-----|
              | Paris | FR      |
            ];

```

A line that does not start with "|" is ignored, a row whose cells are all dashes is ignored, and the first remaining row is the header.

```

CSV:       Cities := csv[
              city,country
              Paris,FR
            ];

```

Blank lines are ignored and the first line is the header. A field may be double-quoted, with "" for a quote inside it.

Cells are trimmed. A cell's type is inferred: a column whose every value is a number is NUMBER, anything else is STRING. */

```

inline_table ::= MARKDOWN_TABLE
              | "csv" CSV_TABLE

```

Relational expressions

```

/* What goes inside { ... }. Binary operators associate to the left, and bind in
   three tiers, tightest first:
   1. joins, ×, ÷, ° and LATERAL
   2. n
   3. U, U, U, -, Δ
   so A ⋈ B U C n D is (A ⋈ B) U (C n D). Parenthesise when in doubt. */
rel_expr      ::= intersect_expr ( union_tier_op intersect_expr )*
intersect_expr ::= join_expr ( intersect_op join_expr )*
join_expr     ::= postfix_expr join_tail*

union_tier_op  ::= "U" | "UNION"
                | "U" | "UALL"
                | "U" | "OUNION"
                | "-" | "DIFF" | "MINUS" | "EXCEPT" /* "-" is U+2212, not the ASCII hyphen */
                | "Δ" | "SYMDIFF"
intersect_op   ::= "n" | "INTER" | "INTERSECT"

/* A join's condition is written BETWEEN the operator and its right input:
   Users ⋈ Users.id = Orders.user_id Orders */
join_tail     ::= natural_join_op postfix_expr
                | conditional_join_op predicate postfix_expr
                | asof_join
                | interval_join
                | product_op postfix_expr
                | lateral_join

natural_join_op ::= "⋈" | "JOIN"
conditional_join_op ::= "⋈" | "><" /* theta join */
                | "⋈" | "><" | "LJOIN" /* left outer */
                | "⋈" | "><" | "RJOIN" /* right outer */
                | "⋈" | "><" | "FJOIN" /* full outer */
                | "⋈" | "SEMI" /* semi-join */
                | "⋈" | "ANTI" /* anti-join */
                | "USEMI" /* pairwise universal semi-join */
product_op     ::= "×" | "CROSS"
                | "÷" | "DIV"
                | "°" | "COMPOSE"

/* L ASOF [INNER] L.ts >= R.ts [WITHIN DURATION 'PT5M'] [TIES(FIRST)] R */
asof_join     ::= "ASOF" "INNER"? predicate ( "WITHIN" operand )?
                ( "TIES" "(" ( "FIRST" | "LAST" ) ")" )? postfix_expr

/* L IJOIN OVERLAPS (L.start, L.end, R.start, R.end) R */
interval_join ::= "IJOIN" allen_relation "(" dotted_name "," dotted_name "," dotted_name ")" postfix_expr
allen_relation ::= "INTERSECTS" | "OVERLAPS" | "OVERLAPPED_BY" | "DURING" | "CONTAINS"
                | "STARTS" | "STARTED_BY" | "FINISHES" | "FINISHED_BY" | "EQUALS"
                | "MEETS" | "MET_BY" | "PRECEDES" | "PRECEDED_BY"

/* L LATERAL fn(L.col, 3) – the right side is always a function call. */
lateral_join  ::= "LATERAL" name "(" ( operand ( "," operand )* )? ")"

/* R* OVER (src, dst) transitive closure; R* OVER (src, dst) reflexive. */
postfix_expr  ::= primary_rel ( ( "+" | "*" ) "OVER" "(" name edge_separator name ")" )?

/* "," reads an edge one way; "↔" (ASCII "<->") reads it both ways. */
edge_separator ::= "," | "↔" | "<->"

primary_rel   ::= "(" rel_expr ")"
                | truth_relation
                | relation_call
                | relation_ref
                | unary_operation

/* UNIT (alias DEE): one row, no columns. EMPTY (alias DUM): no rows, no columns. */
truth_relation ::= "UNIT" | "DEE" | "EMPTY" | "DUM"

```

```

/* Orders, warehouse.orders, relix.relations – a name, possibly dotted. */
relation_ref      ::= relation_name ( "." name )*
/* A table-valued function: recentOrders(30). */
relation_call     ::= relation_name ( "." name )* "(" ( operand ( "," operand )* )? ")"
relation_name     ::= name - relation_keyword

/* Words that begin an expression and so cannot name a relation unless
   backticked: `order` is the relation named order. */
relation_keyword  ::= "UNIT" | "DEE" | "EMPTY" | "DUM"
                  | "PROJECT" | "SELECT" | "RENAME" | "GROUP" | "SORT" | "ORDER"
                  | "LIMIT" | "DISTINCT" | "UNNEST" | "WHY"
                  | "CLOSURE" | "RCLOSURE" | "CLUSTER" | "PATH" | "TRACE" | "FIX"
                  | "ITERATE" | "FORALL" | "SAMPLE" | "SOLVE" | "OPTIMIZE" | "TOP" | "COVER"
                  | "DOWNSAMPLE" | "ROLLING" | "WINDOW" | "SESSIONIZE"
                  | "PIVOT" | "UNPIVOT" | "TREE"

```

Unary operators

```
/* Every unary operator takes its input LAST, in parentheses:  $\sigma$  p (R). */
unary_operation ::= projection | selection | rename | aggregation | sort | limit
                | distinct | why | unnest
                | closure | cluster | path | trace | fixpoint | iterate
                | universal | sample | solve | optimize | top_k | cover
                | downsample | rolling | window | sessionize | tree | pivot | unpivot

input           ::= "(" rel_expr ")"
arrow           ::= "→" | "->"

/*  $\pi$  name, price * qty → total (Orders) */
projection      ::= ( " $\pi$ " | "PROJECT" ) projected ( "," projected ) * input
projected       ::= operand ( arrow name ) ?

/*  $\sigma$  amount > 100  $\wedge$  status = 'open' (Orders) */
selection       ::= ( " $\sigma$ " | "SELECT" ) predicate input

/*  $\rho$  Staff (Employees)                rename the relation
    $\rho$  (id → staff_id, name → staff) (E)  rename some columns
    $\rho$  Staff (sid, sname, dept, mgr, pay) (E) rename every column, in order */
rename          ::= ( " $\rho$ " | "RENAME" )
                  ( name ( "(" rename_list ")" ) ? | "(" rename_list ")" ) input
rename_list     ::= name ( "," name ) *
                  | name arrow name ( "," name arrow name ) *

/*  $\gamma$  customer_id, SUM(amount) → total, COUNT(*) → orders (Orders)
   Grouping keys come first as a plain list; at least one aggregate is required.
   With no keys the whole input is one group. A key is an operand, so {k} is a
   struct and groups by a one-field struct rather than by the column k. */
aggregation     ::= ( " $\gamma$ " | "GROUP" ) "BY"?
                  ( grouping_key ( "," grouping_key ) * "," ? ) ?
                  aggregate ( "," aggregate ) * "," ? input
grouping_key    ::= operand ( arrow name ) ?
aggregate       ::= ( "SUM" | "AVG" | "MIN" | "MAX" | "COLLECT" ) "(" operand ")" ( arrow name ) ?
                  | "COUNT" "(" ( operand | "*" ) ")" ( arrow name ) ?
                  | ( "ARGMAX" | "ARGMIN" ) "(" operand "," operand ")" ( arrow name ) ?

/*  $\tau$  amount DESC, name (Orders); ORDER BY amount DESC (Orders) */
sort            ::= ( " $\tau$ " | "SORT" | "ORDER" ) "BY"? sort_key ( "," sort_key ) * input
sort_key        ::= operand ( "ASC" | "DESC" ) ?

/*  $\lambda$  10 (R): the first 10 rows.  $\lambda$  20, 10 (R): skip 20, then take 10. */
limit          ::= ( " $\lambda$ " | "LIMIT" ) INTEGER ( "," INTEGER ) ? input

distinct        ::= ( " $\delta$ " | "DISTINCT" ) input
/*  $\omega$  (R): each row with its lineage as a nested provenance column. */
why             ::= ( " $\omega$ " | "WHY" ) input

/*  $\mu$  items WITH ORDINALITY pos (Orders): one row per array element. */
unnest          ::= ( " $\mu$ " | "UNNEST" ) name ( "WITH" "ORDINALITY" name ) ? input

/* Graph operators. */
closure         ::= ( "CLOSURE" | "RCLOSURE" ) name edge_separator name input
cluster         ::= "CLUSTER" name "," name "AS" name input
/* HOPS n is short for HOPS 1 TO n; bounds must satisfy  $1 \leq \min \leq \max$ . */
path            ::= "PATH" name edge_separator name "HOPS" INTEGER ( "TO" INTEGER ) ?
                  "AS" name input
trace           ::= "TRACE" name edge_separator name "VIA" name
                  ( "MINIMIZE" | "MAXIMIZE" ) "AS" name input

/* FIX Reach (base, step): inside step, Reach names the relation being built. */
fixpoint        ::= "FIX" IDENTIFIER "(" rel_expr "," rel_expr ")"

/* ITERATE Board (base, step) ROUNDS 4: inside step, Board names the previous
   round, and each round replaces it. ROUNDS, UNTIL and STABLE are matched by
```



```

    their text rather than reserved, so they remain usable as names. */
iterate      ::= "ITERATE" IDENTIFIER "(" rel_expr "," rel_expr ")" iterate_stop
iterate_stop ::= "ROUNDS" INTEGER
              | "UNTIL" ( "STABLE" | name ( "," name ) * "WITHIN" NUMBER
                          "PER" name ( "," name ) * ) "MAX" INTEGER "ROUNDS"

/* ∀ student : grade >= 50 (Results) */
universal    ::= ( "∀" | "FORALL" ) ( name ( "," name ) * )? ":" predicate input

/* SAMPLE 0.1 (R): each row with probability 0.1. SAMPLE 100 ROWS (R): exactly 100. */
sample      ::= "SAMPLE" NUMBER ( "SEED" INTEGER )? input
              | "SAMPLE" INTEGER "ROWS" ( "SEED" INTEGER )? input

/* SOLVE total = price * qty (R) */
solve       ::= "SOLVE" operand "=" operand input

/* OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(weight) <= 50 PER owner (Items)
   OPTIMIZE ALLOCATE (0, 1) MINIMIZE SUM(cost) SUBJECT TO SUM(share) = 1 -> share (Plans) */
optimize    ::= "OPTIMIZE" objective "SUBJECT" "TO" constraints per_identifiers? input
              | "OPTIMIZE" "ALLOCATE" "(" signed_number "," signed_number ")"
              objective "SUBJECT" "TO" constraints arrow IDENTIFIER
              per_identifiers? input
objective   ::= ( "MAXIMIZE" | "MINIMIZE" ) sum_term
constraints ::= constraint ( ( "∧" | "AND" ) constraint ) *
constraint  ::= sum_term ( "<=" | "≤" | ">=" | "≥" | "=" ) signed_number
sum_term    ::= "SUM" "(" operand ")"
signed_number ::= "-"? NUMBER
per_identifiers ::= "PER" IDENTIFIER ( "," IDENTIFIER ) *

/* TOP 3 amount DESC PER customer_id (Orders). TOP 10, 5 ...: skip 10, keep 5.
   Without PER, the top rows overall. */
top_k       ::= "TOP" INTEGER ( "," INTEGER )? sort_key ( "," sort_key ) *
              per_identifiers? input

/* COVER 2 (Params): a small set of rows covering every pair of column values. */
cover       ::= "COVER" "EXACT"? INTEGER input

/* DOWNSAMPLE ts BY '5m' USING AVG PER sensor FOR 1000 ROWS (Readings) */
downsample  ::= "DOWNSAMPLE" name "BY" NONBLANK_STRING "USING"
              ( "AVG" | "MIN" | "MAX" | "SUM" | "COUNT" )
              per_names? ( "FOR" POSITIVE_INTEGER "ROWS" )? input
per_names   ::= "PER" name ( "," name ) *

/* ROLLING AVG(price) OVER 3 ROWS SORT ts PER ticker AS avg3 (Trades) */
rolling     ::= "ROLLING" ( "SUM" | "AVG" | "COUNT" | "MIN" | "MAX" ) "(" operand ")"
              "OVER" ( INTEGER | "ALL" ) "ROWS"
              ( "τ" | "SORT" ) sort_key ( "," sort_key ) * per_names? "AS" name input

/* WINDOW RANK() SORT amount DESC PER customer_id AS rnk (Orders) */
window      ::= "WINDOW" window_function
              ( "τ" | "SORT" ) sort_key ( "," sort_key ) * per_names? "AS" name input
window_function ::= ( "ROW_NUMBER" | "RANK" | "DENSE_RANK" | "PERCENT_RANK" ) "(" " )"
              | "NTILE" "(" operand ")"
              | ( "LAG" | "LEAD" ) "(" operand ( "," operand ( "," operand )? )? ")"
              | ( "FIRST_VALUE" | "LAST_VALUE" ) "(" operand ")"

/* SESSIONIZE ts GAP DURATION 'PT30M' PER user_id AS session (Events) */
sessionize  ::= "SESSIONIZE" name "GAP" operand per_names? "AS" name input

/* TREE id BY parent_id ORDER position AS children (Nodes) */
tree        ::= "TREE" name "BY" name ( "ORDER" sort_key ( "," sort_key ) * )?
              "AS" name input

/* PIVOT revenue BY month PER region (Sales) */
pivot       ::= "PIVOT" name "BY" name per_names? input
/* UNPIVOT (jan, feb, mar) AS (month, revenue) (Sales) */
unpivot     ::= "UNPIVOT" "(" name ( "," name ) * ")" "AS" "(" name "," name ")" input

```

Predicates

```
/* NOT binds tightest, then AND, then OR. */
predicate      ::= and_predicate ( ( "v" | "OR" ) and_predicate )*
and_predicate  ::= not_predicate ( ( "Λ" | "AND" ) not_predicate )*
not_predicate   ::= ( "¬" | "NOT" ) not_predicate
                | primary_predicate
primary_predicate ::= "(" predicate ")"
                | operand condition_tail

condition_tail  ::= comparison_op operand
                | ( "∈" | "IN" ) set_expr
                | ( "∉" | ( "¬" | "NOT" ) "IN" ) set_expr
                | ( ( "¬" | "NOT" ) )? "LIKE" operand
                | ( "=" | "≠" | "!=" ) null_word           /* a null test */
                | "IS" ( ( "¬" | "NOT" ) )? null_word       /* a null test */

comparison_op   ::= "=" | "≠" | "!=" | "<" | "≤" | "<=" | ">" | "≥" | ">="
null_word       ::= "1" | "NULL"

/* x IN {1, 2, 3}: a set is written in braces, never in parentheses.
   x IN tags: an array-valued column. */
set_expr        ::= "{" ( operand ( "," operand )* )? "}"
                | operand
```

Operands (scalar expressions)

```
/* * and / bind tighter than + and -; all four associate to the left.
   A leading "-" negates the term right after it. */
operand        ::= term ( ( "+" | "-" ) term )*
term           ::= factor ( ( "*" | "/" ) factor )*
factor         ::= "-" factor
                | primary_operand

primary_operand ::= literal
                | function_call
                | column_ref
                | "(" operand ")"
                | "(" predicate ")"           /* a condition's truth value, as a BOOLEAN */
                | struct_construction
                | array_construction

/* Orders.amount, addr.city – a column, a qualified column, or a nested field. */
column_ref     ::= operand_name ( "." name )*

/* Round(price, 2). The "(" must follow the name with NO space between
   (CALL_OPEN): "Round (price)" is the column Round followed by a parenthesis. */
function_call  ::= operand_name ( "." name )* CALL_OPEN ( argument ( "," argument )* )? ")"
/* An argument may be a condition: IIf(amount > 100, 'big', 'small'). */
argument       ::= operand
                | predicate

operand_name   ::= name - ( "TRUE" | "FALSE" | "DATE" | "TIME" | "TIMESTAMP" | "DURATION" )

literal        ::= STRING
                | NUMBER
                | "TRUE" | "FALSE"
                | ( "DATE" | "TIME" | "TIMESTAMP" | "DURATION" ) STRING
                /* ISO-8601 text: DATE '2026-03-01', TIMESTAMP '2026-03-01T09:30:00Z',
                   TIME '09:30', DURATION 'PT30M' */

/* {id, total: amount * 2}: a struct. {id} is short for {id: id}. */
struct_construction ::= "{" ( IDENTIFIER ( ":" operand )? ( "," IDENTIFIER ( ":" operand )? )* )? "}"
/* [a, b, 3]: an array. */
array_construction ::= "[" ( operand ( "," operand )* )? "]"
```

Names and tokens

```
/* In most positions any word is accepted as a name, reserved or not.
   IDENTIFIER is the stricter form a few positions require (FIX, ITERATE, the PER
   list of OPTIMIZE and TOP, struct field names); a reserved word there
   must be backticked. */
name                ::= WORD | DELIMITED_IDENTIFIER
dotted_name         ::= name ( "." name ) *
IDENTIFIER           ::= ( WORD - reserved_word ) | DELIMITED_IDENTIFIER

/* The words the expression layer reserves. Letter case is ignored. */
reserved_word       ::= "TRUE" | "FALSE" | "NULL" | "UNIT" | "DEE" | "EMPTY" | "DUM"
                    | "DATE" | "TIME" | "TIMESTAMP" | "DURATION"
                    | "SUM" | "AVG" | "COUNT" | "MIN" | "MAX" | "COLLECT" | "ARGMAX" | "ARGMIN"
                    | "ASC" | "DESC" | "PROJECT" | "SELECT" | "RENAME" | "GROUP" | "SORT"
                    | "ORDER" | "LIMIT" | "DISTINCT" | "UNNEST" | "WITH" | "ORDINALITY"
                    | "CLOSURE" | "RCLOSURE" | "CLUSTER" | "PATH" | "HOPS" | "AS" | "OVER"
                    | "FIX" | "ITERATE" | "FORALL" | "SAMPLE" | "SEED" | "SOLVE" | "OPTIMIZE"
                    | "ALLOCATE" | "MAXIMIZE" | "MINIMIZE" | "SUBJECT" | "TO" | "TOP" | "PER"
                    | "ROWS" | "COVER" | "EXACT" | "DOWNSAMPLE" | "BY" | "USING" | "FOR"
                    | "LATERAL" | "ROLLING" | "WINDOW" | "SESSIONIZE" | "GAP" | "TRACE"
                    | "VIA" | "PIVOT" | "UNPIVOT" | "TREE" | "WHY"
                    | "JOIN" | "SEMI" | "ANTI" | "LJOIN" | "RJOIN" | "FJOIN" | "USEMI"
                    | "ASOF" | "IJOIN" | "WITHIN" | "TIES" | "CROSS" | "UNION" | "UALL"
                    | "OUNION" | "DIFF" | "MINUS" | "EXCEPT" | "INTER" | "INTERSECT"
                    | "DIV" | "SYMDIFF" | "COMPOSE" | "AND" | "OR" | "NOT" | "IS" | "IN" | "LIKE"

/* ---- Tokens -----
Whitespace separates tokens and is otherwise ignored, except that a
function call's "(" must touch its name.

Comments:      "--" to the end of the line, and a block that opens with
                slash-star and closes with star-slash, as in C.
                "/*" is not a comment.

WORD           a letter or "_", then letters, digits and "_". Any Unicode
                letter counts, except the operator glyphs ( $\sigma$ ,  $\pi$ ,  $\rho$ , ...).

DELIMITED_IDENTIFIER
                "`" any characters except a newline "`", with "`" standing for
                one backtick: `order`, `line total`. Never empty, and never
                only whitespace.

NUMBER         digits, optionally "." and more digits: 42, 3.14.
                No exponent, no leading "." and no sign (a "-" before a
                number is the minus operator).

INTEGER        a NUMBER without a fractional part.
POSITIVE_INTEGER
                an INTEGER of at least 1.

CALL_OPEN      a "(" written directly after a name, with no space or
                comment between them. Wherever the grammar asks for "(",
                a CALL_OPEN is accepted too; the reverse is not so.

STRING         expression layer: '...' or "...". Escapes: \\ \" \' \n \t;
                any other backslash sequence is an error.
STRING_DQ      statement layer: "..." only. \" is a quote; any other
                backslash is kept as written, so "C:\data\a.csv" needs no
                doubling.
NONBLANK_STRING, NONBLANK_STRING_DQ
                a STRING or STRING_DQ holding more than whitespace.

MARKDOWN_TABLE [" raw text "]"      (see Inline tables)
CSV_TABLE      [" raw text "]"

----- */
```

Rules the grammar does not show

A few rules depend on context in a way EBNF cannot state compactly, and a few more are easy to miss in the rules above.

- **Keywords ignore case; names keep it.** `select`, `SELECT` and `Select` are the same keyword. A name is stored as written.
- **A function call's parenthesis touches the name.** `Round(price, 2)` is a call. `Round(price, 2)` is a column called `Round` followed by a parenthesis. This is what lets `τ name (Users)` sort `Users` by the column `name`.
- **A join's condition comes before its right input.** `A ⋈ A.id = B.a_id B`. An AS-OF join puts `INNER`, `WITHIN` and `TIES` there too.
- **A predicate always compares.** A condition is a comparison, a membership or pattern test, or a null test, joined by `∧`, `∨` and `¬`. A bare boolean does not stand alone: write `σ active = true (R)`, not `σ active (R)`, and `σ IsNull(x) = true (R)` rather than `σ IsNull(x) (R)`.
- **Sets use braces.** `x IN {1, 2}` tests membership. `x IN (1, 2)` is a syntax error: a parenthesis opens a single expression.
- **Braces elsewhere build a struct.** Outside a set, `{a}` is a struct with one field `a`. A grouping key is an operand, so `γ {region}, COUNT(*) (Orders)` parses and groups by a struct, not by the column. Write `γ region, COUNT(*) (Orders)`.
- **There is no NULL value to write.** `1` and `NULL` appear only in a null test: `x = 1`, `x = NULL`, `x IS NULL` and `x IS NOT NULL` all ask whether `x` is `NULL`. Anywhere else, the word `null` names a column.
- **<> is not an operator.** Write `!=` or `≠`.
- **Two minus signs.** The ASCII hyphen `-` subtracts numbers. The Unicode minus `−` (U+2212) is set difference, whose ASCII spellings are `DIFF`, `MINUS` and `EXCEPT`.
- **The two layers quote and point differently.** Outside braces, strings are double-quoted and the arrow is `->`. Inside braces, strings take either quote and the arrow is `→` or `->`.
- **Reserved words as names.** A reserved word can name a column almost anywhere (`γ region, COUNT(*) → count (Orders)`). In relation position it cannot, nor as a struct field, a `FIX` or `ITERATE` name or a `PER` key of `OPTIMIZE` or `TOP`; there it must be backticked: `π name (`order`)`. Backticks are always safe.
- **query names one plain name.** `query Totals;` works, but a dotted name needs the expression form: `query { relix.plan };.`
- **Whole numbers must fit.** An `INTEGER` that counts rows or bounds (`λ`, `TOP`, `SAMPLE ... ROWS`, `SEED`, `DOWNSAMPLE ... FOR`, a multiplicity, a pagination default) must fit in a 64-bit signed integer, and one that counts hops, a covering strength or a window frame in a 32-bit one. A larger number is rejected where it is written.
- **Temporal literals are checked as they are read.** `DATE '2024-02-30'` has the shape the grammar describes but is not a real date, so the parser rejects it. The same holds for `TIME`, `TIMESTAMP` and `DURATION`, whose strings must be valid ISO-8601 values.
- **Statement order.** `namespace` must be the first statement and `env` the next; the other statements may come in any order.
- **Nesting depth.** One expression may nest at most 250 levels deep.

The grammar says what parses. Analysis then checks what it means: that every column exists, that a positional rename names every column, that `SAMPLE`'s probability is between 0 and 1. A script can parse and still be rejected there, with a message naming the problem.

Examples

The shape of a whole script: an inline table, a function, two views and two results.

```
Regions := [
region	manager
EU	Ana
US	Ben
];

def with_tax(amount: NUMBER): NUMBER := { amount * 1.2 };

Big := { σ amount > 100 (Orders) };

Summary := {
  γ region, SUM(with_tax(amount)) → gross, COUNT(*) → orders (Big) ⋈ Regions
};

query Summary;
query { τ gross DESC (Summary) };
```

A join's condition sits between the operator and the right-hand relation:

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

A function call's parenthesis touches the name. With a space, `amount` is a column and `(Orders)` is the input:

```
query { τ amount DESC (Orders) };
```

A set is written in braces:

```
query { σ status IN {'open', 'held'} (Orders) };
```

Parentheses do not make a set:

```
query { σ status IN ('open', 'held') (Orders) };
```

SQL's `<>` is not an operator:

```
query { σ status <> 'open' (Orders) };
```

A null test, in each of its spellings:

```
query { σ email = ⊥ ∨ phone IS NOT NULL (Customers) };
```

Grouping keys come first, with no braces, and at least one aggregate is required:

```
query { γ region (Orders) };
```

Braces around a grouping key are not an error, but they change the result. This groups by a one-field struct and returns one struct column instead of `region`:

```
query { γ {region}, COUNT(*) → orders (Orders) };
```

To list the distinct values of a column, project it and remove the duplicates:

```
query { δ (π region (Orders)) };
```

Statement-layer strings take double quotes only:

```
source Orders2 from csv('orders.csv') { schema: { id: NUMBER } };
```

// is not a comment:

```
query { Orders }; // every order
```

See Also

- [operator spellings](#): every glyph beside its ASCII form, as a table
- [comments](#): the two comment forms
- [delimited identifier](#): backticks, and why a reserved word needs them
- [inline table](#): the Markdown and CSV table forms in full
- [source](#), [http source](#), [connection](#): what each source field does
- [assignment & query](#), [def](#), [import](#), [relate](#): the statements

namespace (script namespace declaration)

Syntax

```
namespace <name>;  
  
namespace analytics;
```

Description

A namespace declaration at the top of a script labels everything it defines with a name, helping keep symbols from different files or domains from clashing. Think of it as the package or schema name for the script's relations and functions.

Technical Description

`namespace name;` sets the script's namespace, recorded in the `SemanticModel`. Symbols are registered under it, but lookups fall through: `lookupRelation(name)` searches the `default` namespace, then `builtin`, then all other registered namespaces — so a script that declares a namespace still resolves its own and imported symbols throughout the pipeline.

Examples

Label a script's symbols:

```
namespace analytics;  
  
source Orders from database { url: "${DB}", table: "orders",  
    schema: { id: NUMBER, amount: NUMBER } };  
query { γ SUM(amount) → total (Orders) };
```

Limitations

One namespace per script, declared first. Namespacing organises symbols but does not by itself isolate them — all imported symbols remain globally resolvable.

Alternatives

Omit the declaration to use the default namespace.

See Also

[import](#), [source](#), [def](#)

Notes

The namespace appears in the IR report header (RELIX IR namespace=...).

import (reuse symbols from another file)

Syntax

```
import "<path>";                -- everything exported
import { Name1, Name2 } from "<path>";    -- specific symbols
import <Name> from "<path>";        -- a single symbol
import source|relation|function <Name> from "<path>";    -- qualified

import { Users, Orders } from "../db.relix";
```

Description

Import pulls definitions — relations, sources, functions — from another .relix file into the current one, so you can share common sources and helper functions across scripts instead of copying them. Paths are relative to the file doing the importing.

Technical Description

Imports are resolved depth-first with cycle detection. All symbols from an imported file are registered in the global symbol table regardless of import form; named imports control only what is re-exported to files that in turn import this one — they do not restrict what you can reference locally. The qualified forms (`import source/relation/function Name`) disambiguate when a name is otherwise ambiguous. Circular imports are reported as an error.

Examples

Bring in two named relations:

```
import { Users, Orders } from "../db.relix";
```

Import everything a shared file exports:

```
import "../shared.relix";
```

A single symbol, unqualified:

```
import Employees from "../hr.relix";
```

Qualified to disambiguate kind:

```
import function discount from "../pricing.relix";
```

Limitations

Circular imports are not allowed. Named imports affect re-export visibility, not local availability (all imported symbols are globally registered). Symbols must be declared before they are used. The path must not be blank:

```
import "";
```

Alternatives

Inline the source/function directly when it is only used in one script.

See Also

[source](#), [def](#), [def-relation](#), [namespace](#)

Notes

Use named imports to keep a clean public surface for files that are themselves imported by others.

def (scalar function definition)

Syntax

```
def <name>(<param>: <TYPE> [, ...]) : <RETURN_TYPE> := { <expression> };
private def <name>(...) : <TYPE> := { <expression> };

def double(x: NUMBER) : NUMBER := { x * 2 };
def add(a: NUMBER, b: NUMBER) : NUMBER := { a + b };
```

Description

`def` declares your own scalar function — a reusable calculation you can call by name inside conditions and projections. Define a discount formula, a name formatter, or a tax calculation once, then use it wherever you need it. Mark it `private` to keep it out of what other files see when they import this one.

Technical Description

A scalar `def` binds a `ScalarFunctionSymbol` with typed parameters and a return type (`NUMBER`, `STRING`, `BOOLEAN`, `ANY`, or a temporal type). The body is a single operand expression over the parameters. Calls resolve in predicates and projected expressions via the symbol table, falling back to the builtin registry. `private` suppresses re-export on import. Functions must be declared before they are called.

Examples

A simple calculation:

```
def double(x: NUMBER) : NUMBER := { x * 2 };
query { σ double(salary) > 100000 (Employees) };
```

Multiple parameters:

```
def add(a: NUMBER, b: NUMBER) : NUMBER := { a + b };
query { π add(price, tax) → total (Products) };
```

Zero-argument constant:

```
def vat() : NUMBER := { 0.2 };
```

Private helper, not exported:

```
private def clean(s: STRING) : STRING := { Trim(s) };
```


Limitations

The body is a single scalar expression, not a multi-statement procedure. Parameter and return types are scalar — the nested struct and array forms a source column may declare are not available here. For a function that returns a whole relation, use the `RELATION` return type (a table-valued function).

Alternatives

A table-valued function (`: RELATION`) returns a relation rather than a value. The built-in scalar functions cover common string/math/date operations already.

See Also

[def-relation](#), [import](#), [project](#), [select](#)

Notes

User functions are evaluated per row by the engine and are not pushed down to SQL.

def ... : RELATION (table-valued function)

Syntax

```
def <name>(<param>: <TYPE> [, ...]) : RELATION := { <relational expression> };
def <name>(<param>: RELATION(<column> [: <TYPE>], ...) [, ...]) : RELATION := { ... };

def ordersFor(cid: NUMBER) : RELATION := {
  σ customer_id = cid ∧ status = "completed" (Orders)
};

def sources(E: RELATION(src, dst)) : RELATION := { π src (E) };
```

Description

A table-valued function is a parameterised view: a named query that takes arguments and returns a whole relation. Define "the completed orders for customer X" once, then call it with different customers wherever a relation is expected — in a join, a projection, or as a query on its own. It keeps repeated query shapes DRY.

A parameter can also be a **relation**. Then the function is a rule written once and applied to whichever relation you pass: one generation of a cellular automaton, a cleaning step, a graph measure over any edge list. The parameter declares the columns the body reads, and any relation with those columns can be passed.

Technical Description

A `def name(...) : RELATION := { body }` binds a `RelationFunctionSymbol` whose body is a relational-algebra expression; its result schema is the body's inferred schema (like a view). Binding is by substitution — the planner inlines the body with each parameter reference replaced by the supplied argument, then re-infers its schema. Scalar arguments must be constant expressions (no column references); a correlated argument needs a `LATERAL` join instead. Nested TVF calls inline transitively; recursion is rejected at plan time. The `RELATION` keyword is case-insensitive.

Relation parameters. `E: RELATION(src, weight: NUMBER)` declares a relation with those columns; a column with no type is `ANY`. Only a table-valued function can take one. Inside the body, `E` is a relation with exactly that heading, so the body is checked once, at the `def`, and a column the heading does not declare is an error there. `E.src` qualifies a column by the parameter's name, and the parameter means `E` even where the script defines a relation of the same name.

The argument is a relation's **name** — a table, a source or a view. A call is checked against the heading: the relation must have every declared column, at a compatible type, and may have more. The body sees the argument **narrowed to the heading**: binding substitutes `p E (π <declared columns> (argument))`, so a column the heading leaves out cannot reach the body, and a natural join in it can never match on a column its author did not declare. A parameter can be passed on to another function, which checks it against its own heading in turn.

Inside an `ITERATE` step, the argument can be the iterated name itself: the step hands the previous round to the function, which is how a rule written once is iterated. A `FIX` step cannot do the same, because `FIX` needs to see that its step reads its name monotonically, and a function body is out of its sight.

Examples

A filtered, parameterised view:

```
def ordersFor(cid: NUMBER) : RELATION := {
  σ customer_id = cid ∧ status = "completed" (Orders)
};
query { ordersFor(2) };
```

Use it inside a join:

```
query { π name, amount (Customers ⋈ ordersFor(2)) };
```

Zero-argument reusable query:

```
def activeUsers() : RELATION := { σ active = true (Users) };
```

A rule over any edge list, applied to two:

```
def outDegree(E: RELATION(src, dst)) : RELATION := { γ src, COUNT(*) → out (E) };
query { outDegree(Edges) };
query { outDegree(Graph) };
```

The same rule iterated, a round at a time:

```
def extend(R: RELATION(src, dst)) : RELATION := {
  π src, dst2 → dst (R ⋈ π src → dst, dst → dst2 (Edges))
};
query { ITERATE R (π src, dst (Edges), extend(R)) ROUNDS 3 };
```

Only a table-valued function takes a relation:

```
def width(E: RELATION(x)) : NUMBER := { 1 };
```

Worked Example

One generation of Conway's Game of Life, written once and applied twice. A cell lives if it has three live neighbours, or two and was already alive; `offsets` are the eight directions to a neighbour.

```

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

Blinker := [
x	y
1	0
1	1
1	2
];

def generation(G: RELATION(x: NUMBER, y: NUMBER)) : RELATION := {
  π x, y (σ n = 3 (ρ C(x, y, n) (γ nx, ny, COUNT(*) → n (π x + dx → nx, y + dy → ny (G × Offsets)))))
  ∪ π x, y (σ n = 2 (ρ C(x, y, n) (γ nx, ny, COUNT(*) → n (π x + dx → nx, y + dy → ny (G × Offsets))))) ⋈ G
};

Gen1 := { generation(Blinker) };
Gen2 := { generation(Gen1) };

```

A blinker is three cells in a line, and each generation turns it through a quarter turn. The first generation lies it down:

```
query { τ x, y (Gen1) };
```

```

x  y
—  —
0  1
1  1
2  1
(3 rows)

```

and the second stands it up again, where it started: `Gen2` holds the same three cells as `Blinker`. `Gen2` passes `Gen1` — itself a call to `generation` — to `generation`, which is not recursion: the argument belongs to the call, not to the function's body.

Limitations

Scalar arguments must be constant — you cannot pass a column from the surrounding query; use a `LATERAL` join for that. A relation argument is a relation's name: to pass an expression, name it as a view first. A function with a relation parameter cannot be called through `LATERAL`, whose arguments are the left side's values, row by row. A TVF body cannot reference itself (recursion is rejected). The body is not pushed down to SQL as a unit.

Alternatives

A plain view (`Name := { ... };`) when no parameters are needed. `FIX` for genuine recursion, and `ITERATE` for repeating a step — which can itself call a function with a relation parameter.

See Also

[def](#), [assignment](#), [import](#), [natural-join](#), [iterate](#)

Notes

Because binding is by inlining, a TVF behaves exactly as if you had written its body inline with the arguments substituted — a relation argument narrowed to the parameter's heading, and renamed to the parameter's name.

Relation catalog (`relix.relations`)

Syntax

```
relix.relations -- every relation the script can name (name, kind, namespace, materialization, row_count, boundedness)
```

Description

`relix.relations` has one row per relation the script can name — the sources it declares, the inline tables it embeds, the connection-backed tables it reaches, and every view it defines with `:=`.

It is the catalog the others hang off: `relix.columns` names the same relations one row per column, `relix.keys` one row per key column, and a query joining any of them to this one is how a question about *all* the relations gets asked.

Columns:

- `name` (STRING) — the relation's name, as it was written.
- `kind` (STRING) — what it is: "SRC" a declared source, "INL" an inline table, "DB" a connection-backed table, "QR" a view, "SYS" an engine-computed catalog. The same codes the IR prints.
- `namespace` (STRING) — the namespace the relation was declared in.
- `materialization` (STRING) — how much of the relation has to be held at once to produce a row: "stream" (none), "bag", "set", "sort".
- `row_count` (NUMBER) — how many rows it has, where that was measured. NULL where it was not.
- `boundedness` (STRING) — whether it is finite: "bounded", "unbounded", or "unknown".

Two questions about size, and only one of them is measured

`row_count` and `boundedness` sound like the same question and are not. A count is *observed* — from the rows of an inline table, from a generator that knows its own length, from the catalog of an introspected database table. Boundedness is *proved*, from what a relation is and what it reads.

That is why the pairs which look inconsistent are the informative ones:

| <code>boundedness</code> | <code>row_count</code> | What it means |
|--------------------------|------------------------|---|
| <code>bounded</code> | a number | finite, and counted |
| <code>bounded</code> | NULL | finite, but nobody has counted it — a CSV file nothing has read |
| <code>unbounded</code> | NULL | it has no count because it has no end |
| <code>unknown</code> | NULL | nothing here shows it finite either way |

Reading a NULL `row_count` as "empty" is the mistake the two columns together prevent: a table with no count is one nothing has counted.

Where boundedness comes from

A relation that holds its own rows — an inline table, a file, an HTTP resource, a database table, one of these catalogs — is `bounded`. Each is a finite extent that somebody is holding.

A **generator** is the exception, and the only leaf that can be otherwise: a generator declares whether it ends, so `Range` is `bounded` while `Naturals` and `Primes` are `unbounded`.

A **view** is neither declared nor measured but *derived*, from the body it was defined as — so it is exactly as bounded as what it reads, through however many other views:

```
source Naturals from generator { name: "Naturals" };
Big      := { σ n > 1000000 (Naturals) };    -- still unbounded: filtering removes no end
FirstTen := { λ 10 (Naturals) };             -- bounded: λ is what bounds it
```

That derivation is why the column is worth having on views at all. Selecting from an infinite relation leaves it infinite, and `λ` is the operator that ends it — the same rule the engine uses when it refuses to sort or group an endless input.

unknown is an answer, not a gap

`boundedness` is never NULL. "unknown" means *nothing here shows this finite either way*, which is a claim, where a NULL `row_count` is the absence of one.

Three things say it. A generator this session cannot place — because no generator of that name is installed. A relation named in a view that resolves to nothing. And a definitional cycle: A defined from B and B from A puts neither on any footing from which finiteness can be shown.

"unknown" does not mean a query over the relation is refused. The engine refuses what is *provably* endless, and treats not-knowing as permission to proceed — so this column tells you where a query is running on trust.

Technical Description

The extent is fixed when analysis finishes: it describes the script, and querying it runs nothing against a source. A view's boundedness is derived from the same walk the planner uses, over the same lattice — the two cannot disagree about whether a query's input ends.

Like the other catalogs, this one describes only relations *outside* the reserved `rel` namespace, which is what keeps it about your script. Use `rel.catalog` to see the namespace itself.

Examples

Every relation the script can name:

```
query { rel.relations };
```

Just the views, in order:

```
query { τ name (σ kind = "QR" (rel.relations)) };
```

What is finite, and therefore safe to sort, group or collect:

```
query { π name (σ boundedness = "bounded" (rel.relations)) };
```

The ones to be careful with — endless, or not shown to be otherwise:

```
query { π name, kind, boundedness (σ boundedness ≠ "bounded" (relix.relations)) };
```

Finite but never counted, which is the pair a NULL count on its own reads wrongly:

```
query { π name (σ boundedness = "bounded" ∧ row_count IS NULL (relix.relations)) };
```

How much of the script is of each kind, and how much of it ends:

```
query { γ kind, boundedness, COUNT(*) → relations (relix.relations) };
```

Which relations buffer their whole input before emitting a row, joined to their size so the expensive ones stand out:

```
query {  
  τ row_count DESC (σ materialization ≠ "stream" (relix.relations))  
};
```

Limitations

`row_count` is a count where one was taken, not an estimate; a view never carries one. See *Two questions about size* above.

`boundedness` is a lower bound on what is provable, not on what is true. A `FIX` whose step computes new values can have an endless extent and still read `bounded` — whether such a recursion terminates is not decidable, so no analysis of this kind answers it. What stops that query is the engine's runtime limit, not this column.

A table-valued function's body is not walked, so a view calling one is derived from the rest of its body alone.

See also

- **candidate keys** — `relix.keys`, the other thing the statistics know about a relation.
- **introspection stdlib** — `relix.schema(rel)` and `relix.find(col)`, the column catalog read from either side.
- **reserved namespace** — `relix.catalog`, this relation's counterpart for the `relix.*` names themselves.
- `relix help catalog` — every `relix.*` relation and its columns.

Introspection standard library (`relix.unused` / `relix.deps` / `relix.impact` / `relix.find` / `relix.schema` / `relix.cycles` / `relix.funcs`)

Syntax

| | |
|---|---|
| <code>relix.unused</code> | -- relations nothing else references (one column: name) |
| <code>relix.deps(<name>)</code> | -- everything <name> transitively depends on |
| <code>relix.impact(<name>)</code> | -- everything that transitively depends on <name> |
| <code>relix.find(<column>)</code> | -- relations that have a column with that name (+ its type) |
| <code>relix.schema(<relation>)</code> | -- the columns of one relation (column, type, ordinal) |
| <code>relix.cycles</code> | -- relations caught in a circular definition (one column: name) |
| <code>relix.funcs(<category>)</code> | -- functions in a category (name, arity, return_type) |

Description

Relix ships its own introspection logic *written in Relix*, in the reserved `relix` namespace, right next to the catalog relations it queries (`relix.relations`, `relix.dependencies`, `relix.plan`, ...). These are ordinary views and table-valued functions — there is no new language construct — so you call them like any other relation and read their definitions with the REPL's `:source` command. "See how Relix uses Relix."

- `relix.unused` — a view yielding the name of every relation that no other relation references. Handy for spotting dead definitions (and, by construction, top-level outputs, which nothing references).
- `relix.deps(r)` — a table-valued function returning the relations `r` transitively depends on (its full upstream lineage).
- `relix.impact(r)` — the reverse: every relation that transitively depends on `r` (its blast radius / downstream impact).
- `relix.find(col)` — a table-valued function returning every relation that has a column named `col`, paired with the type it carries there. Handy for tracing where a column (e.g. `customer_id`) lives across a schema.
- `relix.schema(rel)` — the mirror of `relix.find`: a table-valued function returning the columns of one relation, each with its type and ordinal position. Where `find` asks "which relations have this column?", `schema` asks "what columns does this relation have?" — the data twin of the REPL's `:schema`.
- `relix.cycles` — a view yielding the name of every relation caught in a circular definition. Empty for a well-formed (acyclic) script.
- `relix.funcs(cat)` — a table-valued function filtering the function library by category (e.g. "math", "string", "user"), returning each function's name, arity, and return_type. Data-driven off `relix.functions'` properties.

`relix.functions` itself carries one row per function — name, category, arity, max_arity, return_type, the pure / deterministic / idempotent flags, and is_builtin. A function that accepts a range of argument counts is still one row: `arity` is the fewest arguments a call may pass and `max_arity` the most, with `-1` meaning any number. `Round(x)` and `Round(x, places)` are the same function, so it reads `arity = 1`, `max_arity = 2`; `Coalesce` reads `1` and `-1`. `relix.funcs` projects `arity` alone, which is the lower bound.

Technical Description

All three are layered over the `relix.dependencies` catalog relation (itself derived from `relix.plan`). `relix.unused` is the set difference of all relation names and the depended-on names:

```
(π name (relix.relations)) - (π depends_on→name (relix.dependencies))
```

`relix.deps/relix.impact` are the two directions of the transitive closure of the dependency edge, restricted to the seed `r`:

```
relix.deps(r)  = π depends_on (σ dependent = r (CLOSURE dependent, depends_on (relix.dependencies))
relix.impact(r) = π dependent (σ depends_on = r (CLOSURE dependent, depends_on (relix.dependencies))
```

The exploration helpers are filters over the other catalog relations:

```
relix.find(col)  = π relation, type (σ column = col (relix.columns))
relix.schema(rel) = π column, type, ordinal (σ relation = rel (relix.columns))
```

```
relix.funcs(cat) =  $\pi$  name, arity, return_type ( $\sigma$  category = cat (relix.functions))
```

`find` and `schema` read the same catalog relation from its two sides, and each drops the column it filtered on, since that one is constant. `schema` also drops `distinct_count` and `null_count` — they report what the statistics know rather than what the heading is; query `relix.columns` directly for those. A relation with an open (schema-on-read) heading contributes a single `*` row of type `?`.

`relix.cycles` is the *diagonal* of the dependency closure — the edges whose two endpoints coincide, i.e. relations reachable from themselves:

```
relix.cycles =  $\delta$   $\pi$  dependent-name ( $\sigma$  dependent = depends_on (CLOSURE dependent, depends_on (relix.dep
```

Because the seed is a function parameter, the call argument is a constant string. The definitions live in the `relix` namespace and the catalog relations exclude that namespace, so the `stdlib` never describes itself and the layering stays acyclic. They run through the normal parser $\rightarrow$ semantic $\rightarrow$ optimizer $\rightarrow$ planner $\rightarrow$ executor pipeline — so they are exercised by the very engine they describe — and are never pushed down to a backend.

Examples

Given a script:

```
Users := [| id | name |
          | 1 | Alice |];
Adults := {  $\sigma$  id > 0 (Users) };
Report := {  $\pi$  name (Adults) };
```

The dependency edges are `Adults  $\rightarrow$  Users` and `Report  $\rightarrow$  Adults`.

Relations nothing references (here, the top-level `Report`):

```
query { relix.unused };
-- name
-- -----
-- Report
```

Everything `Report` is built from (upstream lineage):

```
query { relix.deps("Report") };
-- depends_on
-- -----
-- Adults
-- Users
```

Everything that would break if `Users` changed (downstream impact):

```
query { relix.impact("Users") };
-- dependent
-- -----
-- Adults
-- Report
```

Which relations carry a `name` column, and as what type:

```
query { relix.find("name") };
-- relation type
-- -----
-- Users      S
-- Adults     S
```

What columns one relation has, and in what order:


```

query { relix.schema("Users") };
-- column  type  ordinal
-- -----  -
-- id      N      0
-- name    S      1

```

The math functions in the library:

```

query { relix.funcs("math") };
-- name  arity  return_type
-- ----  -
-- Abs   1      N
-- Sqr   1      N
-- ...

```

Relations caught in a circular definition (empty here — the example is acyclic):

```

query { relix.cycles };
-- name
-- ----
-- (no rows)

```

Read the shipped definition in the REPL:

```

relix> :source relix.impact
— source for relix.impact —————
  relix.impact(r: STRING): RELATION := {
    π dependent (σ depends_on = r (CLOSURE dependent, depends_on (relix.dependencies)))
  };

```

`:source <name>` (alias `:def`) works on any session relation or function too, not just the stdlib — printing a view's `:=` body or a function's definition, rendered from its AST so it round-trips through the parser.

Five of these entries also have a REPL command as shorthand — `:impact`, `:find`, `:unused`, `:cycles` and `:funcs` — each running the definition above and printing its rows under a header naming the exact call. See the REPL page for the group. Calling the entry yourself is what you want as soon as the question is not exactly the one the command asks: the result is an ordinary relation, so it filters, projects and joins against the other catalog relations like anything else.

Observability feed (`relix.events` / `relix.rules`)

Syntax

```

relix.events  -- the previous run's decisions
              -- (seq, stage, code, description, target, rows, elapsed)
relix.rules   -- the optimizer's slice of that feed (seq, code, description, target)

```

Description

Every other `relix.*` relation answers *what is* — which relations exist, their columns, the shape of each expression tree. `relix.events` answers *what happened*: it is the same feed of engine decisions that `--trace` prints, exposed as rows so `σ`, `γ`, and `τ` can read it instead of your eyes.

- `relix.events` — one row per observed decision, with columns:
- `seq (NUMBER)` — the event's 1-based position in the feed.

- `stage` (STRING) — "OPTIMIZE" (a logical rewrite rule fired), "PLAN" (a physical planning choice), or "EXECUTE" (a decision taken while running).
- `code` (STRING) — the short identifier, e.g. "SEL-001", "JOIN".
- `description` (STRING) — a one-line summary of the decision.
- `rows` (NUMBER) — the quantity the event measured, NULL for the majority that measured nothing. Every executed query reports one, under `code = "ROWS"`.
- `elapsed` (DURATION) — how long the step took, NULL wherever `rows` is. Wall clock, so it is a number to diagnose a slow query with rather than one to build a check on.
- `target` (STRING) — the query or relation it applies to, NULL when the emitter did not know one.
- `relix.rules` — the "OPTIMIZE" rows of `relix.events`, with the (constant) `stage` column dropped. The direct answer to "which rewrites fired?".

The feed is the *previous* statement's

`relix.events` holds the events of the statement that ran **before** the one querying it — never its own. A statement's events do not exist until it has been optimized, planned, and executed, and all three happen after the analysis that has to register the relation, so a statement cannot observe itself.

What follows from that:

- **In the REPL**, where each statement is re-analysed, `relix.events` shows the run immediately before. Run a query, then ask what it did.
- **In a one-shot script run**, the whole file is analysed once, before anything executes, so there is no earlier feed to carry in and `relix.events` is empty. Use `--trace` there.
- Each observed run **replaces** the window rather than adding to it, so the relation describes one run, not a session total.

That last point has a consequence worth stating outright: **the query that reads the feed is itself a run, and replaces it**. So a feed is readable once — after

```
:opt Report
query { relix.rules };           -- the firings
query { γ code, COUNT(*) → fired (relix.rules) }; -- empty: reads the reader
```

the second query sees the feed of the first, not of `:opt`. Slice a feed in one query, or re-run the statement that produced it.

The retained feed is capped, so a very long run keeps its most recent events; `seq` numbers what was retained.

Which stages appear

A run only reports what it actually did. A plain query in the REPL is planned and executed but not optimized, so it contributes `PLAN` and `EXECUTE` rows and no `OPTIMIZE` ones; `:opt` and `:explain` run the optimizer and contribute the `OPTIMIZE` rows that `relix.rules` reads. Asking for the feed does not cause work that the run would not otherwise do.

Technical Description

`relax.events` is a read-only system relation in the reserved `relax` namespace, like the rest of the catalog: a user definition can never redefine it, and it is never pushed down to a backend — the rows are engine state, not table data.

`relax.rules` is not computed separately. It is a *view* over the feed, in the same derived-over-primitive layering as `relax.dependencies` over `relax.plan`:

```
relax.rules =  $\pi$  seq, code, description, target ( $\sigma$  stage = "OPTIMIZE" (relax.events))
```

so the two cannot disagree about what fired. `seq` is kept rather than dropped because the feed carries no timestamp: arrival order is its only temporal fact, and a τ needs a stored key to sort on — the algebra guarantees no order that a scan did not declare.

`rows` is what makes the feed arithmetic rather than prose. A `description` is written for a person reading a trace, so a consumer that wants to *compute* with what a run observed — comparing an estimate against the cardinality actually produced — would otherwise have to read a number back out of a sentence. It is NULL for every event that measured nothing: a rule firing describes a rewrite, it does not count anything. `relax.rules` therefore has no `rows` column at all, since its slice is exactly the events that never carry one.

The count an executed query reports is the rows it **delivered**, which equals the query's cardinality whenever the result was read to the end. A query that matched nothing still reports, as 0 — "returned no rows" and "counted nothing" are different facts, and the column distinguishes them: 0 for the first, NULL for the second. So γ SUM(`rows`) (σ code = "ROWS" (relax.events)) totals a run's output without an empty result silently dropping out of the sum.

Three EXECUTE codes carry one, and they measure different things. "ROWS" is the query's own output, above. "SCAN" is a base relation read to its end — the relation's real size, which is why a scan stopped early by a λ reports nothing: that count would be about the consumer. "MATERIALIZE" is a blocking operator reporting what it buffered, named by its plan operator (Sort, Aggregate, Join) rather than by a relation, since the buffer belongs to the operator and not to anything a user named. It is how a query's memory becomes answerable: an operator that cannot emit a row before it has read its last holds its whole input, and nothing in the plan says how much that is. A join that buffers **both** sides emits two.

A query that **failed** part-way reports no count at all — not even a partial one. It delivered no result, so there is no cardinality to attribute to it, and a partial figure in the feed would be indistinguishable from a small one. The absence is the signal: a run whose feed carries fewer "ROWS" events than it ran queries had one fail.

Where the time went

`eLapsed` is the profiling half of the feed, and the three codes that carry it time deliberately different things.

"ROWS" is the query's own wall clock, from the first pull to the last, and is therefore **inclusive** — it covers every scan, buffer and join done on the query's behalf, because a pull-based engine does that work when it is asked and there is no interval that excludes the asking.

"SCAN" and "MATERIALIZER" are **self** time, and that distinction is the whole point of the column. Only the pull is timed: a scan reports what producing its rows cost, not what the join above it then did with them. Timed the other way, a scan under a join would report roughly the whole query and the feed would say the leaf was the problem in every plan ever written. So the numbers nest — each "SCAN" and "MATERIALIZER" is a share of its query's "ROWS" — and the arithmetic that answers *is the join the problem, or one of the scans?* is the query's total against the sum of the parts beneath it:

```
query { τ elapsed DESC (σ elapsed IS NOT NULL (reltx.events)) };
```

Two things follow from `elapsed` being wall clock. It varies run to run with the machine, the page cache and whatever else the host is doing, so it is read to find the operator worth looking at and not to certify that anything is fast enough. And it is a real `DURATION` rather than a count of milliseconds, so `SECONDS(elapsed)` and `τ elapsed DESC` work on it without a unit having to be agreed with whoever reads the query.

The column is `elapsed` and not `duration` because `DURATION` opens a temporal literal, so a column of that name could never be projected or sorted on.

`target` is `NULL` for an event no query owns, and `NULL` is a value the ordinary predicates treat as unknown, so `σ target = "X"` drops those rows. Use `σ target IS NOT NULL` when you want exactly the attributable ones.

Examples

Given a session that has just run a join:

```
Users := [| id | name |
          | 1 | Alice |];
Orders := [| id | amount |
           | 1 | 10 |];
query { Users ⋈ Orders };
```

The next statement can read what the engine decided, in order:

```
query { τ seq (π seq, stage, code, target (reltx.events)) };
-- seq stage code target
-- --- -----
-- 1 PLAN JOIN Joined
```

Whether one particular rewrite fired on the last run:

```
query { σ code = "SEL-001" (reltx.rules) };
```

How often each optimizer rule fired — the feed aggregated like any relation:

```
query { γ code, COUNT(*) → fired (reltx.rules) };
-- code      fired
-- -----
-- SEL-001    2
-- PROJ-003    1
```

How the work split across stages:

```
query { γ stage, COUNT(*) → decisions (reltx.events) };
-- stage      decisions
-- -----
-- OPTIMIZE    3
-- PLAN        1
```

Only the decisions attributable to a query, newest first:

```
query { τ seq DESC (σ target IS NOT NULL (relix.events)) };
```

Where the last run's time went, slowest step first — the steps that measured one, ordered by how long they took:

```
query { τ elapsed DESC (π target, code, elapsed (σ elapsed IS NOT NULL (relix.events))) };
```

Joining the feed to the structural catalog — which of the relations the optimizer touched are views rather than base tables:

```
Touched := { δ (π target→name (σ target IS NOT NULL (relix.events))) };  
query { σ kind = "QR" (Touched ⋈ relix.relations) };
```

See also

- `introspection stdlib` — `relix.unused` / `relix.deps` / `relix.impact` and friends, the structural half of the same idea.
- `relix help catalog` — every `relix.*` relation and its columns.
- `optimizer` — what the rule codes in `relix.rules` mean.

Component inventory (`relix.version`)

Syntax

```
relix.version -- what is installed in this process (component, kind, version)
```

Description

`relix.version` answers *what is running*: one row per component the process has loaded — the engine itself, the embedding API, and every discovered provider.

Providers are found by service discovery, which means a missing one fails nothing until a query needs it, and an unexpected one is invisible until it wins a name clash. Neither is visible to a compiler, a unit test, or a stack trace. This relation makes both visible, and makes them visible to a *query* — so the answer can be filtered, joined and printed with the operators you already have.

Columns:

- `component` (STRING) — what the thing calls itself: a function library's name, a solver's name, a connector's type token, a driver's class.
- `kind` (STRING) — one of "engine", "facade", "function-library", "connector", "solver", "driver".
- `version` (STRING) — the version the runtime can read, or "unknown".

Why a version can be `unknown`

A version is read from the packaging: the module descriptor when running as modules, the jar manifest when running from jars. A component built into neither — loose classes on a class path, most often a test or a development build — carries no version anywhere the runtime can ask for.

"unknown" is therefore the honest answer, and it is not a failure: the row is still there, and *which* implementations are loaded is the question this relation exists to answer. The version sharpens the answer; the name is the answer.

What each host contributes

The engine reports itself and the function libraries it resolves against, because it holds them. Connectors, solvers and JDBC drivers are reported by whatever assembled the process, because they live outside the engine — a JDBC driver most plainly, since the engine carries no database dependency at all.

A host that supplies nothing still gets the engine's own rows. That is the case for a bare engine embedding, and its inventory is complete for what it is: a process with no connectors installed reports no connectors, which is true.

Examples

The whole inventory:

```
query { relix.version };
```

Just the providers, engine and facade aside:

```
query { σ kind ∉ {"engine", "facade"} (relix.version) };
```

Whether anything can talk to a database at all — a question with two independent halves, the connector and the driver:

```
query { γ kind, COUNT(*) → installed (σ kind ∈ {"connector", "driver"} (relix.version)) };
```

Components whose version the runtime could not determine, which for a shipped build is a packaging problem worth knowing about:

```
query { π component, kind (σ version = "unknown" (relix.version)) };
```

Limitations

A version is only as good as the packaging it is read from; see *Why a version can be unknown* above.

The inventory is what is loaded *now*. A JDBC driver registered later in the run appears in a later statement's answer, because the rows are gathered per analysis rather than cached at start-up.

See also

- [observability feed](#) — `relix.events`, which answers what the last run *did* rather than what is installed.
- [introspection stdlib](#) — the structural catalogs.
- `relix help catalog` — every `relix.*` relation and its columns.

Candidate keys (`relix.keys`)

Syntax

```
relix.keys  -- the candidate keys of every relation (relation, key, ordinal, column)
```

Description

`relix.keys` is the collected candidate keys as data: one row per *column* of every candidate key of every relation.

It is the one statistic with no flat column to sit in. The scalar figures live where they belong — `row_count` on `relix.relations`, `distinct_count` and `null_count` on `relix.columns` — because a key is neither scalar nor per-column: it is a *list* of columns, and one relation may have several.

Columns:

- `relation` (STRING) — the relation the key belongs to.
- `key` (NUMBER) — which candidate key, numbered from 0 within each relation.
- `ordinal` (NUMBER) — the column's position *within* that key, from 0.
- `column` (STRING) — the column name.

A relation with one single-column key contributes one row, with `key` and `ordinal` both 0. A two-column key contributes two rows sharing a `key` and differing in `ordinal`.

Column order is data, not row order

`ordinal` is meaningful. The columns of a composite key are ordered, and the order is the one an index on that key would be built in — which decides whether a lookup on the first column alone can use it. Reading the columns of a key back in the order the rows happen to arrive would lose that, so the position is a column.

Where the keys come from

Two sources, and they make different claims:

- **An inline relation** contributes keys computed exactly from its rows: every column with no NULLs whose values are all distinct. This is a fact about *the data present*, not a constraint — in a two-row table, a column is a candidate key as soon as its two values differ.
- **A connection-backed table** contributes the key the database declares, read from the catalog when its schema is introspected. That one *is* a constraint.

The relation does not distinguish the two, because nothing in the collected statistics records which it was. Read a key over inline data as "nothing in these rows contradicts it".

Absent is not empty

A relation whose statistics were never collected contributes no rows — a view, a CSV or HTTP source that was not scanned. That is not the claim that it has no key; it is the absence of a claim. `relix.relations` tells the two apart: a relation with a `row_count` was measured.

Technical Description

The rows are read from the same `RelationStatistics` the cost model uses, so a key listed here is a key the optimizer is entitled to act on — whole-row distinctness and candidate-key derivation both consume it, which is what lets a redundant `δ` be removed over an input already known to be duplicate-free.

The extent is fixed at analysis time, like every structural catalog: it describes the script being analysed, and no query over it runs anything against a source.

Like the other catalogs, it describes only relations *outside* the reserved `relix` namespace. Use `relix.catalog` to see the namespace itself.

Examples

Every candidate key in the script:

```
query { relix.keys };
```

The keys of one relation, in key and column order:

```
query { τ key, ordinal (σ relation = "Orders" (relix.keys)) };
```

Which relations have a *composite* key — more than one column in the same key:

```
query { σ columns > 1 (γ relation, key, COUNT(column) → columns (relix.keys)) };
```

Whether a particular column takes part in any key:

```
query { δ (π relation (σ column = "customer_id" (relix.keys))) };
```

A relation that was measured but has no key at all — a join of `relix.relations` against the keys, keeping the rows with no match:

```
query {  
  π name (  
    σ row_count ≥ 0 (relix.relations) ▷ relix.relations.name = relix.keys.relation relix.keys  
  )  
};
```

Limitations

A key over inline data is observed rather than declared; see *Where the keys come from* above.

Only relations whose statistics were collected appear. A view never does: its distinctness is derived from its inputs by the optimizer rather than stored.

Boundedness is not here. Whether a relation is finite is not a statistic — it is proved rather than collected — and it is a scalar, so it sits on `relix.relations` as its own column.

See also

- `relation catalog` — `relix.relations`, which carries the scalar figures a key is not: `row_count` and `boundedness`.
- `introspection stdlib` — `relix.schema(rel)`, the columns of one relation, which pairs naturally with its keys.
- `reserved namespace` — `relix.catalog`.
- `relix help catalog` — every `relix.*` relation and its columns.

Reserved namespace (`relix.catalog`)

Syntax

```
relix.catalog  -- the relix.* surface itself (name, kind, arity, schema_code)
```


Description

`relix.catalog` is the reserved namespace describing itself: one row per `relix.*` relation and table-valued function.

Every other catalog describes *your* relations and deliberately excludes the reserved namespace — which is what keeps `relix.relations` about your script. The side effect was that the introspection surface was the one thing you could not introspect: to find out what `relix.*` holds, you had to read the documentation. This relation answers that from inside the engine.

Columns:

- `name` (STRING) — the qualified name, exactly as you would type it (`relix.plan`), because the use of a listing is that you can copy a row out of it.
- `kind` (STRING) — "SYS" for an engine-computed relation, "QR" for one of the `stdlib` views, "TVF" for a table-valued function. The same codes the IR prints.
- `arity` (NUMBER) — how many arguments a call passes. NULL for a relation, which is not called.
- `schema_code` (STRING) — the heading it yields, in the compact form `relix.plan.schema_code` also uses: `{name:S,kind:S,...}`.

It changes nothing else

Asking is naming it. `relix.relations`, `relix.columns`, `relix.functions` and `relix.keys` keep exactly the extents they had — user relations only. There is no flag and no mode; the reserved namespace is described by one relation, and you get it when you write it.

It lists itself

`relix.catalog` has a row of its own. A listing with a hole where the entry point is would be a poor listing, and there is no danger in it here: the extent is computed directly from what is registered, never derived from another catalog, so it cannot recurse.

There is no description column

What each relation is *for* is what this manual is. A one-line summary carried in the engine as well would be a second copy of that text with nothing keeping the two in step, so the relation carries the shape and leaves the meaning to the pages.

Technical Description

The extent is computed at analysis time from the symbols registered in the reserved namespace, so it reflects what this build actually ships rather than a list maintained beside it — a relation added to the catalog appears here without anyone remembering to add it.

`kind` distinguishes the two ways a `relix.*` relation is produced, which is worth reading: a `SYS` row is a relation the engine computes in Java, while a `QR` or `TVF` row is written in Relix and layered over those. `:source` in the REPL prints the definition of the second kind.

Examples

The whole reserved namespace:

```
query { relix.catalog };
```

Just the callable ones, with how many arguments each takes:

```
query { σ kind = "TVF" (relix.catalog) };
```

Which parts of the introspection surface are themselves written in Relix, rather than computed by the engine:

```
query { π name (σ kind ≠ "SYS" (relix.catalog)) };
```

How much of the surface there is, by kind:

```
query { γ kind, COUNT(name) → entries (relix.catalog) };
```

Everything that yields a relation name you could then look up — the entries whose heading mentions a `relation` column:

```
query { π name, schema_code (σ schema_code LIKE "%relation:%" (relix.catalog)) };
```

Limitations

It describes shape, not meaning; see *There is no description column* above.

`schema_code` is a rendered string rather than rows, so it is read rather than joined. The relational form of a heading is `relix.columns`, which covers user relations only.

Scalar functions are not here. This relation lists what lives in the reserved namespace, and the function library is not reserved — `relix.functions` lists every callable function, and `relix.funcs(cat)` filters it by category.

See also

- [introspection stdlib](#) — the `relix.*` views and TVFs this relation lists, and their definitions.
- [candidate keys](#) — `relix.keys`.
- [component inventory](#) — `relix.version`, which answers what is *installed* rather than what is reserved.
- `relix help catalog` — the same surface as prose, with examples.

3. Core operators

The everyday relational-algebra operators — the ones you reach for first.

| Operator | Unicode / ASCII | What it does |
|------------|----------------------------|--|
| Selection | σ / SELECT | Keep rows that match a condition |
| Projection | π / PROJECT | Pick, reorder, rename or compute columns |
| Rename | ρ / RENAME | Rename a relation or its columns |
| Distinct | δ / DISTINCT | Remove duplicate rows |
| Sort | τ / SORT / ORDER [BY] | Order rows by one or more keys |
| Limit | λ / LIMIT | Take the first N rows (with optional offset) |

Selection (σ / SELECT)

Syntax

```
 $\sigma$  <condition> (Relation)
SELECT <condition> (Relation)

 $\sigma$  age > 18 (Users)
SELECT age > 18 (Users)
```

Description

Selection keeps only the rows that match a condition and throws the rest away. Think of it as the filter on a spreadsheet: "show me only the rows where...". The relation that comes out has exactly the same columns as the one that went in — just fewer rows.

The condition can test a column against a value, combine several tests with AND / OR / NOT, check for missing values, or test membership in a set.

Technical Description

$\sigma_p(R)$ returns the subset of tuples of R for which predicate p evaluates to TRUE. A row whose predicate evaluates to NULL/UNKNOWN is dropped (three-valued logic). Selection is a streaming operator — it does not buffer the input — and the optimizer pushes it as close to the data source as possible, including down into SQL/Mongo backends as a WHERE / \$match clause.

Examples

Find adult users:

```
 $\sigma$  age >= 18 (Users)
```

Customers in a specific city, by exact match on text:

```
 $\sigma$  city = "London" (Customers)
```

Combine conditions — active accounts created this year:

```
 $\sigma$  active = true  $\wedge$  year = 2026 (Accounts)
```

Orders that are NOT cancelled:

```
σ ¬(status = "cancelled") (Orders)
```

Filter on a computed value — high-value line items:

```
σ (price * qty) > 1000 (LineItems)
```

Rows with a missing email address:

```
σ email = NULL (Users)
```

Members of a set of departments:

```
σ dept IN {"hr", "eng", "finance"} (Employees)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { σ amount > 100 (Sales) };
```

| region | rep | amount | bonus |
|--------|-----|--------|-------|
| east | Ada | 120 | 10 |
| west | Cy | 200 | 25 |

(2 rows)

Same four columns, fewer rows — selection never changes the schema. Note what happens to the two NULL `bonus` values: nothing. The condition tests `amount`, so the NULLs are irrelevant here. Had the condition been `bonus > 5`, Bo and Dee would have been **dropped**, because a comparison against NULL is UNKNOWN, not true — see [is-null](#) for how to keep them.

Limitations

A row whose condition is unknown because of a NULL is excluded, not kept. To keep NULLs you must test for them explicitly (e.g. `status = "x" v status = NULL`).

Alternatives

For "keep rows that have a match in another relation" use a SEMI join ($\bowtie$) rather than a selection with a subquery — `relix` has no scalar subqueries.

See Also

[project](#), [theta-join](#), [semi-join](#), [and](#), [or](#), [not](#), [is-null](#), [in](#)

Notes

Selection is one of the most heavily optimized operators: adjacent selections are merged, conjunctions are split so each part can be pushed independently, and selections slide below projections, renames, and into join inputs.

Projection (π / PROJECT)

Syntax

```
 $\pi$  <columns or expressions> (Relation)
PROJECT <columns or expressions> (Relation)
```

```
 $\pi$  name, age (Users)
 $\pi$  price * 1.05  $\rightarrow$  adjusted_price (Products)
```

Description

Projection chooses which columns appear in the result, in what order, and lets you rename or compute new ones. It is the "pick / rearrange / calculate columns" operator — the SELECT list of SQL. The number of rows is unchanged; only the shape of each row changes.

Any projected expression can be renamed with $\rightarrow$ (ASCII $\rightarrow$). Unaliased columns keep their original name, and you can freely mix plain columns with computed ones.

Technical Description

$\pi_L(R)$ returns R restricted to the attribute list L , where each list entry is an operand expression with an optional alias. Projection can drop columns, reorder them, apply scalar functions/arithmetic, and build nested struct/array values. Under set semantics projection can introduce duplicates (two distinct input rows may project to the same output row); *relix* preserves bag semantics unless a DISTINCT (δ) is applied. It is a streaming operator and pushes down to SQL/Mongo.

A **parenthesised condition** is an operand like any other, so a boolean can be projected directly rather than through `lf()`: π (amount > 100) $\rightarrow$ big (Orders) yields a BOOLEAN column. It carries three-valued logic with it — the column is NULL, not false, for a row whose amount is missing, because that is what the comparison is worth there.

Examples

Keep only a couple of columns:

```
 $\pi$  name, email (Users)
```

Reorder and rename for a report:

```
 $\pi$  id  $\rightarrow$  user_id, name  $\rightarrow$  user_name (Users)
```

Mix plain and aliased columns:

```
 $\pi$  id, name  $\rightarrow$  user_name (Users)
```

Compute a new column — add 5% to a price:

```
 $\pi$  name, price * 1.05  $\rightarrow$  adjusted_price (Products)
```

Apply a function while projecting:

```
π UCase(name) → name_upper (Customers)
```

Disambiguate a shared column name above a join with a relation qualifier — here `rooms.name` names the room's column even though `devices` also has a `name`:

```
π devices.name → device, rooms.name → room  
(devices ⋈ devices.room_id = rooms.room_id rooms)
```

A qualified reference resolves to exactly the column it names, or is a validation error — see [theta-join](#) for the full worked example.

The same dotted spelling reads a **field of a struct column** when it is not a relation qualifier — `π person.first` — and the two are tried in that order, so a relation qualifier always wins. See [struct-construction](#).

Reshape flat columns into a nested struct:

```
π id, { name, age } → person (Users)
```

Build an array column:

```
π label, [x, y] → coords (Points)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [  
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { π rep, amount * 2 → doubled (Sales) };
```

| rep | doubled |
|-----|---------|
| Ada | 240 |
| Bo | 160 |
| Cy | 400 |
| Dee | 100 |

(4 rows)

Four rows in, four rows out — projection changes the *columns*, never the row count. `region` and `bonus` are gone, and `doubled` is computed rather than copied, which is why projection is more than SQL's column list: any expression can appear, with `→` naming the result.

Limitations

Projection alone does not remove duplicate rows that result from dropping columns — follow with `δ` (DISTINCT) if you need unique rows. Nested struct/array construction is not pushed down to SQL backends.

Alternatives

To rename a relation or all its columns without choosing a subset, use `ρ` (RENAME). To aggregate rather than compute per-row, use `γ` (GROUP).

See Also

[select](#), [rename](#), [distinct](#), [group](#), [struct-construction](#), [array-construction](#)

Notes

The optimizer eliminates a projection that selects every column unchanged, merges consecutive projections, and pushes projections below selections to prune columns early.

Rename (ρ / RENAME)

Syntax

```
 $\rho$  <NewName> (Relation)                -- rename the relation only
 $\rho$  <NewName>(col1, col2, ...) (Relation)  -- rename relation and all columns
 $\rho$  <NewName> (old  $\rightarrow$  new, ...) (Relation) -- rename only the listed columns
 $\rho$  (old  $\rightarrow$  new, ...) (Relation)      -- rename columns, keep the relation name
RENAME ...

 $\rho$  E (Employees)
 $\rho$  E(eid, ename, dept_id) (Employees)
 $\rho$  E (name  $\rightarrow$  full_name) (Employees)
 $\rho$  (name  $\rightarrow$  full_name) (Employees)
```

Description

Rename gives a relation — and optionally some or all of its columns — new names. It is most useful when you need to join a relation to itself, or when two relations have clashing column names and you want to tell them apart.

There are three column forms:

- **relation only** — name the relation, leave every column untouched;
- **positional** — supply a full list of new column names (the list must match the column count exactly);
- **pairs** — `old  $\rightarrow$  new` renames only the columns you list; every unlisted column passes through unchanged. This is the textbook $\rho_{\{a \rightarrow b\}}(R)$ form, and it is the terse way to disambiguate a single colliding column in a wide relation.

The pair form and the positional form cannot be mixed in one list:

```
query {  $\rho$  Staff (id, name  $\rightarrow$  full_name) (Users) };
```

In the pair form the new relation name is optional: ρ (name $\rightarrow$ full_name) (R) renames the column in place, keeping the relation's existing name and each surviving column's origin qualifier.

Technical Description

$\rho_S(R)$ renames the relation to S ; $\rho_{S(a_1, \dots, a_n)}(R)$ additionally renames the attributes positionally (arity equality with R 's schema is required, or it is a validation error). $\rho_{S(a \rightarrow x, b \rightarrow y)}(R)$ renames exactly the named attributes and leaves the rest, matching source names case-insensitively. A rename in the pair form is a validation error when a source column does not exist, when the same source is renamed twice, or when a target name collides with another surviving or renamed column.

Rename is a streaming, order-preserving operator; a relation-only rename is transparent to the optimizer and to ordering derivation. Any column-renaming form (positional or pairs) invalidates the input's delivered ordering column names, so it is not treated as order-preserving.

Rename also **re-anchors column provenance**: after `p S (...)` every output column's relation qualifier becomes `S`, so a later qualified reference `S.column` resolves to it. This is what makes a self-join work — `p M(...) (Employees)` gives the second copy the qualifier `M`, so `M.mid` and `Employees.manager_id` refer to different sides even though both originate from `Employees`. When the pair form omits the new relation name, each renamed column keeps its existing origin qualifier under its new name.

Examples

Self-join needs two names for the same table — find employees who manage someone:

```
Mgr := { p M(mid, mname, mdept, mmanager, msalary) (Employees) };
query { π mname (Employees ⋈ Employees.manager_id = Mgr.mid Mgr) };
```

Give a result set tidy column names before exporting:

```
p Report(country, total_sales) (γ region, SUM(amount) → s (Sales))
```

Rename just the relation (columns kept) to disambiguate a union:

```
p Archived (OldOrders) u p Live (NewOrders)
```

Worked example — disambiguate one colliding column with a pair rename.

`devices` and `rooms` both carry a `name` column, so joining them makes `name` ambiguous. The pair form renames only `rooms.name`, leaving `room_id` alone, so the join reads cleanly — no need to restate every column of `rooms`.

devices

| device_id | room_id | name |
|-----------|---------|-------|
| 1 | 10 | Therm |
| 2 | 20 | Lamp |

rooms

| room_id | name |
|---------|------------|
| 10 | LivingRoom |
| 20 | Bedroom |

```
Joined := {
  π name → device, room_name → room
  (devices ⋈ devices.room_id = RoomsR.room_id
   (p RoomsR (name → room_name) (rooms)))
};
```

`p RoomsR (name → room_name) (rooms)` yields `room_id`, `room_name` (a two-column relation whose `room_id` is untouched), so the join key stays `room_id` and the two name columns are now distinct. The result:

| device | room |
|--------|------------|
| Therm | LivingRoom |
| Lamp | Bedroom |

Worked example — rename a field of a schema-on-read source. `people.json` holds documents that do not all carry the same fields:


```
[{"id": 1, "fullname": "Ada Lovelace", "city": "London"},
{"id": 2, "fullname": "Alan Turing"}]
```

```
source people from json("people.json");
```

```
query {  $\pi$  id, name, city (  $\rho$  (fullname  $\rightarrow$  name) (people)) };
```

Each document's `fullname` field is renamed to `name`. The second document carries no `city`, so that column is `NULL`:

| id | name | city |
|----|--------------|--------|
| 1 | Ada Lovelace | London |
| 2 | Alan Turing | NULL |

After the rename, `fullname` names nothing and reads `NULL`, and `people.name` names the renamed field. Renaming `fullname  $\rightarrow$  city` instead stops the query at the first document, which already carries a `city`.

Limitations

The positional form names columns **by position**, which a schema-on-read source (JSON, HTTP, MongoDB) does not have — each document carries the fields it carries, in its own order. Use the pair form there, `$\rho$  (old  $\rightarrow$  new, ...)`, which names the columns explicitly.

Over a schema-on-read source, which fields exist is a fact about each document, so the pair form is checked as each document is read rather than in advance. A document that lacks a listed field is passed through unchanged. A document that already carries a field under the new name stops the query with an error, as a declared relation would refuse the same rename before running.

Over a schema-on-read input, a name the rename hides is not refused afterwards. Over a declared relation, `$\pi$  E.name (  $\rho$  M (E) )` is an analysis error. Over a JSON, HTTP or MongoDB source, `E.name` is read as a path into a field called `E`, and yields `NULL`. Refer to the columns by the new name; see [theta-join](#) for a worked case.

The positional form must list exactly as many names as the relation has columns. To rename a subset, use the pair form `(  $\rho$  (old  $\rightarrow$  new, ...) (R) )` or projection with aliases `(  $\pi$  old  $\rightarrow$  new, ... )`. A pair rename must not rename a column onto a name that another column already keeps.

Alternatives

`$\pi$` with `$\rightarrow$` aliases renames selected columns and can also drop/reorder them. Use `$\rho$` when you want to keep all columns and only change names — the pair form makes a subset rename as terse as a projection alias while preserving column order and every unlisted column.

See Also

[project](#), [natural-join](#), [theta-join](#), [union](#)

Notes

Because a relation-only rename changes no data, the optimizer treats it as order- and distinctness-preserving and can push selections through it. A column-renaming rename (positional or pairs) still keeps row identity, so distinctness survives, but the optimizer rewrites a selection's column references through the rename when pushing it below.

Distinct (δ / DISTINCT)

Syntax

```
 $\delta$  (Relation)
DISTINCT (Relation)

 $\delta$  (Users)
DISTINCT (Users JOIN Orders)
```

Description

Distinct removes duplicate rows, so each unique row appears exactly once. It is the de-duplication step you reach for after a projection or a join has produced repeated rows — for example, "the list of distinct cities our customers live in".

Technical Description

$\delta(R)$ collapses R to set semantics: rows equal across all columns are merged to one. Two rows are equal when their values are, column by column, and a number is one value however it was written: 5 and 5.0 are the same number, so δ keeps one of them. Grouping, the set operations and a join key all read the same rule.

A column's type decides what makes two of its values one value. In a column declared `STRING`, the values are strings, so `'2024-01-15T10:00:00Z'` and `'2024-01-15T11:00:00+01:00'` are two values and δ keeps both — even though `=` matches either against the `TIMESTAMP` they denote, because that is a different question. Where a column is `ANY`, the type has settled nothing, so identity follows the same coercion `=` does: a column holding both `true` and the string that spells it holds one value, and δ keeps one row. This is the case that arises over schema-on-read sources, where one field can hold either. It is normally a blocking ([set]) operator, but becomes a single streaming pass (adjacent-distinct) when the input already delivers an ordering whose keys cover the whole row. The optimizer eliminates a δ over an input it can prove is already duplicate-free (DIST-001) — e.g. directly over a γ , a set operation, or a candidate-key-bearing source — and eliminates one *below* an aggregation that cannot tell duplicates apart (DIST-002).

Over a SQL source the δ is **pushed to the database** as `SELECT DISTINCT`, so the duplicates are never sent over the wire and the engine buffers no hash set. It folds after the select list (`DISTINCT` deduplicates the projected rows) and before `ORDER BY/LIMIT`, which apply to the deduplicated result. A δ over a `GROUP BY` is not rendered — γ already emits one row per key, so DIST-001 has removed it.

Examples

The distinct set of cities customers live in:

```
 $\delta$  ( $\pi$  city (Customers))
```

Unique (product, region) pairs that have ever sold:

```
δ (π product, region (Sales))
```

Drop duplicates created by a join:

```
δ (π name (Customers ⋈ Orders))
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { δ (π region (Sales)) };
```

```
region
-----
east
west
(2 rows)
```

Four rows collapse to two. Note the order of operations: the projection to region comes **first**, creating the duplicates that δ then removes — over the full Sales relation every row is already unique, so δ (Sales) would change nothing. That is the usual shape: project to the columns you care about, then deduplicate.

Limitations

Distinct compares whole rows — to dedupe on a subset of columns, project to those columns first. Over an unbounded generator δ needs a bound below it unless the input is already ordered.

Alternatives

γ (GROUP) on the columns you want unique achieves the same and lets you aggregate at the same time. UNION (u) deduplicates as part of combining two relations.

See Also

[project](#), [group](#), [union](#), [sort](#)

Notes

The optimizer often deletes a redundant δ entirely; check --optimize output for a DIST-001 or DIST-002 record. When kept over a database source it becomes SELECT DISTINCT (visible in --explain as a PushedScan); when kept in the engine, an ordered input turns it into a cheap streaming pass (Distinct streaming).

Sort (τ / SORT / ORDER [BY])

Syntax

```
 $\tau$  <key> [ASC|DESC] [, <key> [ASC|DESC]] ... (Relation)
SORT ...
ORDER ...           -- SQL-prior synonym
ORDER BY ...        -- SQL-prior synonym; BY is an optional noise word

 $\tau$  name (Users)           -- ascending by default
 $\tau$  salary DESC (Employees)
 $\tau$  dept ASC, salary DESC (Employees)
 $\tau$  to_timestamp(logged) DESC (RawLogs)  -- sort by a computed value
ORDER BY salary DESC (Employees)        -- same AST as  $\tau$  salary DESC (Employees)
```

Description

Sort orders the rows of a relation by one or more keys. Each sort key can go ascending (ASC, the default) or descending (DESC). When you sort by several keys, the first is the primary order, the next breaks ties, and so on — just like sorting a spreadsheet by multiple columns.

A sort key can be a bare column or a full expression — e.g. `to_timestamp(logged)` or `price * qty` — so you can order by a computed value without projecting it first.

Sorting is most often combined with LIMIT to answer "top N" questions.

Technical Description

τ establishes a delivered ordering over R: a sequence of (expression, direction) sort specifications. It is a blocking operator ([sort]) that buffers its input. The cost model tracks the delivered order so the optimizer can (a) eliminate a redundant τ whose input already satisfies the order (SORT-001), (b) feed a merge join without re-sorting, and (c) enable streaming δ/γ — all reasoned about over bare-column keys; a derived key (e.g. `to_timestamp(logged)`) sorts in-engine and does not participate in ordering-based rewrites or ORDER BY pushdown. A τ by bare columns over a single backend connection pushes down as ORDER BY, and a pushed ORDER BY can satisfy a downstream merge join (source-sorted merge).

Within a type, values order as that type orders. Strings order by **code point** — the order UTF-8 bytes sort in, and the order a SQL binary collation gives — rather than by the UTF-16 code unit a Java `char` is, so a character outside the basic multilingual plane sorts after every character in it rather than in the middle. For text inside that plane, which is all of ASCII and every 8-bit character set, the two rules are the same rule. NULLs sort last in both directions.

τ is not the only source of a delivered order: a merge join emits its output in join-key order, so a chain of merge joins on the same key sorts once rather than once per join, and a δ or γ over that output can stream. A semi merge join carries its left input's order through unchanged; an anti merge join advertises no order, because it emits left rows with a NULL join key ahead of the sorted run.

Examples

Alphabetical list of users:

```
τ name (Users)
```

Highest-paid employees first:

```
τ salary DESC (Employees)
```

Group by department, then highest salary within each:

```
τ dept ASC, salary DESC (Employees)
```

Top 10 orders by amount (sort then limit):

```
λ 10 (τ amount DESC (Orders))
```

Most recent events first (temporal sort):

```
τ occurred_at DESC (Events)
```

Sort by a parsed timestamp — order raw string logs by their real instant:

```
τ to_timestamp(logged) DESC (RawLogs)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { τ amount DESC (Sales) };
```

| region | rep | amount | bonus |
|--------|-----|--------|-------|
| west | Cy | 200 | 25 |
| east | Ada | 120 | 10 |
| east | Bo | 80 | NULL |
| west | Dee | 50 | NULL |

(4 rows)

The same four rows in a new order. Sorting is the one core operator that must see every row before it can emit the first, which is why it is tagged `[sort]` in the IR and why an unnecessary `τ` is worth removing — the optimizer does exactly that (`Sort-001`) when the input already arrives in the required order.

Where the NULLs go. Sorting by a column that has some nulls settles the rule, which is otherwise the kind of thing every database answers differently:

```
query { τ bonus DESC (Sales) };
```

| region | rep | amount | bonus |
|--------|-----|--------|-------|
| west | Cy | 200 | 25 |
| east | Ada | 120 | 10 |
| east | Bo | 80 | NULL |
| west | Dee | 50 | NULL |

(4 rows)

NULLs sort **last**, and last in *both* directions — ascending or descending, a missing value goes to the end rather than swapping ends with the sort. This holds whether the τ runs in-engine or is pushed into a database: the SQL that comes out states the placement explicitly rather than inheriting the backend's default, which would otherwise put a query's NULLs first on one connection and last on another.

Limitations

Sort must see all rows before emitting any, so it cannot run over an unbounded generator without a bound below it. NULL placement in merge joins is still being refined; for plain τ , NULLs sort last in both directions, in-engine and pushed alike.

Alternatives

For "N highest per group" use TOP ... PER instead of sorting the whole relation. For a single extreme value per group use γ with MIN/MAX or ARGMAX/ARGMIN.

See Also

[limit](#), [top](#), [group](#), [distinct](#)

Notes

Sorting can be free when the data already arrives ordered (e.g. an indexed source pushed as ORDER BY); the optimizer removes the in-engine sort in that case.

Limit (λ / LIMIT)

Syntax

```
 $\lambda$  <count> (Relation)           -- first N rows
 $\lambda$  <offset>, <count> (Relation) -- skip offset rows, then take count
LIMIT ...

 $\lambda$  10 (Orders)
 $\lambda$  20, 10 (Orders)             -- skip 20, take 10 (pagination)
```

Description

Limit restricts how many rows come out. The one-number form keeps the first N rows; the two-number form skips some rows first and then keeps the next batch — exactly what you need for paging through results a screenful at a time.

Limit is almost always paired with SORT so that "first N" means something predictable (e.g. the 10 largest, the 5 most recent).

Technical Description

λ applies an optional offset and a row cap to its input. It is a streaming operator that stops pulling once the cap is reached — which is what makes it safe over an unbounded generator (it bounds the stream). The optimizer pushes λ below projections (LIM-001) and folds it into SQL as LIMIT/OFFSET (dialect-aware) or Mongo as \$skip/\$limit.

Examples

First 10 orders:

```
λ 10 (Orders)
```

Page 3 of results, 10 per page (skip 20, take 10):

```
λ 20, 10 (Orders)
```

Top 10 salaries (sort then limit):

```
λ 10 (τ salary DESC (Employees))
```

Take a finite prefix of an infinite generator — first 100 prime numbers:

```
λ 100 (Primes)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { λ 2 (Sales) };
```

| region | rep | amount | bonus |
|--------|-----|--------|-------|
| east | Ada | 120 | 10 |
| east | Bo | 80 | NULL |

(2 rows)

The **first** two rows — and "first" is only meaningful if the input has an order. Without a `τ` beneath it, a limit takes whatever the source happens to deliver, which for a database table is not guaranteed to be stable between runs. `τ amount DESC (Sales)` then `λ 2` is the "top 2" idiom; for top-N *per group* reach for **TOP ... PER** instead. Writing the idiom costs nothing: the optimizer fuses `λ` over `τ` into a bounded top-N rather than sorting the whole input (LIM-003, see the [optimizer](#) page).

Limitations

Without a `SORT` below it, "first N" is whatever order the source happens to deliver — not guaranteed. Limit caps rows but does not sample randomly; for that use `SAMPLE`.

The count and the offset are whole numbers. A fraction, or a number too large for a 64-bit integer, is a parse error:

```
query { λ 2.5 (Orders) };
```

Alternatives

For a random subset use `SAMPLE p` (probabilistic) or `SAMPLE n ROWS` (exact count). For "N per group" rather than N overall use **TOP ... PER**.

See Also

[sort](#), [top](#), [sample-bernoulli](#), [sample-reservoir](#)

Notes

λ is the canonical "bound" that rescues an otherwise-blocking query over an unbounded source: `γ count (λ 1000 (Primes))` is allowed, `γ count (Primes)` is a plan-time error.

4. Predicates

The conditions that go inside a σ (selection) or join.

| Page | Unicode / ASCII | What it does |
|----------------|---|--|
| Comparison | <code>=</code> <code>≠</code> <code><</code> <code>≤</code> <code>></code> <code>≥</code> | Compare two values |
| And | <code>∧</code> / AND | Both conditions hold |
| Or | <code>∨</code> / OR | Either condition holds |
| Not | <code>¬</code> / NOT | Negate a condition |
| Null test | <code>⊥</code> / NULL / IS [NOT] NULL | Test for a missing value |
| Set membership | <code>∈</code> / IN, <code>∉</code> / NOT IN | Test membership in a set |
| Pattern match | LIKE, NOT LIKE | Wildcard string match (<code>%</code> = any run, <code>_</code> = any one char) |

Comparison Operators (`=` `≠` `<` `≤` `>` `≥`)

Syntax

```
attr = value      attr != value    (or attr ≠ value)
attr < value      attr > value
attr <= value     attr >= value    (or attr ≤ / ≥ value)
```

```
 $\sigma$  age >= 18 (Users)
 $\sigma$  status != "cancelled" (Orders)
```

Description

Comparison operators are the building blocks of conditions in selection (σ) and join conditions. They test how one value relates to another: equal, not equal, less than, greater than, and the "or equal to" variants. The values can be columns, numbers, strings, booleans, or temporal values.

Technical Description

Comparisons form ComparisonPredicates over two operands. `=` and `≠` test equality; `<` `≤` `>` `≥` require ordered types and work on numbers, strings (lexicographic), and temporal types (one temporal type on both sides, or a temporal against a text cell holding its ISO-8601 form). A comparison involving a NULL evaluates to UNKNOWN (three-valued logic), so the row is dropped by σ . ASCII forms: `!=` for `≠`, `<=` for `≤`, `>=` for `≥`.

Examples

Numeric thresholds:

```
 $\sigma$  age >= 18 (Users)
```

And the strict form:

```
 $\sigma$  price < 100 (Products)
```

Exact text match:

```
 $\sigma$  status = "active" (Orders)
```

Mismatch — note that a NULL status matches neither form:

```
σ status != "cancelled" (Orders)
```

Boolean flag:

```
σ verified = true (Accounts)
```

Temporal comparison — `created_at` may be a `TIMESTAMP` column, or a text column holding ISO-8601 timestamps (as every inline-table, CSV, and JSON cell is):

```
σ created_at >= TIMESTAMP '2026-01-01T00:00:00Z' (Events)
```

Compare two columns:

```
σ spent > budget (Departments)
```

Limitations

Ordered comparisons require both operands to be the same comparable type — you cannot compare a `NUMBER` to a `STRING`, or two different temporal types (a `DATE` against a `TIMESTAMP`). Text that spells a value counts as that value, not as a `STRING`, so a temporal-shaped or boolean-shaped string compares with the typed form; by the same rule it does *not* compare with plain text, and ordering "2024-01-15" against "pending" is a `DATE` against a `STRING` and is an error. Cast the column to settle what it holds. `NULL` on either side makes the comparison `UNKNOWN` (row excluded); test for `NULL` explicitly with `= NULL` / `= ⊥`.

Alternatives

For "is this missing" use the `NULL` test (`= ⊥` / `= NULL`). For set membership use `IN` (`∈`).

See Also

[select](#), [and](#), [or](#), [not](#), [is-null](#), [in](#)

Notes

String comparison is lexicographic; temporal comparison is chronological. Which of the two a text cell gets is decided by what the text spells: a cell holding `2024-01-15T11:00:00+01:00` is the instant it denotes and is ordered chronologically, so it sorts *before* `2024-01-15T10:30:00Z` even though it sorts after it as text. Only text that spells no value is ordered as text.

A comparison is a boolean **value**, so besides forming the condition of `σ` / a join it may appear in **operand position** — most usefully as a function argument. That is what lets a conditional receive its test directly, e.g. `IIf(price > 100, "expensive", "cheap")`. The comparison (optionally combined with `∧` / `∨` / `¬`, or written as `IN` / `LIKE` / `= ⊥`) is evaluated per row to true/false with the same semantics `σ` uses. A bare column argument stays a plain operand — the comparison form is only entered when a comparison-style operator follows.

A boolean-shaped `STRING` **is** a `BOOLEAN`. Inline-table, CSV, and JSON cells holding `true/false` infer as `STRING`, but a query naturally writes `= true/= false` (a boolean literal), so the text is read as the boolean it names, case-insensitively — `true/TRUE/True` are one value. Hence `σ is_active = true` (Rules) matches a row whose `is_active` cell is the string "true". A string that names no boolean stays a `STRING` and simply does not match (`≠/≠`).

never error, whatever the types). This mirrors the string→temporal reading for ISO-8601 cells.

What a string means does **not** depend on what it is compared against: it is read the same way whether its partner is typed, another string, or nothing at all (a hash-join key is canonicalised before it is known which row it will meet). That is what makes the ordering a genuine one — under the older, partner-dependent rule a `TIMESTAMP` equalled two spellings that did not equal each other, which left the result of `=` over such a column dependent on the order the rows arrived in.

Logical AND (`^` / `AND`)

Syntax

```
<condition1> ^ <condition2>
<condition1> AND <condition2>

σ age > 18 ^ active = true (Users)
```

Description

`AND` combines two conditions and keeps a row only when **BOTH** are true. Use it to narrow a filter — "adults who are also active", "orders that are paid and shipped". Chain as many as you like.

Technical Description

`^` forms an `AndPredicate`. Under three-valued logic, `TRUE ^ TRUE = TRUE`; any `FALSE` makes it `FALSE`; otherwise `UNKNOWN`. `AND` binds tighter than `OR` and looser than `NOT`. The optimizer splits a conjunctive selection into separate selections so each conjunct can be pushed independently toward its source (SEL-001).

Examples

Two conditions both required:

```
σ age > 18 ^ active = true (Users)
```

Three-way conjunction:

```
σ region = "EMEA" ^ amount > 1000 ^ status = "paid" (Orders)
```

In a join condition:

```
A ⋈ A.id = B.id ^ A.active = true B
```

Limitations

With `NULLs`, a row where one side is `UNKNOWN` is not kept unless the other side makes the whole expression `FALSE-or-TRUE` per three-valued logic. Use parentheses when mixing with `OR` to make precedence explicit.

Alternatives

`OR` (`v`) for "either condition". `NOT` (`¬`) to negate. `IN` (`∈`) is cleaner than a long chain of `= ... v` equalities.

See Also

[or](#), [not](#), [comparison](#), [select](#)

Notes

Because conjuncts are split and pushed independently, writing one σ with several ANDs is as efficient as nesting separate σ operators.

Logical OR ($\vee$ / OR)

Syntax

```
<condition1>  $\vee$  <condition2>  
<condition1> OR <condition2>
```

```
 $\sigma$  status = "new"  $\vee$  status = "pending" (Orders)
```

Description

OR combines two conditions and keeps a row when EITHER is true. Use it to widen a filter — "orders that are new or pending", "users in sales or marketing".

Technical Description

$\vee$ forms an OrPredicate. Under three-valued logic, $\text{FALSE} \vee \text{FALSE} = \text{FALSE}$; any TRUE makes it TRUE; otherwise UNKNOWN. OR binds looser than AND, so $a \wedge b \vee c$ means $(a \wedge b) \vee c$; use parentheses to be explicit.

Examples

Either of two statuses:

```
 $\sigma$  status = "new"  $\vee$  status = "pending" (Orders)
```

Mix with AND (parenthesise for clarity):

```
 $\sigma$  (region = "EMEA"  $\vee$  region = "APAC")  $\wedge$  amount > 1000 (Orders)
```

Keep rows where a value is present OR explicitly null:

```
 $\sigma$  score >= 50  $\vee$  score = NULL (Results)
```

Limitations

A disjunction over the same column equality is more cleanly written with IN ($\in$).
Precedence with AND can surprise — parenthesise when mixing.

Alternatives

IN ($\in$) for many equality alternatives on one column. UNION ($\cup$) to combine the results of two separately-filtered queries.

See Also

[and](#), [not](#), [in](#), [comparison](#), [select](#)

Notes

$x = a \vee x = b \vee x = c$ is equivalent to, and clearer as, $x \in \{a, b, c\}$.

Logical NOT ($\neg$ / NOT)

Syntax

```
 $\neg$ (<condition>)  
NOT (<condition>)  
  
 $\sigma$   $\neg$ (status = "inactive") (Users)  
 $\sigma$  NOT (status = "inactive") (Users)
```

Description

NOT flips a condition: it keeps the rows that do NOT satisfy it. Use it to exclude — "everything except inactive users", "anything that isn't a refund".

Technical Description

$\neg$ forms a NotPredicate and is the tightest-binding logical operator (applied before AND/OR). Under three-valued logic, NOT TRUE = FALSE, NOT FALSE = TRUE, NOT UNKNOWN = UNKNOWN — so negating a comparison against a NULL still yields UNKNOWN and the row is dropped. A double negation is simplified away by the optimizer (PRED-002).

Examples

Exclude a status:

```
 $\sigma$   $\neg$ (status = "inactive") (Users)
```

Negate a compound condition:

```
 $\sigma$  NOT (region = "EMEA"  $\wedge$  amount < 100) (Orders)
```

Often clearer to use the opposite comparison:

```
 $\sigma$  status != "inactive" (Users)    -- same intent, no NOT
```

Worked Example

Negation is the place NULL stops behaving like an ordinary value, so it is worth seeing once. Cara has never ordered, so her `amount` after the outer join is NULL:

```
Customers := [  
customer_id	name
1	Alice
2	Bob
3	Cara
];  
  
Orders := [  
order_id	customer_id	amount
100	1	120
101	2	45
];  
  
WithOrders := { Customers  $\bowtie$  Customers.customer_id = Orders.customer_id Orders };  
  
query {  $\sigma$  amount > 100 (WithOrders) };
```

| customer_id | name | order_id | customer_id_r | amount |
|-------------|-------|----------|---------------|--------|
| 1 | Alice | 100 | 1 | 120 |

(1 row)

```
query { σ ¬(amount > 100) (WithOrders) };
```

| customer_id | name | order_id | customer_id_r | amount |
|-------------|------|----------|---------------|--------|
| 2 | Bob | 101 | 2 | 45 |

(1 row)

One row and one row, out of three. **Cara is in neither result.** `amount > 100` is UNKNOWN for her — there is no amount to compare — and `¬UNKNOWN` is still UNKNOWN, which a selection drops just as it drops false. Negating a condition does not turn "we do not know" into "no".

This is worth internalising because it breaks an assumption that holds everywhere else: `σ p` and `σ ¬p` do **not** partition a relation between them. If you need the missing rows on one side, ask for them:

```
query { σ ¬(amount > 100) ∨ amount = NULL (WithOrders) };
```

| customer_id | name | order_id | customer_id_r | amount |
|-------------|------|----------|---------------|--------|
| 2 | Bob | 101 | 2 | 45 |
| 3 | Cara | NULL | NULL | NULL |

(2 rows)

Now the two halves add up.

Limitations

NOT does not turn NULLs into matches — `NOT (x = 5)` excludes rows where `x` is NULL. To include NULLs, add an explicit `∨ x = NULL`. Parenthesise the operand.

Alternatives

Use the direct negative comparison (`≠`, `<`, `>`) where possible. For "no match in another relation" an ANTI join (`▷`) is the relational form of NOT EXISTS.

See Also

[and](#), [or](#), [comparison](#), [anti-join](#), [is-null](#)

Notes

`¬(¬ p)` collapses to `p`; the optimizer removes such double negations.

Null Test (`⊥` / NULL)

Syntax

```
attr = ⊥   attr = NULL   attr IS NULL       -- IS NULL
attr ≠ ⊥   attr != NULL  attr IS NOT NULL    -- IS NOT NULL

σ email = NULL (Users)
σ email IS NULL (Users)
σ email IS NOT NULL (Users)
```

Description

The null test checks whether a value is missing. Comparing a column to $\perp$ (or the ASCII keyword NULL, or the SQL-prior `IS NULL`) keeps rows where the value is absent; the not-equal / `IS NOT NULL` form keeps rows where a value is present. This is how you find records with missing fields, or filter them out.

Technical Description

`attr =  $\perp$  / attr = NULL / attr IS NULL` form a NullPredicate testing `IS NULL`; `attr  $\neq$   $\perp$  / attr != NULL / attr IS NOT NULL` test `IS NOT NULL` — all three spellings parse to the identical AST. This is distinct from an ordinary comparison: because a normal `= value` against a NULL yields UNKNOWN, missing values can only be detected with the dedicated null test. $\perp$ is U+22A5.

Examples

Rows with a missing email:

```
σ email = NULL (Users)
```

Rows that DO have a value:

```
σ phone != NULL (Contacts)
```

Combine — present email but missing phone:

```
σ email != NULL ∧ phone = NULL (Contacts)
```

Find unmatched rows after a left outer join:

```
σ Orders.order_id = NULL  
  (Customers ⋈ Customers.customer_id = Orders.customer_id Orders)
```

Worked Example

A left outer join is the usual way a NULL appears in the middle of a query, so it is also the clearest place to see what a null test does. Cara has never ordered:

```
Customers := [  
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];		
  
Orders := [  
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
];		
  
WithOrders := { Customers ⋈ Customers.customer_id = Orders.customer_id Orders };  
  
query { σ order_id = NULL (WithOrders) };  
  
  customer_id  name  city  order_id  customer_id_r  amount  
  -----  -----  -----  -----  -----  -----
```

| | | | | | |
|---|------|--------|------|------|------|
| 3 | Cara | Lisbon | NULL | NULL | NULL |
|---|------|--------|------|------|------|

(1 row)

That is the "customers who have never ordered" idiom: outer join, then keep the rows the right side failed to fill in.

Why an ordinary comparison cannot do this. A comparison never matches a NULL, whichever way round it is written. Ask for the rows whose amount *is* 120:

```
query { σ amount = 120 (WithOrders) };
```

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 100 | 1 | 120 |

(1 row)

then for the rows whose amount is *not* 120:

```
query { σ amount ≠ 120 (WithOrders) };
```

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 101 | 1 | 80 |
| 2 | Bob | Berlin | 102 | 2 | 45 |

(2 rows)

One row and two rows — and Cara is in neither, though every row of `WithOrders` is in one or the other of those two questions as a human would read them. Her amount is missing, so neither "is it 120?" nor "is it something else?" has an answer, and `σ` keeps only rows that answer yes. A missing value is invisible to comparison, which is why the null test exists.

Limitations

You must use the null test for missing values — an ordinary `= value` comparison never matches a NULL (it is UNKNOWN). The Unicode $\perp$, the keyword NULL, and the SQL-prior `IS [NOT] NULL` are interchangeable.

Alternatives

The `IsNull()` scalar function returns a BOOLEAN you can use in expressions; `Nz()` and `Coalesce()` substitute a default for a NULL.

A null test also applies to a **parenthesised condition**, and that is how you ask whether a comparison came out UNKNOWN rather than false:

```
(amount > 100) = ⊥      -- the comparison could not be decided
```

A null test on a *value* asks whether the value is missing; a null test on a *condition* asks about its truth value, which is NULL exactly when the condition is UNKNOWN. The two stop being the same question as soon as the condition reads more than one column.

`IsNull((amount > 100))` is the same test written as a function call, for a place that wants a BOOLEAN rather than a predicate.

It matters because negation does not give back the rows a filter dropped. `σ ¬(amount > 100)` keeps the rows whose amount is 100 or less and **not** the rows whose amount is missing, because `¬UNKNOWN` is UNKNOWN and a selection keeps only what is true. To keep both:

```
σ ¬(amount > 100) ∨ (amount > 100) = ⊥ (Orders)
```


For a condition over one column you can spell the same thing with a null test on that column (`amount = NULL` v `¬(amount > 100)`). Testing the condition itself is the general form, and the one that stays right as the condition grows: a comparison between two columns is UNKNOWN when *either* side is missing, and a conjunction is UNKNOWN in more cases than either of its parts.

See Also

[comparison](#), [isnull](#), [nz](#), [coalesce](#), [left-outer-join](#)

Notes

The "find the missing ones" idiom — outer join then `σ key = NULL` — is the most common use; an ANTI join expresses the same thing more directly.

Set Membership ($\in$ / IN, $\notin$ / NOT IN)

Syntax

```
attr ∈ { value1, value2, ... }      attr IN { value1, value2, ... }
attr ∉ { value1, value2, ... }      attr NOT IN { value1, value2, ... }

σ dept ∈ {"hr", "eng"} (Employees)
σ dept NOT IN {"mgmt"} (Employees)
```

Description

The membership test keeps rows whose value is one of a listed set ($\in$ / IN), or is NOT one of them ($\notin$ / NOT IN). It is the tidy way to write "status is new, pending, or paid" without a long chain of OR equalities.

Technical Description

$\in$ forms an `ElementOfPredicate` against a set literal `{...}`; $\notin$ is its negated form. Equivalent to a disjunction of equalities but expressed once. ASCII: IN, and NOT IN (two tokens). The set literal holds constant operands.

Examples

One of several departments:

```
σ dept ∈ {"hr", "eng", "finance"} (Employees)
```

Exclude a set of statuses:

```
σ status NOT IN {"cancelled", "refunded"} (Orders)
```

Numeric set:

```
σ priority IN {1, 2, 3} (Tickets)
```

Worked Example

One membership test standing in for a chain of equalities:

```
Tickets := [
id	priority	status
1	1	open
]
```

```
2	2	cancelled
3	3	open
4	5	refunded
];
```

```
query { σ priority ∈ {1, 2, 3} (Tickets) };
```

| id | priority | status |
|----|----------|-----------|
| 1 | 1 | open |
| 2 | 2 | cancelled |
| 3 | 3 | open |

(3 rows)

The same rows come back from $\sigma \text{ priority} = 1 \vee \text{priority} = 2 \vee \text{priority} = 3$; the set literal is the readable spelling of that disjunction.

$\notin$ is the exclusion form, and reads the way you would say it out loud — "not cancelled or refunded":

```
query { σ status ∉ {"cancelled", "refunded"} (Tickets) };
```

| id | priority | status |
|----|----------|--------|
| 1 | 1 | open |
| 3 | 3 | open |

(2 rows)

Limitations

The set is a literal list of constant values, not a subquery — to test membership in another relation's values, use a SEMI join ($\bowtie$) instead. A NULL value on either side makes the test UNKNOWN and the row is dropped: a NULL column is neither in the set nor out of it, and if the set holds a NULL then failing to match the other elements settles nothing — the unknown one might have been the value — so $\notin$ keeps that row no more than $\in$ does.

Alternatives

A SEMI join ($\bowtie$) for "value appears in another relation". A chain of $= \dots \vee = \dots$ is the long-hand form IN replaces.

See Also

[or](#), [comparison](#), [semi-join](#), [select](#)

Notes

Prefer IN over a long OR chain for readability; for dynamic sets drawn from data, reach for a semi join.

Pattern Match (LIKE / NOT LIKE)

Syntax

```
attr LIKE "pattern"
attr NOT LIKE "pattern"
attr LIKE <string expression>      -- the pattern can also be a column or expression

σ name LIKE "A%" (Customers)
σ email NOT LIKE "%@example.com" (Users)
```

Description

LIKE keeps rows where a string column matches a wildcard pattern. The pattern can use `%` to stand for any sequence of characters (including none) and `_` to stand for exactly one character. NOT LIKE keeps rows that do NOT match. This is the standard way to filter by a partial string without needing a full regular expression.

Technical Description

LIKE forms a PatternPredicate over two operands: the value and the pattern, which may be any string expression. Wildcards: `%` translates to `.*` (any run of characters) and `_` to `.` (any single character); all other regex metacharacters in the pattern are escaped, so the pattern is treated as a literal substring except for `%` and `_`. Matching is case-sensitive and anchored (the whole value must match). NULL on either side makes the predicate UNKNOWN, and a selection keeps only rows that are true — so a NULL value is dropped by LIKE and by NOT LIKE alike, rather than falling through to the negative side.

Examples

Names starting with a capital A:

```
σ name LIKE "A%" (Customers)
```

Email addresses that end with a specific domain:

```
σ email LIKE "%@acme.com" (Users)
```

Exactly five-character product codes:

```
σ code LIKE "_____" (Products)
```

Codes containing the substring "ERR":

```
σ message LIKE "%ERR%" (Logs)
```

Exclude internal test accounts:

```
σ email NOT LIKE "%@test.%" (Accounts)
```

Combine with other predicates:

```
σ name LIKE "Smith%" ∧ active = true (Customers)
```

Worked Example

The three wildcards side by side on one set of product codes:

```
Products := [
sku	name
```

```
AB-1001	Widget
AB-1002	Widget Large
XY-1001	Bracket
AB-2001-X	Widget Deluxe
];
```

```
query { σ sku LIKE "AB-%" (Products) };
```

| sku | name |
|-----------|---------------|
| AB-1001 | Widget |
| AB-1002 | Widget Large |
| AB-2001-X | Widget Deluxe |

(3 rows)

% stands for any run of characters, including none. Matching is **anchored** — the whole value must match the pattern — which is why "AB-%" means "starts with AB-" rather than "contains it". To ask for a substring, put a % on both ends.

_ stands for exactly one character, so a pattern of fixed length filters by width:

```
query { σ sku LIKE "AB-____" (Products) };
```

| sku | name |
|---------|--------------|
| AB-1001 | Widget |
| AB-1002 | Widget Large |

(2 rows)

Seven characters exactly, and the right seven. XY-1001 is the correct length but the wrong prefix; AB-2001-X has the right prefix but is too long. Both are dropped, because an anchored pattern has to account for the whole value.

The pattern does not have to be a literal. It can be any string expression, such as a column, so the patterns can come from data. Here each family's prefix lives in a table of rules:

```
Rules := [
family	pattern
AB	AB-%
XY	XY-%
];
```

```
query { π family, sku (σ sku LIKE pattern (Products × Rules)) };
```

| family | sku |
|--------|-----------|
| AB | AB-1001 |
| AB | AB-1002 |
| XY | XY-1001 |
| AB | AB-2001-X |

(4 rows)

Limitations

A pattern that is not a string literal is not pushed down to MongoDB, so that predicate runs in the engine. SQL sources receive it as written. Matching is case-sensitive; use UCase() or LCase() on both sides for case-insensitive matching:

```
σ UCase(name) LIKE "ALICE%" (Users)
```

The only special characters are `%` (any run) and `_` (any one character); there is no escape character for a literal `%` or `_` in the pattern. If you need to match a literal `%` or `_`, use the `Replace()` function to pre-process the value, or anchor with comparison operators instead.

Alternatives

For an exact match, use the equality comparison `= "value"`. For membership in a fixed set of values, use `IN (€)`. For more complex patterns such as word boundaries or alternation, the `InStr()` function can test for substring presence.

See Also

[comparison](#), [in](#), [and](#), [select](#), [instr](#), [ucase](#), [lcase](#)

Notes

LIKE predicates push down to SQL backends as native `LIKE` / `NOT LIKE` clauses — except on SQLite, whose `LIKE` ignores case for ASCII letters whatever the column's collation. There a literal pattern is rewritten as the case-sensitive `GLOB`, `%` becoming `*` and `_` becoming `?` (code `LIKE 'AB-%'` is sent as code `GLOB 'AB-*`), and a pattern that is not a literal runs in the engine. SQL Server's `LIKE` reads `[` as the start of a character class, so a literal pattern is sent with each `[` written `[[]`, and a pattern that is not a literal runs in the engine there too. For MongoDB connections the pattern is converted to an anchored regular expression and pushed as a `$regex` / `$not $regex` aggregation filter — the negated form paired with a null check, since `$not` alone also matches a document that has no such field, which is not what a NULL value means here. In both cases a fallback to in-engine evaluation occurs when the operands are not translatable (e.g. a computed expression on the left-hand side).

5. Joins

Combining two relations. The outer joins keep unmatched rows (with NULLs); the semi/anti joins filter the left relation by the existence of a match.

| Join | Unicode / ASCII | What it does |
|------------------|--------------------------|--|
| Natural join | $\bowtie$ / JOIN | Match on shared column names |
| Theta join | $\bowtie$ / >< | Match on an explicit condition |
| Left outer join | $\bowtie$ / >< / LJOIN | Keep all left rows |
| Right outer join | $\bowtie$ / >< / RJOIN | Keep all right rows |
| Full outer join | $\bowtie$ / >< / FJOIN | Keep all rows from both sides |
| Semi join | $\ltimes$ / SEMI | Left rows that <i>have</i> a match |
| Anti join | $\not\bowtie$ / ANTI | Left rows that have <i>no</i> match |
| AS-OF join | ASOF | Match each row to the nearest-in-time row |
| Interval join | IJOIN | Match intervals by an Allen relation (OVERLAPS, DURING, ...) |
| Lateral join | LATERAL | Invoke a TVF per left row with correlated column arguments |

Natural Join ($\bowtie$ / JOIN)

Syntax

```
Relation1  $\bowtie$  Relation2
Relation1 JOIN Relation2
```

```
Users  $\bowtie$  Orders
Users JOIN Orders
```

Description

A natural join stitches two relations together by matching rows that agree on every column they share by name. If Users and Orders both have a `user_id` column, `Users  $\bowtie$  Orders` pairs each user with their orders automatically — no condition needed. The shared column appears once in the result.

It is the simplest join to write when your tables are designed with consistent column names.

Technical Description

$R \bowtie S$ joins on the intersection of their attribute names (equality on all common columns) and projects away the duplicate join columns, keeping one copy, which holds the left input's value. Two inputs with no common column are an analysis error; use $\times$ (CROSS) for a Cartesian product. It is commutative and the planner may choose the build side by cardinality. A natural join of two tables on the same connection pushes down to a SQL `JOIN ... ON` that equates the common columns, provided each common column has the same type on both sides.

Examples

Pair each user with their orders (shared `user_id`):

```
Users ⋈ Orders
```

Enrich line items with product details (shared `product_id`):

```
LineItems ⋈ Products
```

Three-way join chains naturally on shared keys:

```
(Customers ⋈ Orders) ⋈ LineItems
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];
```

```
Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];
```

```
query { Customers ⋈ Orders };
```

| customer_id | name | city | order_id | amount |
|-------------|-------|--------|----------|--------|
| 1 | Alice | London | 100 | 120 |
| 1 | Alice | London | 101 | 80 |
| 2 | Bob | Berlin | 102 | 45 |

(3 rows)

Three rows, because Alice has two orders — a join multiplies, it does not merely look up. Cara and order 103 both vanish: a natural join keeps only rows that match on **every** shared column, here just `customer_id`. Note that `customer_id` appears **once**: the natural join projects the duplicate away.

Limitations

A natural join needs both sides to **declare** their columns: it matches on the shared ones, and those are settled before any row is read. Over a schema-on-read source (JSON, HTTP, MongoDB) there is nothing declared to share, so the join is rejected rather than silently matching nothing. Use a theta join (`L ⋈ L.id = R.id R`) there — it names its columns explicitly and resolves them per row, so it works over documents.

Natural join depends entirely on column names lining up — if the keys are named differently (`user_id` vs `uid`) it will not match them, and if unrelated columns share a name it will match on those too. Use a theta join ($\bowtie$) for explicit control.

Alternatives

Theta join ($\bowtie$) with an explicit condition when names differ or you need a non-equality match. Composition ($\circ$) joins on shared columns and then drops them all.

See Also

[theta-join](#), [left-outer-join](#), [semi-join](#), [composition](#), [rename](#)

Notes

Rename (ρ) is the usual fix when key names don't match or when self-joining.

Theta Join ($\bowtie$ / $><$)

Syntax

```
Relation1  $\bowtie$  <condition> Relation2  
Relation1  $><$  <condition> Relation2
```

```
Users  $\bowtie$  Users.id = Orders.user_id Orders  
Users  $><$  Users.id = Orders.user_id Orders
```

Description

A theta join combines two relations using an explicit condition that you write out — most often "this column equals that column", but it can be any comparison. Use it when the matching columns have different names, or when matching is not a simple equality (e.g. a price within a range). Unlike a natural join it never guesses; you state exactly how the two sides line up.

Technical Description

$R \bowtie_{\theta} S$ returns the rows of the cross product $R \times S$ that satisfy predicate θ . θ may reference qualified columns (`Relation.column`) from both sides and use any comparison/logical operators. It keeps both sides' columns (no implicit projection). The planner picks join algorithm (hash/merge) and build side via the cost model; same-connection equi-joins push down to a SQL `JOIN ... ON`.

Qualified column references above a join

When both sides expose a column of the same name, the joined relation keeps both. A **relation-qualified** reference — `Relation.column` — resolves to the exact column it names, anywhere above the join (in σ , π , γ , τ , λ), not just inside the join condition. Each column remembers which relation it came from (its *provenance*), and a rename re-anchors it: after ρR (...) every column's qualifier becomes `R`.

A qualified reference either resolves to precisely one column or is a **validation error** — a stale qualifier (a relation that is not a source of that column here) is never silently bound to an unrelated first match.

Worked example — a name collision. `devices` and `rooms` both have a `name` column:

| devices | | | rooms | |
|-----------|---------|--------------|---------|---------------|
| device_id | room_id | name | room_id | name |
| 1 | 10 | "Thermostat" | 10 | "Living Room" |
| 2 | 20 | "Lamp" | 20 | "Bedroom" |

Query — the device together with the room it is in:

```
query {
  π devices.name → device, rooms.name → room
    (devices ⋈ devices.room_id = rooms.room_id rooms)
};
```

Output — `devices.name` and `rooms.name` each resolve to their own side:

| device | room |
|--------------|---------------|
| "Thermostat" | "Living Room" |
| "Lamp" | "Bedroom" |

Schema-on-read inputs are the exception. The check relies on a declared heading, which records where each column came from. A document from a JSON, HTTP or MongoDB source records nothing, and over it a dotted name may also be a path into one of its fields (`address.city`). So once a view or a rename has hidden a relation's name, a reference that still uses the hidden name is **not** refused: it is read as a path, finds no such field, and yields NULL.

This is deliberate. Refusing the name would also refuse a genuine path into a document field that happens to share it.

Worked example — a hidden name over a document source. `orders` is a declared CSV file and `products` is a JSON file, and the view `Sales` joins them:

| orders.csv | | products.json |
|------------|----------|--|
| product_id | quantity | [{"product_id": 1, "name": "Novel"},
{"product_id": 2, "name": "Trowel"}] |
| 1 | 2 | |
| 2 | 5 | |

```
source orders from csv("orders.csv") {
  header: true, schema: { product_id: NUMBER, quantity: NUMBER }
};
source products from json("products.json");

Sales := { orders ⋈ orders.product_id = products.product_id products };

query { π orders.quantity → stale, Sales.quantity → quantity, Sales.name → name (Sales) };
```

Output — outside the view its columns answer to `Sales`, so `orders.quantity` names nothing and reads NULL:

| stale | quantity | name |
|-------|----------|--------|
| NULL | 2 | Novel |
| NULL | 5 | Trowel |

Over two declared relations the same `orders.quantity` is an analysis error. Over a document source, use the view's own name: `Sales.quantity`, and `Sales.product_id_r` for the copy of `product_id` the join renamed. The same holds for a rename, `p P (products)`, whose fields answer to `P` only.

Legacy `_r` disambiguation

For backward compatibility the joined schema still renames the right side's colliding column with an `_r` suffix (`name` on the left, `name_r` on the right), so an **unqualified** `name_r` continues to work. Prefer the qualified form `rooms.name` — it names a real column that appears in the source schema, whereas `name_r` is an engine-invented name. The `_r` convention is retained but considered legacy.

Examples

Join when key names differ:

```
Users ⋈ Users.id = Orders.user_id Orders
```

Compound condition — match and also require the order be active:

```
Users ⋈ Users.id = Orders.user_id ∧ Orders.active = true Orders
```

Range / band join — price tiers:

```
Sales ⋈ Sales.amount >= Tiers.min ∧ Sales.amount <= Tiers.max Tiers
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];

Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];

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

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 100 | 1 | 120 |
| 1 | Alice | London | 101 | 1 | 80 |
| 2 | Bob | Berlin | 102 | 2 | 45 |

(3 rows)

Same three rows as the natural join — but the condition is written out, so it works even when the keys are named differently, and the result keeps **both** columns. The right-hand `customer_id` is renamed `customer_id_r` to keep the output schema unambiguous.

Limitations

Joins work over schema-on-read sources (JSON, HTTP, MongoDB), which name their columns per row rather than up front: a qualified reference resolves against the document, a field neither side carries reads as NULL, and a right-hand field whose name the left already uses is suffixed `_r`, as it is under a declared heading. The two forms that emit an **unmatched right** row — `⋈` and `⋈_r` — are the exception: that row has no left row to collide with, so the same field would be named one way when it matched and another when it did not.

A non-equality (range/inequality) theta join cannot use a hash join and may be expensive; the engine handles it but it is not pushed as efficiently.

Concretely, a join whose condition contains no `left.col = right.col` conjunct is planned as a **nested loop**: every left row is compared against every right row, $O(n \times m)$. That is not a silent fallback — `:explain` shows the algorithm (`Join INNER/NESTED_LOOP`), and `--trace` reports it as a `PLAN/JOIN-NESTED-LOOP` event naming the reason. If a query is unexpectedly slow, that pair is the first thing to look at: adding one equality conjunct is what turns it into a hash (or, over sorted inputs, a merge) join.

Alternatives

Natural join (`⋈`) when shared columns already line up by name. Outer joins (`⋈⋈` `⋈_r`) when you need to keep non-matching rows too. AS-OF join (ASOF) for nearest-in-time matching.

See Also

[natural-join](#), [left-outer-join](#), [right-outer-join](#), [full-outer-join](#), [asof-join](#)

Notes

Qualify columns as `Relation.column` inside the condition so the planner can route each reference to the correct side — important when both sides share a column name.

Left Outer Join (`⋈ / |>< / LJOIN`)

Syntax

```
Relation1 ⋈ <condition> Relation2
Relation1 |>< <condition> Relation2
Relation1 LJOIN <condition> Relation2      -- SQL-prior synonym

Customers ⋈ Customers.customer_id = Orders.customer_id Orders
Customers LJOIN Customers.customer_id = Orders.customer_id Orders  -- identical AST
```

Description

A left outer join keeps every row from the left relation, attaching matching rows from the right where they exist and filling the right-hand columns with NULL where they don't. It answers "list all customers, with their orders if they have any" — including customers who have placed no orders at all.

The ASCII form `|><` is a mnemonic: the `|` is on the left, so the left relation contributes all its rows.

Technical Description

$R \bowtie_{\theta} S$ returns $R \bowtie_{\theta} S$ plus, for each left row with no match, one row padded with NULLs in S 's columns. Output keeps both schemas. It is a streaming join (the left side drives). Because unmatched left rows are preserved, a selection on the right-side columns generally cannot be pushed below the join.

Examples

All customers, with order totals where they exist:

```
Customers ⋈ Customers.customer_id = Orders.customer_id Orders
```

Find customers with no orders (outer join then keep the NULLs):

```
σ Orders.order_id = NULL  
(Customers ⋈ Customers.customer_id = Orders.customer_id Orders)
```

Attach optional profile data without dropping profile-less users:

```
Users ⋈ Users.id = Profiles.user_id Profiles
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [  
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];		
  
Orders := [  
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];		
  
query { Customers ⋈ Customers.customer_id = Orders.customer_id Orders };
```

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 100 | 1 | 120 |
| 1 | Alice | London | 101 | 1 | 80 |
| 2 | Bob | Berlin | 102 | 2 | 45 |
| 3 | Cara | Lisbon | NULL | NULL | NULL |

(4 rows)

Cara is kept, padded with NULLs — that is the whole point of the left outer join: every left row survives, matched or not. Order 103 still disappears, since nothing on the right is preserved. Filtering this result with $\sigma \text{ order_id} = \text{NULL}$ gives "customers who have never ordered" — the idiom an ANTI join expresses more directly.

Limitations

Joins work over schema-on-read sources (JSON, HTTP, MongoDB), which name their columns per row rather than up front: a qualified reference resolves against the document, a field neither side carries reads as NULL, and a right-hand field whose name the left already uses is suffixed `_r`, as it is under a declared heading. The two forms that emit an **unmatched right** row — `⋈` and `⋈` — are the exception: that row has no left row to collide with, so the same field would be named one way when it matched and another when it did not.

NULL padding means downstream conditions on right columns must account for NULLs. Pushing filters across an outer join is restricted to preserve its semantics.

Alternatives

Right outer join (`⋈`) keeps the right side instead; full outer (`⋈`) keeps both. ANTI join (`⋈`) directly gives "left rows with no match" without the NULL test.

See Also

[right-outer-join](#), [full-outer-join](#), [theta-join](#), [anti-join](#), [is-null](#)

Notes

The common "rows missing on the right" idiom is `left ⋈ cond right` followed by `σ right.key = NULL`; an ANTI join is the more direct expression.

Right Outer Join (`⋈` / `><|` / `RJOIN`)

Syntax

```
Relation1 ⋈ <condition> Relation2
Relation1 ><| <condition> Relation2
Relation1 RJOIN <condition> Relation2      -- SQL-prior synonym

Orders ⋈ Orders.customer_id = Customers.customer_id Customers
Orders RJOIN Orders.customer_id = Customers.customer_id Customers  -- identical AST
```

Description

A right outer join keeps every row from the right relation, attaching matching rows from the left where they exist and filling the left-hand columns with NULL where they don't. It is the mirror image of a left outer join — use whichever reads more naturally for the relation you want to preserve in full.

The ASCII form `><|` is a mnemonic: the `|` is on the right, so the right relation contributes all its rows.

Technical Description

`R ⋈_θ S` returns `R ⋈_θ S` plus, for each right row with no match, a row padded with NULLs in R's columns. Equivalent to `S ⋈_θ R` with column order from R then S. Output keeps both schemas.

Examples

Keep all customers (on the right), with orders where present:

```
Orders ⋈ Orders.customer_id = Customers.customer_id Customers
```

List every product, with sales figures where they exist:

```
Sales ⋈ Sales.product_id = Products.id Products
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];		

Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];		

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

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 100 | 1 | 120 |
| 1 | Alice | London | 101 | 1 | 80 |
| 2 | Bob | Berlin | 102 | 2 | 45 |
| NULL | NULL | NULL | 103 | 4 | 60 |

(4 rows)

The mirror image: **order 103 is kept** with NULLs on the customer side, and Cara is dropped. This is how orphaned rows are found — an order whose customer no longer exists shows up here as a row with a NULL `name`.

Limitations

Not available over a schema-on-read source (JSON, HTTP, MongoDB). This join emits rows from the right that matched nothing, and such a row has no left row to collide with — so a right field whose name the left also uses would be renamed on a matched row and not on an unmatched one, leaving the column meaning two different things. The other joins have no such ambiguity.

As with any outer join, NULL padding on the unmatched side must be handled by downstream conditions, and filter pushdown is restricted to preserve semantics.

Alternatives

Left outer join ($\bowtie$) — usually you can swap operands and use a left join, which many find easier to reason about. Full outer ($\bowtie$) keeps both sides.

See Also

[left-outer-join](#), [full-outer-join](#), [theta-join](#), [is-null](#)

Notes

A right outer join is exactly a left outer join with the operands swapped; pick the form that puts the "keep all of this one" relation where it reads clearly.

Full Outer Join ($\bowtie$ / $|><|$ / FJOIN)

Syntax

```
Relation1  $\bowtie$  <condition> Relation2
Relation1  $|><|$  <condition> Relation2
Relation1 FJOIN <condition> Relation2      -- SQL-prior synonym

A  $\bowtie$  A.id = B.id B
A FJOIN A.id = B.id B                      -- identical AST
```

Description

A full outer join keeps every row from BOTH relations: matched rows are paired, and any row without a partner on the other side is kept with the other side's columns filled with NULL. It is the join to use when you want a complete picture that loses nothing — for example, reconciling two lists to see what is in each, in both, or in only one.

The ASCII $|><|$ has a $|$ on each side: both relations contribute all rows.

Technical Description

$R \bowtie_{\theta} S = (R \bowtie_{\theta} S) \cup (R \bowtie_{\theta} S)$: the inner matches plus unmatched left rows (NULL-padded right) plus unmatched right rows (NULL-padded left). It is a blocking ([bag]) operator and keeps both schemas.

Examples

Reconcile two account lists — see matches and orphans on both sides:

```
LedgerA  $\bowtie$  LedgerA.ref = LedgerB.ref LedgerB
```

Records only in the left (right side is NULL):

```
 $\sigma$  B.id = NULL (A  $\bowtie$  A.id = B.id B)
```

Records only in the right (left side is NULL):

```
 $\sigma$  A.id = NULL (A  $\bowtie$  A.id = B.id B)
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];

Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];

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

| customer_id | name | city | order_id | customer_id_r | amount |
|-------------|-------|--------|----------|---------------|--------|
| 1 | Alice | London | 100 | 1 | 120 |
| 1 | Alice | London | 101 | 1 | 80 |
| 2 | Bob | Berlin | 102 | 2 | 45 |
| NULL | NULL | NULL | 103 | 4 | 60 |
| 3 | Cara | Lisbon | NULL | NULL | NULL |

(5 rows)

Both unmatched rows survive: Cara with no order, and order 103 with no customer. Five rows — the three matches plus one from each side. This is the join to reach for when reconciling two sources and you need to see what each one has that the other does not.

Limitations

Not available over a schema-on-read source (JSON, HTTP, MongoDB). This join emits rows from the right that matched nothing, and such a row has no left row to collide with — so a right field whose name the left also uses would be renamed on a matched row and not on an unmatched one, leaving the column meaning two different things. The other joins have no such ambiguity.

Full outer join materialises and is the heaviest of the joins. Both unmatched sides produce NULLs, so downstream conditions must be NULL-aware.

Alternatives

Symmetric difference (Δ) gives "rows in exactly one side" for union-compatible relations. Left/right outer when you only need one side preserved.

See Also

[left-outer-join](#), [right-outer-join](#), [symmetric-difference](#), [is-null](#)

Notes

The "in exactly one side" question is often cleaner via Δ (SYMDIFF) when the two relations share the same columns.

Semi Join ($\ltimes$ / SEMI)

Syntax

```
Relation1  $\ltimes$  <condition> Relation2
Relation1 SEMI <condition> Relation2
```

```
Employees  $\ltimes$  Employees.dept = Departments.dept Departments
```

Description

A semi join keeps rows from the left relation that HAVE a match on the right — but it does not bring any of the right relation's columns along. It answers "which customers have placed at least one order?" without duplicating a customer once per order. The result is a filtered version of the left relation, same columns, no fan-out.

Technical Description

$R \ltimes_{\theta} S$ returns the left rows for which at least one matching right row exists, projected back to R's schema, with no duplication of left rows (existential match). It is the relational form of SQL `WHERE EXISTS (...)`. Left-side distinctness and ordering are preserved.

Examples

Customers who have at least one order:

```
Customers  $\ltimes$  Customers.customer_id = Orders.customer_id Orders
```

Products that have ever been reviewed:

```
Products  $\ltimes$  Products.id = Reviews.product_id Reviews
```

Employees in a department that exists in the Departments table:

```
Employees  $\ltimes$  Employees.dept = Departments.dept Departments
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];

Orders := [
order_id	customer_id	amount
```

```
100	1	120
101	1	80
102	2	45
103	4	60
];
```

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

```
customer_id  name  city
-----
           1  Alice London
           2   Bob  Berlin
(2 rows)
```

Two rows, not three — and the schema is **unchanged**. A semi join filters the left relation by existence: Alice appears **once** even though she has two orders, because it asks "is there a match?", not "which matches?". That is the difference from a natural join, and the reason a semi join is the right tool when you only want to filter.

Limitations

A semi join cannot return any column from the right side — if you need right-side data, use a regular (theta/natural) join. Each left row appears at most once.

Alternatives

A theta join then δ achieves a similar set but risks fan-out and needs the dedupe; semi join is cleaner and avoids duplicating left rows. ANTI join ($\triangleright$) is the complement ("no match").

See Also

[anti-join](#), [theta-join](#), [natural-join](#), [forall](#), [distinct](#)

Notes

Prefer $\times$ over δ (π left.* (left $\bowtie$ right)) — it expresses existence directly and avoids materialising duplicates.

Anti Join ($\triangleright$ / ANTI)

Syntax

```
Relation1  $\triangleright$  <condition> Relation2
Relation1 ANTI <condition> Relation2
```

```
Customers  $\triangleright$  Customers.customer_id = Orders.customer_id Orders
```

Description

An anti join keeps rows from the left relation that have NO match on the right. It is the direct way to ask "which customers have never placed an order?" or "which products have never sold?" — the things that are missing. The result has the left relation's columns and only its unmatched rows.

Technical Description

$R \triangleright_{\theta} S$ returns the left rows for which no matching right row exists, projected back to R's schema (the relational `WHERE NOT EXISTS (...)`). Left-side distinctness and ordering are preserved. It is the complement of the semi join.

Examples

Customers who have never ordered:

```
Customers ▷ Customers.customer_id = Orders.customer_id Orders
```

Products with no reviews:

```
Products ▷ Products.id = Reviews.product_id Reviews
```

Active users who have not logged in this month:

```
Users ▷ Users.id = MonthlyLogins.user_id MonthlyLogins
```

Worked Example

The same two relations run through every join on this page, so the results are directly comparable. Two rows do the work: **Cara** has placed no orders, and **order 103** belongs to customer 4, who is not in `Customers`. Which of those two survives is exactly what distinguishes one join from the next.

```
Customers := [
customer_id	name	city
1	Alice	London
2	Bob	Berlin
3	Cara	Lisbon
];
```

```
Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
103	4	60
];
```

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

| customer_id | name | city |
|-------------|------|--------|
| 3 | Cara | Lisbon |

(1 row)

Exactly the complement of the semi join: the one customer who has **never** ordered. Same unchanged schema, no NULL padding, and no need to filter afterwards — compare the left-outer-join route, which produces four rows and then needs `σ order_id = NULL` to get to this same answer.

Limitations

Like the semi join, it returns only left-side columns. NULLs in the join keys follow not-exists semantics — a left row with a NULL key matches nothing and is kept.

Alternatives

A left outer join then `σ right.key = NULL` produces the same set but is more verbose; the anti join states the intent directly. Set difference (`−`) works when the two relations are union-compatible.

See Also

[semi-join](#), [left-outer-join](#), [difference](#), [is-null](#)

Notes

Prefer `▷` over the outer-join-plus-NULL-test idiom for "the ones that are missing" — it is clearer and the planner handles it as a first-class join.

AS-OF Join (ASOF)

Syntax

```
Left ASOF <condition> Right
Left ASOF INNER <condition> Right
Left ASOF <condition> WITHIN <duration> Right
Left ASOF <condition> TIES(FIRST|LAST) Right
Left ASOF INNER <condition> WITHIN <duration> TIES(FIRST|LAST) Right
```

The condition is one or more equality keys joined by ``^`` (partition), plus exactly one ordering inequality (``<`` ``<=`` ``>`` ``>=``) on the match column.

```
Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts Quotes
Trades ASOF INNER Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts Quotes
Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts WITHIN DURATION 'PT1H' Quotes
Trades ASOF Trades.ts >= Quotes.ts TIES(FIRST) Quotes
```

Description

An AS-OF join matches each left row to the single nearest right row in time rather than to an exact match. It answers the classic time-series question "for each trade, what was the most recent quote at-or-before it?" — pairing each event with the latest (or earliest) related record. It is the relational form of kdb's AS-OF / "by" join.

Three optional clauses refine the matching behaviour:

- **INNER** — by default, left rows with no match keep NULLs on the right (left-outer). Add `INNER` immediately after `ASOF` to drop those unmatched probes entirely (inner join semantics).
- **WITHIN duration** — an optional tolerance bound. Even if a qualifying right row is found, it is rejected if its temporal distance from the probe exceeds the bound (e.g. "the most recent quote, but no older than an hour"). The bound is a `DURATION` literal, and the match column must hold a temporal value — see *Technical Description* for what counts as a distance.
- **TIES(FIRST|LAST)** — when multiple right rows share the same nearest match value, `TIES(LAST)` (the default) keeps the last one in input order and `TIES(FIRST)` keeps the

first.

Technical Description

The condition is a flat conjunction ($\wedge$ -only) of equality comparisons (the partition keys — equivalent to kdb's "by") plus exactly one ordering inequality on the match column. The inequality direction selects the matching direction:

| | |
|------------------------------|--|
| <code>L.ts >= R.ts</code> | backward: greatest <code>R.ts ≤ L.ts</code> ("as of" default) |
| <code>L.ts > R.ts</code> | strict backward: <code>R.ts</code> strictly before <code>L.ts</code> |
| <code>L.ts <= R.ts</code> | forward: least <code>R.ts ≥ L.ts</code> |
| <code>L.ts < R.ts</code> | strict forward: <code>R.ts</code> strictly after <code>L.ts</code> |

For each left (probe) row the engine:

1. Finds the right-side partition whose equality keys match the probe.
2. Sorts that partition ascending by match column (stable, so input order within equal match values is preserved).
3. Binary-searches for the nearest qualifying value (greatest at-or-before for backward; least at-or-after for forward).
4. Among tied values, picks the last (default) or first (TIES(FIRST)) in input order.
5. If a tolerance is set, rejects the candidate when the temporal distance between probe and candidate exceeds the bound.

A `WITHIN` bound measures the same distance the language's own subtraction does, so it is defined between two `TIMESTAMPS`, two `DATES` (a whole-day span) and two `DURATIONS`. Anything else — text, numbers, a `TIME` (whose difference wraps) — has no distance, and a `WITHIN` over such a match column is an error at analysis time rather than a bound that quietly does nothing. Text holding ISO-8601 timestamps still *orders* correctly, which is what makes the silent version of this so easy to miss: convert the column first, with `π to_timestamp(ts) → ts (...)`.

1. Emits the concatenated row, or a NULL-padded row (left-outer) / drops the probe (INNER).

Examples

Each trade paired with the prevailing quote (latest at-or-before), per symbol:

```
Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts Quotes
```

Forward match — next scheduled price after each event:

```
Events ASOF Events.sku = Prices.sku ^ Events.at <= Prices.at Prices
```

Strict backward match (exclude exact-time ties):

```
Readings ASOF Readings.sensor = Cal.sensor ^ Readings.t > Cal.t Cal
```

Inner: only emit trades that actually had a quote (drop unmatched):

```
Trades ASOF INNER Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts Quotes
```

With tolerance — quote must be within 30 minutes of the trade:

```
Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts
      WITHIN DURATION 'PT30M' Quotes
```

Prefer first-seen quote when two quotes share the same timestamp:

```
Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts TIES(FIRST) Quotes
```

All three modifiers combined:

```
Trades ASOF INNER Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts
  WITHIN DURATION 'PT1H'
  TIES(FIRST) Quotes
```

Worked Example

Basic AS-OF: trade slippage calculation

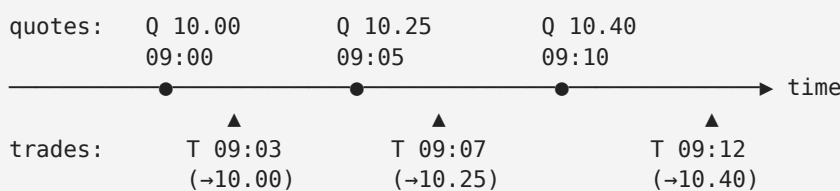
A trading system records executed trades and a separate stream of market quotes. For each trade it needs the *prevailing* quote — the most recent quote for that symbol at or before the trade's timestamp — so it can compute slippage. The timestamps rarely match exactly, which is precisely why an equi-join fails and an AS-OF join is the right tool.

```
Quotes := [
sym	ts	price
ACME	2026-01-02T09:00:00Z	10.00
ACME	2026-01-02T09:05:00Z	10.25
ACME	2026-01-02T09:10:00Z	10.40
];
```

```
Trades := [
sym	ts	qty
ACME	2026-01-02T09:03:00Z	100
ACME	2026-01-02T09:07:00Z	50
ACME	2026-01-02T09:12:00Z	75
];
```

```
Filled := { Trades ASOF Trades.sym = Quotes.sym ^ Trades.ts >= Quotes.ts Quotes };
```

Backward matching (`Trades.ts >= Quotes.ts`) snaps each trade left to the latest quote at or before it:



Each trade keeps the price of the quote immediately to its left:

```
query { Filled };
```

| sym | ts | qty | sym_r | ts_r | price |
|------|----------------------|-----|-------|----------------------|-------|
| ACME | 2026-01-02T09:03:00Z | 100 | ACME | 2026-01-02T09:00:00Z | 10 |
| ACME | 2026-01-02T09:07:00Z | 50 | ACME | 2026-01-02T09:05:00Z | 10.25 |
| ACME | 2026-01-02T09:12:00Z | 75 | ACME | 2026-01-02T09:10:00Z | 10.4 |

(3 rows)

The matched quote is carried through whole, not just its price: Quotes' `sym` and `ts` collide with the left side's, so they arrive as `sym_r` and `ts_r` — `ts_r` is the useful one, since it names *which* quote was snapped to. Project down to the columns you want if the extra pair

is noise.

WITHIN: staleness guard

A quote that is too old is not useful. Adding `WITHIN DURATION 'PT5M'` keeps only trades where the most recent quote was within 5 minutes — stale trades get no match. Because the default behaviour is left-outer, stale trades emit NULLs; combining with `INNER` drops them entirely.

`WITHIN` measures a real temporal distance, so the match column has to be a `TIMESTAMP`. An inline table holds text, so `to_timestamp` converts it first — without that the columns are strings, which still *order* correctly (ISO-8601 sorts chronologically) but have no distance for the tolerance to measure, and the analyzer rejects the query rather than ignoring the bound:

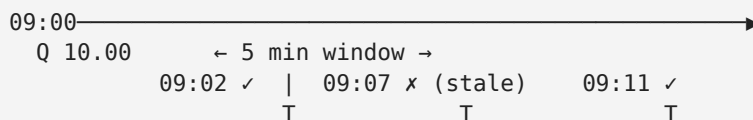
```
StaleFeed := [
sym	ts	price
ACME	2026-01-02T09:00:00Z	10.00	-- stale for trades after 09:05
ACME	2026-01-02T09:08:00Z	10.35
];

Trades2Raw := [
sym	ts	qty
ACME	2026-01-02T09:02:00Z	100	-- 2 min after 09:00 → within 5 min ✓
ACME	2026-01-02T09:07:00Z	50	-- 7 min after 09:00 → stale      x
ACME	2026-01-02T09:11:00Z	75	-- 3 min after 09:08 → within 5 min ✓
];

QuotesStale := { π sym, to_timestamp(ts) → ts, price (StaleFeed) };
Trades2      := { π sym, to_timestamp(ts) → ts, qty (Trades2Raw) };

FreshOnly := {
  Trades2 ASOF INNER
    Trades2.sym = QuotesStale.sym ∧ Trades2.ts >= QuotesStale.ts
  WITHIN DURATION 'PT5M'
  QuotesStale
};
```

The staleness check rejects the 09:07 trade because the nearest prior quote (09:00) is 7 minutes away — over the 5-minute limit. `INNER` then drops that row instead of NULL-padding it.



Result — only the two in-window trades:

```
query { FreshOnly };
```

| sym | ts | qty | sym_r | ts_r | price |
|------|----------------------|-----|-------|----------------------|-------|
| ACME | 2026-01-02T09:02:00Z | 100 | ACME | 2026-01-02T09:00:00Z | 10 |
| ACME | 2026-01-02T09:11:00Z | 75 | ACME | 2026-01-02T09:08:00Z | 10.35 |

(2 rows)

TIES: deterministic first-seen

An exchange sometimes pushes two quotes in the same millisecond. The default `TIES(LAST)` matches the last-received quote; `TIES(FIRST)` instead matches the first-received, preserving the arrival order visible in the source data.

```
Quotes3 := [
sym	ts	price	feed
ACME	2026-01-02T09:05:00Z	10.25	NASDAQ
ACME	2026-01-02T09:05:00Z	10.26	NYSE	-- same instant, second arrival
];

Trades3 := [
sym	ts	qty
ACME	2026-01-02T09:07:00Z	100
];

-- Default: last-in-input = NYSE quote
ByLast := { Trades3 ASOF Trades3.sym = Quotes3.sym ^ Trades3.ts >= Quotes3.ts Quotes3 };

-- First-in-input = NASDAQ quote
ByFirst := { Trades3 ASOF Trades3.sym = Quotes3.sym ^ Trades3.ts >= Quotes3.ts TIES(FIRST) Quotes3 };
```

The default takes the last of the tied quotes — the NYSE one:

```
query { ByLast };
```

| sym | ts | qty | sym_r | ts_r | price | feed |
|------|----------------------|-----|-------|----------------------|-------|------|
| ACME | 2026-01-02T09:07:00Z | 100 | ACME | 2026-01-02T09:05:00Z | 10.26 | NYSE |

(1 row)

`TIES(FIRST)` takes the NASDAQ one instead — same trade, same matched instant, different fill:

```
query { ByFirst };
```

| sym | ts | qty | sym_r | ts_r | price | feed |
|------|----------------------|-----|-------|----------------------|-------|--------|
| ACME | 2026-01-02T09:07:00Z | 100 | ACME | 2026-01-02T09:05:00Z | 10.25 | NASDAQ |

(1 row)

SQL pushdown

On a **PostgreSQL**, **DuckDB** or **Db2** connection an AS-OF over two bare connection tables folds into a single `LEFT JOIN LATERAL (SELECT ... ORDER BY <match> DESC|ASC LIMIT 1) ON TRUE` pushed scan, so the database does the nearest-match lookup. The ordering inequality fixes the `ORDER BY` direction (backward → `DESC`, forward → `ASC`); the inner variant uses a plain `JOIN LATERAL ... ON TRUE`. On a **SQL Server** connection the same lookup is spelled `OUTER APPLY (SELECT TOP 1 ... ORDER BY <match> DESC|ASC)`, and the inner variant `CROSS APPLY`. `LATERAL` is absent from H2 (the `GENERIC` dialect) and version-gated in MySQL (8.0.14+), and SQLite has none, so those fall back to the in-engine executor. A `WITHIN` tolerance also falls back (the bound leans on temporal arithmetic with no portable SQL form), as does a join spanning two connections. MongoDB pushdown is not implemented.

When pushed, ties on the match column are resolved by the database's `LIMIT 1` row ordering rather than the in-engine `TIES(FIRST|LAST)` rule; the two agree unless the tied

right rows differ in a selected column (a documented divergence).

Limitations

Exactly one ordering inequality is required; zero inequalities (equality only) or two or more is a validation error. The tolerance clause only applies when both probe and candidate are temporal values (`TIMESTAMP`, `DATE`, `TIME`) whose subtraction yields a `DURATION`; for other types the bound is not enforced at plan time but the Relix executor will reject a non-temporal distance at runtime.

Alternatives

A theta join with a range condition plus `TOP 1 PER` can emulate nearest-match but is far more verbose and has no tolerance clause. For interval overlap matching (two-sided temporal ranges) see `IJOIN`.

See Also

[interval-join](#), [theta-join](#), [top](#), [sort](#), [duration-literal](#), [timestamp-literal](#)

Notes

ASOF pairs naturally with the first-class temporal types (`TIMESTAMP/DATE/TIME`). The match column is typically a `TIMESTAMP` but any ordered type works. For a two-sided interval overlap query use `IJOIN` instead.

Interval Join (IJOIN)

Syntax

```
...
Left IJOIN <relation> (lStart, lEnd, rStart, rEnd) Right
...

'<relation>' is one of the thirteen Allen interval-relation keywords –
`PRECEDES`/`PRECEDED_BY`, `MEETS`/`MET_BY`, `OVERLAPS`/`OVERLAPPED_BY`,
`STARTS`/`STARTED_BY`, `DURING`/`CONTAINS`, `FINISHES`/`FINISHED_BY`, and
`EQUALS` – plus the convenience superset `INTERSECTS` (any shared time).
...

Stays      IJOIN OVERLAPS  (Stays.checkin, Stays.checkout, Bookings.from, Bookings.to) Bookings
Meetings   IJOIN INTERSECTS(Meetings.start, Meetings.end, Blocks.start, Blocks.end)   Blocks
Events     IJOIN PRECEDES  (Events.start, Events.end, Deadlines.start, Deadlines.end) Deadlines
...
```

Description

An interval join matches two relations where each row carries a temporal (or numeric) interval `[start, end]` and the two intervals satisfy a named geometric relationship. Rather than writing a multi-condition theta join, you state the relationship by name — one of Allen's thirteen interval relations (`OVERLAPS`, `DURING`, `CONTAINS`, `MEETS`, `PRECEDES`, `STARTS`, `FINISHES`, `EQUALS`, and their converses) or the symmetric convenience `INTERSECTS` — and Relix applies the right predicate for you.

The thirteen base relations form seven converse pairs — for every relation `X` there is an `X_BY` such that $\ell \text{ X } r$ iff $r \text{ X_BY } \ell$ (with `EQUALS` its own converse). Naming the converse explicitly lets you keep the two inputs in their natural order instead of swapping them.

IJOIN is an *inner* join: only pairs of rows whose intervals satisfy the named relationship are emitted. There is no NULL-padding.

Technical Description

Intervals are half-open: `[start, end)` — start is inclusive, end is exclusive. This matches the standard convention for date/time ranges (e.g. a hotel stay `[check-in, check-out)` or a calendar event `[start, end)`).

The fourteen supported relations (Allen's thirteen + INTERSECTS) and their endpoint predicates are:

| | | |
|---------------|--|---|
| INTERSECTS | $l.s < r.e \wedge r.s < l.e$ | any part of L overlaps any part of R (symmetric) |
| PRECEDES | $l.e < r.s$ | L ends strictly before R begins (gap between them) |
| PRECEDED_BY | $l.s > r.e$ | L begins strictly after R ends (converse of PRECEDES) |
| MEETS | $l.e = r.s$ | L ends exactly where R begins (adjacent, no gap) |
| MET_BY | $l.s = r.e$ | L begins exactly where R ends (converse of MEETS) |
| OVERLAPS | $l.s < r.s \wedge l.e > r.s$
$\wedge l.e < r.e$ | L starts before R and ends inside R
(strictly — L neither contains nor equals R) |
| OVERLAPPED_BY | $r.s < l.s \wedge r.e > l.s$
$\wedge r.e < l.e$ | R starts before L and ends inside L (converse) |
| STARTS | $l.s = r.s \wedge l.e < r.e$ | same start, L ends first |
| STARTED_BY | $l.s = r.s \wedge r.e < l.e$ | same start, R ends first (converse of STARTS) |
| DURING | $r.s < l.s \wedge l.e < r.e$ | L is strictly within R (both endpoints interior) |
| CONTAINS | $l.s < r.s \wedge r.e < l.e$ | L strictly contains R (converse of DURING) |
| FINISHES | $l.e = r.e \wedge r.s < l.s$ | same end, L starts later |
| FINISHED_BY | $l.e = r.e \wedge l.s < r.s$ | same end, R starts later (converse of FINISHES) |
| EQUALS | $l.s = r.s \wedge l.e = r.e$ | identical intervals (self-converse) |

Semantics. All thirteen base relations use strict endpoint bounds (Allen 1983), so they are mutually exclusive and jointly exhaustive — any pair of intervals stands in **exactly one** of them. Shared-boundary cases are named by their own relation (STARTS, FINISHES, EQUALS) rather than folded into DURING/CONTAINS: an interval with the *same* start as another is STARTS (or STARTED_BY/EQUALS), never DURING. The only relation that overlaps the others is the convenience INTERSECTS, which subsumes every relation that shares any time.

Column names may be qualified with a table prefix (`Table.column`). The four endpoint columns are resolved to indices at plan time; the validator checks they exist in their respective schemas and share a compatible temporal type. The operation runs in-engine and never pushes down to SQL or MongoDB backends.

Execution is endpoint-based rather than a quadratic nested loop. For the eleven relations whose intervals must overlap or touch, IJOIN sorts the interval endpoints of both sides and runs a **plane sweep**: as the sweep crosses each endpoint it keeps a set of currently-active intervals per side and pairs each opening interval only with the active intervals on the other side — exactly the candidate pairs whose intervals intersect — then confirms the precise Allen relation. The two disjoint relations (PRECEDES/PRECEDED_BY) instead sort the right side once and binary-search the boundary, emitting the matching suffix/prefix per left row. Both strategies cost $O((n+m) \cdot \log(n+m))$ plus the number of result rows, versus $O(n \cdot m)$ for a full nested loop.

Allen Relation Reference

Allen's thirteen relations cover every distinct way two intervals can relate on a timeline, exhaustively and (in their strict form) exclusively. The diagrams below use $L = [====]$ for the left interval and $R = [====]$ for the right. Each relation is paired with its converse — flipping L and R turns X into X_BY .

INTERSECTS – any shared time (symmetric, convenience superset)

$L: [=====]$

$R: [=====]$

Rule: $L.start < R.end$ AND $R.start < L.end$

PRECEDES / PRECEDED_BY – L entirely before R (gap between them)

$L: [====]$

$R: [=====]$

Rule: $L.end < R.start$ (PRECEDED_BY: $L.start > R.end$)

MEETS / MET_BY – L ends exactly where R begins (adjacent, no overlap, no gap)

$L: [=====]$

$R: [=====]$

Rule: $L.end = R.start$ (MET_BY: $L.start = R.end$)

OVERLAPS / OVERLAPPED_BY – L starts before R and ends inside R

$L: [=====]$

$R: [=====]$

Rule: $L.start < R.start$ AND $L.end > R.start$ AND $L.end < R.end$

Note: differs from INTERSECTS because L does not extend beyond R 's end.

OVERLAPPED_BY is the mirror case (R starts first).

STARTS / STARTED_BY – both begin together, L ends first

$L: [====]$

$R: [=====]$

Rule: $L.start = R.start$ AND $L.end < R.end$

(STARTED_BY: $R.end < L.end$)

DURING – L strictly inside R

$L: [====]$

$R: [=====]$

Rule: $R.start < L.start$

AND $L.end < R.end$

CONTAINS – L strictly contains R

$L: [=====]$

$R: [====]$

Rule: $L.start < R.start$

AND $R.end < L.end$

FINISHES / FINISHED_BY – both end together, L starts later

$L: [=====]$

$R: [=====]$

Rule: $L.end = R.end$ AND $R.start < L.start$

(FINISHED_BY: $L.start < R.start$)

EQUALS – identical intervals (self-converse)

$L: [=====]$

$R: [=====]$

Rule: $L.start = R.start$ AND $L.end = R.end$

INTERSECTS is the symmetric superset: any pair satisfying OVERLAPS, OVERLAPPED_BY, DURING, CONTAINS, STARTS, STARTED_BY, FINISHES, FINISHED_BY, or EQUALS also satisfies INTERSECTS (these all share time). MEETS/MET_BY (touching, half-open $\rightarrow$ no shared instant) and PRECEDES/PRECEDED_BY (disjoint) are the relations *not* captured by INTERSECTS.

Examples

Find every hotel stay that overlaps an existing booking:

```
Stays IJOIN OVERLAPS (Stays.checkin, Stays.checkout, Bookings.from, Bookings.to) Bookings
```

Find shifts that a maintenance window runs into from earlier (the window started before the shift and ends partway through it — the mirror of OVERLAPS):

```
Shifts IJOIN OVERLAPPED_BY (Shifts.start, Shifts.end, Windows.start, Windows.end) Windows
```

Any meeting that intersects (any overlap with) a maintenance window:

```
Meetings IJOIN INTERSECTS (Meetings.start, Meetings.end, Windows.start, Windows.end) Windows
```

Tasks that fall entirely within a project's active phase:

```
Tasks IJOIN DURING (Tasks.start, Tasks.end, Phases.start, Phases.end) Phases
```

Find project phases that fully contain a task (inverse — phase CONTAINS task):

```
Phases IJOIN CONTAINS (Phases.start, Phases.end, Tasks.start, Tasks.end) Tasks
```

Events that finish exactly when the next event begins (scheduling chains):

```
Events IJOIN MEETS (Events.start, Events.end, Next.start, Next.end) Events Next
```

Sessions that start the instant the preceding break ends (back-to-back, no gap — the converse of MEETS):

```
Sessions IJOIN MET_BY (Sessions.start, Sessions.end, Breaks.start, Breaks.end) Breaks
```

Short launch promos that kick off a longer campaign (same start, promo ends first):

```
Promos IJOIN STARTS (Promos.start, Promos.end, Campaigns.start, Campaigns.end) Campaigns
```

Campaigns viewed from the other side — those that open with a launch promo (same start, campaign runs longer — the converse of STARTS):

```
Campaigns IJOIN STARTED_BY (Campaigns.start, Campaigns.end, Promos.start, Promos.end) Promos
```

Steps that are entirely before their dependencies start:

```
Steps IJOIN PRECEDES (Steps.start, Steps.end, Deps.start, Deps.end) Deps
```

Shifts that share their end-of-day with the store's closing window (FINISHES — same end, shift starts later than the window opens):

```
Shifts IJOIN FINISHES (Shifts.start, Shifts.end, Hours.open, Hours.close) Hours
```

Billing periods that end with a final grace window (same end, period starts earlier — the converse of FINISHES):

```
Periods IJOIN FINISHED_BY (Periods.start, Periods.end, Grace.start, Grace.end) Grace
```

Two reservations covering exactly the same slot (EQUALS):

```
A IJOIN EQUALS (A.start, A.end, B.start, B.end) B
```

A late job whose interval begins only after an earlier job has ended (PRECEDED_BY — the converse of PRECEDES, keeping Late on the left):

```
Late IJOIN PRECEDED_BY (Late.start, Late.end, Early.start, Early.end) Early
```

Worked Example

Hotel availability check

A booking platform stores two kinds of intervals: **Requests** (potential bookings not yet confirmed) and **Confirmed** (accepted reservations). Before accepting a new request, the platform needs to know which existing confirmed reservations it would conflict with.

```
Confirmed := [
booking_id	room	checkin	checkout
B1	101	2026-03-01T14:00:00Z	2026-03-05T11:00:00Z
B2	101	2026-03-08T14:00:00Z	2026-03-10T11:00:00Z
B3	102	2026-03-03T14:00:00Z	2026-03-07T11:00:00Z
];

Requests := [
request_id	room	from_date	to_date
R1	101	2026-03-04T14:00:00Z	2026-03-09T11:00:00Z	-- spans B1 end and B2 start
R2	102	2026-03-01T14:00:00Z	2026-03-03T14:00:00Z	-- ends exactly when B3 starts
R3	102	2026-03-10T12:00:00Z	2026-03-12T11:00:00Z	-- after all B3 (no conflict)
];
```

The intervals look like this on a timeline (per room):

```
Room 101:
  Confirmed:   B1 [====Mar01-Mar05====]      B2 [=Mar08-Mar10=]
  Request:    R1 [====Mar04-Mar09====]
               ← conflicts with both B1 and B2

Room 102:
  Confirmed:   B3 [====Mar03-Mar07====]
  Request:    R2 [Mar01-Mar03]      R3 [Mar10-Mar12]
               ← meets B3 (no overlap)      ← after B3 (no conflict)
```

Step 1 — find overlapping conflicts using **INTERSECTS**:

```
Conflicts := {
  σ room = room_r (
    Requests IJOIN INTERSECTS
      (Requests.from_date, Requests.to_date, Confirmed.checkin, Confirmed.checkout)
    Confirmed
  )
};
```

INTERSECTS requires $R.from < C.checkout$ AND $C.checkin < R.to$ — any shared time is a conflict. Because **IJOIN** matches on the interval endpoints only, the $\sigma \text{ room} = \text{room}_r$ keeps just the *same-room* pairs (the join disambiguates **Confirmed**'s room to **room_r**). Note that **R2** ends at 2026-03-03T14:00:00Z exactly when **B3** (its room) begins — half-open intervals make this a **MEETS** relationship, not **INTERSECTS** — so **R2** does **not** appear:

```
query { Conflicts };
```

| request_id | room | from_date | to_date | booking_id | room_r | checkin | checkout |
|------------|------|----------------------|----------------------|------------|--------|----------------------|----------------------|
| R1 | 101 | 2026-03-04T14:00:00Z | 2026-03-09T11:00:00Z | B1 | 101 | 2026-03-01T14:00:00Z | 2026-03-05T11:00:00Z |
| R1 | 101 | 2026-03-04T14:00:00Z | 2026-03-09T11:00:00Z | B2 | 101 | 2026-03-08T14:00:00Z | 2026-03-10T11:00:00Z |

(2 rows)

R1 conflicts with both **B1** and **B2**. **R2** and **R3** are conflict-free.

Step 2 — report only the conflicting requests (without the booking detail):

```
BlockedRequests := {
  π request_id, room (Conflicts)
};
```

| request_id | room |
|------------|------|
| R1 | 101 |

Step 3 — to also detect the MEETS case ("back-to-back with no gap, but no actual overlap"), use a separate query:

```
BackToBack := {
  σ room = room_r (
    Requests IJOIN MEETS
      (Requests.from_date, Requests.to_date, Confirmed.checkin, Confirmed.checkout)
    Confirmed
  )
};
```

As in Step 1, the `σ room = room_r` keeps only same-room adjacency — R2 ends exactly when B3 begins, both in room 102:

| request_id | room | from_date | to_date | booking_id | room_r | checkin | checkout |
|------------|------|----------------------|----------------------|------------|--------|----------------------|----------------------|
| R2 | 102 | 2026-03-01T14:00:00Z | 2026-03-03T14:00:00Z | B3 | 102 | 2026-03-03T14:00:00Z | 2026-03-07T11:00:00Z |

Whether a back-to-back booking is a conflict depends on business rules (e.g. does housekeeping need a gap?). By keeping the two concepts separate in the query — `INTERSECTS` for overlap, `MEETS` for adjacency — the business rule lives in the RA script, not hidden inside a comparison.

Schedule gap detection

A maintenance team schedules maintenance windows. A gap between two consecutive windows means a service period — the `PRECEDES` relationship identifies windows followed by another window with an unscheduled gap in between.

```
Windows := [
id	start	end
W1	2026-04-01T02:00:00Z	2026-04-01T04:00:00Z
W2	2026-04-01T04:00:00Z	2026-04-01T06:00:00Z	-- meets W1 exactly
W3	2026-04-01T08:00:00Z	2026-04-01T09:00:00Z	-- gap after W2
];

-- Windows that PRECEDE another (a gap exists between them)
Gaps := {
  π W.id, W.end, Later.start
  (σ W.id != Later.id
   (ρ W (Windows) IJOIN PRECEDES
     (W.start, W.end, Later.start, Later.end)
   ρ Later (Windows)))
};
```

| id | end | start |
|----|----------------------|----------------------|
| W2 | 2026-04-01T06:00:00Z | 2026-04-01T08:00:00Z |

W2 precedes W3 (gap from 06:00 to 08:00). W1 does not precede W2 — it *meets* it exactly (no gap). The gap size can be computed from the result: `Later.start - W.end` gives `DURATION 'PT2H'`.

Limitations

IJOIN is an inner join — there is no outer form or NULL-padding. Endpoints must be valid comparable values (temporal or numeric); NULL endpoints cause that row to produce no matches (silently dropped). All thirteen Allen relations and their converses are first-class keywords (plus the `INTERSECTS` convenience), so you never need to swap Left and Right or hand-write a theta join to express a relation — including `EQUALS`, `OVERLAPPED_BY`, `MET_BY`, and `PRECEDED_BY`.

SQL pushdown. Over two bare connection tables on one JDBC connection, an IJOIN folds into a single `JOIN ... ON <endpoint predicate>` pushed scan whose `ON` clause is the chosen relation's endpoint comparison — e.g. `OVERLAPS` → `l.s < r.s AND l.e > r.s AND l.e < r.e`. The comparisons mirror the in-engine test exactly, and SQL's three-valued logic excludes any NULL-endpoint row (matching the engine's NULL-endpoint drop). It works on **every** SQL dialect (only `</=>` are used — no SQL:2011 period-predicate syntax is required); a join spanning two connections, or a non-database source, falls back to the in-engine executor. MongoDB pushdown is not implemented.

When not pushed, IJOIN runs in-engine (endpoint plane sweep / sorted band — see Technical Description). The sweep is output-sensitive: a relation that genuinely matches most pairs still produces a large result, so on equality-partitioned data add a selection (σ) on the partition keys (e.g. same room, same sensor) before the IJOIN to keep each sweep small. A merge variant skips the sort when both inputs already arrive ordered, reusing the engine's ordering machinery.

Alternatives

A theta join ($\bowtie$ / $><$) can express any single interval predicate as two range comparisons, but requires writing out the predicates by hand. IJOIN names the relationship declaratively. For *nearest-in-time* point matching (one side is a point, not a range) use `ASOF` instead.

See Also

[asof-join](#), [theta-join](#), [duration-literal](#), [timestamp-literal](#), [select](#)

Notes

Half-open interval semantics `[start, end)` are the standard convention for calendar intervals — a stay from 2026-03-01 to 2026-03-05 does **not** include the checkout date itself. The `MEETS` predicate (`L.end = R.start`) therefore represents a clean handoff with no overlap and no gap — a common boundary condition in scheduling. Use `INTERSECTS` (not `OVERLAPS`) when you want any shared time regardless of which side starts first.

Lateral Join (LATERAL)

Syntax

```
left LATERAL fn(arg, ...)  
  
Customers LATERAL ordersFor(customer_id)
```

Description

A LATERAL join invokes a table-valued function (TVF) once per row of the left relation, with arguments that may reference columns from that row. The output is the left row concatenated with every row the TVF returns for it. A left row that matches no TVF rows simply produces no output rows (inner-join semantics).

This is the correlated cousin of a plain TVF call: where `ordersFor(2)` always returns the same orders, `Customers LATERAL ordersFor(customer_id)` personalises the call for each customer.

Technical Description

For each row r in left:

1. Evaluate the argument expressions against r (they may reference any left column).
2. Call `fn(evaluated args)` to obtain a relation `TVF_r`.
3. Emit one output row for each row t in `TVF_r`: $r \parallel t$ (left columns then TVF columns).

The output schema is the left schema concatenated with the TVF's return schema. If the TVF returns an empty relation for a given left row, that left row contributes no output rows: LATERAL has inner-join semantics, and there is no LATERAL LEFT JOIN form.

Arguments must be scalar expressions; argument expressions referencing columns from the left relation are fully supported. Arguments may also be constants or arithmetic expressions combining constants and left-side columns.

Every scalar type may be an argument: NUMBER, STRING, BOOLEAN, and the four temporal types DATE, TIME, TIMESTAMP and DURATION. The value is carried into the function body as a literal of its own type, so a DATE argument arrives as a DATE rather than as its text, and a boolean argument may be received by a `BOOLEAN` parameter.

LATERAL shares the same precedence as $\bowtie$ (natural join) and is left-associative:

```
A  $\bowtie$  B LATERAL f(x)  $\rightarrow$  (A  $\bowtie$  B) LATERAL f(x)
```

The TVF is defined with `def name(p: T, ...): RELATION := { ... }`; (see `def ... : RELATION`). The planner inlines the TVF body per-row at execution time.

Examples

Golden path: each customer's orders

```
def ordersFor(cid: NUMBER): RELATION := { σ customer_id = cid (Orders) };

Orders := [| order_id | customer_id | amount |
           | 1       | 2       | 100   |
           | 2       | 3       | 50    |
           | 3       | 2       | 75    |];

Customers := [| customer_id | name |
              | 2         | Alice |
              | 3         | Bob  |];

query { Customers LATERAL ordersFor(customer_id) };
```

Result — Alice is paired with her two orders; Bob with his one:

| customer_id | name | order_id | amount |
|-------------|-------|----------|--------|
| 2 | Alice | 1 | 100 |
| 2 | Alice | 3 | 75 |
| 3 | Bob | 2 | 50 |

Filtering the LATERAL output

```
query { σ amount > 60 (Customers LATERAL ordersFor(customer_id)) };
```

Result — only Alice's large orders survive:

| customer_id | name | order_id | amount |
|-------------|-------|----------|--------|
| 2 | Alice | 1 | 100 |
| 2 | Alice | 3 | 75 |

Projecting over a LATERAL join

```
query { π name, amount (Customers LATERAL ordersFor(customer_id)) };
```

Two-parameter correlated TVF

```
def topOrders(cid: NUMBER, lim: NUMBER): RELATION :=
  { σ customer_id = cid ∧ amount >= lim (Orders) };

Customers := [| customer_id | name | min_amount |
              | 2         | Alice | 80         |
              | 3         | Bob  | 40         |];

query { Customers LATERAL topOrders(customer_id, min_amount) };
```

Each customer's `min_amount` column is passed as the second argument, giving each customer a personalised threshold.

Left-associativity with other joins

```
def f(v: NUMBER) : RELATION := { σ amount = v (Orders) };

query { (A ⋈ B) LATERAL f(x) }; -- explicit grouping
query { A ⋈ B LATERAL f(x) }; -- same: left-associative at join precedence
```

What the engine does with a repeated or constant argument

A LATERAL join is expensive in a way its syntax hides: the TVF body is *re-planned* for each left row, not just re-executed. Two things cut that down, and neither changes an answer.

An uncorrelated LATERAL is decorrelated. When no argument reads a left column *and* the function body is reproducible, every left row would instantiate the same relation, so the optimizer rewrites the join to a plain product over a single TVF call (LATERAL-001):

```
Customers LATERAL ordersFor(2) → Customers × ordersFor(2)
```

One plan, one execution. The bigger gain is that a $\times$ is a shape every other rule understands: a selection above it can become a theta join, filters push into the left input, and the planner costs it normally — none of which it can do with a LATERAL, whose function body is invisible to tree traversal. See [optimizer](#) for the rule's exact conditions.

A repeated argument tuple is memoized. When the arguments genuinely are correlated, the engine caches the planned body — and, for a reproducible body within a bounded budget, the rows it produced — under the argument tuple. A lateral over a low-cardinality foreign key hits the same values on row after row, and each repeat then costs a map lookup instead of a planner run. The cache is per query execution and bounded in both entries and retained rows, so a correlation whose values are nearly all distinct falls back to the plain per-row path rather than growing.

Neither applies to a function whose body is not reproducible. A body reading `Rand()` or `NOW()`, drawing an unseeded `SAMPLE`, or calling a view or nested TVF that does, is evaluated once per left row exactly as written — the decorrelation rule declines to fire, and the memo keeps only the *plan*, which is reusable because planning evaluates nothing. Add a `SEED` to a sampling body if you want it treated as reproducible.

Worked Example

The manual's usual customers and orders, joined through a function instead of a predicate:

```
Customers := [
customer_id	name
1	Alice
2	Bob
3	Cara
];

Orders := [
order_id	customer_id	amount
100	1	120
101	1	80
102	2	45
];

def ordersFor(cid: NUMBER) : RELATION := { σ customer_id = cid (Orders) };
```

```
query { Customers LATERAL ordersFor(customer_id) };
```

| customer_id | name | order_id | customer_id_r | amount |
|-------------|-------|----------|---------------|--------|
| 1 | Alice | 100 | 1 | 120 |
| 1 | Alice | 101 | 1 | 80 |
| 2 | Bob | 102 | 2 | 45 |

(3 rows)

`ordersFor` was called once per customer, each time with that customer's own `customer_id` — which is the whole difference from `ordersFor(1)`, a call that would return the same rows no matter which row it was joined to. Cara produced no rows and therefore contributes none: a lateral join is an inner join.

The case a plain join cannot do. Because the function body is evaluated per outer row, a `λ` inside it means "per outer row", not "overall" — so the top-order-per-customer question needs no window and no self-join:

```
def topOrderFor(cid: NUMBER) : RELATION := {
  λ 1 (τ amount DESC (σ customer_id = cid (Orders)))
};
```

```
query { Customers LATERAL topOrderFor(customer_id) };
```

| customer_id | name | order_id | customer_id_r | amount |
|-------------|-------|----------|---------------|--------|
| 1 | Alice | 100 | 1 | 120 |
| 2 | Bob | 102 | 2 | 45 |

(2 rows)

One row per customer who has any order, each the largest. Written as an ordinary join with `λ 1` above it, the limit would apply to the joined result and return a single row overall.

Limitations

- **Inner-join semantics only:** left rows with no TVF matches are dropped. There is no `LATERAL LEFT JOIN` form.
- **Arguments must be scalar and non-NULL:** an argument evaluating to NULL, or to a nested struct/array value, produces a runtime error naming the value's kind. There is no literal that denotes NULL, and substituting one that did not mean NULL would run the function on a value the outer row never held.
- **No pushdown:** LATERAL joins are always executed in-engine. The TVF body may itself reference pushed-down sources; only the LATERAL operator stays in-engine.
- **No recursion:** a TVF may not call itself (directly or transitively) via LATERAL.
- **A base relation is assumed stable for the duration of a query.** The rewrites above reuse one evaluation of the function body, which re-reads whatever tables it scans. That is the same assumption `×`, every hash join, `÷` and the set operations already make when they materialise one side and replay it.

Alternatives

A plain TVF call (`Relation ⋈ myFn(constant)`) works when the argument does not depend on the outer row. Use LATERAL when the per-row column reference is the point.

See Also

[def ... : RELATION](#), [natural-join](#), [theta-join](#), [semi-join](#)

6. Set operations

Treating relations as sets of rows.

| Operation | Unicode / ASCII | What it does |
|----------------------|---------------------------|--|
| Union | ⋈ / UNION | Rows in either (deduplicated) |
| Union all | ⋈ / UALL | Rows in either (keeps duplicates) |
| Outer-union | ⋈ / OUNION | Merge different-schema relations (align common columns, NULL-pad the rest) |
| Intersection | ∩ / INTER / INTERSECT | Rows in both |
| Difference | − / DIFF / MINUS / EXCEPT | Rows in the first but not the second |
| Symmetric difference | Δ / SYMDIFF | Rows in exactly one |
| Cartesian product | × / CROSS | Every combination of rows |
| Division | ÷ / DIV | Which X relate to <i>all</i> of the Y |
| Composition | ∘ / COMPOSE | Chain on a shared column and drop it |

Union (⋈ / UNION)

Syntax

```
Relation1 ⋈ Relation2  
Relation1 UNION Relation2
```

```
CurrentCustomers ⋈ ArchivedCustomers
```

Description

Union stacks two relations on top of each other into one, and removes duplicate rows. Both relations must have the same shape (same columns). Use it to combine records from two sources into a single list — for example active and archived customers — without listing anyone twice.

Technical Description

$R \cup S$ requires union-compatible schemas (same arity, compatible column types) and returns the set of rows appearing in either, de-duplicated (set semantics). It is a blocking ([set]) operator.

Compatibility is **positional**: the two sides must agree on the number of columns and on the type of each position, and need not agree on what those columns are called. Rows are matched by position for the same reason, so a row of L and a row of R holding the same values in the same order are one row. The result carries the left side's column names.

Examples

Combine two customer lists into one de-duplicated list:

```
CurrentCustomers u ArchivedCustomers
```

All cities mentioned in either shipping or billing addresses:

```
( $\pi$  city (ShippingAddresses)) u ( $\pi$  city (BillingAddresses))
```

Worked Example

Two mailing lists with one row in common — `bo@example.com`, who appears in both, identically. Every set operation on this page is the same question about that overlap, asked differently.

```
Subscribers := [  
email	plan
ada@example.com	pro
bo@example.com	free
cy@example.com	pro
];
```

```
TrialUsers := [  
email	plan
bo@example.com	free
dee@example.com	trial
];
```

```
query { Subscribers u TrialUsers };
```

| email | plan |
|-----------------|-------|
| ada@example.com | pro |
| bo@example.com | free |
| cy@example.com | pro |
| dee@example.com | trial |

(4 rows)

Four rows from three plus two: `bo@example.com` is in both lists and appears **once**. Union is set semantics — it deduplicates. When you want to keep the duplicate (to count contributions, say), use `u` / `UALL` instead, which returns five rows here.

Limitations

Both sides must be union-compatible: the same number of columns, and compatible types position by position. Column *names* need not match, and do not affect which rows are duplicates. Duplicates are removed; to keep them use `UNION ALL` (`u`).

Alternatives

Union all (`u`) keeps duplicates and is cheaper. Intersection (`n`) keeps only rows in both; difference (`-`) keeps rows in the first but not the second.

See Also

[union-all](#), [intersection](#), [difference](#), [symmetric-difference](#), [distinct](#)

Notes

If you know the inputs are already disjoint, $\cup$ (UNION ALL) avoids the de-duplication cost.

Union All ($\cup$ / UALL)

Syntax

```
Relation1  $\cup$  Relation2
Relation1 UALL Relation2

JanuarySales  $\cup$  FebruarySales
```

Description

Union all stacks two relations into one and KEEPS every row, including duplicates. It is the right choice when you are concatenating data that is already distinct (e.g. monthly partitions) and you do not want to pay for de-duplication, or when duplicate counts actually matter.

Technical Description

$R \cup S$ requires union-compatible schemas and returns the bag (multiset) union — multiplicities add. It is a blocking ([bag]) operator and is cheaper than $\cup$ because it skips de-duplication. Selections replicate into both branches (SEL-006).

Examples

Concatenate monthly partitions you know are disjoint:

```
JanuarySales  $\cup$  FebruarySales  $\cup$  MarchSales
```

Pool measurements from several sensors, keeping every reading:

```
SensorA  $\cup$  SensorB  $\cup$  SensorC
```

Worked Example

Two mailing lists with one row in common — `bo@example.com`, who appears in both, identically. Every set operation on this page is the same question about that overlap, asked differently.

```
Subscribers := [
email	plan
ada@example.com	pro
bo@example.com	free
cy@example.com	pro
];

TrialUsers := [
email	plan
bo@example.com	free
dee@example.com	trial
];

query { Subscribers  $\cup$  TrialUsers };
```

| email | plan |
|-----------------|-------|
| ada@example.com | pro |
| bo@example.com | free |
| cy@example.com | pro |
| bo@example.com | free |
| dee@example.com | trial |

(5 rows)

Five rows — `bo@example.com` appears **twice**, once from each side. That is the whole difference from `u`, which would return four. Union-all never compares rows, so it is also the cheaper of the two: no buffering, no dedup pass.

Limitations

Keeps duplicates by design — if you need a de-duplicated result use `UNION (u)` or follow with `δ`.

Alternatives

Union (`u`) when you want duplicates removed.

See Also

[union](#), [distinct](#), [difference](#)

Notes

A filter applied to a union-all is pushed into both branches by the optimizer, so `σ p (A ∪ B)` becomes `σ p (A) ∪ σ p (B)`.

Outer-union / heterogeneous merge (`⋈` / `OUNION`)

Syntax

```
Relation1 ⋈ Relation2
Relation1 OUNION Relation2

People ⋈ Places
```

Description

Outer-union merges two relations that need **not** have the same columns. Columns that appear in both inputs are lined up by name; a column that appears in only one input is filled with `NULL` for the rows that came from the other input. Duplicate result rows are removed. Use it to stack records from differently-shaped sources into one table — exactly the situation a plain `UNION` rejects.

Technical Description

$R \sqcup S$ produces the **column-union** of the two schemas: every column of R , followed by every column of S whose name is not already present (matched case-insensitively). A column present in both inputs whose types disagree is widened to `ANY`. Each input row is re-projected onto this merged schema — columns the input does not carry are `NULL`-padded — and the combined rows are de-duplicated (set semantics). It is a blocking (`[set]`) operator and is never pushed down to a backend (there is no portable heterogeneous-merge SQL form).

Unlike $\cup/\cup/n/-$, an outer-union imposes **no** union-compatibility requirement: the inputs may differ in both width and column names. If either input has an open (schema-on-read) schema, the result is open too.

Examples

Two sources describing overlapping-but-different facts, stacked into one table (worked through with real output below):

```
People  $\sqcup$  Places
```

Combine event streams from systems that log different attributes:

```
WebEvents  $\sqcup$  MobileEvents  $\sqcup$  KioskEvents
```

(Outer-union is left-associative and shares the precedence tier of $\cup/\cup/-$, so a chain of $\sqcup$ merges all of them.)

Worked Example

Two relations that a plain $\cup$ would reject: they share only `id`.

```
People := [  
id	name
1	Alice
2	Bob
];
```

```
Places := [  
id	city
2	Berlin
3	Lisbon
];
```

```
query { People  $\sqcup$  Places };
```

| id | name | city |
|----|-------|--------|
| 1 | Alice | NULL |
| 2 | Bob | NULL |
| 2 | NULL | Berlin |
| 3 | NULL | Lisbon |

(4 rows)

The merged heading is `(id, name, city)` — every column of the left input, then every column of the right that is not already there. No row is *combined*: `id 2` appears twice, once from each side, because an outer union stacks rows rather than matching them. That is the

difference from a join, and the reason each row is NULL in exactly the columns its own input does not have.

Limitations

Set semantics only — duplicate rows are removed (there is no "outer-union all" variant). Never pushed down to a source; the merge runs in-engine. A type conflict on a shared column resolves to **ANY** rather than raising an error.

Alternatives

Union (U) when the inputs are already union-compatible — it is cheaper and checks the schemas match. Natural join (⋈) when you want to *combine* matching rows side-by-side rather than stack them.

See Also

[union](#), [union-all](#), [intersection](#), [difference](#), [distinct](#)

Notes

Because the output schema is computed from both inputs' static schemas, an outer-union stays in the closed-schema core whenever its inputs do — the dynamic (schema-on-read) machinery only engages if an input is itself open.

Intersection (n / INTER / INTERSECT)

Syntax

```
Relation1 n Relation2
Relation1 INTER Relation2
Relation1 INTERSECT Relation2      -- SQL-prior synonym

PremiumCustomers n ActiveCustomers
```

Description

Intersection keeps only the rows that appear in BOTH relations. It answers "what do these two lists have in common" — for example customers who are both premium and active. Both relations must have the same number of columns, with compatible types position by position; what the columns are called need not match.

Technical Description

$R \cap S$ requires union-compatible schemas and returns rows present in both (set semantics). It is a blocking ([set]) operator. Equivalent to $R - (R - S)$.

Compatibility and row matching are both **positional**: two rows holding the same values in the same order are the same row, whatever each side calls its columns. The result carries the left side's names.

Examples

Customers who are both premium and active:

```
PremiumCustomers n ActiveCustomers
```

Skills required by a role that a candidate also has:

```
(π skill (RoleRequirements)) n (π skill (CandidateSkills))
```

Worked Example

Two mailing lists with one row in common — `bo@example.com`, who appears in both, identically. Every set operation on this page is the same question about that overlap, asked differently.

```
Subscribers := [
email	plan
ada@example.com	pro
bo@example.com	free
cy@example.com	pro
];

TrialUsers := [
email	plan
bo@example.com	free
dee@example.com	trial
];

query { Subscribers n TrialUsers };
```

| email | plan |
|----------------|------|
| bo@example.com | free |

(1 row)

Only `bo@example.com` — the one row present in both. Note that intersection compares **whole rows**, not just the key: had Bo's `plan` differed between the two lists, the row would not match and the result would be empty. To intersect on a key alone, project to that key first, or use a SEMI join.

Limitations

Both sides must be union-compatible. Compares whole rows; project to the relevant columns first if you want to intersect on a subset.

Alternatives

A semi join ($\bowtie$) gives "left rows that have a match" while keeping the left's extra columns. Union ($\cup$) and difference ($-$) are the other set combinators.

See Also

[union](#), [difference](#), [semi-join](#)

Notes

For "common values of one key" project both sides to that key before intersecting.

Set Difference (– / DIFF / MINUS / EXCEPT)

Syntax

```
Relation1 – Relation2
Relation1 DIFF Relation2
Relation1 MINUS Relation2      -- SQL-prior synonym (Oracle-style)
Relation1 EXCEPT Relation2   -- SQL-prior synonym (Postgres-style)

AllCustomers – CustomersWhoOrdered
```

Description

Difference keeps the rows that are in the first relation but NOT in the second. It is the "what's left after removing these" operator — for example all customers minus those who ordered gives customers who never ordered. Both relations must have the same number of columns, with compatible types position by position; what the columns are called need not match.

Note the symbol is the true minus sign – (U+2212), not the ASCII hyphen; use the keyword DIFF (or the SQL-prior MINUS / EXCEPT) if that is easier to type. The MINUS keyword is set difference; the arithmetic `-` operator inside expressions is unaffected.

Technical Description

Compatibility and row matching are both **positional**: a row of R is removed when S holds the same values in the same order, whatever each side calls its columns.

$R - S$ requires union-compatible schemas and returns the rows of R that do not appear in S (set semantics). It is a blocking ([set]) operator. In recursion it is non-monotone, so it is forbidden in the recursive step of a FIX.

Examples

Customers who never ordered (set-style):

```
AllCustomers – CustomersWhoOrdered
```

Products in the catalogue but not in stock:

```
( $\pi$  sku (Catalogue)) – ( $\pi$  sku (InStock))
```

Which expected files are missing from the upload:

```
ExpectedFiles – UploadedFiles
```

Worked Example

Two mailing lists with one row in common — `bo@example.com`, who appears in both, identically. Every set operation on this page is the same question about that overlap, asked differently.

```
Subscribers := [
email	plan
ada@example.com	pro
bo@example.com	free
cy@example.com	pro
];
```

```

TrialUsers := [
email	plan
bo@example.com	free
dee@example.com	trial
];

```

```

query { Subscribers - TrialUsers };

```

```

email          plan
-----
ada@example.com pro
cy@example.com  pro
(2 rows)

```

The subscribers who are not also trial users. Bo is removed; Dee is **not** added — difference only ever removes rows from the left, never contributes from the right, so the result is a subset of Subscribers. Reverse the operands and you get a different answer (dee@example.com alone).

Limitations

Both sides must be union-compatible. To "subtract" based on a key while keeping extra columns from the left, an ANTI join ($\triangleright$) is usually the better tool.

Alternatives

Anti join ($\triangleright$) when matching is by key and you want to keep the left's full columns. Symmetric difference (Δ) for "in one or the other but not both".

See Also

[anti-join](#), [union](#), [intersection](#), [symmetric-difference](#)

Notes

Because difference is non-monotone it cannot appear in a recursive FIX step — use monotone operators there.

Symmetric Difference (Δ / SYMDIFF)

Syntax

```

Relation1  $\Delta$  Relation2
Relation1 SYMDIFF Relation2

```

```

LedgerA  $\Delta$  LedgerB

```

Description

Symmetric difference keeps the rows that are in exactly one of the two relations — in the first or the second, but not in both. It is the natural "what's different between these two lists" operator, ideal for reconciling two snapshots and surfacing only the discrepancies.

Technical Description

$R \Delta S = (R - S) \cup (S - R)$, defined over union-compatible relations. The planner desugars it into difference and union over set-semantics inputs (materialises SET). It does not push down.

Examples

Reconcile two account ledgers — show only the entries that differ:

```
LedgerA Δ LedgerB
```

Rows that changed between yesterday's and today's snapshot:

```
Yesterday Δ Today
```

Worked Example

Two mailing lists with one row in common — `bo@example.com`, who appears in both, identically. Every set operation on this page is the same question about that overlap, asked differently.

```
Subscribers := [
email	plan
ada@example.com	pro
bo@example.com	free
cy@example.com	pro
];

TrialUsers := [
email	plan
bo@example.com	free
dee@example.com	trial
];

query { Subscribers Δ TrialUsers };
```

| email | plan |
|-----------------|-------|
| ada@example.com | pro |
| cy@example.com | pro |
| dee@example.com | trial |

(3 rows)

Everyone who is in exactly one of the two lists — the union minus the intersection. Unlike `+`, this **is** symmetric: swapping the operands gives the same three rows. It is the "what disagrees between these two sources?" operator, which is why it is useful for reconciliation.

Limitations

Both sides must be union-compatible. Like difference, it is non-monotone and cannot appear in a recursive FIX step. It materialises rather than streams.

Alternatives

A full outer join (`⋈`) with NULL tests gives a similar reconciliation when you also want to see which side each row came from. Difference (`-`) for one-directional "in A not B".

See Also

[difference](#), [union](#), [full-outer-join](#)

Notes

Cleaner than a full-outer-join-plus-NULL-tests when the two relations already share the same columns and you only care about the discrepancies.

Cartesian Product (× / CROSS)

Syntax

```
Relation1 × Relation2
Relation1 CROSS Relation2

Colours × Sizes
```

Description

A cross product pairs every row of the left relation with every row of the right — all possible combinations. With 4 colours and 3 sizes you get 12 colour/size pairs. It is how you generate combinations from scratch, and it is the raw material a join is built from (a join is a cross product followed by a filter).

Technical Description

$R \times S$ returns every (r, s) pair, with the combined schema of both inputs. Output size is $|R| \cdot |S|$, so it grows fast. A selection applied on top of a product over both sides is rewritten by the optimizer into a theta join (JOIN-001).

Examples

All combinations of test parameters:

```
Browsers × OperatingSystems × Versions
```

Build a colour/size matrix:

```
Colours × Sizes
```

Cross then filter (the optimizer turns this into a join):

```
σ Employees.dept = Departments.dept (Employees × Departments)
```

Worked Example

The Cartesian product pairs every row on the left with every row on the right. Its most common honest use is generating combinations — here, every size/colour variant of a product.

```
Sizes := [
size
S
M
];

Colours := [
| colour |
```

```
|-----|
| red   |
| blue  |
|;

```

```
query { Sizes × Colours };
```

```
size  colour
----  -
S     red
S     blue
M     red
M     blue
(4 rows)
```

$2 \times 2 = 4$ rows, and that multiplication is the thing to keep in mind: two thousand-row relations produce a million rows. A cross product followed by a σ that compares the two sides is just a theta join written the long way — and the optimizer will usually turn it into one — but writing the join directly is clearer and never risks materialising the full product.

Limitations

The output can be enormous — guard it with a selection (which the optimizer folds into a join) or by keeping inputs small. A blocking step downstream over a large product can be costly.

Alternatives

If you immediately filter on a matching condition, write a theta join ($\bowtie$) directly. For combinatorial test coverage use COVER to keep a small representative subset.

See Also

[theta-join](#), [natural-join](#), [cover](#)

Notes

Cross products feed the COVER operator's "all combinations" idiom, e.g. `COVER 2 (Type × Format × Size)` for all-pairs test generation.

Relational Division ($\div$ / DIV)

Syntax

```
Relation1 ÷ Relation2
Relation1 DIV Relation2

Enrolments ÷ RequiredCourses
```

Description

Division answers "which X are related to ALL of the Y?" — the textbook example being "which students are enrolled in every required course?" or "which suppliers can supply all the parts we need?". You divide a relation of pairs (student, course) by a relation of the required courses, and you get back the students that cover the entire set.

Technical Description

For $R(A,B) \div S(B)$, division returns the values of A that are paired with every value of B present in S . Formally the largest set Q such that $Q \times S \subseteq R$. It is a first-class blocking ([bag]) operator (not re-expressed via $\forall$). The divisor's columns must be a subset of the dividend's.

Examples

Students enrolled in every required course:

```
Enrolments ÷ RequiredCourses
-- Enrolments(student, course) ÷ RequiredCourses(course) → (student)
```

Suppliers that can supply all needed parts:

```
Supplies ÷ NeededParts
```

Users who have every required permission:

```
UserPermissions ÷ RequiredPermissions
```

Worked Example

A university wants the students who have enrolled in every required course. The dividend is the (student, course) enrolment relation; the divisor is the set of required courses.

```
Enrolments := [
student	course
Ann	Math
Ann	Physics
Ann	Art
Ben	Math
Ben	Art
Cara	Math
Cara	Physics
];
```

```
Required := [
course
Math
Physics
];
```

```
Qualified := { Enrolments ÷ Required };
```

It helps to see it as a coverage matrix — division keeps the rows of the left margin whose required cells (Math AND Physics) are all ticked:

| | Math | Physics | has ALL required? |
|------|------|---------|-------------------|
| Ann | ✓ | ✓ | ✓ → kept |
| Ben | ✓ | ✗ | ✗ (no Physics) |
| Cara | ✓ | ✓ | ✓ → kept |

The result keeps only the *quotient* column (student — the dividend columns not in the divisor), and Ann's extra Art enrolment is irrelevant; division asks "covers all of the divisor", not "matches it exactly":

```
query { Qualified };
```

```
student
```

```
Ann  
Cara  
(2 rows)
```

The same shape answers "suppliers who can supply every needed part" (`Supplies ÷ NeededParts`) or "users who hold every required permission" (`UserPermissions ÷ RequiredPermissions`).

Limitations

The result heading is the left relation's columns minus the right's, so both must be declared. A schema-on-read input (JSON, HTTP, MongoDB) is rejected; project it into a declared heading first.

The divisor relation's columns must be a subset of the dividend's columns. The result keeps only the "quotient" columns (those in the dividend but not the divisor). It materialises rather than streams.

Alternatives

$\forall$ (FORALL) expresses related "every row satisfies" group tests; division is the specialised set-algebra form of "relates to all of these values".

See Also

[forall](#), [semi-join](#), [group](#)

Notes

A classic use is requirement-coverage checks: divide a "has" relation by a "needs" relation to get the entities that fully cover the requirement.

Composition (◦ / COMPOSE)

Syntax

```
Relation1 ◦ Relation2  
Relation1 COMPOSE Relation2
```

```
Manages ◦ p M2(employee, report) (Manages)
```

Description

Composition joins two relations on the columns they share and then DROPS those shared columns, keeping only the outer ends. The classic use is chaining relationships: if `Manages(manager, employee)` says who manages whom, composing it with itself links each manager to the people their reports manage — a "manager → grand-report" relation, with the middle person removed.

Technical Description

$R \circ S$ natural-joins R and S on their shared columns and projects them away, keeping the non-shared columns of both. The planner desugars it to π (non-shared) ($R \bowtie S$). Schema inference errors if the inputs share no column or share every column. It does not push down.

Examples

Two-hop management chain (manager to their reports' reports):

```
Manages ◦ ρ M2(employee, report) (Manages)
```

Friends-of-friends from a friendship edge relation:

```
Friends ◦ ρ F2(friend, other) (Friends)
```

Compose a "lives in city" with "city in country" to get "person in country":

```
LivesIn ◦ CityCountry
```

Worked Example

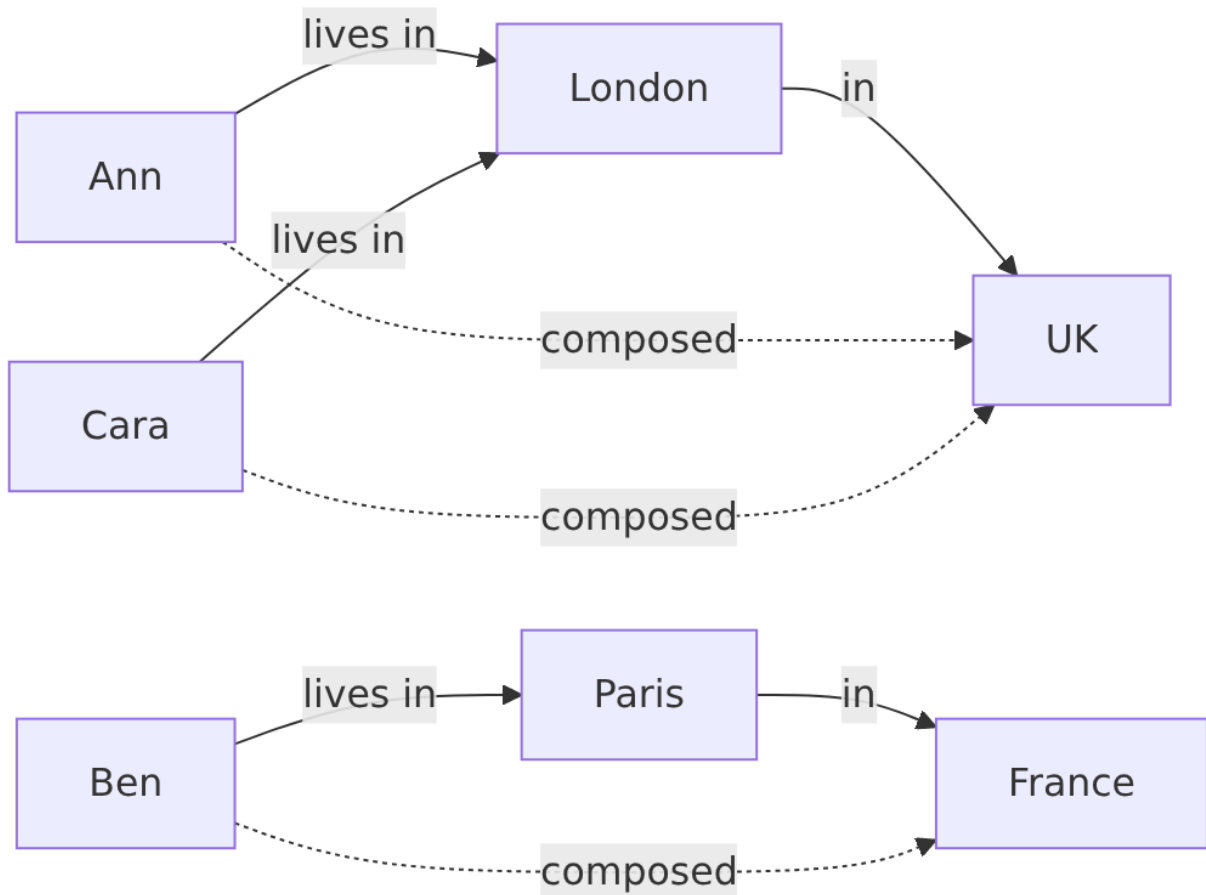
A directory knows which city each person lives in, and a separate table maps cities to countries. Composition chains the two into a direct "person → country" relation, dropping the intermediate city.

```
LivesIn := [  
person	city
Ann	London
Ben	Paris
Cara	London
];
```

```
CityCountry := [  
city	country
London	UK
Paris	France
];
```

```
PersonCountry := { LivesIn ◦ CityCountry };
```

The shared column `city` is the hinge: it lines the two relations up, then gets removed, leaving only the outer ends:



The result keeps (person, country) — city is gone:

```
query { PersonCountry };
```

| person | country |
|--------|---------|
| Ann | UK |
| Ben | France |
| Cara | UK |

(3 rows)

If you compose a relation with *itself* — $\text{Manages} \circ \rho \text{ M2}(\text{employee}, \text{report})$ (Manages) over $\text{Manages}(\text{manager}, \text{employee})$ — you get the two-hop "manager $\rightarrow$ grand-report" relation. That is a single hop of composition; iterate it to all depths and you have transitive closure (CLOSURE / FIX).

Limitations

Composition drops the shared columns, so its result heading depends on knowing which they are. A schema-on-read input (JSON, HTTP, MongoDB) declares none, and the operator is rejected there; project the relation into a declared heading first.

The two relations must share at least one column but not all of them. For full transitive reachability (any number of hops) use CLOSURE / FIX rather than a single composition.

Alternatives

A natural join ($\bowtie$) keeps the shared columns; closure (CLOSURE) repeats the relationship to all depths.

See Also

[natural-join](#), [closure](#), [fix](#), [project](#)

Notes

A single composition is one hop; transitive closure is composition iterated to a fixpoint.

7. Grouping & aggregation

Collapse many rows into summary rows with `γ` (GROUP) and aggregate functions.

| Page | What it does |
|---------------------|---|
| Group / Aggregation | <code>γ / GROUP [BY]</code> — group rows and aggregate |
| SUM | Total of a numeric expression |
| AVG | Mean of a numeric expression |
| COUNT | Rows (<code>COUNT(*)</code>) or non-NULL values (<code>COUNT(expr)</code>) |
| MIN | Smallest value |
| MAX | Largest value |
| COLLECT | Gather a group's values into an array (<code>NEST</code>) |
| ARGMAX | A value from the row with the maximum |
| ARGMIN | A value from the row with the minimum |
| Window / Rolling | <code>ROLLING ... OVER ... ROWS</code> — sliding / cumulative aggregate added as a column (no row collapse) |
| Window / Ranking | <code>WINDOW ROW_NUMBER/RANK/DENSE_RANK/PERCENT_RANK/NTILE</code> — per-partition rank column added to every row |
| Window / Offset | <code>WINDOW LAG/LEAD/FIRST_VALUE/LAST_VALUE</code> — adjacent / boundary row value added as a column (no row collapse) |

Group / Aggregation (`γ / GROUP [BY]`)

Syntax

```
γ <grouping-key> [→ alias] [, ...], <agg>(expr) → alias [, ...] (Relation)
GROUP <grouping-key> [-> alias] [, ...], <agg>(expr) -> alias [, ...] (Relation)
GROUP BY ...          -- SQL-prior synonym; BY is an optional noise word

γ dept, SUM(salary) → total (Employees)
GROUP BY dept, SUM(salary) -> total (Employees)  -- same AST as the γ form above
γ YEAR(order_date) → yr, SUM(amount) → revenue (Orders)  -- derived grouping key
γ SUM(amount) → grand_total (Orders)          -- no grouping = whole-relation total
```

Description

Group is how you summarise data: collapse many rows into one row per group and compute totals, averages, counts, minimums, maximums and more. It answers questions like "total sales per region" or "how many orders did each customer place". List the keys to group by first, then the aggregate calculations.

A grouping key can be a bare column or a full expression — e.g. `YEAR(order_date) → yr` or `price * qty → line` — with an optional `→ alias` naming the output column (a bare column keeps its own name). This mirrors projection: you can group by a computed value, not only a stored column. Write the keys as a plain comma-separated list, without braces: `{region}`

is an expression too (a struct), so braces change what you group by instead of raising an error. The worked example below shows the difference.

If you give no grouping keys, the whole relation collapses to a single summary row.

Technical Description

γ partitions R by the grouping-key tuple and applies each aggregate function to its group. Both grouping keys and aggregate arguments are full operand expressions (e.g. group by `DATE_TRUNC('day', ts)`, aggregate `SUM(price*qty)`), not just column names; each grouping key takes an optional $\rightarrow$ alias (else the column name for a bare column, or an expression-derived name). Supported aggregates: SUM, AVG, COUNT, MIN, MAX, COLLECT (NEST $\rightarrow$ array), ARGMAX, ARGMIN. Output schema = grouping-key columns + one column per aggregate (named by its alias, else `op_arg`). γ is a blocking/materialising operator ([bag]) but becomes a single streaming pass when its input is already ordered by the grouping keys. SUM/AVG/COUNT/MIN/MAX over bare-column grouping keys push down to SQL GROUP BY (a derived grouping key aggregates in-engine).

Examples

Total salary cost per department:

```
 $\gamma$  dept, SUM(salary)  $\rightarrow$  total (Employees)
```

Several aggregates at once — headcount and average pay per department:

```
 $\gamma$  dept, COUNT(emp_id)  $\rightarrow$  headcount, AVG(salary)  $\rightarrow$  avg_pay (Employees)
```

Grand total over everything (no grouping key):

```
 $\gamma$  SUM(amount)  $\rightarrow$  grand_total (Orders)
```

Aggregate an expression, not just a column — revenue per region:

```
 $\gamma$  region, SUM(price * qty)  $\rightarrow$  revenue (Sales)
```

Group by a derived key — annual revenue, bucketing by the order year:

```
 $\gamma$  YEAR(order_date)  $\rightarrow$  yr, SUM(amount)  $\rightarrow$  revenue (Orders)
```

Gather each customer's order ids into an array (NEST):

```
 $\gamma$  customer_id, COLLECT(order_id)  $\rightarrow$  order_ids (Orders)
```

Row-with-the-maximum — the biggest order id per customer:

```
 $\gamma$  customer_id, ARGMAX(amount, order_id)  $\rightarrow$  top_order (Orders)
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, SUM(amount) → total, COUNT(*) → deals (Sales) };
```

| region | total | deals |
|--------|-------|-------|
| east | 200 | 2 |
| west | 250 | 2 |

(2 rows)

Four rows collapse into one per distinct `region`. The output schema is exactly the grouping keys plus the aggregates — the input's other columns (`rep`, `bonus`) are **gone**, because there is no single value for them within a group. If you need one, say which: `ARGMAX(amount, rep)` picks the `rep` from the group's biggest deal, and `COLLECT(rep)` keeps them all.

Do not put braces around the grouping keys. The keys are a plain comma-separated list. Braces are legal there, but they do not group: `{region}` **builds a struct**, so the braced form groups by a one-field struct and produces a different heading. It raises no error:

```
query { γ {region}, SUM(amount) → total (Sales) };
```

| group | total |
|----------------|-------|
| {region: east} | 200 |
| {region: west} | 250 |

(2 rows)

The groups are the same, but there is no `region` column any more. There is one struct column, called `group` because an expression key with no alias is named that way. A later `σ region = 'east'` over this result fails analysis because the column does not exist. `γ {region, rep}` is the same mistake with two keys: it groups by the pair, but returns a single struct column where you wanted two. Write `γ region, rep, ...` instead.

Limitations

`COLLECT`, `ARGMAX` and `ARGMIN` do not push down to SQL (no portable equivalent) and run in the engine. A group with all-NULL aggregate inputs follows SQL-style NULL handling per aggregate.

At least one aggregate is required. Grouping keys on their own do not form a `γ`:

```
query { γ region (Sales) };
```

To list the distinct values of a column, project it and remove duplicates with `δ (π region (Sales))`.

Alternatives

To keep the actual rows that achieve a maximum per group (not just one value), use `TOP k ... PER`. To test a condition across a whole group, use `∀ (FORALL)`.

See Also

[sum](#), [avg](#), [count](#), [min](#), [max](#), [collect](#), [argmax](#), [argmin](#), [top](#), [forall](#), [sort](#)

Notes

When the planner can prove the input is already sorted on the grouping keys it chooses a streaming aggregate (no hash table, no full buffering) — visible in `--explain as Aggregate streaming`.

SUM (aggregate)

Syntax

```
γ <grouping-cols>, SUM(<expr>) → <alias> (Relation)

γ region, SUM(amount) → revenue (Sales)
γ SUM(amount) → grand_total (Orders)      -- whole-relation total
```

Description

SUM adds up the values in each group. It is the workhorse for totals — total sales per region, total hours per project, grand totals across everything. Use it inside a GROUP (γ); the column it produces is named with the → arrow.

Technical Description

SUM is an aggregate function used only inside γ (AggregationNode). Its argument is a full operand expression, not just a column name, so SUM(price * qty) works. It reduces a group's multiset to one number. With no grouping key it produces a single grand total. Pushes down to SQL SUM(...).

Examples

Revenue per region:

```
γ region, SUM(amount) → revenue (Sales)
```

Grand total over the whole table:

```
γ SUM(amount) → grand_total (Orders)
```

Sum of a computed expression — revenue from price × quantity:

```
γ region, SUM(price * qty) → revenue (Sales)
```

Multiple totals at once:

```
γ dept, SUM(salary) → payroll, SUM(salary) → bonuses (Employees)
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, SUM(amount) → total (Sales) };
```

| region | total |
|--------|-------|
| east | 200 |
| west | 250 |

(2 rows)

`SUM` over `bonus` instead would give 10 and 25 — the NULLs are **skipped**, not treated as zero. That distinction matters when the sum feeds a division: summing a column that is half NULL and dividing by `COUNT(*)` gives a different answer from `AVG`, which divides by the count of non-NULL values.

Limitations

Only meaningful on numeric values. NULL inputs are skipped per standard aggregate NULL handling, and a group with no non-NULL value sums to **NULL, not 0** — so an empty total stays distinguishable from a real total of zero. To fold that back to zero, project over the γ result (a scalar function cannot wrap an aggregate in-place):

```
Totals := {  $\gamma$  region, SUM(amount)  $\rightarrow$  revenue (Sales) };
{  $\pi$  region, Nz(revenue, 0)  $\rightarrow$  revenue (Totals) }
```

Must appear inside a γ .

Alternatives

AVG for the mean, COUNT for how many.

See Also

[group](#), [avg](#), [count](#), [min](#), [max](#)

Notes

The aggregate argument is an expression, so you rarely need a separate projection to compute the thing you are summing.

AVG (aggregate)

Syntax

```
 $\gamma$  <grouping-cols>, AVG(<expr>)  $\rightarrow$  <alias> (Relation)
 $\gamma$  dept, AVG(salary)  $\rightarrow$  avg_salary (Employees)
```

Description

AVG computes the average (mean) of the values in each group — average salary per department, average order value per customer, average score per class. Use it inside a GROUP (γ).

Technical Description

AVG is an aggregate function used only inside γ . Its argument is a full operand expression. It returns the arithmetic mean of the non-NULL values in each group. With no grouping key it averages the whole relation. It is **computed by the engine** rather than pushed to the database: an average divides, and a relix average is exact decimal to ten places rounded half-up, which no backend's `AVG` carries. `SUM` and `COUNT` do push, and dividing them yourself puts the rounding under your control.

Examples

Average salary per department:

```
γ dept, AVG(salary) → avg_salary (Employees)
```

Overall average order value:

```
γ AVG(amount) → avg_order (Orders)
```

Average of a computed expression:

```
γ store, AVG(price * qty) → avg_line_value (LineItems)
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, AVG(amount) → mean (Sales) };
```

```
  region  mean
  -----
east      100
west      125
(2 rows)
```

Over bonus the whole-relation average is 17.5, not 8.75: AVG skips NULLs on both sides of the division — it is $35 / 2$, not $35 / 4$. If you want the NULLs counted as zeros, say so explicitly with `AVG(Nz(bonus, 0))`.

Limitations

Numeric only; NULLs are excluded from the average. Must appear inside a γ . An empty group has no average.

Alternatives

$SUM \div COUNT$ computed yourself if you need control over NULL handling; AVG is the direct form.

See Also

[group](#), [sum](#), [count](#), [min](#), [max](#)

Notes

Combine with τ and λ for "top N by average" reports, e.g. `λ 10 (τ avg_salary DESC (γ dept, AVG(salary) → avg_salary (Employees)))`.

COUNT (aggregate)

Syntax

```
γ <grouping-cols>, COUNT(<expr>) → <alias> (Relation)

γ customer_id, COUNT(order_id) → num_orders (Orders)
```

Description

COUNT tells you how many rows are in each group — how many orders each customer placed, how many employees per department, how many events per day. Use it inside a GROUP (γ).

Technical Description

COUNT is an aggregate function used only inside γ, and comes in two forms:

- COUNT(*) counts **rows** — every row in the group, whatever it contains.
- COUNT(expr) counts **non-NULL values** of the expression. A group whose values are all NULL counts 0.

That is SQL's distinction exactly. COUNT(*) is a pure synonym for COUNT(1) — a literal is never NULL, so counting it counts every row — and the two parse to the identical tree, so :tree renders either as COUNT(1).

With no grouping key, COUNT runs over the whole relation. Both forms push down to SQL (COUNT(*) as COUNT(1), which every backend treats identically).

Examples

Orders per customer:

```
γ customer_id, COUNT(order_id) → num_orders (Orders)
```

Total row count of a table:

```
γ COUNT(*) → total_rows (Users)
```

Rows versus values — how many orders each customer placed, and how many of those recorded a shipping date:

```
γ customer_id, COUNT(*) → orders, COUNT(shipped_at) → shipped (Orders)
```

Headcount and payroll together:

```
γ dept, COUNT(emp_id) → headcount, SUM(salary) → payroll (Employees)
```

Count within a filtered subset:

```
γ region, COUNT(customer_id) → active_count (σ active = true (Customers))
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
```

```
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, COUNT(*) → deals, COUNT(bonus) → bonused (Sales) };
```

| region | deals | bonused |
|--------|-------|---------|
| east | 2 | 1 |
| west | 2 | 1 |

(2 rows)

The two columns differ, and that is the entire point: `COUNT(*)` counts **rows** (2 per region), `COUNT(bonus)` counts **non-NULL values** (1 per region, since one rep in each has no bonus). This is SQL's distinction exactly.

So `COUNT(col)` is a row count only when `col` is never NULL. When you mean "how many rows", write `COUNT(*)`.

Limitations

Must appear inside a `γ`. `COUNT(col)` counts non-NULL values, so it is a row count only when `col` is never NULL — use `COUNT(*)` when you mean "how many rows". `*` is special only as the sole argument to `COUNT`, and `COUNT(a * b)` is still multiplication. Any other aggregate needs an expression:

```
query { γ SUM(*) → total (Orders) };
```

Alternatives

δ then a whole-relation `COUNT` for distinct-value counts; for "does any exist" a SEMI join is cheaper than counting.

See Also

[group](#), [sum](#), [avg](#), [distinct](#), [semi-join](#)

Notes

For approximate counts on huge tables, count over a `SAMPLE` and scale up.

MIN (aggregate)

Syntax

```
γ <grouping-cols>, MIN(<expr>) → <alias> (Relation)

γ product_id, MIN(price) → lowest_price (Listings)
```

Description

MIN finds the smallest value in each group — the cheapest price, the earliest date, the lowest score. Use it inside a `GROUP (γ)`.

Technical Description

MIN is an aggregate used only inside `γ`; its argument is a full operand expression. It returns the minimum non-NULL value in each group and works on numbers, strings (lexicographic), and temporal types. A group with no non-NULL value yields NULL. Pushes down to SQL `MIN(...)`.

Examples

Lowest listed price per product:

```
γ product_id, MIN(price) → lowest_price (Listings)
```

Earliest order date per customer:

```
γ customer_id, MIN(order_date) → first_order (Orders)
```

Smallest absolute deviation per sensor (expression argument):

```
γ sensor, MIN(Abs(delta)) → closest (Readings)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, MIN(amount) → smallest (Sales) };
```

| region | smallest |
|--------|----------|
| east | 80 |
| west | 50 |

(2 rows)

MIN returns the smallest **value**, not the row it came from — `rep` is not available here, because there is no single rep per group. When you want the row, use **ARGMIN**: `ARGMIN(amount, rep)` returns `Bo` and `Dee`.

Over an all-NULL group MIN yields NULL rather than an error, since every NULL is skipped and nothing is left to compare.

Limitations

Returns only the value, not the row that achieved it — for the whole row use **ARGMIN** or **TOP ... PER**. Must appear inside a `γ`.

Alternatives

`ARGMIN(rank, yield)` returns a value from the minimum-rank row; `TOP 1 ... PER` keeps the actual lowest row.

See Also

[group](#), [max](#), [argmin](#), [top](#)

Notes

MIN over a temporal column gives the chronologically earliest value.

MAX (aggregate)

Syntax

```
γ <grouping-cols>, MAX(<expr>) → <alias> (Relation)
```

```
γ product_id, MAX(price) → highest_price (Listings)
```

Description

MAX finds the largest value in each group — the highest price, the latest date, the top score. Use it inside a GROUP (γ).

Technical Description

MAX is an aggregate used only inside γ; its argument is a full operand expression. It returns the maximum non-NULL value in each group and works on numbers, strings (lexicographic), and temporal types. A group with no non-NULL value yields NULL. Pushes down to SQL MAX(...).

Examples

Highest listed price per product:

```
γ product_id, MAX(price) → highest_price (Listings)
```

Most recent login per user:

```
γ user_id, MAX(login_at) → last_seen (Logins)
```

Largest single line total per order:

```
γ order_id, MAX(price * qty) → biggest_line (LineItems)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, MAX(amount) → largest (Sales) };
```

| region | largest |
|--------|---------|
| east | 120 |

```
west      200
(2 rows)
```

As with `MIN`, this is the largest **value** and not the row that carried it. The "who had the biggest deal?" question is `ARGMAX`, which is a genuinely different operation — in SQL it needs a window function or a self-join.

Limitations

Returns only the value, not the row that achieved it — for the whole row use `ARGMAX` or `TOP ... PER`. Must appear inside a `γ`.

Alternatives

`ARGMAX(rank, yield)` returns a value from the maximum-rank row; `TOP 1 ... PER` keeps the actual highest row.

See Also

[group](#), [min](#), [argmax](#), [top](#)

Notes

`MAX` over a temporal column gives the chronologically latest value.

COLLECT (aggregate / NEST)

Syntax

```
γ <grouping-cols>, COLLECT(<expr>) → <alias> (Relation)

γ customer_id, COLLECT(order_id) → order_ids (Orders)
```

Description

`COLLECT` gathers all the values in a group into a single array, producing nested output. Instead of one summary number it gives you the whole list — each customer's order ids bundled into an array, each article's tags in one column. It is the inverse of `UNNEST (μ)`, which explodes an array back into rows.

Technical Description

`COLLECT` is the `NEST` aggregate, used only inside `γ`. It gathers each group's values into an `ArrayValue`, so the output column has an array type (NF^2 nested relation). Its argument is a full operand expression, so you can collect structs too: `COLLECT({order_id, amount})`. It produces nested output and never pushes down to SQL.

Examples

Each customer's order ids as an array:

```
γ customer_id, COLLECT(order_id) → order_ids (Orders)
```

Collect structs — bundle each customer's orders as objects:

```
Built := { π customer, { order_id, amount } → o (Orders) };
query { γ customer, COLLECT(o) → orders (Built) };
```


Tags per article:

```
γ article_id, COLLECT(tag) → tags (ArticleTags)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, COLLECT(rep) → reps (Sales) };
```

```
region  reps
-----
east    [Ada, Bo]
west    [Cy, Dee]
(2 rows)
```

Where every other aggregate reduces a group to one scalar, `COLLECT` keeps all the values as an array — the relation is no longer flat. μ is the exact inverse, expanding that array back into rows.

`COLLECT` is also the one aggregate that **keeps NULLs** (matching `array_agg` and `$push`): `COLLECT(bonus)` yields `[10, NULL]` and `[25, NULL]`, not `[10]` and `[25]`.

Limitations

Produces nested (array-valued) columns, which do not push down to SQL and may need `UNNEST` (μ) to flatten again downstream. Order within the collected array is not guaranteed unless the input is ordered. Unlike the other aggregates, `COLLECT` **keeps NULLs** rather than skipping them (matching `array_agg` / `$push`), so the collected array always has one element per row in the group.

Alternatives

`UNNEST` (μ) is the opposite direction. For a single representative value per group use `MIN`/`MAX`/`ARGMAX` instead of collecting all of them.

See Also

[unnest](#), [group](#), [struct-construction](#), [array-construction](#)

Notes

`COLLECT` followed by μ round-trips back to the original rows; the optimizer can collapse that round trip (NEST-001).

ARGMAX (aggregate)

Syntax

```
γ <grouping-cols>, ARGMAX(<rank-expr>, <yield-expr>) → <alias> (Relation)
```

```
γ customer_id, ARGMAX(amount, order_id) → biggest_order (Orders)
```

Description

ARGMAX answers "for the row with the biggest X, give me Y". For each group it finds the row where the rank expression is largest and returns the yield expression's value from that row — for example, the order id of each customer's largest order, or the name of the top scorer in each class. It is the "row with the maximum" lookup that SQL needs a window function or self-join for.

Technical Description

ARGMAX(rank, yield) is a two-argument aggregate used only inside γ: it returns the yield expression evaluated on the row whose rank expression is the group maximum. Ties resolve to the first such row; rows with a NULL rank are skipped; an empty group yields NULL. It infers as the yield expression's type and never pushes down to SQL.

Examples

The id of each customer's largest order:

```
γ customer_id, ARGMAX(amount, order_id) → biggest_order (Orders)
```

The name of the top scorer per class:

```
γ class, ARGMAX(score, student_name) → top_student (Results)
```

Yield a computed expression from the max-revenue row:

```
γ region, ARGMAX(revenue, units * price) → top_line_value (Sales)
```

Worked Example

Four rows of sales, two of them with no bonus recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, ARGMAX(amount, rep) → top_rep (Sales) };
```

| region | top_rep |
|--------|---------|
| east | Ada |
| west | Cy |

(2 rows)

`MAX(amount)` would return 120 and 200 — the amounts. `ARGMAX(amount, rep)` returns *who* achieved them. Read the two arguments as "rank by the first, return the second".

That is the "row with the maximum" question, which SQL answers only with a window function or a self-join. Because the answer is a specific row's value, `ARGMAX` never pushes down to SQL — it is evaluated in-engine.

Limitations

Returns a single value from one row, not the whole row — for the entire row use `TOP 1 ... PER`. Ties pick the first encountered row (not all of them).

Alternatives

`TOP 1 rank DESC PER` keys keeps the actual top row with all its columns. `MAX` returns the maximum value itself rather than another column from that row.

See Also

[argmin](#), [max](#), [top](#), [group](#)

Notes

`ARGMAX` is the aggregate counterpart to the `TOP` operator; use `ARGMAX` for a value, `TOP` for the rows.

ARGMIN (aggregate)

Syntax

```
γ <grouping-cols>, ARGMIN(<rank-expr>, <yield-expr>) → <alias> (Relation)
```

```
γ route_id, ARGMIN(price, carrier) → cheapest_carrier (Fares)
```

Description

`ARGMIN` answers "for the row with the smallest X, give me Y". For each group it finds the row where the rank expression is smallest and returns the yield expression's value from that row — for example the carrier offering the cheapest fare on each route, or the date of each account's first transaction.

Technical Description

`ARGMIN(rank, yield)` is a two-argument aggregate used only inside `γ`: it returns the `yield` expression evaluated on the row whose `rank` expression is the group minimum. Ties resolve to the first such row; rows with a `NULL` rank are skipped; an empty group yields `NULL`. It infers as the yield expression's type and never pushes down to SQL.

Examples

The cheapest carrier on each route:

```
γ route_id, ARGMIN(price, carrier) → cheapest_carrier (Fares)
```

The product with the lowest stock per warehouse:

```
γ warehouse, ARGMIN(stock, product_id) → scarcest (Inventory)
```

Date of each account's first transaction:

```
γ account_id, ARGMIN(txn_date, txn_date) → first_txn (Transactions)
```

Worked Example

Four rows of sales, two of them with no `bonus` recorded — the NULLs are there deliberately, because they are what makes the rules below visible.

```
Sales := [
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query { γ region, ARGMIN(amount, rep) → weakest_rep (Sales) };
```

| region | weakest_rep |
|--------|-------------|
| east | Bo |
| west | Dee |

(2 rows)

The mirror of **ARGMAX**: rank by `amount`, return that row's `rep`, taking the group **minimum**. `MIN(amount)` gives 80 and 50; this gives the reps who recorded them.

Neither argument has to be a bare column. Ranking by an expression asks a different question — "whose deal was closest to the 100 target?":

```
query { γ region, ARGMIN(Abs(amount - 100), rep) → closest (Sales) };
```

| region | closest |
|--------|---------|
| east | Ada |
| west | Dee |

(2 rows)

The result takes the yield expression's type.

Limitations

Returns a single value from one row, not the whole row — for the entire row use TOP 1 rank ASC ... PER. Ties pick the first encountered row.

Alternatives

TOP 1 rank ASC PER keys keeps the actual lowest row with all its columns. MIN returns the minimum value itself rather than another column from that row.

See Also

[argmax](#), [min](#), [top](#), [group](#)

Notes

ARGMIN is the minimum-side mirror of ARGMAX.

Window — sliding / cumulative aggregate (ROLLING)

Syntax

```
ROLLING <agg>(<expr>) OVER <n> ROWS SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
ROLLING <agg>(<expr>) OVER ALL ROWS SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
```

where <agg> ∈ { SUM, AVG, COUNT, MIN, MAX }.

```
ROLLING AVG(price) OVER 3 ROWS SORT trade_time ASC PER ticker AS avg3 (StockTicks)
ROLLING SUM(revenue) OVER ALL ROWS SORT month ASC PER region AS running (MonthlySales)
ROLLING MAX(score) OVER ALL ROWS SORT ts ASC AS running_max (Events)
```

Description

A window computes a per-row value over a group of neighbouring rows and **adds it as a new column without collapsing or filtering any rows**. It is the composable, algebraic form of SQL's `OVER (PARTITION BY ... ORDER BY ... ROWS ...)` clause — the tool for moving averages, running totals, and other "value computed over an ordered window" questions.

`ROLLING` partitions the input by the optional `PER` keys (the whole relation is a single partition when `PER` is omitted), orders each partition by the `SORT` keys, and for every row aggregates over the rows in its **frame**:

- `OVER n ROWS` — a *trailing* window: the current row plus the preceding $n - 1$ rows (so `OVER 3 ROWS` is a 3-row sliding window). This is the moving-average case.
- `OVER ALL ROWS` — the *cumulative* frame: every row of the partition up to and including the current row (running total, running max, ...).

The output schema is the input schema with the `AS` column appended. Every input row is preserved and carries its computed value.

Technical Description

`ROLLING` is the window operator (`WindowNode`). Unlike `TOP` (which filters rows) and `y/DOWNSAMPLE` (which collapse rows), a window adds a column without removing any. It buffers per partition (`[bag]` materialisation).

A **bounded** frame (`OVER n ROWS`) is materialisation-safe over an unbounded input: the per-partition buffer never exceeds n rows, so the boundedness checker treats it like λ 's rescue and permits it over a generator. A **cumulative** frame (`OVER ALL ROWS`) is blocking and is rejected over a provably-unbounded input.

SQL pushdown

When the input is backed by a JDBC connection and the dialect supports window functions (PostgreSQL, DuckDB, SQLite, SQL Server, Db2 and the GENERIC H2 dialect; MySQL is conservative since the minimum MySQL 5.x version predates window functions), `ROLLING` is folded into the SQL scan as an ANSI SQL:2003 `OVER` clause:

```
SELECT ticker, t, price,
       AVG(price) OVER (PARTITION BY ticker ORDER BY t ASC
                       ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS avg3
FROM ticks
```

A `σ` (WHERE) above a `ROLLING` blocks because SQL `WHERE` cannot reference a window column at the same query level; the selection is evaluated in-engine instead. `τ` (ORDER BY) and `λ` (LIMIT) remain pushable on top of a pushed window expression.

SUM/AVG/MIN/MAX ignore NULL values within the frame (matching `y`); COUNT counts the rows in the frame. The output column type is the argument expression's type for SUM/AVG/MIN/MAX and NUMBER for COUNT.

Examples

3-row moving average of closing price per ticker — the frame is the current row and the two before it, so it is short at the start of a partition and slides thereafter (worked through with real output below):

```
ROLLING AVG(price) OVER 3 ROWS SORT trade_time ASC PER ticker AS avg3 (Ticks)
```

Cumulative revenue per region, ordered by month:

```
ROLLING SUM(revenue) OVER ALL ROWS SORT month ASC PER region AS running_total (MonthlySales)
```

Global running maximum (no PER — one partition over the whole relation):

```
ROLLING MAX(score) OVER ALL ROWS SORT ts ASC AS running_max (Events)
```

Worked Example

Monthly revenue for two regions. A window adds a column and keeps every row — which is what makes it different from `y`, and what makes it worth seeing beside its input.

```
MonthlyRevenue := [
region	month	revenue
East	1	100
East	2	120
East	3	90
West	1	80
West	2	95
];

query {
  ROLLING SUM(revenue) OVER ALL ROWS SORT month ASC PER region AS running
    (MonthlyRevenue)
};
```

| region | month | revenue | running |
|--------|-------|---------|---------|
| East | 1 | 100 | 100 |
| East | 2 | 120 | 220 |
| East | 3 | 90 | 310 |
| West | 1 | 80 | 80 |
| West | 2 | 95 | 175 |

(5 rows)

Five rows in, five rows out, each carrying the running total of its own region: East accumulates 100 → 220 → 310, and West starts again from 80 because PER region makes each region its own partition.

The bounded frame is the other half of the operator. OVER 2 ROWS averages the current row and the one before it:

```
query {
  ROLLING AVG(revenue) OVER 2 ROWS SORT month ASC PER region AS avg2
    (MonthlyRevenue)
};
```

| region | month | revenue | avg2 |
|--------|-------|---------|------|
| East | 1 | 100 | 100 |
| East | 2 | 120 | 110 |
| East | 3 | 90 | 105 |
| West | 1 | 80 | 80 |
| West | 2 | 95 | 87.5 |

(5 rows)

Month 1 of each region has no preceding row, so its frame holds one row and the average is just that row's own value — a trailing frame is not skipped at the partition's edge, it is simply shorter.

Limitations

The aggregate must be one of `SUM`, `AVG`, `COUNT`, `MIN`, `MAX` (not `COLLECT`, `ARGMAX`, or `ARGMIN`):

```
query { ROLLING COLLECT(price) OVER 3 ROWS SORT trade_time AS recent (StockTicks) };
```

The `AS` column must not clash with an existing input column. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it. SQL pushdown is supported on PostgreSQL and GENERIC (H2) JDBC backends; MySQL and MongoDB sources always evaluate `ROLLING` in-engine. A `σ` immediately above a `ROLLING` over a pushed source runs in-engine (SQL `WHERE` cannot reference window results at the same query level).

Alternatives

For "N highest rows per group" use `TOP ... PER` (a row filter). For one collapsed row per group use `γ`. For time-bucket consolidation that collapses rows use `DOWNSAMPLE`.

See Also

[log source](#) — `ROLLING` over a web server's access log, among other windowing examples on one real dataset

[group, sort, top](#)

Notes

`ROLLING` and `WINDOW` are reserved keywords; relations or columns previously named `rolling/window` must be renamed. The frame keyword `ALL` is contextual and remains usable as a column name elsewhere.

Partition pruning (WINDOW-001). A selection that fixes a `PER` (partition) key to a constant above a rolling window — e.g. `σ ticker = "ACME" (ROLLING AVG(price) ... PER ticker (Ticks))` — is pushed *below* the operator by the optimizer, so the rolling aggregate is computed only for the matching partition: `ROLLING AVG(price) ... PER ticker (σ ticker = "ACME" (Ticks))`. This is safe because the frame never crosses a partition boundary. Only equalities on a partition key are pushed; predicates on other columns stay as a residual `σ` above the operator. Look for a `WINDOW-001` record in `--optimize` output.

Window — ranking functions (WINDOW ROW_NUMBER / RANK / DENSE_RANK / PERCENT_RANK / NTILE)

Syntax

```
WINDOW ROW_NUMBER()    SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW RANK()          SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW DENSE_RANK()    SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW PERCENT_RANK()  SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW NTILE(<n>)      SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)

WINDOW RANK()          SORT units_sold DESC PER category_id AS rnk    (Products)
WINDOW NTILE(4)        SORT ytd_revenue DESC PER region              AS qtile (Sales)
```

Description

A ranking window assigns each row a number based on its position within an ordered partition and **adds it as a new column without collapsing or filtering any rows**. All five functions share the same structure — partition by the optional `PER` keys, order each partition by the `SORT` keys, then number the rows:

- `ROW_NUMBER()` — a unique, sequential 1-based integer per partition (1, 2, 3, ...); ties are broken by the row order within equal sort keys.
- `RANK()` — tied rows (equal under the `SORT` keys) share a rank; the next rank **skips** the gap left by the tie (1, 1, 3, ...).
- `DENSE_RANK()` — tied rows share a rank; the next rank does **not** skip (1, 1, 2, ...).
- `PERCENT_RANK()` — the relative rank $(\text{rank} - 1) / (\text{rows} - 1)$ as a fraction in $[0.0, 1.0]$; fixed at 0.0 for a single-row partition.
- `NTILE(n)` — divides the partition into `n` buckets as evenly as possible and assigns each row its bucket number 1..n. When the partition does not divide evenly, the leftover rows go to the **lower** buckets (bucket 1 first).

The output schema is the input schema with the `AS` column (always `NUMBER`) appended. Every input row is preserved and carries its rank.

Technical Description

Ranking functions are the window operator (`WindowNode` with a `RankingWindow` function). Unlike `TOP` (which filters to the top-k rows and adds no column) and `y/DOWNSAMPLE` (which collapse rows), a window adds a column without removing any. The operator buffers per partition (`[bag]` materialisation) and is blocking over a provably-unbounded input.

SQL pushdown

When the source is backed by a JDBC connection and the dialect supports window functions (PostgreSQL, DuckDB, SQLite, SQL Server, Db2 and the GENERIC H2 dialect; MySQL is conservative), the ranking window is folded into the SQL scan:

```
SELECT dept, emp_id, salary, ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC) AS rn
FROM employees
```

A `σ` immediately above a ranking window over a pushed source is evaluated in-engine (SQL `WHERE` cannot reference window columns at the same query level). MongoDB sources evaluate windows in-engine.

Ties are detected by comparing rows under the `SORT` keys: two rows are tied iff they compare equal on every sort key. `NTILE(n)` requires a positive integer literal `n` (no column references).

`TOP k ... PER` and ranking windows are complementary, not overlapping: `TOP` keeps only the top-`k` rows, while a ranking window keeps **every** row with its rank and lets the caller decide what to do — typically filter downstream with `σ`.

Examples

Top sellers per category — rank, then filter (the three integer functions are compared on one result below):

```
Ranked := { WINDOW RANK() SORT units_sold DESC PER category_id AS rnk (Products) };
query { σ rnk ≤ 3 (Ranked) };
```

Latest event per user — deduplication with `ROW_NUMBER`: number each user's events newest-first, then keep only number 1. This is the "one row per key" idiom SQL writes as `ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...) = 1`, and it is worked through with real output below.

```
Latest := { WINDOW ROW_NUMBER() SORT event_time DESC PER user_id AS rn (Events) };
query { σ rn = 1 (Latest) };
```

Second-highest distinct salary per department (`DENSE_RANK`) — because `DENSE_RANK` never skips, `dr = 2` selects the runner-up salary *tier*, even when several employees share the top salary:

```
Tiered := { WINDOW DENSE_RANK() SORT salary DESC PER dept_id AS dr (Salaries) };
query { σ dr = 2 (Tiered) };
```

`RANK` would label that same employee 3 when two share the top salary, which is why `DENSE_RANK` is the tool for "Nth-highest distinct value".

Salary percentile within department (`PERCENT_RANK`):

```
WithPct := { WINDOW PERCENT_RANK() SORT salary ASC PER dept AS percent_rank (Employees) };
query { σ percent_rank ≥ 0.9 (WithPct) }; -- the top decile per department
```

Quartiles by YTD revenue per region (`NTILE`):

```
WINDOW NTILE(4) SORT ytd_revenue DESC PER region AS qtile (Sales)
```

A 5-row partition split into 2 buckets (`NTILE(2)`) assigns 1, 1, 1, 2, 2 — the remainder row goes to the lower bucket.

Worked Example

The three integer ranking functions differ only in how they treat a tie, so the way to see them is side by side over data that has one. `toys` has two products tied on 30 units:

```
Products := [
category	product	units
toys	yo-yo	30
toys	kite	30
toys	drum	10
books	atlas	20
];
```

```

query {
    WINDOW DENSE_RANK() SORT units DESC PER category AS dense (
        WINDOW RANK() SORT units DESC PER category AS rnk (
            WINDOW ROW_NUMBER() SORT units DESC PER category AS row_num
                (Products)))
};

```

| category | product | units | row_num | rnk | dense |
|----------|---------|-------|---------|-----|-------|
| toys | yo-yo | 30 | 1 | 1 | 1 |
| toys | kite | 30 | 2 | 1 | 1 |
| toys | drum | 10 | 3 | 3 | 2 |
| books | atlas | 20 | 1 | 1 | 1 |

(4 rows)

Windows compose, which is why all three fit in one query: each adds its column and passes every row on.

Read the `toys` partition across: `row_num` numbers the tied rows 1 and 2 — it never ties, so it has to break the tie arbitrarily. `rnk` gives both 1 and then jumps to 3, leaving a gap the size of the tie. `dense` gives both 1 and continues at 2. `books` has a single row, so all three read 1.

One row per key. The most common use of `ROW_NUMBER` is not to show the number but to throw it away again: number each partition in the order you care about, keep number 1, drop the column.

```

Events := [
user_id	event_time	action
7	2026-06-01	login
7	2026-06-03	logout
7	2026-06-02	view
9	2026-06-05	login
];

Latest := { WINDOW ROW_NUMBER() SORT event_time DESC PER user_id AS rn (Events) };

query { π user_id, event_time, action (σ rn = 1 (Latest)) };

```

| user_id | event_time | action |
|---------|------------|--------|
| 7 | 2026-06-03 | logout |
| 9 | 2026-06-05 | login |

(2 rows)

User 7's other two events were numbered 2 and 3 and filtered out; user 9's single event is its own number 1. Note that the window had to run over every row to produce a result of two — ranking is blocking, and `σ rn = 1` is a filter on its output, not a shortcut through it.

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

`NTILE(n)` takes a positive integer literal; `ROW_NUMBER`/`RANK`/`DENSE_RANK`/`PERCENT_RANK` take no argument. The `AS` column must not clash with an existing input column. Offset window functions (`WINDOW LAG / LEAD / FIRST_VALUE / LAST_VALUE`) are a sibling family — see

Window / Offset. SQL pushdown is supported on PostgreSQL, DuckDB, SQLite, SQL Server, Db2 and GENERIC (H2) backends; MySQL and MongoDB always evaluate ranking windows in-engine.

Alternatives

For "N highest rows per group" as a row filter use **TOP ... PER**. For sliding/cumulative aggregates use **ROLLING**. For one collapsed row per group use **Y**.

See Also

[rolling, group, sort, top](#)

Notes

ROLLING and **WINDOW** are reserved keywords; the function names (**ROW_NUMBER**, **RANK**, **DENSE_RANK**, **PERCENT_RANK**, **NTILE**) are **contextual** — parsed only inside a **WINDOW** clause — so they remain usable as column names elsewhere.

Partition pruning (WINDOW-001). A selection that fixes a **PER** (partition) key to a constant above a window — e.g. `σ region = "WEST" (WINDOW ROW_NUMBER() ... PER region (Sales))` — is pushed *below* the operator by the optimizer, so the window is computed only for the matching partition and the others are never ranked: `WINDOW ROW_NUMBER() ... PER region (σ region = "WEST" (Sales))`. This is safe because the window frame never crosses a partition boundary. Only equalities on a partition key are pushed; a predicate on a non-partition column or on the computed window column (e.g. `rk ≤ 3`) stays as a residual `σ` above the operator. Look for a **WINDOW-001** record in `--optimize` output.

Window — offset functions (WINDOW LAG / LEAD / FIRST_VALUE / LAST_VALUE)

Syntax

```
WINDOW LAG(<expr> [, <offset> [, <default>]]) SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW LEAD(<expr> [, <offset> [, <default>]]) SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW FIRST_VALUE(<expr>) SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)
WINDOW LAST_VALUE(<expr>) SORT <col> [ASC|DESC] [, ...] [PER <key> [, ...]] AS <col> (Relation)

WINDOW LAG(revenue, 1) SORT month ASC PER region AS prev_revenue (MonthlyRevenue)
WINDOW LEAD(event_time) SORT event_time ASC PER user_id AS next_time (ClickEvents)
WINDOW FIRST_VALUE(price) SORT trade_time ASC PER session_id AS open_price (Trades)
```

Description

An offset window reads a value from **another row** within the same ordered partition and **adds it as a new column without collapsing or filtering any rows**. The four functions differ only in which row they read:

- **LAG**(`expr`, `n`, `default`) — the value of `expr` `n` rows **before** the current row in the sort order. At the partition's leading edge (no such row) it emits `default`, or `NULL` when `default` is omitted.
- **LEAD**(`expr`, `n`, `default`) — the value of `expr` `n` rows **after** the current row. Past the partition's end it emits `default`, or `NULL` when omitted.
- **FIRST_VALUE**(`expr`) — the value of `expr` in the **first** row of the partition, repeated for every row.

- `LAST_VALUE(expr)` — the value of `expr` in the **last** row of the partition, repeated for every row.

For `LAG/LEAD` the offset `n` defaults to `1` (the immediately adjacent row) and must be a positive integer literal. `FIRST_VALUE/LAST_VALUE` take only the value expression.

The output schema is the input schema with the `AS` column appended; its type follows the referenced expression's type (ANY for a complex expression). Every input row is preserved.

Technical Description

Offset functions are the window operator (`WindowNode` with an `OffsetWindow` function). Like the ranking and rolling families they add a column without removing any row. The operator buffers each partition, sorts it by the `SORT` keys, and indexes by position (`[bag]` materialisation); it is blocking over a provably-unbounded input.

SQL pushdown

When the source is backed by a JDBC connection and the dialect supports window functions (PostgreSQL, DuckDB, SQLite, SQL Server, Db2 and the GENERIC H2 dialect; MySQL is conservative), the offset window is folded into the SQL scan:

```
SELECT region, month, revenue, LAG(revenue, 1) OVER (PARTITION BY region ORDER BY month ASC) AS prev
FROM monthly_revenue
```

A `σ` immediately above an offset window over a pushed source is evaluated in-engine. MongoDB sources evaluate windows in-engine.

`LAG/LEAD` require a constant, positive-integer offset (no column references). The default value, when given, is evaluated against the partition's first row (constants are the typical case).

Examples

Month-over-month revenue delta (`LAG`), worked through with real output below:

```
WithPrev := { WINDOW LAG(revenue, 1) SORT month ASC PER region AS prev_revenue (MonthlyRevenue) };
query { π region, month, revenue, revenue - prev_revenue → mom_delta (WithPrev) };
```

The first row of each partition has no predecessor, so `prev_revenue` is `NULL` and the arithmetic delta propagates `NULL`. (Supply a default — `LAG(revenue, 1, 0)` — to anchor the first delta to `0` instead.)

Time between events / session gap (`LEAD` over a `TIMESTAMP`, the substrate for sessionization) — `next_time - event_time` uses temporal arithmetic and yields a `DURATION`:

```
ClickEvents(user_id, event_time):
  7 2026-06-01T10:00:00Z      7 2026-06-01T10:05:00Z      7 2026-06-01T10:40:00Z
```

```
WithNext := { WINDOW LEAD(event_time, 1) SORT event_time ASC PER user_id AS next_time (ClickEvents) };
query { π user_id, event_time, next_time - event_time → gap (WithNext) };
```

The last event per user has no successor, so its ``gap`` is ``NULL``.

Opening and closing price per trading session (`FIRST_VALUE/LAST_VALUE`):

```
WithBounds := {
  WINDOW FIRST_VALUE(price) SORT trade_time ASC PER session_id AS open_price (Trades)
};
```

Every row of a session carries that session's first (opening) price; swap in `LAST_VALUE` for the closing price.

Worked Example

The month-over-month question, which is what LAG is for:

```
MonthlyRevenue := [  
region	month	revenue
East	1	100
East	2	120
East	3	90
West	1	80
West	2	95
];  
  
query {  
  WINDOW LAG(revenue, 1) SORT month ASC PER region AS prev  
    (MonthlyRevenue)  
};
```

| region | month | revenue | prev |
|--------|-------|---------|------|
| East | 1 | 100 | NULL |
| East | 2 | 120 | 100 |
| East | 3 | 90 | 120 |
| West | 1 | 80 | NULL |
| West | 2 | 95 | 80 |

(5 rows)

Each row now carries the previous month's revenue from its own region. Month 1 has no previous row, so `prev` is `NULL` — and note that West's month 1 does **not** read East's month 3: `PER region` stops the window at the partition edge rather than running off the end of one partition into the next.

Projecting the difference is the usual next step:

```
query {  
  π region, month, revenue - prev → change (  
    WINDOW LAG(revenue, 1) SORT month ASC PER region AS prev  
      (MonthlyRevenue))  
};
```

| region | month | change |
|--------|-------|--------|
| East | 1 | NULL |
| East | 2 | 20 |
| East | 3 | -30 |
| West | 1 | NULL |
| West | 2 | 15 |

(5 rows)

The rows with no predecessor come out `NULL` rather than as a change of zero, which is the honest answer: there is no previous month to have changed from. Give LAG a third argument (`LAG(revenue, 1, 0)`) if a zero suits the report better.

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

LAG/LEAD take an optional positive-integer-literal offset (default 1) and an optional default value (default NULL); FIRST_VALUE/LAST_VALUE take only the value expression. The AS column must not clash with an existing input column. SQL pushdown is supported on PostgreSQL, DuckDB, SQLite, SQL Server, Db2 and GENERIC (H2) backends; MySQL and MongoDB always evaluate offset windows in-engine.

Alternatives

For sliding/cumulative aggregates use [ROLLING](#). For per-partition rank columns use [Window / Ranking](#). For one collapsed row per group use [γ](#).

See Also

[log source](#) — WINDOW LAG over a web server's access log, among other windowing examples on one real dataset

[window-ranking](#), [rolling](#), [sort](#), [group](#)

Notes

WINDOW is a reserved keyword; the function names (LAG, LEAD, FIRST_VALUE, LAST_VALUE) are **contextual** — parsed only inside a WINDOW clause — so they remain usable as column names elsewhere.

Partition pruning (WINDOW-001). A selection that fixes a PER (partition) key to a constant above a window — e.g. σ region = "WEST" (WINDOW LAG(amount) ... PER region (Sales)) — is pushed *below* the operator by the optimizer, so the offset is computed only for the matching partition: WINDOW LAG(amount) ... PER region (σ region = "WEST" (Sales)). This is safe because an offset never reaches across a partition boundary. Only equalities on a partition key are pushed; predicates on other columns stay as a residual σ above the operator. Look for a WINDOW-001 record in `--optimize` output.

8. Nested data (NF²)

Relix relations can hold structs and arrays. These operators build and flatten nested values.

| Page | What it does |
|---------------------|--|
| Struct construction | Build a nested object { ... } in a projection |
| Array construction | Build an array [...] in a projection |
| COLLECT | Aggregate a group's values into an array (NEST) |
| Unnest | μ / UNNEST — expand an array into rows (the inverse of COLLECT) |
| WITH ORDINALITY | Add a 1-based position column to an unnest |
| Unpivot | UNPIVOT — fold named columns into rows (wide → long) |
| Pivot | PIVOT ... BY ... PER — turn distinct key values into columns (long → wide) |
| Tree | TREE key BY parentKey [ORDER ...] AS children — fold an adjacency relation into nested documents (recursive COLLECT) |

Struct Construction ({ ... })

Syntax

```
{ field: expr, ... }  
{ name }           -- shorthand for { name: name }  
  
 $\pi$  id, { name, age } → person (Users)  
 $\pi$  id, { full: name } → person (Users)
```

Description

Struct construction builds a nested object value inside a projection — bundling several columns into one structured column. Instead of flat `name` and `age` columns you can produce a single `person` column holding { `name`, `age` }. It is how you reshape flat rows into nested, document-style output.

Technical Description

{ `field`: `expr`, ... } builds a `StructValue` (`StructConstruction` operand); the shorthand { `name` } expands to { `name`: `name` }. It is parsed wherever an operand may appear: a projection, a grouping key, a predicate, a function argument. After `IN`, braces are a set literal instead. It evaluates to a nested struct value and is never pushed down to SQL. Its type is a `StructType` of its fields' own inferred types, computed to any depth: {`base`: `salary`, `banded`: `salary * 1.1`} infers as {`base`: `N`, `banded`: `N`} because both expressions infer as `NUMBER`.

Examples

Bundle columns into a struct (shorthand):

```
π id, { name, age } → person (Users)
```

Explicit field names:

```
π id, { full: name, years: age } → person (Users)
```

Combine with COLLECT to build an array of structs:

```
Built := { π customer, { order_id, amount } → o (Orders) };  
query { γ customer, COLLECT(o) → orders (Built) };
```

Worked Example

Flat columns folded into a struct, and read back out again.

```
People := [  
id	first	last	city
1	Ada	Byron	London
2	Grace	Hop	Boston
];
```

```
Nested := { π id, {first: first, last: last, home: {city: city}} → person (People) };
```

```
query { Nested };
```

```
id  person  
---  -----  
1  {first: Ada, last: Byron, hom...  
2  {first: Grace, last: Hop, hom...  
(2 rows)
```

One column instead of three, and the engine knows its shape. Constructing a value does not put it beyond inference: each field takes the type of the expression that built it, to any depth. The catalog is where that claim can be read back rather than taken on trust:

```
query { π column, type (σ relation = "Nested" (relidx.columns)) };
```

```
column  type  
-----  ---  
id      N  
person  {first:S,last:S,home:{city:S}}  
(2 rows)
```

person is a struct of three known field types, not ANY. That matters downstream, because a known shape is a shape the engine can check a field name against.

Reading a field is a dotted projection, and it nests as far as the data does:

```
query { π id, person.first → first, person.home.city → city (Nested) };
```

```
id  first  city  
---  -----  
1  Ada    London  
2  Grace  Boston  
(2 rows)
```

Without an alias the output column takes the *field's* name rather than the whole path, so `π person.first` produces a column called `first`.

A field reference works anywhere an ordinary column reference does — inside a predicate, a function call, an aggregate:

```
query { γ MAX(Len(person.first)) → longest (σ person.home.city = 'London' (Nested)) };
```

```
  longest
  ───
      3
(1 row)
```

A field name the struct does not declare is an **analysis** error, not a surprise at run time — the same check a misspelled column gets.

Limitations

Produces a nested value that does not push down to SQL. A field whose expression is itself untyped (a path through an ANY column, say) contributes ANY to the struct's type — the shape is only as precise as what it is built from.

Alternatives

Array construction ([...]) for ordered lists rather than named fields. Keep columns flat if the consumer cannot handle nested output.

See Also

[array-construction](#), [project](#), [collect](#), [unnest](#)

Notes

Because a struct is an operand, braces around a γ grouping key are not an error. γ {region}, SUM(amount) → total (Sales) groups by a one-field struct and returns one struct column called group in place of region. See [group](#) for the example.

Access a struct field downstream with dotted projection (e.g. π person.name) — worked through above. The same spelling is used for a relation qualifier above a join (π rooms.name), and that reading is tried first, so a struct column can never shadow one. A path whose last segment happens to name a column of the same relation still reads as a path: over a heading of (name, skills) where a skill has its own name, skills.name is the skill's, because no column named name originates from a relation called skills.

Array Construction ([...])

Syntax

```
[ expr, ... ]
```

```
π label, [x, y] → coords (Points)
```

Description

Array construction builds a list value inside a projection — gathering several expressions into one ordered array column. For example, packing x and y into a single coords array. It is the per-row counterpart to COLLECT (which builds an array across the rows of a group).

Technical Description

[expr, ...] builds an ArrayValue (ArrayConstruction operand) from the listed expressions in order. It is parsed in operand position only. It evaluates to a nested array value and is never pushed down to SQL. Its type is an ArrayType of the type its elements agree on — [x, y] over two NUMBER columns infers as [N] — falling back to [?] when they disagree or the array is empty, since a heterogeneous list has no element type to name.

Examples

Pack coordinates into an array:

```
π label, [x, y] → coords (Points)
```

Build a fixed-size feature vector:

```
π id, [height, weight, age] → features (Subjects)
```

Then explode it back with UNNEST:

```
μ coords (π label, [x, y] → coords (Points))
```

Worked Example

A weather station records three readings per site and the analysis wants them as one ordered vector per row, rather than as three columns that every downstream query has to name individually.

```
Readings := [
site	dawn	noon	dusk
north	4	17	9
south	8	21	13
];
```

```
Vectors := { π site, [dawn, noon, dusk] → temps (Readings) };
```

```
query { Vectors };
```

| site | temps |
|-------|-------------|
| north | [4, 17, 9] |
| south | [8, 21, 13] |

(2 rows)

The array's type is the type its elements agree on — all three are NUMBER here, so the column is an array of NUMBER rather than an array of anything:

```
query { π column, type (σ relation = "Vectors" (relix.columns)) };
```

| column | type |
|--------|------|
| site | S |
| temps | [N] |

(2 rows)

Ordering is positional and preserved, which is what makes μ ... WITH ORDINALITY meaningful over it: the index is the reading's place in the day.

```
query { μ temps WITH ORDINALITY slot (Vectors) };
```

| site | temps | slot |
|-------|-------|------|
| north | 4 | 1 |
| north | 17 | 2 |
| north | 9 | 3 |
| south | 8 | 1 |
| south | 21 | 2 |
| south | 13 | 3 |

(6 rows)

Mix the element types and there is no common type left to name, so the element type falls back to `?`:

```
Mixed := { π site, [dawn, site] → pair (Readings) };
query { π column, type (σ relation = "Mixed" (relix.columns)) };
```

| column | type |
|--------|------|
| site | S |
| pair | [?] |

(2 rows)

Limitations

Produces a nested value that does not push down to SQL. Recognised only in operand position within a projection. Elements may be of mixed type, and an array of mixed elements infers as an array of ANY — the element type is the one thing the construction cannot state when its elements disagree.

Alternatives

COLLECT (inside γ) builds an array from many rows of a group, not from columns of one row. Struct construction (`{ ... }`) for named fields rather than an ordered list.

See Also

[struct-construction](#), [collect](#), [unnest](#), [project](#)

Notes

Array construction (per row) and COLLECT (per group) are the two ways to produce array columns; UNNEST (μ) flattens either back into rows.

Unnest (μ / UNNEST)

Syntax

```
 $\mu$  <array-column> (Relation)
 $\mu$  <array-column> WITH ORDINALITY <name> (Relation)
UNNEST ...

 $\mu$  items (Orders)
 $\mu$  items WITH ORDINALITY pos (Orders)
```

Description

Unnest takes a column that holds an array and expands it into one row per element — flattening nested data back into flat rows. If an order row has a list of three items, unnesting the items column turns it into three rows, one per item. It is the inverse of COLLECT (which gathers values into an array).

The optional WITH ORDINALITY clause adds a column carrying each element's position (1, 2, 3, ...), restarting for each input row.

Technical Description

$\mu_c(R)$ replaces the array column c with one output row per element of the array, carrying the element value in place of the array. WITH ORDINALITY appends a NUMBER column with the 1-based element index. There is an OUTER variant (programmatic only) that emits a NULL-element row when the array is empty. μ streams; it pushes down to Mongo as \$unwind (inner, non-ordinality form).

Examples

One row per line item from an order that stored items as an array:

```
 $\mu$  items (Orders)
```

Number each element as you explode it:

```
 $\mu$  tags WITH ORDINALITY tag_position (Articles)
```

Round-trip with COLLECT — group, gather, then explode back:

```
Built := {  $\gamma$  customer, COLLECT(order_id)  $\rightarrow$  ids (Orders) };  
query {  $\mu$  ids (Built) };
```

Worked Example

μ is the inverse of COLLECT: where COLLECT folds a group's values into one array, unnest expands an array back into one row per element. Round-tripping the two shows it directly.

```
Sales := [  
region	rep	amount	bonus
east	Ada	120	10
east	Bo	80	NULL
west	Cy	200	25
west	Dee	50	NULL
];
```

```
query {  $\mu$  reps ( $\gamma$  region, COLLECT(rep)  $\rightarrow$  reps (Sales)) };
```

| region | reps |
|--------|------|
| east | Ada |
| east | Bo |
| west | Cy |
| west | Dee |

(4 rows)

The γ produces two rows (east $\rightarrow$ [Ada, Bo], west $\rightarrow$ [Cy, Dee]); μ expands them back to the original four. The column keeps its name but changes type — reps is an array in the input and a single value in the output.

Add `WITH ORDINALITY` when the position matters:

```
Grouped := { γ region, COLLECT(rep) → reps (Sales) };
query { μ reps WITH ORDINALITY pos (Grouped) };
```

| region | reps | pos |
|--------|------|-----|
| east | Ada | 1 |
| east | Bo | 2 |
| west | Cy | 1 |
| west | Dee | 2 |

(4 rows)

The counter restarts at 1 for each input row, not once across the whole result.

Limitations

Unnest works on array-typed columns; over a schema-on-read (open) relation the column must actually hold an array at runtime. Ordinality-preserving and outer \$unwind do not push down to Mongo (run in-engine).

Alternatives

COLLECT (inside γ) is the opposite direction — many rows into one array. For a struct field whose name you know, use projection (π s.field) rather than unnest. For an object whose keys are the data — counts or settings keyed by a name the schema does not know in advance — Entries reads it as an array of {key, value} entries, which this operator then explodes.

See Also

[collect](#), [group](#), [with-ordinality](#), [array-construction](#), [entries](#)

Notes

The nest/unnest round-trip laws let the optimizer collapse μ (γ COLLECT) back to a projection (NEST-001) when the shapes line up.

WITH ORDINALITY (unnest index clause)

Syntax

```
μ <array-column> WITH ORDINALITY <name> (Relation)
UNNEST <array-column> WITH ORDINALITY <name> (Relation)

μ items WITH ORDINALITY pos (Orders)
```

Description

WITH ORDINALITY is an optional clause on UNNEST that adds a column recording each element's position as the array is exploded — 1 for the first element, 2 for the second, and so on, restarting for each input row. Use it when the order of items matters: line-item numbers, ranked tags, sequence positions.

Technical Description

The clause appends a NUMBER column (named by the required `name`) carrying the 1-based index of each unnested element within its source array; numbering restarts per input row. With the OUTER unnest variant, a no-element row gets a NULL element and NULL ordinality. The ordinality-preserving form does not push down to Mongo \$unwind (runs in-engine).

Examples

Number each exploded line item:

```
μ items WITH ORDINALITY line_no (Orders)
```

Position of each tag on an article:

```
μ tags WITH ORDINALITY tag_position (Articles)
```

Keep the sequence when flattening steps:

```
μ steps WITH ORDINALITY step_number (Recipes)
```

Worked Example

Line items gathered into an array and exploded again, numbered as they go.

```
OrderItems := [
order_id	item
100	widget
100	bolt
100	bracket
101	nut
];

Bagged := { γ order_id, COLLECT(item) → items (OrderItems) };

query { μ items WITH ORDINALITY line_no (Bagged) };
```

| order_id | items | line_no |
|----------|---------|---------|
| 100 | widget | 1 |
| 100 | bolt | 2 |
| 100 | bracket | 3 |
| 101 | nut | 1 |

(4 rows)

`line_no` restarts at 1 for each input row, so order 101's only item is line 1 rather than line 4 — the numbering is per array, not per result. Without the clause the same query returns the items with no way to recover the order they were stored in.

Limitations

The ordinality name must not clash with an existing column. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it. The clause is only valid on UNNEST (μ). The position reflects array order, which is only meaningful if the array itself is ordered.

Alternatives

Plain UNNEST (μ) without the clause when element position does not matter.

See Also

[unnest](#), [collect](#), [array-construction](#)

Notes

This mirrors SQL's UNNEST(...) WITH ORDINALITY and is the way to recover element order after flattening nested data.

Unpivot (UNPIVOT)

Syntax

```
UNPIVOT (col1, col2, ...) AS (nameColumn, valueColumn) (Relation)
```

```
UNPIVOT (q1, q2, q3, q4) AS (quarter, revenue) (QuarterlySales)
```

Description

UNPIVOT folds a set of named columns into rows: for each input row it emits one output row per listed column. The output contains all the non-listed input columns unchanged, plus two new columns: one for the source column name (a STRING) and one for the cell value (ANY).

UNPIVOT is the inverse of PIVOT — it normalises "wide" tables (one column per category) into "long" tables (one row per category).

Technical Description

For an input row r and listed columns $[c1, c2, \dots, cn]$, UNPIVOT emits n output rows. Each output row contains:

- every column of r that is NOT in the listed column set (pass-through columns), in their original order
- nameColumn: a STRING holding the source column name ($c1, c2, \dots$, or cn)
- valueColumn: the cell value $r[ci]$ from that input row

UNPIVOT is streaming — it never buffers the full input; each input row fans out to exactly n output rows. The operator is never pushed down to a source backend.

Requires at least one column in the column list. The nameColumn and valueColumn names must not clash with any of the pass-through (non-listed) columns, and must not clash with each other.

Examples

Wide quarterly sales table → long format, one row per quarter (worked through with real output below):

```
UNPIVOT (q1, q2, q3, q4) AS (quarter, revenue) (QuarterlySales)
```

Temperature sensors with side-by-side readings → normalised:

```
UNPIVOT (sensor_a, sensor_b, sensor_c) AS (sensor, temp) (Readings)
```

Combine with PIVOT for wide ↔ long round-trips:

```
Long := { UNPIVOT (q1, q2, q3, q4) AS (quarter, revenue) (QuarterlySales) };
Wide := { PIVOT revenue BY quarter PER region (Long) };
```

Worked Example

The wide table PIVOT produces, folded back into rows.

```
QuarterlySales := [
region	q1	q2
East	100000	120000
West	80000	90000
];

query { UNPIVOT (q1, q2) AS (quarter, revenue) (QuarterlySales) };
```

| region | quarter | revenue |
|--------|---------|---------|
| East | q1 | 100000 |
| East | q2 | 120000 |
| West | q1 | 80000 |
| West | q2 | 90000 |

(4 rows)

Two input rows fan out to four: one per listed column, per row. `region` is a pass-through column and repeats; `quarter` holds the *name* of the column the value came from, and `revenue` its value.

Feeding that straight back into PIVOT returns the original shape — the two operators are inverses:

```
query {
  PIVOT revenue BY quarter PER region
    (UNPIVOT (q1, q2) AS (quarter, revenue) (QuarterlySales))
};
```

| region | q1 | q2 |
|--------|--------|--------|
| East | 100000 | 120000 |
| West | 80000 | 90000 |

(2 rows)

Limitations

NULL values in the listed columns are preserved as NULL in valueColumn — there is no automatic NULL-filtering (add σ valueColumn $\neq \perp$ downstream if needed). The output schema is closed (column headers are fixed at parse time). UNPIVOT is never pushed down to SQL or MongoDB.

Alternatives

PIVOT is the reverse: turns distinct values in a column into new column headers.

See Also

[pivot](#)

Notes

The output type of `valueColumn` is `ANY` — it may hold values of different types when the listed columns have different types. Use σ or π to restrict or recast after unpivoting if uniform types are needed.

Pivot (PIVOT)

Syntax

```
PIVOT valueColumn BY keyColumn (Relation)
PIVOT valueColumn BY keyColumn PER groupKey1, groupKey2, ... (Relation)

PIVOT revenue BY quarter PER region (QuarterlySales)
PIVOT amount BY category (Transactions)
```

Description

PIVOT rotates rows into columns: it groups the input by the optional PER keys (or treats the whole relation as one group when PER is absent), then turns each distinct value of `keyColumn` into a new output column whose cell is taken from `valueColumn` for the matching row. When a group has no row for a particular key value the cell is `NULL`.

PIVOT is the inverse of UNPIVOT — it denormalises "long" tables (one row per category) into "wide" tables (one column per category).

Technical Description

PIVOT is a blocking operator (materialisation mode BAG) — it must consume the entire input before emitting output, because the full set of column headers (the distinct key values) is not known until all rows have been seen.

For a group identified by `groupKey` values `G`, and distinct key values $\{k_1, k_2, \dots, k_n\}$ seen across all rows:

- One output row is emitted per distinct group `G`
- The output row contains the `groupKey` columns (the PER columns), followed by one column per distinct key value; the column name is the string representation of the key value
- Each pivot cell holds the `valueColumn` value from the unique row in `G` whose `keyColumn` equals that header value; it is `NULL` when no such row exists

Two cases follow from "the unique row" not always being unique, or present:

- When a group holds **more than one** row for the same key value, the last one read wins and the earlier values are discarded without a diagnostic. Input order therefore decides the cell, so put a τ below the PIVOT if it matters — or aggregate first (γ) if what you wanted was a total rather than a pick.
- A row whose `keyColumn` is **NULL** contributes no column, since there is no header to name. It has still been seen, so its group is emitted like any other, holding `NULL` in every cell no other row filled.

The output schema is open (`Schema.open()`) because the column headers depend on runtime data and cannot be determined at parse time. Column order is the order the

headers were first seen, so it too depends on the input. PIVOT is never pushed down to a source backend.

Examples

Long quarterly sales table → wide format, one column per quarter (worked through with real output below):

```
PIVOT revenue BY quarter PER region (QuarterlySales)
```

Category totals as columns:

```
PIVOT total BY category (CategoryTotals)
```

Global pivot (no PER — one row total across all groups):

```
PIVOT amount BY status (Orders)
```

Combine with filtering and sorting:

```
TopN      := { σ score > 80 (Results) };
Rotated := { PIVOT score BY subject PER student_id (TopN) };
```

Worked Example

Three sales rows, two regions, two quarters — and deliberately **no q2 row for West**, because what PIVOT does with a missing cell is the part prose cannot settle.

```
QuarterlySales := [
region	quarter	revenue
East	q1	100000
East	q2	120000
West	q1	80000
];

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

| region | q1 | q2 |
|--------|--------|--------|
| East | 100000 | 120000 |
| West | 80000 | NULL |

(2 rows)

One row per region, and one column per *distinct value* of `quarter` — the column headers came out of the data, which is why the output schema is open. West's `q2` cell is NULL: there was no row to take a value from.

Limitations

The output schema is open — downstream operators that require a closed schema (such as certain projections or joins on named columns) may need a ρ rename or σ predicate based on runtime column names. PIVOT is never pushed down to SQL or MongoDB. When a group has multiple rows with the same key value, only the value from the first-encountered row is used (last-write-wins is not guaranteed; use γ SUM/MAX/MIN/... to aggregate before pivoting if deduplication is needed).

Alternatives

UNPIVOT is the reverse: folds listed columns into rows. γ + COLLECT produces a similar wide-to-row grouping when the data is already flat.

See Also

[unpivot](#), [group](#), [unnest](#)

Notes

Column headers in the output are the `asDisplayString()` rendering of the key values. For NUMBER keys the rendering is the numeric string (e.g. "2024", "3.14"). For STRING keys it is the string value itself. NULL key values are silently skipped (they do not generate a new header column).

Tree (TREE)

Syntax

```
TREE key BY parentKey [ORDER col [ASC|DESC], ...] AS childrenColumn (Relation)
```

```
TREE node_id BY parent_id ORDER ordinal AS children (reltx.plan)
```

Description

TREE folds a self-referential **adjacency relation** into a **forest of nested documents**: one output row per *root*, each carrying its whole subtree in an added nested array column. It is the recursive generalisation of COLLECT — where COLLECT (NEST) gathers a single level of children into an array, TREE follows the `key` → `parentKey` edge to fixpoint and assembles an unbounded-depth tree in one pass.

Its marquee use is dogfooding the engine's own IR — `TREE(reltx.plan)` renders a view's logical plan as a single nested document — but it is general: org charts, bills-of-materials, threaded comments, file trees, any parent-pointer hierarchy.

Technical Description

Given an input relation R with a node-identity column `key` and a parent-pointer column `parentKey`:

- A row whose `parentKey` is NULL, or references a `key` value absent from R, is a **root** (forest semantics — there may be several).
- Every other row is a **child** of the row whose `key` equals its `parentKey`.

The output schema is the input schema plus one column, `childrenColumn`, of type ANY (schema-on-read; there is no recursive static type in v1). TREE emits one row per root: the root's own (flat) input columns, followed by `childrenColumn` holding an array of child **documents**. Each child document is a struct of that child's input columns plus its own `childrenColumn` array, recursively. A leaf carries an empty children array `[]`.

Sibling order within every level — and the order of the roots themselves — follows the optional `ORDER` clause (multiple keys, each `ASC` (default) or `DESC`); with no `ORDER` clause, input order is preserved.

TREE is **blocking** (it must see the whole relation to build the forest) and is never pushed down to a source backend. Over a provably unbounded input it is a plan-time error, like other blocking operators.

Well-formedness

A tree assumes an acyclic, single-parent forest. TREE rejects malformed input at evaluation time:

- A **duplicate** key value is a key violation → evaluation error.
- A **cycle** in the key → parentKey graph (a component with no root) is a user error → evaluation error, rather than looping forever.

A **NULL** key is neither, and is not refused. Such a row is placed by its parent pointer like any other, so it appears wherever that pointer puts it — but nothing can ever be its child, because a NULL is not an identity and a parent cell holding NULL means "root" rather than "points at the keyless row". It is therefore always a leaf.

Examples

Example 1 — dogfood the IR (relix.plan)

relix.plan exposes every view's logical plan as flat adjacency — one row per node, with (node_id, parent_id) encoding the tree. TREE nests it back into the shape it describes. Consider the view Revenue := π region, amount (Orders $\bowtie$ Customers), whose four plan rows (filtered to this query) are:

```
--  $\sigma$  query = "Revenue" (relix.plan)
-- query | node_id | parent_id | ordinal | op | label
-- Revenue | 0 | (none) | 0 | Projection |  $\pi$  region, amount ← root
-- Revenue | 1 | 0 | 0 | NaturalJoin |  $\bowtie$ 
-- Revenue | 2 | 1 | 0 | Relation | Orders
-- Revenue | 3 | 1 | 1 | Relation | Customers
```

(node 0's parent_id is absent from the result, so it is the root.)

```
TREE node_id BY parent_id ORDER ordinal AS children
( $\sigma$  query = "Revenue" (relix.plan))
```

```
-- Output: one row (the Projection root) whose `children` nests the whole tree:
-- node_id | op | label | children
-- 0 | Projection |  $\pi$  region, amount | [ { node_id: 1, op: NaturalJoin, label:  $\bowtie$ ,
-- | | | | children: [ { node_id: 2, op: Relation,
-- | | | | label: Orders, children: [] },
-- | | | | { node_id: 3, op: Relation,
-- | | | | label: Customers, children: [] } ] } ] ]
```

ASCII view of the same fold (flat adjacency → nested forest):

```
0 ┌
1 └ parent 0      →      0 (Projection)
2 ┌ parent 1      ┌      1 (NaturalJoin)
3 └ parent 1      │      └ 2 (Orders)
                  └      3 (Customers)
```

Example 2 — org chart

Employees with a manager pointer; a top exec whose manager_id is NULL or points at a non-existent id becomes a root.

```
-- Input Employees: id | manager_id | name
--                  1 | 0 | Ada (manager 0 is absent → root)
```

```

--          2 | 1          | Grace
--          3 | 1          | Lin
--          4 | 2          | Edsger

TREE id BY manager_id ORDER name AS reports (Employees)

-- Output (one row, the root Ada):
--   id | manager_id | name | reports
--   1  | 0          | Ada  | [ { id: 2, name: Grace,
--                               reports: [ { id: 4, name: Edsger, reports: [] } ] },
--                               { id: 3, name: Lin, reports: [] } ] ]

```

From there it is ordinary nested data: `μ reports` walks a level (UNNEST), and `π reports` reads the nested column.

Example 3 — threaded comments

A comment thread is an adjacency relation (`parent_id` is the comment replied to); TREE rebuilds the reply tree, oldest-first, in one pass — no per-level self-join.

```

-- Input Comments: id | parent_id | author | posted_at
--                  10 | (none)    | ann    | 09:00      (top-level → root)
--                  11 | 10     | bo     | 09:05
--                  12 | 11     | ann    | 09:10
--                  13 | 10     | cy     | 09:07

TREE id BY parent_id ORDER posted_at AS replies (Comments)

-- Output (one row, comment 10):
--   id | author | replies
--   10 | ann    | [ { id: 11, author: bo,
--                               replies: [ { id: 12, author: ann, replies: [] } ] },
--                               { id: 13, author: cy, replies: [] } ] ]

```

Worked Example

An org chart is the shape everyone already has an intuition for: each row points at its manager, and Ada's pointer goes nowhere, which makes her 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 out of four, because there is one root — and that row carries the entire hierarchy in `reports`. The table format elides a nested value at the column width, which is the first thing to know about querying nested data from a terminal: use `--format json` (or project into the nesting) to see it whole. Laid out, that one cell holds

```

Ada
├─ Bob

```

```
|  └─ Dan
└─ Cara
```

nested to whatever depth the data has. The fold followed the `id → manager_id` edge to fixpoint in a single pass; nothing in the query says how deep the tree is, and nothing needs to.

`ORDER id ASC` fixes the sibling order. Without it, siblings appear in whatever order the input arrived in, which is rarely what a report wants.

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

The nested children column is typed ANY — there is no recursive static type in v1. TREE is blocking and is never pushed down to SQL or MongoDB (a Postgres `RECURSIVE CTE + json_agg` lowering is a documented follow-on). Cyclic or duplicate-key input is a runtime error, not a silently-dropped subset.

Alternatives

`COLLECT` (inside γ) nests a *single* level of children into an array; μ (`UNNEST`) is the one-level inverse. `CLOSURE` / `PATH` / `FIX` traverse the same key edges but produce reachability pairs or paths rather than nested structure.

See Also

[unnest](#), [COLLECT](#)

Notes

TREE reuses the recursion engine's cycle guard, `COLLECT`-style nesting, and the ANY schema-on-read document model. When the input schema is open (schema-on-read), the output rows are open documents with the children field added.

9. Literals & temporal types

Writing constant values, including the first-class temporal types.

| Page | What it does |
|-------------------|--|
| DATE literal | DATE '2026-01-31' |
| TIME literal | TIME '09:30:00' |
| TIMESTAMP literal | TIMESTAMP '2026-01-31T09:30:00Z' |
| DURATION literal | DURATION 'PT90M' |
| Truth relations | UNIT / EMPTY (aliases DEE / DUM) — the two zero-column relations |

DATE literal

Syntax

```
DATE '<YYYY-MM-DD>'
```

```
DATE '2026-06-15'
```

Description

A DATE literal is a calendar day with no time-of-day — just year, month, and day, like a birthday or a due date. Write it as the keyword DATE followed by an ISO-8601 date in quotes. Use it to filter or compute against date columns.

Technical Description

DATE '...' parses an ISO-8601 calendar date (YYYY-MM-DD) into a zone-less civil `java.time.LocalDate` at parse time; a malformed payload is a positioned parse error. It infers as the DATE scalar type. DATE literals push down to SQL (per dialect), except to SQLite, which has no date type. Single- or double-quoted payloads are both accepted.

Examples

Filter rows on or after a date:

```
σ due_date >= DATE '2026-01-01' (Invoices)
```

Date arithmetic — days until due (DATE – DATE → DURATION):

```
π id, (due_date - DATE '2026-06-15') → remaining (Invoices)
```

A computed deadline column:

```
π id, DATE '2026-12-31' → year_end (Projects)
```

Limitations

A DATE has no time component (use TIMESTAMP for date+time). Comparisons must be against another DATE (same-type only). The payload must be valid ISO-8601.

Alternatives

TIMESTAMP for an instant with time-of-day; TIME for a clock time only; to_date(str) to parse a date from a string column.

See Also

[timestamp-literal](#), [time-literal](#), [duration-literal](#), [to_date](#), [year](#), [month](#), [day](#)

Notes

DATE – DATE yields a DURATION; DATE ± DURATION yields a TIMESTAMP.

TIME literal

Syntax

```
TIME '<HH:MM[:SS]>'
```

```
TIME '09:30:00'
```

Description

A TIME literal is a clock time with no date — like an opening hour or a daily deadline. Write it as the keyword TIME followed by an ISO-8601 time in quotes. Use it to compare against time-of-day columns or to compute time offsets.

Technical Description

TIME '...' parses an ISO-8601 local time into a zone-less civil java.time.LocalTime at parse time; a malformed payload is a positioned parse error. It infers as the TIME scalar type. TIME literals push down to SQL (per dialect), except to SQLite, which has no time type. Single- or double-quoted payloads are both accepted.

Examples

Rows during business hours:

```
σ start_time >= TIME '09:00:00' ∧ start_time < TIME '17:00:00' (Shifts)
```

Shift a time by a duration (TIME ± DURATION → TIME, wraps at 24h):

```
π id, (start_time + DURATION 'PT30M') → grace_end (Shifts)
```

A fixed cut-off:

```
π id, TIME '23:59:59' → cutoff (Slots)
```

Limitations

A TIME has no date component, and arithmetic wraps around at 24 hours. Comparisons must be against another TIME (same-type only).

Alternatives

TIMESTAMP for a full date+time instant; DATE for a calendar day; to_time(str) to parse a time from a string.

See Also

[timestamp-literal](#), [date-literal](#), [duration-literal](#), [to_time](#), [hour](#), [minute](#), [second](#)

Notes

TIME ± DURATION yields a TIME and wraps over midnight.

TIMESTAMP literal

Syntax

```
TIMESTAMP '<ISO-8601 date-time, optional offset>'

TIMESTAMP '2026-06-15T13:40:00Z'
TIMESTAMP '2026-06-15T14:40:00+01:00'    -- normalised to UTC
```

Description

A TIMESTAMP literal is a precise moment in time — a date and a time together, as an instant. It is the type for event times, log entries, and "when did this happen" columns. Write it as the keyword `TIMESTAMP` followed by an ISO-8601 date-time in quotes; an offset is converted to UTC, and an offsetless value is read as UTC.

Technical Description

`TIMESTAMP '...'` parses an ISO-8601 date-time into a UTC `java.time.Instant` at parse time (an offset is normalised to UTC; an offsetless payload is interpreted as UTC). A malformed payload is a positioned parse error. It infers as the `TIMESTAMP` scalar type and pushes down to SQL (per dialect: a UTC wall-clock for `GENERIC/MYSQL/DB2`, a `Z`-instant for `POSTGRES` and `DUCKDB`, a `DATETIME2` of the UTC wall-clock for `SQLSERVER`) and to `Mongo` (`$date`). `SQLite`, which has no date type, is sent `none`: a predicate holding one runs in the engine. Single- or double-quoted payloads are accepted.

Examples

Events within a time window:

```
σ occurred_at >= TIMESTAMP '2026-01-01T00:00:00Z'
  ∧ occurred_at < TIMESTAMP '2026-02-01T00:00:00Z' (Events)
```

Elapsed time between two instants (TIMESTAMP – TIMESTAMP → DURATION):

```
π id, (closed - opened) → held (Tickets)
```

Add a window to an instant (TIMESTAMP ± DURATION → TIMESTAMP):

```
π id, (started + DURATION 'PT30M') → expires (Sessions)
```

Limitations

Stored as a UTC instant — there is no separate time-zone field; a local offset is folded into UTC at parse time. Comparisons must be against another `TIMESTAMP`.

Alternatives

`DATE` for a calendar day only; `TIME` for a clock time only; `to_timestamp(str)` to parse from a string column.

See Also

[date-literal](#), [time-literal](#), [duration-literal](#), [to_timestamp](#), [now](#), [date_trunc](#), [asof-join](#)

Notes

TIMESTAMP – TIMESTAMP gives a DURATION; this pairing powers AS-OF joins and sessionization over event time.

DURATION literal

Syntax

```
DURATION '<ISO-8601 duration>'

DURATION 'PT90M'      -- 90 minutes
DURATION 'PT2H30M'    -- 2 hours 30 minutes
DURATION 'P1D'        -- 1 day
```

Description

A DURATION literal is a length of time — "90 minutes", "2 hours 30 minutes", "3 days" — as opposed to a point in time. Use it to add or subtract spans from timestamps, to express thresholds (e.g. "gaps longer than 30 minutes"), and as the result of subtracting two timestamps.

Technical Description

DURATION '...' parses an ISO-8601 duration (e.g. PT90M, PT2H30M, P1D) into an exact java.time.Duration at parse time; a malformed payload is a positioned parse error. It infers as the DURATION scalar type.

Pushdown: on **Postgres** a DURATION literal folds to INTERVAL '<ISO-8601>' (e.g. INTERVAL 'PT30M'), so predicates like `σ held > DURATION 'PT30M'` push to WHERE ("held" > INTERVAL 'PT30M'). On **DuckDB**, which rejects the ISO-8601 string, it folds to an exact count of microseconds, `to_microseconds(18000000000)`; a duration finer than a microsecond does not push. On MySQL, SQLite, SQL Server, Db2 and GENERIC dialects, DURATION literals do **not** push (MySQL INTERVAL needs per-unit keywords that don't map cleanly to ISO-8601) — the predicate runs in-engine.

Examples

Sessions held longer than 30 minutes:

```
σ MINUTES(closed - opened) > 30 (Sessions)
```

Add a window to a timestamp:

```
π id, (started + DURATION 'PT30M') → expires (Sessions)
```

Scale a duration by a number (DURATION × NUMBER → DURATION):

```
π id, (base_slot * 3) → triple_slot (Bookings)
```

Limitations

Only pushed to Postgres (as `INTERVAL`) and DuckDB (as `to_microseconds`); MySQL and GENERIC run in-engine. Built from exact time units (hours/minutes/seconds/days); calendar-aware spans like "1 month" are not a fixed DURATION.

Alternatives

The MINUTES/SECONDS/DAYS functions measure a DURATION as a plain NUMBER for comparisons that read naturally (e.g. `MINUTES(gap) > 30`).

See Also

[timestamp-literal](#), [date-literal](#), [time-literal](#), [minutes](#), [seconds](#), [days](#)

Notes

DURATION arithmetic: $\text{DURATION} \pm \text{DURATION} \rightarrow \text{DURATION}$, $\text{DURATION} \times / \div \text{NUMBER} \rightarrow \text{DURATION}$, $\text{DURATION} \div \text{DURATION} \rightarrow \text{NUMBER}$, unary $-\text{DURATION} \rightarrow \text{DURATION}$.

Truth relation literals (UNIT / EMPTY, a.k.a. DEE / DUM)

Syntax

```
UNIT      -- alias: DEE
EMPTY     -- alias: DUM

query { Orders × UNIT };
query { EMPTY };
```

Description

`UNIT` and `EMPTY` are the two relations that have no columns at all. A relation with no columns can hold at most one row — the row with nothing in it — so there are exactly two of them, and they read as truth values: `UNIT` holds that one empty row ("yes"), `EMPTY` holds nothing ("no").

They are useful for two things. `UNIT` is the neutral element of $\times$: crossing any relation with it gives you that relation back unchanged, which lets a composed pipeline have a "nothing to add here" case that needs no special handling. And because a no-key $\forall$ answers a yes/no question by producing one of these two relations, crossing a result with that answer gates the result on the question — rows survive only when the answer was yes.

`DEE` and `DUM` are accepted as aliases; they are the names these relations carry in Tutorial D and the relational-model literature.

Technical Description

`UNIT/DEE` and `EMPTY/DUM` are keywords in **relation** position that parse to a `TruthRelationNode` — a leaf of the `RelNode` hierarchy, holding one boolean: `UNIT` denotes cardinality 1, `EMPTY` cardinality 0. Both infer to `Schema.empty()`, the *closed* zero-column heading (distinct from the *open* schema, which is also column-less but resolves every column reference). This is the same heading the no-key whole-relation universal quantifier $\forall : P(R)$ already produces, so every phase below the parser is width-0 tolerant by construction.

Keyword matching is case-insensitive, and there is no glyph form — like `TRUE` and `FALSE` these are literals rather than operators. All four words are reserved in relation position; backtick them (``unit``) to name a relation `unit`.

Downstream properties: cardinality is exact (1 or 0) rather than estimated, the cost tier is `INLINE`, the literal is whole-row distinct and `BOUNDED`, and it is never pushed down to a backend (it has no connection to push to). Under `--provenance` the literals *are* the semiring identities: `UNIT` is the empty tuple annotated `one()` (contributing no lineage of its own) and `EMPTY` is the empty K-relation. The planner emits an ordinary inline scan over the empty schema, so no new physical operator or executor path exists for them.

The optimizer rule **PROD-001** removes a product against `UNIT` ($R \times \text{UNIT} \rightarrow R$, $\text{UNIT} \times R \rightarrow R$). There is deliberately no companion rule for `EMPTY`: $R \times \text{EMPTY}$ yields no rows but keeps `R`'s heading, so it is not the zero-column `EMPTY` and the two cannot be substituted for each other.

Examples

`UNIT` is the identity of $\times$

`Trades  $\times$  UNIT` pairs every trade row with the one empty row, which appends nothing — the same rows, in the same column order:

| trade_id | desk | status | $\times \text{UNIT} \rightarrow$ | trade_id | desk | status |
|----------|-------|---------|----------------------------------|----------|-------|---------|
| 1 | FX | settled | | 1 | FX | settled |
| 2 | FX | pending | | 2 | FX | pending |
| 3 | RATES | settled | | 3 | RATES | settled |

`Trades  $\times$  EMPTY` is the other case: no row of `Trades` finds a partner, so the result is **empty but still three columns wide** — it is not `EMPTY` itself.

```
Trades := [
trade_id	desk	status
1	FX	settled
2	FX	pending
3	RATES	settled
];

-- Identical to `query { Trades }`; the optimizer drops the  $\times$  (PROD-001).
query { Trades  $\times$  UNIT };
```

Gating a result on a yes/no question

A no-key $\forall$ asks a question about a whole relation and answers with one of these two relations: `UNIT` when every row satisfies the predicate, `EMPTY` when some row does not. Crossing a report with that answer therefore *gates* the report — it survives only when the answer is yes.

The desk-sign-off scenario: publish the end-of-day banner only if every trade has settled.

```
Trades := [
trade_id	desk	status
1	FX	settled
```

```
| 2      | FX    | pending |
| 3      | RATES | settled |
];
```

```
Banner := [
message
Book is flat – signed off.
];
```

```
-- UNIT if every trade settled, EMPTY otherwise (a zero-column relation).
AllSettled := { ∀ : status = "settled" (Trades) };
```

```
-- One banner row when AllSettled is UNIT; no rows when it is EMPTY.
query { Banner × AllSettled };
```

Trade 2 is still pending, so `AllSettled` is `EMPTY` and the query returns no rows. Settle trade 2 and `AllSettled` becomes `UNIT`, so the same query returns the single banner row — the `×` did not have to change.

```
Trades → ∀ : status = "settled" → AllSettled (UNIT or EMPTY, 0 columns)
Banner —————→ × → 1 row, or none
```

Worked Example

The two zero-column relations, printed. This is worth seeing once, because a relation with no columns and one row does not look like anything else:

```
query { UNIT };
```

```
(1 row)
```

```
query { EMPTY };
```

```
(0 rows)
```

One row against no rows — with nothing in either row, since neither relation has a column to hold anything. That is the whole difference between them, and it is why they read as yes and no.

Crossing with them is where they earn their place. `UNIT` is the identity of `×`:

```
Orders := [
order_id	amount
100	120
101	80
];
```

```
query { Orders × UNIT };
```

```
order_id  amount
-----
      100      120
      101       80
(2 rows)
```

Unchanged — same rows, same heading, because concatenating a zero-column row adds nothing. `EMPTY` is the annihilator:

```
query { Orders × EMPTY };
```

```
order_id  amount
-----
(0 rows)
```

Every row of the left had to pair with some row of the right, and there were none. Note the heading survives even though no row does: `Orders × EMPTY` is an empty relation *of orders*, not a relation with no columns.

Limitations

Zero-column relations are not union-compatible with anything else, so `R ∪ UNIT` is a validation error unless `R` also has the empty heading. There is no glyph form. Neither literal pushes down to a SQL or MongoDB backend. `R × EMPTY` is **not** rewritten to `EMPTY` — it keeps `R`'s heading.

Because all four spellings are reserved in relation position, a relation actually named `unit`, `dee`, `empty`, or `dum` is only reachable through a [delimited identifier](#) — the declaration itself is accepted, but a bare reference reads as the literal.

Alternatives

A no-key `∀ : P (R)` produces these relations without naming them, and is the usual way one appears in a query. To express "no rows" with a specific heading, filter that relation with a false predicate rather than reaching for `EMPTY`.

See Also

[forall](#), [cross](#), [delimited-identifier](#), [inline-table](#)

Notes

The lineage names are Tutorial D's `TABLE_DEE` and `TABLE_DUM` (Date & Darwen, *The Third Manifesto*), named after Tweedledee and Tweedledum — a pair with no semantic cue as to which is which, which is exactly why Relix leads with `UNIT/EMPTY` and keeps `DEE/DUM` as aliases.

10. Built-in functions

Scalar functions usable in any expression (projection, selection, aggregate argument).

Len (string length)

Syntax

```
Len(<string>)  
  
π Len(name) → name_length (Users)
```

Description

Len returns the number of characters in a string — useful for validating input lengths, filtering by name size, or reporting field widths.

Technical Description

Len(s: STRING) → NUMBER. Returns the character count of s, counted in code points — so a character outside the basic multilingual plane, an emoji say, counts once. NULL input returns NULL. PURE and DETERMINISTIC, so the optimizer may fold it on constants.

Examples

Length of each name:

```
π name, Len(name) → chars (Users)
```

Filter to short codes:

```
σ Len(code) <= 5 (Products)
```

Find suspiciously long entries:

```
σ Len(description) > 1000 (Articles)
```

Pushdown

SQL: folds to `CHAR_LENGTH(<s>)` on the **MySQL**, **PostgreSQL** and **DuckDB** dialects, and to `LENGTH(<s>)` on **SQLite**, which lacks the name but counts the same way, and to `CHARACTER_LENGTH(<s>, CODEUNITS32)` on **Db2**, whose own functions count bytes unless told otherwise. Each counts a string in characters, which is what `relx` counts in, and each was put on that list by running it against text holding a character outside the basic multilingual plane. H2 (the generic dialect) counts UTF-16 code units, so its same-named function answers differently for such text, and is offered nothing.

Limitations

Operates on STRING values; a non-string argument is an evaluation error. NULL in → NULL out.

See Also

[left](#), [right](#), [mid](#), [instr](#), [trim](#)

UCase (upper-case)

Syntax

```
UCase(<string>)  
  
π UCase(code) → code_upper (Products)
```

Description

UCase converts a string to UPPER CASE. Use it to normalise text for display or to make comparisons case-insensitive by upper-casing both sides.

Technical Description

UCase(s: STRING) → STRING. Upper-cases s (Locale.ROOT). NULL input returns NULL. PURE, DETERMINISTIC, and IDEMPOTENT — applying it twice is the same as once, and the optimizer collapses nested calls.

Examples

Normalise country codes:

```
π UCase(country) → country (Customers)
```

Case-insensitive match:

```
σ UCase(name) = "ACME" (Suppliers)
```

Limitations

STRING only; NULL in → NULL out. Case folding follows Locale.ROOT (locale-neutral).

See Also

[Icase](#), [trim](#), [replace](#)

LCase (lower-case)

Syntax

```
LCase(<string>)  
  
π LCase(email) → email (Users)
```

Description

LCase converts a string to lower case. Commonly used to normalise emails, usernames, or tags so that comparisons ignore capitalisation.

Technical Description

LCase(s: STRING) → STRING. Lower-cases s (Locale.ROOT). NULL input returns NULL. PURE, DETERMINISTIC, and IDEMPOTENT.

Examples

Normalise emails to lower case:

```
π LCase(email) → email (Users)
```


Case-insensitive filter:

```
σ LCase(tag) = "urgent" (Tickets)
```

Limitations

STRING only; NULL in → NULL out. Case folding follows Locale.ROOT.

See Also

[ucase](#), [trim](#), [replace](#)

Trim (strip surrounding whitespace)

Syntax

```
Trim(<string>)
```

```
π Trim(name) → name (Users)
```

Description

Trim removes whitespace from both ends of a string. It is the standard cleanup for imported or user-entered data where stray leading/trailing spaces creep in.

Technical Description

Trim(s: STRING) → STRING. Strips leading and trailing whitespace (Java strip(), Unicode-aware). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Clean up a name column:

```
π Trim(name) → name (Customers)
```

Match ignoring stray spaces:

```
σ Trim(code) = "A100" (Parts)
```

Limitations

STRING only; NULL in → NULL out. Removes whitespace only — not other characters.

Alternatives

LTrim / RTrim to strip only one side. Replace to remove specific characters.

See Also

[ltrim](#), [rtrim](#), [replace](#)

LTrim (strip leading whitespace)

Syntax

```
LTrim(<string>)
```

```
π LTrim(line) → line (RawLines)
```

Description

LTrim removes whitespace from the start (left side) of a string, leaving any trailing spaces intact. Useful when only leading indentation needs removing.

Technical Description

LTrim(s: STRING) → STRING. Strips leading whitespace (Java stripLeading()). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Remove leading indentation:

```
π LTrim(line) → line (RawLines)
```

Limitations

STRING only; NULL in → NULL out. Leading whitespace only.

Alternatives

Trim for both ends; RTrim for the trailing side.

See Also

[trim](#), [rtrim](#)

RTrim (strip trailing whitespace)

Syntax

```
RTrim(<string>)
```

```
π RTrim(line) → line (RawLines)
```

Description

RTrim removes whitespace from the end (right side) of a string, leaving any leading spaces intact.

Technical Description

RTrim(s: STRING) → STRING. Strips trailing whitespace (Java stripTrailing()). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Remove trailing spaces from imported cells:

```
π RTrim(cell) → cell (Imported)
```

Limitations

STRING only; NULL in → NULL out. Trailing whitespace only.

Alternatives

Trim for both ends; LTrim for the leading side.

See Also

[trim](#), [ltrim](#)

Left (leftmost characters)

Syntax

```
Left(<string>, <n>)
```

```
π Left(phone, 3) → area_code (Contacts)
```

Description

Left returns the first *n* characters of a string. Use it to pull out prefixes — area codes, country prefixes, the first initial, a fixed-width code's leading segment.

Technical Description

Left(*s*: STRING, *n*: NUMBER) → STRING. Returns the first *n* characters (or the whole string if shorter). *n* must be non-negative (else an evaluation error). NULL string input returns NULL. PURE, DETERMINISTIC.

Examples

First three digits of a phone number:

```
π Left(phone, 3) → area_code (Contacts)
```

First initial:

```
π Left(first_name, 1) → initial (People)
```

Pushdown

SQL: folds to `LEFT(<s>, <n>)` on the **MySQL**, **PostgreSQL** and **DuckDB** dialects, and to `SUBSTR(<s>, 1, <n>)` on **SQLite**, which lacks the name but counts the same way, and to `SUBSTRING(<s>, 1, <n>, CODEUNITS32)` on **Db2**, whose own functions count bytes unless told otherwise. Each counts a string in characters, which is what `rel` counts in, and each was put on that list by running it against text holding a character outside the basic multilingual plane. H2 (the generic dialect) counts UTF-16 code units, so its same-named function answers differently for such text, and is offered nothing.

Limitations

n must be ≥ 0 . STRING first argument; NULL in → NULL out.

Alternatives

Right for the end of the string; Mid for a middle slice.

See Also

[right](#), [mid](#), [len](#), [instr](#)

Right (rightmost characters)

Syntax

```
Right(<string>, <n>)
```

```
π Right(card_number, 4) → last_four (Cards)
```

Description

Right returns the last *n* characters of a string. Common uses: the last four digits of a card or account number, a file extension, a trailing suffix.

Technical Description

Right(*s*: STRING, *n*: NUMBER) → STRING. Returns the last *n* characters (or the whole string if shorter). *n* must be non-negative. NULL string input returns NULL. PURE, DETERMINISTIC.

Examples

Last four digits of a card:

```
π Right(card_number, 4) → last_four (Cards)
```

Trailing year of a reference:

```
π Right(ref, 4) → year (Documents)
```

Pushdown

SQL: folds to `RIGHT(<s>, <n>)` on the **MySQL**, **PostgreSQL** and **DuckDB** dialects, and to `SUBSTR(<s>, MAX(LENGTH(<s>) - <n>, 0) + 1)` on **SQLite**, which lacks the name but counts the same way, and to a `SUBSTRING(..., CODEUNITS32)` on **Db2**, whose own functions count bytes unless told otherwise. Each counts a string in characters, which is what *relix* counts in, and each was put on that list by running it against text holding a character outside the basic multilingual plane. H2 (the generic dialect) counts UTF-16 code units, so its same-named function answers differently for such text, and is offered nothing.

Limitations

n must be ≥ 0 . STRING first argument; NULL in → NULL out.

Alternatives

Left for the start; Mid for a middle slice.

See Also

[left](#), [mid](#), [len](#), [instr](#)

Mid (substring)

Syntax

```
Mid(<string>, <start>)           -- from start to the end
Mid(<string>, <start>, <length>) -- length characters from start
```

```
π Mid(sku, 4, 3) → category_code (Products)
```

Description

Mid extracts a slice from the middle of a string. Give it a starting position (counting from 1) and optionally how many characters to take. Use it to pull a field out of a fixed-format code or to grab a known section of text.

Technical Description

Mid(s, start[, length]) → STRING. start is 1-based; the two-argument form returns from start to the end, the three-argument form returns at most length characters. start must be ≥ 1 and length ≥ 0 (else an evaluation error). Positions and lengths are counted in code points, so a slice never cuts a character in half. Out-of-range slices clamp to the string. NULL string input returns NULL. PURE, DETERMINISTIC.

Examples

A 3-character category code starting at position 4:

```
π Mid(sku, 4, 3) → category_code (Products)
```

Everything from the 5th character on:

```
π Mid(reference, 5) → suffix (Documents)
```

Pushdown

SQL: folds to SUBSTRING(<s>, <start>[, <length>]) on the **MySQL**, **PostgreSQL** and **DuckDB** dialects, and to SUBSTR(...) on **SQLite**, which lacks the name but counts the same way, and to SUBSTRING(..., CODEUNITS32) on **Db2**, whose own functions count bytes unless told otherwise. Each counts a string in characters, which is what relix counts in, and each was put on that list by running it against text holding a character outside the basic multilingual plane. H2 (the generic dialect) counts UTF-16 code units, so its same-named function answers differently for such text, and is offered nothing.

Limitations

start is 1-based and must be ≥ 1 ; length must be ≥ 0 . STRING first argument; NULL in → NULL out.

Alternatives

Left / Right for prefixes and suffixes; InStr to locate a position first.

See Also

[left](#), [right](#), [instr](#), [len](#)

InStr (find substring position)

Syntax

```
InStr(<string>, <find>)
```

```
InStr(<start>, <string>, <find>)
```

```
π InStr(email, "@") → at_pos (Users)
```

Description

InStr finds where one string appears inside another and returns its position (counting from 1), or 0 if it is not found. Use it to locate a delimiter — the @ in an email, a separator in a code — often as the input to Mid/Left/Right.

Technical Description

InStr(s, find) → NUMBER returns the 1-based index of the first occurrence of find in s, or 0 if absent. The three-argument form InStr(start, s, find) begins searching at the 1-based start position. NULL arguments return NULL. PURE, DETERMINISTIC.

Examples

Position of the @ in an email:

```
π InStr(email, "@") → at_pos (Users)
```

Split a "name:value" pair — get the name before the colon:

```
π Left(pair, InStr(pair, ":") - 1) → name (Pairs)
```

Find a later occurrence (search from position 5):

```
π InStr(5, path, "/") → next_slash (Paths)
```

Limitations

Returns 0 (not NULL) when the substring is absent — guard for 0 before using it as an index. STRING arguments; a NULL argument returns NULL.

Alternatives

IsNumeric / pattern checks for validation; Replace to substitute rather than locate.

See Also

[mid](#), [left](#), [right](#), [replace](#), [len](#)

Chr (character from code)

Syntax

```
Chr(<code>)
```

```
π Chr(65) → letter (Codes)      -- "A"
```

Description

Chr turns a numeric character code into the corresponding single character — for example 65 becomes "A". It is the inverse of Asc, and is handy for building characters programmatically (e.g. a tab or newline by code).

Technical Description

Chr(n: NUMBER) → STRING. Returns the single character whose code point is n. Requires 0 ≤ n ≤ 1114111 (U+10FFFF, the last code point), and rejects a surrogate (0xD800-0xDFFF), which is half of a character rather than one. Else an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Code to letter:

```
π Chr(65) → letter (Codes)      -- "A"
```

Build a delimiter character:

```
π Chr(9) → tab (Config)         -- tab character
```

Limitations

The code must be in 0-65535. NUMBER argument; NULL in → NULL out.

Alternatives

Asc converts the other way (character → code).

See Also

[asc](#), [replace](#)

Asc (code of first character)

Syntax

```
Asc(<string>)  
π Asc(letter) → code (Letters)      -- "A" → 65
```

Description

Asc returns the numeric character code of the first character of a string — for example "A" becomes 65. It is the inverse of Chr, useful for sorting by code or for character-level computation.

Technical Description

Asc(s: STRING) → NUMBER. Returns the code point of s's first character. An empty string is an evaluation error; NULL input returns NULL. PURE, DETERMINISTIC.

Examples

First character's code:

```
π Asc(letter) → code (Letters)
```

Bucket by leading-letter code:

```
σ Asc(name) >= 65 ∧ Asc(name) <= 90 (People)  -- starts A-Z
```

Limitations

Errors on an empty string. STRING argument; NULL in → NULL out. Reads only the first character.

Alternatives

Chr converts the other way (code → character).

See Also

[chr](#), [left](#), [len](#)

Replace (substitute text)

Syntax

```
Replace(<string>, <find>, <replacement>)  
  
π Replace(phone, "-", "") → digits (Contacts)
```

Description

Replace swaps every occurrence of one piece of text for another. Use it to strip characters (replace with an empty string), reformat values, or fix known typos in bulk.

Technical Description

Replace(s, find, replacement) → STRING. Replaces all non-overlapping occurrences of the literal find within s with replacement (Java String.replace — literal, not regex). NULL string input returns NULL. PURE, DETERMINISTIC.

Examples

Strip dashes from phone numbers:

```
π Replace(phone, "-", "") → digits (Contacts)
```

Normalise a separator:

```
π Replace(path, "\\ ", "/") → path (Files)
```

Redact a token:

```
π Replace(note, "SECRET", "[redacted]") → note (Logs)
```

Pushdown

SQL: folds to `REPLACE(<s>, <find>, <replacement>)` on all dialects. It is the one string built-in that does. Most SQL backends' REPLACE searches case-sensitively whatever the column's collation, which is what the engine does; SQL Server's searches under the collation, whose default ignores case, so there the string is handed over under a binary one: `REPLACE((<s>) COLLATE Latin1_General_100_BIN2, ...)`. The rest of the string library is evaluated in-engine because SQL's same-named function answers differently — see [pushdown](#).

Limitations

Matching is literal text, not a regular expression. STRING arguments; a NULL string returns NULL.

Alternatives

Trim/LTrim/RTrim to remove only surrounding whitespace; Mid/Left/Right to extract rather than substitute.

See Also

[trim](#), [mid](#), [instr](#), [ucase](#), [lcase](#)

Abs (absolute value)

Syntax

```
Abs(<number>)
```

```
π Abs(delta) → magnitude (Readings)
```

Description

Abs returns the size of a number ignoring its sign — -7 and 7 both become 7 . Use it to measure how far a value is from zero regardless of direction (e.g. the size of an error or a price change).

Technical Description

Abs(x : NUMBER) → NUMBER. Returns $|x|$ with full BigDecimal precision. NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Magnitude of a signed delta:

```
π Abs(delta) → magnitude (Readings)
```

Filter on absolute deviation:

```
σ Abs(forecast - actual) > 100 (Forecasts)
```

Smallest absolute error per sensor:

```
γ sensor, MIN(Abs(error)) → best (Readings)
```

Pushdown

SQL: folds to `ABS(<e>)` on all dialects. The result is exact on an exact numeric column, so the database's answer and the engine's are the same digits.

Limitations

NUMBER only; NULL in → NULL out.

See Also

[sgn](#), [round](#), [int](#), [fix](#)

Int (floor to integer)

Syntax

```
Int(<number>)
```

```
π Int(price) → whole_pounds (Products)
```

Description

Int rounds a number DOWN to the nearest whole number (toward negative infinity). 3.9 becomes 3, and -1.1 becomes -2 . Use it when you want the integer floor of a value.

Technical Description

Int(x : NUMBER) $\rightarrow$ NUMBER. Rounds toward negative infinity (RoundingMode.FLOOR — Jet behaviour, same as mathematical floor). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Whole-pound part of a price:

```
π Int(price) → whole_pounds (Products)
```

Bucket ages into whole years (floor):

```
π Int(age) → year_bucket (People)
```

Pushdown

SQL: folds to `FLOOR(<e>)` on every dialect but SQLite, where it is a floating-point function and an optional part of the build. Exact on an exact numeric column.

Limitations

NUMBER only; NULL in $\rightarrow$ NULL out. Floors negatives downward (Int(-1.1) = -2) — differs from Fix, which truncates toward zero.

Alternatives

Fix truncates toward zero; Ceil rounds up; Round rounds to nearest.

See Also

[fix](#), [ceil](#), [round](#), [abs](#)

Fix (truncate toward zero)

Syntax

```
Fix(<number>)
```

```
π Fix(value) → truncated (Readings)
```

Description

Fix removes the fractional part of a number, rounding toward zero. 3.9 becomes 3 and -3.9 becomes -3 (it just drops the decimals). Use it when you want to discard the fraction without changing the sign's direction.

Technical Description

Fix(x : NUMBER) $\rightarrow$ NUMBER. Truncates toward zero (RoundingMode.DOWN). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Drop the fraction:

```
π Fix(value) → whole (Readings)
```

Whole hours from a fractional hours figure:

```
π Fix(hours) → whole_hours (Timesheets)
```

Pushdown

SQL: dialect-dependent, because truncation towards zero is the one rounding SQL never settled on a single name for.

- **MySQL:** `TRUNCATE(<e>, 0)`.
- **Postgres, DuckDB and Db2:** `TRUNC(<e>)`.
- **SQL Server:** `ROUND(<e>, 0, 1)`, whose third argument truncates.
- **SQLite:** not pushed. Its `TRUNC` is a floating-point function behind a compile-time option.
- **GENERIC:** not pushed. A name that has to be chosen per dialect is one an unidentified backend has not been confirmed to have.

Limitations

NUMBER only; NULL in → NULL out. Differs from Int on negatives — $\text{Fix}(-3.9) = -3$, $\text{Int}(-3.9) = -4$.

Alternatives

Int floors toward negative infinity; Ceil rounds up; Round rounds to nearest.

See Also

[int](#), [ceil](#), [round](#)

Ceil (round up)

Syntax

```
Ceil(<number>)
```

```
π Ceil(items / 10) → pages (Orders)
```

Description

Ceil rounds a number UP to the nearest whole number. 3.1 becomes 4. Use it for "how many full containers do I need" calculations — pages, boxes, batches — where any remainder needs a whole extra unit.

Technical Description

`Ceil(x: NUMBER) → NUMBER`. Rounds toward positive infinity (`RoundingMode.CEILING`). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Pages needed at 10 items per page:

```
π Ceil(item_count / 10) → pages (Orders)
```

Boxes needed (round any remainder up):

```
π Ceil(units / box_size) → boxes (Shipments)
```

Pushdown

SQL: folds to `CEILING(<e>)` on every dialect but SQLite — the SQL standard's spelling. Exact on an exact numeric column. On SQLite it is a floating-point function and an optional part of the build, so the call runs in the engine.

Limitations

NUMBER only; NULL in → NULL out. Always rounds upward, including for negatives (`Ceil(-1.5) = -1`).

Alternatives

Int / Fix round down/toward zero; Round rounds to nearest.

See Also

[int](#), [fix](#), [round](#)

Round (round to nearest)

Syntax

```
Round(<number>)
```

```
Round(<number>, <places>)
```

```
π Round(price, 2) → price (Products)
```

Description

Round rounds a number to the nearest whole number, or to a given number of decimal places. With no places it rounds to an integer; with 2 it rounds to the nearest penny/cent. Half-way values round up.

Technical Description

`Round(x[, places])` → NUMBER. Rounds to `places` decimal places (0 if omitted) using `RoundingMode.HALF_UP`. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Round money to two decimals:

```
π Round(price, 2) → price (Products)
```

Round to the nearest whole number:

```
π Round(score) → grade (Results)
```

Round to the nearest hundred (negative places not supported — scale instead):

```
π Round(amount, 0) → whole (Orders)
```

Pushdown

SQL: folds to `ROUND(<e>)` and `ROUND(<e>, <places>)` on every dialect but SQLite — on SQL Server, whose `ROUND` always takes its places, as `ROUND(<e>, 0)`. Both round half away from zero over an exact numeric column, which is what the engine does. SQLite has no decimal type and rounds the nearest double instead — `-49.555` rounds to `-49.55` there and `-49.56` here — so on SQLite the call runs in the engine.

Limitations

NUMBER first argument; NULL in → NULL out. Uses `HALF_UP` (0.5 rounds away from zero). `places` is the number of decimal places (≥ 0 in practice).

Alternatives

`Int` / `Fix` / `Ceil` for directional rounding rather than nearest.

See Also

[int](#), [fix](#), [ceil](#)

Sgn (sign)

Syntax

```
Sgn(<number>)
```

```
π Sgn(change) → direction (Prices)
```

Description

`Sgn` reports the sign of a number: `-1` for negatives, `0` for zero, and `1` for positives. Use it to classify values by direction — up/down/flat, gain/loss — in a single tidy number.

Technical Description

`Sgn(x: NUMBER) → NUMBER`. Returns the signum of `x` (`-1`, `0`, or `1`). NULL input returns NULL. PURE, DETERMINISTIC, IDEMPOTENT.

Examples

Direction of a price change:

```
π Sgn(close - open) → direction (Prices)
```

Count gainers vs losers:

```
γ Sgn(change) → dir, COUNT(id) → n (Stocks)
```

Pushdown

SQL: folds to `SIGN(<e>)` on all dialects.

Limitations

NUMBER only; NULL in → NULL out.

See Also

[abs](#), [round](#)

Sqr (square root)

Syntax

```
Sqr(<number>)
```

```
π Sqr(area) → side (Squares)
```

Description

Sqr returns the square root of a number — the value that, multiplied by itself, gives the input. Use it in geometric and statistical calculations (e.g. recovering a side length from an area, or a standard deviation from a variance).

Technical Description

Sqr(x: NUMBER) → NUMBER. Returns $\sqrt{x}$ (Math.sqrt). The argument must be non-negative (else an evaluation error). NULL input returns NULL. PURE, DETERMINISTIC. Result is a floating-point approximation.

Examples

Side length from an area:

```
π Sqr(area) → side (Squares)
```

Standard deviation from a variance column:

```
π Sqr(variance) → std_dev (Stats)
```

Limitations

Errors on a negative argument. NUMBER only; NULL in → NULL out. Floating-point result, not exact.

Alternatives

Power(x, 0.5) is equivalent; Power(x, 2) squares.

See Also

[power](#), [exp](#), [log](#)

Log (natural logarithm)

Syntax

```
Log(<number>)
```

```
π Log(value) → log_value (Measurements)
```

Description

Log returns the natural logarithm (base e) of a number. It is used to compress wide-ranging values onto a smaller scale and in growth/decay and statistical calculations.

Technical Description

Log(x: NUMBER) → NUMBER. Returns $\ln(x)$ (Math.log). The argument must be positive (else an evaluation error). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Log-transform a measurement:

```
π Log(value) → log_value (Measurements)
```

Filter on a log-scaled threshold:

```
σ Log(amount) > 10 (Transactions)
```

Limitations

Errors on a non-positive argument ($x \leq 0$). NUMBER only; NULL in → NULL out. This is the natural log (base e); divide by Log(base) for other bases.

Alternatives

Exp is the inverse (e^x). For log base 10, use $\text{Log}(x) / \text{Log}(10)$.

See Also

[exp](#), [power](#), [sqr](#)

Exp (exponential)

Syntax

```
Exp(<number>)
```

```
π Exp(rate * years) → growth_factor (Investments)
```

Description

Exp returns e raised to the power of the input ($e \approx 2.718$). It is the inverse of Log and appears in compound growth, decay, and probability calculations.

Technical Description

Exp(x: NUMBER) → NUMBER. Returns e^x (Math.exp). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Continuous-growth factor:

```
π Exp(rate * years) → growth_factor (Investments)
```

Undo a natural log:

```
π Exp(log_value) → original (Measurements)
```

Limitations

NUMBER only; NULL in → NULL out. Floating-point result; very large inputs overflow to infinity.

Alternatives

Log is the inverse; Power(base, exp) for a base other than e.

See Also

[log](#), [power](#), [sqr](#)

Power (raise to a power)

Syntax

```
Power(<base>, <exponent>)
```

```
π Power(2, n) → capacity (Levels)
```

Description

Power raises a base to an exponent — Power(2, 10) is 1024. Use it for exponential quantities (doubling, compound factors) and for roots (a fractional exponent like 0.5 is a square root).

Technical Description

Power(base: NUMBER, exp: NUMBER) → NUMBER. Returns base^{exp} (Math.pow). A NULL in either argument returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Powers of two:

```
π Power(2, level) → capacity (Levels)
```

Compound factor over n periods:

```
π Power(1 + rate, periods) → factor (Loans)
```

Cube root via fractional exponent:

```
π Power(volume, 0.333333) → side (Cubes)
```

Limitations

NUMBER arguments; NULL in either → NULL out. Floating-point result. Some base/exponent combinations (e.g. negative base with a fractional exponent) yield NaN.

Alternatives

Sqr for the square root specifically; Exp/Log for base-e work.

See Also

[sqr](#), [exp](#), [log](#)

Sin (sine)

Syntax

```
Sin(<radians>)
```

```
 $\pi$  Sin(angle) → s (Angles)
```

Description

Sin returns the sine of an angle given in radians. It is used in geometry, signal, and wave calculations. (Multiply degrees by $\pi/180$ to convert to radians first.)

Technical Description

Sin(x: NUMBER) → NUMBER. Returns sin(x) where x is in radians (Math.sin). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Resolve a wind reading into its east/west component:

```
 $\pi$  station, speed * Sin(direction_rad) → east (Wind)
```

From degrees:

```
 $\pi$  Sin(degrees * 3.14159265 / 180) → s (Angles)
```

Limitations

Argument is in radians, not degrees. NUMBER only; NULL in → NULL out. Floating point.

See Also

[cos](#), [tan](#), [atn](#)

Cos (cosine)

Syntax

```
Cos(<radians>)
```

```
 $\pi$  Cos(angle) → c (Angles)
```

Description

Cos returns the cosine of an angle given in radians, used in geometry and wave calculations. (Multiply degrees by $\pi/180$ to convert to radians first.)

Technical Description

Cos(x: NUMBER) → NUMBER. Returns cos(x) where x is in radians (Math.cos). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Resolve a wind reading into its north/south component:

```
 $\pi$  station, speed * Cos(direction_rad) → north (Wind)
```

Distance along the ground from a slant range and angle:

```
π id, slant_range * Cos(elevation_rad) → ground_distance (Sightings)
```

Limitations

Argument is in radians, not degrees. NUMBER only; NULL in → NULL out. Floating point.

See Also

[sin](#), [tan](#), [atn](#)

Tan (tangent)

Syntax

```
Tan(<radians>)
```

```
π Tan(angle) → t (Angles)
```

Description

Tan returns the tangent of an angle given in radians, used in geometry and slope calculations. (Multiply degrees by $\pi/180$ to convert to radians first.)

Technical Description

Tan(x: NUMBER) → NUMBER. Returns tan(x) where x is in radians (Math.tan). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Height of an object from its angle of elevation and distance:

```
π id, distance * Tan(elevation_rad) → height (Sightings)
```

Gradient of a ramp as a rise over a fixed run:

```
π id, Tan(slope_rad) * run → rise (Ramps)
```

Limitations

Argument is in radians. NUMBER only; NULL in → NULL out. Near odd multiples of $\pi/2$ the result is extremely large (asymptote). Floating point.

See Also

[sin](#), [cos](#), [atn](#)

Atn (arctangent)

Syntax

```
Atn(<number>)
```

```
π Atn(slope) → angle_rad (Lines)
```

Description

Atn returns the arctangent — the angle (in radians) whose tangent is the input. It is the inverse of Tan, useful for recovering an angle from a slope or ratio.

Technical Description

Atn(x: NUMBER) → NUMBER. Returns atan(x) in radians, in the range $(-\pi/2, \pi/2)$ (Math.atan). NULL input returns NULL. PURE, DETERMINISTIC. Floating-point result.

Examples

Angle of a slope, in radians:

```
π Atn(rise / run) → angle_rad (Lines)
```

Convert the result to degrees:

```
π Atn(slope) * 180 / 3.14159265 → angle_deg (Lines)
```

Limitations

Returns radians in $(-\pi/2, \pi/2)$. NUMBER only; NULL in → NULL out. Single-argument form only (no two-argument atan2). Floating point.

Alternatives

Tan is the forward direction (angle → ratio).

See Also

[tan](#), [sin](#), [cos](#)

Rand (random number)

Syntax

```
Rand()
```

```
π id, Rand() → r (Users)
```

Description

Rand returns a fresh random number between 0 (inclusive) and 1 (exclusive) each time it is called. Use it to shuffle, assign random buckets, or sample rows — `Rand() < 0.1` keeps roughly 10% of rows.

Technical Description

Rand() → NUMBER in [0, 1). Zero-argument. It reads the system RNG, so it is neither PURE nor DETERMINISTIC — the optimizer never folds or dedupes it, and it produces a new value per evaluation. No seed (not reproducible). Underpins the SAMPLE p operator (which desugars to `σ Rand() < p`).

Examples

Randomly keep about 10% of rows:

```
σ Rand() < 0.1 (Events)
```

Assign a random sort key (shuffle):

```
τ r ASC (π id, Rand() → r (Deck))
```

Random bucket 0–9:

```
π id, Int(Rand() * 10) → bucket (Users)
```

Limitations

Non-deterministic and unseeded — results differ every run and are not reproducible. Because it is impure it is never constant-folded.

Alternatives

SAMPLE p for probabilistic sampling and SAMPLE n ROWS for an exact-count sample express the common cases more directly.

See Also

[sample-bernoulli](#), [sample-reservoir](#), [int](#)

Notes

Use Rand() with τ for a random shuffle, or with σ for ad-hoc sampling.

NOW (current timestamp)

Syntax

```
NOW()
```

```
π id, NOW() → fetched_at (Records)
```

Description

NOW returns the current moment as a `TIMESTAMP` (in UTC). Use it to stamp results with the time they were produced, or to compute how long ago something happened relative to right now.

Technical Description

NOW() → `TIMESTAMP`. Returns the current instant (`Instant.now()`, UTC). Zero-argument. It reads system state, so it is neither `PURE` nor `DETERMINISTIC` — the optimizer never folds or dedupes it, and every evaluation reflects the call time.

Examples

Stamp each row with the fetch time:

```
π id, NOW() → fetched_at (Records)
```

Records from the last hour:

```
σ created_at >= (NOW() - DURATION 'PT1H') (Events)
```

Age of each ticket:

```
π id, (NOW() - opened_at) → age (Tickets)
```

Pushdown

The **call** is never handed to a backend — a database asked to evaluate `NOW()` answers from its own clock, and the session's clock is what decides what "now" means here. The **value** is: the engine evaluates `NOW()` once for the run and sends the resulting `TIMESTAMP` literal, which a backend has no opinion about. So `σ at < NOW() (conn.events)` folds into a `WHERE` and filters at the source, with the session's instant in it.

Every current-time call in one run reads the same moment. That is what makes the substitution honest rather than convenient: the value sent to the database is the same value the rest of the query computes here, so a query half folded and half not still compares against one instant.

Reproducible runs (pinned clock)

By default `NOW()` reads the wall clock, so a query with a relative window returns different rows each time it runs — which makes a result impossible to reproduce later. Pass `relix --now <ISO-8601 UTC instant>` (or set `RELIX_NOW`) to pin `NOW()`, `CURRENT_DATE()` and `CURRENT_TIME()` to a fixed instant for the whole run:

```
relix --exec --now 2026-07-22T12:00:00Z report.relax
```

Every evaluation in that run then sees the same instant, so replaying the script over the same data yields byte-identical results however much later it runs. The functions stay non-PURE and non-DETERMINISTIC regardless — under the default clock they still advance, so the optimizer must never fold or dedupe them.

Limitations

Non-deterministic (reads the clock); not folded by the optimizer. Always UTC.

Alternatives

`CURRENT_DATE` for today's date only; `CURRENT_TIME` for the clock time only.

See Also

[current_date](#), [current_time](#), [timestamp-literal](#), [date_trunc](#)

CURRENT_DATE (today)

Syntax

```
CURRENT_DATE()
```

```
σ due_date = CURRENT_DATE() (Invoices)
```

Description

`CURRENT_DATE` returns today's date (in UTC) as a `DATE` — no time of day. Use it to find records due today, compute days remaining, or filter to the current day.

Technical Description

`CURRENT_DATE()` → `DATE`. Returns `LocalDate.now(UTC)`. Zero-argument. Reads system state, so it is neither PURE nor DETERMINISTIC and is never folded by the optimizer.

Examples

Invoices due today:

```
σ due_date = CURRENT_DATE() (Invoices)
```

Days until a deadline:

```
π id, (deadline - CURRENT_DATE()) → days_left (Tasks)
```

Records on or after today:

```
σ event_date >= CURRENT_DATE() (Calendar)
```

Pushdown

The **call** is never handed to a backend — a database would answer from its own clock, and the session's clock is what decides what "now" means here. The **value** is: the engine evaluates it once for the run and sends the resulting literal, so an operator containing it still folds. Every current-time call in one run reads the same moment, so the folded and unfolded halves of a query agree.

Reproducible runs (pinned clock)

By default NOW() reads the wall clock, so a query with a relative window returns different rows each time it runs — which makes a result impossible to reproduce later. Pass `relix --now <ISO-8601 UTC instant>` (or set RELIX_NOW) to pin NOW(), CURRENT_DATE() and CURRENT_TIME() to a fixed instant for the whole run:

```
relix --exec --now 2026-07-22T12:00:00Z report.relax
```

Every evaluation in that run then sees the same instant, so replaying the script over the same data yields byte-identical results however much later it runs. The functions stay non-PURE and non-DETERMINISTIC regardless — under the default clock they still advance, so the optimizer must never fold or dedupe them.

Limitations

Non-deterministic (reads the clock); not folded. UTC-based, so "today" is the UTC day.

Alternatives

NOW() for a full timestamp; CURRENT_TIME() for the clock time only.

See Also

[now](#), [current_time](#), [date-literal](#), [to_date](#)

CURRENT_TIME (current clock time)

Syntax

```
CURRENT_TIME()
```

```
π id, CURRENT_TIME() → at (Hits)
```

Description

CURRENT_TIME returns the current time of day (in UTC) as a TIME — no date. Use it when only the clock time matters, such as comparing against business hours.

Technical Description

CURRENT_TIME() → TIME. Returns LocalTime.now(UTC). Zero-argument. Reads system state, so it is neither PURE nor DETERMINISTIC and is never folded.

Examples

Stamp the clock time:

```
π id, CURRENT_TIME() → at (Hits)
```

Within business hours right now:

```
σ CURRENT_TIME() >= TIME '09:00:00' ∧ CURRENT_TIME() < TIME '17:00:00' (Flags)
```

Pushdown

The **call** is never handed to a backend — a database would answer from its own clock, and the session's clock is what decides what "now" means here. The **value** is: the engine evaluates it once for the run and sends the resulting literal, so an operator containing it still folds. Every current-time call in one run reads the same moment, so the folded and unfolded halves of a query agree.

Reproducible runs (pinned clock)

By default NOW() reads the wall clock, so a query with a relative window returns different rows each time it runs — which makes a result impossible to reproduce later. Pass `relix --now <ISO-8601 UTC instant>` (or set RELIX_NOW) to pin NOW(), CURRENT_DATE() and CURRENT_TIME() to a fixed instant for the whole run:

```
relix --exec --now 2026-07-22T12:00:00Z report.relax
```

Every evaluation in that run then sees the same instant, so replaying the script over the same data yields byte-identical results however much later it runs. The functions stay non-PURE and non-DETERMINISTIC regardless — under the default clock they still advance, so the optimizer must never fold or dedupe them.

Limitations

Non-deterministic (reads the clock); not folded. UTC-based.

Alternatives

NOW() for a full timestamp; CURRENT_DATE() for today's date only.

See Also

[now](#), [current_date](#), [time-literal](#), [to_time](#)

YEAR (extract year)

Syntax

```
YEAR(<date-or-timestamp>)
```

```
γ YEAR(order_date) → yr, SUM(amount) → revenue (Orders)
```

Description

YEAR pulls the year number out of a date or timestamp. Use it to group or filter by year — annual totals, "everything from 2026", year-over-year comparisons.

Technical Description

YEAR(d) → NUMBER. Accepts a DATE or TIMESTAMP (a TIMESTAMP is read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Annual revenue:

```
γ YEAR(order_date) → yr, SUM(amount) → revenue (Orders)
```

Filter to one year:

```
σ YEAR(created_at) = 2026 (Accounts)
```

Pushdown

SQL: folds to `EXTRACT(YEAR FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(YEAR, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$year": "$<col>"}` inside a `$project` pipeline stage when the argument is an attribute and an alias is supplied (e.g. `π YEAR(at) → yr (Events)`).

Limitations

Argument must be a DATE or TIMESTAMP. TIMESTAMP is interpreted at UTC. NULL in → NULL out.

See Also

[month](#), [day](#), [date_trunc](#), [hour](#)

MONTH (extract month)

Syntax

```
MONTH(<date-or-timestamp>)
```

```
γ MONTH(order_date) → mo, SUM(amount) → revenue (Orders)
```

Description

MONTH pulls the month number (1-12) out of a date or timestamp. Use it to group or filter by month — monthly reports, seasonal patterns, "everything in December".

Technical Description

MONTH(d) → NUMBER (1-12). Accepts a DATE or TIMESTAMP (read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Monthly revenue within a year:

```
γ MONTH(order_date) → mo, SUM(amount) → revenue
  (σ YEAR(order_date) = 2026 (Orders))
```

December records:

```
σ MONTH(event_date) = 12 (Calendar)
```

Pushdown

SQL: folds to `EXTRACT(MONTH FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(MONTH, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$month": "$<col>"}` inside a `$project` stage (requires alias).

Limitations

Argument must be a DATE or TIMESTAMP. Returns 1-12. TIMESTAMP at UTC. NULL in → NULL out.

See Also

[year](#), [day](#), [date_trunc](#)

DAY (extract day of month)

Syntax

```
DAY(<date-or-timestamp>)
```

```
σ DAY(invoice_date) = 1 (Invoices)
```

Description

DAY pulls the day-of-month (1-31) out of a date or timestamp. Use it for day-level filters — "invoices dated the 1st", end-of-month checks, day buckets.

Technical Description

DAY(d) → NUMBER (1-31). Accepts a DATE or TIMESTAMP (read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

First-of-the-month invoices:

```
σ DAY(invoice_date) = 1 (Invoices)
```

Pushdown

SQL: folds to `EXTRACT(DAY FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(DAY, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$dayOfMonth": "$<col>"}` inside a `$project` stage (requires alias). Note: MongoDB uses `$dayOfMonth`, not `$day`.

Limitations

Argument must be a DATE or TIMESTAMP. Returns the day of month (1–31), not the day of week. TIMESTAMP at UTC. NULL in → NULL out.

See Also

[year](#), [month](#), [date_trunc](#)

HOUR (extract hour)

Syntax

```
HOUR(<time-or-timestamp>)
```

```
γ HOUR(occurred_at) → hr, COUNT(id) → hits (Events)
```

Description

HOUR pulls the hour-of-day (0–23) out of a time or timestamp. Use it to bucket activity by hour — traffic by time of day, peak-hour analysis.

Technical Description

HOUR(t) → NUMBER (0–23). Accepts a TIME or TIMESTAMP (a TIMESTAMP is read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Hits per hour of day:

```
γ HOUR(occurred_at) → hr, COUNT(id) → hits (Events)
```

Overnight records:

```
σ HOUR(logged_at) < 6 (Logs)
```

Pushdown

SQL: folds to `EXTRACT(HOUR FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(HOUR, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$hour": "$<col>"}` inside a `$project` stage (requires alias).

Limitations

Argument must be a TIME or TIMESTAMP. Returns 0–23. TIMESTAMP at UTC. NULL in → NULL out.

See Also

[minute](#), [second](#), [date_trunc](#), [year](#)

MINUTE (extract minute)

Syntax

```
MINUTE(<time-or-timestamp>)  
  
π id, MINUTE(start_time) → min (Slots)
```

Description

MINUTE pulls the minutes part (0-59) out of a time or timestamp. Useful for fine-grained time bucketing or schedule alignment.

Technical Description

MINUTE(t) → NUMBER (0-59). Accepts a TIME or TIMESTAMP (read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Minutes part of a slot start:

```
π id, MINUTE(start_time) → min (Slots)
```

On-the-hour events:

```
σ MINUTE(occurred_at) = 0 (Events)
```

Pushdown

SQL: folds to `EXTRACT(MINUTE FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(MINUTE, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$minute": "$<col>"}` inside a `$project` stage (requires alias).

Limitations

Argument must be a TIME or TIMESTAMP. Returns 0-59. TIMESTAMP at UTC. NULL in → NULL out.

See Also

[hour](#), [second](#), [date_trunc](#)

SECOND (extract second)

Syntax

```
SECOND(<time-or-timestamp>)  
  
π id, SECOND(captured_at) → sec (Samples)
```

Description

SECOND pulls the seconds part (0-59) out of a time or timestamp — the finest whole-second component.

Technical Description

SECOND(t) → NUMBER (0-59). Accepts a TIME or TIMESTAMP (read at UTC); other types are an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Readings captured exactly on the minute (a sign of a scheduled, not sampled, feed):

```
σ SECOND(captured_at) = 0 (Samples)
```

Spot a burst — how many events landed in each second of a minute:

```
γ SECOND(occurred_at) → sec, COUNT(id) → hits (Events)
```

Pushdown

SQL: folds to `EXTRACT(SECOND FROM <col>)` on every dialect but SQLite, which has no date type to extract from. On SQL Server, which has no `EXTRACT`, it is `DATEPART(SECOND, SWITCHOFFSET(<col>, '+00:00'))` — the value switched to UTC first, so a `DATETIMEOFFSET` is read at UTC as the engine reads it. MongoDB: folds to `{"$second": "$<col>"}` inside a `$project` stage (requires alias).

Limitations

Argument must be a TIME or TIMESTAMP. Returns whole seconds 0-59 (no fractional part). TIMESTAMP at UTC. NULL in → NULL out.

See Also

[hour](#), [minute](#), [date_trunc](#)

DATE_TRUNC (truncate timestamp to a unit)

Syntax

```
DATE_TRUNC(<unit>, <timestamp>)
```

```
γ DATE_TRUNC('day', occurred_at) → day, COUNT(id) → hits (Events)
```

Description

DATE_TRUNC rounds a timestamp DOWN to the start of a chosen unit — the start of the hour, day, month, or year. It is the time-series workhorse: truncate every event to its day (or hour) and group, and you get a clean per-day (or per-hour) series.

Technical Description

DATE_TRUNC(unit: STRING, ts: TIMESTAMP) → TIMESTAMP. Truncates ts (at UTC) to the start of `unit` ∈ {year, month, day, hour, minute, second}; an unknown unit or a non-TIMESTAMP value is an evaluation error. NULL ts returns NULL. PURE, DETERMINISTIC.

Examples

Daily event counts:

```
γ DATE_TRUNC('day', occurred_at) → day, COUNT(id) → hits (Events)
```

Hourly revenue series:

```
γ DATE_TRUNC('hour', sold_at) → hr, SUM(amount) → revenue (Sales)
```

Monthly buckets:

```
π id, DATE_TRUNC('month', created_at) → month (Accounts)
```

Limitations

The value must be a `TIMESTAMP` (not `DATE/TIME`). Units are year/month/day/hour/minute/second only. Truncation is at UTC. `NULL` in $\rightarrow$ `NULL` out.

Pushdown

SQL: dialect-dependent, and the one built-in whose spelling differs in shape rather than in name.

- **Postgres, DuckDB and Db2:** `date_trunc('<unit>', <col>)` — standard Postgres form, which DuckDB and Db2 share.
- **MySQL:** `CAST(DATE_FORMAT(<col>, '<pattern>') AS DATETIME)`, with one pattern per unit — `'%Y-01-01 00:00:00'` for a year, `'%Y-%m-%d %H:00:00'` for an hour, and so on. MySQL and MariaDB have no `DATE_TRUNC` function; `DATE_FORMAT` reaches the same value by writing out the parts to keep and zeroing the rest, and the cast is what makes the result a `TIMESTAMP` rather than the string `DATE_FORMAT` returns. The unit must be a literal: a call whose unit is a column or an expression is evaluated in-engine, since the unit is what chooses the pattern.
- **SQL Server:** not pushed. Its `DATETRUNC` arrived in SQL Server 2022, and a declared dialect cannot confirm the server is that new.
- **SQLite:** not pushed. SQLite has no date type, and `strftime` reads whatever text a column holds.
- **GENERIC:** not pushed; the call is evaluated in-engine, and everything above the operator containing it stays in the engine with it. The generic dialect is an unidentified backend, so it is offered no spelling that has not been confirmed against it.

MongoDB: folds to `{"dateTrunc": {"date": <tsExpr>, "unit": "<unit>"}}` inside a `$project` stage when the timestamp argument is an attribute and an alias is supplied (e.g. `π DATE_TRUNC('hour', at) → hr (Events)`).

Alternatives

`YEAR/MONTH/DAY/HOUR` extract a single numeric component instead of a bucket timestamp.

See Also

[year](#), [month](#), [day](#), [hour](#), [now](#), [timestamp-literal](#)

Notes

DATE_TRUNC is the building block for time-series bucketing and sessionization.

MINUTES (measure a duration in minutes)

Syntax

```
MINUTES(<duration>)
```

```
σ MINUTES(closed - opened) > 30 (Sessions)
```

Description

MINUTES converts a length of time (a DURATION) into a plain number of minutes, so you can compare it naturally — "sessions longer than 30 minutes" reads as `MINUTES(gap) > 30` rather than wrestling with an interval literal.

Technical Description

MINUTES(d: DURATION) → NUMBER. Returns the whole number of minutes in d (Duration.toMinutes — truncated, not rounded). A non-DURATION value is an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Sessions held longer than 30 minutes:

```
σ MINUTES(closed - opened) > 30 (Sessions)
```

Average handling time in minutes:

```
γ agent, AVG(MINUTES(handled)) → avg_min (Calls)
```

Limitations

Argument must be a DURATION (e.g. the result of `TIMESTAMP – TIMESTAMP`). Truncates to whole minutes. NULL in → NULL out.

Alternatives

SECONDS / DAYS measure the same duration in other units.

See Also

[seconds](#), [days](#), [duration-literal](#), [timestamp-literal](#)

Notes

The common pattern is `MINUTES(ts2 - ts1)` — subtract two timestamps to a DURATION, then measure it.

SECONDS (measure a duration in seconds)

Syntax

```
SECONDS(<duration>)
```

```
π id, SECONDS(elapsed) → secs (Requests)
```

Description

SECONDS converts a length of time (a DURATION) into a plain number of seconds, for natural comparisons and reporting of short spans like request latency.

Technical Description

SECONDS(d: DURATION) → NUMBER. Returns the whole number of seconds in d (Duration.getSeconds). A non-DURATION value is an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Latency in seconds:

```
π id, SECONDS(finished - started) → secs (Requests)
```

Requests slower than 5 seconds:

```
σ SECONDS(finished - started) > 5 (Requests)
```

Limitations

Argument must be a DURATION. Whole seconds (no fractional part). NULL in → NULL out.

Alternatives

MINUTES / DAYS for coarser units.

See Also

[minutes](#), [days](#), [duration-literal](#)

DAYS (measure a duration in days)

Syntax

```
DAYS(<duration>)
```

```
π id, DAYS(closed - opened) → days_open (Tickets)
```

Description

DAYS converts a length of time (a DURATION) into a plain number of whole days — ideal for "how many days was this ticket open" or "age in days" reporting.

Technical Description

DAYS(d: DURATION) → NUMBER. Returns the whole number of days in d (Duration.toDays — truncated). A non-DURATION value is an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Days a ticket stayed open:

```
π id, DAYS(closed - opened) → days_open (Tickets)
```

Items older than 90 days:

```
σ DAYS(NOW() - created_at) > 90 (Items)
```

Limitations

Argument must be a DURATION. Whole days (truncated). NULL in → NULL out. For calendar-aware month/year spans, compute from DATE parts instead.

Alternatives

MINUTES / SECONDS for finer units.

See Also

[minutes](#), [seconds](#), [duration-literal](#), [now](#)

to_date (parse a string to a DATE)

Syntax

```
to_date(<string>)
```

```
π id, to_date(date_str) → day (RawImport)
```

Description

to_date turns a text string into a proper DATE value, so date data that arrived as text (from a CSV, a JSON field, or an open schema) can be compared and computed like a real date.

Technical Description

to_date(s: STRING) → DATE. Parses s as an ISO-8601 date (YYYY-MM-DD, LocalDate.parse); a malformed string is an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Convert an imported text date:

```
π id, to_date(date_str) → day (RawImport)
```

Filter parsed dates:

```
σ to_date(due) >= DATE '2026-01-01' (Tasks)
```

Limitations

Expects ISO-8601 (YYYY-MM-DD); other formats error. STRING argument; NULL in → NULL out.

Alternatives

to_timestamp for date+time strings; to_time for clock-time strings; a DATE '...' literal when the value is constant.

See Also

[to_timestamp](#), [to_time](#), [date-literal](#)

to_time (parse a string to a TIME)

Syntax

```
to_time(<string>)  
  
π id, to_time(time_str) → at (RawImport)
```

Description

`to_time` turns a text string into a proper TIME (a clock time), so time-of-day data that arrived as text can be compared and computed like a real time.

Technical Description

`to_time(s: STRING) → TIME`. Parses `s` as an ISO-8601 local time (`LocalTime.parse`); a malformed string is an evaluation error. NULL input returns NULL. PURE, DETERMINISTIC.

Examples

Convert an imported text time:

```
π id, to_time(time_str) → at (RawImport)
```

Within business hours:

```
σ to_time(start) >= TIME '09:00:00' (RawShifts)
```

Limitations

Expects ISO-8601 time (HH:MM[:SS]); other formats error. STRING argument; NULL in → NULL out.

Alternatives

`to_date` / `to_timestamp` for date or date-time strings; a TIME '...' literal for constants.

See Also

[to_date](#), [to_timestamp](#), [time-literal](#)

to_timestamp (parse a string to a TIMESTAMP)

Syntax

```
to_timestamp(<string>)  
  
σ to_timestamp(ts_str) >= TIMESTAMP '2026-01-01T00:00:00Z' (RawEvents)
```

Description

`to_timestamp` turns a text string into a proper TIMESTAMP (an instant), so date-time data that arrived as text can be filtered, sorted, and used in time arithmetic like a real timestamp.

Technical Description

`to_timestamp(s: STRING) → TIMESTAMP`. Parses `s` as an ISO-8601 date-time via the same rule as the `TIMESTAMP` literal (an offset is normalised to UTC; an offsetless payload is interpreted as UTC). A malformed string is an evaluation error. `NULL` input returns `NULL`. PURE, DETERMINISTIC.

Examples

Parse and window imported event times:

```
σ to_timestamp(ts_str) >= TIMESTAMP '2026-01-01T00:00:00Z' (RawEvents)
```

Sort by a parsed timestamp:

```
τ to_timestamp(logged) DESC (RawLogs)
```

Limitations

Expects ISO-8601 date-time; other formats error. Offsets fold to UTC; offsetless values are read as UTC. `STRING` argument; `NULL` in → `NULL` out.

Alternatives

`to_date` / `to_time` for date-only or time-only strings; a `TIMESTAMP '...'` literal for constants.

See Also

[to_date](#), [to_time](#), [timestamp-literal](#), [now](#)

IIf (inline if / conditional value)

Syntax

```
IIf(<condition>, <value-if-true>, <value-if-false>)
```

```
π name, IIf(score >= 50, "pass", "fail") → result (Results)
```

Description

`IIf` chooses between two values based on a condition — the spreadsheet IF. If the condition is true it returns the first value, otherwise the second. Use it to derive labels, flags, or category columns from a test.

Technical Description

`IIf(condition, trueValue, falseValue) → ANY`. The first argument must evaluate to a `BOOLEAN` (else an evaluation error); a `NULL` condition returns `NULL`. Returns `trueValue` when true, else `falseValue`. PURE, DETERMINISTIC.

`IIf` **short-circuits**: the condition is evaluated, then *only* the branch it selects — the other is never evaluated (and a `NULL` condition evaluates neither). That is what makes `IIf` usable as a guard: the untaken branch may be an expression that would fail on those rows, such as converting a value that is not a number. [Nz](#) and [Coalesce](#) are lazy in the same way, and these three are the only built-ins that are — every other function evaluates all its arguments.

Examples

Pass/fail label:

```
π name, IIf(score >= 50, "pass", "fail") → result (Results)
```

Bucket a numeric flag:

```
π order_id, IIf(amount > 1000, "large", "small") → size (Orders)
```

Default a display value:

```
π order_id, IIf(active = true, status, "inactive") → display_status (Accounts)
```

Limitations

The condition must be BOOLEAN. The result type is ANY (the two branches may differ in type). A NULL condition yields NULL — neither branch is evaluated in that case.

Alternatives

Nz / Coalesce specifically for substituting a value when something is NULL.

See Also

[nz](#), [coalesce](#), [isnull](#), [comparison](#)

Notes

For "replace NULL with a default" the dedicated Nz/Coalesce read more clearly than `IIf(IsNull(x), default, x)`.

Nz (null-to-value)

Syntax

```
Nz(<value>)                -- NULL becomes ""
Nz(<value>, <replacement>) -- NULL becomes the replacement

π name, Nz(nickname, name) → display_name (Users)
```

Description

Nz substitutes a fallback when a value is missing (NULL). With one argument a NULL becomes an empty string; with two, a NULL becomes whatever replacement you give. Use it to fill in defaults so missing data doesn't leave blanks downstream.

Technical Description

`Nz(value) → ANY` returns value if non-NULL, else an empty string `""`. `Nz(value, replacement)` returns value if non-NULL, else replacement — as-is, even when the replacement is itself NULL. PURE, DETERMINISTIC.

Nz **short-circuits**: the replacement is evaluated only when the value is NULL. So it may be an expression that would fail on the rows where it is not needed.

Examples

Show a nickname, falling back to the real name:

```
π name, Nz(nickname, name) → display_name (Users)
```

Default a missing count to zero:

```
π id, Nz(views, 0) → views (Pages)
```

Turn a missing note into an empty string:

```
π id, Nz(note) → note (Tickets)
```

Pushdown

SQL: the two-argument form folds to `COALESCE(<value>, <valueIfNull>)` on all dialects. The one-argument form is not pushed: its substitute is the empty string for a `STRING` and `NULL` otherwise, so the answer depends on the argument's type, which the database is not told.

Limitations

The one-argument form's default is an empty string `""`, which may not suit numeric columns — pass an explicit replacement (e.g. `0`) for those.

Alternatives

Coalesce returns the first non-`NULL` of several values. If with `IsNull` for more complex conditions.

See Also

[coalesce](#), [iif](#), [isnull](#), [is-null](#)

Coalesce (first non-null)

Syntax

```
Coalesce(<value>, <default>)
```

```
π id, Coalesce(mobile, landline) → phone (Contacts)
```

Description

Coalesce returns the first value that isn't missing, scanning its arguments left to right. Use it to pick the best available value from several candidates — a mobile number, falling back to a landline, falling back to nothing.

Technical Description

`Coalesce(v1, v2, ...) → ANY` returns the first non-`NULL` argument, or `NULL` if all are `NULL`. Requires at least one argument and accepts any number. `PURE`, `DETERMINISTIC`.

Coalesce **short-circuits**: arguments are evaluated left to right and evaluation stops at the first non-`NULL` one. A later argument may therefore be an expression that would fail on this row — it is never reached.

Examples

First available phone number:

```
π id, Coalesce(mobile, landline) → phone (Contacts)
```

Preferred then fallback display name:

```
π id, Coalesce(display_name, username) → name (Users)
```

Pushdown

SQL: folds to `COALESCE(<a>, <b>, ...)` on all dialects, at any argument count. The two are the same function: first non-NULL, or NULL when there is none.

Limitations

Returns NULL only if every argument is NULL. Result type is ANY. At least one argument is required.

Alternatives

Nz substitutes a single fallback for a NULL; IIf for a general true/false choice.

See Also

[nz](#), [iif](#), [isnull](#), [is-null](#)

IsNull (test for missing value)

Syntax

```
IsNull(<value>)
```

```
σ IsNull(email) = true (Users)
```

Description

IsNull reports whether a value is missing (NULL), returning true or false. Use it in expressions where you need a boolean for missingness — for example inside IIf, or as a computed flag column.

Technical Description

IsNull(value) → BOOLEAN. Returns true iff value is NULL. PURE, DETERMINISTIC. Unlike the `= 1/= NULL` null test (a predicate), IsNull yields a BOOLEAN value usable anywhere an operand is expected.

Examples

Flag rows with a missing email:

```
π id, IsNull(email) → email_missing (Users)
```

Combine with IIf to derive a label:

```
π id, IIf(IsNull(phone), "no phone", phone) → contact (Users)
```

Pushdown

SQL: folds to `<e> IS NULL` on all dialects — an operator rather than a call. SQL Server has no boolean value for it to produce, so there it is `(CASE WHEN <e> IS NULL THEN 1 ELSE 0 END)`, the `BIT` the question answers as. SQL asks the same question of the same three-valued world, so there is no NULL rule to reconcile.

Limitations

For filtering, the dedicated null-test predicate (`email = NULL`) is the idiomatic form; `IsNull` is for use as a boolean value in expressions.

Alternatives

The `= 1 / = NULL` predicate for selection conditions; `Nz/Coalesce` to substitute rather than test.

See Also

[is-null](#), [nz](#), [coalesce](#), [iif](#), [isnumeric](#)

IsNumeric (test if value looks numeric)

Syntax

```
IsNumeric(<value>)  
  
σ IsNumeric(code) = true (RawImport)
```

Description

`IsNumeric` reports whether a value is a number, or a string that can be read as one. Use it to validate messy imported data before converting it — skip the rows whose "amount" isn't actually a number.

Technical Description

`IsNumeric(value) → BOOLEAN`. Returns true for a `NumberValue`, or for a `StringValue` that parses as a `BigDecimal`; false otherwise (including for nested/temporal/NULL values). PURE, DETERMINISTIC.

Examples

Keep only rows whose code parses as a number:

```
σ IsNumeric(code) = true (RawImport)
```

Guard a conversion — `IIf` short-circuits, so `CDBl` never sees the bad rows:

```
π id, IIf(IsNumeric(raw), CDBl(raw), 0) → amount (Imported)
```

Or drop the bad rows entirely rather than defaulting them:

```
π id, CDBl(raw) → amount (σ IsNumeric(raw) = true (Imported))
```

Limitations

Tests parseability, not the declared type — a numeric-looking string returns true. Does not itself convert (pair with `CInt/CDBl`). An *unguarded* conversion of a non-numeric value is an error — `CDBl: cannot convert "abc" to a number` — so pair it with `IIf` or `σ` as above.

Alternatives

`CInt/CDBl` to actually convert; `to_date/to_timestamp` to parse a temporal value.

See Also

[isnull](#), [cint](#), [cdbl](#)

CStr (convert to string)

Syntax

```
CStr(<value>)
```

```
π id, CStr(amount) → amount_text (Orders)
```

Description

CStr converts a value to its text form — turning a number, boolean, or date into a string. Use it when you need a value as text, for example to build a label or concatenate it with other text.

Technical Description

CStr(value) → STRING. Returns the value's canonical display string (asDisplayString). A NULL argument is an evaluation error (CStr cannot convert NULL). PURE, DETERMINISTIC.

Examples

Numeric id as text:

```
π order_id, CStr(order_id) → ref (Orders)
```

Boolean flag as text:

```
π id, CStr(active) → active_text (Accounts)
```

Limitations

Errors on NULL (guard with Nz/IsNull first). The output format is the value's canonical display form.

Alternatives

CInt / CDbI convert toward numbers; to_date / to_timestamp / to_time parse a string into a real temporal value.

See Also

[cint](#), [cdbl](#), [to_date](#), [nz](#)

CInt (convert to integer)

Syntax

```
CInt(<value>)
```

```
π id, CInt(score_text) → score (RawImport)
```

Description

CInt converts a value to a whole number. A numeric value is rounded to the nearest integer, a numeric string is parsed, and a boolean becomes 1 (true) or 0 (false). Use it to coerce text or fractional data into integers.

Technical Description

CInt(value) → NUMBER. NumberValue → rounded to 0 places (HALF_UP); StringValue → parsed numerically; BooleanValue → 1/0. NULL, nested (struct/array), and temporal values are evaluation errors. PURE, DETERMINISTIC.

Examples

Parse a numeric text column to an integer:

```
π id, CInt(score_text) → score (RawImport)
```

Round a fractional value to a whole number:

```
π id, CInt(rating) → rating_int (Reviews)
```

Boolean to 0/1:

```
π id, CInt(active) → active_flag (Accounts)
```

Limitations

Errors on NULL, nested, or temporal values, and on non-numeric strings. Rounds (it does not truncate) — use Fix/Int for directional rounding.

Alternatives

CDBl keeps the fractional part; Round/Int/Fix for explicit rounding behaviour.

See Also

[cdbl](#), [cstr](#), [round](#), [int](#), [fix](#), [isnumeric](#)

CDBl (convert to number)

Syntax

```
CDBl(<value>)
```

```
π id, CDBl(amount_text) → amount (RawImport)
```

Description

CDBl converts a value to a number, keeping any fractional part. A numeric string is parsed, a number passes through, and a boolean becomes 1 or 0. Use it to turn text-form decimals from imports into real numbers you can compute with.

Technical Description

CDBl(value) → NUMBER. NumberValue → itself; StringValue → parsed numerically; BooleanValue → 1/0. NULL, nested, and temporal values are evaluation errors. PURE, DETERMINISTIC.

Examples

Parse a decimal text column:

```
π id, CDb1(amount_text) → amount (RawImport)
```

Sum a parsed numeric column:

```
γ SUM(CDb1(raw_amount)) → total (Imported)
```

Limitations

Errors on NULL, nested, or temporal values, and on non-numeric strings. Unlike CInt it does not round to a whole number.

Alternatives

CInt rounds to an integer; CStr converts toward text.

See Also

[cint](#), [cstr](#), [isnumeric](#)

Entries (read an object's fields as rows)

Syntax

```
Entries(<object>)
```

```
π order_id, Entries(attributes) → a (Orders)
```

Description

Entries turns an object whose keys are data into something you can query. It returns one entry per field, each carrying the field's name as key and the field's value as value, so unnesting the result gives you one row per field.

Use it when a document holds counts, settings or attributes keyed by a name the schema does not know in advance — `{"gift_wrap": "yes", "channel": "web"}` rather than a fixed set of columns. Unnest explodes arrays, so an array-valued field reaches the algebra on its own; this is what lets a keyed object reach it too.

Technical Description

Entries(object) → ANY, an array of `{key: STRING, value: ANY}` structs, one per field of the argument, in the object's own field order. Anything that is not an object — NULL, a scalar, an array — returns NULL, so an inner `μ` drops the row. An object with no fields returns an empty array, which an inner `μ` also drops. PURE, DETERMINISTIC. Never pushed down.

Examples

One row per keyed field, reading the key and the value as columns:

```
query {
  π order_id, a.key → attribute, a.value → value (
    μ a ( π order_id, Entries(attributes) → a (Orders) )
  )
};
```

Given an order with attributes of {"gift_wrap": "yes", "channel": "web"}, that returns two rows for it — one naming gift_wrap, one naming channel.

Which attribute names the data uses at all, which no heading can answer:

```
query {  
  ⚡ ( π a.key → attribute (  
    μ a ( π Entries(attributes) → a (Orders) ) ) )  
};
```

How often each is used, once the keys are rows like any other:

```
query {  
  γ attribute, COUNT(*) → orders (  
    π a.key → attribute ( μ a ( π Entries(attributes) → a (Orders) ) )  
  )  
};
```

Limitations

Reads objects only. An array-valued field is unnested directly with `μ` and returns NULL here, so that there is one way to explode an array rather than two.

Entry values are typed ANY, because an object whose keys are data has no declared shape for its values either. A path into an entry value resolves at runtime, so a field name that is not there reads as NULL rather than failing analysis.

Runs in the engine. MongoDB's `$objectToArray` is the same idea with different entry field names, so a pushed query would not return the same values.

Alternatives

`μ` / UNNEST explodes an array-valued field, which is the same move for the shape whose positions rather than names are the data. For a struct field whose name you know, read it with projection (`π s.field`) and no unnest at all. COLLECT gathers a group's values back into an array.

See Also

[unnest](#), [collect](#), [isnull](#)

11. Advanced operators

Beyond standard SQL — recursion, per-group ranking, sampling, the declarative solver, combinatorial covering, and universal quantification. Each of these pages includes a fully worked example.

| Page | What it does |
|---------------------------|---|
| Universal quantification | <code>∀ / FORALL</code> — groups where every row satisfies a condition |
| Top-K per group | <code>TOP ... PER</code> — the N highest rows in each group |
| Transitive closure | <code>CLOSURE / RCLOSURE</code> — reachability over a binary relation |
| Connected components | <code>CLUSTER ... AS</code> — label each node with its undirected component id |
| Bounded path reachability | <code>PATH ... HOPS m TO n AS</code> — pairs reachable within a hop window, with shortest distance |
| Optimal-path extraction | <code>TRACE ... VIA ... MINIMIZE MAXIMIZE AS</code> — cheapest/longest path with route array |
| General recursion | <code>FIX</code> — least-fixpoint recursion (WITH RECURSIVE) |
| Iteration | <code>ITERATE</code> — repeat a step until it settles: PageRank, state machines, convergence |
| Goal-seek | <code>SOLVE</code> — fill the one blank in an equation, per row |
| Declarative optimisation | <code>OPTIMIZE</code> — knapsack / allocation under constraints |
| Combinatorial covering | <code>COVER</code> — minimal all-pairs (pairwise) subset |
| Bernoulli sampling | <code>SAMPLE p</code> — keep each row with probability p |
| Reservoir sampling | <code>SAMPLE n ROWS</code> — an exact-count uniform sample |
| DOWNSAMPLE | <code>DOWNSAMPLE ts BY '5m' USING AVG</code> — time-series bucket aggregation |
| Sessionization | <code>SESSIONIZE ts GAP DURATION 'PT30M' PER ... AS</code> — split an ordered stream into sessions by an idle gap |
| Why (lineage provenance) | <code>ω / WHY</code> — reify a tuple's lineage (which source rows produced it) as a queryable nested column |
| Query optimizer | The logical rewrite rules (<code>SEL-...</code> , <code>LIM-...</code>) and how to see which fired |
| Pushdown | What the backend computes and what the engine does — the boundary <code>--explain</code> draws |

Universal Quantification ($\forall$ / FORALL)

Syntax

```
 $\forall$  <grouping-keys> : <condition> (Relation)
FORALL <grouping-keys> : <condition> (Relation)
 $\forall$  : <condition> (Relation)           -- no key: whole-relation test

 $\forall$  customer_id : status = "completed" (Orders)
FORALL region, dept : active = true (Staff)
```

Description

For-all keeps a group only when EVERY row in that group satisfies the condition. It answers questions of the form "which customers have ALL their orders completed?" or "which departments have every member active?" — the cases SQL expresses awkwardly with NOT EXISTS (... NOT ...). The result is just the grouping keys of the qualifying groups.

With no grouping key, $\forall$: P (R) asks the yes/no question "does every row satisfy P?" and returns a one-row truth relation (present = yes, empty = no).

Technical Description

$\forall$ _keys:P(R) groups R by the key tuple and emits a key tuple iff all rows in its group satisfy P. NULL semantics are strict: a row whose predicate is UNKNOWN disqualifies its group. Output schema = the grouping keys (or the empty, zero-column schema for the no-key form, which yields DEE/DUM — one empty tuple for true, none for false; vacuously true on empty input). It is a blocking ([bag]) grouped operator. A single-connection keyed $\forall$ pushes down as GROUP BY ... HAVING COUNT(*) = COUNT(CASE WHEN P THEN 1 END) on every SQL dialect.

Examples

Customers whose orders are all completed:

```
 $\forall$  customer_id : status = "completed" (Orders)
```

Departments where everyone is active:

```
FORALL dept : active = true (Staff)
```

Suppliers who only ever shipped on time:

```
 $\forall$  supplier_id : delivered_on_time = true (Shipments)
```

Whole-relation check — did every test pass?

```
 $\forall$  : passed = true (TestRuns)
```

Worked Example

Customer success wants the customers whose orders are *all* completed — accounts with nothing outstanding. The trap with a plain selection is that filtering to status = "completed" and grouping would also pick up customers who happen to have *one* completed order alongside open ones. $\forall$ asks the stricter question.

```
Orders := [
customer	order_id	status
```

```
Alice	1	completed
Alice	2	completed
Bob	3	completed
Bob	4	pending
Cara	5	completed
];
```

```
AllDone := { ∀ customer : status = "completed" (Orders) };
```

Grouping by customer, a group survives only if **every** row in it satisfies the predicate:

```
Alice → completed, completed → all match → kept
Bob   → completed, pending   → one fails  → dropped
Cara  → completed           → all match  → kept
```

The output is just the qualifying grouping keys:

```
query { AllDone };
```

```
customer
-----
Alice
Cara
(2 rows)
```

Contrast the quantifiers: $\forall$ is "all orders completed"; a SEMI join ($\bowtie$) answers "has *at least one* completed order" (which would also include Bob); an ANTI join ($\triangleright$) answers "has *no* completed order". The no-key form $\forall : \text{status} = \text{"completed"} (\text{Orders})$ collapses this to a single yes/no: it returns one (empty) row meaning "every order in the whole table is completed" — here, no row, because Bob's order #4 is pending.

Limitations

Only the keyed form pushes down: the no-key whole-relation form has no natural SQL shape and is computed in the engine.

Alternatives

For "at least one row matches" use a SEMI join ($\bowtie$); for "no row matches" use an ANTI join ($\triangleright$). Division ($\div$) expresses "relates to all of" set-style.

Written out by hand, $\forall$ is **not** "every key, less the keys of a row that fails". That spelling keeps a group it could not evaluate: $\neg P$ over a missing value is itself unknown, so the selection drops the row instead of counting it against the group, and the group survives on a row that never satisfied anything.

```
 $\pi$  customer_id (Orders) -  $\pi$  customer_id ( $\sigma \neg(\text{status} = \text{"completed"})$  (Orders))
```

The equivalent has to subtract the unknowns alongside the failures — which is the whole of what $\forall$ spares you, and the reason it is an operator rather than sugar:

```
 $\pi$  customer_id (Orders) -  $\pi$  customer_id ( $\sigma \neg(\text{status} = \text{"completed"}) \vee \text{status IS NULL}$  (Orders))
```

See Also

[semi-join](#), [anti-join](#), [division](#), [group](#), and

Notes

$\forall$ is first-class (not desugared) and pushes down in the keyed, single-connection case; otherwise it runs in the engine. Every dialect is pushed the same strict form, so a group holding an UNKNOWN row is dropped wherever the query runs — a native `bool_and` would skip that row like any aggregate and keep the group, which is a second answer to the question $\forall$ already answers.

Top-K Per Group (TOP ... PER)

Syntax

```
TOP <count> <sort-specs> [PER <keys>] (Relation)
TOP <offset>, <count> <sort-specs> [PER <keys>] (Relation)

TOP 3 amount DESC PER customer_id (Orders)
TOP 5, 3 amount DESC PER customer_id (Orders)  -- skip 5, take 3 per group
TOP 3 amount DESC (Orders)                    -- one global group: the top 3 overall
```

Description

TOP keeps the highest (or lowest) N rows within each group. It answers "the 3 biggest orders for each customer", "the 5 most recent events per device", "the top earner in each department" — the per-group ranking that SQL needs a window function for. You give the number to keep, how to rank (a sort spec), and the column(s) that define each group (PER).

Technical Description

PER is optional. Written without it, there is one global group and TOP is the top `count` rows overall — which is what λ over τ means, and what the optimizer rewrites that pair into (LIM-003). Writing it directly says the same thing and costs the reader nothing to recognise.

TOP returns, per partition defined by the PER keys, the `count` rows ranked highest by the sort specifications (optionally skipping `offset` rows first) — the `ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...) ≤ k` idiom. Requires ≥ 1 sort spec. Output = full input rows (a windowed filter). It cannot desugar (no per-group ranking primitive) and does not push down ([bag]).

Examples

The 3 biggest orders per customer:

```
TOP 3 amount DESC PER customer_id (Orders)
```

The most recent reading per sensor:

```
TOP 1 reading_time DESC PER sensor_id (Readings)
```

2nd and 3rd place per category (skip the winner):

```
TOP 1, 2 score DESC PER category (Entries)
```

Top earner in each department:

```
TOP 1 salary DESC PER dept (Employees)
```

Worked Example

A retailer wants the two biggest orders for each customer — a leaderboard per group, keeping the full order rows (not just the amount).

```
Orders := [
customer	order_id	amount
Alice	1	120
Alice	2	80
Alice	3	200
Bob	4	50
Bob	5	75
];
```

```
Biggest := { TOP 2 amount DESC PER customer (Orders) };
```

Within each customer partition the rows are ranked by amount DESC and the top two kept:

```
query { Biggest };
```

| customer | order_id | amount |
|----------|----------|--------|
| Alice | 3 | 200 |
| Alice | 1 | 120 |
| Bob | 5 | 75 |
| Bob | 4 | 50 |

(4 rows)

Bob has only two orders, so both survive. To get *2nd and 3rd place* per customer (skip the winner) use the offset form — TOP 1, 2 amount DESC PER customer (Orders) skips 1 row and takes the next 2. And to keep only a single value per group rather than the whole row — e.g. just the highest amount — reach for the ARGMAX aggregate instead.

Limitations

Requires at least one sort spec: without one, "highest" has no meaning.

```
query { TOP 3 (Orders) };
```

Returns whole rows, not a single aggregated value.

The count and the offset are whole numbers; a fraction is a parse error:

```
query { TOP 2.5 amount DESC PER customer_id (Orders) };
```

Alternatives

γ with ARGMAX/ARGMIN returns one value from the extreme row per group (not the whole row, and only one). λ over τ for a global top-N without grouping, which the optimizer rewrites into a keyless TOP (LIM-003, see the [optimizer](#) page).

See Also

[log source](#) — TOP over a web server's access log, among other windowing examples on one real dataset

[argmax](#), [argmin](#), [group](#), [sort](#), [limit](#)

Notes

TOP is the "keep the actual rows" counterpart to the ARGMAX aggregate (which returns a value).

Ties. Where the sort specifications cannot separate two rows, the one that reached the operator first is kept — the same rows a full sort followed by a limit would leave. So `TOP 1 amount DESC PER customer` over two orders of equal amount returns the earlier of the two rather than an arbitrary one. A relation has no inherent row order, though, so "earlier" is only as meaningful as the order the input delivers: add a tie-breaking sort key when the choice matters.

Partition pruning (TOPK-001). A selection that fixes a PER key to a constant above a TOP — `σ customer = "Alice" (TOP 2 amount DESC PER customer (Orders))` — is pushed *below* the operator by the optimizer, so top-k is computed only for the matching group instead of every group: `TOP 2 amount DESC PER customer (σ customer = "Alice" (Orders))`. This is safe because top-k is computed independently per group. Only equalities on a PER key are pushed; a predicate on any other column (or on a computed rank) stays as a residual σ above the operator. Look for a TOPK-001 record in `--optimize` output.

Transitive Closure (CLOSURE / RCLOSURE, or postfix + / *)

Syntax

```
CLOSURE <from>, <to> (Edges)      -- R+: all pairs one or more hops apart
RCLOSURE <from>, <to> (Edges)     -- R*: also adds identity pairs (n, n)
Edges+ OVER (<from>, <to>)        -- same as CLOSURE (+ = U+207A)
Edges* OVER (<from>, <to>)        -- same as RCLOSURE

CLOSURE <from> ↔ <to> (Edges)      -- read both ways (ASCII: <->)
Edges+ OVER (<from> ↔ <to>)       -- same, postfix

CLOSURE manager, report (OrgChart)
```

Description

Closure follows a relationship over and over until it can go no further, giving you everything reachable. Over an edge relation like (parent, child) it computes all ancestor/descendant pairs; over (manager, report) it gives the full management chain; over (part, subpart) it gives a complete bill-of-materials explosion. CLOSURE gives pairs that are one or more hops apart; RCLOSURE also includes each node paired with itself.

Technical Description

Given a binary edge relation projected to two columns, CLOSURE computes R^+ (the transitive closure) and RCLOSURE computes R^* (reflexive-transitive) under set semantics via a semi-naïve least fixpoint. Cyclic graphs terminate because a derived pair is never re-added. Output schema is the two edge columns. A row with a null endpoint is not an edge and is skipped entirely, so neither end of it is a node — which is what decides whether RCLOSURE pairs that value with itself. It does not push down. Under `--provenance` it becomes a semiring-weighted closure (reachability / path count / shortest path / cheapest-route).

Examples

All ancestor/descendant pairs from a parent→child edge list:

```
CLOSURE parent, child (Family)
```

Everyone in a manager's chain of command:

```
CLOSURE <from> ↔ <to> (Edges)      -- read both ways (ASCII: <->)
```

Edges+ OVER (<from> ↔ <to>) -- same, postfix

CLOSURE manager, report (OrgChart)

Reflexive — reachable nodes including the node itself:

```
RCLASURE src, dst (Network)
```

Everyone connected to everyone, ignoring who added whom:

```
CLOSURE person ↔ friend (Friendships)
```

Postfix glyph form (identical result):

```
Family+ OVER (parent, child)
```

Shortest path with weighted closure (run via the CLI):

```
relax --provenance --semiring tropical --weight cost graph.relax
```

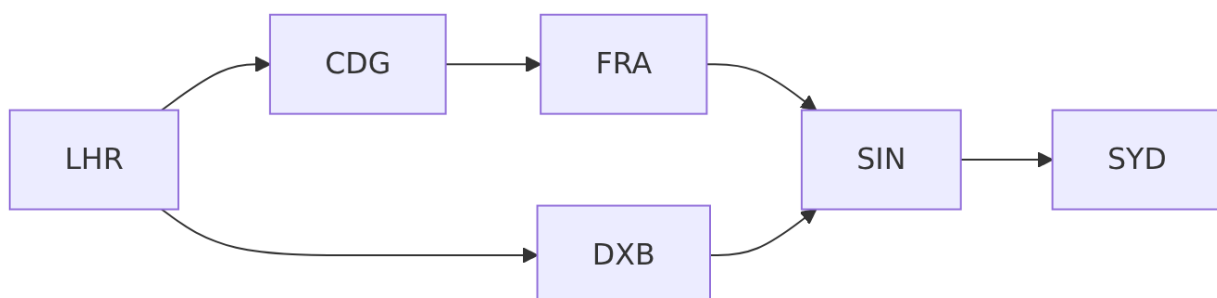
Worked Example

A small airline publishes its direct flights as an edge list. Travel planning needs the *reachability* relation: from each origin, every destination you can get to with any number of connections.

```
Flights := [  
origin	dest
LHR	CDG
CDG	FRA
FRA	SIN
SIN	SYD
LHR	DXB
DXB	SIN
];
```

```
Reachable := { CLOSURE origin, dest (Flights) };
```

The direct network:



Reachable adds every multi-hop pair. From LHR you can reach SYD two ways (LHR→CDG→FRA→SIN→SYD and LHR→DXB→SIN→SYD) but the pair (LHR, SYD) appears just once — closure runs under set semantics, so a derived pair is never re-added:

```
query { τ origin, dest (Reachable) };
```

| origin | dest |
|--------|------|
| CDG | FRA |
| CDG | SIN |
| CDG | SYD |
| DXB | SIN |
| DXB | SYD |
| FRA | SIN |
| FRA | SYD |
| LHR | CDG |
| LHR | DXB |
| LHR | FRA |
| LHR | SIN |
| LHR | SYD |
| SIN | SYD |

(13 rows)

Thirteen pairs from six direct flights: the six edges themselves plus seven derived ones. LHR → FRA is reached via CDG, LHR → SIN via either CDG/FRA or DXB, and LHR → SYD is the full connecting itinerary. The τ sorts the output for readability — closure itself yields a set, in no guaranteed order.

"Can I fly from LHR to Sydney at all?" is then a one-line selection:

```
CanReachSYD := { σ origin = "LHR" ∧ dest = "SYD" (Reachable) };
```

Swap CLOSURE for RCLOSURE to also include the zero-hop identity pairs (LHR, LHR), (SYD, SYD), ... — useful when "stay where you are" counts as a valid itinerary.

Optimization — endpoint pushdown (single-source reachability)

Computing the *all-pairs* closure just to keep one origin (or one destination) is wasteful: the work for every other source is thrown away. When a selection fixes an endpoint to a constant, the optimizer folds that equality **into** the operator as a source/target bound, turning all-pairs reachability into single-source, single-target, or single-pair traversal. This is the canonical *magic-sets / sideways-information-passing* rewrite, recorded as CLOSURE-001 on the relix-events feed (visible via --trace):

```
-- All three of these read directly from the edge relation; after view inlining
-- the σ sits directly above the CLOSURE, so the bound is folded in.
FromLHR := { σ origin = "LHR" (CLOSURE origin, dest (Flights)) }; -- single-source
IntoSYD := { σ dest = "SYD" (CLOSURE origin, dest (Flights)) }; -- single-target (reversed adjacency)
LhrToSyd := { σ origin = "LHR" ∧ dest = "SYD" (CLOSURE origin, dest (Flights)) }; -- single-pair
```

| Form | Pushed bound | Work done |
|-------------------------|------------------------|--|
| σ origin = c | source = c | BFS seeded from c over forward edges |
| σ dest = c | target = c | BFS seeded from c over reversed edges |
| σ origin = a ∧ dest = b | source = a, target = b | single-pair reachability check a → b |

The rewrite is exact: origin/dest are *partitioning* dimensions, so the all-pairs result filtered to origin = c is identical to seeding only from c. Only top-level **equalities** on the endpoint

columns against **literals** are pushed; anything else (an inequality, a predicate on another column, `origin = dest`) stays as a σ above the now-bounded closure, so the optimization can never change the answer. The bound selects the graph's own nodes, by the same equality the σ would have applied — so a literal of a type the nodes cannot be compared with selects none of them, exactly as the σ would have kept no rows. The bound is shown in the physical plan, e.g. `Closure origin=dest [origin="LHR"]` (use `--explain`).

Limitations

CLOSURE works on exactly two columns (a binary relation). For multi-column or more general recursion use FIX. The result can be large on dense graphs; the `--max-fixpoint-rounds` guard aborts runaway iteration.

With `--semiring tropical`, the weight column is read as a double and path costs are summed with double arithmetic. That addition is exact for whole numbers below 2^{53} — hop counts, whole-unit costs, distances in metres — and inexact for values binary floating point cannot represent, which includes most decimals. The practical effect is that a total cost can differ in its final bits depending on the order edges were combined: `0.01 + 0.01 + 0.04` is `0.06` grouped one way and `0.060000000000000005` grouped the other. The route chosen is unaffected unless two routes are within one ulp of each other. Where an exact total matters, scale the weight column to whole units first — cents rather than pounds — the way money is normally stored.

Alternatives

FIX is the general monotone-recursion form (CLOSURE is its specialised, faster two-column case). Composition ($\circ$) is a single hop.

See Also

[fix](#), [composition](#), [natural-join](#)

Notes

CLOSURE and RCLOSURE produce the same AST as the postfix `+ / *` OVER forms; prefer CLOSURE when the two-column form suffices — it uses a faster adjacency-indexed loop than the general FIX.

Connected Components (CLUSTER)

Syntax

```
CLUSTER <from>, <to> AS <label> (Edges)
```

```
CLUSTER src, dst AS island_id (NetworkTraffic)
```

Description

CLUSTER finds the "islands" in a graph: it groups together every node that is connected — directly or through any chain of intermediate nodes — and stamps each node with the id of the island it belongs to. It is the operator behind fraud-ring detection, duplicate-account / entity resolution, and network isolation, where the question is "*which of these things are tied together into one cluster?*" — work that otherwise needs a hand-rolled recursive self-join.

Technical Description

Given an edge relation, CLUSTER reads the two named columns as **undirected** edges (direction is ignored) and computes the graph's connected components by an in-engine union-find least fixpoint. The output is a binary relation with one row per distinct node: the node identifier (keeping the `from` column's name and type) plus a `NUMBER` component-label column named by `AS <label>`. Labels are **canonical** — each component is assigned a dense, 1-based integer in ascending order of the component's minimum node id — so a given graph always produces the same labels regardless of input row order. A row with a null endpoint is **not an edge**, so it is skipped entirely: neither end of it is a node, including the end that is present. A value that appears only in such a row is therefore absent from the output rather than reported as an island of its own — which would also have shifted every later label, since the labels are dense. It is a blocking operator (it must see the whole edge set) and never pushes down to a source.

Examples

Group all connected devices into isolated network islands:

```
CLUSTER src, dst AS island_id (NetworkTraffic)
```

Resolve duplicate accounts linked by shared attributes:

```
CLUSTER account_a, account_b AS entity_id (SharedSignals)
```

Find the components of an undirected friendship graph:

```
CLUSTER person, friend AS community (Friendships)
```

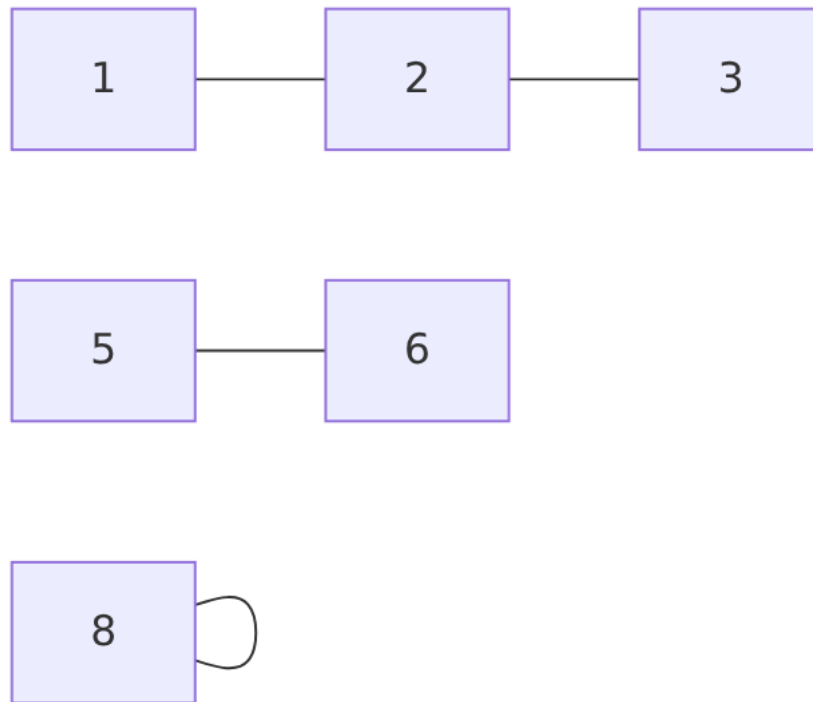
Worked Example

A fraud team has a list of accounts linked by shared signals (same device, same card, same address). Any two accounts in the same connected cluster are likely the same actor. The link list is symmetric data — `a` linked to `b` is the same fact as `b` linked to `a` — exactly what CLUSTER's undirected reading expects.

```
Links := [
a	b
1	2
2	3
5	6
8	8
];

Rings := { CLUSTER a, b AS ring_id (Links) };
```

The link graph has three islands — `{1,2,3}`, `{5,6}`, and the lone self-linked `{8}`:



Rings labels every node with its component, numbered by ascending minimum node id (so the {1,2,3} ring is 1, {5,6} is 2, {8} is 3):

```
query { Rings };
```

| a | ring_id |
|---|---------|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 5 | 2 |
| 6 | 2 |
| 8 | 3 |

(6 rows)

"How many distinct fraud rings, and how big is each?" is then an ordinary aggregation over the labelled output:

```
RingSizes := { γ ring_id, COUNT(a) → members (Rings) };
```

Limitations

CLUSTER is **undirected** connectivity only — there is no directed strongly-connected-components variant. It works on exactly two endpoint columns (a binary edge relation). Isolated nodes that never appear in any edge do not appear in the output (give them a self-edge (n, n) to surface them). Edge-weight / minimum-cut variants are out of scope. The result can be large on dense graphs; as a blocking operator it is subject to the boundedness check over unbounded inputs.

Alternatives

CLOSURE / RCLOSURE compute directed reachability (which nodes reach which) rather than partitioning into components. FIX is the general monotone-recursion form for recursions CLUSTER and CLOSURE do not cover.

See Also

closure, fix, aggregation

Notes

Component labels are deterministic (ordered by each component's minimum node id), so two runs over the same data — in any row order — produce identical labels.

Bounded Path Reachability (PATH)

Syntax

```
PATH <from>, <to> HOPS <m> TO <n> AS <depth> (Edges)

PATH <from>, <to> HOPS <n> AS <depth> (Edges)      // shorthand for HOPS 1 TO n

PATH <from> ↔ <to> HOPS <m> TO <n> AS <depth> (Edges)  // read both ways (ASCII: <->)

PATH src_account, dst_account HOPS 1 TO 3 AS depth (Transfers)
```

Description

PATH answers "*what is reachable within N hops, and how far away is it?*" over a directed graph. It is the bounded, distance-aware companion to CLOSURE: where CLOSURE returns every reachable pair with no notion of distance, PATH returns only the pairs connected by a path whose length falls inside an explicit hop window, and stamps each with its shortest hop distance. It is the operator behind "accounts within N transfer hops of a flagged account" (AML tracing), "people within 3 introductions" (networking), and "components within K dependency levels" (impact analysis) — work that otherwise needs a WITH RECURSIVE CTE plus a depth guard.

Technical Description

Given an edge relation, PATH reads the two named columns as **directed** edges (`from → to`) and performs an in-engine bounded breadth-first traversal over the whole edge set. For every (`from`, `to`) pair connected by a directed path of length `d` with `m ≤ d ≤ n`, it emits one row carrying both endpoints plus a NUMBER distance column named by `AS <depth>`, where `d` is the **shortest** such length. Because the distance is minimal, (`from`, `to`) is a candidate key of the result (no pair appears at two depths). The single-bound form `HOPS n` is shorthand for `HOPS 1 TO n`.

Output schema: the source-endpoint column (keeping the `from` name and type), the target-endpoint column (keeping the `to` name and type), and the `depth` column. A row with a null endpoint is not an edge and is skipped entirely, so neither end of it is a node; cyclic graphs terminate because the window caps the path length. It is a blocking operator (it must see the whole edge set) and never pushes down to a source.

The start node is **not** baked into the operator. To scope a traversal to a particular origin, compose an ordinary selection on the output — the RA-native way:

```
σ src = "1001" (PATH src, dst HOPS 1 TO 3 AS depth (Edges))
```

Examples

Accounts within 1–3 transfer hops of any account:

```
PATH src_account, dst_account HOPS 1 TO 3 AS depth (Transfers)
```

People reachable in exactly 2 introductions:

```
PATH person, contact HOPS 2 TO 2 AS hops (Knows)
```

Degrees of separation, regardless of who introduced whom:

```
PATH person ↔ contact HOPS 1 TO 6 AS degrees (Knows)
```

Everything within 4 dependency levels:

```
PATH module, dependency HOPS 4 AS levels (DependsOn)
```

Worked Example

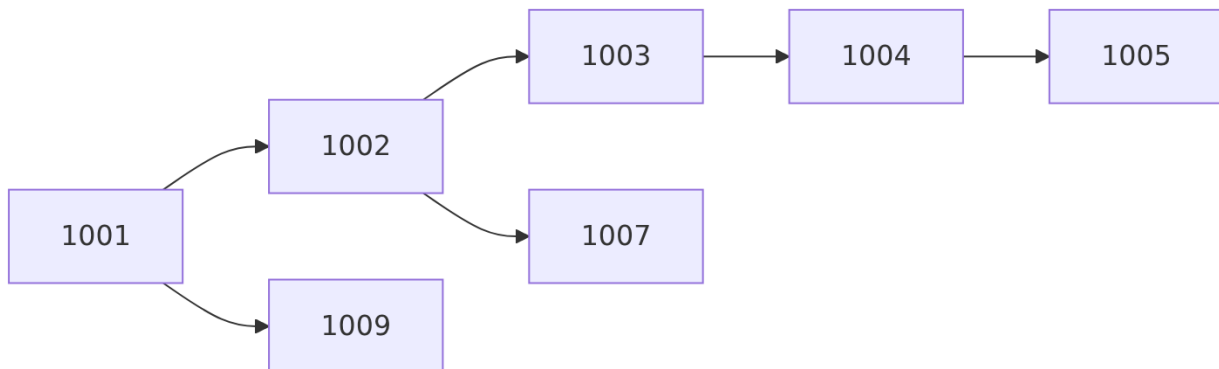
An anti-money-laundering team wants every account reachable from a flagged mule account (1001) within three transfer hops — the laundering chain. Accounts further than three hops are out of scope for this review.

```
Transfers := [  
src_account	dst_account
1001	1002
1001	1009
1002	1003
1002	1007
1003	1004
1004	1005
];
```

```
Reach := { PATH src_account, dst_account HOPS 1 TO 3 AS depth (Transfers) };
```

```
Trail := { π dst_account, depth (σ src_account = 1001 (Reach)) };
```

The money-flow graph downstream of 1001:



Trail keeps the five accounts within three hops, each with its distance — 1005 is four hops out and is correctly excluded:

```
query { τ depth, dst_account (Trail) };
```

| dst_account | depth |
|-------------|-------|
| 1002 | 1 |
| 1009 | 1 |
| 1003 | 2 |
| 1007 | 2 |
| 1004 | 3 |

(5 rows)

Enriching with the account holders is then an ordinary join back to the accounts relation:

```
Suspects := {  $\pi$  holder, bank, risk, depth (Accounts  $\bowtie$  account_id = dst_account Trail) };
```

Optimization — endpoint pushdown (single-source traversal)

A selection fixing an endpoint to a constant is folded into the operator (PATH-001), so the traversal is seeded at that node instead of at every node:

```
 $\sigma$  src = "1001" (PATH src, dst HOPS 1 TO 3 AS depth (Transfers))
```

becomes a single-source breadth-first search rather than an all-pairs one. A bound on the target column is the same search over the reversed adjacency, and both together is a single-pair search.

This is why the start node is deliberately not part of the syntax: scoping a traversal is an ordinary selection, and the optimizer turns it into the cheap operator. The depth column is unaffected — the shortest path from one node does not depend on which other nodes were searched from — so the bounded answer is a slice of the unbounded one.

A predicate on depth, an inequality on an endpoint, or a disjunction is not pushable and remains as a selection above the operator.

Limitations

PATH is **directed** reachability. Edges are read from $\rightarrow$ to; for undirected traversal, union the reversed edges into the input first. The output reports only the shortest distance per pair, not every path length and not the route itself — use TRACE for the optimal route, or CLUSTER for undirected connectivity. The hop window must satisfy $1 \leq m \leq n$. The result can be large on dense graphs; as a blocking operator it is subject to the boundedness check over unbounded inputs and never pushes down to a source.

Alternatives

CLOSURE / RCLOSURE compute **unbounded** directed reachability (every reachable pair, no distance). CLUSTER partitions an undirected graph into connected components. TRACE returns the optimal (min/max-weight) route as an ordered node sequence. FIX is the general monotone-recursion form underlying all of these.

See Also

[closure](#), [cluster](#), [trace](#), [fix](#)

Notes

Output is ordered by (from, to) and is therefore reproducible regardless of input row order.

Optimal-Path Extraction (TRACE)

Syntax

```
TRACE <from>, <to> VIA <weight> MINIMIZE|MAXIMIZE AS <path> (Edges)
```

```
TRACE <from>  $\leftrightarrow$  <to> VIA <weight> MINIMIZE|MAXIMIZE AS <path> (Edges) -- both ways
```

```
TRACE origin, dest VIA cost MINIMIZE AS route (Flights)
```


Description

TRACE finds the cheapest (or highest-score) path between every pair of reachable nodes in a directed weighted graph and returns the path itself as a first-class ordered array value. It answers the question *"what is the optimal route between these two nodes, and exactly which nodes does it pass through?"* — work that CLOSURE/RCLOSURE cannot do (they produce only reachability, not paths or costs).

Technical Description

Given an edge relation, TRACE reads the `from` and `to` columns as directed endpoints and the `weight` column as a numeric edge cost. It computes all-pairs optimal paths using an iterative relaxation algorithm (Bellman-Ford style):

1. **Seed** the path table with each direct edge: `best[(from,to)] = (weight, [from, to])`.
2. **Relax**: for every known path ending at `v` and every outgoing edge (`v→w, ew`), if extending to `w` would be better (lower cost for MINIMIZE, higher for MAXIMIZE), update `best[(origin, w)]` and record the extended path.
3. **Iterate** until no improvement occurs.

Writing `↔` (ASCII `<->`) in place of the comma reads the two columns as an **undirected** edge: each edge is traversable in either direction at the same cost, so a route may run against the direction the row was written in. The `(from, to)` pairs are then symmetric, and a pair's `weight` is the same either way.

Cycle avoidance: a node is never revisited within a single path, so the algorithm terminates on graphs that contain cycles.

The output schema is `(from:T, to:T, weight:NUMBER, path:array<T>)` where `T` is the type of the `from` column. Null endpoints and non-numeric weights are silently skipped.

Unreachable pairs are absent from the output (the operator produces directed reachability only). It is a blocking operator (must see the whole edge set) and never pushes down to a source. The `--max-fixpoint-rounds` guard applies to the relaxation loop.

Examples

Find the cheapest flight itinerary between all connected airport pairs:

```
TRACE <from> ↔ <to> VIA <weight> MINIMIZE|MAXIMIZE AS <path> (Edges)  -- both ways
```

TRACE origin, dest VIA cost MINIMIZE AS route (Flights)

Find the highest-scoring game route between levels:

```
TRACE src, dst VIA score MAXIMIZE AS path (Graph)
```

Use with a selection to trace routes only from a specific origin:

```
σ origin = "JFK" (TRACE origin, dest VIA cost MINIMIZE AS route (Flights))
```

Worked Example

An airline has a table of direct flights with fares. A query analyst wants to find the cheapest itinerary (with intermediate stops) between all airport pairs, and see exactly which airports the route passes through.

```
Flights := [
| origin | dest | cost |
```

```
|-----|-----|-----|
| JFK   | ORD   | 200 |
| ORD   | LAX   | 150 |
| JFK   | LAX   | 500 |
| ORD   | DEN   | 100 |
| DEN   | LAX   | 80  |
|];
```

```
CheapestRoutes := { TRACE origin, dest VIA cost MINIMIZE AS route (Flights) };
```

The flight graph:

```
JFK --200--> ORD --150--> LAX
      |
      100
      |
      DEN --80--> LAX
```

```
JFK --500--> LAX (direct, expensive)
```

`CheapestRoutes` finds the optimal path for every reachable pair:

```
query { τ origin, dest (CheapestRoutes) };
```

| 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)

One row per reachable pair, each carrying the winning route. `JFK → LAX` costs 350 through ORD, beating the 500 direct hop, and the `route` column names the stops that make up the total. `ORD → LAX` keeps the direct 150 rather than the 180 available through DEN: a pair appears **once**, holding its optimum, so the losing path is not in the output at all. The `τ` sorts for readability — the fixpoint yields a set in no guaranteed order.

Filter to just JFK departures:

```
JfkRoutes := { σ origin = "JFK" (CheapestRoutes) };
```

Drill into the cheapest JFK→LAX route's stops:

```
JfkLax := { σ origin = "JFK" ∧ dest = "LAX" (CheapestRoutes) };
```

Optimization — endpoint pushdown (single-source path search)

Computing the *all-pairs* optimal-path table just to keep one origin (or one destination, or one pair) is wasteful. When a selection fixes an endpoint to a constant, the optimizer folds that equality **into** the operator as a source/target bound, turning the all-pairs relaxation into single-source, single-target, or single-pair search — the *magic-sets* / *sideways-information-passing* rewrite specialised to optimal paths. It is recorded as `TRACE-001` on the `relax-events` feed (visible via `--trace`):

```
-- After view inlining the σ sits directly above the TRACE, so the bound folds in.
FromJfk := { σ origin = "JFK" (TRACE origin, dest VIA cost MINIMIZE AS route (Flights)) }; -- single-source
IntoLax := { σ dest = "LAX" (TRACE origin, dest VIA cost MINIMIZE AS route (Flights)) }; -- single-target (reversed graph)
JfkToLax := { σ origin = "JFK" ∧ dest = "LAX" (TRACE origin, dest VIA cost MINIMIZE AS route (Flights)) }; -- single-pair
```

| Form | Pushed bound | Work done |
|--|------------------------|---|
| $\sigma \text{ origin} = c$ | source = c | relaxation seeded only from c |
| $\sigma \text{ dest} = c$ | target = c | single-source from c over the reversed graph |
| $\sigma \text{ origin} = a \wedge \text{dest} = b$ | source = a, target = b | single-pair search $a \rightarrow b$ |

The rewrite is exact: origin/dest index independent sub-computations, so the all-pairs result filtered to `origin = c` is identical to seeding the search from c alone (the optimal cost and path are unchanged). Only top-level **equalities** on the endpoint columns against **literals** are pushed; anything else — an inequality such as a `cost < 300` residual, a predicate on another column, `origin = dest` — stays as a σ above the now-bounded trace, so the optimization can never change the answer. The bound is shown in the physical plan, e.g. `Trace origin, dest via cost MINIMIZE AS route [origin="JFK"]` (use `--explain`).

When **both** endpoints are bound and the sense is `MINIMIZE`, the planner makes a second, *physical* decision: it switches the algorithm from the all-pairs Bellman-Ford relaxation to a **single-pair Dijkstra** search with goal-directed early termination (it halts the moment the bound target is finalized). This is a `Stage.PLAN` event (code `"TRACE"`, distinct from the `Stage.OPTIMIZE TRACE-001` boundary event) and shows in the plan as a `[dijkstra]` tag:

```
Trace origin, dest via cost MINIMIZE AS route [origin="JFK"] [dest="LAX"] [dijkstra]
```

Dijkstra is only sound for **non-negative** weights — a precondition the planner cannot verify (edge weights are data, not known at plan time). The executor checks it at runtime and transparently **falls back** to the relaxation if it finds a negative weight, so the answer is always correct regardless of the planner's choice. (`MAXIMIZE` always stays on the relaxation.)

MINIMIZE vs MAXIMIZE

`MINIMIZE` returns the path with the lowest accumulated weight (shortest path, cheapest itinerary). `MAXIMIZE` returns the path with the highest accumulated weight (longest weighted path, highest-scoring route). The objective applies to the sum of edge weights along the path; no path visits a node more than once.

Relationship to CLOSURE

`CLOSURE src, dst (Edges)` answers "is `dst` reachable from `src`?" and emits only the `(from, to)` pair. `TRACE` additionally emits the cost of the best path and the sequence of intermediate nodes. For unweighted reachability, prefer `CLOSURE`; for path-finding with costs, use `TRACE`.

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

`TRACE` works on directed graphs only. The weight column must be `NUMBER` (or `ANY`, resolved at runtime). Cycles are handled by the no-revisit rule (a node is not visited twice in one path), so simple-cycle graphs terminate correctly. `MAXIMIZE` over graphs with

positive-weight improvement-cycles may require a round cap (`--max-fixpoint-rounds`). The operator always runs in-engine and is never pushed down to a SQL or MongoDB source.

See Also

[closure](#), [cluster](#), [fix](#)

Notes

The path column contains an ordered `array<T>` value — the sequence of node identifiers from origin to destination inclusive (first element = origin, last element = destination). Use `μ route` (`CheapestRoutes`) to explode the stops into individual rows if needed.

General Recursion (FIX)

Syntax

```
FIX <name> (<base>, <step>)  
  
FIX Reach (  
  π src, dst (Edges),  
  π src, dst2 → dst (Reach ⋈ π src → dst, dst → dst2 (Edges))  
)
```

Description

FIX is the general recursive query operator — the equivalent of SQL's WITH RECURSIVE or a single Datalog rule. You give it a starting relation (the base) and a rule that derives more rows from what it has so far (the step), and it repeats the rule until nothing new appears. Use it for graph reachability, hierarchy traversal, parts explosions, and any "keep expanding until done" computation that a plain join can't express.

The recursive name is bound inside the step only and refers to "everything computed so far". There it also qualifies the relation's columns — `Reach.dst` — as the name of any relation does.

Technical Description

FIX computes the least fixpoint of `step` seeded by `base`, under set semantics (a derived row is never re-added). The step must be monotone (allowed: σ π ρ δ μ and $\bowtie$ / $\ltimes$ / $\ltimes$ / $\ltimes$ / $\cup$ / $\sqcup$ / $\cap$; forbidden: $-$, $\div$, outer joins, γ , $\forall$, etc.) and linear (exactly one occurrence of the bound name). Output schema = base schema; the step must be union-compatible with the base. Evaluated by a semi-naïve loop.

Set semantics is what makes **cyclic data** terminate: going round a cycle re-derives a row that is already held, so the round adds nothing and the loop closes. That is a claim about cycles in the *data*, not a guarantee about every query — see *Limitations*.

Examples

Transitive closure expressed as general recursion:

```
FIX Reach (  
  π src, dst (Edges),  
  π src, dst2 → dst (Reach ⋈ π src → dst, dst → dst2 (Edges))  
)
```

Bill-of-materials explosion (all sub-components of an assembly).

Ancestor hierarchy from a parent edge.

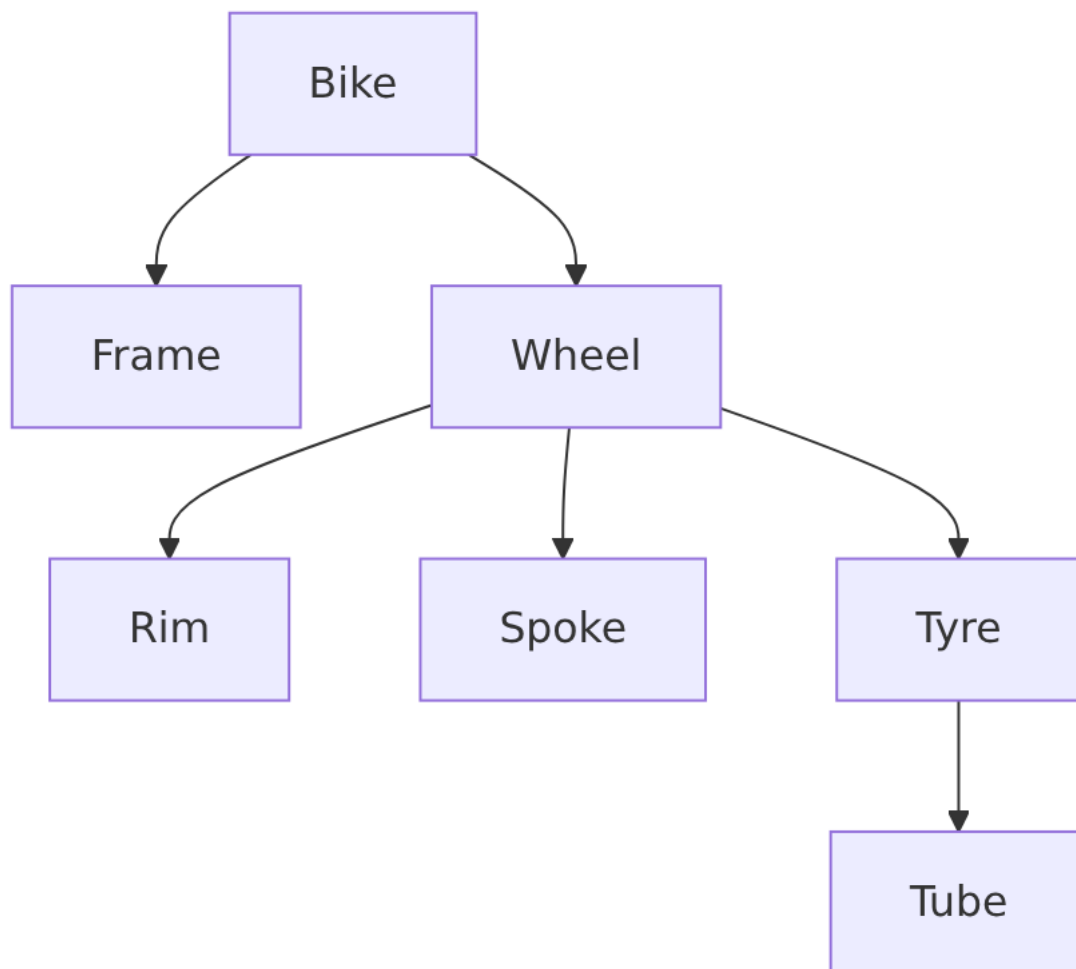
Worked Example

A manufacturer stores its bill of materials as direct "assembly contains part" edges. Procurement needs the full *explosion*: every component of a bike at any depth, no matter how deeply nested.

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

-- (assembly, component) pairs at every depth
Explosion := { FIX BOM (
  Contains,
  π assembly, sub → part (BOM ⋈ ρ Edge(part, sub) (Contains))
) };
```

The nesting it walks:



How the step works: the bound name `BOM` is "everything found so far", with columns `(assembly, part)`. Renaming `Contains` to `Edge(part, sub)` makes its first column share the name `part`, so the natural join `BOM ⋈ Edge` links each known `part` to the things *it* contains; the projection then re-labels the deeper `sub` back to `part` so the step stays union-compatible with the base. Iterating to a fixpoint yields the transitive parts list:

```
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)

Six direct edges become eleven (assembly, descendant) pairs. `Bike` gains `Rim`, `Spoke` and `Tyre` via `Wheel`, and `Tube` two levels down via `Wheel → Tyre`; the `τ` sorts the output for

readability, since the fixpoint yields a set in no guaranteed order.

"Everything I must order to build a bike" is then a selection:

```
BikeParts := {  $\pi$  part ( $\sigma$  assembly = "Bike" (Explosion)) };
```

This particular query is plain two-column reachability, so `CLOSURE` `assembly`, `part` (`Contains`) gives the same answer more concisely — `FIX` earns its keep when the step needs extra columns, a filter, or a join against other relations on each round.

Optimization — magic-sets pushdown (frozen columns, FIX-001)

Computing the *whole* fixpoint just to keep one slice of it is wasteful. When a selection above a `FIX` constrains a **frozen** column — one the step carries through *unchanged* from the recursive reference to its output, round after round — the optimizer pushes that filter **into** the recursion: it seeds the base with it *and* re-applies it to the recursive reference, so the fixpoint only ever derives rows relevant to the bound. This is general *magic-sets / sideways-information-passing*, recorded as `FIX-001` on the `relix-events` feed (visible via `--trace`):

```
-- After view inlining the  $\sigma$  sits directly above the FIX. `assembly` is frozen
-- (the step copies it straight from BOM), so the bound is folded into the recursion:
--    $\sigma$  assembly = "Bike" (FIX BOM (Contains,  $\pi$  assembly, sub  $\rightarrow$  part (BOM  $\bowtie$  ...)))
--    $\rightarrow$  FIX BOM ( $\sigma$  assembly = "Bike" (Contains),
--                $\pi$  assembly, sub  $\rightarrow$  part ( $\sigma$  assembly = "Bike" (BOM)  $\bowtie$  ...))
BikeParts := {  $\pi$  part ( $\sigma$  assembly = "Bike" (Explosion)) };
```

The explosion is now computed **only for the bike** instead of for every assembly — an asymptotic, semantically exact win, and the generalisation of which `CLOSURE`'s single-source pushdown is the two-column special case.

What is pushed: a top-level conjunct is folded in iff every column it references is frozen (carried unchanged by the step), it references at least one column, and it contains no function call. This is broader than `CLOSURE`'s equality-only rule — ranges and `IN` over a frozen column push too. A conjunct on a column the step *produces* anew each round (e.g. `part` above) is **not** frozen and stays as a residual σ above the recursion, so the optimization can never change the answer. The frozen-column analysis is conservative (it recognises $\sigma/\delta/\pi$ -passthrough/ ρ and inner joins where the column comes from the recursive side); anything it cannot prove frozen is simply left un-optimised. Soundness rests on the step being linear and monotone — exactly what `FIX` already requires.

Limitations

`v1` supports linear recursion only (exactly one reference to the bound name in the step). The step must be monotone — no difference, division, aggregation, or outer joins inside it — and it must read the bound name directly: passing it to a table-valued function would hide the body's operators from that check, so it is refused. For the common two-column reachability case `CLOSURE` is simpler and faster.

A `FIX` whose step computes new values may never finish. The termination argument above rests on the step *carrying values through* rather than making them up: if every value it emits already appears somewhere in its inputs, there are finitely many rows it could ever derive, so the loop must run out of new ones. A step that computes breaks that. This one derives a row no round has seen before, every round, forever:

```
Zero := [
n
0
];

query { FIX N (Zero,  $\pi$  n + 1  $\rightarrow$  m (N)) };
```

It is a well-formed FIX — linear, monotone, union-compatible — and nothing rejects it, because "does this step invent values?" is a question about arithmetic rather than about relational shape. NUMBER has no width limit, so the counter never wraps round to a value already held.

This is not a corner case to avoid; it is the reason FIX exists as a general operator rather than a spelling of CLOSURE. Counting paths, accumulating a cost, building up a string, numbering rounds — all of them compute, and all of them are worth writing. Whether such a query finishes depends on the query and the data, and in general there is no way to answer it but to run it. Bound the ones you are unsure of (see *Notes*).

The same applies one step further out: a LATERAL call inside a step passes column values to a table-valued function, and a function that computes over its argument invents values exactly as a projection does.

Alternatives

CLOSURE / RCLOSURE for binary transitive closure (the specialised, preferred form when it suffices). ITERATE for a computation whose rounds replace one another rather than add to each other — PageRank, or a cellular automaton — which needs the aggregation and outer joins FIX's step forbids.

See Also

[closure](#), [iterate](#), [composition](#), [natural-join](#), [union](#)

Notes

Two caps turn a runaway recursion into a clear error instead of a hang, and both are off unless you ask for them:

- `--max-fixpoint-rounds N` — how many rounds one FIX may iterate. Bounds recursion *depth*, so 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 N` — how many rows one operator may hold. For a FIX that is everything it has derived so far, which is what a runaway recursion actually exhausts memory with.

Reach for the round cap when you know the shape of a terminating answer and want to catch the case where you were wrong about it; reach for the row cap when you do not, since it bounds the memory rather than the depth. Setting both is reasonable.

Iteration (ITERATE)

Syntax

```
ITERATE <name> (<base>, <step>) ROUNDS <n>
ITERATE <name> (<base>, <step>) UNTIL STABLE MAX <n> ROUNDS
ITERATE <name> (<base>, <step>) UNTIL <column>, ... WITHIN <tolerance> PER <key>, ... MAX <n> ROUNDS

ITERATE Board (Start, <next generation of Board>) ROUNDS 4
```

Description

ITERATE repeats a computation, each round starting from what the previous round produced. You give it a starting relation (the base) and a rule for turning one round into the next (the step), and say when to stop: after a fixed number of rounds, when a round changes nothing, or when the numbers it computes stop moving.

It is the operator for anything that *settles* rather than *accumulates*: PageRank and other scores that are refined until they converge, label propagation, and state machines such as a cellular automaton, where each generation replaces the last. FIX, by contrast, only ever adds rows, and so cannot express any of these.

Technical Description

ITERATE name (base, step) evaluates base, then evaluates step with name bound to the whole of the previous round's output, which the step's output then replaces. The result is the last round. Each round is a set — duplicate rows are collapsed — and the step must be positionally union-compatible with the base, whose column names the output keeps.

name is visible in the step only, where it also qualifies the relation's columns — R.rank — as the name of any relation does. The step may hand the relation to a table-valued function that takes a relation parameter — ITERATE G (Start, generation(G)) ROUNDS 5 — so a rule written once as a **function** is what each round applies. Because nothing accumulates, the step is free of FIX's rules: it may reference name any number of times, through any operator, aggregation and outer joins included. It must reference it at least once, and it must be deterministic — a step calling Rand() or NOW(), sampling without a seed, or calling a user-defined function is rejected, since a round that could answer differently for the same input makes a convergence test meaningless.

The stop clause is one of three:

- ROUNDS n applies the step exactly n times (ROUNDS 0 is the base). It makes no claim that anything settled, and always finishes.
- UNTIL STABLE MAX n ROUNDS stops at the first round whose output equals its input.
- UNTIL c, ... WITHIN ϵ PER k, ... MAX n ROUNDS pairs each row with the row holding the same key in the previous round, and stops when no value in the UNTIL columns moved by more than ϵ : the largest absolute change, per row, across every named column. A key that appears or disappears counts as a change, and a NULL is unchanged only against another NULL. The UNTIL columns must be NUMBER, and the PER columns must identify one row in every round — a round in which two rows share a key is an error.

The two UNTIL forms have a mandatory round cap, and do not return a result they did not reach. There are two ways to miss it, reported differently:

- **The iteration cycled.** A round reproduced an earlier round exactly. The step is deterministic, so every round after it repeats the same cycle, and the test can never pass: the error names the period.
- **The cap was reached** without the test passing and without a cycle — the iteration may be converging slowly, or not at all.

ITERATE materialises its input, so it is refused over an unbounded relation.

Examples

A cellular automaton — the next generation of Conway's Life, where a cell lives if it has three live neighbours, or two and was already alive:

```
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
];

-- four generations later, the glider has moved one cell diagonally
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))))))
  ∪ 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 };
```

The same step with `UNTIL STABLE MAX 50 ROUNDS` in place of `ROUNDS 4` settles a still life, reports a blinker as a cycle of period 2, and reports a glider as not settling within 50 rounds — it never repeats, it only moves.

A round cap is required for an `UNTIL` test:

```
Start := [
x
1
];

query { ITERATE S (Start, S) UNTIL STABLE };
```

Worked Example

A small web of four pages links to one another, and the question is which pages matter most. PageRank answers it by imagining a reader who follows a random link from each page 85% of the time and jumps to a random page otherwise: a page's rank is the share of time the reader spends there. Every page starts equal, and each round passes each page's rank along its outgoing links until the ranks stop moving.

```

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

Pages := { ⋈ (π src → page (Links) ∪ π dst → 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 };

```

How the step works: `R` is the previous round's (`page`, `rank`). Renaming it to (`src`, `rank`) lets it join the links leaving each page, each carrying its page's out-degree, so `rank / out` is the share passed along one link, and the `γ` sums what arrives at each destination. The left outer join keeps a page that nothing links to — without it, `D` would drop out of the next round — and `Coalesce` gives it nothing passed in. The `UNTIL` clause stops when no page's rank moved by more than 0.0001, pairing the rows of two rounds by page.

```
query { τ rank DESC (Rank) };
```

| page | rank |
|------|----------------|
| C | 0.394199878685 |
| A | 0.37249131329 |
| B | 0.19580880811 |
| D | 0.0375 |

(4 rows)

`C` is linked from three pages and ranks highest; `A` is linked only from `C`, but `C`'s whole rank flows to it. `D` has no incoming links, so its rank is the jump share alone. The ranks sum to 1 — the reader has to be somewhere — to within the rounding of each division.

Limitations

The `UNTIL ... WITHIN` test needs the base's columns to be declared; over a relation whose schema is read from the data it is an error at run time. `NUMBER` division rounds to 10 decimal places, so a tolerance below 0.0000000001 cannot be told apart from zero.

The optimizer does not move a selection or a limit from above an `ITERATE` into it, because a filter on one row can change the others' values — restricting PageRank to one page changes that page's rank. Rewrites inside the step, which preserve each round, still apply, and a sub-expression of the step that does not read the iterated name is evaluated once for the whole iteration rather than once per round.

Alternatives

FIX for recursion that only adds rows — reachability, bills of materials, hierarchies. It needs no round cap and terminates on its own when the data is finite. CLOSURE, PATH and TRACE for the graph questions they name.

See Also

[fix](#), [closure](#), [path](#), [left-outer-join](#), [group](#)

Notes

The `--max-fixpoint-rounds` and `--max-materialized-rows` caps described on the [fix](#) page apply to ITERATE too. The first bounds the rounds of any one ITERATE; the second the rows it holds, which is the current round, the one being built, and one earlier round kept to detect a cycle.

Goal-Seek (SOLVE)

Syntax

```
SOLVE <left-expression> = <right-expression> (Relation)
```

```
SOLVE total = principal * rate (Loans)
```

Description

SOLVE fills in the one missing (NULL) value in an arithmetic equation, per row, by rearranging the formula. Give it an equation like `total = principal * rate` and for each row it figures out whichever single value is blank — if `total` is missing it multiplies, if `rate` is missing it divides. It is a spreadsheet-style goal-seek applied across every row at once.

Technical Description

SOLVE inverts an arithmetic equation per row to fill the single NULL participating column. Both sides are restricted to invertible arithmetic (column / number / + − × ÷ / unary minus) with each column appearing at most once, so the inversion is a deterministic tree-walk. Rows with zero or two-or-more NULL participating columns pass through unchanged. Output schema = input schema. Streaming; does not push down.

Examples

Fill whichever of total/principal/rate is blank, per row:

```
SOLVE total = principal * rate (Loans)
```

Complete a units × price = revenue table where any one column may be missing:

```
SOLVE revenue = units * price (Sales)
```

Recover a missing dimension from area:

```
SOLVE area = width * height (Rectangles)
```

Worked Example

An invoice import arrives with gaps: some rows know the quantity and unit price but not the line total, others know the total and one of its factors. The relationship is always $\text{line_total} = \text{qty} * \text{unit_price}$, so a single SOLVE fills whichever value is blank on each row — the direction is decided per row.

```
-- a blank cell is NULL
Invoice := [
item	qty	unit_price	line_total
Widget	3	5.00	
Gadget		12.00	60.00
Sprocket	4		10.00
];

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

Each row is inverted independently:

```
query { Completed };
```

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

(3 rows)

Widget multiplies ($3 \times 5 = 15$); Gadget and Sprocket divide ($60 \div 12 = 5$ and $10 \div 4 = 2.5$). Numbers print in their shortest exact form, so the 5.00 in the input reads back as 5.

A row that already has every value (nothing blank) or has two blanks among the participating columns is left exactly as it came in — SOLVE only acts when there is precisely one hole to fill.

Limitations

Each participating column may appear at most once on each side, and only basic arithmetic ($+$ $-$ $\times$ $\div$, unary minus) is invertible. A row must have exactly one blank among the participating columns to be solved; otherwise it is left as-is. This is the goal-seeking half of the declarative solver; the optimisation half is OPTIMIZE.

Alternatives

OPTIMIZE for choosing values subject to constraints (rather than inverting a single equation). Projection with arithmetic when nothing is missing and you just want to compute a column.

See Also

[optimize](#), [project](#)

Notes

Direction is decided per row, so the same SOLVE statement fills `total` in one row and `rate` in another depending on which is NULL.

Declarative Optimisation (OPTIMIZE)

Syntax

```
-- MIP subset selection (default):
OPTIMIZE MAXIMIZE|MINIMIZE SUM(<objective>)
  SUBJECT TO SUM(<expr>) <=|>|= <bound> [AND ...] [PER <keys>] (Relation)

-- LP continuous allocation (ALLOCATE):
OPTIMIZE ALLOCATE (<lo>, <hi>) MAXIMIZE|MINIMIZE SUM(<objective>)
  SUBJECT TO SUM(<expr>) <=|>|= <bound> -> <col> [PER <keys>] (Relation)

OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(weight) <= 100 PER region (Candidates)
```

Description

OPTIMIZE solves an optimisation problem directly in a query. In its default mode it picks the best SUBSET of rows that maximises or minimises a total subject to limits — the classic "knapsack": choose items to maximise value while staying under a weight budget. In ALLOCATE mode it instead assigns each row a continuous weight (e.g. a portfolio allocation) to optimise a goal under constraints. Add PER to solve a separate problem for each group.

Technical Description

Default mode is a per-group binary 0/1 knapsack (MIP): per group it chooses the subset of candidate rows that maximises/minimises SUM(objective) subject to one or more SUM(expr) $\leq|\geq|=$ bound constraints; output = the chosen input rows (schema = input — a windowed filter). ALLOCATE mode is an LP: each row gets a continuous variable in [lo, hi]; all rows are returned with an appended allocation column (-> col). PER is optional (absent = whole relation). Solved by the installed mathematical-programming solver (ojAlgo ships with Relix): a group the solver proves infeasible is logged and skipped, while a search that stops without deciding raises an error rather than quietly contributing no rows. [bag]; never pushes down.

Examples

Knapsack — pick the highest-value items under a weight budget, per region:

```
OPTIMIZE MAXIMIZE SUM(value)
  SUBJECT TO SUM(weight) <= 100 PER region (Candidates)
```

Minimise cost while meeting a coverage requirement:

```
OPTIMIZE MINIMIZE SUM(cost)
  SUBJECT TO SUM(coverage) >= 10 (Plans)
```

Portfolio allocation — weights summing to 1, maximising expected return:

```
OPTIMIZE ALLOCATE (0.0, 1.0) MAXIMIZE SUM(ret)
  SUBJECT TO SUM(1) = 1.0 -> weight PER sector (Assets)
```

Worked Example

A product team has a fixed engineering budget of 50 person-days for the next quarter and a backlog of features, each with an estimated business value and a build cost. They want the subset of features that delivers the most value without exceeding the budget — the classic 0/1 knapsack.

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

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

The solver evaluates every feasible subset. The best achievable value inside the budget is 220 — beating, say, SearchRevamp + MobileSync + DarkMode (value 200, cost 35). OPTIMIZE returns the *chosen input rows* (schema unchanged), not an aggregate:

```
query { Roadmap };
```

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

(3 rows)

Two different subsets reach 220 here: the three above at a cost of 45, and MobileSync + Dashboards at exactly 50. Both are optimal, and the solver returns one of them — when an objective ties, which winner you get is the solver's choice, not something the query specifies. Add a tie-breaker to the objective (MAXIMIZE SUM(value) - SUM(cost), say) if you need one particular answer.

Add PER to solve an independent knapsack for each group — one statement, one problem per team, each against its own budget:

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

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

```
query { TeamRoadmaps };
```

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

(3 rows)

Each team is solved against 30 of its own: `search` takes `Dashboards` alone, since adding `SearchRevamp` to it costs 40; `mobile` takes both of its features for 140 at a cost of 25. Neither group's budget constrains the other.

If a group cannot satisfy its constraints (e.g. every feature in it costs more than the budget), that group simply contributes no rows — it is logged and skipped rather than failing the whole query.

Optimization — group pruning (solve only the asked-for group)

`OPTIMIZE` solves an independent problem for each `PER` group, so computing every group's MIP/LP and then keeping one is wasteful — and the solver search is the dominant cost. When a selection fixes a grouping key to a constant, the optimizer pushes that equality **below** the operator, so only the matching group is solved — the *partition-pruning* member of the magic-sets / SIP family. It is recorded as `OPTIMIZE-001` on the `relax-events` feed (visible via `--trace`):

```
-- After view inlining the  $\sigma$  sits directly above the OPTIMIZE, so the bound folds in.
WestPick := {  $\sigma$  region = "WEST"
              (OPTIMIZE MAXIMIZE SUM(value)
               SUBJECT TO SUM(weight) <= 100 PER region (Candidates)) };
--    $\rightarrow$  OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(weight) <= 100 PER region
--       ( $\sigma$  region = "WEST" (Candidates))
```

The rewrite is exact: each `PER` group is a separate sub-problem, so the all-groups result filtered to `region = "WEST"` is identical to solving the `WEST` group alone. Only top-level **equalities** on a grouping key against a **literal** are pushed; everything else stays as a σ above the operator, so the optimization can never change the answer.

In particular a predicate on a **non-grouping** column (e.g. `value > 50`) is *not* pushed: under `OPTIMIZE`'s emit-the-optimum semantics a candidate row the solver would have chosen, then discarded by the post-filter, is **not** the same as a row the solver never sees — removing it from the input would change the chosen optimum. So such a predicate is left as a residual σ above the operator (correctness by construction). `SOLVE` (a per-row equation goal-seek with no search space or group dimension) has nothing to prune and is unaffected.

Limitations

At least one constraint is required, so `SUBJECT TO` cannot be left out:

```
query { OPTIMIZE MAXIMIZE SUM(value) (Candidates) };
```

Constraint operators are $\leq$, $\geq$, or $=$. Infeasible groups produce no rows (MIP) / are skipped (LP). It materialises and never pushes down. Objective and constraints are linear in SUM form.

Alternatives

`SOLVE` for inverting a single equation (no choice/constraints). γ with MAX/MIN for a plain extreme value rather than a constrained optimum.

See Also

[solve](#), [group](#), [top](#)

Notes

OPTIMIZE is the optimisation half of the declarative-solver pair (SOLVE is the goal-seek half), backed by a real mixed-integer / linear-programming solver.

Combinatorial Covering (COVER)

Syntax

```
COVER <strength-t> (Relation)
```

```
COVER 2 (Substrate × Temperature × Coating)  -- all-pairs (pairwise) coverage
COVER 3 (Candidates)                          -- all-triples coverage
```

Description

COVER picks a small subset of rows that still covers every combination of values up to a chosen "strength" — the technique behind all-pairs (pairwise) coverage. Whenever you face an exploding number of combinations but can only afford to try a few, COVER keeps just enough rows that every PAIR of values still appears together at least once. Think of a coatings lab that can run only a handful of expensive physical trials: rather than build all $3 \times 3 \times 3 = 27$ combinations of substrate, curing temperature and coating, COVER 2 finds a far smaller set in which every pair of factor levels is still exercised at least once.

Description (continued)

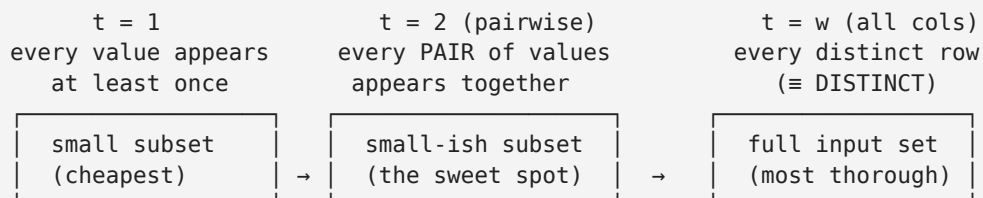
The strength t controls how thorough: $t=1$ means every single value appears at least once; $t=2$ (pairwise) means every pair of column values appears together; $t=w$ (the number of columns) means every distinct row appears (equivalent to δ).

Technical Description

COVER keeps a subset of candidate rows such that every distinct t -column value combination occurring in the input occurs in at least one output row (a windowed filter — output $\subseteq$ input, schema = input). Strength t is a required integer ≥ 1 . The coverage universe is derived from the (already-constrained) candidate set, so excluded combinations are never demanded — constraints come free. Greedy algorithm with a deterministic earliest-buffered tie-break; order the input via τ to bias which candidates win.

Diagram

Strength t interpolates between "every value once" and "every row":



Examples

All-pairs coverage of a parameter space:

```
COVER 2 (Type × Format × Size)
```

All-triples coverage:

```
COVER 3 (Candidates)
```

Bias selection toward high-priority candidates:

```
COVER 2 (τ priority DESC (Candidates))
```

Federation — derive the factor domains live, then cover:

```
Devices := { δ (π device (Sessions)) };
Plan    := { COVER 2 (Devices × Networks × Locales) };
```

Verify coverage in-language (this is empty when fully covered):

```
Missing := { (π device, network (Devices × Networks)) - (π device, network (Plan)) };
```

Worked Example

A materials lab is screening a new protective coating. Three factors each have three levels, so the exhaustive design is $3 \times 3 \times 3 = 27$ physical samples — too many to fabricate. The team only needs every *pair* of factor levels tried together at least once, so they use pairwise (strength-2) coverage.

```
Substrate := [
substrate
Steel
Aluminium
Titanium
];

Temperature := [
temp
120
150
180
];

Coating := [
coating
Epoxy
Polyurethane
Ceramic
];

-- 27 exhaustive combinations collapse to a pairwise-covering subset
Trials := { COVER 2 (Substrate × Temperature × Coating) };
```

Trials contains ten rows instead of 27, yet every (substrate, temperature), (substrate, coating) and (temperature, coating) pair is present somewhere in the set — the row Titanium / 180 / Epoxy, for instance, covers both the (Titanium, 180) and (180, Epoxy) pairs at once:

```
query { τ substrate, temp (Trials) };
```

| substrate | temp | coating |
|-----------|------|--------------|
| Aluminium | 120 | Polyurethane |
| Aluminium | 150 | Epoxy |

```

Aluminium 150 Ceramic
Aluminium 180 Polyurethane
Steel      120 Epoxy
Steel      150 Polyurethane
Steel      180 Ceramic
Titanium   120 Ceramic
Titanium   150 Polyurethane
Titanium   180 Epoxy
(10 rows)

```

Ten is what the greedy search finds; nine is the known optimum for three 3-level factors, so the design is one row off ideal and still under 40% of the exhaustive 27. The `τ` sorts the output for readability — the generator emits rows in the order it selects them.

Prove the design is complete without an external oracle — each of these is empty when every pair is covered:

```

MissingST := { (π substrate, temp (Substrate × Temperature)) - (π substrate, temp (Trials)) };
MissingSC := { (π substrate, coating (Substrate × Coating)) - (π substrate, coating (Trials)) };
MissingTC := { (π temp, coating (Temperature × Coating)) - (π temp, coating (Trials)) };

```

To make the lab fabricate its preferred substrate first when there is a tie, order the candidates before covering:

```

Trials := { COVER 2 (τ substrate ASC (Substrate × Temperature × Coating)) };

```

Limitations

Strength `t` has no default — it is required:

```

query { COVER (Substrate × Temperature × Coating) };

```

The algorithm is greedy (near-minimal, not provably minimal). Output is a subset of the input rows (schema unchanged); it does not push down.

Alternatives

`δ` (DISTINCT) is the `t = (all columns)` extreme. A raw cross product (`×`) gives every combination when you actually want the full exhaustive set.

See Also

[cross](#), [distinct](#), [sort](#), [difference](#)

Notes

Coverage is verifiable in-language with a difference (`−`) of projected column combinations — no external oracle needed.

Bernoulli Sampling (SAMPLE p)

Syntax

```

SAMPLE <probability> (Relation)
SAMPLE <probability> SEED <integer> (Relation)

SAMPLE 0.1 (Events)           -- keep roughly 10% of the rows (non-deterministic)
SAMPLE 0.1 SEED 42 (Events)  -- reproducible 10% sample

```

Description

`SAMPLE p` keeps each row independently with the given probability (between 0 and 1). With `SAMPLE 0.1` each row has a 1-in-10 chance of being kept, so you get roughly 10% of the data — handy for quick exploration of a huge table without processing all of it. The exact number of rows varies run to run because it is random.

The optional `SEED <integer>` clause makes the sample reproducible: two runs with the same seed and the same input produce identical row selections. This is useful for stable test fixtures, benchmarks, or any workflow where the same random sample must be replayed exactly.

Without a seed the sample uses fresh randomness each run.

Technical Description

`SAMPLE p` keeps each input row with independent probability $p \in [0, 1]$. The optional `SEED` clause initialises a seeded `java.util.Random`; without it, `ThreadLocalRandom.current()` supplies fresh randomness. A real `BernoulliSample` physical operator is emitted (formerly desugared to `σ Rand() < p`); the `STREAM` materialisation mode and cost-scaling-by-p are unchanged. Never pushed down to a backend.

Examples

Roughly 10% of an events table for a quick look:

```
SAMPLE 0.1 (Events)
```

Half of the rows, randomly:

```
SAMPLE 0.5 (Survey)
```

Reproducible 30% sample — same rows every run:

```
SAMPLE 0.3 SEED 2026 (Survey)
```

Sample then aggregate for an approximate count:

```
γ region, COUNT(id) → approx (SAMPLE 0.05 (Clicks))
```

Stable test fixture — pin the random rows:

```
SAMPLE 0.2 SEED 99 (LargeTable)
```

Worked Example

Ten rows, each kept with probability 0.5, under a fixed seed so the result can be written down at all:

```
Events := [
id	kind
1	click
2	view
3	click
4	view
5	click
6	view
7	click
8	view
9	click
```

```
| 10 | view |
];

query { SAMPLE 0.5 SEED 42 (Events) };
```

```
id  kind
---  ---
 3  click
 4  view
 7  click
 8  view
 9  click
(5 rows)
```

Five rows out of ten — and that is luck, not arithmetic. Each row was decided independently, so the count is a random variable: with this data and seed it lands on the average, and with another seed it would as easily be four or seven. Over a large relation the count is close to $p \times n$; over a small one it can be anything, including zero. If you need an exact count, that is reservoir sampling, not this.

Change the seed and a different subset comes back; keep it and this exact subset comes back every time, which is what makes a sampled fixture usable in a test.

Limitations

The output size is approximate, not exact — for a fixed count use `SAMPLE n ROWS`. Different seeds produce different (independent) samples; the same seed always produces the same sample for a given input.

Alternatives

`SAMPLE n ROWS` for an exact-count uniform sample. `LIMIT` for a deterministic prefix (not random). `SAMPLE n ROWS SEED k` for a reproducible fixed-count sample.

See Also

[sample-reservoir](#), [limit](#), [rand](#)

Notes

The `SEED` value is an integer literal written in the script. The seed applies to the per-row Bernoulli coin flips only; the same seed with different input data produces a different set of output rows.

Reservoir Sampling (SAMPLE n ROWS)

Syntax

```
SAMPLE <count> ROWS (Relation)
SAMPLE <count> ROWS SEED <integer> (Relation)

SAMPLE 100 ROWS (Events)           -- keep a uniform 100-row sample (non-deterministic)
SAMPLE 100 ROWS SEED 42 (Events)  -- reproducible 100-row sample
```

Description

`SAMPLE n ROWS` keeps exactly `n` rows chosen uniformly at random (or all rows when the relation has fewer than `n`). Unlike `SAMPLE p`, where the count varies, this gives you a precise sample size — the fixed-count version of "give me 100 random rows". Perfect for building a representative fixed-size sample for testing or preview.

The optional `SEED <integer>` clause makes the sample reproducible: two runs with the same seed and the same input always pick the same `n` rows in the same order. This is invaluable for stable test data, reproducible benchmarks, or any workflow where you need to replay the exact same random sample.

Without a seed the chosen rows differ run to run.

Technical Description

`SAMPLE n ROWS` returns exactly $\min(n, |\text{input}|)$ rows drawn uniformly via Vitter's Algorithm R — a single streaming pass holding at most `n` rows in a reservoir. The optional `SEED` clause initialises a seeded `java.util.Random`; without it, `ThreadLocalRandom.current()` supplies fresh randomness. [bag] materialisation; never pushes down to a backend. Cost estimate = $\min(n, |\text{input}|)$.

Examples

Exactly 100 random rows for a test fixture:

```
SAMPLE 100 ROWS (Events)
```

A uniform 1000-row preview of a large import:

```
SAMPLE 1000 ROWS (RawImport)
```

Reproducible 50-row sample — same rows every run:

```
SAMPLE 50 ROWS SEED 2026 (Events)
```

Sample within a filtered subset:

```
SAMPLE 50 ROWS (σ region = "EMEA" (Customers))
```

Reproducible filtered sample for a stable benchmark:

```
SAMPLE 200 ROWS SEED 42 (σ status = "active" (Users))
```

Worked Example

The same ten rows, asked for exactly three:

```
Events := [
id	kind
1	click
2	view
3	click
4	view
5	click
6	view
7	click
8	view
9	click
10	view
```

```
];

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

id  kind
—  ———
 4  view
 2  view
 3  click
(3 rows)
```

Exactly three, every time — that is the difference from `SAMPLE p`, where the count varies. Note that they do not come back in input order: a reservoir fills with the first n rows and then replaces slots at random, so the surviving rows sit in whichever slots they landed in. The draw is uniform, so no position is favoured — the last row of a stream is as likely to be kept as the first, which is the property that lets the algorithm work without knowing the row count in advance.

Asking for more rows than the relation holds returns all of them rather than failing:

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

id  kind
—  ———
 1  click
 2  view
 3  click
 4  view
 5  click
 6  view
 7  click
 8  view
 9  click
10  view
(10 rows)
```

Limitations

It buffers a reservoir, so — unlike `SAMPLE p` — it cannot be pushed to the data source. If the input has fewer than n rows you simply get all of them. Different seeds produce different (independent) selections; the same seed always produces the same selection for a given input.

Alternatives

`SAMPLE p` (Bernoulli) for an approximate-percentage sample that streams. `LIMIT` for a deterministic (non-random) prefix. `SAMPLE p SEED k` for a reproducible approximate-percentage sample.

See Also

[sample-bernoulli](#), [limit](#)

Notes

The disambiguation from Bernoulli SAMPLE is the `ROWS` keyword after an integer count. The SEED value is an integer literal written in the script. The seed applies to the Algorithm R reservoir-replacement decisions only; the same seed with different input data produces a different set of output rows.

DOWNSAMPLE

Syntax

```
```
DOWNSAMPLE <tsCol> BY '<interval>' USING <fn> (Relation)
DOWNSAMPLE <tsCol> BY '<interval>' USING <fn> PER <key> [, <key> ...] (Relation)
DOWNSAMPLE <tsCol> BY '<interval>' USING <fn> PER <key> [, ...] FOR <n> ROWS (Relation)
```
```

``fn`` is one of: ``AVG``, ``SUM``, ``MIN``, ``MAX``, ``COUNT``

Interval formats:

- Shorthand: ``5m``, ``1h``, ``1d``, ``30s``, ``1w`` (seconds / minutes / hours / days / weeks)
- ISO-8601: ``PT5M``, ``PT1H``, ``P1D``, ``P7D``

Description

DOWNSAMPLE is the time-series aggregation operator. It groups rows into fixed-width time windows (buckets) and consolidates columns within each window using the chosen function. It answers questions like "give me the average CPU per 5-minute window" or "total sales per hour, per region".

Each output row represents one bucket. The `bucket` column holds the aligned start time of the window (a `TIMESTAMP`).

Which columns are consolidated depends on the function. `AVG` and `SUM` are arithmetic and consolidate the `NUMBER` columns only; every other column that is not a grouping key or the timestamp is dropped. `MIN` and `MAX` compare rather than compute, and rank a date, a time or a string exactly as `MIN(...)` does, so they consolidate every scalar column — `min_host` is the alphabetically first host in the bucket. Nested (array/struct) columns are never consolidated.

The optional `PER` clause adds grouping dimensions — one output bucket row per distinct combination of (grouping keys + bucket start). The optional `FOR n ROWS` clause keeps only the `N` most recent buckets (sorted by bucket descending), useful for "last `N` windows" queries.

Output schema

For `AVG` / `SUM` (numeric columns only):

```
`<key_col_1>`, ..., `bucket: TIMESTAMP`, `<fn>_<numeric col>: NUMBER`, ...
```

For `MIN` / `MAX` (every scalar column, keeping its type):

```
`<key_col_1>`, ..., `bucket: TIMESTAMP`, `<fn>_<col>: <type of col>`, ...
```

For `COUNT`:

```
`<key_col_1>`, ..., `bucket: TIMESTAMP`, `count: NUMBER`
```


Column order: grouping keys (original types), then `bucket`, then consolidated columns in input-schema order.

Technical Description

DOWNSAMPLE maps each input `TIMESTAMP` to a bucket start via `floor(epoch_seconds / interval_seconds) * interval_seconds`. Rows are then grouped by (groupingKeys, bucket) and the eligible non-key, non-timestamp columns are consolidated with the chosen function.

The consolidation is performed by the same aggregate a `y` would use, so the two agree: `NULL` values are skipped in `AVG`/`SUM`/`MIN`/`MAX` (SQL-style), a bucket with no non-`NULL` value for a column yields `NULL` there, and `COUNT` counts every row including those whose columns are all `NULL`. `MIN`/`MAX` order values the way an ascending `t` does, which is what lets them consolidate timestamps and strings.

Materialisation: `[bag]` — all input rows are buffered before any output row is emitted. Never pushes down to SQL or MongoDB.

Examples

Basic: average CPU per 5-minute window

```
source Metrics from csv("metrics.csv") {
  schema: { ts: TIMESTAMP, host: STRING, cpu: NUMBER, mem: NUMBER }
};
query { DOWNSAMPLE ts BY '5m' USING AVG (Metrics) };
```

Rows are assigned to the bucket their timestamp falls in — seconds 0-299 to the `00:00` bucket, 300-599 to `00:05` — and `AVG` is computed independently per numeric column. Worked through with real output below.

With `PER` grouping: total events per hour per region

```
source Events from csv("events.csv") {
  schema: { ts: TIMESTAMP, region: STRING, amount: NUMBER }
};
query { DOWNSAMPLE ts BY '1h' USING SUM PER region (Events) };
```

With `PER`, bucketing happens within each group: four rows in the same hour but two regions give two output rows, one per region, each summing only its own.

MIN over columns that are not numbers

```
source Spans from csv("spans.csv") {
  schema: { ts: TIMESTAMP, host: STRING, started: TIMESTAMP, latency: NUMBER }
};
query { DOWNSAMPLE ts BY '1h' USING MIN (Spans) };
```

Every scalar column is consolidated, each keeping its own type: the alphabetically first host, the earliest start time, the smallest latency. `ts` is the bucketing column and is replaced by `bucket`. The same query `USING AVG` would produce `avg_latency` alone — averaging a hostname means nothing, so `AVG` and `SUM` take the numeric columns only. Worked through with real output below.

FOR n ROWS: last 24 one-hour buckets

```
query { DOWNSAMPLE ts BY '1h' USING AVG PER host FOR 24 ROWS (Metrics) };
```

Keeps only the 24 most recent hour buckets (most recent first in the ordering applied for selection; output is otherwise unordered).

COUNT: how many events per day

```
query { DOWNSAMPLE ts BY '1d' USING COUNT (Events) };
```

Output schema: `bucket: TIMESTAMP, count: NUMBER`. Each row counts how many input rows fall in that 24-hour UTC window.

Worked Example

Four CPU readings three minutes apart, bucketed into five-minute windows. An inline table's columns are `STRING` or `NUMBER`, so the timestamp is converted first — `DOWNSAMPLE` needs a real `TIMESTAMP` to align buckets against:

```
Readings := [
host	at	cpu
web1	2026-06-01T10:00:00Z	20
web1	2026-06-01T10:03:00Z	40
web1	2026-06-01T10:07:00Z	60
web1	2026-06-01T10:09:00Z	80
];
```

```
Typed := { π host, to_timestamp(at) → at, cpu (Readings) };
```

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

| host | bucket | avg_cpu |
|------|----------------------|---------|
| web1 | 2026-06-01T10:00:00Z | 30 |
| web1 | 2026-06-01T10:05:00Z | 70 |

(2 rows)

Two buckets out of four rows. The `bucket` column holds the **aligned start** of each window — 10:00 and 10:05, not the timestamp of the first row that fell into it — so buckets from different hosts or different queries line up. The 10:00 bucket averages 20 and 40; the 10:05 bucket averages 60 and 80.

Note what `AVG` did to `host`: it survived because it is a `PER` key. Had it not been, an arithmetic consolidation would have dropped it — there is no meaningful average of a hostname.

MIN and MAX are the other kind. They compare rather than compute, so they consolidate every scalar column, not just the numeric ones:

```
Spans := [
at	host	started	latency
2026-06-01T10:00:00Z	web3	2026-06-01T10:00:10Z	12
2026-06-01T10:20:00Z	web1	2026-06-01T10:19:55Z	30
2026-06-01T10:40:00Z	web2	2026-06-01T10:39:58Z	7
];
```

```
TypedSpans := {
  π to_timestamp(at) → at, host, to_timestamp(started) → started, latency (Spans)
};
```

```
query { DOWNSAMPLE at BY '1h' USING MIN (TypedSpans) };
```

| bucket | min_host | min_started | min_latency |
|---------------------------------|----------|----------------------|-------------|
| 2026-06-01T10:00:00Z
(1 row) | web1 | 2026-06-01T10:00:10Z | 7 |

One bucket, and three consolidated columns that came from three different rows: the alphabetically first host, the earliest start, the smallest latency. Nothing here is a row that existed — a bucket is an aggregate, and `MIN` is applied per column exactly as `γ MIN(...)` would.

Limitations

- The timestamp column must be of type `TIMESTAMP` (or `ANY`); other types are a validation error.
- `AVG` and `SUM` are defined over numbers only; a column of another type is dropped rather than consolidated. `MIN/MAX` have no such restriction.
- Nested (array or struct) columns are never consolidated — the engine defines no ordering over them.
- `DOWNSAMPLE` does not push down to SQL or MongoDB — it always runs in-engine.
- The interval must not be empty, and `FOR n ROWS` keeps at least one bucket. Both are parse errors:

```
query { DOWNSAMPLE ts BY '' USING AVG (Metrics) };
```

```
query { DOWNSAMPLE ts BY '1h' USING AVG FOR 0 ROWS (Metrics) };
```

- The `FOR n ROWS` limit applies globally (not per grouping key). If you have multiple `PER` keys, the `N` most-recent buckets across all key combinations are kept.
- Week intervals (`1w`) are counted as $7 \times 86\,400$ seconds from the Unix epoch (UTC) and are not calendar-aware.
- A bucket with no input rows produces no output row, so the series has gaps where nothing happened. An operator over rows that follows, such as `ROLLING ... OVER 3 ROWS`, then counts the buckets that are present, not consecutive intervals: its window can span far more wall-clock time than three buckets' width. The [log source](#) page shows this on real traffic.

See Also

[log source](#) — `DOWNSAMPLE` over a web server's access log, among other windowing examples on one real dataset

[group](#), [sort](#), [limit](#), [TIMESTAMP literal](#), [DATE_TRUNC](#)

Gap-and-Island / Sessionization (SESSIONIZE)

Syntax

```
SESSIONIZE <orderColumn> GAP <threshold> [PER <key> [, <key> ...]] AS <session> (R)
```

```
SESSIONIZE ts GAP DURATION 'PT30M' PER user_id AS session (Events)
```

Description

SESSIONIZE solves the famous "gap-and-island" / sessionization problem: it walks an ordered stream and splits it into **sessions** wherever there is an idle gap. It is the operator behind "group these clicks into browsing sessions", "collapse these sensor pings into bursts of activity", and "find runs of consecutive events" — work that in SQL is a multi-CTE LAG/running-sum incantation. Every input row is kept; SESSIONIZE only *adds* a session-id column, so it composes cleanly with the rest of your query.

Technical Description

Within each partition (the optional PER keys; absent means one partition over the whole relation), rows are ordered ascending by <orderColumn> and scanned once. The first row of each partition is session 1; a new session begins whenever the gap to the prior row exceeds the threshold:

```
orderi - orderi-1 > threshold      ⇒  new session
```

The appended AS <session> column is a dense, 1-based NUMBER that increments at each boundary and restarts at 1 for every partition. The order column is typically a TIMESTAMP (with a DURATION gap, e.g. DURATION 'PT30M') or a NUMBER (with a NUMBER gap) — leaning on the first-class temporal arithmetic substrate (TIMESTAMP - TIMESTAMP → DURATION). The gap is **strict**: a gap exactly equal to the threshold stays in the same session. A NULL order value never opens a new session (it stays in the running one). The output schema is the input schema with the session column appended.

SESSIONIZE buffers and sorts each partition before emitting, so it is a blocking operator ([bag] materialisation), never pushes down to a source, and is subject to the boundedness check over unbounded inputs.

Examples

Group web events into 30-minute browsing sessions per user:

```
SESSIONIZE ts GAP DURATION 'PT30M' PER user_id AS session (Events)
```

Collapse consecutive sensor readings into bursts (idle > 5 sequence units):

```
SESSIONIZE seq GAP 5 AS burst (Readings)
```

Sessionize across the whole stream with no partition:

```
SESSIONIZE at GAP DURATION 'PT1H' AS visit (PageViews)
```

Worked Example

A product team wants to count distinct browsing sessions, where a session ends after 30 minutes of inactivity. The raw event log is one row per page view:

```
Events := [
user_id	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
];
```

```
Sessions := { SESSIONIZE ts GAP DURATION 'PT30M' PER user_id AS session (Events) };
```

Within user 1, the gap 10:00 → 10:20 is 20 minutes (≤ 30 , same session) but 10:20 → 11:30 is 70 minutes (> 30 , new session). User 2 has a single event. The session id restarts per user:

```
query { Sessions };
```

| user_id | 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)

"How many sessions did each user have, and how long was each?" is then an ordinary aggregation over the labelled output:

```
SessionSpans := {
  y user_id, session, COUNT(ts) → events, MIN(ts) → started, MAX(ts) → ended (Sessions)
};
```

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

SESSIONIZE orders by, and computes the gap on, a **single** column. The threshold must be a constant literal — a **NUMBER** for a numeric order column or a **DURATION** for a temporal one. The boundary test is a fixed gap > threshold rule; arbitrary boundary predicates (e.g. "new session when the page changes") are not yet expressible. As a blocking operator it is subject to the boundedness check over unbounded inputs.

Alternatives

WINDOW LAG can compute the inter-row gap as a column if you want to apply your own boundary logic, but it does not assign the running session id. TOP and the window ranking functions rank within a partition but do not segment by a gap.

See Also

[log source](#) — SESSIONIZE over a web server's access log, among other windowing examples on one real dataset

[window-offset](#), [downsample](#), [aggregation](#), [duration literal](#)

Notes

The session id is deterministic for a given partition and order column. The order column is sorted ascending before sessions are assigned, so unsorted input is handled correctly.

Why (WHY / ω)

Syntax

WHY (Relation)

```
 $\omega$  ( $\pi$  region, amount (Orders  $\bowtie$  Orders.customer_id = Customers.customer_id Customers))
```

Description

WHY **reifies a tuple's lineage provenance as ordinary data**: it emits every result tuple of its input unchanged, plus one added nested column, `provenance`, that records *which source rows produced that tuple and how they combined*. Provenance is thereby promoted from a terminal side-channel report (`--provenance JSON`) to a first-class relation $\rightarrow$ relation operator, so the rest of the algebra — σ , π , μ — can slice lineage like any other column.

WHY is a **self-terminating reification boundary**: the lineage semiring is threaded *below* it (exactly as the `--provenance --semiring lineage` side-channel does), frozen into the `provenance` column *at* it, and inert ordinary data *above* it. Nothing above WHY re-threads the annotation, so the rest of a query operates on plain nested values, not live bookkeeping.

Technical Description

WHY(R) evaluates R as a why-provenance K-relation over the polynomial-lineage semiring $\mathbb{N}[X]$ (inlining views and reading every non-positive operator — γ , τ , δ , difference, outer joins, a nested WHY, ... — as an opaque base relation). Each base-tuple occurrence is lifted to a distinct provenance variable carrying its `SourceRef`: the producing relation, a 1-based occurrence ordinal, and the base tuple's captured column values.

The output schema is the input schema plus one column, `provenance`, of type ANY (schema-on-read). For each result tuple the value is the tuple's lineage polynomial reified as a **faithful two-level document** — an array with one element per monomial (distinct derivation):

```
provenance = [ { coefficient : <number>,
                 variables  : [ { relation : <string>,
                                ordinal   : <number>,
                                columns   : { <captured base columns...> } }, ... ],
                 truncated  : <boolean> }, ... ]
```

This preserves the two ways derivations combine:

- **Joint** derivation ($\cdot$, an AND — tuples combined by a join/product) $\rightarrow$ **one monomial** with **several** variables.
- **Alternative** derivation ($+$, an OR — the same result reachable via a union or a duplicate) $\rightarrow$ **several monomials**.

`coefficient` is the derivation's multiplicity. `columns` is the contributing source tuple's captured values, so a consumer can match it back to the exact row. The polynomial-level `truncated` flag rides every derivation element (so it survives a μ `provenance explode`); it is true when the polynomial exceeded the representation cap (`MAX_MONOMIALS = 256`) and some derivations were dropped — a sound under-approximation: the surviving terms are real, but the list is incomplete.

WHY is **blocking** (the canonical K-relation is built before output), a **hard optimizer barrier** (no pass rewrites inside or across it — a column-dropping π that merged duplicates, a δ -elimination, or view inlining that replaced a reference with an aliased body, would silently change the reified lineage), and is **never pushed down** to a SQL or MongoDB backend.

The aggregation wall

WHY reifies only what the lineage evaluator can thread — the positive operators. A γ (or any non-positive operator) below WHY is an opaque boundary: its output rows are read as a fresh source (`Aggregation#n`), not lineage threaded through the grouping. Deep-through-aggregation lineage is a documented post-v1 follow-on.

Examples

Example 1 — base relation: each tuple names its own source row

```
Orders := [ | id | region |
            | 1 | west  |
            | 2 | east  | ];
```

WHY (Orders)

Each output row is the input row plus its (single-derivation, single-variable) provenance:

```
-- id | region | provenance
-- 1 | west  | [ { coefficient: 1,
--      variables: [ { relation: Orders, ordinal: 1,
--                    columns: { id: 1, region: west } } ],
--      truncated: false } ]
-- 2 | east  | [ { coefficient: 1,
--      variables: [ { relation: Orders, ordinal: 2,
--                    columns: { id: 2, region: east } } ],
--      truncated: false } ]
```

Example 2 — joint vs. alternative derivations

A **join** combines two tuples into one result row → **one monomial, two variables**:

```
-- Orders(oid:1, cid:7) ⋈ Customers(cid:7, name:Ada)
WHY (Orders ⋈ Orders.customer_id = Customers.customer_id Customers)

-- provenance of the single result row:
-- [ { coefficient: 1,
--     variables: [ { relation: Orders,   ordinal: 1, columns: {oid: 1, cid: 7} },
--                 { relation: Customers, ordinal: 1, columns: {cid: 7, name: Ada} } ],
--     truncated: false } ]
```

A **union of a duplicated value** yields the same tuple by two routes → **two monomials**:

```
-- Left(x:1) ∪ Right(x:1) (the value 1 is produced by both)
WHY (Left ∪ Right)

-- provenance of the single result row (x = 1):
-- [ { coefficient: 1, variables: [ { relation: Left,   ordinal: 1, columns: {x: 1} } ], truncated: false },
--   { coefficient: 1, variables: [ { relation: Right,  ordinal: 1, columns: {x: 1} } ], truncated: false } ]
```

Example 3 — fully worked: which source rows produced each `west` result

The reified column is ordinary data, so `μ` (UNNEST) explodes its arrays and path navigation reads its fields — no new operators needed.

Source 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 | ];
```

Reify the lineage of a joined, projected view, then flatten it — worked through with real output below:

```
Aug      := { WHY (π region, amount
                  (Orders ⋈ Orders.customer_id = Customers.customer_id Customers)) };
PerDeriv := { μ provenance (Aug) };
Lifted   := { π region, amount, provenance.variables → variables (PerDeriv) };
PerVar    := { μ variables (Lifted) };
West      := { π variables.relation, variables.ordinal, variables.columns
              (σ region = "west" (PerVar)) };
```

(`μ` takes a column, not a path, which is why the array is lifted out of the derivation with `π` before it is exploded.)

`Aug` has three rows (the three orders, each joined to its customer), each carrying a one-monomial provenance with two variables (the `Orders` row and the `Customers` row). `μ` provenance then `μ` variables flatten that to one row per contributing source tuple; the final `σ region = "west"` + `π` names exactly the source rows behind every `west` result:

```
-- West:
-- variables.relation | variables.ordinal | variables.columns
-- Orders             | 1              | { oid: 100, cid: 7, region: west, amount: 40 }
-- Customers          | 1              | { cid: 7, name: Ada }
-- Orders             | 2              | { oid: 101, cid: 7, region: west, amount: 25 }
-- Customers          | 1              | { cid: 7, name: Ada }
```

Data flow through the operator and its `μ/σ/π` consumers:

```
Orders ⋈ Customers — π region, amount —
                                     ω (freeze lineage into `provenance`)
                                     |
Aug rows: (region, amount, provenance: [ {coeff, variables:[...]} ])
                                     |
                                     μ provenance — one row per derivation (monomial)
                                     μ variables — one row per contributing source tuple
                                     σ region = "west" — keep the `west` results
                                     π variables.relation, ...columns — name the exact source rows
```

Worked Example

Two customers, three orders, and the question every lineage feature exists to answer: *which source rows produced this result row?*

```
Orders := [
| oid | cid | region | amount |
```



```

|-----|-----|-----|-----|
| 100 | 7 | west | 40 |
| 101 | 7 | west | 25 |
| 102 | 9 | east | 10 |
];

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

Aug := { WHY (π region, amount (Orders ⋈ 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)

Every result row came through unchanged, with one column added. The table format elides a nested value at the column width — `provenance` is an array of derivations, each an array of contributing source tuples — so reading it in a terminal means taking it apart, which is the point of reifying it as data in the first place.

`μ` explodes the array, giving one row per *derivation* rather than per result row — `provenance` is ordinary data, and the ordinary operators apply to it:

```

query { μ provenance (Aug) };

```

| region | amount | provenance |
|--------|--------|----------------------------------|
| west | 40 | {coefficient: 1, variables: [... |
| west | 25 | {coefficient: 1, variables: [... |
| east | 10 | {coefficient: 1, variables: [... |

(3 rows)

Three rows still, because each of these results has exactly one derivation: a single join path produced it. A result reachable two ways — the same tuple coming from both sides of a `⋈`, say — would appear once per derivation here, which is what makes the exploded form the one to count and group over.

Naming the source rows. One more level down is the list of contributing tuples. `μ` explodes an array held in a *column*, so the array is lifted out of the derivation struct with `π` first, and then the fields of each variable are read the same dotted way:

```

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 `west` result names the two source tuples that combined to produce it — one from `Orders`, one from `Customers` — because a join is a *joint* derivation: one monomial, two variables. The `ordinal` distinguishes occurrences, so the two `west` rows point at different `Orders` tuples while sharing the same `Customers` one.

Nothing in that query is special to provenance. π , μ and σ are doing to the provenance column exactly what they do to any nested column — which is the whole argument for reifying lineage as data instead of reporting it out of band.

Limitations

The added column needs a name the input does not already use. Over a schema-on-read source (JSON, HTTP, MongoDB) there is no declared heading to clash with, so the operator always runs; if a document turns out to carry a field of that name, the added column replaces it.

The provenance column is typed ANY, so paths into it (`variables.relation`, `variables.columns.oid`) are not statically type-checked until a Map type / field-access operand lands. `v1` reifies the **lineage** semiring only; the cheap semirings (boolean / counting / tropical / cost) stay on the `--provenance --semiring ...` side-channel. Lineage stops at the nearest non-positive operator (the aggregation wall). A polynomial past `MAX_MONOMIALS` (256) is truncated and flagged. `WHY` is a hard barrier and is never pushed to SQL/MongoDB.

Alternatives

`--provenance --semiring lineage` is the streaming **side-channel sibling**: the same lineage, surfaced as a terminal report rather than a data column. Use `WHY` when you want to *query* lineage with ordinary views; use the side-channel for a one-shot dump or for the cheap semirings (reachability, path count, shortest path, cheapest route). The reflective `relix.plan` relation is the analogous surface for *plan* metadata (`EXPLAIN-as-data`) rather than data provenance.

See Also

[unnest](#), [project](#), [select](#)

Notes

`WHY` adds no new evaluation concepts — it reuses the provenance evaluator, μ (`UNNEST`), and `ValuePath` navigation. Annotations are a side-channel by default, reified into a data column **only** through the explicit `WHY` boundary. `provenance` is a reserved output column name; a clash with an existing input column is a validation error.

Query optimizer (rewrite rules)

Syntax

```
relix --optimize script.relix      -- print which rules fired, and the before/after trees
relix --trace script.relix         -- a live feed of optimizer, planner and execution decisions

-- in the REPL
:opt <query>                       -- the optimization report for one query
:explain <query>                   -- the physical plan the optimized tree produced
```

Description

Relix rewrites every query's expression tree before planning it. The rewrite is purely *logical* — the result is a different tree that computes the **same relation**, chosen because it does less work. Nothing here changes an answer; a rule that cannot prove itself safe simply does not fire.

Each firing is recorded under a stable **rule code** (SEL-007, LIM-003, ...) so `--optimize` and `:opt` can show exactly what happened and why a plan looks the way it does. The code is the contract: it is stable across releases, greppable in logs, and is what to quote in a bug report.

Views are **inlined first** (INLINE-001), so a rule can optimise across a former view boundary — a filter written outside a view still reaches the table inside it. Inlining leaves an alias wrapper behind, which RENAME-001/RENAME-002 then clear away (see *Rename rules* below).

Why it matters: pushdown compounds

Relix pushes work into the backend where it can. A σ that the optimizer moves *below* a γ becomes a SQL `WHERE` instead of a `HAVING`, so the database filters **before** grouping rather than after. The same holds for the other rules: every operator a filter or limit crosses in the engine is one more thing the source can do instead.

Expression and predicate rules (EXPR-*nnn*, PRED-*nnn*)

These run first, so every later rule matches against settled expressions.

| Code | Rewrite |
|----------|---|
| EXPR-001 | Fold a constant arithmetic expression ($2 * 3 \rightarrow 6$) |
| EXPR-002 | Fold a constant string concatenation |
| EXPR-003 | Drop an additive identity ($x + 0 \rightarrow x$) |
| EXPR-004 | Drop a multiplicative identity ($x * 1, x / 1 \rightarrow x$) |
| EXPR-005 | Replace a multiplication by zero with 0 |
| EXPR-006 | Remove a double negation ($- (-x) \rightarrow x$) |
| EXPR-007 | Reassociate constants so they can be folded ($(x + 1) + 2 \rightarrow x + 3$) |
| EXPR-008 | Drop a redundant call to an idempotent built-in ($\text{UCASE}(\text{UCASE}(x)) \rightarrow \text{UCASE}(x)$) |
| PRED-001 | Fold a constant branch away ($\text{TRUE} \wedge p \rightarrow p$) or short-circuit an absorbing one ($\text{FALSE} \wedge p \rightarrow \text{FALSE}$) |
| PRED-002 | Remove a double negation ($\neg\neg p \rightarrow p$) |

| Code | Rewrite |
|----------|---|
| PRED-003 | Normalise a comparison to attribute-on-left ($5 > \text{age} \rightarrow \text{age} < 5$) |
| PRED-004 | Collapse a conjunction whose bounds on one column cannot all hold ($x > 5 \wedge x < 3 \rightarrow \text{false}$) |
| PRED-005 | Drop a bound another conjunct already implies ($x > 5 \wedge x > 3 \rightarrow x > 5$) |
| PRED-006 | Drop a repeated conjunct ($p \wedge p \rightarrow p$) |

Division by zero is never folded — the error belongs at run time, on the rows that actually hit it, not at plan time on a query that may never reach them. `NOW()`, `CURRENT_DATE()`, `CURRENT_TIME()` and `Rand()` are never folded or de-duplicated either: they read the clock or the RNG, so two calls are not the same value.

Where they apply: every expression, on every operator

An expression is not only a σ predicate or a π attribute. Wherever an operator evaluates a scalar expression or a predicate, both families of rule reach it:

| Operator | The expressions it carries |
|--|--|
| σ | the predicate |
| π | each projected attribute |
| γ | each grouping key, each aggregate argument, an ARGMAX/ARGMIN yield |
| τ , TOP | each sort key |
| TREE | each sibling-ordering key |
| $\forall$ | the predicate |
| $\bowtie$, $\bowtie_L$, $\bowtie_R$, $\bowtie_{>}$, $\bowtie_{<}$, $\bowtie_{\neq}$ | the join condition |
| ASOF | the condition and the tolerance |
| CLOSURE, TRACE | the source and target endpoint bounds |
| SOLVE | both sides of the equation |
| OPTIMIZE | the objective and each constraint expression |
| WINDOW, ROLLING | the function's operands and the SORT keys |
| SESSIONIZE | the gap threshold |
| a generator relation | the PRODUCE ... WHILE bound |
| a table-valued function call | each argument |
| LATERAL | each argument |

So the redundant `* 1` below is folded once, at plan time, rather than being re-evaluated for every row the γ reduces:

```

Orders := [
cust	region	amount	qty
A	west	10	2
A	west	30	1
B	east	5	4

```

```
];
query { γ cust, SUM(amount * qty * 1) → total (Orders) };
```

The γ evaluates `amount * qty` per row instead of `amount * qty * 1`, and the result is unchanged — $A \rightarrow 50$, $B \rightarrow 20$.

The list above is not maintained by hand. Both passes drive their rewrite through the AST's single exhaustive registry of "which expressions does this node carry", so a new operator cannot be added without declaring its expressions, and therefore cannot silently opt out of simplification.

One consequence worth knowing. An *unaliased* π attribute or aggregate takes its output column name from the shape of its expression — a compound expression projects as `_col0` and aggregates as `sum_expr`, while a bare column projects as its own name and aggregates as `sum_amount`. Folding an expression down to a bare column therefore also changes that derived name. Write an explicit $\rightarrow$ alias whenever you refer to the column downstream:

```
Orders := [
cust	amount
A	10
B	5
];
Totals := { γ cust, SUM(amount * 1) → total (Orders) };
query { σ total > 6 (Totals) };
```

Conditions written in operand position

`IIf(cond, a, b)` takes a **predicate as an argument**. That condition is an ordinary predicate in an ordinary expression, and both families of rule reach into it:

```
π IIf(qty * 1 > 0, "some", "none") → stock (Inventory)
π IIf(qty > 0, "some", "none") → stock (Inventory)          -- EXPR-004

σ IIf(5 > age, "child", "adult") = "child" (People)
σ IIf(age < 5, "child", "adult") = "child" (People)          -- PRED-003
```

The rewrite only changes how the condition is *spelled*. `IIf`, `Nz` and `Coalesce` short-circuit, and simplifying a condition never changes which branch they evaluate.

Selection rules (SEL-nnn)

The σ rules move a filter as close to the data as it can legally get.

| Code | Rewrite |
|---------|--|
| SEL-001 | Split a conjunction into stacked selections, so each conjunct moves independently |
| SEL-002 | Merge adjacent selections back into one conjunction (cleanup) |
| SEL-003 | Push σ below π |
| SEL-004 | Push σ below ρ , rewriting column references to their pre-rename names |
| SEL-005 | Push σ into the join input its predicate belongs to |
| SEL-006 | Replicate σ into both branches of $\cup$ |
| SEL-007 | Push σ below γ — SQL's HAVING $\rightarrow$ WHERE demotion |

| Code | Rewrite |
|---------|---|
| SEL-008 | Push σ below δ or τ |
| SEL-009 | Distribute σ over $\cup$, n , Δ and $-$ — both branches, subtrahend included |
| SEL-010 | Merge $\cup/n$ over two selections of the same input into one selection (cleanup) |

SEL-007 — HAVING $\rightarrow$ WHERE

$$\sigma p (\gamma \text{ keys, aggs } (R)) \equiv \gamma \text{ keys, aggs } (\sigma p (R)) \quad \text{when } \text{attrs}(p) \subseteq \text{keys}$$

A condition on a **grouping key** does not need the groups to exist yet: a group either satisfies it or is not wanted at all, so the filter can run on the rows going *in*. A condition on an **aggregate output** genuinely needs the group — that one stays above the γ as a residual σ (a real HAVING).

The rule matches a grouping key only when it is a bare, unaliased column, so the column has the same name below the γ . A derived key (YEAR(order_date) $\rightarrow$ yr) or an aliased one (customer_id $\rightarrow$ cust) is left alone: its output name names no column of the input.

SEL-008 — below δ and τ

$$\begin{aligned} \sigma p (\delta R) &\equiv \delta (\sigma p R) \\ \sigma p (\tau k R) &\equiv \tau k (\sigma p R) \end{aligned}$$

Unconditional. A filter neither adds rows nor changes columns, so deduplicating or sorting the survivors gives the same answer as filtering the deduplicated or sorted rows. It is worth doing because **δ and τ both block**: a σ left above one makes the engine sort or dedupe rows it is about to discard.

SEL-009 — over the set operations

$$\sigma p (A \cup B) \equiv (\sigma p A) \cup (\sigma p B) \quad \text{likewise } n, \Delta \text{ and } -$$

The predicate goes into **both** branches, the subtrahend of $-$ included. That looks like it should be the exception — filtering the subtrahend takes rows out of it, and a row the subtrahend no longer removes sounds like a row added to the result. It is not, because the *same* predicate goes into the minuend:

$$\begin{aligned} \sigma p (A - B) &= \{ r : r \in A, r \notin B, p(r) \} \\ \sigma p A - \sigma p B &= \{ r : r \in A, p(r), \neg(r \in B \wedge p(r)) \} \\ &= \{ r : r \in A, p(r), r \notin B \} \quad \text{-- } p(r) \text{ holds} \end{aligned}$$

A row the filtered B stops removing is a row the filtered A no longer offers. What *is* unsound is pushing a predicate into the subtrahend **alone**, which is not what distribution does.

Three-valued logic needs no separate argument. Where $p(r)$ is UNKNOWN the row is dropped by the σ on the left-hand form and is absent from $\sigma p A$ on the right-hand one, so it is missing from both.

The gain is a real filter on the subtrahend: over a database source that is a WHERE on the right-hand scan rather than a whole-table read.

$\sqcup$ (outer union) is excluded: its branches have different schemas, so a predicate valid against one may name a column the other does not have. $\div$ and $\circ$ are excluded too — neither is a row filter with respect to σ .

SEL-010 — a set operation over two selections of one input

$$\begin{aligned}\sigma_i(A) \cup \sigma_q(A) &\equiv \delta \sigma(i \vee q)(A) \\ \sigma_i(A) \cap \sigma_q(A) &\equiv \delta \sigma(i \wedge q)(A) \\ \sigma_i(A) - \sigma_q(A) &\equiv \delta \sigma(i \wedge (\neg q \vee q \text{ IS UNKNOWN}))(A)\end{aligned}$$

When both branches filter the **same** input, the operation is doing by row-matching what one predicate does by evaluation. This is the inverse of SEL-009, so the two live in different phases — SEL-009 in **pushdown**, this one in **cleanup** — and that is also the useful order: a σ that was distributed and could then be carried no further is merged back into one.

It is worth more than the comparisons it saves, for two reasons:

- **$\cup$ and $\cap$ block.** Each buffers a hash set of one side. The rewrite removes the buffer, not some of the probing.
- **It restores pushdown.** $\sigma_i(\text{Orders}) \cup \sigma_q(\text{Orders})$ over a database source is two **SELECTs** and an in-engine de-duplication, because the SQL renderer folds σ but not **UNION**. One **WHERE** crosses the boundary the set operation was.

```
Orders := [
cust	region	amount
A	west	10
B	east	5
C	west	70
];
query {  $\sigma$  region = "west" (Orders)  $\cup$   $\sigma$  amount > 50 (Orders) };
```

becomes $\delta \sigma(\text{region} = \text{"west"} \vee \text{amount} > 50)(\text{Orders})$ — one scan, one filter.

The δ is part of the rewrite, not decoration. $\cup$ and $\cap$ de-duplicate; a merged σ over a bag input does not. Handing back the bare selection would state a distinctness the input need not have, and a δ that **DIST-001** removed on the strength of it would not come back. The δ is dropped only where the input is provably duplicate-free — either way the set operation is gone.

The $-$ arm is not $i \wedge \neg q$, and the difference is rows. A row whose q is UNKNOWN is absent from $\sigma_q(A)$, so the difference *keeps* it — while $\neg \text{UNKNOWN}$ is UNKNOWN and a selection keeps only what is true, so a bare $\neg q$ drops it. The correction is the selection's **complement**, $\neg q \vee q \text{ IS UNKNOWN}$, written with a null test over the condition's own truth value. See *Reading a condition's truth value* below. $\cup$ is excluded as a bag operation, and Δ would need an XOR.

Volatility. The rewrite reads the input once where the query read it twice, so a branch that is not reproducible — an unseeded **SAMPLE**, a **Rand()** in the predicate — declines.

Projection rules (PROJ-nnn)

The π rules control how **wide** the rows are that flow through the query.

| Code | Rewrite |
|----------|---|
| PROJ-001 | Remove a π that outputs exactly its input's columns (a no-op) |
| PROJ-002 | Merge two stacked π into one |
| PROJ-003 | Push π below σ , so the filter runs on narrower rows |

| Code | Rewrite |
|----------|---|
| PROJ-004 | Prune every base relation to the columns the query actually reads |

PROJ-004 — column pruning

```

γ region, SUM(amount) → total (Orders)
  = γ region, SUM(amount) → total (π region, amount (Orders))

```

PROJ-001..003 are local rewrites of a π that is already written down. PROJ-004 is the one that *introduces* one. It walks the tree **top-down** carrying a "required columns" set: each operator is told which of its output columns the operator above reads, works out what it therefore needs from each input, and passes that down. At a base relation the accumulated set is compared with the table's columns, and anything unread is projected away right at the leaf.

Each operator contributes what it reads itself:

| Operator | What its input must still produce |
|--|---|
| σ | what is read above + the predicate's columns |
| τ | what is read above + the sort keys |
| λ | what is read above — a limit reads no column |
| π | the attribute expressions' own column references |
| γ | the grouping keys + the aggregate arguments — nothing from above |
| ρ | what is read above, mapped back through the renames |
| $\bowtie$ $\bowtie$ $\bowtie$ $\bowtie$
$\bowtie$ $\bowtie$ | what is read above + the join condition, split by side |
| $\bowtie \triangleright$ | as above on the left; on the right, only the condition — a semi- or anti-join emits no right-hand column |

A $\bowtie$ additionally keeps every column the two sides have in common, read or not: those columns *are* its join keys, so pruning one would silently change what the join matches on.

Why this one is worth the most

Width multiplies with everything downstream. A pruned leaf means a narrower `SELECT` list over JDBC, Mongo or HTTP — bytes off the wire, not just cycles — a smaller hash-join build side, and less row width through every operator that has to buffer (τ , γ , `WINDOW`, $\bowtie$, $\cup$, $\div$).

What it will not prune

- **δ , and the deduplicating set operations** $\cup$ $\cap$ $-$ Δ . They compare *whole rows*, so π a (δ R) is not δ (π a R): dropping a column before the dedup merges rows that were distinct. $\div$, $\circ$ and $\cup$ are excluded for the same family of reasons — their semantics are stated in terms of the operand headings. ($\cup$ is safe in principle but its branches are matched positionally, so it is left alone rather than risked.)
- **The positional form of ρ** (ρ E (a, b, c)) — it is arity-bound, so narrowing its input would break the rename. The `old → new` pair form prunes normally.
- **A schema-on-read source** — an open schema has no column list to prune against, so nothing is dropped.

An operator that blocks pruning blocks it only at that edge. A π further down starts a fresh requirement of its own, so $\delta (\pi \text{ a } (\sigma \dots (R)))$ still prunes R .

The rule runs **last**, after every other phase has settled: it inserts a π directly above a base relation, and several rules (GEN-001 most directly) match on a shape that a π between the operator and its leaf would hide.

Limit rules (LIM-*nnn*)

| Code | Rewrite |
|---------|---|
| LIM-001 | Push λ below π |
| LIM-002 | Push λ below ρ |
| LIM-003 | Fuse λ over τ into a keyless TOP — the top-N rewrite |
| LIM-004 | Replicate λ into both branches of $\cup$ |

A λ crosses only operators that are **row-count and order neutral**: π narrows columns and ρ touches names, so neither can filter or reorder. It does *not* cross a σ — fewer than n rows would survive the filter — nor a γ , δ , or set operation.

LIM-003 — top-N

$$\lambda(\text{offset}, n) (\tau \text{ k } (R)) \equiv \text{TOP } n \text{ OFFSET } \text{offset} \text{ ORDER BY } k (R)$$

λ over τ is the only way to write a global "top N overall", and taken literally it sorts the **entire** input to then discard all but n rows. TOP is the same thing done with a bounded heap, so the pair is fused into one. The produced TOP carries no PER key — one global group — which is a shape with no surface syntax of its own precisely because λ/τ already is one.

Over a pushdown-capable source the fused form still goes to the backend as ORDER BY ... LIMIT, exactly as the pair did.

LIM-004 — bounding the branches of a union-all

$$\lambda \text{ n } (A \cup B) \equiv \lambda \text{ n } ((\lambda \text{ n } A) \cup (\lambda \text{ n } B))$$

Neither branch can contribute more than n rows to a limit of n , so producing more is wasted work — and over a database source the branch limit becomes a real LIMIT. The outer λ has to stay: a branch may supply fewer than n rows, so the union still needs capping. With an offset the branch bound is $\text{offset} + \text{count}$, because every skipped row might have come from the same branch.

Join rules (JOIN-*nnn*)

| Code | Rewrite |
|----------|---|
| JOIN-001 | Turn σ over $\times$ into a theta join |
| JOIN-002 | Push σ into the join input its predicate belongs to |
| JOIN-004 | Demote an outer join whose padded rows cannot survive the filter above it |

There is no JOIN-003: it swapped a commutative join's inputs to put the cheaper source on the left, and was removed because that permutes the join's *ordered* output columns — which a positional ρ and the whole-row set operations read by position, so it returned wrong rows. The planner already picks the cheaper build side without touching the logical

tree. **Join shape is fixed by the query text**; the optimizer does no reordering.

JOIN-004 — outer to inner

$\sigma p (A \bowtie B) \equiv \sigma p (A \bowtie_{\theta} B)$ when p rejects NULLs on a column of B

An outer join emits, besides the matched rows, rows padded with NULLs on the side that had no match. If the filter above can never be *true* of a row whose B column is NULL, those padded rows all get thrown away — so producing them was work for nothing.

The demotion is a gate in front of several things an outer join cannot have: σ pushdown into **both** sides (SEL-005 pushes into $\bowtie$'s left only), a sort-merge plan (only INNER/SEMI/ANTI are merge-eligible), a JOIN ... ON pushed to SQL, and EQ-001. The planner also picks the cheaper build side of an inner join by cost, which it does not do for the asymmetric kinds.

A $\bowtie$ loses one half at a time, because each half is killed by a predicate on the *other* side's columns — the unmatched-left rows are the ones carrying NULL right columns:

| Join | filter rejects on left | filter rejects on right |
|-----------|------------------------|-------------------------|
| $\bowtie$ | — | $\bowtie$ |
| $\bowtie$ | $\bowtie$ | — |
| $\bowtie$ | $\bowtie$ | $\bowtie$ |

A $\bowtie$ whose filter rejects on both sides goes straight to $\bowtie$.

What rejects a NULL. A comparison, a LIKE, or an $\in$ whose operand *is* the column; c IS NOT NULL; a $\wedge$ where either side rejects; a $\vee$ where **both** sides do.

What does not, and the first is the one that matters:

- c IS NULL. $\sigma B.x$ IS NULL $(A \bowtie B)$ is the anti-join idiom — the query is asking for exactly the padded rows. Demoting it would return the opposite answer.
- Anything **computed** from the column. Nz, Coalesce and IIf exist precisely to turn a NULL into something truthy, so a function call over c is never assumed to reject.
- $\neg(c = 5)$, deliberately. Only $\neg(c$ IS NULL) is treated as rejecting; the general negated case needs a second, dual analysis rather than an inversion of this one. Under-firing costs an optimization, getting it wrong costs the answer.

A column that either input could own is never attributed to the padded side.

Lateral decorrelation (LATERAL-nnn)

| Code | Rewrite |
|-------------|--|
| LATERAL-001 | Turn an uncorrelated LATERAL TVF join into a $\times$ over a single TVF call |

L LATERAL $f(c...)$ $\equiv$ $L \times f(c...)$ when no argument reads a column of L

A LATERAL join evaluates its arguments against each left row, instantiates the function body with those values, and concatenates the left row with every row the body returns. When the arguments contain no column reference, they evaluate to the same values for every left row — so every instantiation is the *same* relation, and pairing that one relation with each left row is a Cartesian product.

The per-row work it removes. The lateral executor calls back into **the planner** once per left row: N argument round trips (runtime value $\rightarrow$ AST literal) and N freshly planned bodies for N rows. Decorrelated, that is one plan and one execution.

The visibility it buys, which is worth more. `LATERAL` reports only its left input as a child, so the function body sits outside structural traversal and no other rule can see into or through the operator. As a $\times$ over a plain TVF call it is ordinary again: `JOIN-001` can fold a selection above it into a theta join, `JOIN-002/SEL-005` push filters into the left input, and the planner costs it like any other product.

Two properties make the swap safe, and both are about *ordering* rather than row content:

- the output heading is `left ++ body` either way (the same `concat`), so a positional consumer downstream sees no change — the trap the removed `JOIN-003` fell into; and
- $\times$ iterates the left input in the outer loop and the right in the inner one, which is exactly the order the lateral emits its per-row groups in.

When it does not fire. Every condition asks the same question — *would running the body once give what running it once per row gives?*

- **Any `AttributeOperand` anywhere in any argument.** That is the definition of a correlated lateral, and the test is deliberately blunt, since a false "uncorrelated" returns wrong answers.
- **A non-deterministic argument.** `LATERAL f(Rand())` draws once per left row; folding it to a single call would draw once in total. A function counts as deterministic only when the builtin registry tags it so, which makes a user-defined scalar function non-deterministic — the same conservative reading `DIST-002` uses.
- **A non-deterministic function body**, which is the same objection one level down. `LATERAL sampleOf(0.5)` over a body reading `Rand()` or `NOW()`, drawing an unseeded `SAMPLE`, or calling a view or nested TVF that does, produces a different relation per left row even with constant arguments. Answered by walking the body — through the views and nested TVFs it references — which is why this rule needs the symbol table and runs in the preamble rather than as a pipeline rule. A function that cannot be resolved is treated as non-deterministic.

What is deliberately *not* treated as volatile is a **base relation**. Re-reading a source table could in principle return different rows, but the engine already assumes a source is stable for the duration of a query: $\times$ materialises its right input once and replays it for every left row, and every hash join, $\div$ and set operation does the same. Reusing one evaluation of a scan is a liberty the executor takes already.

Equality propagation (EQ-nnn)

| Code | Rewrite |
|--------|---|
| EQ-001 | Carry a literal binding across an equi-join, so both sides are filtered |

$$A.x = B.x \wedge A.x = 5 \quad \models \quad B.x = 5$$

Without this, $\sigma A.x = 5 (A \bowtie B)$ filters `A` and scans `B` whole — every `B` row crosses the wire for the join to throw away. Deriving `B.x = 5` and putting it on `B`'s side means **both backends filter independently**.

```

Customers := [
cid	name
1	ada
2	bea
];	
Sales := [	
cid	amount
-----	-----
1	10
2	40
];
query { σ Customers.cid = 1
        (Customers ⋈ Customers.cid = Sales.cid Sales) };

```

The optimizer derives $\sigma \text{ Sales.cid} = 1$ and places it directly on `Sales`. Over two database sources that is two `WHERE cid = 1` clauses instead of one.

Inner joins only. An outer join's condition does not hold of the rows it pads with NULLs, so an equality read out of $\bowtie/\bowtie/\bowtie$ and pushed into the null-supplying side would delete rows the join is required to keep. JOIN-004 demotes an outer join to inner where it can, and runs first — so the demoted join becomes eligible here in the same phase.

Only *conjuncts* count as facts: an equality inside a $\vee$ holds in one branch only and is never read. NULLs need no special case, because equality is NULL-rejecting — a row satisfying $A.x = B.x \wedge A.x = 5$ has neither column NULL, so the derived predicate discards only rows the join would have discarded.

The column-to-column half is not implemented. $A.x = B.x \wedge B.x = C.x \models A.x = C.x$ is sound, but nothing consumes it today: the optimizer does no join reordering at all, so an implied cross-join equality widens no search anything performs and would only add a predicate to evaluate.

Partition pruning

$$\sigma k = c \text{ (OP ... PER } k \text{ (R))} \quad \equiv \quad \text{OP ... PER } k \text{ (} \sigma k = c \text{ (R))}$$

Five operators compute their result **independently per partition**, so an equality fixing the partition key selects exactly one partition — and pushing it below the operator computes only that one instead of all of them.

| Code | Operator | What is skipped |
|----------------|-----------------------------|-----------------------------------|
| WINDOW-001 | WINDOW ... PARTITION BY k | other partitions' frames |
| TOPK-001 | TOP n ... PER k | other groups' rankings |
| OPTIMIZE-001 | OPTIMIZE ... PER k | other groups' MIP/LP solves |
| SESSION-001 | SESSIONIZE ... PER k | other partitions' sort + gap walk |
| DOWNSAMPLE-001 | DOWNSAMPLE ... PER k | other groups' bucketing |

All five block — they buffer every partition to produce the one you asked for — so the saving is real work, not rows discarded slightly later.

```

Readings := [
user_id	seq	v
u1	1	10

```

```

| u1      | 40 | 30 |
| u2      | 2  | 40 |
];
query { σ user_id = "u1"
      (SESSIONIZE seq GAP 5 PER user_id AS run (Readings)) };

```

becomes `SESSIONIZE seq GAP 5 PER user_id AS run (σ user_id = "u1" (Readings))` — `u2`'s rows are never buffered or sorted. The pushed equality is *removed* from above: after the operator has run over only that partition, every output row satisfies it already.

What is pushable: a top-level conjoined `=` against a constant, on a bare partition column. Inequalities, predicates on a computed output column (a window rank, a session id), equalities on non-partition columns, and disjunctions stay as a residual `σ` above the operator. An operator with no `PER` keys has no partition dimension and is left alone.

`DOWNSAMPLE ... FOR n ROWS` **does not prune**. `FOR n ROWS` keeps the `n` most recent buckets **across all groups**, so filtering after the operator can leave fewer rows than filtering before it. This is a counterexample, not a conservative choice.

A predicate on `DOWNSAMPLE`'s *timestamp* column is a **range** restriction on the buckets rather than a partition selection — a different rewrite, and not one of these.

Endpoint bounds on a graph traversal

```

σ from = c (OP from, to ... (E))   ≡   OP from, to ... {from=c} (E)

```

Three graph operators compute an **all-pairs** answer over an edge relation, and a selection fixing an endpoint to a constant throws almost all of it away. Because the endpoint is a partitioning dimension — the all-pairs answer filtered to `from = X` is what seeding the traversal at `X` alone produces — the equality folds into the operator and the traversal starts where it was going to end up.

| Code | Operator | What is skipped |
|-------------|--------------------|--|
| CLOSURE-001 | CLOSURE / RCLOSURE | reachability from every other node |
| TRACE-001 | TRACE | every other node's optimal-path table |
| PATH-001 | PATH | every other node's breadth-first traversal |

A bound on the **target** is the same traversal over the reversed adjacency, and both bounds together is a single-pair search.

```

Transfers := [
src	dst
a1	a2
a2	a3
b1	b2
];
query { σ src = "a1" (PATH src, dst HOPS 1 TO 3 AS depth (Transfers)) };

```

becomes `PATH src, dst HOPS 1 TO 3 AS depth {src="a1"} (Transfers)` — `b1`'s traversal never runs. That spelling is the one the operator pages recommend, since a start node is a selection rather than syntax, so the unbounded form is what following the manual gives you and this rule is what makes it cheap.

The distance column is unaffected. PATH's `depth` is the *shortest* path length, which is why `(from, to)` is a candidate key of its result; the shortest path from one node does not depend on which other nodes were searched from, so the bounded result is a slice of the unbounded one rather than a recomputation of it.

What is pushable: a top-level conjoined `=` against a literal, on the operator's own `from` or `to` column, where that endpoint is not already bound. A predicate on a computed output column (`depth`, TRACE's `weight` or `path`), an inequality, a second equality on an already-bound endpoint, a cross-column correlation like `from = to`, and any disjunction all stay as a residual σ above the bounded operator — so an unrecognised conjunct is never dropped.

Distinctness rules (DIST-`nnn`)

δ buffers a whole hash set, so a δ that changes nothing is one of the more expensive no-ops a plan can contain. Two rules remove one — they look in opposite directions.

| Code | Rewrite |
|----------|---|
| DIST-001 | Drop a δ whose input is already duplicate-free |
| DIST-002 | Drop a δ whose consumer is a γ that ignores multiplicity |

DIST-001 asks what is below: the output of a γ , of a set operation $(\cup/\cap/-/\Delta)$, of $\div$, of a closure, of $\forall$, or of another δ is already a set, and that property survives the row-subset operators $(\sigma, \tau, \lambda, \text{the left of } \ltimes/\triangleright)$ and renames. A base relation counts too when its declared statistics carry a key.

DIST-002 asks what is above:

$\gamma \text{ keys, aggs } (\delta R) \equiv \gamma \text{ keys, aggs } (R) \quad \text{when no aggregate reads multiplicity}$

An aggregation groups its input, so deduplicating first is work its consumer would have done anyway — **provided** the aggregates give the same answer over a bag as over the set beneath it. MIN and MAX do; a γ with no aggregates at all (pure grouping) does. SUM, COUNT, AVG and COLLECT do not, and ARGMAX/ARGMIN are excluded as well. Every aggregate argument and grouping key must additionally be **deterministic** — the rule changes how many rows the expression is evaluated over, so `MIN(Rand())` is not invariant under it.

The rule looks *through* a σ , π , ρ or τ sitting between the γ and the δ : each emits at most one row per input row, so re-admitting duplicates below it cannot change which rows arrive. A λ , a sample, or a TOP is **not** transparent that way — how many rows they keep depends on how many they are given, which is exactly what the δ changes.

Set-operation identities (SET-`nnn`)

Nothing else in the optimizer compares the **two sides of a binary operator** for structural equality. This family does, and what it finds is queries that ask for work whose answer is already in front of them:

| Code | Rewrite |
|---------|--|
| SET-001 | $R \cup R, R \cap R \rightarrow \delta R; R - R, R \Delta R \rightarrow \emptyset$ |
| SET-002 | $\sigma k(R) \cup R \rightarrow \delta R; \sigma k(R) \cap R \rightarrow \delta \sigma k(R); \sigma k(R) - R \rightarrow \emptyset; R - \sigma k(R) \text{ and } \sigma k(R) \Delta R \rightarrow \delta \sigma (\neg k \vee k \text{ IS UNKNOWN})(R)$ |

| Code | Rewrite |
|---------|---|
| SET-003 | $\rho s(R) \oplus \rho s(Q) \rightarrow \rho s(R \oplus Q)$ for $\oplus$ in $\cup, \cap, -$ Δ $\sqcup$ |
| SET-004 | $\pi c(A) \cup \pi c(B) \rightarrow \delta \pi c(A \cup B)$ |
| SET-005 | $A \times B \cup A \times C \rightarrow \delta (A \times (B \cup C))$; likewise a shared right operand |

The sides are compared with the same structural equivalence the planner uses to find shared sub-expressions. It ignores source positions, so the same expression written out twice in two places matches — which is what makes these fire on queries people actually write rather than on contrived ones.

What sharing already buys, and what it does not

A repeated sub-expression is already **planned once** and read twice, under a spool. That is the expensive half and it needs no rule. What survives is the set operation itself: $\cup$ and $\cap$ each buffer a hash set of one side and probe it with the other, so $R \cup R$ builds a whole-relation hash table in order to discover it already had every row. These rules make that vanish rather than make it cheaper — and once one fires the spool goes too, there being one reader left.

The δ is the rewrite, not decoration

$\cup$, $\cap$ and $-$ de-duplicate. π and $\times$ do not, and neither does a bare relation. So a rewrite that removes a set operation has to say what happened to the de-duplication it was doing:

```
A := [
id	note
1	x
];	
B := [	
id	note
----	-----
1	y
];
query {  $\delta (\pi id (A \cup B))$  };
```

$\pi id (A) \cup \pi id (B)$ is $[1]$; hoisting the π without a δ would give $[1, 1]$, because the union can no longer see that the rows differ. The δ is dropped only where the result is provably duplicate-free, which is the better outcome and not always available.

SET-003 **is the exception, and the reason is worth keeping.** A rename changes no value and drops no column, and the whole-row set operations match rows **positionally and name-blind** — so the set being de-duplicated is identical before and after, and hoisting a ρ moves no de-duplication anywhere. That is also why SET-003 covers $\sqcup$, which the other two arms cannot: with nothing to de-duplicate, a bag union's multiplicities are untouched.

SET-005 factors on the same side only

$A \times B \cup C \times A$ has the common operand at opposite ends. The headings are $A \mathrel{++} B$ and $C \mathrel{++} A$, so factoring A out would permute the **ordered** heading that the positional form of ρ and the whole-row set operations read — the same trap that got a join-commuting rule withdrawn. The rule matches a shared *left* operand or a shared *right* one, and nothing else.

What is deliberately left out

- **A self-join** — $R \bowtie R \rightarrow R$ — is not offered. A natural join's keys are every column and the engine's joins skip NULL keys, so a row with a NULL anywhere would not match itself and would be **dropped**; and a row present twice matches itself four ways. Nullability is not something a heading states, so there is nothing to decide it on. The outer forms inherit the first problem and $R \bowtie R.x = R.x$ inherits it too.
- $\sqcup$ (outer union), whose branches have different headings by construction.

Reading a condition's truth value

Three of these rewrites have to name *the rows a selection did not keep*, and that is not $\neg p$. A selection keeps a row only when its predicate is TRUE, so the rows it drops are those where the predicate is FALSE **or** UNKNOWN — and $\neg \text{UNKNOWN}$ is UNKNOWN, which a selection also drops. The complement is $\neg p \vee p \text{ IS UNKNOWN}$, and the second disjunct is a null test over the condition itself:

```
 $\sigma \neg(\text{amount} > 100) \vee (\text{amount} > 100) = \perp$  (Orders)
```

A parenthesised predicate in operand position reads its **truth value**, which is NULL exactly when the condition is UNKNOWN. So $(\text{amount} > 100) = \perp$ is true for the rows where the amount is missing — the rows $\sigma \text{ amount} > 100$ silently dropped and $\sigma \neg(\text{amount} > 100)$ does not give back.

The optimizer writes this form itself; you rarely need to. It is documented because it is what :opt and --explain will show you for these rules, and because reading $(p) = \perp$ as "p was UNKNOWN" is not obvious the first time.

Volatility

SET-001, SET-002 and SET-005 each collapse two evaluations of one expression into one, so each asks whether that expression is **reproducible** first. $X \Delta X$ over an unseeded SAMPLE is a query *about* two draws, and the answer is not $\emptyset$. SET-003 and SET-004 need no such question — both branches are still evaluated exactly once.

Contradiction and the empty relation (PRED-004..006, EMPTY-*nnn*)

A filter can be unsatisfiable — $\text{amount} > 50 \wedge \text{amount} < 20$ admits no row of any table. Without this family the engine scans, joins and aggregates its way to an empty answer anyway; with it, the query stops being work.

Reading a conjunction as bounds

The comparisons a conjunction places on one column fold into an interval. $>$ and $\geq$ move the lower endpoint, $<$ and $\leq$ the upper, and $=$ sets both. Three things fall out of that one interval:

| Code | Rewrite |
|----------|---|
| PRED-004 | An empty interval means the conjunction is unsatisfiable $\rightarrow$ false |
| PRED-005 | A bound that is not the tightest in its direction is implied , and dropped |
| PRED-006 | A conjunct that repeats an earlier one is dropped |

Bounds are compared **numerically**, so 5 and 5.0 are one value, and within one literal kind only — a **NUMBER** bound and a **DATE** bound on the same column are left alone rather than

compared. Temporal bounds (DATE, TIME, TIMESTAMP, DURATION) participate on the same terms as numbers and strings.

A column's **qualifier is part of its identity**: $A.x > 5 \wedge B.x < 3$ names two different columns and is perfectly satisfiable. $\neq$ is treated as unconstraining — it excises a point from the interior rather than moving an endpoint — so $x \neq 5 \wedge x = 5$ is a contradiction the analysis does not report.

△ PRED-004 **does not fire under** $\neg$

$x > 5 \wedge x < 3$ is FALSE for every non-NULL x and **UNKNOWN** when x is NULL. A filter drops the row either way, so treating it as false is sound there. Under a negation the two part company: $\neg$ FALSE keeps a row where $\neg$ UNKNOWN drops it, so folding the inside of $\neg(x > 5 \wedge x < 3)$ would silently add back exactly the rows where x is NULL. PRED-004 is therefore suppressed in a negated position. PRED-005 and PRED-006 carry no such restriction — dropping an implied or repeated conjunct preserves the three-valued answer exactly.

Propagating the empty relation

| Code | Rewrite |
|-----------|---|
| EMPTY-001 | $\sigma \text{ false } (R) \rightarrow \emptyset$ — the sub-tree is replaced, not just filtered |
| EMPTY-002 | An operator whose output must be empty because an input is $\rightarrow \emptyset$ |
| EMPTY-003 | $X \cup \emptyset \rightarrow X$ |

$\emptyset$ is an internal node with **no rows and the heading of whatever it replaced**, so a query above it still type-checks and still names the right columns. It is not the EMPTY/DUM literal, which is the *zero-column* relation (see [truth relations](#)); $\emptyset$ has to keep a heading, EMPTY has none to keep.

An operator collapses only when an empty input makes its output empty *for certain*:

- **$\sigma, \pi, \rho, \tau, \delta, \lambda, \mu$** — each emits at most one row per input row.
- **γ and $\forall$ only when they group.** A *scalar* aggregate over nothing is one row, not none: $\gamma \text{ COUNT}(*)$ of an empty relation is 0, and a no-key $\forall$ is vacuously true. Collapsing either would delete a row the query must return.
- **Joins** — $\times, \bowtie, \bowtie\emptyset, \circ$ and the interval join: either side. $\bowtie$: either side, since an empty right matches nothing. $\triangleright$ and $\bowtie$: left only. $\bowtie$: right only. $\bowtie$: both. AS-OF and lateral: left only, each being driven by its left input.
- **Set operations** — $\cap$: either side. $-$ and $\div$: left only. $\cup, \sqcup, \sqcup, \Delta$: both.

Three identities that look safe are **not**, and are left out:

- $\emptyset \cup X \rightarrow X$ would rename the result's columns. $\cup$ takes its output schema from its **left** branch, so dropping an empty left branch hands the naming to the right one. The mirror $X \cup \emptyset \rightarrow X$ keeps the left schema *and*, $\cup$ being a bag operation, the left multiplicities — so only that direction ships, as EMPTY-003.
- $X - \emptyset \rightarrow X$ and $\emptyset \cup X \rightarrow X$ would break a *declared* property. $-$ and $\cup$ produce sets, and DIST-001 removes a δ above them on exactly that basis; replacing one with a possibly-duplicated X would make the claim false and the δ would not come back.
- $X \div \emptyset$ is **not empty**. Division by an empty divisor is vacuously satisfied by every row of the quotient's projection, so only the left-empty direction collapses.

What it looks like

```
-- as written
σ amount > 50 ∧ amount < 20 (Orders) ⋈ Customers

-- PRED-004: the conjunction admits no value
σ false (Orders) ⋈ Customers

-- EMPTY-001: the filter's whole sub-tree goes
∅ ⋈ Customers

-- EMPTY-002: a join with an empty side is empty
∅
```

Neither table is read, and `:explain` shows a single `Empty` node.

Rename rules (RENAME-*nnn*)

| Code | Rewrite |
|------------|---|
| RENAME-001 | Collapse a relation-only ρ into the ρ directly beneath it |
| RENAME-002 | Drop a relation-only ρ whose alias nothing in the query names |
| RENAME-003 | Drop rename pairs that give a column the name it already has |
| RENAME-004 | Compose a pair-form ρ with the pair-form ρ beneath it |

Inlining a view V produces $\rho \ V \ (\text{body})$. The wrapper is needed at that moment — it keeps V a resolvable alias, so a qualified $V.\text{col}$ still finds the right side of a join — but once nothing names V it is dead weight that sits between operators later rules need to see adjacent.

```
Orders := [
cust	region	amount
A	west	10
B	east	5
];
Big := { σ amount > 5 (Orders) };
query { σ cust = "A" (Big) };
```

Inlining gives $\sigma \text{ cust} = \text{"A"} \ (\rho \text{ Big} \ (\sigma \text{ amount} > 5 \ (\text{Orders})))$. Dropping the ρ leaves the two selections adjacent, so `SEL-002` merges them into one filter that `SEL-005/pushdown` can then carry to the source as a single `WHERE`.

When it does not fire. A relation-only ρ renames no column, so removing it can only change how a *relation-qualified* reference resolves — and a qualified reference resolves by column provenance, falling back to a bare-name match when the qualifier names nothing. Two conditions therefore have to hold, and the second is the one that is easy to miss:

- nothing references the alias itself (`Big.cust` keeps $\rho \text{ Big}$); **and**
- nothing references a relation name the input would *re-expose*. In $(\rho \ V \ (\text{Orders})) \ ⋈ \text{Orders}$, a reference to `Orders.id` names exactly one column today because the left side is stamped V . Drop the ρ and both sides are `Orders`: the reference turns ambiguous and silently falls back to a first match.

`RENAME-001` needs neither condition. The outer ρ re-anchors every column to its own name, so the inner alias is already unresolvable one level up — collapsing loses a name that was

invisible anyway.

RENAME-003 / RENAME-004 — the column renames

RENAME-001 and RENAME-002 are about the *relation* name. Two more rules tidy the **column** renames:

| | | | |
|---|---------------|---|------------|
| $\rho \text{ (id} \rightarrow \text{id, a} \rightarrow \text{q) (A)}$ | $\rightarrow$ | $\rho \text{ (a} \rightarrow \text{q) (A)}$ | RENAME-003 |
| $\rho \text{ (b} \rightarrow \text{c) (}\rho \text{ (a} \rightarrow \text{b) (R))}$ | $\rightarrow$ | $\rho \text{ (a} \rightarrow \text{c) (R)}$ | RENAME-004 |

Neither changes what a query returns; what they buy is adjacency. A ρ sitting between two operators is a tax every shape-matching rule pays, which is why pushdown into a recursion grew explicit logic to peel renames out of the prefix it matches on.

RENAME-003 drops the pairs unconditionally. Dropping the **node** is a different question — a ρ whose pairs all cancel may still carry a relation name something resolves against — so what it does is reduce the node to its relation-only form and let RENAME-002's reference sweep decide whether it lives. The comparison is exact, not case-insensitive: $a \rightarrow A$ changes the spelling a result's heading is printed with, which is an answer rather than a detail.

RENAME-004 chases each inner pair's target through the outer's sources, so $a \rightarrow b$ under $b \rightarrow c$ becomes $a \rightarrow c$, while an outer pair renaming a pass-through column is carried over as it stands. The merged ρ keeps the outer's relation name where it has one and the inner's otherwise, so nothing a qualified reference could resolve against is lost; where both carry one the inner's goes, for RENAME-001's reason exactly.

It **declines** when the two renames name the same source column. With the inner renaming $a \rightarrow b$, an outer $a \rightarrow c$ renames nothing today — there is no a left — and composing them naively would produce a single ρ carrying both $a \rightarrow b$ and $a \rightarrow c$.

There is deliberately no rule that cancels $a \rightarrow b$, $b \rightarrow a$ *within one* ρ : that is a **swap** of two column names, not a no-op. A rename cycle written as two stacked ρ composes to an identity pair here, which RENAME-003 then drops — so the cycle is handled without a rule that would have deleted a real rename.

Both rules are **pair-form only**. The positional form ($\rho \text{ E (a, b, c)}$) renames by position and is arity-bound, which makes composing it a different question.

How the rules are ordered: phases

The rules are not one flat list. They are grouped into six **phases**, which run once each, in this order:

| Phase | Rules | What it is for |
|----------|--|---|
| simplify | EXPR-001..008, PRED-001..006 | Fold constants and normalise comparisons, so every later rule matches against settled expressions |
| pushdown | SEL-001, JOIN-004, EQ-001, SEL-003..009, NEST-001..003, WINDOW-001, TOPK-001, OPTIMIZE-001, SESSION-001, DOWNSAMPLE-001, JOIN-001..002 | Move filters toward the data and shape the joins |

| Phase | Rules | What it is for |
|---------|---|--|
| cleanup | SEL-002, SEL-010, PROJ-001..003, AGG-001, DIST-001..002, SET-001..005, PROD-001, EMPTY-001..003, SORT-001 | Put conjunctions back together and drop what a rewrite made redundant |
| sip | CLOSURE-001, TRACE-001, PATH-001, FIX-001, GEN-001 | Fold a constraint into an expensive operator, once the σ above it has settled |
| limit | LIM-001..004 | Move limits down, and fuse $\lambda \circ \tau$ into TOP |
| prune | PROJ-004 | Narrow every leaf to the columns the query reads |

Within a phase the rules are **swept repeatedly** until a sweep changes nothing, so one rule's rewrite can expose the pattern another matches on. Across phases there is no loop — and that separation is load-bearing, not tidiness:

- SEL-003 pushes σ below π and PROJ-003 pushes π below σ ;
- SEL-001 splits a conjunction and SEL-002 merges it back;
- SEL-009 distributes a σ over a set operation and SEL-010 merges one back.

Each pair is deliberately split across two phases. Iterating either pair together would never terminate — it would just trade the two operators' positions forever. Sweeps are additionally capped, so even a rule pair that misbehaves costs bounded work rather than hanging the optimizer.

Phase order is also why :opt lists rules in the order it does: a SEL-001 split always precedes the SEL-002 merge that undoes it, and PROJ-004 is always last.

Six rules run *before* the phases, in a per-query preamble, and all six for the same reason — each needs the symbol table, which the phases deliberately do without: INLINE-001 expands the views, RENAME-001..RENAME-004 clear up after it, and LATERAL-001 resolves a TVF and classifies its body. They sit outside the loop because they run before schemas are re-inferred for the expanded tree.

The WHY barrier covers them too. A view referenced under a WHY is left as a reference rather than inlined, so :opt reports no INLINE-001 for it: WHY reifies lineage against the base tables and expands the view itself, which is how the result names the source rows instead of the view's own output.

Shared sub-expressions

Everything above rewrites the *logical* tree. One decision that changes how much work a query does is made later, when the physical plan is built: whether a sub-expression that is read in more than one place is **evaluated** more than once.

A plan is a tree, so by default it is. Two things put a sub-expression in more than one place — the query itself, and the engine's own desugaring — and both are handled the same way: plan it once, mark it as shared, and let every reader read that one plan.

Symmetric difference is the clearest case of the second kind, because the duplication is not something you wrote: $A \Delta B$ is computed as $(A - B) \cup (B - A)$, which reads both A and B twice. :explain shows the mark:

```
Events := [| order_id | customer | event |
           | 1       | acme     | shipped |
           | 2       | globex | shipped |
           | 2       | globex | invoiced |
           | 3       | initech | invoiced |];

Shipped := {  $\pi$  order_id, customer ( $\sigma$  event = "shipped" (Events)) };
Invoiced := {  $\pi$  order_id, customer ( $\sigma$  event = "invoiced" (Events)) };

query { Shipped  $\Delta$  Invoiced };
```

```
SetOp UNION ~4 rows
├ SetOp DIFFERENCE ~? rows
│   ├── Spool #1 ~2 rows
│   │   └ Project ~2 rows
│   │       └ Select ~2 rows
│   │           └ Scan Events ~4 rows
│   └ Spool #2 ~2 rows
│       └ Project ~2 rows
│           └ Select ~2 rows
│               └ Scan Events ~4 rows
└ SetOp DIFFERENCE ~? rows
    ├── Spool #2 (shared) ~2 rows
    └ Spool #1 (shared) ~2 rows
```

Each input appears under a **Spool** with an id, and the second branch reads the **same** two spools rather than sub-plans of its own — **(shared)** marks the reader that does not own the sub-tree, which is why the sub-tree is drawn only once. Evaluating each input twice becomes evaluating it once.

The inputs are **views** here for a reason: evaluating one means evaluating its body, which is work worth doing once. Two inline tables in their place would show no spool at all — their rows are in memory already, so reading them a second time repeats nothing. That rule is stated in full below.

Repeats you wrote

The planner also looks for sub-expressions the *query* reads more than once, and recognises two of them by their **structure** rather than by name or position — so the same expression written out twice in two different places is one sub-expression, and a view referenced twice is planned once rather than expanded twice:

```
Orders := [| order_id | customer | amount | status |
           | 1       | acme     | 100   | shipped |
           | 2       | globex   | 250   | pending |
           | 3       | acme     | 75    | shipped |];

Large := {  $\sigma$  amount > 80 (Orders) };

query { ( $\gamma$  customer, SUM(amount)  $\rightarrow$  total (Large))  $\bowtie$  ( $\pi$  customer, status (Large)) };
```

```
Join NATURAL/HASH build=RIGHT ~1 rows
├ Aggregate ~1 rows
│   └ Spool #1 ~1 rows
│       └ Select ~1 rows
│           └ Scan Orders ~3 rows
└ Project ~1 rows
    └ Spool #1 (shared) ~1 rows
```

Large is written twice and planned once: the selection appears under a single spool, and the projection on the other side of the join reads it.

What counts is **how many places evaluate it**, not how many times it appears. In $(\gamma \dots (\sigma \dots R)) \cup (\gamma \dots (\sigma \dots R))$ the σ is written twice, but sharing the γ above it already reduces the σ to a single evaluation — holding its rows as well would keep a second copy for one reader. So only the γ is shared. Write that same σ a third time *outside* both γ s and it has a reader of its own: then both are shared, and the γ 's spool reads the σ 's.

A sub-expression whose second evaluation would repeat no work is left alone: the `UNIT` and `EMPTY` literals, and a reference to an inline or `relix.*` relation, whose rows are in memory already. A **view** reference is not one of these — evaluating it means evaluating its body — and neither is a file or database relation, where evaluating it again is another read of the source.

Repeats the iteration makes

A `FIX` step runs once per iteration — that is what the operator does — so a sub-expression inside a step is evaluated as many times as the recursion runs, however many times it is *written*. Where it does not name the recursion, every one of those evaluations produces the same rows, so it is shared exactly like a repeat you wrote out twice:

```
Edges := [| src | dst |
          | 1  | 2  |
          | 2  | 3  |
          | 3  | 4  |];
Seed := [| src | dst |
         | 1  | 2  |];
```

```
query { FIX Reach (Seed,  $\pi$  src, dst2  $\rightarrow$  dst (Reach  $\bowtie$  ( $\pi$  src  $\rightarrow$  dst, dst  $\rightarrow$  dst2 (Edges)))) };
```

```
FIX Reach ~? rows
├─ Scan Seed ~1 rows
└─ Project ~? rows
   └─ Join NATURAL/HASH build=RIGHT ~? rows
      ├─ REF Reach ~? rows
      └─ Spool #1 ~3 rows
         └─ Project ~3 rows
            └─ Scan Edges ~3 rows
```

The edge side is written once and spooled anyway: the join reads it on every round, and reads the same rows every time, so one evaluation serves the whole recursion — a table scanned once instead of once per round. The seed is not shared: it runs before the first round and never again. Neither is the join, nor the projection above it; both read `Reach`, whose rows are the ones that change.

That the join is re-run every round is why sharing the rows is not the whole of it. A join over an unchanging side still has to **build its lookup table** from those rows, and it was doing so once per iteration — the rows were free by then, the work over them was not. A build side the planner has spooled is prepared once and reused, so what a round costs is the part of it that actually differs: the newly-derived rows, probed against a table built before the first round.

What it costs

Sharing means holding rows while another reader still needs them, so the held rows grow only as far as the readers actually read. Two readers that stop after five rows between them cost five rows, not the whole relation; a reader that drains the sub-expression — a grouping, a set operation, anything that must see every row — fills it completely, as it always did.

That matters beyond memory: it is why sharing a sub-expression never costs more work than repeating it. The rows one reader pulls are the rows the next one reads, so the total is what the hungriest reader wanted, never the sum.

Holding is bounded, by a fixed number of rows across one query. A sub-expression whose readers want more than that stops being held: the reader that outruns it carries on by itself, and any other re-computes the part it has not already seen — which costs exactly what not sharing would have cost, and returns the same rows, because only a reproducible sub-expression is shared in the first place. The budget is first-come-first-served, so one very large shared sub-expression can use it up and leave a later one in the same query unshared.

Some sub-expressions are not shared at all, because their two evaluations are entitled to disagree — and where one of these sits above a sub-expression that *can* be shared, the one underneath still is:

- **It reads system state.** `Rand()` and an unseeded `SAMPLE` draw afresh each time, and `X Δ X` over one of them is a query *about* that difference. Replaying a single draw would answer something else.
- **It reads a recursive relation from the `FIX` around it.** A recursive reference means the current iteration's rows, so a buffer filled on the first iteration would still be answering for the first iteration on the tenth. The rest of the step is unaffected: a sub-expression that does not name the recursion is the same in every iteration, so it is computed once for all of them.
- **It is a table-valued function body under a `LATERAL join`.** That body is planned per left row, and its rows belong to that row.

Sharing never changes an answer, and never changes the order of one. A spooled sub-expression emits the same rows, in the same order, under the same schema as the sub-expression it wraps, so nothing above it — including the operators that read their inputs by position — can tell the difference.

Examples

Filtering one region out of a per-region rollout

```
Sales := [
rep	region	amount
ana	west	300
ben	west	120
cleo	east	90
dara	east	400
eli	north	50
];

query { σ region = "west" (γ region, SUM(amount) → total (Sales)) };
```

Written that way the engine would group **all five** rows into three groups, then throw two groups away. SEL-007 rewrites it to

```
γ region, SUM(amount) → total (σ region = "west" (Sales))
```

so only the two **west** rows are ever grouped. The answer is identical:

| region | total |
|--------|-------|
| west | 420 |

Over a JDBC source the difference is visible in the SQL that reaches the database — **WHERE** `region = 'west'` **GROUP BY** `region` instead of **GROUP BY** `region` followed by an in-engine filter.

A DISTINCT the aggregation above it makes pointless

```
Readings := [
sensor	site	celsius
s1	north	12
s1	north	12
s2	north	9
s3	south	21
s3	south	21
];
```

```
query { γ site, MIN(celsius) → coldest (δ (Readings)) };
```

The **δ** was written to "clean up" the duplicate readings, and it costs a full hash set over the input. **DIST-002** removes it: the **γ** below groups the rows anyway, and **MIN** over {12, 12, 9} is the same as **MIN** over {12, 9}.

```
γ site, MIN(celsius) → coldest (Readings)
```

| site | coldest |
|-------|---------|
| north | 9 |
| south | 21 |

Change **MIN** to **COUNT** and the rule stops firing, because now the duplicates are the answer: **COUNT** over the deduplicated rows says north has 2 readings, over the raw rows 3. The rewrite is not "the **δ** looks unnecessary" — it is "this consumer cannot tell the difference".

A mixed condition: half demoted. half kept

```
Sales := [
rep	region	amount
ana	west	300
ben	west	120
cleo	east	90
dara	east	400
];
```

```
query { σ region ≠ "north" ∧ total > 200 (γ region, SUM(amount) → total (Sales)) };
```

`region ≠ "north"` is a key condition and moves below the **γ**; `total > 200` reads an aggregate and cannot. The optimized shape is

```
σ total > 200 (γ region, SUM(amount) → total (σ region ≠ "north" (Sales)))
```


which is exactly the `WHERE ... GROUP BY ... HAVING` split SQL would write by hand.

| region | total |
|--------|-------|
| west | 420 |
| east | 490 |

Filtering one branch of a difference

```
Active := [  
id	status
1	live
2	live
3	draft
];
```

```
Retired := [  
id	status
2	live
];
```

```
query { σ status = "live" (Active - Retired) };
```

SEL-009 pushes the filter into the **left** branch only:

```
(σ status = "live" (Active)) - Retired
```

Rows 1 and 2 survive the filter, and `Retired` removes row 2 — leaving row 1. Pushing the same filter into `Retired` as well would have been wrong in a way that is easy to miss: `Retired`'s only row *does* satisfy `status = "live"` here, but if it did not, the filter would have emptied the subtrahend and row 2 would have come back.

| id | status |
|----|--------|
| 1 | live |

Top three by amount, without sorting everything

```
Sales := [  
rep	region	amount
ana	west	300
ben	west	120
cleo	east	90
dara	east	400
eli	north	50
];
```

```
query { λ 3 (τ amount DESC (Sales)) };
```

LIM-003 fuses the pair into a keyless TOP, so the executor keeps a three-element heap instead of sorting all five rows (and all five million, on a real table):

```
TOP 3 amount DESC (Sales)
```

| rep | region | amount |
|------|--------|--------|
| dara | east | 400 |
| ana | west | 300 |
| ben | west | 120 |

:opt shows the firing:

LIM-003 limit over sort fused into TOP 3 (bounded heap instead of a full sort)

Reading two columns of a six-column table

```
Orders := [
order_id	cust	region	amount	placed_by	note
1	ana	west	300	kim	rush
2	ben	west	120	kim	gift wrap
3	cleo	east	90	raj	fragile
4	dara	east	400	raj	signature
5	eli	north	50	kim	none
];
```

```
query { γ region, SUM(amount) → total (σ amount > 60 (Orders)) };
```

The query names exactly two columns — `region` groups, `amount` is filtered and summed. `order_id`, `cust`, `placed_by` and `note` are carried through the selection and into the blocking `γ` purely to be discarded. PROJ-004 rewrites it to

```
γ region, SUM(amount) → total (σ amount > 60 (π region, amount (Orders)))
```

| region | total |
|--------|-------|
| west | 420 |
| east | 490 |

:opt names the relation and the width it shed:

PROJ-004 columns pruned at Orders: 2 of 6 read

Over a JDBC source that is the difference between `SELECT order_id, cust, region, amount, placed_by, note FROM orders` and `SELECT region, amount FROM orders` — four columns × every row that never leaves the database.

Both sides of a join, narrowed

```
Orders := [
order_id	cust	amount	note
1	ana	300	rush
2	ben	120	gift wrap
3	cleo	90	fragile
];

Customers := [
cust	city	tier	signup
ana	leeds	1	2024-01-04
ben	york	2	2024-03-11
cleo	hull	1	2025-02-20
];

query {  $\pi$  amount, city (Orders  $\bowtie$  Customers) };
```

The result reads two columns, one from each side, so the requirement splits by side and each leaf is pruned independently:

```
 $\pi$  amount, city (( $\pi$  cust, amount (Orders))  $\bowtie$  ( $\pi$  cust, city (Customers)))
```

Note what survived on **both** sides: `cust`. Nothing above the join reads it — but it is the natural join's key, and dropping it would have turned a join on `cust` into a Cartesian product. That is why `$\bowtie$` keeps every common column whether or not anything reads it.

| amount | city |
|--------|-------|
| 300 | leeds |
| 120 | york |
| 90 | hull |

The hash join now builds its index over two-column rows instead of four-column ones, and over two JDBC tables on one connection the whole thing is still a single `SELECT ... JOIN ... ON` — just a narrower one.

See Also

- [Aggregation \(\$\gamma\$ \)](#)
- [Selection \(\$\sigma\$ \)](#)
- [Limit \(\$\lambda\$ \)](#) · [Top-K per group](#)
- Interactive REPL — `:opt` shows the report for a single query

Pushdown (what the backend computes)

Syntax

```
relx --explain script.relix      -- the physical plan, with any pushed-down query
relx --trace script.relix        -- a live feed including the planner's decisions

-- in the REPL
:explain <query>                 -- the plan for one query
```

Description

Relix does not fetch a table and then filter it. Where the source is capable, the engine folds as much of the query as it can into the source's own language — a SQL `SELECT` for a JDBC connection, an aggregation pipeline for MongoDB — and computes only the remainder itself. That fold is called **pushdown**, and it is the difference between a filter that runs in the database and a filter that runs after a million rows have crossed the network.

Nothing is lost when a fold fails. The answer is identical either way; what changes is where the work happens and how much data moves. So pushdown is never a correctness question, only a performance one — and the plan is where you read the answer:

```
Select                                     ← computed by the engine
└─ PushedScan [jdbc/warehouse] SELECT ... WHERE ...      ← computed by the database
```

Read a plan as a boundary. Everything below a `PushedScan` happened in the backend; everything above it happened in the engine, in this process's memory.

Two properties of that boundary are worth internalising, because together they explain almost every plan that surprises someone. Pushdown is **contiguous** — it folds a *sub-tree* that sits directly over one connection's tables, so a single operator the backend cannot express stops the fold at that point, and everything above it stays in the engine even if the backend could have done it. And it is **per connection** — two connections are two backends that know nothing of each other, so a join across them is always computed here, over whatever each side returns.

Technical Description

Which sources push down at all

| Source | Folds |
|--|--|
| connection ... from jdbc (and database) | a single SQL <code>SELECT</code> |
| connection ... from mongodb | one aggregation pipeline |
| csv, json, http, generator, inline tables, rows supplied by a host program | nothing — the whole query runs in the engine |

The second row is not a shortcoming to work around. A CSV file has no query engine, so the engine reading the file *is* the only place the work can happen.

The clause-order rule (SQL)

A pushed SQL query is one `SELECT`, so the operators fold in the order that statement's clauses run:

```
SELECT [DISTINCT] <select list> FROM ... [JOIN ...] WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT ...
```

An operator folds only if its clause comes *after* everything already folded. This is the single mechanical rule behind most of the table below: a σ folds onto a scan, but not on top of a γ , because `WHERE` cannot follow `GROUP BY` — that one needs a sub-query, and the engine computes it instead. Likewise a projection or a grouping **fixes the select list**, so nothing that needs the table's raw columns folds above it, and only one `ORDER BY` and one `LIMIT` can fold.

Reordering is the optimizer's job, not the renderer's, and it happens first: a σ the optimizer pushes below a γ arrives at the renderer already in a position where it can fold. That is why the two features compound — see [optimizer](#).

SQL: which operators fold

| Operator | Folds as | Does not fold when |
|-----------------------------|---|--|
| relation over a table | <code>FROM table</code> | the source is not a connection-backed table |
| σ selection | <code>WHERE</code> , or <code>HAVING</code> when a γ is already folded | a projection, sort, limit or window is already folded, or the predicate has an untranslatable part |
| π projection | the select list | a projection or a grouping is already folded; a <i>column-dropping</i> π keeps the scan foldable, a computing or renaming one fixes the list |
| δ distinct | <code>SELECT DISTINCT</code> | a <code>GROUP BY</code> , sort or limit is already folded |
| γ aggregation | <code>GROUP BY</code> | any grouping key is not a bare column — a computed key such as <code>YEAR(ts)</code> , or a renamed one — or an aggregate has no SQL spelling |
| $\forall$ universal | <code>GROUP BY ... HAVING</code> | there are no grouping keys |
| τ sort | <code>ORDER BY</code> | a projection, sort or limit is already folded, or a sort key is not a bare column. Above a γ the key may be one of its output columns |
| λ limit | <code>LIMIT/OFFSET</code> , or <code>OFFSET ... FETCH</code> on SQL Server and Db2 | a limit is already folded, or — on SQL Server, whose <code>OFFSET ... FETCH</code> is legal only after an <code>ORDER BY</code> — nothing below it is sorted |
| <code>TOP</code> | <code>ORDER BY ... LIMIT</code> | it is per-group (<code>PER ...</code>) — only the whole-relation form folds |
| $\bowtie$ theta join | <code>JOIN ... ON</code> | either side is not a bare table scan, the two sides are on different connections or are the same relation (a self-join), or the condition has an untranslatable part |
| $\bowtie$ natural join | <code>JOIN ... ON</code> equating the shared columns | either side is not a bare table scan, the two sides are on different connections or are the same relation, or a shared column has a different type on each side |
| ASOF join | <code>LEFT JOIN LATERAL (... ORDER BY ... LIMIT 1) ON TRUE</code> , or <code>JOIN LATERAL</code> for the INNER form | the dialect is not PostgreSQL, DuckDB, Db2 or SQL Server (which spells it <code>OUTER APPLY (SELECT TOP 1 ...)</code>), or the join carries a <code>WITHIN</code> tolerance |
| <code>IJOIN</code> interval | <code>JOIN ... ON</code> over the endpoints | either side is not a bare scan, or they are on different connections |

| Operator | Folds as | Does not fold when |
|------------------|--------------------------------------|--|
| WINDOW / ROLLING | OVER (PARTITION BY ... ORDER BY ...) | the dialect is MySQL, or a projection, grouping or limit is already folded |

A join folds the same way whether its tables are declared sources or are named directly as `connection.table`. The generated statement names each side by an alias of its own: the relation's name when that is a plain identifier, otherwise the table's name, and `t0/t1` when neither is. A condition still refers to a side by its relation name, so over two undeclared tables

```
connection shop from database { url: "jdbc:h2:mem:shop" };
query { shop.customers ⋈ shop.customers.customer_id = shop.orders.customer_id shop.orders };
```

folds to one statement:

```
SELECT customers.customer_id, customers.name, orders.order_id, orders.customer_id, orders.amount
FROM customers customers JOIN orders orders ON (customers.customer_id = orders.customer_id)
```

Above a γ

A grouping does not end the statement. A σ above one becomes its `HAVING`, a τ becomes an `ORDER BY` over the aggregate, and the λ above that becomes the `LIMIT` — so the ordinary shape of a report reaches the database whole:

```
query {
  λ 10 (τ revenue DESC, region ASC
    (σ revenue > 1000
      (γ region, SUM(amount) → revenue (Orders))))
};
```

folds to one statement, and ten rows come back rather than one per region:

```
SELECT region, SUM(amount) FROM orders
GROUP BY region
HAVING (SUM(amount) > 1000)
ORDER BY (SUM(amount) IS NULL) ASC, SUM(amount) DESC, (region IS NULL) ASC, region ASC
LIMIT 10
```

Note what the aggregate looks like in the `HAVING` and `ORDER BY`: the expression is repeated rather than referred to by its `rel` name. Standard SQL does not let a `HAVING` clause name a select-list alias, and `ORDER BY` accepts one only as a bare column reference — not inside the expression that places NULLs where the engine places them. Repeating it is legal everywhere, so it is what every dialect gets.

A π above a γ is the one that does not fold: it would have to rewrite the select list the grouping fixed, and the projection happens in the engine instead.

Every other operator is computed by the engine. That includes the outer joins `⋈⋈⋈`, the semi and anti joins `⋈⋈`, every set operation (`∪ ∩ ∪ ∩ − Δ × ÷ ∘`), `μ` `unnest`, the recursion and graph operators (`CLOSURE`, `CLUSTER`, `PATH`, `FIX`, `TRACE`), `PIVOT/UNPIVOT`, `TREE`, `SESSIONIZE`, `DOWNSAMPLE`, `SAMPLE`, `COVER`, `SOLVE`, `OPTIMIZE` and `ω` `WHY`. An outer join over two tables of one database is therefore an engine join over two pushed scans — each side's filters still fold, and the join itself does not.

SQL: which expressions fold

Inside a folded operator, an expression folds if every part of it does.

| Folds | Does not fold |
|---|---|
| column references | struct {...} and array [...] construction |
| number, string, boolean literals | a predicate used as a value |
| DATE, TIME, TIMESTAMP literals, on every dialect but SQLite | temporal literals on SQLite, which has no date type |
| DURATION literals, on PostgreSQL and DuckDB | DURATION literals on other dialects |
| = < > <= >=, ^ v ~, IS [NOT] NULL, IN, LIKE | |
| + - * and unary minus | division / |

A part that does not fold does not merely omit itself — it stops the whole operator folding, and everything above that operator with it.

Division is computed here, on every backend. A `relax /` is exact decimal arithmetic to ten places, rounded half-up, and no backend's `/` answers that same question. H2 and PostgreSQL divide two integers as integers, so `81 / 2` is `40` there and `40.5` here — a filter over it keeps different rows, not a differently-rounded number. MySQL does produce a decimal and still differs: its scale is the operand's plus four by default, so `amount / 3` comes back `33.3333` against `33.3333333333`. Division by zero disagrees a third way, the engine and PostgreSQL raising where MySQL answers `NULL`.

A cast would settle the first of those and not the others, because a backend's decimal scale is a configuration rather than a spelling. So a filter or projection containing a division reads its rows and finishes here.

Where that matters on a large table, the lever is to filter on something that folds and divide above it. Multiplication is exact on every backend, so a threshold on a ratio can often be written as one on the numerator instead — though the two are the same question only where the divisor's sign is known and it is never zero, since multiplying across a comparison turns it around for a negative divisor, and the division would have raised where the multiplication has an answer.

SQL: which functions fold

This is the sharp edge, and it is worth knowing before you write a filter over a large table. Of the built-in scalar functions, **twenty-one** have a SQL spelling:

| Function | SQL |
|--|---|
| YEAR, MONTH, DAY, HOUR, MINUTE, SECOND | <code>EXTRACT(unit FROM e); DATEPART(unit, SWITCHOFFSET(e, '+00:00'))</code> on SQL Server; SQLite declines them |
| DATE_TRUNC | <code>date_trunc(...)</code> on PostgreSQL, DuckDB and Db2, <code>CAST(DATE_FORMAT(...) AS DATETIME)</code> on MySQL; the generic dialect, SQLite and SQL Server decline it |
| Abs, Int, Ceil, Sgn, Round | <code>ABS, FLOOR, CEILING, SIGN, ROUND (ROUND(e, 0)</code> on SQL Server); SQLite, which rounds in floating point, declines Int, Ceil and Round |

| Function | SQL |
|--|--|
| Fix | TRUNCATE(e, 0) on MySQL, TRUNC(e) on PostgreSQL, DuckDB and Db2, ROUND(e, 0, 1) on SQL Server; the generic dialect and SQLite decline it |
| Coalesce, Nz
(two-argument
form) | COALESCE(...) |
| IsNull | (e IS NULL); (CASE WHEN e IS NULL THEN 1 ELSE 0 END) on SQL Server |
| Replace | REPLACE(s, find, replacement), the string under a binary collation on SQL Server |
| Len, Left, Right,
Mid | CHAR_LENGTH, LEFT, RIGHT, SUBSTRING on MySQL, PostgreSQL and DuckDB ; LENGTH and SUBSTR on SQLite ; CHARACTER_LENGTH and SUBSTRING in CODEUNITS32 on Db2 ; nowhere else |

Everything else is evaluated in the engine, and two families are worth knowing by name because SQL has a same-named function for every one of them and answers differently:

- **Most of the string library** — UCase, LCase, Trim, LTrim, RTrim, InStr, Chr, Asc. Case mapping belongs to the column's collation rather than to the function: MySQL upper-cases Straße to STRASSE where relix produces STRASSE. Whitespace is not one set: SQL TRIM removes spaces where relix removes every Unicode whitespace character, so a tab-padded value comes back trimmed from the engine and untrimmed from the database. And searching is collation-dependent, so SQL's LOCATE finds a case-insensitive match where relix finds none.
- **The transcendental numerics** — Sqr, Log, Exp, Power, Sin, Cos, Tan, Atn. A database computes these to within about one unit in the last place, and so does the engine, and they are entitled to land on either side of it: MySQL answers TAN(4) with 1.1578212823495775 where relix answers 1.1578212823495777. Four of them additionally have no answer for some arguments — relix rejects a negative square root and a non-positive logarithm, and cannot represent the infinity an overflowing Exp produces, where SQL returns NULL. A query that succeeds pushed and fails in the engine is worse than one that disagrees in the last digit, and both are reasons to compute here.

In every case the same query would answer differently depending on where the planner happened to put the work, and that is the one thing a spelling must never cause. A slower plan is the deliberate trade.

Len, Left, Right and Mid are the four that are offered to **named** dialects rather than to SQL at large, and the reason is worth knowing because it is not about the functions. relix counts and slices a string in **code points**, and so do MySQL, PostgreSQL, DuckDB, SQLite and Db2 — but H2, which resolves to the generic dialect, counts UTF-16 code units, so its CHAR_LENGTH and LEFT are different functions wearing familiar names and disagree on any text holding a character outside the basic multilingual plane. "SQL counts characters" is exactly the kind of claim that survives being read and not being run, so each dialect on that list was put there by running it against a fixture holding an astral character.

A call is offered to a backend only when it is **deterministic**. A function whose value depends on when it runs — NOW, CURRENT_DATE, CURRENT_TIME, Rand — is never evaluated by the database, because it would answer from its own clock and its own random source rather than from the ones this session was given.

For a clock call the engine does something better than declining: it evaluates the call **once for the run** and pushes the resulting literal. `σ at < NOW()` reaches the database as a comparison against a timestamp, so it filters at the source, and the instant in it is the session's. This is sound because such a function declares itself *stable* — one value for a whole run — and because a run reads its clock once, so the value sent to the database is the same value the unfolded half of the same query computes. `Rand` declares nothing, which means its value may differ per row: there is no single value to send, so it stays in the engine.

Like everything else here it is a property of the call rather than a list of names: a function library of your own gets the same treatment from the properties its signatures declare.

The cost is worth pricing. A `σ` over `NOW()` cannot fold, so it reads its input and filters here. Where that matters, compute the instant in the host program and compare against a `TIMESTAMP` literal, which folds and means exactly what it says.

Every other built-in — the whole string family (`UCase`, `Trim`, `Left`, `Replace`, `Len`, ...), the whole math family (`Abs`, `Round`, `Power`, ...), the conversions (`CStr`, `CInt`, `CDBl`), the conditionals (`IIf`, `Nz`, `Coalesce`), and the type checks — is evaluated by the engine. So `σ UCase(name) = 'ADA' (Customers)` reads the whole table and filters it here, while `σ name = 'Ada' (Customers)` filters in the database. When a filter over a large table has to be case-insensitive, consider a source-side column or a comparison the backend can spell.

A function library of your own may supply a spelling for its functions, and the engine folds those exactly as it folds these.

Of the aggregates, `SUM`, `COUNT`, `MIN` and `MAX` fold. `COLLECT`, `ARGMAX` and `ARGMIN` are computed by the engine, so a `γ` using one of them keeps its whole grouping here.

AVG is computed here too, for the reason division is. It is the only aggregate that divides, and a *relix* average is exact decimal to ten places rounded half-up, which no backend's `AVG` carries: asked for the mean of three values summing to 457, the engine answers `152.3333333333` and H2 answers `152.33333333333334`. A `γ` or a `ROLLING` over `AVG` therefore reads its rows and finishes here. Where that matters on a large table, `SUM` and `COUNT` both fold and the division above them is one row's worth of work — and the rounding then becomes yours to choose rather than the backend's.

SQL dialects

The dialect comes from the connection's `dialect:` property, or from its JDBC URL when that property is absent.

`generic` is an assumption, not a description

Every other dialect names a database whose behaviour *relix* has been checked against. `generic` is what an *unrecognised* backend gets, and there is nothing to check it against — so what it renders is the portable spelling plus an assumption that the backend behaves in the common way. Mostly it does. Where it does not, a pushed query can answer differently from the same query run in the engine, and these are the two known places:

- **String ordering.** *relix* orders strings by code point. A backend that orders by UTF-8 bytes or by a binary collation agrees. One that orders by a locale collation does not — PostgreSQL under `en_US.UTF-8` sorts `'a'` before `'B'`, where *relix* sorts every uppercase letter first. H2 orders by UTF-16 code unit, which agrees with *relix* except when a

character outside the basic multilingual plane is compared with one in U+E000..U+FFFF.

- **String equality.** `relix` compares strings exactly. A backend whose columns carry a case- or accent-insensitive collation does not, and an unrecognised URL is exactly the case where `relix` cannot tell.

Both are narrow in practice and neither touches 8-bit text: for ASCII, Latin-1, or any data inside the basic multilingual plane, code point and code unit are the same number and the orders agree exactly. The difference needs both an unusual backend and unusual data.

Two things make it avoidable when it matters. Naming the dialect (`dialect: mysql`, `dialect: postgres`, `dialect: duckdb`, `dialect: sqlite`, `dialect: sqlserver`, `dialect: db2`) moves the connection off generic and onto a checked answer. Declaring collation: `database` tells `relix` to make no assumption about string comparison at all, at the cost of computing those in the engine — see [connection](#).

| Dialect | Identifiers | Notable |
|-------------------|---|--|
| generic (default) | unquoted, unless not a plain identifier | no <code>DATE_TRUNC</code> ; relies on the backend folding unquoted identifiers, and on its string comparison being the common one — see below |
| postgres | "quoted" | with DuckDB, the only dialects that fold an ASOF join, <code>DURATION</code> literals and <code>NULLS LAST</code> ; string <i>ordering</i> is collated, string equality is not |
| mysql | `quoted` | no window functions, so those are computed in the engine; <code>DATE_TRUNC</code> folds through <code>DATE_FORMAT</code> ; string comparison and ordering are collated |
| db2 | unquoted, unless not a plain identifier | Db2 11.5 or later; PostgreSQL's SQL for <code>NULLS LAST</code> , <code>date_trunc</code> and an ASOF join, <code>FETCH FIRST</code> for a limit; strings compare and order exactly in a UTF-8 database — see below |
| sqlserver | [bracketed] | <code>OFFSET ... FETCH</code> , so a limit folds only over a sort; <code>NULLs</code> placed with <code>CASE</code> ; ASOF through <code>APPLY</code> ; no <code>DURATION</code> , <code>DATE_TRUNC</code> or string counting; string comparison and ordering are collated — see below |
| sqlite | "quoted" | no date or time types, so no temporal literal, <code>EXTRACT</code> or <code>DATE_TRUNC</code> folds; <code>LIKE</code> is sent as <code>GLOB</code> ; rounding runs in the engine; strings compare and order exactly |
| duckdb | "quoted" | PostgreSQL's SQL — <code>NULLS LAST</code> , <code>date_trunc</code> , an ASOF join through <code>LATERAL</code> , window functions — with a binary default collation, so neither string comparison nor string ordering is collated |

Each dialect writes a table name one part at a time, so a schema-qualified table: `"sales.orders"` becomes `"sales"."orders"` on PostgreSQL. On generic, a name that is a plain identifier (letters, digits and `_`, not starting with a digit) is written bare, and any other name is double-quoted exactly as written. A table created as `"order-lines"` is therefore read as `"order-lines"`, and its name must match the stored name's case.

DuckDB, and the file it names

DuckDB runs inside the process that opens it, so a DuckDB connection names a database *file* rather than a server:

```
connection lake from database { url: "jdbc:duckdb:/data/lake.duckdb" };
```

A bare `jdbc:duckdb:`, DuckDB's in-memory form, is private to the one JDBC connection that opened it. `relix` opens connections of its own, so each would see a fresh, empty database — name a file. The dialect is recognised from the `jdbc:duckdb:` prefix, or can be named with `dialect: duckdb`. Its SQL is PostgreSQL's in every place `relix` renders differently, and its default collation is binary, so a DuckDB connection folds every string comparison and ordering as written.

SQLite, and the dates it does not have

SQLite has no date or time types. A column declared `DATETIME` holds whatever the application wrote into it — text, a Julian day number, seconds since the epoch — and a comparison against it compares that text or number. A `TIMESTAMP '...'` sent as a string would agree with `relix` only while every stored value was written in exactly the form the literal is, which nothing `relix` can see reports. So on a SQLite connection no temporal literal, `EXTRACT` or `DATE_TRUNC` is folded: those predicates read their rows and are decided in the engine, over the values the driver read, where they mean what they say whatever the storage convention.

Two things still fold, and both rest on one assumption worth knowing: that a column holds its dates in *one* form. Sorting on a temporal column, and comparing two of them, compare the stored text or numbers directly — which is right for a column written consistently, as SQLite's own date functions also assume, and wrong for one that mixes forms.

SQLite's `LIKE` ignores the case of ASCII letters, and no collation changes that, so a pattern is sent as the case-sensitive `GLOB` instead — see [like](#). Its `ROUND`, `FLOOR` and `CEILING` answer in binary floating point, so `Round`, `Int` and `Ceil` are computed here. It has no `LATERAL`, so an `ASOF` join runs in the engine. A SQLite connection, like a DuckDB one, names a file: `jdbc:sqlite::memory:` is private to the JDBC connection that opened it.

SQL Server, and the clauses it spells otherwise

SQL Server is recognised from a `jdbc:sqlserver:` URL, or named with `dialect: sqlserver`. Most of what `relix` renders it spells otherwise, and three of the differences change *what* folds rather than how it is written:

- **A limit needs a sort.** T-SQL has no `LIMIT`; its `OFFSET ... FETCH` is legal only after an `ORDER BY`. So `λ 10 (τ oid (Orders))` folds, into `ORDER BY ... OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY`, and `λ 10 (Orders)` does not — the rows are read and the first ten kept here. The `ORDER BY (SELECT NULL)` idiom would make it legal, and would also let the database choose the rows.
- **No session time zone.** `DATETIME2` is zone-less and `DATETIMEOFFSET` carries its own offset, and neither reads a session setting, so there is none to pin. An extraction is sent as `DATEPART(unit, SWITCHOFFSET(e, '+00:00'))`, which reads any of them at UTC as `relix` does. `DATE_TRUNC` is not sent: `DATETRUNC` arrived in SQL Server 2022, which a declared dialect cannot confirm.
- **No boolean value.** `TRUE` is written `1`, `IsNull` is a `CASE`, and `NULLs` are placed last with `CASE WHEN e IS NULL THEN 1 ELSE 0 END` rather than a boolean key.

An `ASOF` join folds as `OUTER APPLY (SELECT TOP 1 ...)`, or `CROSS APPLY` for the `INNER` form — the same correlated lookup `LATERAL` is elsewhere. A string literal is written `N'...'`, so a character outside the database's code page survives the trip. `Len`, `Left`, `Right` and `Mid` are computed

here: T-SQL's `LEN` counts UTF-16 units and ignores trailing spaces.

SQL Server's default collation, `SQL_Latin1_General_CP1_CI_AS`, is case-insensitive, so strings are compared under `Latin1_General_100_BIN2`, in the places MySQL's are converted. Two differences remain that no collation removes, and both are narrow:

- **Trailing spaces.** SQL Server compares strings padded, as the SQL standard's `PAD SPACE` rule says, so `'a' = 'a '` is true there under every collation it has; `relx` compares them as different values. A column whose values differ only in trailing spaces — a `CHAR(n)` column, say — compared or grouped on SQL Server can answer differently than in the engine.
- **Ordering beyond the basic multilingual plane.** A binary collation orders by UTF-16 code unit, as H2 does, and differs from `relx`'s code-point order only for a supplementary character compared with one in `U+E000..U+FFFF`.

Db2

Db2 for Linux, UNIX and Windows is recognised from a `jdbc:db2:` URL, or named with dialect: `db2`, and is supported from 11.5. Most of what `relx` renders it takes as written — `NULLS LAST`, typed temporal literals, `date_trunc`, window functions, and an ASOF join as `LEFT JOIN LATERAL (... LIMIT 1) ON TRUE`. A limit is the standard's `FETCH FIRST n ROWS ONLY`, which Db2 accepts with or without an `ORDER BY`.

Three things differ:

- **Names are left bare.** Db2 folds an unquoted name to upper case, so a table created as `orders` is stored as `ORDERS`. `relx` writes a plain identifier unquoted, as the generic dialect does, and delimits only a name like `order-lines` that could not be written bare. A table whose name was created quoted and in lower case must be declared exactly as it was created.
- **Characters are counted in CODEUNITS32.** Db2's `LENGTH`, `LEFT` and `RIGHT` count bytes by default, and its `LEFT` pads with spaces when asked for more than a string holds, so `Len`, `Left`, `Right` and `Mid` go through `CHARACTER_LENGTH` and `SUBSTRING` told to count in code points.
- **No zone to pin.** Db2 for LUW has no time-zone-bearing timestamp, so there is no session setting a temporal fold could be shifted by; a `TIMESTAMP` column holds the wall clock `relx` reads as UTC.

Strings are compared and ordered as written. A Db2 database created in UTF-8 — the default — uses the `IDENTITY` collation, which compares bytes and so orders by code point exactly as `relx` does. A database created with a locale collation does not, and should declare `collation: database`. Like SQL Server, Db2 compares strings padded, so `'a' = 'a '` is true there and false in `relx`.

Strings, and the collation they are compared under

`relx` compares strings exactly — character for character. A database compares them under the column's **collation**, and a collation may say that two values differing in case or in accent are the same value. MySQL's default (`utf8mb4_0900_ai_ci`) says exactly that, so `'Gold'`, `'GOLD'` and `'gold'` are one value there and three here.

That would make a query answer differently depending on whether the planner folded it, which is the one thing pushdown must never do. So on a dialect whose collation is not the

engine's comparison, a string is rendered with an explicit exact one:

```
WHERE CONVERT(`region` USING utf8mb4) COLLATE utf8mb4_0900_bin = 'west'
```

The `COLLATE` replaces the column's collation; the `CONVERT` is what makes that legal for a column of any character set, since a collation belongs to one. This applies wherever strings are compared for **equality** — `=`, `<`, `>`, `IN`, `LIKE`, a `GROUP BY`, a `SELECT DISTINCT`, a join condition — and it costs the column's index for that comparison, which is the right trade: the alternative is not an indexed lookup but no fold at all, so the choice is between a scan inside the database and every row of the table crossing the wire to be scanned here.

Ordering is collated the same way and pushed the same way. `relax` orders strings by **code point** — the order a binary collation gives, and the order UTF-8 bytes already sort in — so `ORDER BY` on a string key, a `</>` between strings, and `MIN/MAX` over one all reach the database.

Comparing and ordering are two questions, and PostgreSQL answers them differently. Its default collation is *deterministic*, which means it decides whether two strings are equal by comparing their bytes — so `=`, `IN`, `LIKE`, a `GROUP BY` and a `DISTINCT` mean there exactly what they mean here, and are pushed untouched, keeping the column's index. It decides their *order* by the locale, and a locale sorts the way a dictionary does: under `en_US.UTF-8`, `'Alan'` sorts before `'a😊b'` sorts before `'grace'`, where by code point the emoji comes after both. So only the ordering is wrapped, in the collation that orders by code point:

```
ORDER BY ("name") COLLATE "C" ASC NULLS LAST
```

That split is why a MySQL connection and a PostgreSQL one are collated in different places, and it is measured rather than reasoned: the agreement suite runs one corpus against both, and on PostgreSQL every equality case agreed and the three ordering ones did not.

One shape does not: ordering by a *grouping key* that had to be collated. MySQL matches an `ORDER BY` term against the `GROUP BY` expression as text, and placing `NULLS` wraps the key, so the term is no longer the expression MySQL is looking for and it rejects the query. That query is computed in the engine, and folds normally on a connection that needs no collating.

A connection whose string columns really are declared with a binary collation can say so, and gets all of it back — the comparison rendered as written, and string ordering pushed:

```
connection shop from database { url: "${SHOP_DB}", collation: exact };
```

The opposite, `collation: database`, asks for the conservative reading on a dialect whose default is `exact`. Neither changes the answer `relax` gives; they change which work the database does. See [connection](#).

Time zones, and the session relax opens

A `relax` `TIMESTAMP` is an instant, and the engine reads and computes it at UTC. A truncation or an extraction that folds is computed by the **server**, over whatever wall clock the session presents — so on a session an hour off UTC, `DATE_TRUNC('day', ts)` buckets by a different day than the same expression buckets by in the engine, and the query returns different rows depending on whether it folded.

So `relax` **pins the session to UTC** on every connection it opens to an identified backend — `SET TIME ZONE 'UTC'` on PostgreSQL, `SET time_zone = '+00:00'` on MySQL, `SET TimeZone = 'UTC'` on DuckDB (SQL Server has no session zone, and is handled by reading each value at

UTC instead) — and a temporal fold means the same thing wherever it runs. On PostgreSQL there is no other way to say it: the driver takes the session's zone from the client machine's default and ignores one named in the connection string, so without this a query's answers moved with the time zone of the host the engine ran on.

Two consequences worth knowing. A `TIMESTAMP WITH TIME ZONE` column is read as the instant it holds, which is what it always meant. And a connection whose backend relx cannot identify — the generic dialect — is left alone rather than sent a statement it might reject, so its session is whatever the driver made it.

MongoDB

A MongoDB connection folds a shorter list, into one aggregation pipeline:

| Operator | Pipeline stage |
|----------------------------|-------------------------------|
| relation over a collection | the collection |
| σ selection | <code>\$match</code> |
| μ unnest | <code>\$unwind</code> |
| π projection | <code>\$project</code> |
| λ limit | <code>\$skip / \$limit</code> |

Sorting, grouping and joins are computed by the engine, as is a negated predicate ($\neg$). The six EXTRACT-style date parts and `DATE_TRUNC` have pipeline spellings and fold inside a `$match` or `$project`.

Examples

A query that folds completely. The filter, the grouping and the aggregate all reach the database, so the rows that cross the network are the answer rather than the input:

```
connection warehouse from database { url: "${DB_URL}" };

source WarehouseOrders from warehouse {
  table: "orders",
  schema: { order_id: NUMBER, customer_id: NUMBER, amount: NUMBER, status: STRING }
};

query {  $\gamma$  customer_id, SUM(amount)  $\rightarrow$  total ( $\sigma$  status = 'OPEN' (WarehouseOrders)) };
```

The same query with one changed grouping key does not fold its γ , because a computed key is not a bare column. The σ still folds, so the database still filters — the grouping happens here, over the filtered rows:

```
connection warehouse from database { url: "${DB_URL}" };

source WarehouseOrders from warehouse {
  table: "orders",
  schema: { order_id: NUMBER, amount: NUMBER, status: STRING, placed_at: TIMESTAMP }
};

query {  $\gamma$  YEAR(placed_at)  $\rightarrow$  yr, SUM(amount)  $\rightarrow$  total ( $\sigma$  status = 'OPEN' (WarehouseOrders)) };
```

A join across two connections is always computed by the engine. Each side folds separately, which is the useful part: both databases filter before anything is sent:

```

connection warehouse from database { url: "${WAREHOUSE_URL}" };
connection archive from database { url: "${ARCHIVE_URL}" };

source LiveOrders from warehouse {
  table: "orders", schema: { order_id: NUMBER, status: STRING }
};
source ClosedOrders from archive {
  table: "closed", schema: { order_id: NUMBER, closed_on: DATE }
};

query {
  (σ status = 'OPEN' (LiveOrders)) ⋈ LiveOrders.order_id = ClosedOrders.order_id (ClosedOrders)
};

```

Reading a CSV file, nothing folds and the plan says so — there is no backend to fold into, and every operator is the engine's:

```

source LocalOrders from csv("./orders.csv") {
  header: true,
  schema: { order_id: NUMBER, amount: NUMBER, status: STRING }
};

query { γ status, SUM(amount) → total (σ amount > 100 (LocalOrders)) };

```

Limitations

Pushdown reaches one connection's tables at a time, and stops at the first operator or expression the backend cannot spell. An operator above that point is computed in the engine even where the backend could have expressed it in isolation.

The engine's own buffering is the practical consequence. A γ , τ , δ or set operation that does not fold holds its input in this process's memory while it runs, so the operators that stop folding are the ones worth checking on a large table. `--explain` is how you check.

Alternatives

When a query has to do work the fold cannot reach, define a **view in the database** and point a source at it with `table:`. A view is read exactly as a table is, so the backend computes whatever the view expresses and the engine folds its own filters and limits on top of the result — see [source](#).

See Also

[optimizer](#), [source](#), [connection](#), [group](#), [repl](#)

Notes

`--explain` is the only authority on what a particular query folded. The tables here describe the shapes the renderers accept, but whether *your* query matches one of them depends on the tree the optimizer produced, and reading the plan is faster than reasoning about it.