

Relix

Programming Guide

Using Relix from Java — building queries, inspecting what the engine makes of them, and running them.

Generated 1 October 2026 · revision e87dc7b0e95d

Contents

About this manual	4
1. Introduction	5
Getting the library	5
Getting started	7
The session	11
2. Building a query	16
Composing a query	16
Aggregation and analytics	24
Recursion and graphs	28
Time	33
Goal-seek and optimisation	40
Nested data	44
3. Getting data in	50
Bringing your own data and code	50
Generating data	54
Against a database	59
Secrets in declarations	65
Sandboxing a session	67
Working without a database	70
4. Getting answers out	74
Reading results	74
Looking at what the engine did	83
The engine describes itself	86
Provenance: where a row came from	91
5. Building tools	96

Building tools on Relix 96

About this manual

Using Relix from Java: building queries, inspecting what the engine makes of them, and running them.

Every Java example in this guide is compiled against the shipped API and run on each build, and the block underneath it holds what it actually printed. An example that stops compiling, or stops printing what its page claims, fails the build — so a page here cannot quietly fall behind the code it describes. Imports are elided where they are noise and shown where they are the point.

This guide is about the Java API. The language itself — every operator, predicate and function, with worked examples — is the subject of the other manual, the Relix language reference. The two share one vocabulary: each combinator on a relation is named exactly as the reference names the operator, so a question about what an operator *means* is answered there.

1. Introduction

Where to start: how to get the library onto a classpath, then one page that opens a session, gets data in, writes a query both ways, runs it, and looks at what the engine did with it.

Page	What it covers
Getting the library	The artifact and its coordinate; Java 21; adding a JDBC driver; checking what a session found
Getting started	The session and the relation; Java data in; the two authoring surfaces; the row terminals; the optimizer and the plan; errors
The session	Declaring, running a script, reading a session back, diagnostics, what a session can be given, closing

Getting the library

Relix is one artifact. Everything the rest of this guide uses — the engine, the default function library, the CSV/JSON/HTTP/JDBC connectors, the bundled solver — is inside it, so a program that queries a database needs this dependency and a driver, and nothing else.

The coordinate

The artifact is `com.darkcollective.relix:relix`, on Maven Central. That is one coordinate: the twenty-one modules the engine is assembled from are internal to the build and are not published separately — the module graph is a claim this build enforces about itself, not a set of names a program should depend on.

A consuming Gradle build names it:

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.darkcollective.relix:relix:1.0.0-rc2'
}
```

and a Maven build the same way:

```
<dependency>
  <groupId>com.darkcollective.relix</groupId>
  <artifactId>relix</artifactId>
  <version>1.0.0-rc2</version>
</dependency>
```

`ojalgo` is a runtime dependency of the published POM rather than a merged part of the jar. Shading it would hide it from your dependency report and from whatever scans that report for vulnerabilities, which is a worse trade than one visible transitive dependency.

To work on Relix itself, or to depend on a change that is not released, `./gradlew publishToMavenLocal` publishes the same coordinate to your local Maven repository under the version in the build; a consuming build reaches it by adding `mavenLocal()` to the repositories above.

Java 21

Relix targets **Java 21** and the jar is compiled for it. A build on an earlier JDK fails at class-load time, which is a confusing way to learn a version requirement, so it is worth stating in the consuming build:

```
java {  
    toolchain { languageVersion = JavaLanguageVersion.of(21) }  
}
```

A newer JDK is fine — 21 is the floor, not the ceiling.

The engine ships as a set of JPMS modules and works equally on the class path or the module path. On the module path, the module to require is `com.darkcollective.relix.embed`.

A driver is yours to add

The JDBC connector is in the jar; JDBC **drivers** are not. A session that reads PostgreSQL needs the PostgreSQL driver on your classpath, exactly as any other JDBC program does:

```
runtimeOnly 'org.postgresql:postgresql:42.7.4'
```

A session downloads nothing on its own — a library call that fetched a jar into your home directory would be a surprise you could not see coming. `Builder.provisioners` is how an application that means to offer that says so.

Checking what you have

`relix.version` is a relation listing one row per component actually present, which is the direct way to confirm the classpath is what you think it is:

```
import com.darkcollective.relix.embed.Relix;  
  
try (Relix relix = Relix.open()) {  
    relix.relation("π component (σ kind = 'facade' (relix.version))")  
        .toList()  
        .forEach(System.out::println);  
}  
  
(component=relix-embed)
```

`Relix.version()` is the engine's own version, the value the engine's row in that relation reports, for a program that wants it without running a query:

```
System.out.println(Relix.version());  
  
<version>
```

Provider discovery is a classpath scan, so a missing connector or function library fails nothing a compiler can see. Asking this relation is how the failure becomes visible before a query depends on it — filter it by `kind` to list the connectors, function libraries, solvers and drivers a session found.

Where to go next

[Getting started](#) opens a session and runs a first query.

Getting started

Relix is a relational algebra engine. This guide is about using it from Java: building queries, looking at what the engine makes of them, and running them.

Everything below is real code. Every example is compiled against the shipped API and run, and the block underneath it is what it actually printed.

A session and a relation

Two types carry the whole API.

A `Relix` **session** holds what a script would declare — sources, connections, views, functions — together with the bindings that connect them outward, such as a live `DataSource`. A `Relation` is a value: an expression together with the analysis it resolves against.

```
import com.darkcollective.relax.embed.Relix;
import com.darkcollective.relax.embed.Relation;

try (Relix relix = Relix.open()) {
    relix.define("""
        source Orders from csv("./orders.csv") {
            header: true,
            schema: { order_id: NUMBER, customer: STRING, status: STRING, amount: NUMBER }
        };
        """);

    Relation open = relix.relation("σ status = 'OPEN' (Orders)");
    System.out.println(open.render());
}

σ status = "OPEN" (Orders)
```

Nothing was read. `Relix.open()` reaches no database and that CSV file does not exist — the source *declares* its own schema, so the session has everything it needs to resolve the name, infer the heading, and hand back a relation.

That is the shape of the whole API: **a relation is a value, not a pending execution**. Composing one, rendering it, optimising it and asking for its schema are whole uses of the engine, and none of them runs anything.

Note also what `render()` gave back: `'OPEN'` came out as `"OPEN"`. Rendering is not an echo of what you typed — it prints the tree, so formatting and the spelling of a literal are normalised. What it guarantees is that the text parses back to the same expression.

A session is `AutoCloseable`, and closing it releases the database connections it opened. The `DataSource` you hand it stays yours and is left alone.

Data the program already holds

Reference data, fixtures, a result computed elsewhere — rows a Java program is holding go in directly, and are queryable beside everything else.

The rest of this page shares one session and one small table. A session is usually long-lived — a field closed at shutdown, rather than a `try` block per query — so here it is opened plainly:

```
Relix relix = Relix.open();
```

```
relix.table("Orders", List.of("order_id", "customer", "status", "amount"), List.of(
    Map.of("order_id", 1, "customer", "Ada", "status", "OPEN", "amount", 100),
    Map.of("order_id", 2, "customer", "Grace", "status", "SHIPPED", "amount", 250),
    Map.of("order_id", 3, "customer", "Ada", "status", "OPEN", "amount", 75)));
```

The heading is stated first, and each row is read by name — so the maps need no order of their own, and a row that omits a column has NULL for it. Types are **inferred** — a column of numbers is `NUMBER`, everything else is `STRING`. Where the types matter, declare a source with a schema instead ([Bringing your own data](#)).

Writing the query

There are two ways to say what you want, and they build the same tree.

The first is the language. `relation(String)` takes any relational expression, in either spelling Relix accepts — the operator glyph or its ASCII keyword:

```
System.out.println(relix.relation("SELECT status = 'OPEN' (Orders)").render());
```

```
σ status = "OPEN" (Orders)
```

The second is the combinators. Every operator that takes an input relation is a method on `Relation`, named as the reference manual names it:

```
Relation open = relix.relation("Orders")
    .select(eq(attr("status"), str("OPEN")))
    .project("customer", "amount");
```

```
System.out.println(open.render());
```

```
π customer, amount (σ status = "OPEN" (Orders))
```

`eq`, `attr` and `str` come from `Expr`, which is where an expression is built out of Java values rather than out of source text. **That is how a value is bound to a query**, and the reason no program here has to assemble one by concatenating strings:

```
long minimum = 90;
Relation large = relix.relation("Orders").select(gt(attr("amount"), num(minimum)));
```

```
System.out.println(large.render());
```

```
σ amount > 90 (Orders)
```

Getting the rows out

Three terminals, and the first is the one to reach for.

`toList()` drains the result and closes it, so the common case — a result that fits in memory — cannot leak the database connection an open result holds.

`stream()` is the opt-in for a caller who wants the engine's laziness and accepts what comes with it: **the stream is a resource, and you close it**. Closing early is also how a partial read is cancelled, because the engine is pull-based from end to end — once nobody pulls, nothing further is computed.

`count()` answers how many rows there are without materialising them. It is not a special counting mode: it is `γ COUNT(*)` over the relation, an operator that pushes down like any other, so against a database the backend does the counting and one row crosses the wire.

```
Relation orders = relix.relation("Orders");
```

```

System.out.println("count: " + orders.count());

for (Tuple row : relix.relation("σ status = 'OPEN' (Orders)").toList()) {
    System.out.println(row.string("customer") + " " + row.decimal("amount"));
}

try (Stream<Tuple> rows = orders.stream()) {
    System.out.println("streamed: " + rows.count());
}

```

```

count: 3
Ada 100
Ada 75
streamed: 3

```

Some relations never end — a generator over the natural numbers has no last row. `stream()` will happily read one, which is what generators are for; `toList()` refuses it rather than hanging.

Reading a row

A row is a `Tuple`: the engine's row, plus the conversions to Java types. `string`, `decimal`, `longValue`, `doubleValue`, `booleanValue`, `instant`, `date`, `time`, `duration`, `array` and `struct` each name a type and give you that type — **or null, if the value is NULL**, which is why none of them returns a primitive.

An accessor that meets a value of the wrong type raises rather than coercing. Coercion would make the accessor a second, weaker statement of what the column holds, and the failure would then arrive as a wrong answer instead of a wrong call:

```

Tuple first = relix.relation("Orders").toList().getFirst();

System.out.println(first.string("customer") + " / " + first.longValue("order_id"));

try {
    first.longValue("customer");
} catch (RelixException e) {
    System.out.println(e.getMessage());
}

```

Ada / 1
column 'customer' holds string, not number – read it with `get(...)` and match on the type, or render it with `get(...).asDisplayString()`

For a column whose type genuinely varies — an `ANY` column over schema-on-read data — the accessors are the wrong tool and `Value` is a sealed hierarchy, so a `switch` over it is exhaustive and the compiler names the case you forgot:

```

Value amount = relix.relation("Orders").toList().getFirst().get("amount");

System.out.println(switch (amount) {
    case com.darkcollective.relix.value.NumberValue n -> "number " + n.value();
    case com.darkcollective.relix.value.StringValue s -> "string " + s.value();
    default -> amount.asDisplayString();
});

```

```

number 100

```

That is the authoritative route, and the accessors are a convenience over the common case rather than a replacement for it.

Seeing what the engine did

The engine's phases are worth looking at, so they are terminals rather than internals.

`optimized()` hands back the rewritten relation, and `events()` says which rules produced it:

```
Relation written = relix.relation("σ status = 'OPEN' (σ amount > 50 (Orders))");
Relation rewritten = written.optimized();
```

```
System.out.println("as written: " + written.render());
System.out.println("rewritten:  " + rewritten.render());
rewritten.events().forEach(event -> System.out.println("  " + event.code()));
```

```
as written: σ status = "OPEN" (σ amount > 50 (Orders))
rewritten:  σ (status = "OPEN") ∧ (amount > 50) (Orders)
  SEL-002
```

`events()` is what the rewriter did. To see the whole run — the rewrite, the planner's choices and the execution — hand a listener to `stream()`, which reports as it goes rather than collecting the rows first:

```
import com.darkcollective.relix.events.QueryEvent;

List<QueryEvent> feed = new java.util.ArrayList<>();
try (Stream<Tuple> rows = relix.relation("σ amount > 50 (Orders)").stream(feed::add)) {
    rows.forEach(row -> { });
}

feed.forEach(event -> System.out.println(event.stage() + " " + event.code()
    + " " + event.description()));
```

```
EXECUTE SCAN scanned 3 rows
```

That is also how a query too large to hold is traced: `run()` gives you the rows and the feed together, and this gives you the feed alone.

There is one rule behind all of this: **inspection is staged, execution is not**. `render()` and `optimized().render()` differ, because the first shows what you wrote. `stream()` and `optimized().stream()` do not differ, because every execution terminal optimises first. `optimized()` is there to let you see the rewrite, not to switch it on.

`explain()` shows the physical plan — join algorithms, what was folded into a database query, and each step's estimated row count. Like `render()`, it is staged: this is the plan for the query as written.

```
System.out.println(relix.relation("σ status = 'OPEN' (Orders)").explain());
```

```
Select ~2 rows
└─ Scan Orders ~3 rows
```

Errors

Everything the facade refuses, it refuses by throwing `RelixException` — with one exception, and the exception is the interesting case.

```
try {
    relix.relation("σ status = 'OPEN' (NoSuchRelation)");
} catch (RelixException e) {
    System.out.println("refused");
}
```

```
refused
```

A typo in a query you wrote is a bug, and a bug should throw. A query assembled from someone else's input is different: there an error is an expected outcome to render, not a defect to raise. `validate` is for that, and returns the diagnostics as data:

```
for (Diagnostic problem : relix.validate("Bad := { σ nope = 1 (Orders) };")) {
    System.out.println(problem.message());
}
```

```
Selection σ: attribute 'nope' not found in input schema (available: order_id, customer, status, amount)
```

A failure that only appears once rows are moving is a third case: it arrives as the engine's own exception rather than as a `RelixException`, because there is nothing the facade could usefully add to it. The session has the detail.

Where to go next

What each operator *means*, and the full syntax of the language `relation(String)` accepts, are in the language reference. Every combinator on `Relation` carries the name its reference page gives the operator, so the two read together.

The session

A Relix session holds what a script would declare — sources, connections, views, functions — together with the bindings that connect them outward: a live `DataSource`, a catalog, a clock. It is the thing a program keeps.

A session is meant to be **long-lived**. The declarations it accumulates and the connections it pools are both worth keeping, so the usual shape is a field created at startup and closed at shutdown rather than a try block per query.

```
Relix relix = Relix.open();
```

`Relix.open()` is the everything-default session: no database bound, no catalog, the system clock, and the bundled functions and connectors installed. `Relix.builder()` is the same thing with the knobs exposed.

Declaring things

`define` takes declarations and returns the session, so it chains. It accumulates: two calls are one session, not two.

```
relix.define("""
    source Orders from csv("./orders.csv") { header: true, schema: {
        order_id: NUMBER, customer_id: NUMBER, status: STRING, amount: NUMBER } };
    """);

relix.define("OpenOrders := { σ status = 'OPEN' (Orders) };");

Relation openOrders = relix.relation("OpenOrders");
System.out.println(openOrders.render() + " — " + openOrders.schema().columns().size() + " columns");

OpenOrders — 4 columns
```

The view resolves against the source declared in the previous call, and its heading is the source's — one session, not two.

`define` refuses a query statement, and the refusal is the useful part: a query is not a declaration, it is a result, and `script` is what hands those back.

```
try {
    relix.define("query { σ amount > 10 (Orders) };");
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

define() takes declarations; this text has 1 query statement(s). Use script(...) to get a Relation for each.

Running a whole script

script is what a .relix file goes through. It installs the declarations and returns one Relation per query statement, in source order:

```
List<Relation> results = relix.script("""
    Shipped := { σ status = 'SHIPPED' (Orders) };
    query Shipped;
    query { π order_id (Orders) };
    """);

for (Relation result : results) {
    System.out.println(result.label().orElse("(unlabelled)") + " → " + result.render());
}
```

```
Shipped → Shipped
<expression 2> → π order_id (Orders)
```

label() is how a result names itself: the name for a query Shipped;, and a positional <expression N> for an unnamed one. It is the same rule the engine's own event feed uses, so a printed heading and a traced event agree about what to call a result.

Reading the session back

definitions() renders the session as .relix text, and statements() gives the same content as AST statements:

```
System.out.println(relix.definitions());
System.out.println(relix.statements().size() + " statements");

source Orders from csv("./orders.csv") { header: true, schema: { order_id: number, customer_id: number, status: string, amount: number }
↳ };
OpenOrders := { σ status = "OPEN" (Orders) };
Shipped := { σ status = "SHIPPED" (Orders) };

3 statements
```

A session that renders back as text is a session that can be saved, diffed, or handed to someone as a reproduction — which is why the printer exists at all.

Errors, and the one place they are not thrown

Everything the facade refuses, it refuses by throwing RelixException. A typo in a query your own program wrote is a bug, and a bug should stop the program:

```
try {
    relix.relation("σ status = 'OPEN' (NoSuchRelation)");
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

```
Undefined relation: 'NoSuchRelation'
```

A query assembled from someone else's input is a different situation. There an error is an expected outcome to render, not a defect to raise, and validate returns the diagnostics as data instead:

```
import com.darkcollective.relix.embed.Diagnostic;
```

```
for (Diagnostic problem : relix.validate("Bad := { σ nope = 1 (Orders) };")) {
    System.out.println(problem.severity() + ": " + problem.message());
}
```

```
ERROR: Selection σ: attribute 'nope' not found in input schema (available: order_id, customer_id, status, amount)
```

Check `severity()` rather than treating every diagnostic as a refusal — a warning is a diagnostic too, and a caller that stops on one stops on things that were fine.

```
System.out.println("errors only: "
    + relix.validate("Bad := { σ nope = 1 (Orders) };").stream()
        .filter(Diagnostic::isError)
        .count());
```

```
errors only: 1
```

`validate` also changes nothing: the text is analysed and discarded, so a session is never half-modified by a script that turned out not to analyse.

One thing does *not* come back as `RelixException`. A failure that only appears once rows are moving — a fixpoint that will not converge, a backend that rejects a query — arrives as the engine's own exception, because flattening every cause into one type would take the diagnosis away. Catch `RuntimeException` around a terminal if you mean to catch everything.

Configuring a session

`Relix.builder()` exposes what a session can be given. Each is a binding to the outside world rather than a tuning parameter:

Method	What it supplies
<code>jdbc(name, dataSource)</code>	a live database handle, under a connection name
<code>catalog(provider)</code>	where schemas and statistics come from — live, or a snapshot
<code>clock(clock)</code>	what <code>NOW()</code> and <code>CURRENT_DATE</code> read
<code>baseDirectory(path)</code>	what a relative file path in a source resolves against
<code>functions(library)</code>	functions of your own, merged with the bundled ones
<code>connector(connector)</code>	a backend of your own
<code>scriptLoader(loader)</code>	what an <code>import</code> statement resolves through
<code>maxFixpointRounds(n)</code>	the recursion safety valve
<code>provisioners(drivers, connectors)</code>	permission to download a missing driver or plugin
<code>relationships(graph)</code>	join edges the schema does not carry
<code>sessionEvents(events)</code>	the feed <code>relix.events</code> reports
<code>allowUnresolved()</code>	compose against names no catalog can resolve

The clock is the one worth showing, because it is what makes a query over "now" reproducible in a test:

```
import java.time.Clock;
import java.time.Instant;
import java.time.ZoneOffset;

try (Relix pinned = Relix.builder()
    .clock(Clock.fixed(Instant.parse("2026-03-01T12:00:00Z"), ZoneOffset.UTC))
    .build()) {

    System.out.println(pinned.relation("π NOW() → t, CURRENT_DATE() → d (UNIT)").toList());
}

[(t=2026-03-01T12:00:00Z, d=2026-03-01)]
```

UNIT there is the one-row, no-column relation — the algebra's way of saying "evaluate this expression once", which is what a scalar query is.

Two defaults are deliberate and worth knowing about, because both are cases where a library should not surprise its caller:

- **A session downloads nothing.** A missing JDBC driver is an error, not a fetch into the user's home directory. `provisioners` is how an application that means to offer that says so.
- **A session serves imports from nothing.** It is assembled in memory and has no file, so an `import` resolves against whatever `scriptLoader` supplies and against nothing otherwise.

Concurrency

A session is meant to be long-lived, and in a server that usually means one field shared by every request thread. So the division matters: **the session is not thread-safe, and the relations it hands out are.**

`define` mutates the session and the analysis is cached, so two threads declaring into one session — or one declaring while another mints a relation — race. Declare from one thread, at startup, before the session is shared.

A `Relation` is a different thing. It pins the model it was analysed against, which is what makes it a value rather than a view onto the session's mutable state, and that is exactly what makes it safe to hold, cache and hand to another thread. Draining two relations from two threads is in contract:

```
import java.util.List;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

// Declared once, before anything is shared – which is the rule, not the example's
// convenience.
relix.table("Payments", List.of(
    Map.of("status", "OPEN"), Map.of("status", "OPEN"), Map.of("status", "PAID")));

Relation openPayments = relix.relation("σ status = 'OPEN' (Payments)");
Relation paid = relix.relation("σ status = 'PAID' (Payments)");

try (ExecutorService threads = Executors.newFixedThreadPool(2)) {
    var counts = List.of(threads.submit(openPayments::count), threads.submit(paid::count));
    for (var pending : counts) {
        System.out.println(pending.get());
    }
}
```

```
}  
}  
  
2  
1
```

Both relations were minted before the threads started, and each carries the analysis it needs. Nothing in the two terminals coordinates, and nothing has to.

Two consequences are worth spelling out, because they are what the rule costs in practice:

- **A relation minted before a `define` does not see it.** That is the pinning rule, not a race, and it is the same behaviour on one thread. Mint the relations you intend to share after the declarations they depend on.
- **Terminals write to the session.** A drained read records what it measured, so the execution path touches session state without your code touching the session at all. That path is safe to run concurrently; it is `define` that is not.

If a request thread genuinely needs to declare something of its own, give it its own session. A session is cheap: what is expensive is the database connection pool, and a `DataSource` you bind is shared rather than copied, so several sessions over one handle pool independently against connections you still own.

Closing

Closing a session releases the database connections it opened. The `DataSource` you handed it is yours and is left alone, and everything else it holds is ordinary garbage:

```
relix.close();  
  
// Still a value: building one and rendering it never needed the session's bindings.  
System.out.println(relix.relation("OpenOrders").render());  
  
try {  
    relix.relation("Orders").toList();  
} catch (RelixException e) {  
    System.out.println(e.getMessage());  
}  
  
OpenOrders  
this session is closed
```

That is the division the whole API is built on, seen at the end of a session's life: a relation is a value, and only the execution terminals need anything from the outside world.

2. Building a query

One method per operator, named as the language reference names it — and the language itself wherever an expression reads better as text.

Page	What it covers
Composing a query	The two authoring surfaces; parameters; the unary operators; joins; set operations; universal quantification; building an AST directly
Aggregation and analytics	Grouping and reducing; NULL rules; ranking, running totals and offsets; top n per group; pivot
Recursion and graphs	Reachability, components, paths, cheapest routes, general recursion, and the safety valve
Time	Temporal types; binding a moment; the AS-OF join and its boundaries; interval joins; SESSIONIZE; DOWNSAMPLE
Goal-seek and optimisation	SOLVE; subset selection and allocation with OPTIMIZE; solved, infeasible and undetermined; checking for a solver
Nested data	Struct and array construction; COLLECT and UNNEST as inverses; TREE; declaring a nested schema and what it types

Composing a query

A relation is a value. Every operator that takes an input relation is a method on `Relation` that returns another one, so a query is built the way any other value is — by composition, in Java, with the compiler checking each step.

This page is about that surface: what the methods are called, how they chain, and when to write the expression as text instead. It uses one small session throughout.

```
import java.util.stream.Collectors;

Relix relix = Relix.open();

relix.define("""
    source Orders from csv("./orders.csv") { header: true, schema: {
        order_id: NUMBER, customer_id: NUMBER, status: STRING, amount: NUMBER,
        placed_at: TIMESTAMP } };
    source Customers from csv("./customers.csv") { header: true, schema: {
        customer_id: NUMBER, name: STRING, region: STRING } };
    """);

// A heading, printed as the reference prints one. Used throughout this page.
class Show {
    static String heading(Schema schema) {
        return schema.columns().stream()
            .map(column -> column.name() + ":" + column.type().display())
            .collect(Collectors.joining(", "));
    }
}
```

Nothing here reads a file. Both sources declare their schemas, so the session can resolve names, infer headings and build trees against a CSV that does not exist — which is the

whole of this page.

Two surfaces, one tree

`relation(String)` parses a relational expression. The combinators build the same nodes without parsing anything. They are not two dialects with two meanings — they are two ways of writing one tree:

```
Relation written = relix.relation("π customer_id, amount (σ status = 'OPEN' (Orders))");

Relation composed = relix.relation("Orders")
    .select(eq(attr("status"), str("OPEN")))
    .project("customer_id", "amount");

System.out.println(written.render());
System.out.println(composed.render());
System.out.println("same tree: " + written.render().equals(composed.render()));

π customer_id, amount (σ status = "OPEN" (Orders))
π customer_id, amount (σ status = "OPEN" (Orders))
same tree: true
```

Chaining reads inside-out relative to the algebra: `.select(...).project(...)` is $\pi (\sigma (...))$, because each call wraps what came before. That is the same order the rows travel in, and the opposite of the order the notation writes.

Which surface to use is a question about where the query comes from. Text is better when a human wrote the query, or when it is a constant. The combinators are better when the program is deciding — a filter that is only applied sometimes is an `if`, not string concatenation:

```
boolean onlyOpen = true;
long floor = 50;

Relation filtered = relix.relation("Orders");
if (onlyOpen) {
    filtered = filtered.select(eq(attr("status"), str("OPEN")));
}
filtered = filtered.select(gt(attr("amount"), num(floor)));

System.out.println(filtered.render());

σ amount > 50 (σ status = "OPEN" (Orders))
```

Parameters: putting a Java value into a query

`num(floor)` above is worth its own section, because it answers a question every embedder has on the first day: *I have a value in a variable — how does it get into the query?*

The answer is a literal factory. A Java value becomes a literal **node**, and the node goes into the tree beside the column it is compared against:

```
long orderId = 4711;

System.out.println(relix.relation("Orders")
    .select(eq(attr("order_id"), num(orderId)))
    .render());

σ order_id = 4711 (Orders)
```

There is one factory per literal kind, and the ones that matter take **Java values** rather than their source text:

Factory	Takes
<code>num(long) / num(double) / num(BigDecimal)</code>	a number
<code>str(String)</code>	a string
<code>bool(boolean)</code>	a boolean
<code>date(LocalDate) / time(LocalTime) / timestamp(Instant) / duration(Duration)</code>	a <code>java.time</code> value
<code>lit(Object)</code>	whichever of the above fits, when you would rather not name the type

`timestamp(Instant)` is the one to reach for when a query is scoped to a moment the program computed — the end of the last run, a request's cut-off, a clock reading — because the alternative is formatting an instant into text and hoping the parser reads it back the same way:

```
import java.time.Instant;
import java.time.temporal.ChronoUnit;

Instant runStartedAt = Instant.parse("2026-03-01T09:00:00Z"); // a clock reading
Instant since = runStartedAt.minus(7, ChronoUnit.DAYS);

System.out.println(relix.relation("Orders")
    .select(allOf(ge(attr("placed_at"), timestamp(since)),
                  gt(attr("amount"), num(new java.math.BigDecimal("99.95")))))
    .render());

σ (placed_at ≥ TIMESTAMP '2026-02-22T09:00:00Z') ∧ (amount > 99.95) (Orders)
```

A list of values is `in`, which takes the same literal nodes:

```
import com.darkcollective.relix.ast.AstBuilders;
import com.darkcollective.relix.ast.Operand;

List<String> wanted = List.of("OPEN", "PENDING");

System.out.println(relix.relation("Orders")
    .select(in(attr("status"), wanted.stream().map(AstBuilders::str).toArray(Operand[]::new)))
    .render());

σ status ∈ {"OPEN", "PENDING"} (Orders)
```

Why this is the safety story as well as the ergonomic one

Nothing above is formatted into text and parsed back, so **there is no spelling for a value to escape through**. A value that arrives from outside stays a value however it is written:

```
String fromTheOutside = "OPEN\" v 1 = 1 --";

Relation guarded = relix.relation("Orders").select(eq(attr("status"), str(fromTheOutside)));

System.out.println(guarded.render());
System.out.println("the text parses back to the same tree: "
    + relix.relation(guarded.render().render().equals(guarded.render())));
```

```
σ status = "OPEN\" v 1 = 1 --" (Orders)
the text parses back to the same tree: true
```

The whole hostile string sits inside one string literal, escaped on the way out — and reading that text back gives the identical tree, so the escaping is exact rather than approximate. Rendering is a view of the tree, not the route the value took into it: the tree was built from a `String` and a column reference, and no arrangement of characters in that string can turn it into two comparisons or a comment.

That is the reason a program building queries from user input should never reach for `σ status = ' ' + input + ' '`. It is not that string concatenation is inelegant; it is that the combinators make the safe thing the shorter thing.

The same holds where the query is not evaluated here. A selection over a database table is folded into a `WHERE` clause, so the value is written out a second time — as a SQL string literal, by the rules of the backend it is going to. Those rules differ: doubling an embedded quote is standard SQL, and MySQL and MariaDB additionally read a backslash as an escape, so a value ending in one would otherwise close the literal that was meant to contain it. Each backend is asked for its own spelling, and the pushed and in-engine paths are held to the same answer over a corpus of values chosen for exactly these characters.

The unary operators

Seven operators take one relation and give back one relation, and they carry the names the language reference gives them.

Method	Operator	What it does
<code>select(predicate)</code>	σ	Keeps the rows that satisfy the predicate
<code>project(columns...)</code>	π	Keeps, computes and renames columns
<code>rename(name) / rename(name, columns)</code>	ρ	Renames the relation, or its columns positionally
<code>distinct()</code>	δ	Removes duplicate rows
<code>sort(keys...)</code>	τ	Orders the rows
<code>limit(count) / limit(offset, count)</code>	λ	Takes a window of rows
<code>aggregate(keys, aggregates)</code>	γ	Groups and reduces

They chain in any order the algebra allows:

```
Relation top = relix.relation("Orders")
    .select(gt(attr("amount"), num(20)))
    .project("customer_id", "amount")
    .distinct()
    .sort(desc("amount"))
    .limit(3);

System.out.println(top.render());

λ 3 (τ amount DESC (δ (π customer_id, amount (σ amount > 20 (Orders)))))
```

`project` has two forms. The varargs one takes column names, which is the common case. The other takes `ProjectedAttributes`, which is how a projection computes something or names its output:

```
Relation withVat = relix.relation("Orders").project(List.of(
    projected(attr("order_id")),
    projected(times(attr("amount"), num(1.2)), "gross")));
```

```
System.out.println(withVat.render());
System.out.println(Show.heading(withVat.schema()));
```

```
π order_id, amount * 1.2 → gross (Orders)
order_id:number, gross:number
```

`schema()` is worth noticing there: the heading of a computed column is known before anything runs, because composing a relation re-runs inference over the node just built.

`limit` **wraps** rather than replacing. Two limits are two windows, one applied to the result of the other, which is what the algebra says and not what a builder that sets a field would do:

```
System.out.println(relix.relation("Orders").limit(10).limit(2).render());
```

```
λ 2 (λ 10 (Orders))
```

Joins

Every join is a method on the left relation, taking the right one as an argument.

```
Relation orders = relix.relation("Orders");
Relation customers = relix.relation("Customers");
```

```
System.out.println(orders.join(customers).render());
System.out.println(orders.join(customers, eq(attr("Orders.customer_id"),
    attr("Customers.customer_id"))).render());
System.out.println(orders.leftJoin(customers, eq(attr("Orders.customer_id"),
    attr("Customers.customer_id"))).render());
```

```
(Orders) ⋈ (Customers)
(Orders) ⋈ Orders.customer_id = Customers.customer_id (Customers)
(Orders) ⋈ Orders.customer_id = Customers.customer_id (Customers)
```

`join(right)` with no condition is the **natural** join — it matches on the columns the two headings share and keeps one copy of each. `join(right, condition)` is the theta join, where the condition is yours. The rest follow the same pattern: `leftJoin`, `rightJoin`, `fullJoin`, `semiJoin`, `antiJoin`.

The two that are not about combining rows are worth separating. A **semi-join** filters the left relation by whether a match exists, keeping only the left's columns; an **anti-join** keeps the left rows that have no match:

```
Relation recognised = orders.semiJoin(customers,
    eq(attr("Orders.customer_id"), attr("Customers.customer_id")));
```

```
System.out.println(recognised.render());
System.out.println(Show.heading(recognised.schema()));
```

```
(Orders) ⋈ Orders.customer_id = Customers.customer_id (Customers)
order_id:number, customer_id:number, status:string, amount:number, placed_at:timestamp
```

Set operations

The set operators take two relations with compatible headings.

```
Relation open = relix.relation("σ status = 'OPEN' (Orders)");
Relation large = relix.relation("σ amount > 100 (Orders)");
```

```
System.out.println(open.union(large).render());
```

```
System.out.println(open.unionAll(large).render());
System.out.println(open.difference(large).render());
System.out.println(open.intersect(large).render());
```

```
(σ status = "OPEN" (Orders)) ∪ (σ amount > 100 (Orders))
(σ status = "OPEN" (Orders)) ∖ (σ amount > 100 (Orders))
(σ status = "OPEN" (Orders)) − (σ amount > 100 (Orders))
(σ status = "OPEN" (Orders)) ∩ (σ amount > 100 (Orders))
```

`union` deduplicates and `unionAll` does not — the same distinction SQL draws, spelled `u` and `∪`. `outerUnion`, `symmetricDifference`, `divide`, `compose` and `cross` complete the set, and each is named as its reference page names the operator.

"Every": universal quantification

Customers who ordered every product is the question SQL makes hardest — the double-negated `NOT EXISTS (... WHERE NOT ...)` everyone has written once and nobody enjoys reading. Relix has operators for it, and which one you want depends on what "every" ranges over.

The examples below run against real rows, so the answers can be checked by eye:

```
relix.table("Purchases", List.of("customer", "product", "returned"), List.of(
    Map.of("customer", "Ada", "product", "Widget", "returned", "no"),
    Map.of("customer", "Ada", "product", "Gadget", "returned", "no"),
    Map.of("customer", "Ada", "product", "Sprocket", "returned", "no"),
    Map.of("customer", "Grace", "product", "Widget", "returned", "no"),
    Map.of("customer", "Grace", "product", "Sprocket", "returned", "yes"),
    Map.of("customer", "Lin", "product", "Widget", "returned", "no"),
    Map.of("customer", "Lin", "product", "Gadget", "returned", "yes"),
    Map.of("customer", "Lin", "product", "Sprocket", "returned", "no")));

relix.table("Catalogue", List.of("product", "price"), List.of(
    Map.of("product", "Widget", "price", 10),
    Map.of("product", "Gadget", "price", 40),
    Map.of("product", "Sprocket", "price", 25)));
```

When "every" ranges over the values in **another relation**, it is division. Divide the pairs by the set they must cover, and what comes back is the side that covers it:

```
Relation completists = relix.relation("π customer, product (Purchases)")
    .divide(relix.relation("π product (Catalogue)"));

System.out.println(completists.render());
completists.toList().forEach(System.out::println);

(π customer, product (Purchases)) ÷ (π product (Catalogue))
(customer=Ada)
(customer=Lin)
```

Ada and Lin have bought all three products; Grace has never bought a Gadget. Read the rows and you can check it, which is the whole reason this example is eight rows long.

When "every" ranges over the rows of **one group**, it is `forall`. The keys are the grouping columns and the predicate is what every row in the group has to satisfy:

```
Relation neverReturned = relix.relation("Purchases")
    .forall(List.of("customer"), eq(attr("returned"), str("no")));

System.out.println(neverReturned.render());
neverReturned.toList().forEach(System.out::println);
```

```
∀ customer : returned = "no" (Purchases)
(customer=Ada)
```

Only Ada. Note how differently the two questions cut: Lin bought everything *and* returned something, so she is in one answer and not the other. Note also that the output is the grouping keys alone — a group either qualifies or it does not, and there is nothing else to report about it.

The trap this operator exists to avoid is worth seeing once. Filtering and then grouping answers a *different* question — "who has at least one unreturned purchase?" — and it quietly includes Grace and Lin:

```
relix.relation("δ (π customer (σ returned = 'no' (Purchases)))")
    .sort(asc("customer")).toList().forEach(System.out::println);

(customer=Ada)
(customer=Grace)
(customer=Lin)
```

The other two forms

With no grouping keys at all, `forall` asks a yes/no question about the whole relation and answers with a **truth relation** — one empty tuple for yes, no rows for no:

```
Relation allClean = relix.relation("Purchases").forall(List.of(), eq(attr("returned"), str("no")));
Relation adaClean = relix.relation("σ customer = 'Ada' (Purchases)")
    .forall(List.of(), eq(attr("returned"), str("no")));

System.out.println("every purchase kept:      " + (allClean.count() == 1));
System.out.println("every Ada purchase kept:  " + (adaClean.count() == 1));

every purchase kept:      false
every Ada purchase kept:  true
```

A relation with no columns has exactly two possible values, and they are the two truth values — which is why "did every row pass?" needs no special result type.

`forall(Relation, Predicate)` is a third operator behind the same method name, and it is easy to reach for by accident. It is the $\forall$ dual of the semi-join: it keeps a **left** row when every row of the **right** relation satisfies the condition paired with it. "Products every department can afford" is that shape — the left row is the product, and the right relation is the set of budgets it has to clear:

```
relix.table("Limits", List.of("department", "cap"), List.of(
    Map.of("department", "ops", "cap", 30),
    Map.of("department", "lab", "cap", 50)));

Relation affordable = relix.relation("Catalogue")
    .forall(relix.relation("Limits"), le(attr("price"), attr("cap")));

System.out.println(affordable.render());
affordable.toList().forEach(System.out::println);

(Catalogue) USEMI price ≤ cap (Limits)
(product=Widget, price=10)
(product=Sprocket, price=25)
```

The Gadget clears the lab's budget but not the ops one, so it fails "every". As with a semi-join, only the left relation's columns come out — the right side is a test, not a source of data.

When to write the expression instead

The combinators cover every operator that takes an input relation. For an operator with several parameters, that means a call with several arguments, and the text is usually easier to read:

```
Relation ranked = relix.relation("""
    WINDOW ROW_NUMBER() SORT amount DESC PER customer_id AS rank (Orders)
    """);
```

```
System.out.println(ranked.render());
```

```
WINDOW ROW_NUMBER() SORT amount DESC PER customer_id AS rank (Orders)
```

The combinator builds the same node, and shows why the text won here — a window carries a function, partition keys, an ordering, a frame and an output column, so the call is five arguments long:

```
import com.darkcollective.relix.ast.RankingFunction;
import com.darkcollective.relix.ast.WindowFrame;
import com.darkcollective.relix.ast.WindowFunction;
import java.util.Optional;
```

```
Relation rankedByHand = relix.relation("Orders").window(
    new WindowFunction.RankingWindow(RankingFunction.ROW_NUMBER, Optional.empty()),
    List.of("customer_id"),
    List.of(desc("amount")),
    new WindowFrame.PartitionFrame(),
    "rank");
```

```
System.out.println(rankedByHand.render());
```

```
WINDOW ROW_NUMBER() SORT amount DESC PER customer_id AS rank (Orders)
```

Mixing the two is the normal thing to do, because both produce a `Relation` and a `Relation` is a value: parse the awkward part, compose around it.

```
System.out.println(ranked.select(le(attr("rank"), num(1))).project("customer_id", "amount").render());
```

```
π customer_id, amount (σ rank ≤ 1 (WINDOW ROW_NUMBER() SORT amount DESC PER customer_id AS rank (Orders)))
```

Building the tree directly

There is a third surface, one step lower: build the `RelNode` with `AstBuilders` and hand it to `relation(RelNode)`. This is for a program that is *generating* queries — a query builder of your own, a translation from some other language — where composing from a named starting relation is the wrong shape because there is no starting relation.

```
import com.darkcollective.relix.ast.RelNode;

RelNode tree = project(attrs("customer_id", "amount"),
    select(gt(attr("amount"), num(100)), rel("Orders")));
```

```
Relation generated = relix.relation(tree);
System.out.println(generated.render());
```

```
π customer_id, amount (σ amount > 100 (Orders))
```

`rel`, `select`, `project` and `attrs` are the same factories the combinators call, so this is the identical tree by a different route. What the facade adds on top is the analysis: `relation(RelNode)` resolves the names, infers the heading, and refuses a tree that does not

analyse.

A relation keeps the session it was built in

A relation pins the analysis it was created against. Redefining a view afterwards does not reach back into a relation already built over it:

```
relix.define("Recent := { σ order_id > 1 (Orders) };");
Relation before = relix.relation("Recent");

relix.define("Recent := { σ order_id > 99 (Orders) };");
Relation after = relix.relation("Recent");

System.out.println(before.optimized().render());
System.out.println(after.optimized().render());

σ order_id > 1 (Orders)
σ order_id > 99 (Orders)
```

Both relations are named `Recent` and they are different values, which is what makes "a relation is a value" true rather than a figure of speech. Optimising is what makes the difference visible here: the view is inlined, and each relation inlines the definition that was in force when it was built.

Aggregation and analytics

Grouping, ranking, running totals, top-n-per-group. This page runs every example against real rows, so each one shows both the expression and the result it produces.

```
import com.darkcollective.relix.ast.AggregateOperator;

Relix relix = Relix.open();

relix.table("Sales", List.of("day", "region", "rep", "amount"), List.of(
    Map.of("day", 1, "region", "North", "rep", "Ada", "amount", 100),
    Map.of("day", 1, "region", "North", "rep", "Grace", "amount", 250),
    Map.of("day", 2, "region", "North", "rep", "Ada", "amount", 75),
    Map.of("day", 2, "region", "South", "rep", "Lin", "amount", 300),
    Map.of("day", 3, "region", "South", "rep", "Lin", "amount", 120),
    Map.of("day", 3, "region", "South", "rep", "Omar", "amount", 40)));

relix.relation("Sales").toList().forEach(System.out::println);

(day=1, region=North, rep=Ada, amount=100)
(day=1, region=North, rep=Grace, amount=250)
(day=2, region=North, rep=Ada, amount=75)
(day=2, region=South, rep=Lin, amount=300)
(day=3, region=South, rep=Lin, amount=120)
(day=3, region=South, rep=Omar, amount=40)
```

Tuple's own `toString` is what prints those rows, which keeps the examples below about the query rather than about formatting.

Grouping

`aggregate(keys, aggregates)` is γ . It takes the grouping columns and the reductions in **one call**, because γ is one operator — there is no separate "group by" step to attach an aggregate to later.

```
Relation byRegion = relix.relation("Sales").aggregate(
    List.of("region"),
    List.of(agg(AggregateOperator.SUM, "amount", "total"),
            agg(AggregateOperator.COUNT, "amount", "orders")));
```

```
System.out.println(byRegion.render());
byRegion.toList().forEach(System.out::println);
```

```
γ region, SUM(amount) → total, COUNT(amount) → orders (Sales)
(region=North, total=425, orders=3)
(region=South, total=460, orders=3)
```

The alias is the third argument to `agg`. Leave it off and the output column is named after the operator and its argument:

```
Relation unaliased = relix.relation("Sales")
    .aggregate(List.of("region"), List.of(agg(AggregateOperator.AVG, "amount")));
```

```
unaliased.toList().forEach(System.out::println);
```

```
(region=North, avg_amount=141.6666666667)
(region=South, avg_amount=153.3333333333)
```

Grouping by nothing reduces the whole relation to one row. That is a different thing from an empty result, and the distinction matters: a count of nothing is a row saying zero.

```
relix.relation("σ amount > 1000 (Sales)")
    .aggregate(List.of(), List.of(agg(AggregateOperator.COUNT, "amount", "n")))
    .toList()
    .forEach(System.out::println);
```

```
(n=0)
```

Grouping and aggregating by an expression

A grouping key is an expression, not only a column name, and so is an aggregate's argument. `aggregateBy` takes `GroupingKeys` for the first and `aggOf` builds the second:

```
Relation banded = relix.relation("Sales").aggregateBy(
    List.of(key(func("IIf", condition(gt(attr("amount"), num(100))), str("large"), str("small")), "band")),
    List.of(aggOf(AggregateOperator.SUM, times(attr("amount"), num(2)))));
```

```
System.out.println(banded.render());
banded.toList().forEach(System.out::println);
```

```
γ IIf((amount > 100), "large", "small") → band, SUM(amount * 2) (Sales)
(band=small, sum_expr=430)
(band=large, sum_expr=1340)
```

Three things are worth noting there. The key is a function call over the row, so the grouping is computed rather than looked up; `SUM(amount * 2)` reduces an expression, which is why an aggregate takes an operand rather than a column name; and `condition(...)` is what puts a predicate where an operand is expected — `IIf` takes a boolean argument, and a comparison is a `Predicate` until it is wrapped.

The unaliased output column is `sum_expr` rather than `sum_amount`, because the argument is an expression and there is no column name to derive one from. Alias anything you intend to read back by name.

NULLs

Aggregates follow the SQL rule: a row whose argument is NULL is skipped, so `COUNT(expr)` counts non-NULL values and `SUM` of nothing at all is NULL rather than zero. The two rows below that omit value are how a NULL is written when the heading is stated.

```
relix.table("Readings", List.of("sensor", "value"), List.of(
    Map.of("sensor", "a", "value", 10),
    Map.of("sensor", "a"),
    Map.of("sensor", "b")));

relix.relation("Readings").aggregate(
    List.of("sensor"),
    List.of(agg(AggregateOperator.COUNT, "value", "n"),
            agg(AggregateOperator.SUM, "value", "total")))
    .toList()
    .forEach(System.out::println);

(sensor=a, n=1, total=10)
(sensor=b, n=0, total=NULL)
```

`COLLECT` is the deliberate exception — it keeps NULLs, because gathering a group's values into an array is a different question from reducing them.

Ranking and running totals

A window function adds a column to every row rather than reducing the rows away. All three families are the `window` combinator, differing in the function they carry.

Ranking — a position within each partition:

```
import com.darkcollective.relix.ast.RankingFunction;
import com.darkcollective.relix.ast.WindowFrame;
import com.darkcollective.relix.ast.WindowFunction;
import java.util.Optional;

Relation ranked = relix.relation("Sales").window(
    new WindowFunction.RankingWindow(RankingFunction.ROW_NUMBER, Optional.empty()),
    List.of("region"),
    List.of(desc("amount")),
    new WindowFrame.PartitionFrame(),
    "rank");

System.out.println(ranked.render());
ranked.toList().forEach(System.out::println);

WINDOW ROW_NUMBER() SORT amount DESC PER region AS rank (Sales)
(day=1, region=North, rep=Grace, amount=250, rank=1)
(day=1, region=North, rep=Ada, amount=100, rank=2)
(day=2, region=North, rep=Ada, amount=75, rank=3)
(day=2, region=South, rep=Lin, amount=300, rank=1)
(day=3, region=South, rep=Lin, amount=120, rank=2)
(day=3, region=South, rep=Omar, amount=40, rank=3)
```

A running total — the same operator with an aggregate function and a cumulative frame:

```
Relation running = relix.relation("Sales").window(
    new WindowFunction.AggregateWindow(AggregateOperator.SUM, attr("amount")),
    List.of("region"),
    List.of(asc("day")),
    new WindowFrame.CumulativeFrame(),
    "running_total");
```

```
running.toList().foreach(System.out::println);

(day=1, region=North, rep=Ada, amount=100, running_total=100)
(day=1, region=North, rep=Grace, amount=250, running_total=350)
(day=2, region=North, rep=Ada, amount=75, running_total=425)
(day=2, region=South, rep=Lin, amount=300, running_total=300)
(day=3, region=South, rep=Lin, amount=120, running_total=420)
(day=3, region=South, rep=Omar, amount=40, running_total=460)
```

The frame is what separates a running total from a moving one. `CumulativeFrame` is everything up to this row; `BoundedFrame(n)` is the last n rows, which is a moving average; `PartitionFrame` is the whole partition, which is what a ranking function needs.

An offset function reads another row of the same partition:

```
import com.darkcollective.relix.ast.OffsetFunction;

Relation withPrevious = relix.relation("Sales").window(
    new WindowFunction.OffsetWindow(OffsetFunction.LAG, attr("amount"),
        Optional.of(num(1)), Optional.empty()),
    List.of("region"),
    List.of(asc("day")),
    new WindowFrame.PartitionFrame(),
    "previous");

withPrevious.toList().foreach(System.out::println);

(day=1, region=North, rep=Ada, amount=100, previous=NULL)
(day=1, region=North, rep=Grace, amount=250, previous=100)
(day=2, region=North, rep=Ada, amount=75, previous=250)
(day=2, region=South, rep=Lin, amount=300, previous=NULL)
(day=3, region=South, rep=Lin, amount=120, previous=300)
(day=3, region=South, rep=Omar, amount=40, previous=120)
```

Where a window carries this many parameters, the language spelling is often the clearer one, and both produce the same tree:

```
System.out.println(relix.relation(
    "WINDOW LAG(amount, 1) SORT day PER region AS previous (Sales)").render());

WINDOW LAG(amount, 1) SORT day PER region AS previous (Sales)
```

Top n per group

Ranking and then filtering on the rank is the usual way to express "the best three per region", and it works. `top` says it directly, in one operator the planner can act on:

```
Relation best = relix.relation("Sales").top(List.of("region"), List.of(desc("amount")), 1);

System.out.println(best.render());
best.toList().foreach(System.out::println);

TOP 1 amount DESC PER region (Sales)
(day=1, region=North, rep=Grace, amount=250)
(day=2, region=South, rep=Lin, amount=300)
```

The grouping list is what makes it per-group. Pass an empty list and it is the top rows of the whole relation, which is `sort(...).limit(n)` said as one operator.

Reshaping

`pivot` spreads one column's values across the heading, and `unpivot` folds a run of columns back into name/value pairs.

```
Relation pivoted = relix.relation("Sales")
    .aggregate(List.of("day", "region"), List.of(agg(AggregateOperator.SUM, "amount", "total")))
    .pivot("total", "region", List.of("day"));

System.out.println(pivoted.render());
pivoted.toList().forEach(System.out::println);

PIVOT total BY region PER day (γ day, region, SUM(amount) → total (Sales))
(day=1, North=350, South=NULL)
(day=2, North=75, South=300)
(day=3, North=NULL, South=160)
```

A pivot's heading depends on the *data* — one column per distinct value of the key column — so it is one of the few operators whose schema cannot be known before it runs. The analyser types it as an open heading and the executor fills it in.

What each of these means

The semantics — which frames are legal, how ties rank, what a pivot does with a value that appears twice — belong to the language reference, and each combinator here carries the name of the page that answers for it.

Recursion and graphs

An edge list is a relation, so reachability, components, paths and shortest routes are operators over it rather than a separate graph library. This page runs each one against a small network.

```
Relix relix = Relix.open();

// A → B → C → D, with a direct A → C shortcut, and an isolated pair E → F.
relix.table("Edges", List.of("src", "dst", "cost"), List.of(
    Map.of("src", "A", "dst", "B", "cost", 4),
    Map.of("src", "B", "dst", "C", "cost", 3),
    Map.of("src", "C", "dst", "D", "cost", 2),
    Map.of("src", "A", "dst", "C", "cost", 9),
    Map.of("src", "E", "dst", "F", "cost", 1)));

relix.relation("Edges").toList().forEach(System.out::println);

(src=A, dst=B, cost=4)
(src=B, dst=C, cost=3)
(src=C, dst=D, cost=2)
(src=A, dst=C, cost=9)
(src=E, dst=F, cost=1)
```

Reachability

`closure(from, to)` is the transitive closure: every pair connected by one or more edges.

```
Relation reachable = relix.relation("π src, dst (Edges)").closure("src", "dst");

System.out.println(reachable.render());
reachable.toList().forEach(System.out::println);
```

```
CLOSURE src, dst (π src, dst (Edges))
(src=A, dst=B)
(src=B, dst=C)
(src=C, dst=D)
(src=A, dst=C)
(src=E, dst=F)
(src=B, dst=D)
(src=A, dst=D)
```

The projection in front of it is doing real work: the closure is over `(src, dst)` pairs, and carrying `cost` along would ask what the cost of a two-hop pair is — a question the operator does not answer and `trace` does.

The reflexive form adds every node's pair with itself, which is what you want when "can A reach A?" should be true:

```
System.out.println(relix.relation("π src, dst (Edges)")
    .closure("src", "dst", true)
    .count() + " reflexive pairs");
```

```
13 reflexive pairs
```

Components

`cluster` labels each node with the connected component it belongs to, which is how a graph is partitioned without knowing in advance how many parts it has.

```
Relation components = relix.relation("π src, dst (Edges)")
    .cluster("src", "dst", "component");
```

```
System.out.println(components.render());
components.toList().forEach(System.out::println);
```

```
CLUSTER src, dst AS component (π src, dst (Edges))
(src=A, component=1)
(src=B, component=1)
(src=C, component=1)
(src=D, component=1)
(src=E, component=2)
(src=F, component=2)
```

Paths, with their length

`closure` says *whether* two nodes are connected. `path` says how far apart they are, and takes a hop range so a query can ask only for what it needs:

```
Relation twoHops = relix.relation("π src, dst (Edges)")
    .path("src", "dst", 2, 3, "hops");
```

```
System.out.println(twoHops.render());
twoHops.toList().forEach(System.out::println);
```

```
PATH src, dst HOPS 2 TO 3 AS hops (π src, dst (Edges))
(src=A, dst=D, hops=2)
(src=B, dst=D, hops=2)
```

Bounding the hops is not only a filter on the answer — it bounds the work, so a range is worth stating whenever the question has one.

The cheapest route

`trace` walks a weighted edge relation and gives back, for each reachable pair, the best route and the sequence of nodes it goes through:

```
import com.darkcollective.relix.ast.ObjectiveSense;

Relation cheapest = relix.relation("Edges")
    .trace("src", "dst", "cost", ObjectiveSense.MINIMIZE, "route");

System.out.println(cheapest.render());
cheapest.toList().forEach(System.out::println);
```

```
TRACE src, dst VIA cost MINIMIZE AS route (Edges)
(src=A, dst=B, cost=4, route=[A, B])
(src=B, dst=C, cost=3, route=[B, C])
(src=C, dst=D, cost=2, route=[C, D])
(src=E, dst=F, cost=1, route=[E, F])
(src=A, dst=C, cost=7, route=[A, B, C])
(src=B, dst=D, cost=5, route=[B, C, D])
(src=A, dst=D, cost=9, route=[A, B, C, D])
```

A → C is where that pays: there is a direct edge costing 9 and a two-hop route costing 7, and the cheaper one wins. The route column is an array, read from a row with `Tuple.array`:

```
Tuple aToD = cheapest.toList().stream()
    .filter(row -> row.string("src").equals("A") && row.string("dst").equals("D"))
    .findFirst()
    .orElseThrow();

System.out.println(aToD.array("route").stream().map(Value::asDisplayString).toList());

[A, B, C, D]
```

Reading the edges both ways

Every operator above follows the edge from `src` to `dst` and not back. That is the right reading for a dependency graph or a chain of command, and the wrong one for a friendship, a shared border or a cable. Passing `undirected` makes the same edge list readable from either end, without building a second relation holding the reverse of every row:

```
Relation either = relix.relation("π src, dst (Edges)").closure("src", "dst", true, false);

System.out.println(either.render());
System.out.println("pairs: " + either.count());

CLOSURE src ↔ dst (π src, dst (Edges))
pairs: 20
```

The result is symmetric — D now reaches A, which it cannot do along the arrows — and the reference page spells the same thing `CLOSURE src ↔ dst (Edges)`. `path` and `trace` take the flag in the same position, so a shortest route may run against the direction a row was written in:

```
Relation routes = relix.relation("Edges")
    .trace("src", "dst", true, "cost", ObjectiveSense.MINIMIZE, "route");

routes.toList().stream()
    .filter(t -> t.string("src").equals("D") && t.string("dst").equals("A"))
    .forEach(System.out::println);
```

```
(src=D, dst=A, cost=9, route=[D, C, B, A])
```

`cluster` takes no such flag: connected components are an undirected question already, so it has always ignored edge direction.

General recursion

`CLOSURE`, `CLUSTER` and `PATH` are the specialised, fast forms. `FIX` is the general one: a base relation and a step that derives more rows from what has been found so far, repeated until nothing new appears.

The step refers to the accumulated result by the name `FIX` binds, and that name means nothing outside the operator — so in a script a step is written inside its binder:

```
Relation explosion = relix.relation("""
    FIX Reach (
        π src, dst (Edges),
        π src, dst2 → dst (Reach ⋈ π src → dst, dst → dst2 (Edges))
    )
    """);
```

```
explosion.toList().forEach(System.out::println);
```

```
(src=A, dst=B)
(src=B, dst=C)
(src=C, dst=D)
(src=A, dst=C)
(src=E, dst=F)
(src=B, dst=D)
(src=A, dst=D)
```

That is the closure again, written out longhand — which is the useful way to read `FIX`: whatever a specialised operator does, this can express, and rather more besides.

The same tree can be built without the grammar, for a program generating recursive queries. `fixpoint` is the binder and `recRef` is the reference to it, and the reference only analyses because the binder is there to bind it:

```
import com.darkcollective.relix.ast.RelNode;

RelNode edges = project(attrs("src", "dst"), rel("Edges"));
RelNode shifted = project(
    List.of(projected(attr("src"), "dst"), projected(attr("dst"), "dst2")),
    rel("Edges"));
RelNode step = project(
    List.of(projected(attr("src")), projected(attr("dst2"), "dst")),
    naturalJoin(recRef("Reach"), shifted));

System.out.println(relix.relation(fixpoint("Reach", edges, step)).count() + " pairs");

7 pairs
```

From combinators, `fix` takes the step as a function. It is handed a relation standing for the name — everything found so far, with the base's columns qualified by that name — and returns the rows to add:

```
import com.darkcollective.relix.ast.Expr;

Relation pairs = relix.relation("π src, dst (Edges)");
Relation reach = pairs.fix("Reach", r ->
    r.join(pairs.rename("E", List.of("s", "nxt")), Expr.eq(attr("Reach.dst"), attr("E.s")))
```

```

        .project(List.of(projected(attr("Reach.src")), projected(attr("E.next"), "dst"))));
System.out.println(reach.count() + " pairs");
7 pairs

```

Repeating until it settles

FIX only ever adds rows. A computation whose rounds *replace* one another — a score refined until it stops moving, a state that evolves — is `ITERATE`. Here each node starts labelled with its own name and repeatedly takes the smallest label among itself and its neighbours, until no label changes: the connected components, found by propagation.

```

Relation labels = relix.relation("""
    ITERATE L (
         $\pi$  src  $\rightarrow$  node, src  $\rightarrow$  label (Edges)  $\cup$   $\pi$  dst  $\rightarrow$  node, dst  $\rightarrow$  label (Edges),
         $\gamma$  node, MIN(label)  $\rightarrow$  label (
            L
             $\cup$   $\pi$  dst  $\rightarrow$  node, label (L  $\bowtie$   $\pi$  src  $\rightarrow$  node, dst (Edges))
             $\cup$   $\pi$  src  $\rightarrow$  node, label (L  $\bowtie$   $\pi$  dst  $\rightarrow$  node, src (Edges))
        ) UNTIL STABLE MAX 20 ROUNDS
    """);

labels.sort(List.of(asc("node"))).toList().forEach(System.out::println);

(node=A, label=A)
(node=B, label=A)
(node=C, label=A)
(node=D, label=A)
(node=E, label=E)
(node=F, label=E)

```

From combinators it is `iterate`, whose step is a function of the previous round in the same way, and which takes the stopping rule — `rounds`, `untilStable` or `untilConverged` — as its last argument.

When recursion does not terminate

FIX is a *least* fixpoint under set semantics, so a duplicate row is never re-added and cyclic data terminates on its own. What does not terminate is a step that keeps producing genuinely new rows — counting paths around a cycle, for instance — and for that the session carries a round limit:

```

try (Relix bounded = Relix.builder().maxFixpointRounds(2).build()) {
    bounded.table("Chain", List.of("src", "dst"), List.of(
        Map.of("src", "A", "dst", "B"),
        Map.of("src", "B", "dst", "C"),
        Map.of("src", "C", "dst", "D")));

    bounded.relation("""
        FIX Reach (
            Chain,
             $\pi$  src, dst2  $\rightarrow$  dst (Reach  $\bowtie$   $\pi$  src  $\rightarrow$  dst, dst  $\rightarrow$  dst2 (Chain))
        )
    """).toList();
} catch (RuntimeException e) {
    System.out.println(e.getClass().getSimpleName());
    System.out.println(e.getMessage());
}

```

```
QueryExecutionException
FIX 'Reach' exceeded 2 iteration round(s); add a bound (e.g.  $\lambda$  n) below it, or increase --max-fixpoint-rounds
```

The limit fails loudly rather than truncating, because a silently truncated fixpoint is a wrong answer that looks like a right one.

`maxFixpointRounds` bounds how *deep* the recursion goes. Its companion, `maxMaterializedRows`, bounds how *much* one operator holds — and for a `FIX` that is every row it has derived so far, which is what a runaway recursion actually fills memory with. Reach for the round limit when you know roughly how deep a terminating answer should be and want to catch the case where you were wrong; reach for the row limit when you do not, since it bounds the memory rather than the depth. Setting both is reasonable, and either failure names the operator it stopped.

Notice the exception type. A query that does not parse or does not analyse is refused with `RelixException`, because the facade owns that refusal. A failure that only appears once rows are moving comes back as the engine's own exception — there is nothing the facade could add to it, and flattening every cause into one type would take the diagnosis away. Catch `RuntimeException` around a terminal if you mean to catch everything.

Time

Relix has real temporal types — `DATE`, `TIME`, `TIMESTAMP` and `DURATION` are `java.time` values, not strings that happen to sort — and four operators built on top of them: the AS-OF join, the interval join, `SESSIONIZE` and `DOWNSAMPLE`.

This page runs one fleet-telemetry session throughout. Two probes report a temperature every so often; each probe is recalibrated from time to time, and a reading is only trustworthy once the calibration **in effect when it was taken** has been applied. That last phrase is the AS-OF join, and it is the reason this page exists.

```
Relix relix = Relix.open();

relix.table("RawReadings", List.of("device", "at", "temp"), List.of(
    Map.of("device", "probe-1", "at", "2026-03-01T07:30:00Z", "temp", 20.1),
    Map.of("device", "probe-1", "at", "2026-03-01T08:00:00Z", "temp", 20.4),
    Map.of("device", "probe-1", "at", "2026-03-01T08:05:00Z", "temp", 20.6),
    Map.of("device", "probe-1", "at", "2026-03-01T09:45:00Z", "temp", 21.0),
    Map.of("device", "probe-1", "at", "2026-03-01T09:50:00Z", "temp", 21.2),
    Map.of("device", "probe-1", "at", "2026-03-01T13:00:00Z", "temp", 22.3),
    Map.of("device", "probe-2", "at", "2026-03-01T11:00:00Z", "temp", 18.7),
    Map.of("device", "probe-2", "at", "2026-03-01T11:20:00Z", "temp", 18.9)));

relix.table("RawCalibrations", List.of("device", "at", "offset"), List.of(
    Map.of("device", "probe-1", "at", "2026-03-01T08:00:00Z", "offset", 0.5),
    Map.of("device", "probe-1", "at", "2026-03-01T12:00:00Z", "offset", 0.7),
    Map.of("device", "probe-2", "at", "2026-02-20T09:00:00Z", "offset", -0.2)));

relix.define("""
    Readings      := {  $\pi$  device, to_timestamp(at)  $\rightarrow$  at, temp (RawReadings) };
    Calibrations := {  $\pi$  device, to_timestamp(at)  $\rightarrow$  at, offset (RawCalibrations) };
    """);

// A heading, printed compactly. Used a few times below.
class Show {
    static String heading(Schema schema) {
        return schema.columns().stream()
```

```

        .map(column -> column.name() + ":" + column.type().display())
        .collect(java.util.stream.Collectors.joining(", "));
    }
}

```

```

System.out.println>Show.heading(relix.relation("Readings").schema());
System.out.println>Show.heading(relix.relation("RawReadings").schema());

```

```

device:string, at:timestamp, temp:number
device:string, at:string, temp:number

```

A string that sorts correctly is not a timestamp

The two `to_timestamp` projections are not decoration. `table` infers its types, and it has two to choose from: a column of numbers is `NUMBER` and everything else is `STRING`. An ISO-8601 instant written as text **orders** exactly as the instant does, so a `τ` over it looks right, a `σ` over it looks right, and an AS-OF join over it finds the right row.

What it has no answer for is *how far apart* two of them are. There is no distance between two strings, so the tolerance clause below — and every `DURATION` arithmetic — is an analysis error over a text column rather than a bound that quietly does nothing. Converting once, at the edge, is what stops that reaching the query.

A column that is a `TIMESTAMP` from the start is the other route: declare the heading and hand over the rows, which is `source`.

Binding a moment

A `java.time` value goes into a query as a literal *node*, built by `timestamp`, `date`, `time` or `duration`. Nothing is formatted into text and parsed back:

```

import java.time.Instant;

Instant shiftStart = Instant.parse("2026-03-01T09:00:00Z");

Relation afterShift = relix.relation("Readings")
    .select(ge(attr("at"), timestamp(shiftStart)));

System.out.println(afterShift.render());
afterShift.toList().forEach(System.out::println);

σ at ≥ TIMESTAMP '2026-03-01T09:00:00Z' (Readings)
(device=probe-1, at=2026-03-01T09:45:00Z, temp=21)
(device=probe-1, at=2026-03-01T09:50:00Z, temp=21.2)
(device=probe-1, at=2026-03-01T13:00:00Z, temp=22.3)
(device=probe-2, at=2026-03-01T11:00:00Z, temp=18.7)
(device=probe-2, at=2026-03-01T11:20:00Z, temp=18.9)

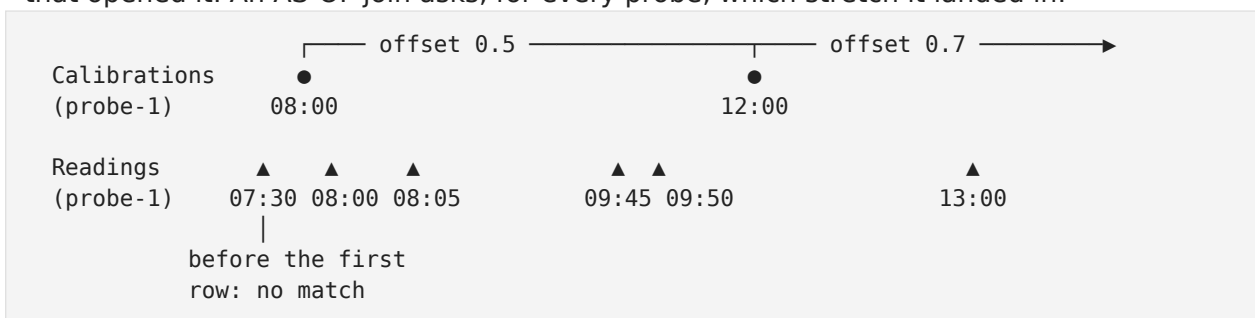
```

`timestamp(Instant)` is the one to reach for when the moment is a variable — a request parameter, a clock reading, the end of the last run. Its sibling `timestamp(String)` takes the ISO-8601 text instead, for when the moment is a constant written in the source.

AS-OF: the row in effect at this time

An AS-OF join matches each left row to the **single nearest** right row rather than to an exact one. The condition carries both halves of that: the equality keys say which rows are even candidates, and the one inequality says which direction "nearest" runs in.

Read as a picture, a history relation cuts the timeline into stretches, each ruled by the row that opened it. An AS-OF join asks, for every probe, which stretch it landed in:



Three of the boundaries in that picture are decisions rather than pictures: whether 08:00 belongs to the stretch it opens, what happens to 07:30 on the left of everything, and whether 13:00 falls off the right-hand end. All three are visible in the result below.

```
Relation corrected = relix.relation("Readings").asOfJoin(
    relix.relation("Calibrations"),
    allOf(eq(attr("Readings.device"), attr("Calibrations.device")),
        ge(attr("Readings.at"), attr("Calibrations.at"))));
```

```
System.out.println(corrected.render());
corrected.toList().forEach(System.out::println);
```

```
(Readings) ASOF (Readings.device = Calibrations.device) ^ (Readings.at ≥ Calibrations.at) (Calibrations)
(device=probe-1, at=2026-03-01T07:30:00Z, temp=20.1, device_r=NULL, at_r=NULL, offset=NULL)
(device=probe-1, at=2026-03-01T08:00:00Z, temp=20.4, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T08:05:00Z, temp=20.6, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T09:45:00Z, temp=21, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T09:50:00Z, temp=21.2, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T13:00:00Z, temp=22.3, device_r=probe-1, at_r=2026-03-01T12:00:00Z, offset=0.7)
(device=probe-2, at=2026-03-01T11:00:00Z, temp=18.7, device_r=probe-2, at_r=2026-02-20T09:00:00Z, offset=-0.2)
(device=probe-2, at=2026-03-01T11:20:00Z, temp=18.9, device_r=probe-2, at_r=2026-02-20T09:00:00Z, offset=-0.2)
```

The two sides are concatenated, and `device` occurs on both, so the right-hand copies come back as `device_r` and `at_r`. Every join in the language disambiguates a repeated name that way; a projection like the one below is how a result stops carrying them.

Five things in that result are worth reading off one at a time, because each is a boundary the operator has to decide and none of them is guessable from a signature.

The 07:30 reading precedes every calibration its probe has, so there is nothing at-or-before it. AS-OF is **left-outer by default**: the probe survives with NULLs on the right rather than disappearing.

The 08:00 reading sits **exactly on** a calibration. `≥` includes the boundary, so it matches; written `>` it would not, and the row would fall back to whatever came earlier. That is the whole difference between the two spellings and it shows up on exactly one row.

The 09:45 and 09:50 readings fall between two calibrations and take the **earlier** one — "most recent at-or-before" is the point, not "closest".

The 13:00 reading is after the last calibration, and keeps it. A history relation does not have to be extended to the end of time; the last row stays in effect.

The probe-2 readings match a calibration from nine days earlier, and *not* probe-1's calibration from three hours earlier. The equality key is what makes that true — without it the nearest row in time would win, and it belongs to another device.

Applying the match

The join's output is one row, so the correction is an ordinary projection over it:

```
corrected.project(List.of(
    projected(attr("Readings.device"), "device"),
    projected(attr("Readings.at"), "at"),
    projected(plus(attr("temp"), attr("offset")), "corrected")))
    .toList()
    .forEach(System.out::println);

(device=probe-1, at=2026-03-01T07:30:00Z, corrected=NULL)
(device=probe-1, at=2026-03-01T08:00:00Z, corrected=20.9)
(device=probe-1, at=2026-03-01T08:05:00Z, corrected=21.1)
(device=probe-1, at=2026-03-01T09:45:00Z, corrected=21.5)
(device=probe-1, at=2026-03-01T09:50:00Z, corrected=21.7)
(device=probe-1, at=2026-03-01T13:00:00Z, corrected=23)
(device=probe-2, at=2026-03-01T11:00:00Z, corrected=18.5)
(device=probe-2, at=2026-03-01T11:20:00Z, corrected=18.7)
```

The unmatched 07:30 row propagates its NULL through the arithmetic, which is the correct answer: there is no calibration, so there is no corrected reading. Filtering those rows out is `σ`, or the `inner` flag on the four-argument form below.

Tolerance: rejecting a match that is too old

A nine-day-old calibration is not really "in effect". `WITHIN` bounds how far back a match may be, and a candidate further away than that is rejected as though it were not there:

```
import java.time.Duration;
import java.util.Optional;
import com.darkcollective.relix.ast.TieBreak;

Relation fresh = relix.relation("Readings").asOfJoin(
    relix.relation("Calibrations"),
    allOf(eq(attr("Readings.device"), attr("Calibrations.device")),
        ge(attr("Readings.at"), attr("Calibrations.at"))),
    Optional.of(duration(Duration.ofDays(7))),
    false,
    TieBreak.LAST);

System.out.println(fresh.render());
fresh.toList().forEach(System.out::println);

(Readings) ASOF (Readings.device = Calibrations.device) ^ (Readings.at ≥ Calibrations.at) WITHIN DURATION 'PT168H' (Calibrations)
(device=probe-1, at=2026-03-01T07:30:00Z, temp=20.1, device_r=NULL, at_r=NULL, offset=NULL)
(device=probe-1, at=2026-03-01T08:00:00Z, temp=20.4, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T08:05:00Z, temp=20.6, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T09:45:00Z, temp=21, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T09:50:00Z, temp=21.2, device_r=probe-1, at_r=2026-03-01T08:00:00Z, offset=0.5)
(device=probe-1, at=2026-03-01T13:00:00Z, temp=22.3, device_r=probe-1, at_r=2026-03-01T12:00:00Z, offset=0.7)
(device=probe-2, at=2026-03-01T11:00:00Z, temp=18.7, device_r=NULL, at_r=NULL, offset=NULL)
(device=probe-2, at=2026-03-01T11:20:00Z, temp=18.9, device_r=NULL, at_r=NULL, offset=NULL)
```

Only the probe-2 rows change: their calibration is nine days old, past the seven-day bound, so they are rejected exactly as the 07:30 row was — the bound removes a match, it does not remove a row. (`Duration.ofDays(7)` normalises to 168 hours on the way out, which is why the rendered text says `PT168H`; it is the same duration.)

The four-argument form is where the rest of the operator's settings live. `inner` is the third argument — pass `true` and an unmatched probe is dropped instead of NULL-padded:

```
Relation freshInner = relix.relation("Readings").asOfJoin(
    relix.relation("Calibrations"),
```

```

    allOf(eq(attr("Readings.device"), attr("Calibrations.device")),
        ge(attr("Readings.at"), attr("Calibrations.at"))),
    Optional.of(duration(Duration.ofDays(7))),
    true,
    TieBreak.LAST);

```

```
System.out.println(freshInner.count() + " readings have a fresh calibration");
```

```
5 readings have a fresh calibration
```

`TieBreak` decides which row wins when two right rows share the same nearest value. The default, `LAST`, keeps the later one in input order; `FIRST` keeps the earlier. A history with no duplicate timestamps never reaches the question — which is the shape to prefer, because a tie has two defensible answers and nothing in the data says which you meant.

The tolerance is measured the same way the language's own subtraction is, so it is defined between two `TIMESTAMPS`, two `DATES` or two `DURATIONS`. A `WITHIN` over a match column with no distance — text, a number, a `TIME` whose difference wraps — is an error at analysis time.

Against a Postgres connection the join itself folds into a `LEFT JOIN LATERAL`; the tolerance is applied in the engine.

Interval joins: overlap rather than a point

AS-OF asks about an instant. When both sides carry a *range*, the question is geometric — does this outage fall inside a planned maintenance window? — and the operator states the relationship by name rather than as four endpoint comparisons.

```

relix.table("RawWindows", List.of("device", "opens", "closes"), List.of(
    Map.of("device", "probe-1", "opens", "2026-03-01T09:00:00Z",
        "closes", "2026-03-01T10:00:00Z"),
    Map.of("device", "probe-2", "opens", "2026-03-01T14:00:00Z",
        "closes", "2026-03-01T15:00:00Z")));

relix.table("RawOutages", List.of("device", "down", "up"), List.of(
    Map.of("device", "probe-1", "down", "2026-03-01T09:30:00Z",
        "up", "2026-03-01T09:40:00Z"),
    Map.of("device", "probe-1", "down", "2026-03-01T11:00:00Z",
        "up", "2026-03-01T11:15:00Z"),
    Map.of("device", "probe-2", "down", "2026-03-01T14:30:00Z",
        "up", "2026-03-01T15:30:00Z")));

relix.define("""
    Windows := { π device, to_timestamp(opens) → opens, to_timestamp(closes) → closes
        (RawWindows) };
    Outages := { π device, to_timestamp(down) → down, to_timestamp(up) → up
        (RawOutages) };
    """);

System.out.println(relix.relation("Outages").count() + " outages, "
    + relix.relation("Windows").count() + " planned windows");

```

```
3 outages, 2 planned windows
```

`intervalJoin` takes the Allen relation and the four endpoint columns. It has no equality key of its own — the relationship it tests is purely geometric — so pairing an outage with *its* own device's window is an ordinary `σ` around it:

```
import com.darkcollective.relix.ast.AllenRelation;
```

```
Relation overlapping = relix.relation("Outages")
    .intervalJoin(relix.relation("Windows"), AllenRelation.INTERSECTS,
        "Outages.down", "Outages.up", "Windows.opens", "Windows.closes")
    .select(eq(attr("Outages.device"), attr("Windows.device")));
```

```
System.out.println(overlapping.render());
overlapping.toList().forEach(System.out::println);
```

```
σ Outages.device = Windows.device ((Outages) IJOIN INTERSECTS (Outages.down, Outages.up, Windows.opens, Windows.closes) (Windows))
(device=probe-1, down=2026-03-01T09:30:00Z, up=2026-03-01T09:40:00Z, device_r=probe-1, opens=2026-03-01T09:00:00Z, closes=2026-03-01T10:00:00Z)
(device=probe-2, down=2026-03-01T14:30:00Z, up=2026-03-01T15:30:00Z, device_r=probe-2, opens=2026-03-01T14:00:00Z, closes=2026-03-01T15:00:00Z)
```

INTERSECTS is the convenience superset — any shared time at all. The thirteen named relations are finer, and the difference between two of them is exactly the question an ops team is asking. An outage that sits **strictly inside** a planned window is explained; one that starts inside and runs past the end is not:

```
Relation explained = relix.relation("Outages")
    .intervalJoin(relix.relation("Windows"), AllenRelation.DURING,
        "Outages.down", "Outages.up", "Windows.opens", "Windows.closes")
    .select(eq(attr("Outages.device"), attr("Windows.device")));
```

```
explained.project(List.of(
    projected(attr("Outages.device"), "device"),
    projected(attr("Outages.down"), "down")))
    .toList()
    .forEach(System.out::println);
```

```
(device=probe-1, down=2026-03-01T09:30:00Z)
```

Two of the three outages intersect a window; only one is DURING it. Intervals are half-open — [start, end) — and the thirteen base relations use strict endpoints, so a shared boundary is named by its own relation (MEETS, STARTS, FINISHES, EQUALS) rather than folded into DURING. The interval join is an inner join: there is no NULL-padding, and an outage with no window simply does not appear.

SESSIONIZE: splitting a stream at the gaps

SESSIONIZE walks an ordered stream and starts a new session wherever the gap to the previous row exceeds a threshold. Every input row is kept — the operator only *adds* a column — so it composes with whatever comes next.

```
Relation bursts = relix.relation("Readings")
    .sessionize("at", duration(Duration.ofMinutes(30)), List.of("device"), "burst");
```

```
System.out.println(bursts.render());
bursts.toList().forEach(System.out::println);
```

```
SESSIONIZE at GAP DURATION 'PT30M' PER device AS burst (Readings)
(device=probe-1, at=2026-03-01T07:30:00Z, temp=20.1, burst=1)
(device=probe-1, at=2026-03-01T08:00:00Z, temp=20.4, burst=1)
(device=probe-1, at=2026-03-01T08:05:00Z, temp=20.6, burst=1)
(device=probe-1, at=2026-03-01T09:45:00Z, temp=21, burst=2)
(device=probe-1, at=2026-03-01T09:50:00Z, temp=21.2, burst=2)
(device=probe-1, at=2026-03-01T13:00:00Z, temp=22.3, burst=3)
(device=probe-2, at=2026-03-01T11:00:00Z, temp=18.7, burst=1)
(device=probe-2, at=2026-03-01T11:20:00Z, temp=18.9, burst=1)
```

The 07:30 and 08:00 readings are **exactly** thirty minutes apart and stay in the same burst. The gap is strict: a new session begins when the gap is *greater than* the threshold, not

when it reaches it. Sessions are numbered from 1 within each partition, so probe-2's first burst is 1 and not 4.

Counting the bursts is then an ordinary `y` over the added column:

```
import com.darkcollective.relix.ast.AggregateOperator;

bursts.aggregate(List.of("device", "burst"),
    List.of(agg(AggregateOperator.COUNT, "temp", "readings"),
        agg(AggregateOperator.MAX, "temp", "peak")))
    .sort(asc("device"), asc("burst"))
    .toList()
    .forEach(System.out::println);

(device=probe-1, burst=1, readings=3, peak=20.6)
(device=probe-1, burst=2, readings=2, peak=21.2)
(device=probe-1, burst=3, readings=1, peak=22.3)
(device=probe-2, burst=1, readings=2, peak=18.9)
```

DOWNSAMPLE: one row per time bucket

DOWNSAMPLE aligns rows onto fixed-width buckets and consolidates each one. Where `y` groups by a value that is already in the data, this one *derives* the group from the clock:

```
import com.darkcollective.relix.ast.ConsolidationFunction;

Relation hourly = relix.relation("Readings")
    .downsample("at", "1h", ConsolidationFunction.AVG, List.of("device"));

System.out.println(hourly.render());
hourly.toList().forEach(System.out::println);

DOWNSAMPLE at BY '1h' USING AVG PER device (Readings)
(device=probe-1, bucket=2026-03-01T07:00:00Z, avg_temp=20.1)
(device=probe-1, bucket=2026-03-01T08:00:00Z, avg_temp=20.5)
(device=probe-1, bucket=2026-03-01T09:00:00Z, avg_temp=21.1)
(device=probe-1, bucket=2026-03-01T13:00:00Z, avg_temp=22.3)
(device=probe-2, bucket=2026-03-01T11:00:00Z, avg_temp=18.8)
```

bucket holds the aligned start of the window. The interval accepts either the shorthand (30s, 5m, 1h, 1d, 1w) or ISO-8601 (PT5M, P1D).

Which columns survive depends on the function, and the rule is about what the function *means* rather than about a list of names. AVG and SUM compute, so they consolidate the NUMBER columns and drop everything else. MIN and MAX compare, and comparison is defined on strings and timestamps as well as on numbers, so they keep every scalar column.

Dropping the PER device key is what makes that visible — device stops being a grouping key and becomes something to consolidate:

```
relix.relation("Readings")
    .downsample("at", "1h", ConsolidationFunction.MAX, List.of())
    .sort(asc("bucket"))
    .toList()
    .forEach(System.out::println);

(bucket=2026-03-01T07:00:00Z, max_device=probe-1, max_temp=20.1)
(bucket=2026-03-01T08:00:00Z, max_device=probe-1, max_temp=20.6)
(bucket=2026-03-01T09:00:00Z, max_device=probe-1, max_temp=21.2)
(bucket=2026-03-01T11:00:00Z, max_device=probe-2, max_temp=18.9)
(bucket=2026-03-01T13:00:00Z, max_device=probe-1, max_temp=22.3)
```

`COUNT` produces a single `count` column and keeps nothing else, which is the shape to reach for when the question is about density rather than about values.

`DOWNSAMPLE` and `SESSIONIZE` both buffer and sort their input before emitting anything, so both are evaluated in the engine and neither is folded into a backend query.

Goal-seek and optimisation

Two operators in Relix do not filter or combine rows — they *decide* something. `SOLVE` fills in a missing value by rearranging an equation, and `OPTIMIZE` chooses the rows, or the weights, that make a total as large or as small as it can be under constraints.

They are ordinary operators. A solved relation is a relation: it joins, it projects, it composes with everything else on the way to an answer.

This page runs one capacity-planning session. A platform team has a set of services, each carrying some business value and needing some CPU, spread over two regions.

```
Relix relix = Relix.open();

relix.table("Workloads", List.of("service", "region", "value", "cpus"), List.of(
    Map.of("service", "search", "region", "eu-west", "value", 60, "cpus", 4),
    Map.of("service", "ingest", "region", "eu-west", "value", 100, "cpus", 8),
    Map.of("service", "dashboards", "region", "eu-west", "value", 120, "cpus", 12),
    Map.of("service", "billing", "region", "us-east", "value", 90, "cpus", 6),
    Map.of("service", "reports", "region", "us-east", "value", 40, "cpus", 3)));

relix.relation("Workloads").toList().forEach(System.out::println);

(service=search, region=eu-west, value=60, cpus=4)
(service=ingest, region=eu-west, value=100, cpus=8)
(service=dashboards, region=eu-west, value=120, cpus=12)
(service=billing, region=us-east, value=90, cpus=6)
(service=reports, region=us-east, value=40, cpus=3)
```

SOLVE: filling in the blank

The provider's invoice arrives with holes in it. Every line obeys the same relationship — $\text{charge} = \text{cpus} \times \text{rate}$ — but which of the three is missing varies by line. A blank cell is a `NULL`, and `table` writes one wherever a row does not mention a column:

```
relix.table("Invoice", List.of("service", "cpus", "rate", "charge"), List.of(
    Map.of("service", "search", "cpus", 4, "rate", 12),
    Map.of("service", "ingest", "rate", 15, "charge", 120),
    Map.of("service", "dashboards", "cpus", 12, "charge", 216),
    Map.of("service", "billing", "cpus", 6, "rate", 12, "charge", 72),
    Map.of("service", "reports", "charge", 30)));

Relation reconciled = relix.relation("Invoice")
    .solve(attr("charge"), times(attr("cpus"), attr("rate")));

System.out.println(reconciled.render());
reconciled.toList().forEach(System.out::println);

SOLVE charge = cpus * rate (Invoice)
(service=search, cpus=4, rate=12, charge=48)
(service=ingest, cpus=8, rate=15, charge=120)
(service=dashboards, cpus=12, rate=18, charge=216)
(service=billing, cpus=6, rate=12, charge=72)
(service=reports, cpus=NULL, rate=NULL, charge=30)
```

One equation, three different directions, decided per row. `search` multiplies; `ingest` divides the charge by the rate; `dashboards` divides it by the CPU count instead. Nothing in the query says which — the operator inverts the arithmetic around whichever column is blank.

The last two rows are the rule stated from the other side. `billing` has nothing missing, and `reports` has two things missing, so neither is solved: `SOLVE` acts only where there is exactly one hole, and a row it cannot solve passes through untouched rather than failing the query. That is what makes it safe to run over a whole import.

`SOLVE` is arithmetic, not search. It rearranges $+$ $-$ $\times$ $\div$ and unary minus, per row, in pure Java — **no solver is involved**, so it works whatever else is on the classpath.

OPTIMIZE: choosing the best subset

The other half is a real search. Reserved capacity is 16 CPUs per region, and the question is which services to place on it:

```
import com.darkcollective.relix.ast.ComparisonOperator;
import com.darkcollective.relix.ast.ObjectiveSense;

Relation placed = relix.relation("Workloads").optimize(
    ObjectiveSense.MAXIMIZE,
    attr("value"),
    List.of(constraint(attr("cpus"), ComparisonOperator.LESS_EQUAL, 16)),
    List.of("region"));

System.out.println(placed.render());
placed.toList().forEach(System.out::println);

OPTIMIZE MAXIMIZE SUM(value) SUBJECT TO SUM(cpus) ≤ 16 PER region (Workloads)
(service=search, region=eu-west, value=60, cpus=4)
(service=dashboards, region=eu-west, value=120, cpus=12)
(service=billing, region=us-east, value=90, cpus=6)
(service=reports, region=us-east, value=40, cpus=3)
```

Four arguments: which way to push the objective, the per-row quantity being totalled, the constraints, and the columns a **separate problem** is solved for. `PER region` is not a grouping of the answer — it is two independent knapsacks, solved one per region, whose chosen rows are then emitted together.

The output is the chosen input rows, with the input's heading. `OPTIMIZE` is a filter that happens to think hard: it does not aggregate, and it does not add a column.

It is a search, not a heuristic

The eu-west answer is worth checking by hand, because the obvious approximation gets it wrong. Ranked by value per CPU, `search` is the best buy at 15, then `ingest` at 12.5, then `dashboards` at 10. Taking them greedily in that order fills 12 of the 16 CPUs with `search` and `ingest` for a value of 160, and `dashboards` no longer fits.

The solver returns `search` and `dashboards` instead: 16 CPUs exactly, for 180. That gap is the whole reason the operator exists rather than a `τ` and a running total.

greedy by value/cpu			the optimum		
search	4 cpu	60	search	4 cpu	60
ingest	8 cpu	100	dashboards	12 cpu	120
12 of 16 cpu			16 of 16 cpu	180	

When two subsets tie on the objective, which one comes back is the solver's choice and not something the query states. Put the tie-break into the objective if a particular answer matters.

A group that cannot be satisfied

Turn the question around — *the cheapest set of services carrying at least 200 points of value* — and one of the two regions cannot answer it at all. `us-east` has 130 points in total, so no subset of it reaches 200:

```
Relation cheapest = relix.relation("Workloads").optimize(
    ObjectiveSense.MINIMIZE,
    attr("cpus"),
    List.of(constraint(attr("value"), ComparisonOperator.GREATER_EQUAL, 200)),
    List.of("region"));
```

```
cheapest.toList().forEach(System.out::println);
```

```
(service=ingest, region=eu-west, value=100, cpus=8)
(service=dashboards, region=eu-west, value=120, cpus=12)
```

`eu-west` is solved — `ingest` and `dashboards`, 220 points for 20 CPUs, the cheapest way to clear the bar. `us-east` contributes **no rows**, because there is nothing it could contribute: the constraint is unsatisfiable there.

That is the outcome worth being precise about, because a search has three of them and only two have an obvious shape:

Outcome	What it means	What you see
Solved	An optimum was found	The chosen rows
Infeasible	No assignment satisfies the constraints	No rows for that group
Undetermined	The search stopped without deciding	An error

The first two are answers and the third is a failure, and the empty result is the trap: a group that is genuinely infeasible and a search that ran out of road both contribute nothing. Relix keeps them apart — an infeasible group is skipped, an undetermined search raises — so an empty group in your result is a fact about the data rather than a symptom you have to go looking for.

Allocation: weights instead of a subset

`ALLOCATE` swaps the binary in-or-out decision for a continuous one. Each row gets a weight in a range, every row comes back, and the weight arrives as a new column. It is the shape for splitting something divisible — a budget, a traffic share, a pool of capacity.

The combinator covers the operator, and the allocation clause is one of the places writing the expression is easier to read than a sixth argument would be:

```
Relation split = relix.relation("""
    OPTIMIZE ALLOCATE (0.0, 0.5) MAXIMIZE SUM(value)
    SUBJECT TO SUM(1) = 1.0 -> share PER region (Workloads)
    """);
```

```
split.sort(asc("region"), desc("share")).toList().forEach(System.out::println);
```

```
(service=ingest, region=eu-west, value=100, cpus=8, share=0.5)
(service=dashboards, region=eu-west, value=120, cpus=12, share=0.5)
```

```
(service=search, region=eu-west, value=60, cpus=4, share=0)
(service=billing, region=us-east, value=90, cpus=6, share=0.5)
(service=reports, region=us-east, value=40, cpus=3, share=0.5)
```

Read it as: split next quarter's pool across each region's services, giving no single service more than half of it, in whatever proportion carries the most value. In eu-west that is half to dashboards and half to ingest, and nothing to search — the cap is what stops the whole pool going to the single best row, which is where an unconstrained linear objective always ends up. us-east has only two services, so the halves are forced.

`SUM(1) = 1.0` is the constraint that makes the weights a *share*: each row contributes a coefficient of 1, so the sum of the chosen weights is pinned to one whole pool.

Composing with the rest of a query

Nothing above is a terminal. The chosen rows are a relation, so the plan can be joined, aggregated and sorted like anything else — here, what each region's placement is worth:

```
import com.darkcollective.relix.ast.AggregateOperator;

placed.aggregate(List.of("region"),
    List.of(agg(AggregateOperator.SUM, "value", "placed_value"),
            agg(AggregateOperator.SUM, "cpus", "cpus_used")))
    .sort(asc("region"))
    .toList()
    .forEach(System.out::println);
```

```
(region=eu-west, placed_value=180, cpus_used=16)
(region=us-east, placed_value=130, cpus_used=9)
```

OPTIMIZE buffers its input and never folds into a backend query — the search happens in the engine, over rows the engine has read. A selection that fixes a PER key to a constant is pushed *below* the operator, so asking for one region's answer solves one region's problem rather than both and discarding one.

Checking that a solver is installed

OPTIMIZE needs an installed mathematical-programming solver, and so does COVER ... EXACT. One ships inside the published artifact, so a session assembled from that dependency has one. A program that assembles its own classpath can end up without.

Providers are found by a ServiceLoader scan, which no compiler can check, so the way to be sure is to ask the running session. `relix.version` lists what is actually present:

```
relix.relation("π component, version (σ kind = 'solver' (relix.version))")
    .toList()
    .forEach(System.out::println);

(component=ojAlgo, version=<version>)
```

With none installed, that query returns nothing, and OPTIMIZE raises **before any input is read** — the check is made while planning, so the failure arrives at once rather than part-way through a scan. Every other operator is unaffected, SOLVE included: inverting an equation is arithmetic and needs nothing installed at all.

Nested data

A Relix column can hold a struct or an array, not only a scalar. The algebra is defined over that from the bottom up — flat relational algebra is the all-scalar special case — so nesting is not a document mode bolted on beside the operators. It is the same operators, with a wider idea of what a value is.

This page is about **producing** nested shapes: constructing them, gathering them, flattening them, and folding a hierarchy into one. Reading them back out in Java — the struct and array accessors on a row — is [Reading results](#).

```
Relix relix = Relix.open();

relix.table("Employees", List.of("id", "manager_id", "name", "team", "salary"), List.of(
    Map.of("id", 1, "manager_id", null, "name", "Ada", "team", "exec", "salary", 200),
    Map.of("id", 2, "manager_id", 1, "name", "Grace", "team", "platform", "salary", 150),
    Map.of("id", 3, "manager_id", 1, "name", "Lin", "team", "data", "salary", 150),
    Map.of("id", 4, "manager_id", 2, "name", "Edsger", "team", "platform", "salary", 120),
    Map.of("id", 5, "manager_id", 2, "name", "Barbara", "team", "platform", "salary", 130),
    Map.of("id", 6, "manager_id", 3, "name", "Alan", "team", "data", "salary", 110)));

relix.relation("Employees").toList().forEach(System.out::println);

(id=1, manager_id=NULL, name=Ada, team=exec, salary=200)
(id=2, manager_id=1, name=Grace, team=platform, salary=150)
(id=3, manager_id=1, name=Lin, team=data, salary=150)
(id=4, manager_id=2, name=Edsger, team=platform, salary=120)
(id=5, manager_id=2, name=Barbara, team=platform, salary=130)
(id=6, manager_id=3, name=Alan, team=data, salary=110)
```

Ada's row omits `manager_id`, so hers is NULL — which is what makes her the root of the tree at the end of this page.

Building a nested value

Two constructors put nesting into a projection. `structOf(field(name, expr), ...)` builds a struct and `arrayOf(expr, ...)` builds an array, and both are ordinary operands: they take expressions, they can be nested inside each other, and they can be aliased like any other projected column.

```
Relation boxed = relix.relation("Employees").project(List.of(
    projected(attr("id")),
    projected(structOf(field("name", attr("name")),
        field("pay", structOf(field("base", attr("salary")),
            field("banded", times(attr("salary"), num(1.1)))))),
    "person")));

System.out.println(boxed.render());
boxed.limit(2).toList().forEach(System.out::println);

π id, {name: name, pay: {base: salary, banded: salary * 1.1}} → person (Employees)
(id=1, person={name: Ada, pay: {base: 200, banded: 220}})
(id=2, person={name: Grace, pay: {base: 150, banded: 165}})
```

The heading is known before anything runs, and it is nested too — the constructors are expressions, so their type is inferred from what they are built out of, all the way down:

```
System.out.println(boxed.schema().columns().stream()
    .map(c -> c.name() + ":" + c.type().display())
    .toList());

[id:number, person:struct{name: string, pay: struct{base: number, banded: number}}]
```

banded is a number because `salary * 1.1` is, and the analyser knows that two levels down. Nesting a value does not put it beyond type inference.

`arrayOf` is the other constructor, and it takes expressions in the same way — an array is a list of values, not a list of columns:

```
relix.relation("Employees")
    .project(List.of(projected(attr("name")),
                          projected(arrayOf(attr("team"), func("LCase", attr("name"))), "tags")))
    .limit(3)
    .toList()
    .forEach(System.out::println);

(name=Ada, tags=[exec, ada])
(name=Grace, tags=[platform, grace])
(name=Lin, tags=[data, lin])
```

COLLECT and UNNEST

`COLLECT` is the aggregate that does not reduce. Where `SUM` turns a group's values into one number, `COLLECT` turns them into one array, so the group survives as a value instead of collapsing into a statistic:

```
import com.darkcollective.relix.ast.AggregateOperator;

Relation teams = relix.relation("Employees")
    .aggregate(List.of("team"), List.of(agg(AggregateOperator.COLLECT, "name", "members")));

System.out.println(teams.render());
teams.toList().forEach(System.out::println);

γ team, COLLECT(name) → members (Employees)
(team=exec, members=[Ada])
(team=platform, members=[Grace, Edsger, Barbara])
(team=data, members=[Lin, Alan])
```

`unnest` is its inverse: it flattens an array column into one row per element, keeping the rest of the row alongside each one.

```
teams.unnest("members").toList().forEach(System.out::println);

(team=exec, members=Ada)
(team=platform, members=Grace)
(team=data, members=Lin)
(team=platform, members=Edsger)
(team=platform, members=Barbara)
(team=data, members=Alan)
```

That round trip is not a coincidence, and the engine knows it. Gathering a group and immediately flattening it again is a great deal of work to arrive back where you started, so the optimizer removes both operators and leaves the projection they amount to:

```
System.out.println(teams.unnest("members").optimized().render());

π team, name → members (Employees)
```

Which is a good reason to write the obvious thing. The law holds in the algebra, so stating it plainly costs nothing at run time.

Positions, and rows the array cannot fill

Flattening loses two things unless you ask for them: where each element sat, and what happened to a row whose array was empty. The three-argument form restores both.

```
import java.util.Optional;

Relation ranked = teams.unnest("members", false, Optional.of("seat"));

System.out.println(ranked.render());
ranked.toList().forEach(System.out::println);

μ members WITH ORDINALITY seat (γ team, COLLECT(name) → members (Employees))
(team=exec, members=Ada, seat=1)
(team=platform, members=Grace, seat=1)
(team=platform, members=Edsger, seat=2)
(team=platform, members=Barbara, seat=3)
(team=data, members=Lin, seat=1)
(team=data, members=Alan, seat=2)
```

`seat` is the 1-based position within the array. The `false` is the *outer* flag: left as it is, a row whose array is empty contributes nothing, and set to `true` it survives once with a `NULL` where an element would have gone — the same choice an outer join makes, applied to a column instead of a relation.

COLLECT and NULLs

Every other aggregate follows the SQL rule and skips a row whose argument is `NULL`. `COLLECT` deliberately does not — an array that silently dropped its holes would not be the group any more:

```
relix.relation("Employees")
    .aggregate(List.of(), List.of(agg(AggregateOperator.COLLECT, "manager_id", "managers"),
                                   agg(AggregateOperator.COUNT, "manager_id", "counted")))
    .toList()
    .forEach(System.out::println);

(managers=[NULL, 1, 1, 2, 2, 3], counted=5)
```

Six employees, five of whom have a manager. `COUNT` reports five because it skips Ada's `NULL`; `COLLECT` returns six elements because it keeps it.

TREE: folding a hierarchy

`COLLECT` gathers one level. `TREE` follows a parent pointer to the bottom and gathers all of them, turning an adjacency list into a forest of nested documents in a single pass.

adjacency (flat)			forest (nested)	
id	manager_id			
1	—	⇒	Ada	
2	1		├	Grace
3	1		├	├ Edsger
4	2		├	└ Barbara
5	2		└	Lin
6	3		└	└ Alan

```
Relation org = relix.relation("Employees").tree("id", "manager_id", "reports");

System.out.println(org.render());
System.out.println(org.count() + " root(s)");

// Walk the forest the way the operator built it.
class Chart {
    static void print(String indent, Value node) {
        Map<String, Value> fields = ((com.darkcollective.relix.value.StructValue) node).fields();
        System.out.println(indent + fields.get("name").asDisplayString());
    }
}
```

```

        ((com.darkcollective.relix.value.ArrayValue) fields.get("reports")).elements()
            .forEach(child -> print(indent + "    ", child));
    }
}

Tuple root = org.toList().getFirst();
System.out.println(root.string("name"));
root.array("reports").forEach(child -> Chart.print("    ", child));

```

```

TREE id BY manager_id AS reports (Employees)
1 root(s)
Ada
    Grace
        Edsger
        Barbara
    Lin
        Alan

```

One output row per **root**, and a root is a row whose parent pointer is NULL or points at an id that is not in the relation — so a forest with several roots is normal, and a relation with none at all is a cycle rather than an empty answer. Every node carries its own columns plus its `reports` array, recursively; a leaf carries an empty one.

Both malformed shapes are refused rather than guessed at. A duplicate key is a key violation, and a cycle is an error rather than a loop:

```

relix.table("Cyclic", List.of("id", "manager_id", "name"), List.of(
    Map.of("id", 1, "manager_id", 2, "name", "Ada"),
    Map.of("id", 2, "manager_id", 1, "name", "Grace")));

try {
    relix.relation("Cyclic").tree("id", "manager_id", "reports").toList();
} catch (RuntimeException e) {
    System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
}

```

```

QueryExecutionException: TREE: cycle detected in the 'id' -> 'manager_id' graph; a tree requires an acyclic, single-parent forest

```

Sibling order is input order unless you ask for something else, and asking is a clause rather than an argument — which is where writing the expression is easier to read than a fifth parameter:

```

Relation ordered = relix.relation("""
    TREE id BY manager_id ORDER name DESC AS reports (Employees)
    """);

Tuple desc = ordered.toList().getFirst();
System.out.println(desc.string("name"));
desc.array("reports").forEach(child -> Chart.print("    ", child));

```

```

Ada
    Lin
        Alan
    Grace
        Edsger
        Barbara

```

TREE has to see the whole relation before it can emit a root, so it buffers, and it is never folded into a backend query.

Declaring a nested schema

Everything above builds nesting out of flat rows. The other direction is a source that is already nested — a document store, a JSON payload — and there the interesting decision is whether to say what the shape is.

A declared schema nests exactly as the values do: `{ ... }` is a struct and `[ ... ]` an array, composing to any depth. Here it is against a connector supplying the documents, which is the shape [Bringing your own data](#) sets up:

```
import com.darkcollective.relix.processor.Row;
import com.darkcollective.relix.processor.connector.ConnectorConfig;
import com.darkcollective.relix.processor.connector.RelixConnector;
import com.darkcollective.relix.value.ArrayValue;
import com.darkcollective.relix.value.NumberValue;
import com.darkcollective.relix.value.StringValue;
import com.darkcollective.relix.value.StructValue;
import java.util.Set;

class Directory implements RelixConnector {
    public Set<String> handles() {
        return Set.of("directory");
    }

    // A struct keeps the field order it is given, so build it in one.
    static StructValue struct(String k1, Value v1, String k2, Value v2) {
        Map<String, Value> fields = new java.util.LinkedHashMap<>();
        fields.put(k1, v1);
        fields.put(k2, v2);
        return new StructValue(fields);
    }

    public Stream<Row> open(ConnectorConfig config, String table, Schema schema) {
        return Stream.of(Row.of(schema,
            new StringValue("Grace"),
            struct("city", new StringValue("Cambridge"),
                "country", new StringValue("GB")),
            new ArrayValue(List.of(
                struct("name", new StringValue("Java"), "years", NumberValue.of("9")),
                struct("name", new StringValue("SQL"), "years", NumberValue.of("4"))))),
        );
    }
}

relix.connector(new Directory());
relix.define("""
    connection people from directory { };
    source Profiles from people { table: "profiles", schema: {
        name: STRING,
        location: { city: STRING, country: STRING },
        skills: [{ name: STRING, years: NUMBER } ] } };
""");

relix.relation("Profiles").toList().forEach(System.out::println);

(name=Grace, location={city: Cambridge, country: GB}, skills=[{name: Java, years: 9}, {name: SQL, years: 4}])
```

A path through a declared column is typed, which is the whole point of declaring one. A dotted reference reaches inside the struct and infers the field's own type:

```
Relation located = relix.relation("π name, location.city → city (Profiles)");

System.out.println(located.schema().columns().stream()
    .map(c -> c.name() + ":" + c.type().display()).toList());
located.toList().forEach(System.out::println);
```

```
[name:string, city:string]
(name=Grace, city=Cambridge)
```

`city` is a `STRING`, not an `ANY`. That difference is not cosmetic: it is what lets the analyser keep checking your query once the data stops being flat. A field the declaration does not mention is caught while analysing rather than arriving as a `NULL` at run time:

```
relix.validate("Bad := { π location.postcode → pc (Profiles) };")
    .forEach(d -> System.out.println(d.severity() + ": " + d.message()));
```

```
ERROR: Projection π: attribute 'location.postcode' not found in input schema (available: name, location, skills)
```

The array is the other half. `μ` over an array of structs gives one row per element, with the element in place of the array — and the declaration types those elements too:

```
Relation flattened = relix.relation("Profiles").unnest("skills");
```

```
System.out.println(flattened.schema().columns().stream()
    .map(c -> c.name() + ":" + c.type().display()).toList());
flattened.toList().forEach(System.out::println);
```

```
[name:string, location:struct{city: string, country: string}, skills:struct{name: string, years: number}]
(name=Grace, location={city: Cambridge, country: GB}, skills={name: Java, years: 9})
(name=Grace, location={city: Cambridge, country: GB}, skills={name: SQL, years: 4})
```

Which means the element's fields are reachable from the query, by the same dotted path a declared struct column takes — the `unnest` changed the column's *type*, not its status:

```
Relation skills = relix.relation("""
    π name, skills.name → skill, skills.years → years
    (τ skills.years DESC (μ skills (Profiles)))
    """);
```

```
System.out.println(skills.schema().columns().stream()
    .map(c -> c.name() + ":" + c.type().display()).toList());
skills.toList().forEach(System.out::println);
```

```
[name:string, skill:string, years:number]
(name=Grace, skill=Java, years=9)
(name=Grace, skill=SQL, years=4)
```

`skills.name` is worth a second look. `name` is also a top-level column here, and the path still reads the *skill's* name — a dotted reference is a relation qualifier first and a struct path second, so `skills.name` can only be a path, because no column called `name` comes from a relation called `skills`.

The same value is reachable from Java instead, through the struct accessor on the row, which is the route to reach for when the shape is being explored rather than queried:

```
flattened.toList().forEach(row -> System.out.println(
    row.string("name") + " knows " + row.struct("skills").get("name").asDisplayString()
    + " (" + row.struct("skills").get("years").asDisplayString() + " years)"));
```

```
Grace knows Java (9 years)
Grace knows SQL (4 years)
```

Declare `ANY` where the shape genuinely varies — schema-on-read is what a document store often wants, and an `ANY` column still reads back through `struct` and `array` at run time. What it cannot do is tell you at analysis time that you misspelled a field.

3. Getting data in

Page	What it covers
Bringing your own data and code	Rows held, rows produced per scan, a connector of your own, a function library of your own
Generating data	Generator relations and their bounds; COVER and all-pairs suites; seeded sampling and what a seed buys
Against a database	Binding a DataSource, introspection, what is pushed down and what is not, joining your data to theirs, connection lifecycle, and what a federated query does not promise
Secrets in declarations	`\${NAME}` placeholders resolved when a query runs, so a token never enters the model; a missing value; declarations built in Java
Sandboxing a session	A closed sandbox for untrusted text: internal declarations only, external ones the sandbox permits, output and input limits, the configuration file
Working without a database	What composes offline; unresolved names; capturing a catalog snapshot and replaying it

Bringing your own data and code

"Here is my in-memory data, joined against my database table" is one of the reasons to embed a query engine at all. Five calls cover it. The first three differ in one axis — how much the program hands over, and how late — and the last two hand over code rather than rows.

Call	For
<code>table(name, rows)</code>	Rows already in memory — reference data, fixtures, a result computed elsewhere
<code>source(name, schema, supplier)</code>	Anything lazy or large: a stream the program produces per scan
<code>materialize(name, relation)</code>	A result this session computed, kept under a name and queried again
<code>connector(connector)</code>	A backend of the program's own, read through the connector SPI
<code>functions(library)</code>	Scalar or aggregate functions of the program's own

Everything on this page composes with everything else a session declares. A supplied relation is a relation: it joins against a database table, it optimises, it pushes what can be pushed and evaluates the rest in-engine.

Rows you already hold

`table` is the small end, and [Getting started](#) opens with it. State the heading, then the rows as maps. Each row is read **by name**, so the maps need no order of their own and `Map.of` is safe here:

```
Relix relix = Relix.open();
```

```

relix.table("Carriers", List.of("carrier", "mode"), List.of(
    Map.of("carrier", "Rail", "mode", "ground"),
    Map.of("carrier", "Air", "mode", "air")));

System.out.println(relix.relation("Carriers").count() + " carriers");

2 carriers

```

Types are inferred — a column of numbers is `NUMBER`, everything else is `STRING`. A column a row does not mention is `NULL` for that row, which is how an optional column is expressed; a key naming no column is refused rather than silently dropped. With the heading stated, a table with no rows at all is expressible too.

There is a two-argument form that takes the heading from the first row's key order, for when the maps are built rather than written out. The first row then has to be a map that *has* an order — a `LinkedHashMap`, not a `Map.of`, whose iteration order is randomised per JVM, so a heading taken from one would differ between runs of the same program. That is refused rather than accepted arbitrarily; a one-column row is exempt, since one key is in only one order.

That inference is the reason `table` is the small end. When a column has to be a real `TIMESTAMP`, or the rows are too many to hold, the heading has to be declared — which is the next call.

Rows you produce on demand

`source` takes a heading and a supplier. The heading is *declared*, so a column is the type you gave it; the supplier is called **once per scan**, so nothing is produced until a query reads it, and a second query gets a second stream.

```

import com.darkcollective.relix.processor.Row;
import com.darkcollective.relix.symbol.ColumnDefinition;
import com.darkcollective.relix.symbol.ScalarType;
import com.darkcollective.relix.value.NumberValue;
import com.darkcollective.relix.value.StringValue;

Schema shipments = new Schema(List.of(
    new ColumnDefinition("order_id", ScalarType.NUMBER),
    new ColumnDefinition("carrier", ScalarType.STRING)));

relix.source("Shipments", shipments, () -> Stream.of(
    Row.of(shipments, NumberValue.of("1"), new StringValue("Rail")),
    Row.of(shipments, NumberValue.of("2"), new StringValue("Air"))));

for (Row row : relix.relation("Shipments").toList()) {
    System.out.println(row.get("order_id").asDisplayString()
        + " " + row.get("carrier").asDisplayString());
}

1 Rail
2 Air

```

Rows are built with `Row.of(schema, values)`, in the heading's column order. The engine closes the stream it is handed, so a supplier holding a resource releases it through `Stream.onClose`.

The engine treats a supplied source as **finite**, so the collecting terminals read it to the end. A supplier whose stream does not terminate is read with `stream()`, which is lazy and stops

the moment you stop pulling.

A result you want to keep

The three calls above bring in data from outside. `materialize` brings in data the session computed itself: it takes a relation, runs it, and registers the rows under a name, so the next query reads a result instead of computing it again.

```
relix.materialize("ByCarrier",
    relix.relation("γ carrier, COUNT(*) → shipments (Shipments)"));

for (Row row : relix.relation("τ carrier (ByCarrier)").toList()) {
    System.out.println(row.get("carrier").asDisplayString()
        + " " + row.get("shipments").asDisplayString());
}
```

```
Air 1
Rail 1
```

This is the one call on a session that runs anything. Everywhere else a declaration is a declaration and building a relation executes nothing; here the argument is drained before the call returns, because computing once and reading the result many times is the whole point of it. A relation that provably never ends is refused, exactly as `toList()` refuses one.

What it registers carries the relation's **own** heading — which is the difference between this and draining the rows into maps and passing them to `table`. There the types would be inferred back to `NUMBER` or `STRING`, so a `TIMESTAMP` column would come back a string.

The rows are a snapshot. Whatever they were computed from may move on; the named relation holds what was true when it was drained.

A backend of your own

`connector` installs an implementation of the connector SPI — the same seam the shipped CSV, JSON, HTTP and JDBC connectors are discovered through. The point of doing it this way rather than through a `META-INF/services` declaration is that a program exercising the SPI inside itself should not have to package itself as a service provider.

A connector claims one or more **type tokens**. Declare a connection of that type, and the session's sources over it are read through your code:

```
import com.darkcollective.relix.processor.connector.ConnectorConfig;
import com.darkcollective.relix.processor.connector.RelixConnector;
import java.util.Set;

class Ledger implements RelixConnector {
    public Set<String> handles() {
        return Set.of("ledger");
    }

    public Stream<Row> open(ConnectorConfig config, String table, Schema schema) {
        // Both halves of the declaration arrive here: the connection's own properties as
        // the config, and the source's `table:` as the name to read.
        return Stream.of(Row.of(schema, NumberValue.of("1"), NumberValue.of("100"),
            new StringValue(config.require("origin") + "/" + table)));
    }
}
```

```

relix.connector(new Ledger());
relix.define("""
    connection books from ledger { origin: "in-process" };
    source Payments from books { table: "payments",
        schema: { order_id: NUMBER, paid: NUMBER, ledger_ref: STRING } };
    """);

System.out.println(relix.relation("Payments").toList()
    .getFirst().get("ledger_ref").asDisplayString());

```

in-process/payments

An installed connector is consulted before the discovered ones, so it wins a token a shipped connector also claims. It is **not** closed with the session: it is yours, exactly as a `DataSource` you bind is.

Functions of your own

`functions` installs a function library, and the engine calls it exactly as it calls the bundled one — the same resolution during analysis, the same invocation during execution, the same eligibility for pushdown when the library says how to spell it.

The properties a signature declares are part of that. `PURE` and `DETERMINISTIC` below are what let the call be folded into a backend query at all: a function whose value depends on when it runs is evaluated here, however it spells itself, because the backend would answer from its own clock rather than the session's.

```

import com.darkcollective.relix.function.FunctionContext;
import com.darkcollective.relix.function.FunctionLibrary;
import com.darkcollective.relix.function.FunctionSignature;
import com.darkcollective.relix.function.ScalarFunction;
import com.darkcollective.relix.function.StrictScalarFunction;
import com.darkcollective.relix.symbol.FunctionProperty;
import com.darkcollective.relix.symbol.ParameterDefinition;
import java.math.BigDecimal;

class WithVat implements StrictScalarFunction {
    public FunctionSignature signature() {
        return FunctionSignature.of("WithVat", ScalarType.NUMBER, "math",
            Set.of(FunctionProperty.PURE, FunctionProperty.DETERMINISTIC),
            new ParameterDefinition("amount", ScalarType.NUMBER));
    }

    public Value invoke(FunctionContext context, List<Value> arguments) {
        BigDecimal amount = ((NumberValue) arguments.getFirst()).value();
        return new NumberValue(amount.multiply(new BigDecimal("1.20")));
    }
}

class Pricing implements FunctionLibrary {
    public String name() {
        return "pricing";
    }

    public List<ScalarFunction> scalarFunctions() {
        return List.of(new WithVat());
    }
}

relix.functions(new Pricing());

```

```
System.out.println(relix.relation("n WithVat(paid) → gross (Payments)").toList()
    .getFirst().get("gross").asDisplayString());
```

120

Installed libraries are merged with the discovered ones rather than replacing them, so the bundled functions are still there and an installed name wins a clash.

What this does not add

There is no write path. Relix reads: a supplied relation is an input to a query, and nothing here sends anything back — `materialize` included, whose rows land in this session's memory and nowhere else.

Generating data

Most relations are read. A few are **computed** — their rows are produced by the engine rather than stored anywhere — and they answer the same reader question in three different ways: *how many rows am I going to get, and can I get them again?*

	Where the rows come from	What bounds them	Can you get them again?
Generators	A named series the engine knows how to enumerate	Their arguments, or nothing at all	Always — they are functions of their arguments
COVER	A subset of a candidate set	Its construction: enough rows to cover every combination	Always — the choice is deterministic
Sampling	A random subset of its input	A probability, or an exact count	Only with a seed

```
Relix relix = Relix.open();
```

Generator relations

A generator is declared like any other source, naming the series rather than a file:

```
relix.define("""
    source Small from generator { name: "Range", lo: "1", hi: "6" };
    """);
```

```
relix.relation("Small").toList().forEach(System.out::println);
```

```
(n=1)
(n=2)
(n=3)
(n=4)
(n=5)
(n=6)
```

`Range` takes `lo`, `hi` and an optional `step`, and produces a single `n` column. It is a relation in every ordinary sense — it joins, it filters, it feeds a `x` — which is what makes it useful as the spine of something else: a calendar to left-join sparse readings onto, a set of ids to probe with, a row source for a test fixture.

Two of the built-in generators have no last row at all. `Naturals` and `Primes` enumerate forever, which is safe to have because the engine knows they do:

```
relix.define("""
    source Primes from generator { name: "Primes" };
    """);

try (Stream<Tuple> primes = relix.relation("Primes").stream()) {
    System.out.println(primes.limit(8).map(row -> row.longValue("n").toString()).toList());
}
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
```

`stream()` is what an unbounded relation is *for*: the engine is pull-based end to end, so eight rows is eight rows of work. The collecting terminals refuse such a relation instead of hanging, and [Reading results](#) covers that side of it.

A bound placed *above* an unbounded generator does not have to be a post-filter. The optimizer pushes it into the generator, so the enumeration itself stops:

```
System.out.println(relix.relation("σ n ≤ 20 (Primes)").optimized().render());
System.out.println(relix.relation("σ n ≤ 20 (Primes)").toList().size() + " primes ≤ 20");
```

```
σ n ≤ 20 (Primes (produce while n ≤ 20))
8 primes ≤ 20
```

The relation went from one that never ends to one that does, which is why a collecting terminal is willing to read it.

COVER: a test suite that is small on purpose

`COVER t` keeps just enough rows that every combination of `t` column values still appears somewhere. It is the operator behind all-pairs testing, and it is worth a worked example because the saving is not small.

A checkout flow has to work across browsers, operating systems, locales, payment methods and network conditions. Each factor is a relation; the exhaustive matrix is their product:

```
relix.table("Browsers", List.of("browser"), List.of(
    Map.of("browser", "Chrome"), Map.of("browser", "Firefox"),
    Map.of("browser", "Safari"), Map.of("browser", "Edge")));

relix.table("Systems", List.of("os"), List.of(
    Map.of("os", "Windows"), Map.of("os", "macOS"), Map.of("os", "Linux"),
    Map.of("os", "iOS"), Map.of("os", "Android")));

relix.table("Locales", List.of("locale"), List.of(
    Map.of("locale", "en-GB"), Map.of("locale", "de-DE"), Map.of("locale", "fr-FR"),
    Map.of("locale", "ja-JP"), Map.of("locale", "pt-BR")));

relix.table("Payments", List.of("payment"), List.of(
    Map.of("payment", "card"), Map.of("payment", "paypal"),
    Map.of("payment", "applepay"), Map.of("payment", "banktransfer")));

relix.table("Networks", List.of("network"), List.of(
    Map.of("network", "wifi"), Map.of("network", "4g"), Map.of("network", "3g")));

relix.define("""
    Matrix := { Browsers × Systems × Locales × Payments × Networks };
    """);
```

```
System.out.println(relix.relation("Matrix").count() + " exhaustive combinations");  
1200 exhaustive combinations
```

Twelve hundred runs of an end-to-end suite is not a thing anyone schedules. Before shrinking it, though, some of those rows are not runnable at all — Safari does not run on Windows, and Apple Pay is not offered off Apple platforms. Those are ordinary selections:

```
relix.define("""  
    Feasible := { σ (browser ≠ "Safari" v os ∈ {"macOS", "iOS"})  
                  ∧ (payment ≠ "applepay" v os ∈ {"macOS", "iOS"}) (Matrix) };  
    """);  
  
System.out.println(relix.relation("Feasible").count() + " runnable combinations");  
885 runnable combinations
```

That the constraints are just a `σ` is the part worth noticing. COVER derives what it has to cover from the candidate set it is given, so a combination you filtered out is never demanded — there is no separate constraint language, and no way for the constraints and the coverage goal to disagree.

Now the suite. `cover(2)` asks for every *pair* of values to appear together somewhere:

```
Relation suite = relix.relation("Feasible").cover(2);  
  
System.out.println(suite.render());  
System.out.println(suite.count() + " runs cover every feasible pair");  
  
COVER 2 (Feasible)  
30 runs cover every feasible pair
```

Thirty runs instead of eight hundred and eighty-five, and the claim is checkable rather than trusted. Every pair of values that occurs in the candidate set has to occur in the suite, so the pairs the suite is *missing* are a set difference — and there are none, for any of the ten pairs of columns:

```
List<String> factors = List.of("browser", "os", "locale", "payment", "network");  
Relation feasible = relix.relation("Feasible");  
  
for (int a = 0; a < factors.size(); a++) {  
    for (int b = a + 1; b < factors.size(); b++) {  
        String x = factors.get(a);  
        String y = factors.get(b);  
        System.out.println(x + " × " + y + " – uncovered: "  
            + feasible.project(x, y).difference(suite.project(x, y)).count()  
            + " of " + feasible.project(x, y).distinct().count());  
    }  
}  
  
browser × os – uncovered: 0 of 17  
browser × locale – uncovered: 0 of 20  
browser × payment – uncovered: 0 of 16  
browser × network – uncovered: 0 of 12  
os × locale – uncovered: 0 of 25  
os × payment – uncovered: 0 of 17  
os × network – uncovered: 0 of 15  
locale × payment – uncovered: 0 of 20  
locale × network – uncovered: 0 of 15  
payment × network – uncovered: 0 of 12
```

That is the check written in the language itself, which is the point of the operator returning a relation rather than a report: "is every pair covered?" is a difference, and an empty difference is the answer.

Strength is the dial between "every value at least once" and "every distinct row". At $t=1$ the suite only has to exercise each individual value; at $t=3$ every triple has to appear; and at a t equal to the number of columns it degenerates to δ :

```
for (int t = 1; t <= 3; t++) {  
    System.out.println("t=" + t + " → " + relix.relation("Feasible").cover(t).count() + " runs");  
}
```

```
t=1 → 5 runs  
t=2 → 30 runs  
t=3 → 108 runs
```

The selection is greedy with a deterministic tie-break, so the same input gives the same suite every time. Ordering the input is how you bias which candidates win — put a t under the **COVER** and the rows you would rather run are the ones picked first. `cover(t, true)` asks for a **minimal** suite instead of a greedy one; that is a search, and it needs the installed solver [Goal-seek and optimisation](#) describes.

Sampling

Sampling answers the same size question with randomness rather than construction. There are two operators, and they differ in which side of the sample you get to fix.

`sample(p)` is Bernoulli: each row is kept independently with probability p , so the *rate* is fixed and the count varies. `sampleReservoir(n)` fixes the count instead: exactly n rows, chosen uniformly, or all of them if there are fewer than n .

Both take a seed, and everything below passes one — an unseeded sample cannot have a printed result at all, which is the most direct possible demonstration of the difference:

```
Relation tenth = relix.relation("Feasible").sample(0.1, 2026);  
Relation five = relix.relation("Feasible").sampleReservoir(5, 2026);
```

```
System.out.println(tenth.render());  
System.out.println(tenth.count() + " rows from a 10% sample of "  
    + relix.relation("Feasible").count());  
System.out.println(five.render());  
System.out.println(five.count() + " rows from the reservoir");
```

```
SAMPLE 0.1 SEED 2026 (Feasible)  
86 rows from a 10% sample of 885  
SAMPLE 5 ROWS SEED 2026 (Feasible)  
5 rows from the reservoir
```

The same seed over the same input picks the same rows — not merely within one run, but for any relation carrying that seed, however it was built:

```
Relation composed = relix.relation("Feasible").sampleReservoir(3, 7);  
Relation written = relix.relation("SAMPLE 3 ROWS SEED 7 (Feasible)");  
  
System.out.println("same rows: " + composed.toList().equals(written.toList()));  
composed.toList().forEach(row -> System.out.println(row.string("browser") + " / "  
    + row.string("os") + " / " + row.string("payment")));
```

```
same rows: true  
Firefox / macOS / card
```

Safari / iOS / card
Chrome / Android / banktransfer

Why a seed is more than convenience

A seeded sample is a *function* of its input. An unseeded one reads system state, and the engine treats it as volatile — which changes what the planner is allowed to do with it.

When one sub-expression is read in two places, the planner evaluates it once and shares the result. A symmetric difference reads both its inputs twice, so writing the same seeded sample on both sides gives the planner four readers of one relation:

```
System.out.println(relix.relation("""
    (SAMPLE 5 ROWS SEED 7 (Feasible)) Δ (SAMPLE 5 ROWS SEED 7 (Feasible))
    """).explain());
```

```
SetOp UNION ~10 rows
├─ SetOp DIFFERENCE ~? rows
│   ├── Spool #1 ~5 rows
│   │   └─ SAMPLE 5 ROWS SEED 7 ~5 rows
│   │       └─ Select ~396 rows
│   │           └─ Join PRODUCT/NESTED_LOOP build=RIGHT ~1200 rows
│   │               ├── Join PRODUCT/NESTED_LOOP build=RIGHT ~400 rows
│   │               │   ├── Join PRODUCT/NESTED_LOOP build=RIGHT ~100 rows
│   │               │   │   ├── Join PRODUCT/NESTED_LOOP build=LEFT ~20 rows
│   │               │   │   │   ├── Scan Browsers ~4 rows
│   │               │   │   │   └─ Scan Systems ~5 rows
│   │               │   └─ Scan Locales ~5 rows
│   │               └─ Scan Payments ~4 rows
│   └─ Scan Networks ~3 rows
└─ Spool #1 (shared) ~5 rows
├─ SetOp DIFFERENCE ~? rows
│   ├── Spool #1 (shared) ~5 rows
│   └─ Spool #1 (shared) ~5 rows
```

Spool #1 sits **on** the sample: the draw happens once and all four readers see the same five rows. Take the seed away and the two spellings become two different draws — that is what the query now says, and sharing one evaluation between them would answer a different question:

```
System.out.println(relix.relation("""
    (SAMPLE 5 ROWS (Feasible)) Δ (SAMPLE 5 ROWS (Feasible))
    """).explain());
```

```
SetOp UNION ~10 rows
├─ SetOp DIFFERENCE ~? rows
│   ├── SAMPLE 5 ROWS ~5 rows
│   │   └─ Spool #1 ~396 rows
│   │       └─ Select ~396 rows
│   │           └─ Join PRODUCT/NESTED_LOOP build=RIGHT ~1200 rows
│   │               ├── Join PRODUCT/NESTED_LOOP build=RIGHT ~400 rows
│   │               │   ├── Join PRODUCT/NESTED_LOOP build=RIGHT ~100 rows
│   │               │   │   ├── Join PRODUCT/NESTED_LOOP build=LEFT ~20 rows
│   │               │   │   │   ├── Scan Browsers ~4 rows
│   │               │   │   │   └─ Scan Systems ~5 rows
│   │               │   └─ Scan Locales ~5 rows
│   │               └─ Scan Payments ~4 rows
│   └─ Scan Networks ~3 rows
└─ SAMPLE 5 ROWS ~5 rows
    └─ Spool #1 (shared) ~396 rows
├─ SetOp DIFFERENCE ~? rows
```

```

├─ SAMPLE 5 ROWS ~5 rows
├─   └─ Spool #1 (shared) ~396 rows
└─ SAMPLE 5 ROWS ~5 rows
    └─ Spool #1 (shared) ~396 rows

```

The spool has moved **below** the sample. The candidate set — the product and the selection over it, by far the expensive part — is still computed once and shared; each `SAMPLE` then draws from it independently. Sharing stops exactly where volatility begins, and not a level higher.

Neither plan is a compromise; each is the correct reading of what was written. It is worth knowing which one you wrote, because a repeated unseeded sample inside a larger query is paid for every time it appears.

Neither sampling operator is folded into a backend query — the rows are drawn in the engine, over rows the engine has read.

Against a database

A session reads a database through a connection. There are two ways to give it one: let the script declare a JDBC URL, or hand the session a `DataSource` the program already has — a pooled one from the application's own configuration, most usefully.

This page uses an in-memory H2 database so that it runs with nothing installed. Everything it shows works the same against any JDBC backend; only the URL changes.

```

import org.h2.jdbcx.JdbcDataSource;
import java.sql.Connection;

JdbcDataSource warehouse = new JdbcDataSource();
warehouse.setURL("jdbc:h2:mem:guide;DB_CLOSE_DELAY=-1");

try (Connection setup = warehouse.getConnection()) {
    setup.createStatement().execute("""
        DROP TABLE IF EXISTS orders;
        CREATE TABLE orders (
            order_id INT, customer_id INT, status VARCHAR(16), amount DECIMAL(10,2));
        INSERT INTO orders VALUES
            (1, 10, 'OPEN', 100), (2, 11, 'SHIPPED', 250), (3, 10, 'OPEN', 75);
        """);
}

```

Binding a handle

`jdbc(name, dataSource)` binds the handle under a connection name. The session declares the connection on your behalf, so nothing in the script has to name a URL — and nothing in the script *can* leak one:

```

Relix relix = Relix.builder().jdbc("warehouse", warehouse).build();

System.out.println(relix.definitions());

connection warehouse from jdbc { };

```

That is the session rendered back as `.relix` text, and it is worth looking at: the connection is there, and the URL is not, because there isn't one — the handle is held by the session rather than described by the script.

Reading a table

A **dotted reference** names a table in a connection and asks the database what its columns are:

```
Relation orders = relix.relation("warehouse.orders");

System.out.println(orders.render());
orders.schema().columns().forEach(column ->
    System.out.println("  " + column.name() + " : " + column.type().display()));
```

```
warehouse.orders
  ORDER_ID : number
  CUSTOMER_ID : number
  STATUS : string
  AMOUNT : number
```

Nothing declared those types. They come from the database's own catalog, read through the handle when the name was resolved — which is why a dotted reference is the form to use when the database is the authority on the shape of the data.

The case of those names comes from the database too. H2, like most SQL databases, folds an unquoted identifier to a canonical case, so a table created as `order_id` reports itself as `ORDER_ID` — and Relix repeats what it was told rather than normalising it, because a column name it invented would not be the one a hand-written query would use. Quote the identifiers in your DDL if you would rather keep them lowercase.

The rows come back through the terminals like any others:

```
orders.toList().forEach(System.out::println);

(ORDER_ID=1, CUSTOMER_ID=10, STATUS=OPEN, AMOUNT=100)
(ORDER_ID=2, CUSTOMER_ID=11, STATUS=SHIPPED, AMOUNT=250)
(ORDER_ID=3, CUSTOMER_ID=10, STATUS=OPEN, AMOUNT=75)
```

The alternative is to declare a source, which states the schema in the script and asks the database nothing:

```
relix.define("""
    source OpenOrders from warehouse { table: "orders",
        schema: { ORDER_ID: NUMBER, STATUS: STRING, AMOUNT: NUMBER } };
    """);

System.out.println(relix.relation("π ORDER_ID, AMOUNT (σ STATUS = 'OPEN' (OpenOrders))")
    .toList());

[(ORDER_ID=1, AMOUNT=100), (ORDER_ID=3, AMOUNT=75)]
```

Declaring it is what makes a query analyse with the database unreachable, and a schema written down is a schema that can go stale. Which to prefer is the subject of [Working without a database](#).

What the database does, and what the engine does

The interesting part of embedding a query engine over a database is the division of labour, and `explain()` is where it is visible:

```
System.out.println(relix.relation(
    "γ CUSTOMER_ID, SUM(AMOUNT) → total (σ STATUS = 'OPEN' (warehouse.orders))")
    .explain());

PushedScan [jdbc/warehouse] SELECT CUSTOMER_ID, SUM(AMOUNT) FROM orders WHERE (STATUS = 'OPEN') GROUP BY CUSTOMER_ID -2 rows
```

The whole query became one `SELECT`. The filter, the grouping and the aggregate were folded into it, so the database does the work and the aggregated rows are what cross the wire — which is the difference between embedding an engine and fetching a table into one.

Not everything folds, and the plan says so plainly. A function the backend cannot spell leaves the work in the engine:

```
System.out.println(relix.relation(
    "σ r > 0.5 (π ORDER_ID, Rand() → r (warehouse.orders))").explain());

Select ~1 rows
└ Project ~3 rows
  └ PushedScan [jdbc/warehouse] SELECT ORDER_ID, CUSTOMER_ID, STATUS, AMOUNT FROM orders ~3 rows
```

Read the plan as a boundary: everything below the pushed scan happened in the database, everything above it happened here. Nothing is lost when a fold fails — the answer is the same, and the plan tells you what it cost.

The headline case: your data, joined to theirs

Rows a Java program is holding go in with `table` and `join` against a database table like anything else. This is the thing an embedded engine can do that a database client cannot:

```
relix.table("Regions", List.of("CUSTOMER_ID", "region"), List.of(
    Map.of("CUSTOMER_ID", 10, "region", "North"),
    Map.of("CUSTOMER_ID", 11, "region", "South")));

Relation enriched = relix.relation("warehouse.orders")
    .join(relix.relation("Regions"),
        eq(attr("orders.CUSTOMER_ID"), attr("Regions.CUSTOMER_ID")))
    .project("ORDER_ID", "region", "AMOUNT");

enriched.toList().forEach(System.out::println);

(ORDER_ID=1, region=North, AMOUNT=100)
(ORDER_ID=2, region=South, AMOUNT=250)
(ORDER_ID=3, region=North, AMOUNT=75)
```

The join is evaluated here, because half its input is in this program's heap. What is pushed is the part that can be: the scan of `orders`.

Connections, and who closes what

A session opens database connections lazily, pools them, and releases them when it closes. The `DataSource` you handed it is yours and is left alone:

```
relix.close();

try {
    relix.relation("warehouse.orders").toList();
} catch (RelixException e) {
    System.out.println(e.getMessage());
}

try (Connection stillMine = warehouse.getConnection()) {
    System.out.println("the DataSource is untouched: " + !stillMine.isClosed());
}

this session is closed
the DataSource is untouched: true
```

Between those two, the rule is: close the session, keep the handle. A session is meant to be long-lived — a field closed at shutdown — because the declarations it accumulates and the connections it pools are both worth keeping.

One thing does need care in a long-lived session. `toList()` drains and closes; `stream()` hands you a live cursor, and a stream you do not close holds a pooled connection until the session ends. Use it in a try-with-resources, every time.

More than one backend at once

A session can hold several connections, of different kinds, and a query may span them. The engine pushes what each backend can do and joins the results here:

```
JdbcDataSource archive = new JdbcDataSource();
archive.setURL("jdbc:h2:mem:guide_archive;DB_CLOSE_DELAY=-1");

try (Connection setup = archive.getConnection()) {
    setup.createStatement().execute("""
        DROP TABLE IF EXISTS closed;
        CREATE TABLE closed (order_id INT, closed_on VARCHAR(10));
        INSERT INTO closed VALUES (2, '2026-01-04');
        """);
}

try (Relix federated = Relix.builder()
    .jdbc("warehouse", warehouse)
    .jdbc("archive", archive)
    .build()) {

    Relation matched = federated.relation("p 0 (warehouse.orders)")
        .join(federated.relation("p C (archive.closed)"),
            eq(attr("0.ORDER_ID"), attr("C.ORDER_ID")));

    Relation written = federated.relation("""
        (p 0 (warehouse.orders)) ⋈ 0.ORDER_ID = C.ORDER_ID (p C (archive.closed))
        """);

    System.out.println(matched.project("ORDER_ID", "STATUS", "CLOSED_ON").toList());
    System.out.println(written.project("ORDER_ID", "STATUS", "CLOSED_ON").toList());
    System.out.println(matched.explain());
}

[(ORDER_ID=2, STATUS=SHIPPED, CLOSED_ON=2026-01-04)]
[(ORDER_ID=2, STATUS=SHIPPED, CLOSED_ON=2026-01-04)]
Join INNER/HASH build=RIGHT ~3 rows
├─ Rename ~3 rows
│   └─ PushedScan [jdbc/warehouse] SELECT ORDER_ID, CUSTOMER_ID, STATUS, AMOUNT FROM orders ~3 rows
└─ Rename ~1 rows
    └─ PushedScan [jdbc/archive] SELECT ORDER_ID, CLOSED_ON FROM closed ~1 rows
```

Each side is a separate database that knows nothing of the other. The plan shows the division: two pushed scans, one per connection, and a join the engine performs over what they return.

Both forms are shown because both work, and the second line of output is there to say so. Each side is a relation in its own right, built and pinned separately — and each introspected its own table as it was built, so the two know different names. Composing them resolves the result against what *both* know, which is why a combinator spans two backends exactly as a single expression does.

One detail makes either form read the way it does: both sides are renamed with `p`, because the qualifier of a dotted reference is the whole dotted name — `warehouse.orders`, not `orders` — and a short alias is easier to write a join condition against.

What a federated query does not promise

Two databases, a file and an API are four independent systems, and one query across them is four independent reads. Nothing coordinates them, and three consequences follow that are worth knowing before such a query carries any weight.

Nothing is contacted until rows are asked for

A session can be built, a source declared and a plan produced with the backend entirely unreachable. Planning asks the catalog and the cost model and reads no rows — and where the script declares the schema, as here, it asks the database nothing at all:

```
import com.darkcollective.relix.embed.QueryExecutionException;

JdbcDataSource offline = new JdbcDataSource();
offline.setURL("jdbc:h2:tcp://127.0.0.1:1/unreachable");

try (Relix down = Relix.builder().jdbc("remote", offline).build()) {
    down.define("""
        source Remote from remote { table: "orders", schema: { id: NUMBER } };
        """);

    System.out.println(down.relation("Remote").explain().strip());

    try {
        down.relation("Remote").toList();
    } catch (QueryExecutionException e) {
        System.out.println(e.getMessage().split(":")[0]);
    }
}
```

```
PushedScan [jdbc/remote] SELECT id FROM orders
JDBC error running pushed-down query on connection 'remote'
```

The plan is not a guess: it is the statement that database would have been sent. What fails is the terminal, and the message names the connection the failure came from — every source does this, a JDBC one naming its connection and table, an HTTP or file source naming the relation. The text before the colon is the engine's; what follows it is the driver's, and varies with the driver.

The type says which kind of wrong this is. A `QueryExecutionException` means the query was runnable and something outside it gave way, which is a fact about the world and may be worth retrying; a plain `RelixException` means the query was never runnable, and retrying a malformed query changes nothing. Both are unchecked, and both are `RelixException`, so a caller who wants neither distinction catches that and is covered.

A collecting terminal has a third answer: `UnboundedRelationException`, for a relation that never ends. Nothing failed there — it is the shape of the question that a list cannot answer, so the remedy is a bound or `stream`.

One input is read to its end before the next is opened

A join has to have one side in hand before it can match the other against it. So the two reads do not overlap — the engine drains one input completely, then opens the second:

```

import com.darkcollective.relix.processor.Row;
import com.darkcollective.relix.symbol.ColumnDefinition;
import com.darkcollective.relix.symbol.ScalarType;
import com.darkcollective.relix.value.NumberValue;
import java.util.ArrayList;
import java.util.stream.IntStream;

List<String> reads = new ArrayList<>();

Schema left = new Schema(List.of(new ColumnDefinition("id", ScalarType.NUMBER)));
Schema right = new Schema(List.of(new ColumnDefinition("rid", ScalarType.NUMBER)));

try (Relix timed = Relix.open()) {
    timed.source("Left", left, () -> {
        reads.add("opened Left");
        return IntStream.rangeClosed(1, 2).mapToObj(i -> {
            reads.add(" read Left " + i);
            return Row.of(left, NumberValue.of(String.valueOf(i)));
        });
    });
    timed.source("Right", right, () -> {
        reads.add("opened Right");
        return IntStream.rangeClosed(1, 2).mapToObj(i -> {
            reads.add(" read Right " + i);
            return Row.of(right, NumberValue.of(String.valueOf(i)));
        });
    });

    timed.relation("Left >= Left.id = Right.rid Right").toList();
}

reads.forEach(System.out::println);

opened Right
 read Right 1
 read Right 2
opened Left
 read Left 1
 read Left 2

```

Which side goes first is the planner's build-side choice and can change with the statistics; that one of them is finished before the other starts cannot. A natural join buffers both sides rather than streaming the second, which changes how much is held, not the ordering.

The consequence is the one to carry away: **a federated result is not a snapshot**. Each source is current as of the moment *it* was read, and those moments are different — far apart, if the first read is large or the second backend is slow. Where that matters, the answer is a column the data already carries — an as-of timestamp to join on, or a version to filter by — rather than anything the engine can arrange across systems it does not control.

What the caller has already seen depends on the terminal

A failure aborts the query. Whether the application has already acted on part of the answer when that happens is decided by which terminal it used, because `toList` finishes before it returns anything and `stream` hands rows over as they are produced:

```

import java.util.function.Supplier;

Schema flaky = new Schema(List.of(new ColumnDefinition("id", ScalarType.NUMBER)));

```

```
Supplier<Stream<Row>> failsAfterTwo = () -> Stream.concat(
    IntStream.rangeClosed(1, 2).mapToObj(i ->
        Row.of(flaky, NumberValue.of(String.valueOf(i)))),
    Stream.generate(() -> { throw new IllegalStateException("connection reset"); }));

try (Relix partial = Relix.open()) {
    partial.source("Flaky", flaky, failsAfterTwo);

    try {
        List<Tuple> all = partial.relation("Flaky").toList();
        System.out.println("toList: caller saw " + all.size() + " rows");
    } catch (RuntimeException e) {
        System.out.println("toList: caller saw nothing");
    }

    List<String> streamed = new ArrayList<>();
    try (Stream<Tuple> rows = partial.relation("Flaky").stream()) {
        rows.forEach(row -> streamed.add(row.decimal("id").toString()));
    } catch (RuntimeException e) {
        System.out.println("stream: caller saw " + streamed);
    }
}
```

```
toList: caller saw nothing
stream: caller saw [1, 2]
```

Neither is the safe one in general. `toList` is all-or-nothing and needs the whole result in memory; `stream` is bounded but can hand over a prefix of an answer that never completes, so an application doing something irreversible per row — posting a payment, sending a message — is doing it to rows that may turn out to be part of a failed query. Collect first, or make the per-row work replayable.

Secrets in declarations

A source that calls an API needs a token, and a connection to a database needs a password. Where the value comes from is the program's business: an environment variable, a vault, a key-management service, a configuration file. What the session needs is a way to be handed it without the value becoming part of the script.

A placeholder, resolved when a query runs

Write `${NAME}` wherever a declaration holds the value, and give the session a function that answers for each name:

```
import java.util.Optional;

Map<String, String> vault = Map.of("ORDERS_TOKEN", "tok-4f9a");

Relix secured = Relix.builder()
    .placeholders(name -> Optional.ofNullable(vault.get(name)))
    .build();

secured.define("""
    source Orders from http {
        url:      "https://api.example.com/orders",
        headers: { "Authorization": "Bearer ${ORDERS_TOKEN}" },
        schema: { id: NUMBER, status: STRING }
    };
});
```

```
""");
```

```
System.out.println(secured.definitions());
```

```
source Orders from http { url: "https://api.example.com/orders", method: GET, headers: { "Authorization": "Bearer ${ORDERS_TOKEN}" }, schema: { id: number, status: string } };
```

The placeholder stays in the declaration. It is resolved each time a query runs, for the declarations that query reaches, and the value goes to the connector for that run and nowhere else. Three things follow from that:

- **Nothing to escape.** The value never passes through the parser, so a token holding a quote or a backslash is sent exactly as the resolver returned it.
- **Nothing to leak.** `definitions()`, `ir()` and an analysis error print the session's declarations, and those hold `${ORDERS_TOKEN}`, not the token.
- **Rotation works.** The resolver is asked again on the next run, so a credential that changed is picked up without redefining anything.

For environment variables the resolver is one line: `name ->`

```
Optional.ofNullable(System.getenv(name)).
```

What is resolved is the string **values** of a declaration: URLs, file paths, table names, header values, request bodies, the credentials in `auth:`, column defaults and a connection's properties. Names (of a header, a column, a property) are not. Within one run each name is asked for once. Analysis asks too, when it introspects a connection's tables.

A session built without `placeholders(...)` leaves `${...}` exactly as written.

A name with no value

When the resolver returns empty for a name a query needs, the query fails before anything is sent, and the error names both the placeholder and the declaration holding it:

```
secured.define("""
    source Invoices from http {
        url:      "https://api.example.com/invoices",
        auth:     bearer("${INVOICES_TOKEN}"),
        schema:   { id: NUMBER }
    };
""");
```

```
try {
    secured.relation("Invoices").toList();
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

```
placeholder ${INVOICES_TOKEN} in source 'Invoices' has no value: the session's placeholder resolver returned none
```

Only the declarations a query reaches are resolved, so a missing value for one source does not stop a query over another. `Orders` above still runs.

Declarations built in Java

A program that already holds the value can build the declaration with `ScriptBuilders` and hand it over with `define(Statement...)`. The header map is a plain Java map, so nothing needs escaping:

```
import com.darkcollective.relix.lang.ast.source.HttpMethod;
import static com.darkcollective.relix.lang.ast.ScriptBuilders.httpSource;
```

```
import static com.darkcollective.relix.lang.ast.ScriptBuilders.source;

Relix typed = Relix.open();
typed.define(source("Customers",
    httpSource("https://api.example.com/customers", HttpMethod.GET,
        Map.of("X-API-Key", "key-\"quoted\"",
            Optional.empty(), Optional.empty(),
            List.of(), Optional.empty(), Optional.empty())));

System.out.println(typed.definitions());

private source Customers from http { url: "https://api.example.com/customers", method: GET, headers: { "X-API-Key": "key-\"quoted\"" } };
```

The value is now part of the session's declarations, and `definitions()` shows it (private because `source(name, config)` builds an unexported declaration). That is the difference between the two routes: prefer a placeholder for a secret, and `define(Statement...)` for a value that is not one.

Sandboxing a session

A session is open by default: it accepts every declaration the language has, including a `source` that reads any file the process can read and a `connection` to any database it can reach. That is right when you wrote the script. It is not right when the script comes from someone else, such as a learner in a tutorial or a language model answering a question, and you want them to explore a fixed set of data and nothing more.

A closed `Sandbox` handles that case. User text can still declare inline tables, views and functions. It can only reach outside the session through the sources and connections the sandbox itself provides, and every query runs under the sandbox's limits.

A closed sandbox

The sandbox's declarations are the data it offers. Here that is one CSV file, and the sandbox also caps a result at two rows:

```
import com.darkcollective.relix.embed.Sandbox;
import java.nio.file.Files;
import java.nio.file.Path;

Path data = Files.createTempDirectory("sandbox");
Files.writeString(data.resolve("orders.csv"), """
    id,customer,amount
    1,Ada,120
    2,Bo,80
    3,Cy,200
    """);

Sandbox sandbox = Sandbox.builder()
    .baseDirectory(data)
    .declarations("""
        source Orders from csv("orders.csv") {
            schema: { id: NUMBER, customer: STRING, amount: NUMBER }
        };
        """)
    .maxOutputRows(2)
    .build();

Relix learner = Relix.builder().sandbox(sandbox).build();
```

```
learner.define("Large := { σ amount > 100 (Orders) }");
System.out.println(learner.relation("Large").toList());
```

```
[(id=1, customer=Ada, amount=120), (id=3, customer=Cy, amount=200)]
```

Orders was declared by the sandbox, so it is there to query. The view `Large` is the user's own, and a view reads nothing outside the session, so it is accepted.

What is refused

A declaration that reaches outside the session is refused unless the sandbox declares the same one. That covers a `source` over a file, a database or an HTTP endpoint, a `connection`, an `import` and an `env` statement. The refusal is a `SandboxViolationException`, and nothing from the refused call is installed:

```
import com.darkcollective.relix.embed.SandboxViolationException;

try {
    learner.define("""
        source Secrets from csv("/etc/passwd") { schema: { line: STRING } };
        """);
} catch (SandboxViolationException e) {
    System.out.println(e.getMessage());
}
```

```
this session's sandbox does not permit the source 'Secrets'; it accepts inline tables, views, functions and the sources and connections it was configured with
```

A user may repeat one of the sandbox's own declarations, for instance because their script is a complete file that declares what it reads. The comparison is on the declaration as printed, so layout and comments do not matter, but every value must match:

```
learner.define("""
    -- the same source, laid out differently
    source Orders from csv("orders.csv") { schema: { id: NUMBER,
                                                    customer: STRING,
                                                    amount: NUMBER } };
    """);
System.out.println(learner.relation("Orders").count());
```

```
3
```

A generator source reads nothing, so it is accepted like an inline table.

`validate(...)` reports a refusal as a diagnostic instead of throwing, which suits a tool that shows problems to its user:

```
System.out.println(learner.validate("connection wh from jdbc { url: \"jdbc:h2:mem:x\" };"));
```

```
[error: this session's sandbox does not permit the connection 'wh'; it accepts inline tables, views, functions and the sources and connections it was configured with]
```

The rules apply to text and statements given to `define`, `script`, `relation` and `validate`. The Java registration methods (`table`, `source`, `connector`, and the builder's `jdbc` bindings) are the host program's own, and are not restricted.

Limits

The output limit cuts a longer result at that many rows. `run()` says when it did:

```
Rows all = learner.relation("Orders").run();
System.out.println(all.size() + " rows, truncated: " + all.truncated());
```

```
2 rows, truncated: true
```

`toList()` and `stream()` are cut the same way, but they carry no events, so a host that must tell its user about the cut uses `run()`, or `stream(listener)` and watches for the `EXECUTE/TRUNCATED` event. `count()` returns a single row, so it counts the whole relation.

Limit	What it bounds
<code>maxInputChars</code>	The length of the text one call may pass. Longer text is refused before it is parsed.
<code>maxOutputRows</code>	The rows one query returns.
<code>maxMaterializedRows</code>	The rows one blocking operator may buffer.
<code>maxFixpointRounds</code>	The rounds one recursion may run.
<code>maxProcessedRows</code>	The work one execution may do, counted as rows passed from one operator to the next.
<code>timeoutMillis</code>	How long one execution may run.

The last four are also the builder's own caps. When both the builder and the sandbox set one, the smaller wins.

The output limit cannot stop a query that works for a long time and produces little. A selection over an endless generator that matches nothing never yields a row, so it never reaches the output limit. The last two limits are for that case:

```
Relix bounded = Relix.builder()
    .sandbox(Sandbox.builder().maxProcessedRows(10_000).build())
    .build();
bounded.define("source Naturals from generator { name: \"Naturals\" };");

try (Stream<Tuple> rows = bounded.relation("σ Len(CStr(n)) = 0 (Naturals)").stream()) {
    rows.toList();
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

query stopped: it processed more than 10000 rows, the limit for one execution

`maxProcessedRows` counts every row an operator hands to the one above it, so the same query over the same data stops at the same point on any machine. The timeout is checked at the same points, so it catches anything that moves rows, but it depends on the machine. It cannot interrupt a call that never returns to the engine, such as a JDBC driver waiting on its database. Prefer `maxProcessedRows` for a reproducible limit and use the timeout as a backstop.

A configuration file

A sandbox is usually configured rather than coded, so the data on offer can change without a rebuild. `Sandbox.load` reads a JSON file:

```
Files.writeString(data.resolve("sandbox.relix"), ""
    source Orders from csv("orders.csv") {
        schema: { id: NUMBER, customer: STRING, amount: NUMBER }
    };
    "");
Files.writeString(data.resolve("sandbox.json"), ""
    {
        "declarations": "sandbox.relix",
```

```

        "limits": { "maxInputChars": 20000, "maxOutputRows": 1000,
                    "maxProcessedRows": 10000000, "timeoutMillis": 30000 }
    }
    """);

```

```

Sandbox configured = Sandbox.load(data.resolve("sandbox.json"));
System.out.println(configured);

```

```

Sandbox[closed, maxInputChars=20000, maxOutputRows=1000, maxMaterializedRows=unlimited, maxFixpointRounds=unlimited, maxProcessedRows=10000000, timeout=PT30S]

```

declarations names a `.relix` file relative to the configuration file, and relative paths inside it resolve against the configuration file's directory. Every key is optional, and an unknown key is refused, so a misspelt limit fails instead of silently not applying. The declarations may use `${NAME}` placeholders, which keeps credentials out of the file (see [Secrets in declarations](#)).

A session built without a sandbox, or with `Sandbox.open()`, enforces nothing.

Working without a database

A relation is a value, and building one runs nothing. So composing a query, rendering it, asking for its schema and optimising it all work with no database in reach — which is what makes "read a query, improve it, hand back better text" a whole use of the engine.

That much is free wherever a source **declares its own schema**. This page is about the two places where it is not free: a name only a database can resolve, and a plan only statistics can choose well.

A source that declares itself needs nothing

```

import com.darkcollective.relax.lang.ast.ConnectionDeclaration;
import com.darkcollective.relax.semantic.CatalogProvider;
import com.darkcollective.relax.semantic.CatalogSnapshot;
import com.darkcollective.relax.symbol.ColumnDefinition;
import com.darkcollective.relax.symbol.RelationStatistics;
import com.darkcollective.relax.symbol.ScalarType;
import java.time.Clock;
import java.time.Instant;
import java.time.ZoneOffset;
import java.util.Optional;

Relix relax = Relix.open();

relax.define("""
    connection warehouse from database { url: "jdbc:postgresql://nowhere/prod" };
    source Orders from warehouse { table: "orders",
        schema: { id: NUMBER, customer: STRING, amount: NUMBER } };
""");

```

```

System.out.println(relax.relation("σ amount > 100 (Orders)").optimized().render());

```

```

σ amount > 100 (Orders)

```

Nothing was opened. The connection is declared, the source names a table in it, and the schema is written down — so the analyser has everything it needs, and the URL is not read until something asks for rows.

A name only the database can resolve

A dotted reference is the other case. `warehouse.orders` has no declared schema by design: its columns come from the database's catalog, and asking is the point of writing it that way.

Offline, the analyser reports it as an unresolved name — and a session **rejects that by default**, because the overwhelmingly common cause is a typo, and a relation built over a name that resolves to nothing is a failure deferred to a worse moment:

```
try {
    relix.relation("warehouse.customers");
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

```
Cannot resolve schema for table 'customers' in connection 'warehouse': no schema available (declare it with a 'source ... from warehouse'
↳ binding, or provide a catalog); Undefined relation: 'warehouse.customers'
```

`Relix.builder().allowUnresolved()` takes the other reading, for the case where composing and rendering against an unreachable database is the actual goal.

A snapshot, so offline is full strength

The message names the real fix: *provide a catalog*. Metadata genuinely does come from the database, so "works offline" and "works well" pull apart unless it can be supplied some other way — and a `CatalogProvider` is where it is supplied.

`captureCatalog()` asks the live database about every connection-backed table the session names, once, and hands back something you can write to a file:

```
// Standing in for the live database, so this page can run with nothing reachable.
// In a program this is the session's own catalog and you do not write it.
class Production implements CatalogProvider {
    public Optional<Schema> tableSchema(ConnectionDeclaration connection, String table) {
        return Optional.of(new Schema(List.of(
            new ColumnDefinition("id", ScalarType.NUMBER),
            new ColumnDefinition("email", ScalarType.STRING)));
    }

    public Optional<RelationStatistics> tableStatistics(
        ConnectionDeclaration connection, String table) {
        return Optional.of(RelationStatistics.of(4_000_000));
    }
}

// The capture time comes from the session's clock, which is pinned here so this page
// prints the same document every time it runs.
Relix live = Relix.builder()
    .catalog(new Production())
    .clock(Clock.fixed(Instant.parse("2026-08-31T09:00:00Z"), ZoneOffset.UTC))
    .build();
live.define("""
    connection warehouse from database { url: "jdbc:postgresql://nowhere/prod" };
    source Customers from warehouse { table: "customers", schema: { id: NUMBER } };
    """);

String captured = live.captureCatalog().toJson();
live.close();

System.out.println(captured);
```

```
{
  "version": 1,
  "captured": "2026-08-31T09:00:00Z",
  "tables": [
    {
      "connection": "warehouse",
      "table": "customers",
      "origin": "introspected",
      "captured": "2026-08-31T09:00:00Z",
      "schema": [
        {
          "name": "id",
          "type": {
            "scalar": "NUMBER"
          }
        },
        {
          "name": "email",
          "type": {
            "scalar": "STRING"
          }
        }
      ],
      "statistics": {
        "rowCount": 4000000
      }
    }
  ]
}
```

Read it back and hand it to a session, and the dotted reference resolves with nothing reachable — and the cost model sees four million rows, so it makes the same decisions the live session would:

```
try (Relix offline = Relix.builder()
    .catalog(CatalogSnapshot.parse(captured))
    .build()) {

    offline.define("""
        connection warehouse from database { url: "jdbc:postgresql://nowhere/prod" };
        """);

    Relation customers = offline.relation("warehouse.customers");
    var planned = customers.plan();

    System.out.println(customers.schema().columns().size() + " columns, ~"
        + planned.estimate().rows(planned.plan()).orElseThrow() + " rows");
}
```

2 columns, ~4000000 rows

That is the practical shape of it: **tune or validate a production query from a machine with no access to production**, and do it in CI.

A snapshot also knows which tables it recorded, so a misspelt one is met with a suggestion rather than only a refusal:

```
try (Relix offline = Relix.builder()
    .catalog(CatalogSnapshot.parse(captured))
    .build()) {

    offline.define("""
        connection warehouse from database { url: "jdbc:postgresql://nowhere/prod" };
        """);

    for (Diagnostic problem : offline.validate("query { warehouse.custmers };")) {
        System.out.println(problem.message());
    }
}
```

```
Cannot resolve schema for table 'custmers' in connection 'warehouse': no schema available (declare it with a 'source ... from warehouse'
binding, or provide a catalog) - did you mean 'warehouse.customers'?
Undefined relation: 'warehouse.custmers'
```

What a run adds

A snapshot carries what the database *said*. A run carries what was actually there, and the two are the same artifact — a row count, differing only in where it came from.

So every terminal that drains a relation measures it. A scan read to the end reports the relation's real size, and the next plan prefers that to whatever it would have estimated:

```
import com.darkcollective.relix.processor.Row;
import com.darkcollective.relix.value.NumberValue;
import java.util.stream.Stream;

Schema readings = new Schema(List.of(new ColumnDefinition("id", ScalarType.NUMBER)));

Relix measured = Relix.open();
measured.source("Readings", readings, () -> Stream.of(
```

```

        Row.of(readings, NumberValue.of("1")),
        Row.of(readings, NumberValue.of("2")),
        Row.of(readings, NumberValue.of("3"))));

var before = measured.relation("Readings").plan();
System.out.println("estimated: " + before.estimated().rows(before.plan()));

measured.relation("Readings").toList();

var after = measured.relation("Readings").plan();
System.out.println("measured: " + after.estimated().rows(after.plan()));
measured.close();

estimated: OptionalLong.empty
measured: OptionalLong[3]

```

The condition is exactly "read to the end". A caller who takes three rows and closes the stream has measured their own patience, not the relation, so a partial read reports nothing — and `stream()`, which hands the caller the lifecycle, usually is a partial read.

A whole expression is measured the same way, keyed by the expression rather than the relation, because a post-selection count is not a fact about any relation. Two limits come with that and are worth knowing. It **memoises rather than learns**: measuring `σ status = 'OPEN'` says nothing about `σ status = 'CLOSED'`, since they are different expressions. And a query parameterised by a literal mints an entry per literal, so the map is bounded and evicts — normalising the literal away would throw out the precision that made the measurement worth keeping.

An expression whose count drifts by construction is not remembered at all. A relation filtered against `NOW()`, or an unseeded `SAMPLE`, produces a number that was true once.

Measurements of connection-backed tables travel: `captureCatalog()` writes them into the snapshot as observed entries, which is the loop closing — a run produces the number, and the snapshot serves it to the next session that cannot reach the database.

What a snapshot does and does not promise

Every entry records where it came from and when, and a lookup prefers a measured observation to an introspected estimate, and a recent one to an old one. Entries are never merged into each other, so a snapshot stays a log of what was seen and when rather than a picture assembled from several moments.

Statistics are advisory throughout the engine, and a snapshot changes nothing about that: a stale row count costs a worse plan, never a wrong answer. A stale **schema** is different — a column that has since been dropped resolves here and fails when the query runs — which is what the capture time is there to let you check.

4. Getting answers out

Page	What it covers
Reading results	The terminals and who closes what; typed row accessors; nested values; streaming; running the tree as written; what a drained read teaches the engine
Looking at what the engine did	Rendering, the rewrite and why it fired, the physical plan and its estimates, the session's own report
The engine describes itself	The <code>relix.*</code> catalogs as relations: what is installed, what the session knows, dependencies, functions, and the last run's feed
Provenance: where a row came from	Lineage as a column; annotating a relation over a semiring; which to reach for

Reading results

Four terminals move rows out of the engine, and a fifth counts them without moving any. They differ in one thing: who owns the lifecycle of the result.

Terminal	Gives back	Who closes it
<code>count()</code>	a <code>long</code>	nothing to close
<code>toList()</code>	<code>List<Tuple></code>	the terminal, before it returns
<code>run()</code>	Rows — the rows and the run's events	the terminal
<code>stream()</code>	<code>Stream<Tuple></code> , lazily	you
<code>stream(listener)</code>	the same, with the run reported as it happens	you

`toList` and `stream` each have a typed form that reads rows straight into a record of yours — see [Rows as your own types](#).

```
Relix relix = Relix.open();

relix.table("Orders", List.of("order_id", "customer", "status", "amount"), List.of(
    Map.of("order_id", 1, "customer", "Ada", "status", "OPEN", "amount", 100),
    Map.of("order_id", 2, "customer", "Grace", "status", "SHIPPED", "amount", 250),
    Map.of("order_id", 3, "customer", "Ada", "status", "OPEN", "amount", 75)));
```

The one to reach for

`toList()` drains the result and closes it. That is the whole reason it is the default: a result stream may hold a live database cursor, and a terminal that closes it cannot leak one.

```
Relation open = relix.relation("σ status = 'OPEN' (Orders)");

List<Tuple> rows = open.toList();
System.out.println(rows.size() + " rows");
System.out.println(rows.getFirst());

2 rows
(order_id=1, customer=Ada, status=OPEN, amount=100)
```

`count()` is not a special counting mode. It is `γ COUNT(*)` over the relation — an ordinary operator, which is why it pushes down like any other and a database counts its own rows:

```
System.out.println(open.count());
```

```
2
```

Reading a row

A `Tuple` is the engine's row plus the conversions to Java types. Each accessor names a type and returns that type, or `null` when the value is `NULL` — which is why none of them returns a primitive:

```
Tuple first = relix.relation("Orders").toList().getFirst();
```

```
System.out.println(first.string("customer"));
System.out.println(first.longValue("order_id"));
System.out.println(first.decimal("amount"));
System.out.println(first.columnNames());
System.out.println(first.isNull("status"));
```

```
Ada
1
100
[order_id, customer, status, amount]
false
```

An accessor that meets a value of another type **refuses** rather than coercing:

```
try {
    first.longValue("customer");
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

```
column 'customer' holds string, not number – read it with get(...) and match on the type, or render it with get(...).asDisplayString()
```

That is deliberate. A coercing accessor is a second, weaker statement of what a column holds, and its failure arrives as a wrong answer rather than as a wrong call. For the same reason `longValue` refuses a fractional number instead of truncating it.

For a column whose type genuinely varies — an `ANY` column over schema-on-read data — the accessors are the wrong tool. `Value` is a sealed hierarchy, so a `switch` over it is exhaustive and the compiler names the case you forgot:

```
import com.darkcollective.relix.value.NumberValue;
import com.darkcollective.relix.value.StringValue;

for (Tuple row : relix.relation("π customer, amount (Orders)").toList()) {
    for (String column : row.columnNames()) {
        String described = switch (row.get(column)) {
            case NumberValue n -> "number " + n.value();
            case StringValue s -> "string '" + s.value() + "'";
            default -> row.get(column).asDisplayString();
        };
        System.out.println(column + " → " + described);
    }
}
```

```
customer → string 'Ada'
amount → number 100
customer → string 'Grace'
```

```
amount → number 250
customer → string 'Ada'
amount → number 75
```

That is the authoritative route; the accessors are a convenience over the common case.

Rows as your own types

The lambda that turns a row into a record of yours is mechanical, is written once per query per program, and is exactly where a renamed column becomes a runtime failure. So `toList` and `stream` will write it: hand either one a record, and each component is read from the column of its own name, through the accessor its type names.

```
import java.math.BigDecimal;

record Order(long order_id, String customer, BigDecimal amount) { }

for (Order order : relix.relation("σ status = 'OPEN' (Orders)").toList(Order.class)) {
    System.out.println(order);
}
```

```
Order[order_id=1, customer=Ada, amount=100]
Order[order_id=3, customer=Ada, amount=75]
```

Nothing is coerced: the accessors are the same ones you would have called, so a component of the wrong type fails exactly as `longValue("customer")` does above. What is different is **when**. The heading is known without running anything, so the whole binding is checked at the call, before a single row moves:

```
record Wrong(long order_id, String custmer) { }

try {
    relix.relation("Orders").toList(Wrong.class);
} catch (RelixException e) {
    System.out.println(e.getMessage());
}
```

```
no column named 'custmer' for Wrong.custmer; the relation has [order_id, customer, status, amount] – rename it in the query (π old → cust
mer) or rename the component
```

Two things the heading cannot settle, and they are settled at the row instead: a column typed `ANY` has no declared type to check against, and whether a value is `NULL` is not a property of a heading. A `NULL` reaching a primitive component is refused there, naming the component and asking for the boxed type — so a column that can be `NULL` wants `Long`, not `long`.

Names must match, and the match is case-insensitive. There is no annotation for saying a component means some other column: the query already renames one, and `π amount → total` says it where a reader of the query will look for it. A component may be a `String`, a `BigDecimal`, `Long/long`, `Integer/int`, `Double/double`, `Boolean/boolean`, an `Instant`, `LocalDate`, `LocalTime` or `Duration`, a `List` for an array column, a `Map` for a struct column, or a `Value` — which is the one for a column whose type varies, and the only one that receives a `NULL` as a `NullValue` rather than as Java `null`.

The record is built through its canonical constructor, reflectively. A record declared in a named module therefore has to be reachable — public in an exported package, or its package opened to `com.darkcollective.relix.embed` — and says so if it is not.

Nested values

A column can hold a struct or an array, and `struct` and `array` are the accessors for them. `COLLECT` is how an array gets there — it gathers a group's values instead of reducing them:

```
import com.darkcollective.relix.ast.AggregateOperator;

Relation gathered = relix.relation("Orders")
    .aggregate(List.of("customer"), List.of(agg(AggregateOperator.COLLECT, "amount", "amounts")));

for (Tuple row : gathered.toList()) {
    System.out.println(row.string("customer") + " → "
        + row.array("amounts").stream().map(Value::asDisplayString).toList());
}
```

Ada → [100, 75]

Grace → [250]

`unnest` is the inverse — it flattens an array column back into one row per element, so `COLLECT` and `unnest` round-trip:

```
gathered.unnest("amounts").toList().forEach(System.out::println);
```

```
(customer=Ada, amounts=100)
(customer=Grace, amounts=250)
(customer=Ada, amounts=75)
```

A struct is read the same way, as a map of its fields:

```
Relation boxed = relix.relation("Orders").project(List.of(
    projected(attr("order_id")),
    projected(structOf(field("who", attr("customer")), field("how_much", attr("amount")),
        "detail"))));
```

```
Tuple boxedFirst = boxed.toList().getFirst();
System.out.println(boxedFirst.struct("detail").get("who").asDisplayString());
System.out.println(boxedFirst.get("detail").asDisplayString());
```

Ada

```
{who: Ada, how_much: 100}
```

Streaming

`stream()` is the opt-in for a caller who wants the engine's laziness and accepts what comes with it: **the stream is a resource and you close it**. Use it in a try-with-resources, every time.

```
try (Stream<Tuple> lazy = relix.relation("Orders").stream()) {
    lazy.limit(2).forEach(row -> System.out.println(row.string("customer")));
}
```

Ada

Grace

Closing early is also how a partial read is cancelled. The engine is pull-based end to end, so once nobody pulls, nothing further is computed — which is what makes a lazy stream the right tool for a result too large to hold, and for one that never ends at all:

```
relix.define("""
    source Naturals from generator { name: "Naturals" };
    """);

try (Stream<Tuple> forever = relix.relation("Naturals").stream()) {
    System.out.println(forever.limit(5).map(row -> row.longValue("n").toString()).toList());
}
```

```
[0, 1, 2, 3, 4]
```

`toList()` refuses that relation rather than hanging. A collecting *consumer* is not part of the query tree, so it is the terminal — not the planner — that has to ask whether the relation ever ends:

```
try {
    relix.relation("Naturals").toList();
} catch (RuntimeException e) {
    System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
}
```

```
UnboundedRelationException: cannot collect an unbounded relation into a list; add a bound (e.g. limit(n)), or stream it instead
```

A stream that got away

A stream you never close goes on holding what it reads through — a live `ResultSet`, its statement, and the pooled database connection behind them. That is the cost of the laziness, and it is why every example above is in a try-with-resources. You do not have to guess whether one escaped: the session counts them.

```
Stream<Tuple> escaped = relix.relation("Orders").stream();
System.out.println(escaped.findFirst().map(row -> row.string("customer")).orElseThrow());
System.out.println("open: " + relix.openStreams());
```

```
escaped.close();
System.out.println("open: " + relix.openStreams());
```

```
Ada
open: 1
open: 0
```

Reading a row is not closing: `findFirst()` takes one and leaves the rest of the query standing. A collecting terminal is different — `toList()`, `count()` and `run()` drain and close, so they never leave anything behind, and a non-zero count is always a `stream()` result.

Closing the session closes whatever is still open and warns with the count, so the connection goes back rather than out with the process. Asserting the count is zero before you close is how that stops being a warning you read and starts being a test that fails.

Size, and what fits in memory

Streaming bounds what a *terminal* holds. It does not bound what the *query* holds, and the difference is where an embedded engine surprises people.

Most operators stream: σ , π , ρ , λ and every join except the full outer one pass rows through as they arrive. The rest have to see all their input before they can emit their first row — γ groups, τ sorts, δ and the deduplicating set operations $\cup$, $\cap$ and $-$ compare against everything seen so far, $\bowtie$ and $\div$ and $\bowtie$ buffer a side. A hash join buffers its build side. Those buffers live in the JVM heap and **nothing spills to disk**: the memory a query needs is set by the largest thing it buffers, which may be far larger than the result you asked for. Two things help — a cap that stops a runaway buffer with a useful error, and a report of what each one actually held.

The engine knows which operators block, and refuses the combination it can prove is hopeless — a blocking operator over a relation with no end:

```
try {
    relix.relation("τ n (Naturals)").stream();
}
```

```

} catch (RuntimeException e) {
    System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
}

```

UnboundedRelationException: cannot materialise unbounded relation for blocking operator τ (SORT); add a bound (e.g. λn) below it

That check is about *boundedness*, not size. A table with fifty million rows is perfectly bounded, so sorting it in-engine is a plan the engine will happily make. From the plan's point of view there is nothing wrong, which is why size is a question you answer for yourself, in two places: a cap you set before the run, and the plan you read.

A cap on what one operator buffers

`maxMaterializedRows` bounds how many rows a single blocking operator may hold. Cross it and the query stops with an error naming the operator, instead of running on until the heap is gone and failing with an `OutOfMemoryError` that names nothing:

```

try (Relix capped = Relix.builder().maxMaterializedRows(2).build()) {
    capped.table("Wide", List.of("n"), List.of(
        Map.of("n", 1), Map.of("n", 2), Map.of("n", 3)));
    try {
        capped.relation(" $\tau$  n (Wide)").toList();
    } catch (RuntimeException e) {
        System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
    }
}

```

QueryExecutionException: Sort buffered more than 2 rows; a blocking operator holds its whole input in memory. Reduce what reaches it (a σ or λ below it, or a pushdown), or raise `maxMaterializedRows`

Read the number as a guard rail rather than a memory limit. It bounds **one** operator, not the sum of every buffer in the run — the same shape `maxFixpointRounds` has, and for the same reason: a recursion that buffers the same thousand rows on each of a hundred rounds never holds more than a thousand, and a running total would refuse it. So a plan with many blocking operators can exceed the cap in aggregate while no single operator does. What it converts is the failure that actually happens — one operator swallowing a table.

It is set by default, to ten million rows per operator, and the number is chosen to be one no reasonable query reaches and every runaway does. What it buys is not memory — ten million rows is a great deal of memory — but attribution: past it the failure names the operator and this knob, where an `OutOfMemoryError` names nothing and takes your process with it. Pass `ExecutionContext.UNLIMITED_MATERIALIZED_ROWS` to turn it off.

`maxFixpointRounds` is deliberately **not** capped by default, and the asymmetry is the point. How many rounds a legitimate recursion needs is a property of your data — a transitive closure over a long chain needs one round per hop — so any default number refuses some correct query, and a truncated answer is harder to notice than a slow one. Set it when you know what your recursion should cost.

Stopping a query

Interrupt the thread running it. The engine checks between rows, and between each row a blocking operator pulls while filling its buffer, so a cancellation lands promptly rather than at the end of a drain:

```

Thread worker = new Thread(() -> relix.relation(" $\tau$  amount (Orders)").toList());
worker.start();
worker.interrupt();

```

That is the mechanism the platform already routes everything through — `Future.cancel(true)`, `ExecutorService.shutdownNow`, and a pool being torn down all raise the same flag — so there is nothing new to hold. The query stops with an `QueryExecutionException` and the flag is left raised, so a thread you hand back to a pool still knows it was interrupted.

One limit is worth stating plainly: a thread blocked inside a JDBC driver is not interruptible. Where the query has been pushed down, the engine is waiting on the database and notices only when it answers. Closing the stream early is the other lever, and it works for the same reason — the engine is pull-based end to end, so nothing runs that nobody asked for.

What a run actually buffered

`run()` reports each buffer as it completes, so you need not guess which operator was the expensive one:

```
relix.relation("γ customer, SUM(amount) → total (Orders)").run().events().stream()
    .filter(e -> e.code().equals("MATERIALIZE"))
    .forEach(e -> System.out.println(
        e.description() + " (" + e.metrics().rows().getAsLong() + ")");
```

```
buffered 3 rows for Aggregate (3)
```

The row count is on the event as a number as well as in the sentence, so a host can compare it against the estimate the plan carried rather than parse it back out of prose.

Reading the plan

`explain()` draws the boundary: anything below a `PushedScan` happened in the backend, and anything above it happened here, on this heap.

```
System.out.println(relix.relation("γ status, SUM(amount) → total (Orders)").explain());
```

```
Aggregate ~2 rows
└─ Scan Orders ~3 rows
```

That γ is above a scan of a table this program is holding, so it groups here — fine for three rows, and the same shape over a database table is the query to worry about. Three things move the boundary:

- **Let the backend do the blocking work.** A γ , τ or δ that folds into a `PushedScan` is buffered by the database, which has disk to spill to. The language reference's *Pushdown* page says exactly which operators fold and where they stop folding.
- **Bound the input, not the output.** A λ above a sort still sorts everything. Put the limit where it can be pushed, and check with `explain()` that it was.
- **Ask for the number rather than the rows.** `count()` is γ `COUNT(*)` and pushes down, so it crosses the wire as one row — `toList().size()` fetches the whole result to measure it.

`toList()` refuses a relation with no end, as the section above shows. It does not refuse a large one: a collecting consumer cannot tell fifty million rows from five until it has read them, which is what the cap is for.

The rows and what produced them

`run()` is `toList()` with the run's own event feed attached — the rules that fired, the planner's choices, what was pushed to a backend, and the row count the run ended with:

```
import com.darkcollective.relix.embed.Rows;

Rows result = relix.relation("σ amount > 50 (Orders)").run();

System.out.println(result.size() + " rows");
result.events().forEach(event ->
    System.out.println(" " + event.stage() + " " + event.code() + " " + event.description()));
```

```
3 rows
EXECUTE SCAN scanned 3 rows
EXECUTE ROWS query delivered 3 rows
```

The row count arrives as an event rather than as a separate field, because a row stream's length is not known until something drains it — so the terminal that drains it is the one that can say so, and it says so on the feed everything else is already on.

For a result too large to collect, `stream(listener)` is the same feed without the collecting:

```
import com.darkcollective.relix.events.QueryEvent;

List<QueryEvent> feed = new java.util.ArrayList<>();
try (Stream<Tuple> traced = relix.relation("Orders").stream(feed::add)) {
    traced.forEach(row -> { });
}

feed.forEach(event -> System.out.println(event.stage() + " " + event.code()));
```

```
EXECUTE SCAN
```

Running the tree you wrote

Every terminal above optimises before it plans. That is deliberate — a library call that quietly skipped the rewriter would hand back a correct answer by a worse plan, with nothing to say it had — and `asWritten()` is how you opt out when you mean to:

```
Relation summed = relix.relation("γ SUM(amount * 1) (Orders)");
```

```
System.out.println(summed.toList().getFirst());
System.out.println(summed.asWritten().toList().getFirst());
```

```
(sum_amount=425)
(sum_expr=425)
```

One number, two column names, and that is the two trees showing through the rows rather than through a plan: the rewriter folds `amount * 1` to `amount`, and an unaliased aggregate takes its output name from the shape of its argument.

What it is for is comparing the two. That a rewrite preserves the answer is a claim, and the only way to test a claim about two trees is to run both of them over the same data — which needs both reachable from one relation:

```
Relation revenue = relix.relation("γ customer, SUM(amount) → total (Orders)");
```

```
System.out.println(revenue.asWritten().count() == revenue.count());
```

```
true
```

It settles execution only. `render()`, `explain()` and `plan()` already describe the relation as written, so they are unaffected, and on a relation that came from `optimized()` it changes nothing — that tree has been rewritten and nothing can un-rewrite it. It does survive composition, because it says how *you* want your query run rather than something about

one tree: `asWritten().limit(5)` still runs as written.

Where the time went

An embedded engine has no profiler attached to it, so the feed is the profiler. Every event that counts rows also times the step that produced them:

```
Rows timed = relix.relation("γ status, SUM(amount) → total (Orders)").run();

timed.events().stream()
    .filter(event -> event.metrics().duration().isPresent())
    .forEach(event -> System.out.println(event.code() + " - " + event.description()));
```

```
SCAN – scanned 3 rows
MATERIALIZE – buffered 3 rows for Aggregate
ROWS – query delivered 2 rows
```

The times themselves are wall clock — they move with the machine, the page cache and whatever else the host is doing — which is why the example above prints *which* steps were timed rather than how long they took. Read them to find the step worth looking at, not to assert that anything is fast enough.

What each one claims is worth knowing, because the three do not measure alike. `ROWS` is the query's own elapsed time and is **inclusive**: it covers every scan, buffer and join done on the query's behalf, since a pull-based engine does that work when it is asked and there is no interval that leaves the asking out. `SCAN` and `MATERIALIZE` are **self** time — only the pull is timed, so a scan reports what producing its rows cost and not what the operator above it then did with them.

That is the difference that makes the numbers usable. Timed inclusively, a scan under a join would report very nearly the whole query, and every plan would look like its leaves were the problem. Timed as they are, the parts nest inside the total, and the question *is the join the problem, or one of the scans?* is answered by comparing them:

```
java.time.Duration whole = timed.events().stream()
    .filter(event -> event.code().equals("ROWS"))
    .findFirst().orElseThrow().metrics().duration().orElseThrow();
java.time.Duration parts = timed.events().stream()
    .filter(event -> !event.code().equals("ROWS"))
    .flatMap(event -> event.metrics().duration().stream())
    .reduce(java.time.Duration.ZERO, java.time.Duration::plus);

System.out.println("the parts fit inside the whole: " + (parts.compareTo(whole) <= 0));

the parts fit inside the whole: true
```

When most of a query's time is unaccounted for by its scans and buffers, the work is in the operators between them.

What a drained read tells the engine

A terminal that reads to the end has measured something true, and the engine keeps it: the next plan over that relation prefers the measured count to the one it would have estimated.

The condition is exactly "read to the end". A caller who takes three rows and closes the stream has measured their own patience rather than the relation, so a partial read reports nothing — which is why `stream()`, the terminal that hands you the lifecycle, usually

measures only whatever it happened to exhaust. [Working without a database](#) follows what happens to those numbers next.

Looking at what the engine did

Reading a query, improving it and handing back better text is a whole use of the engine, and nothing on this page runs anything. The rewrite, the plan, the estimates and the session's own inventory are all available without a row moving.

```
Relix relix = Relix.open();

relix.define("""
    source Orders from csv("./orders.csv") { header: true, schema: {
        order_id: NUMBER, customer_id: NUMBER, status: STRING, amount: NUMBER } };
    source Customers from csv("./customers.csv") { header: true, schema: {
        customer_id: NUMBER, name: STRING } };
    BigOpenOrders := { σ status = 'OPEN' (σ amount > 100 (Orders)) };
    """);
```

The one rule

Inspection is staged; execution is not.

`render()` and `optimized().render()` differ, because the first shows what you wrote and the second shows what the rewriter made of it. `stream()` and `optimized().stream()` do *not* differ, because every execution terminal optimises first.

So `optimized()` exists to let you **see** the rewrite, never to switch it on. Nothing is gained by calling it before running a query, and nothing is lost by not calling it.

The one way out is named at the call site rather than implied: `asWritten()` runs the tree you wrote, which is what comparing the two trees needs. See [Running the tree you wrote](#).

The expression

`render()` prints the tree back as Relix text, and `schema()` gives its heading.

```
Relation query = relix.relation("π customer_id, amount (BigOpenOrders)");

System.out.println(query.render());
query.schema().columns().forEach(column ->
    System.out.println("  " + column.name() + " : " + column.type().display()));

π customer_id, amount (BigOpenOrders)
  customer_id : number
  amount      : number
```

Rendering is not an echo of the input: it prints the tree, so whitespace and the spelling of a literal are normalised. What it guarantees is that the text parses back to the same expression — which is what makes "read a query, improve it, hand back better text" work at all.

The rewrite

`optimized()` returns the rewritten relation. `events()` says which rules fired, and `rewrites()` says what each one fired on.

```
Relation rewritten = query.optimized();
```

```
System.out.println(rewritten.render());
rewritten.events().forEach(event ->
    System.out.println("  " + event.code() + " - " + event.description()));

π customer_id, amount (σ (status = "OPEN") ∧ (amount > 100) (π customer_id, status, amount (Orders)))
  INLINE-001 – view 'BigOpenOrders' inlined
  RENAME-002 – rename removed – nothing references the alias BigOpenOrders
  SEL-002 – two adjacent selections merged into one conjunctive predicate
  PROJ-004 – columns pruned at Orders: 3 of 4 read
```

Two rules are visible there: the view was inlined, and the two selections were merged into one conjunction. That is the difference between the query a person writes and the query the engine runs, and it is worth looking at when a query is slower than it should be.

`rewrites()` is the same story as records rather than as a feed. An event says a rule fired; a record says what it fired on, which is the difference between a trace and a report:

```
rewritten.rewrites().forEach(record ->
    System.out.println(record.code().code() + " on " + record.relationName()
        + " - " + record.detail()));

INLINE-001 on query[1] – view 'BigOpenOrders' inlined
RENAME-002 on query[1] – rename removed – nothing references the alias BigOpenOrders
SEL-002 on query[1] – two adjacent selections merged into one conjunctive predicate
PROJ-004 on query[1] – columns pruned at Orders: 3 of 4 read
```

The plan

`explain()` shows the physical plan — join algorithms, what was folded into a backend query, and, where the cost model has something to say, each step's estimated row count. Like `render()`, it is staged, so this is the plan for the query as written:

```
Relation joined = relix.relation("Orders")
    .join(relix.relation("Customers"),
        eq(attr("Orders.customer_id"), attr("Customers.customer_id")))
    .select(gt(attr("amount"), num(100)));

System.out.println(joined.explain());

Select
└─ Join INNER/HASH build=RIGHT
   └─ Scan Orders
      └─ Scan Customers
```

Planning reads no rows. It asks the catalog for schemas and the cost model for cardinalities, and that is all — which is why an `explain()` works against a database that is not reachable, and why the connector a planning call builds refuses every open.

The same plan is available as JSON, for a tool rather than a person:

```
System.out.println(relix.relation("σ amount > 100 (Orders)").explainJson());

{"op": "Select", "estimatedRows": null, "schema": {"open": false, "columns": [{"name": "order_id", "type": "N"}, {"name": "customer_id", "type": "N"}, {"name": "status", "type": "S"}, {"name": "amount", "type": "N"}]}, "children": [{"op": "Scan", "source": "Orders", "estimatedRows": null, "schema": {"open": false, "columns": [{"name": "order_id", "type": "N"}, {"name": "customer_id", "type": "N"}, {"name": "status", "type": "S"}, {"name": "amount", "type": "N"}]}, "children": []}]}
```

`renderJson()` is the same for the tree as written, before any planning: one object per operator, each with the heading the analyser inferred for it.

```
System.out.println(relix.relation("σ amount > 100 (Orders)").renderJson());

{"op": "Selection", "label": "σ amount > 100", "schema": {"open": false, "columns": [{"name": "order_id", "type": "N"}, {"name": "customer_id", "type": "N"}, {"name": "status", "type": "S"}, {"name": "amount", "type": "N"}]}, "materialization": "stream", "children": [{"op": "Relation", "label": "Order s [SRC]", "relation": "Orders", "kind": "SRC", "schema": {"open": false, "columns": [{"name": "order_id", "type": "N"}, {"name": "customer_id", "type": "N"}, {"name": "status", "type": "S"}, {"name": "amount", "type": "N"}]}, "materialization": "stream", "children": []}]}
```

And as an object, when the interesting part is a number in it rather than the rendering:

```
var planned = relix.relation("σ amount > 100 (Orders)").plan();
```

```
System.out.println(planned.plan().getClass().getSimpleName());
System.out.println(planned.estimate().rows(planned.plan()));
```

```
Select
OptionalLong.empty
```

`estimate()` answers an `OptionalLong`, and the empty case is load-bearing: **unknown is not zero**. A relation nothing has measured and no statistic describes has no estimate, and reporting one as `0` would be a confident wrong answer where an absent one is a true one.

Both at once

`explain(listener)` gives the plan and the planner's decisions from **one** planning pass. Two calls would report the same work twice, which is the sort of double-counting a trace is supposed to prevent:

```
import com.darkcollective.relix.events.QueryEvent;

List<QueryEvent> decisions = new java.util.ArrayList<>();
String plan = joined.explain(decisions::add);

System.out.println(plan.lines().findFirst().orElseThrow());
decisions.forEach(event ->
    System.out.println(" " + event.stage() + " " + event.code() + " " + event.description()));
```

```
Select
  PLAN JOIN INNER join: HASH, build=RIGHT
```

Nothing here is timed, and that is not an omission: planning and rewriting are decisions taken before a row moves, and a duration on each would be noise on the feed rather than information. Time is reported by the events a *run* produces — see [Where the time went](#).

The session itself

A session renders back as `.relix` text, so what it holds is inspectable as source rather than only as objects:

```
System.out.println(relix.definitions());
```

```
source Orders from csv("./orders.csv") { header: true, schema: { order_id: number, customer_id: number, status: string, amount: number }
  };
source Customers from csv("./customers.csv") { header: true, schema: { customer_id: number, name: string } };
BigOpenOrders := { σ status = "OPEN" (σ amount > 100 (Orders)) };
```

`statements()` is the same content as AST statements, for a program that would rather match on it than read it.

`ir()` is the analyser's own report: every symbol with its heading, every view's expression tree, and the root queries. It is the thing to print when a query resolves to something other than what you expected, because it shows what each name was taken to mean. Its symbol table is long — it lists the `relix.*` catalogs alongside your own declarations — so this prints only the trees:

```
String report = relix.ir();
System.out.println(report.substring(report.indexOf("— EXPRESSION TREES")));
```

```
— EXPRESSION TREES —
BigOpenOrders [QR] order_id:N customer_id:N status:S amount:N
  σ status = "OPEN"
```

```
└─ σ amount > 100
   └─ Orders [SRC]
```

The session's `model()` gives the same report, and `ir(relations)` narrows it to the relations named, which is how a tool shows the tree behind one query rather than the whole script:

```
relix.model().ir(List.of("BigOpenOrders", "Orders")).lines()
    .filter(line -> line.startsWith(" BigOpenOrders") || line.startsWith(" Orders"))
    .forEach(System.out::println);
```

BigOpenOrders	QR	order_id:N	customer_id:N	status:S	amount:N
Orders	SRC	order_id:N	customer_id:N	status:S	amount:N

What the engine is made of

`relix.version` is a relation, not a Java accessor, and it lists one row per component of the running engine — itself, the facade, and every provider actually discovered on the classpath:

```
relix.relation("π component, kind (relix.version)").toList()
    .forEach(System.out::println);
```

```
(component=relix-engine, kind=engine)
(component=relix-builtin, kind=function-library)
(component=relix-embed, kind=facade)
(component=clf, kind=connector)
(component=csv, kind=connector)
(component=gedcom, kind=connector)
(component=log, kind=connector)
(component=ojAlgo, kind=solver)
(component=org.h2.Driver, kind=driver)
```

Which providers are installed is otherwise unanswerable from inside a running program, because discovery is a classpath scan and a missing provider fails nothing a compiler can see. Being a relation rather than an accessor is what lets it be filtered and joined with the operators already at hand — and `relix.version` is one of several such catalogs, all queryable the same way. [The engine describes itself](#) is the tour of them.

The engine describes itself

Everything a program might want to know about a session — which relations exist, what columns they carry, which functions are installed, what the last run did — is exposed in the reserved `relix.*` namespace **as relations**.

That is a deliberate choice rather than a shortcut. There is no Java accessor beside these returning the same facts, because a second surface is a second thing to keep true, and two answers to one question eventually disagree. Making the catalog a relation means it is filtered, joined, aggregated and sorted with the operators already learned, and a catalog query composes with an ordinary one because there is no seam between them.

```
Relix relix = Relix.open();

relix.define("""
    source Orders from csv("./orders.csv") { header: true, schema: {
        order_id: NUMBER, customer_id: NUMBER, amount: NUMBER, status: STRING } };
    source Customers from csv("./customers.csv") { header: true, schema: {
        customer_id: NUMBER, name: STRING, region: STRING } };
```

```

source Products from csv("./products.csv") { header: true, schema: {
  sku: STRING, price: NUMBER } };

Open      := { σ status = "OPEN" (Orders) };
ByRegion  := { γ region, SUM(amount) → revenue (Open ⋈ Customers) };
Stale     := { σ amount < 0 (Orders) };
""");

```

Nothing there reads a file. Both sources declare their schemas, so everything on this page is answered from what the session was told.

What is actually installed

Providers — connectors, function libraries, solvers, JDBC drivers — are found by a `ServiceLoader` scan. A missing one fails nothing a compiler can see and nothing a unit test can see; it fails the first query that needs it, at run time, in production. `relix.version` is how a program asks instead of assuming:

```

relix.relation("τ kind, component (relix.version)").toList().forEach(System.out::println);

(component=clf, kind=connector, version=<version>)
(component=csv, kind=connector, version=<version>)
(component=gedcom, kind=connector, version=<version>)
(component=log, kind=connector, version=<version>)
(component=org.h2.Driver, kind=driver, version=2.2)
(component=relix-engine, kind=engine, version=<version>)
(component=relix-embed, kind=facade, version=unknown)
(component=relix-builtin, kind=function-library, version=<version>)
(component=ojAlgo, kind=solver, version=<version>)

```

One row per component the process has loaded. `engine` and `facade` are Relix's own; the rest were contributed by whatever assembled the classpath. `<version>` stands in for the version each component was built at, which is a fact about your build rather than about Relix — it is the one thing on this page that has no value a page could print.

Because it is a relation, the startup check a program actually wants is a query:

```

List<String> connectors = relix.relation("π component (σ kind = 'connector' (relix.version))")
    .toList().stream().map(row -> row.string("component")).sorted().toList();

System.out.println("connectors: " + connectors);
System.out.println("this session can read CSV: " + connectors.contains("csv"));

connectors: [clf, csv, gedcom, log]
this session can read CSV: true

```

That is a real check rather than a decorative one: the tokens listed are the ones a `connection` declaration may name, and a session whose classpath is missing the connector it needs fails at the first query that wants it — unless it asks first.

A `version` of `unknown` is not a failure — it means the packaging carries none, which is normal for loose classes on a classpath. The name is the answer; the version sharpens it.

What the session knows about

`relix.relations` has one row per relation the session can name, and `relix.columns` one row per column of each:

```

relix.relation("π name, kind, materialization (τ name (relix.relations))")
    .toList().forEach(System.out::println);

```

```
(name=ByRegion, kind=QR, materialization=bag)
(name=Customers, kind=SRC, materialization=stream)
(name=Open, kind=QR, materialization=stream)
(name=Orders, kind=SRC, materialization=stream)
(name=Products, kind=SRC, materialization=stream)
(name=Stale, kind=QR, materialization=stream)
```

`kind` distinguishes what a relation *is* — `SRC` a declared source, `QR` a view defined in the session, `SYS` one of these catalogs. A tool built on Relix enumerates what it can query by reading exactly this.

The two obvious questions about columns are the same relation read from its two sides, and both ship as table-valued functions:

```
relix.relation("relix.schema(\"Customers\")").toList().forEach(System.out::println);
relix.relation("relix.find(\"customer_id\")").toList().forEach(System.out::println);
```

```
(column=customer_id, type=N, ordinal=0)
(column=name, type=S, ordinal=1)
(column=region, type=S, ordinal=2)
(relation=Orders, type=N)
(relation=Customers, type=N)
(relation=Open, type=N)
(relation=Stale, type=N)
```

`relix.schema(r)` asks what columns a relation has; `relix.find(c)` asks which relations have a column. Both are ordinary views over `relix.columns` — there is no new language construct, and you can read their definitions the same way you read your own.

Composing a catalog query with an ordinary one

The catalogs being relations is not a stylistic point. It means a program can write a rule about its own schema and *evaluate* it, rather than walking an object graph in Java:

```
Relation missingKey = relix.relation("""
    (π name (σ kind = "SRC" (relix.relations))) - (π relation → name (relix.find("customer_id")))
    """);

System.out.println(missingKey.render());
List<String> offenders = missingKey.toList().stream().map(t -> t.string("name")).toList();
System.out.println(offenders.isEmpty()
    ? "every source carries customer_id"
    : "sources missing customer_id: " + offenders);

(π name (σ kind = "SRC" (relix.relations))) - (π relation → name (relix.find("customer_id")))
sources missing customer_id: [Products]
```

A set difference *is* the rule: subtract the relations that have the column from the ones that should, and whatever is left is the violation. `Products` is a catalogue of stock and genuinely has no customer, which is the other half of a rule like this — it tells you where to look, and you decide. The same shape answers "which relations does nothing reference?" and "does anything depend on this one?" — which ship as views too, over the dependency graph the analyser already built:

```
System.out.println("unused: " + relix.relation("τ name (relix.unused)").toList());
System.out.println("upstream: " + relix.relation("relix.deps(\"ByRegion\")").toList());
System.out.println("impact: " + relix.relation("relix.impact(\"Orders\")").toList());
System.out.println("cycles: " + relix.relation("relix.cycles").toList());

unused: [(name=ByRegion), (name=Products), (name=Stale)]
upstream: [(depends_on=Open), (depends_on=Customers), (depends_on=Orders)]
impact: [(dependent=Open), (dependent=Stale), (dependent=ByRegion)]
```

```
cycles:  []
```

Stale is defined and referenced by nothing, which is what `relix.unused` is for; the top-level outputs appear there too, by construction. `relix.impact` is the blast radius of a change — everything that would have to be re-checked if `Orders` moved.

The installed functions

`relix.functions` carries one row per function the session resolves against, with the properties the optimizer reads:

```
relix.relation("γ category, COUNT(*) → n (relix.functions)")
    .sort(asc("category"))
    .toList()
    .forEach(System.out::println);
```

```
(category=conditional, n=3)
(category=conversion, n=3)
(category=datetime, n=16)
(category=math, n=15)
(category=nested, n=1)
(category=string, n=13)
(category=typecheck, n=2)
```

The properties are the interesting columns, because they are what decides whether a call can be folded into a backend query or hoisted out of a loop. A function that reads the clock or the random-number generator is neither pure nor deterministic, and the catalog says so:

```
relix.relation("""
    π name, pure, deterministic
    (σ pure = "false" v deterministic = "false" (relix.functions))
    """).sort(asc("name")).toList().forEach(System.out::println);
```

```
(name=CURRENT_DATE, pure=false, deterministic=false)
(name=CURRENT_TIME, pure=false, deterministic=false)
(name=NOW, pure=false, deterministic=false)
(name=Rand, pure=false, deterministic=false)
```

A library of your own appears here exactly as the shipped ones do — same columns, same resolution, its own category:

```
import com.darkcollective.relix.function.FunctionContext;
import com.darkcollective.relix.function.FunctionLibrary;
import com.darkcollective.relix.function.FunctionSignature;
import com.darkcollective.relix.function.ScalarFunction;
import com.darkcollective.relix.function.StrictScalarFunction;
import com.darkcollective.relix.symbol.FunctionProperty;
import com.darkcollective.relix.symbol.ParameterDefinition;
import com.darkcollective.relix.symbol.ScalarType;
import java.math.BigDecimal;
import java.util.Set;

class WithVat implements StrictScalarFunction {
    public FunctionSignature signature() {
        return FunctionSignature.of("WithVat", ScalarType.NUMBER, "pricing",
            Set.of(FunctionProperty.PURE, FunctionProperty.DETERMINISTIC),
            new ParameterDefinition("amount", ScalarType.NUMBER));
    }

    public Value invoke(FunctionContext context, List<Value> arguments) {
        return new com.darkcollective.relix.value.NumberValue(
```

```

        ((com.darkcollective.relix.value.NumberValue) arguments.getFirst())
            .value().multiply(new BigDecimal("1.20")));
    }
}

relix.functions(new FunctionLibrary() {
    public String name() {
        return "pricing";
    }

    public List<ScalarFunction> scalarFunctions() {
        return List.of(new WithVat());
    }
});

relix.relation("σ category = 'pricing' (relix.functions)").toList()
    .forEach(System.out::println);

(name=WithVat, category=pricing, arity=1, max_arity=1, return_type=N, pure=true, deterministic=true, idempotent=false, is_builtin=true)

```

Being indistinguishable from a shipped function is the provider seam working rather than an oversight: anything the bundled library can do, an installed one can. The line `is_builtin` draws is a different one — between a function the session **resolved**, shipped or installed alike, and one the script **declared**:

```

relix.define("def Discount(price: NUMBER) : NUMBER := { price * 0.9 };");

relix.relation("""
    π name, category, is_builtin (σ category ∈ {"pricing", "user"} (relix.functions))
    """).sort(asc("name")).toList().forEach(System.out::println);

(name=Discount, category=user, is_builtin=false)
(name=WithVat, category=pricing, is_builtin=true)

```

What the last run did

The other catalogs answer *what is*. `relix.events` answers *what happened* — the same feed of engine decisions that a trace prints, as rows.

It is necessarily the **previous** run's feed, and the reason is worth understanding rather than working around. A statement's events do not exist until it has been optimized, planned and executed, and all three happen after the analysis that would have to register the relation the statement is selecting from. A statement therefore cannot observe itself, and a session that has run nothing has nothing to show:

```

System.out.println(relix.relation("relix.events").count() + " events");

0 events

```

In Java that is rarely the route you want anyway, because a run hands its own feed straight back. `run()` returns the rows **and** the events that produced them:

```

relix.table("Sales", List.of("region", "amount"), List.of(
    Map.of("region", "North", "amount", 100),
    Map.of("region", "South", "amount", 250)));

Rows result = relix.relation("γ region, SUM(amount) → total (Sales)").run();

System.out.println(result.rows());
result.events().forEach(e -> System.out.println(e.stage() + " " + e.code() + " - " + e.description()));

[(region=North, total=100), (region=South, total=250)]
EXECUTE SCAN – scanned 2 rows

```

```
EXECUTE MATERIALIZE – buffered 2 rows for Aggregate
EXECUTE ROWS – query delivered 2 rows
```

Every stage the run actually reached is in there. That one only scanned, buffered and executed, since there was nothing to rewrite; the query below has a view in it, so its feed carries the `OPTIMIZE` rows too.

The middle line is the one to look for when a query is heavier than it should be. A `γ` cannot emit its first row until it has read its last, so it holds its whole input in memory, and `MATERIALIZE` says which operator did that and how many rows it held — the count is on the event as a number (`metrics().rows()`), not only in the sentence, and beside it `metrics().duration()` says how long that operator spent filling the buffer. Two rows here; on a query that is about to exhaust the heap it is the line that names the operator responsible. [Size, and what fits in memory](#) covers the cap that stops one before it does.

The catalog relation earns its place when the feed has to be *queried* rather than read — which is what an interactive host does, keeping one statement's events and handing them to the next. `sessionEvents` on the builder is that seam:

```
relix.define("Totals := { γ region, SUM(amount) → total (Sales) };");
Rows report = relix.relation("σ total > 50 (Totals)").run();

try (Relix next = Relix.builder().sessionEvents(report.events()).build()) {
    next.relation("γ stage, COUNT(*) → events (relix.events)")
        .sort(asc("stage"))
        .toList()
        .forEach(System.out::println);

    next.relation("π code, description (relix.rules)").toList()
        .forEach(System.out::println);
}

(stage=EXECUTE, events=3)
(stage=OPTIMIZE, events=2)
(code=INLINE-001, description=view 'Totals' inlined)
(code=RENAME-002, description=rename removed – nothing references the alias Totals)
```

`relix.rules` is the `OPTIMIZE` slice of the same feed — defined as a view over it rather than computed separately, so the two cannot disagree about what fired. Sorting, grouping and filtering a trace is what having it as rows buys; reading one run's decisions in Java is `events()` and `rewrites()` on the relation itself, which [Looking at what the engine did](#) covers.

One consequence of the scoping catches everyone once: **the query that reads the feed is itself a run**, so a feed is readable once. Take the slices you want out of it in one query, or re-run the statement that produced it.

Provenance: where a row came from

"Which source rows produced this answer, and how many ways?" is a question about a result that the result itself does not carry. Relix answers it two ways: as an operator inside a query, and as a whole-relation annotation over a semiring.

```
Relix relix = Relix.open();

relix.table("Orders", List.of("order_id", "customer_id", "amount"), List.of(
    Map.of("order_id", 1, "customer_id", 10, "amount", 100),
    Map.of("order_id", 2, "customer_id", 11, "amount", 250),
```

```

        Map.of("order_id", 3, "customer_id", 10, "amount", 75)));

relix.table("Customers", List.of("customer_id", "region"), List.of(
    Map.of("customer_id", 10, "region", "North"),
    Map.of("customer_id", 11, "region", "South")));

```

Lineage as a column

`why()` is the ω operator. It passes its input through unchanged and adds one nested column, `provenance`, holding the rows that produced each result:

```

Relation joined = relix.relation("Orders")
    .join(relix.relation("Customers"),
        eq(attr("Orders.customer_id"), attr("Customers.customer_id")))
    .project("order_id", "region");

Relation explained = joined.why();

System.out.println(explained.render());
explained.schema().columns().forEach(column ->
    System.out.println("  " + column.name() + " : " + column.type().display()));

 $\omega$  ( $\pi$  order_id, region ((Orders)  $\bowtie$  Orders.customer_id = Customers.customer_id (Customers)))
  order_id : number
  region   : string
  provenance : any

```

The added column is `ANY`, because its shape is a document rather than a scalar. What it holds is one entry per *derivation* — a distinct way the row could have been produced — each naming the base rows that combined to make it:

```

Tuple firstRow = explained.toList().getFirst();

System.out.println(firstRow.string("region"));
System.out.println(firstRow.get("provenance").asDisplayString());

North
[{coefficient: 1, variables: [{relation: Customers, ordinal: 1, columns: {customer_id: 10, region: North}}, {relation: Orders, ordinal: 1
4 , columns: {amount: 100, customer_id: 10, order_id: 1}}], truncated: false}]

```

Because it is a column, the rest of the algebra applies to it. Lineage can be filtered, projected and unnested like any other nested value — which is the whole point of reifying it rather than reporting it:

```

System.out.println(explained.unnest("provenance").count() + " derivations in total");

3 derivations in total

```

`why()` is a boundary: the lineage bookkeeping runs *below* it and is frozen into the column *at* it. Above it, `provenance` is ordinary data, so nothing further re-threads the annotation and a query does not pay for lineage it has already collected.

Annotating a whole relation

The other form annotates every result row with a value from a **semiring** — an algebra with a "combine alternatives" operation and a "combine steps" operation. Which semiring you pick decides what the annotation means.

```

import com.darkcollective.relix.provenance.CountingSemiring;
import com.darkcollective.relix.processor.provenance.Annotated;

var counted = relix.relation(" $\pi$  region (Orders  $\bowtie$  Customers)")

```

```

        .provenance(CountingSemiring.INSTANCE);

counted.stream().forEach((Annotated<java.math.BigInteger> annotated) ->
    System.out.println(annotated.row() + " x" + annotated.annotation()));

Row{region=North} x2
Row{region=South} x1

```

North appears with a multiplicity of 2 because two orders produced it. That is the counting semiring's answer to "how many ways": under set semantics the duplicate row is gone, and the annotation is what remembers that there were two of it.

The general form of that is worth holding on to, because it is what the number means everywhere: **a counting annotation counts derivations, not rows returned**. For most of the algebra the two coincide — a projection, a filter, a join and a `u` all keep duplicates, so the annotation and the number of rows the query yields agree exactly. They part company at the operators that deduplicate: `A u A` returns each row once and annotates it with the derivations from both sides. Neither number is wrong; they answer different questions. The boolean semiring has no such gap, because `v` is idempotent — which is the same reason set semantics is the boolean reading of a K-relation in the first place.

Swap the semiring and the same query answers a different question. The boolean semiring is plain reachability — "was this row derivable at all":

```

import com.darkcollective.relix.provenance.BooleanSemiring;

relix.relation("π region (Orders ⋈ Customers)")
    .provenance(BooleanSemiring.INSTANCE)
    .stream()
    .forEach(annotated -> System.out.println(annotated.row() + " " + annotated.annotation()));

Row{region=North} true
Row{region=South} true

```

Six semirings ship, and they differ only in what `plus` and `times` mean:

Semiring	An annotation means
<code>BooleanSemiring</code>	whether the row is derivable at all — set semantics, the default
<code>CountingSemiring</code>	how many derivations there are — bag multiplicity, path counts
<code>TropicalSemiring</code>	the cost of the cheapest derivation — shortest path
<code>SecurityLattice</code>	the clearance a row's derivation requires
<code>PolynomialSemiring</code>	the full lineage expression — which rows, in which combinations
<code>PathCostSemiring</code>	the cheapest cost together with the route that achieves it

A weighted variant reads each base row's cost from a column, which is what the tropical semiring needs to have anything to minimise. Customer 10 placed orders of 100 and 75, and the annotation is the cheaper of the two derivations:

```

import com.darkcollective.relix.provenance.TropicalSemiring;

relix.relation("π customer_id (Orders)")
    .provenance(TropicalSemiring.INSTANCE, "amount")
    .stream()
    .forEach(annotated -> System.out.println(annotated.row() + " " + annotated.annotation()));

```

```
Row{customer_id=10} 75.0
Row{customer_id=11} 250.0
```

Writing your own

A `Semiring<K>` is four operations and one reading. `zero`, `one`, `plus` and `times` say how annotations combine; `base` says what a source row is worth in the first place, and is where a weight column enters. Nothing else is needed, and nothing in the engine recognises a semiring by name — an implementation of your own is threaded exactly as a bundled one is, through every positive operator and through weighted transitive closure.

The tropical example above minimised. The same query under a semiring that *adds* its derivations reports each customer's total instead, and the only difference is `plus`:

```
import com.darkcollective.relix.provenance.BaseTuple;
import com.darkcollective.relix.provenance.Semiring;

import java.math.BigDecimal;

// Alternative derivations add; joint requirements multiply; a base row is worth
// whatever the weight column says, and one that carries no weight is neutral.
class TotalSemiring implements Semiring<Double> {
    public Double zero() {
        return 0.0d;
    }

    public Double one() {
        return 1.0d;
    }

    public Double plus(Double a, Double b) {
        return a + b;
    }

    public Double times(Double a, Double b) {
        return a * b;
    }

    public Double base(BaseTuple tuple) {
        return tuple.weight().map(BigDecimal::doubleValue).orElse(one());
    }
}

relix.relation("π customer_id (Orders)")
    .provenance(new TotalSemiring(), "amount")
    .stream()
    .forEach(annotated -> System.out.println(annotated.row() + " " + annotated.annotation()));

Row{customer_id=10} 175.0
Row{customer_id=11} 250.0
```

`BaseTuple` is the source row as a semiring can read it: which leaf it came from, which occurrence it is within that leaf, its weight if it has one, and its column values. The last two are what let an annotation depend on the data — a cost, a multiplicity, a probability — and the first two are what let it name the row, which is how lineage mints a distinct variable per occurrence.

An absent weight is the semiring's own decision rather than the engine's, and the two defensible answers differ: `one()` leaves a row neutral, while a fixed constant makes an unweighted graph a defined special case instead of a degenerate one. A kinship coefficient takes the second reading — every parent edge is half a generation whether or not anyone

wrote a weight down — and gets the textbook answer over a bare (parent, child) relation.

Installing one by name

Implementing the interface is enough to pass a semiring to `provenance(...)` yourself. To make one resolvable *by name* — so that `--provenance --semiring kinship` reaches it, and it appears in the list of installed semirings — publish a `SemiringLibrary`:

```
import com.darkcollective.relix.provenance.NamedSemiring;
import com.darkcollective.relix.provenance.SemiringLibrary;

class KinshipLibrary implements SemiringLibrary {
    public String name() {
        return "kinship";
    }

    public List<NamedSemiring> semirings() {
        return List.of(new NamedSemiring("kinship", List.of("consanguinity"),
            new TotalSemiring(),
            "Coefficient of relationship between two individuals."));
    }
}
```

Declare it as a service — `provides com.darkcollective.relix.provenance.SemiringLibrary` in your `module-info`, and a `META-INF/services` file of the same name — and it is discovered on startup. A module path reads the first and a class path the second, so declaring both is what makes the library work either way. Where two libraries offer the same name the higher `priority()` wins, and the loss is reported rather than resolved silently.

Which to reach for

`why()` when the lineage is part of the answer — an audit column, a "show your working" view, something a downstream query filters on. It is an operator, so it composes.

`provenance(semiring)` when the annotation is *about* the whole result rather than in it, and when the algebra matters: counting derivations, cheapest routes, security levels. It sits off the normal streaming path and is never pushed to a backend, because no backend knows the semiring.

5. Building tools

Page	What it covers
Building tools on Relix	Parsing a script and loading imports; tokens for syntax highlighting; learning the joins a user writes and resolving them; the language reference; connectors and drivers

Building tools on Relix

An editor, a notebook, a command-line shell and an assistant all need more from the engine than a query's rows. They read scripts from somewhere of their own, colour text as the user types it, and learn the joins a user keeps writing. This page covers the three methods that serve them. The command-line tool and the interactive shell that ship with Relix are built on these methods and nothing else.

Parsing a script

`Relix.parse` turns text into a `Script`, the tree `define(Statement...)` takes, without analysing it. Nothing is resolved, so a script may name relations nothing declares:

```
import com.darkcollective.relax.lang.ast.Script;
import com.darkcollective.relax.lang.ast.Statement;

Script script = Relix.parse("""
    Large := { σ amount > 100 (Orders) };
    query Large;
    """);
for (Statement statement : script.statements()) {
    System.out.println(statement.getClass().getSimpleName());
}
```

```
AssignmentStatement
QueryStatement
```

Text that does not parse raises a `ScriptParseException`, which carries the line and column of the failure:

```
import com.darkcollective.relax.lang.ast.ScriptParseException;

try {
    Relix.parse("query { σ amount > (Orders) };");
} catch (ScriptParseException e) {
    System.out.println("line " + e.line() + ", column " + e.column());
}
```

```
line 1, column 29
```

A `ScriptLoader` is where parsing usually happens. A session resolves each `import` through its loader, and a loader reads the file and parses it, naming the file so a failure inside it points there. This one serves files from memory:

```
import java.util.Map;

Map<String, String> files = Map.of("orders.relax", """
    Orders := [| id | amount |
               | 1 | 120    |
```

```

        | 2 | 80 |];

        """);

try (Relix withImports = Relix.builder()
    .scriptLoader(path -> Relix.parse(files.get(path), path))
    .build()) {
    withImports.define("import \"orders.relax\";");
    System.out.println(withImports.relation("Orders").count());
}

2

```

Highlighting as the user types

`Relix.tokens` splits text into classified tokens for syntax highlighting. It is lenient, because an editor asks on every keystroke: the well-formed part is tokenized as usual and the rest becomes one `INCOMPLETE` token. Comments are tokens and whitespace is not. Offsets count chars, as `String.charAt` does.

```

import com.darkcollective.relax.embed.Token;

String line = "σ amount > 100 (Orders) -- the large ones";
for (Token token : Relix.tokens(line)) {
    System.out.println(token.kind() + " " + line.substring(token.start(), token.end()));
}

OPERATOR σ
IDENTIFIER amount
COMPARISON >
NUMBER 100
BRACKET (
IDENTIFIER Orders
BRACKET )
COMMENT -- the large ones

```

Half-typed text still colours up to the point it stops making sense:

```

String typing = "π name (\\"unfinished";
for (Token token : Relix.tokens(typing)) {
    System.out.println(token.kind() + " " + typing.substring(token.start(), token.end()));
}

OPERATOR π
IDENTIFIER name
BRACKET (
INCOMPLETE "unfinished

```

Learning the joins a user writes

A session's schema graph records how relations join: a `relate` statement declares an edge, and `Builder.relationships` supplies more. A tool can also learn edges from the queries its user writes. `demonstratedRelationships()` on a `SemanticModel` lists the column equalities written across a join, in a query or a view, that the graph does not already record:

```

import com.darkcollective.relax.symbol.graph.Relationship;

String data = ""
    Users := [| id | name |
              | 1 | Ada |
              | 2 | Bo  |];
    Issues := [| id | title | assignee | reporter |
              | 10 | Crash | 1       | 2       |];

```

```

        """;

List<Relationship> learned;
try (Relix first = Relix.open()) {
    first.define(data + "Assigned := { Issues ⋈ Issues.assignee = Users.id Users };");
    learned = first.model().demonstratedRelationships();
}
for (Relationship edge : learned) {
    System.out.println(edge.name() + " " + edge.origin());
}

```

```
Issues_assignee  LEARNED
```

Nothing ran to learn that: an analysis alone says which columns a join equated. `resolveJoins` goes the other way, assembling the join between relations from the edges the graph records. Given the learned edge, a later session can write the join without its condition:

```

import com.darkcollective.relix.symbol.graph.JoinResolution;
import com.darkcollective.relix.symbol.graph.SchemaGraph;

try (Relix later = Relix.builder()
    .relationships(SchemaGraph.EMPTY.with(learned.getFirst()))
    .build()) {
    later.define(data);
    JoinResolution resolution =
        later.model().resolveJoins(later.relation("Issues ⋈ Users").node());
    if (resolution instanceof JoinResolution.Resolved resolved) {
        System.out.println(resolved.program());
    }
}

```

```
(Issues) ⋈ Issues.assignee = Users.id (Users)
```

When more than one edge could join the relations, the answer is `Ambiguous` and lists the edges by name, which is a question a user can answer. With both of an issue's links to a user declared:

```

try (Relix tracker = Relix.open()) {
    tracker.define(data + ""
        relate "Assignee" Issues.assignee -> Users.id;
        relate "Reporter" Issues.reporter -> Users.id;
        "");
    JoinResolution resolution =
        tracker.model().resolveJoins(tracker.relation("Issues ⋈ Users").node());
    if (resolution instanceof JoinResolution.Ambiguous ambiguous) {
        for (JoinResolution.Alternative alternative : ambiguous.alternatives()) {
            System.out.println(alternative.label() + ": " + alternative.description());
        }
    }
}

```

```
Assignee: Issues.assignee = Users.id
Reporter: Issues.reporter = Users.id
```

`Relationship.edgeIdentity()` says when two edges join the same columns whatever they are called. A tool that lets its user reject a learned edge keeps the identities it rejected and skips any candidate that matches one.

The language reference

The language reference ships in the artifact, so a tool can show a user the page for what they typed with nothing else installed. `Relix.referencePages()` lists every page with the words it is found by, and `Relix.referencePage(path)` returns its markdown:

```
import com.darkcollective.relix.embed.ReferencePage;

ReferencePage selection = Relix.referencePages().stream()
    .filter(page -> page.keys().contains("σ"))
    .findFirst().orElseThrow();
System.out.println(selection.title() + " (" + selection.symbol() + "): " + selection.summary());
System.out.println(Relix.referencePage(selection.path()).orElseThrow().lines().findFirst().orElseThrow());

Selection (σ): Filter rows by a condition
# Name: Selection (σ / SELECT)
```

A function's page is not among them: it belongs to the library that offers the function, and `FunctionCatalog.documentation(docKey)` serves it, so a function library installed later is documented the same way the shipped one is.

Connectors and drivers

A tool that manages plugins asks where they live and which are installed, and a command-line tool registers the JDBC drivers an earlier run downloaded before it opens a session:

```
import com.darkcollective.relix.connectors.std.DriverProvisioner;
import com.darkcollective.relix.processor.connector.ConnectorProvisioner;

DriverProvisioner.loadInstalled();
System.out.println(new java.util.TreeSet<>(ConnectorProvisioner.installedTypes()));

[clf, csv, gedcom, log]
```

`ConnectorProvisioner.defaultDirectory()` is where a plugin is installed, and `loadInstalled()` returns one description per driver it registered, for a startup log.