1. Introduction 9
1.1. Terminology 10
1.1.1. Logic and Sql Terminology 10
1.1.2. Delightql-specific Terminology 11
2. Data Query Language (DQL) 15
2.1. Basics: Comments, Strings, and Semicolons 15
2.2. All reserved words 16
2.3. Case insensitivity 16
2.3.1. Stropping 16
2.4. From 16
2.4.1. All Columns 17
2.4.2. Argumentative Access 17
2.4.3. Aliasing a Table 17
2.4.4. Table Namespacing 18
2.4.5. Anonymous Table 18
2.4.6. Higher-Order Predicates 21
2.5. Order By 22
2.6. Limit 22
2.7. Projection 23
2.7.1. Project Retain 23
2.7.2. Rename 24
2.7.3. Generalized Projection 24
2.7.4. Argumentative Projection 25
2.7.5. Project Out 25
2.7.6. Map Covering 26
2.7.7. Basic Dimension Covering 27
2.7.8. Conditional Covering (If-Only) 28
2.7.9. Regular Expression Column Addressing 30
2.7.10. Rename Cover 31
2.7.11. Embedding 32
2.7.12. Map Embedding 32
2.7.13. Projection Summary 33
2.8. Distinct and Group By 34
2.8.1. Distinct 35
2.8.2. Group By 35
2.8.3. Group Delegates 36
2.8.4. Whole-Group Aggregate Functions 37
2.8.5. Internal Distinct 37
2.8.6. Filter Clauses 37
2.8.7. Having 38
2.9. Where 39
2.9.1. Domain Predicates 39
2.9.2. Argumentative Grounding 42
2.9.3. Semi-Joins and Anti-Joins 43
2.9.4. The `in` Predicate 44
2.9.5. Relational `in` 44
2.9.6. Inverted `In` 46
2.9.7. Sigma Predicates 47
2.10. Join 49

2.10.1. Cross Join 49
2.10.2. Inner Join 49
2.10.3. `USING` Shorthand 50
2.10.4. ON vs WHERE Inference 50
2.10.5. Multi-Table Joins 51
2.10.6. Self-Join 51
2.10.7. Argumentative Join 51
2.10.8. Outer Joins 52
2.10.9. Lateral Joins 52
2.10.10. ER-Context Joins 52
2.11. Domain Functions 53
2.11.1. Standard Function Invocation 54
2.11.2. Function Pipe Composition 54
2.11.3. Domain Operators 55
2.11.4. `Case` Simple 55
2.11.5. `Case` Search Function 56
2.11.6. Format Function and Strings 57
2.11.7. Citations and Symbols 58
2.11.8. Window/Analytic Functions 59
2.11.9. Scalar Subquery 61
2.11.10. Context-Aware Functions 62
2.11.11. Boolean Expressions as Columns 63
2.11.12. Compound Data Constructors 64
2.11.13. Compound Data Pathing 67
2.12. Addressing Columns by Index 68
2.12.1. Column Ranges 68
2.12.2. Reposition Operator 70
2.13. Common Expressions 72
2.13.1. Common Table Expressions 72
2.13.2. Common Function Expressions 73
2.14. Clause-local parameters 74
2.14.1. Recursive Common Table Expressions 74
2.14.2. Common Higher-order Table Expressions 75
2.14.3. Common Sigma Expressions 77
2.15. Tree Groups 77
2.15.1. Compound Data Constructors (Recap) 78
2.15.2. Aggregation boundaries and computed members 78
2.15.3. Tree Group Syntax 79
2.15.4. Data-Oriented Tree Grouping 79
2.15.5. Metadata-Oriented Tree Grouping 82
2.15.6. Tree Distinction 83
2.15.7. Tree Destructuring 83
2.15.8. Pathing in Tree Patterns 85
2.15.9. Interior Drill-Down 86
2.15.10. Narrowing 87
2.16. Set Operators 89
2.16.1. Union All Corresponding (`;`) 91
2.16.2. Smart Union All (`|;|`) 91
2.16.3. Positional (Traditional) Union All (`||`) 92

2.16.4. Set Semantics vs Multiset Semantics ... 92
2.16.5. Intersects via correlation ... 92
2.16.6. Minus (Except) ... 95
2.17. Interior Relations and Lateral Joins ... 96
2.17.1. Scalar subqueries ... 97
2.17.2. `EXISTS` and `NOT EXISTS` ... 97
2.17.3. Simple Shadowing ... 97
2.17.4. Correlated Table (Lateral Join) ... 98
2.17.5. Summary ... 99
2.18. Higher-Order Pipes ... 99
2.18.1. Piped vs. Direct Invocation ... 100
2.18.2. Multi-Parameter Piped Invocation ... 101
2.19. Pivot and Melt ... 101
2.19.1. Melt ... 101
2.19.2. Pivot ... 102
2.19.3. Pivot Syntax ... 104
2.20. Meta-ize Operator ... 104
2.20.1. Schema as Relation (^) ... 105
2.20.2. Postfix Form ... 105
2.20.3. Composability ... 105
3. Language Fundamentals ... 107
3.1. Unification and Logical Variables ... 107
3.1.1. How Names Are Introduced ... 107
3.1.2. Unification Creates Joins ... 108
3.1.3. Wildcard Access and Qualification ... 108
3.1.4. Qualified References in Argumentative Access ... 108
3.1.5. Literals and Constraints ... 109
3.1.6. Self-Unification ... 109
3.1.7. Anonymous Tables and Unification ... 109
3.1.8. Lvars as Data in Anonymous Tables ... 110
3.1.9. Summary of Unification Rules ... 111
3.2. Identifiers Mirror Call Sites ... 111
3.3. Null Semantics ... 112
3.3.1. Equality expressions ... 112
3.3.2. Unification equality ... 112
3.3.3. Traditional SQL Equals ... 112
3.4. Operational semantics of Query Expressions ... 112
3.5. Operational semantics of Directives ... 113
3.6. Entity Shadowing and Uniqueness ... 114
3.6.1. Contexts ... 115
3.6.2. Context Shadowing ... 116
3.6.3. Aliases name occurrences ... 117
3.6.4. Materialization Shadowing ... 118
3.6.5. Addressing and enlistment ... 119
3.7. Ambiguity in non-shadowing contexts ... 119
3.7.1. Binding follows lexical definition ... 120
3.7.2. Retracting a session definition ... 121
3.8. Default Transpilation Strategy and Resolution ... 121
3.8.1. Table Resolution and Transpilation ... 122

3.8.2. Higher-order Resolution and Transpilation ... 122
3.8.3. Function Resolution and Transpilation ... 122
3.8.4. Sigma Predicate Resolution and Transpilation ... 123
3.9. Variable and Column Scoping ... 123
3.9.1. Relational Continuation ... 123
3.9.2. Left-to-Right Evaluation ... 124
3.9.3. Interior relations and scope ... 126
3.10. Nominalization ... 127
3.10.1. Functor rules ... 127
3.10.2. Minting ... 127
3.10.3. The Post-Fix Operators ... 128
3.11. Moding and currying ... 128
3.11.1. Prolog syntax ... 129
3.11.2. Delightql's answer to moding ... 129
3.11.3. Functions Column Modality ... 130
3.11.4. Tables Column Modality ... 130
3.11.5. Sigma Predicates Column Modality ... 131
3.11.6. Function Facts Column Modality ... 131
3.11.7. Higher-Order Predicates Column Modality ... 132
3.12. Grounding Higher-order Literals ... 133
4. Data Definition Language (DDL) ... 135
4.1. Basics ... 135
4.1.1. Rules ... 135
4.1.2. Facts ... 136
4.2. The Two Necks ... 136
4.2.1. Extent vs Scope ... 136
4.2.2. Shadow Neck : ... 136
4.3. Rule Neck (`:-`) ... 137
4.4. Rules ... 137
4.4.1. Arity and Naming ... 137
4.5. Head Semantics ... 138
4.5.1. Two Head Forms ... 138
4.5.2. Ground Terms in the Head ... 139
4.5.3. Multi-Clause Rules ... 139
4.5.4. Disjunctive Rules: Union Semantics by head form ... 140
4.6. Higher-Order Rules ... 141
4.6.1. Definition Syntax ... 141
4.6.2. Direct Invocation ... 142
4.6.3. Piped Invocation ... 142
4.6.4. Parameters to Higher-order rules ... 143
4.7. Sigma Rules ... 149
4.8. ER-Context Rules ... 150
4.8.1. Defining Relationships ... 150
4.8.2. Multiple Contexts ... 150
4.8.3. Using Contexts ... 151
4.8.4. Direct Join (&) ... 151
4.8.5. Transitive Join (&&) ... 151
4.9. Recursion in Rules ... 152
4.9.1. Two Forms of Recursion ... 152

4.9.2. The Anatomy of Recursion ... 153
4.9.3. Evaluation Model ... 153
4.9.4. What Recursion Can Express ... 154
4.9.5. What Recursion Cannot Express ... 154
4.9.6. Termination ... 156
4.9.7. UNION Badge ... 157
4.9.8. Higher-Order Recursive Predicates ... 159
4.9.9. Example: Mandelbrot Set ... 160
4.9.10. Delightql Recursive Limits ... 160
4.10. Facts ... 161
4.10.1. Standard Facts ... 161
4.10.2. Stacked Facts ... 161
4.10.3. Default Implementation as Views ... 162
4.10.4. Sparse Stacked Facts ... 162
4.11. Citation Rules ... 163
4.12. Function Rules ... 163
4.12.1. Stacked Notation (Named Case) ... 163
4.12.2. Rule Form ... 164
4.12.3. Higher-Order Functions ... 166
4.12.4. Contextual Functions ... 167
4.13. Namespaces ... 168
4.13.1. What Namespaces Are ... 168
4.13.2. Namespace Types ... 168
4.13.3. Namespace Classification ... 169
4.13.4. Conventional Prefixes ... 170
4.13.5. Built-in Namespaces ... 170
4.13.6. main ... 170
4.13.7. home ... 170
4.13.8. Pseudo-predicates and “attaching” context ... 172
4.13.9. Grounding ... 173
4.13.10. Imprinting ... 174
4.14. In-line DDL ... 175
5. Data Manipulation Language (DML) ... 177
5.1. The !! Marker ... 177
5.2. Insert ... 178
5.3. Update ... 178
5.4. Delete ... 179
6. Effects and Directives ... 181
6.1. Receipts ... 181
6.2. Receipt Schema ... 181
6.3. Effect Rules ... 182
6.4. The Unwrap Pipe ... 183
6.5. ETL and scripting ... 183
6.6. Built-in Directives ... 184
6.6.1. The assert! directive ... 185
6.6.2. Consult ... 187
6.6.3. Mount ... 188
7. Settled, Known Limitation, Unspecified ... 191

7.1. Unspecified behaviour ... 191
7.2. Language Limitations ... 191
7.2.1. Higher-Order Moding Limitations ... 192
7.2.2. Non-deterministic functions ... 192
7.2.3. Recursion Limitations ... 197
7.2.4. Autogenerated Names ... 197
7.2.5. JSON-awareness and cataloging ... 197
7.2.6. Lateral Join Limitations ... 197
8. The Delightql Runtime ... 199
8.1. Namespace Contracts ... 199
8.1.1. Built-in Namespaces ... 199
8.1.2. Namespace metadata ... 200
8.1.3. Auto-enlist namespaces ... 200
8.1.4. Special Protected Namespaces ... 200
8.2. REPL vs CLI vs CORE ... 200
8.3. System Tables – A Tour ... 200
8.3.1. Information Schema Namespaces ... 201
8.3.2. Transpilation Rules Tables ... 201
8.4. Override Tables ... 201
9. Annotations ... 203
9.1. The Annotation Framework ... 203
9.2. Placement ... 203
9.3. Annotation Types ... 204
9.4. Error Assertions ... 204
9.4.1. URI Prefix Matching ... 205
9.4.2. Error URI Categories ... 205
9.4.3. Coexistence with Data Assertions ... 205
9.4.4. Scope ... 206
9.5. Danger Gates ... 206
9.5.1. Syntax ... 206
9.5.2. Scoping ... 206
9.5.3. Session Baseline ... 207
9.5.4. Danger URI Reference ... 207
9.6. Docs ... 207
9.6.1. Syntax ... 207
9.6.2. Applicability ... 208
9.6.3. Storage ... 209
10. Appendix: Tree Normal Form ... 211
10.0.1. The Graph ... 211
10.1. The Forms ... 212
10.1.1. TNF-0: Valid JSON ... 212
10.1.2. TNF-T: Well-Typed JSON ... 212
10.1.3. TNF-N: Namespaced ... 213
10.1.4. TNF-G: Grouped ... 213
10.1.5. TNF-M: Metadata-Keyed ... 214
10.1.6. TNF-SR: Simply Relational ... 214
10.1.7. TNF-R: Round-Trippable ... 214
10.1.8. TNF-GN: Grouped with Namespaced Leaves ... 215

10.2. Mixing Forms 215
10.3. Edge Types 216
10.4. Summary 216
10.5. Extending the Graph 217
10.5.1. Guiding Principles 217
Bibliography 219

1. INTRODUCTION



Delightql is a logic-inspired query language that transpiles to SQL. This document provides a reference to the language.

The breakdown for reference should seem familiar to anyone who knows SQL:

- Data Querying Language (DQL): Any feature that is used to return a result via querying.
- Data Definition Language (DDL): Any feature that is used to create or drop views or functions. These correspond to SQL DDL statements and to Prolog assertion-mode rules.
- Data Manipulation Language (DML): Any feature that is used to modify an existing set of data, i.e. `update`, `delete`, or `insert`.
- Fundamentals: How the precedence rules for parsing operators and expressions, both at the level of table expressions and of domain expressions, affect semantics. How LVars create unification.
- Namespace and Directives: The runtime environment for loading and scoping data and DDL.
- Effects and Directives: Directives – functors ending in `!` that write data or change the session – the receipts they return, and scripts of several statements.
- System: A listing of system tables available to introspect code, execution, and the state of the databases.

Throughout this reference, examples of SQL illustrate how a delightql expression could be transpiled. Such SQL examples should be viewed as both descriptive of the translation semantics and accurate as to the final results.

1.1. TERMINOLOGY

1.1.1. Logic and Sql Terminology

Delightql takes inspiration from the logic programming world as well as from SQL, so there will be many places in this reference where terminology from both may be used.

As a mini-tutorial for those unfamiliar, consider the following table of terminology equivalences:

SQL	logic programming
database	database
query	query
query	goal
query execution	unification and resolution
table	relation
table	predicate
table	relation/predicate of only ground terms (extensional)
view	predicate rule (intensional)
view with recursive CTE	predicate rule with multiple clauses and recursion
column	dimension
column	logic variable (LVar)
row	fact
tuple	fact
value	ground term
table definition	rule
view definition	rule
join	logical and
join	conjunction
exists (semi-join)	provable
not exists (anti-join)	not provable
union	logical or
union	disjunction
union	predicate rule with multiple clauses
insert row	assert a fact
delete row	retract a fact
update row	retract and assert fact

SQL	logic programming
number of columns	predicate arity
number of columns	predicate dimensionality
table-valued function	higher-order predicate

Table 1: SQL and logic programming terminology equivalences

This table is neither complete nor free of caveats.

The historic justification for this semi-equivalence is unassailable: Codd and Kowalski showed how relational algebra query languages and Prolog respectively could be centered on a predicate calculus – the first-order language of logic.

1.1.2. Delightql-specific Terminology

Some terms in this reference are used often enough to warrant the following few early paragraphs. These terms are

- domain
- functor
- predicate and relation and their distinction
- sigilization

1.1.2.1. Domain

The term domain is used throughout this book much in the same way it is used throughout mathematics, as the set of possible values for a given quantity. In SQL parlance, this is not to be confused with type, as the domain of heights should not be confused with the domain of Kelvin temperatures even though both could be modeled as positive real numbers.

Highlighting the term domain is done to create an emphasis on non-domain things. Delightql makes this distinction by asserting that things are either domain values or predicate values, either domain expressions or predicate expressions.

This is a somewhat artificial delineation as tables/predicates being cartesian products have their own very real input space and domain.

To be concrete:

- Domain values include `1`, `"roger"`, and `true`; domain expressions include `1+2` and `upper("roger")`.
- Predicate values include `employee`; predicate expressions include `generate_series(1,10)` or `(select i_am_a_subquery from inner) as newtable`.

1.1.2.2. Functor Form, or just Functor

A functor form is an identifier followed by a pair of parentheses and the arguments contained within. This definition is syntactic only, and has no assigned semantics until we establish a context. With apologies to Prolog, we shorten the term functor form to just functor.

Within the context of delightql a functor is usually understood to be a table and always understood to be a predicate (as predicates generalize tables).

```delightql
// Access the table Album and its three columns
// which we call a,b, and c
album(a,b,c)
```

```delightql
// Access the table Artist and all its columns
artist(*)
```

In the context of delightql and most logic languages, the functor notation is self-denoting. This means several things, but practically we can say the following:

- With functor notation, traditional notions of inputs and outputs are blurred, and often meaningless (each argument may be either input or output).
- A functor cannot be assigned to a (scalar) variable because it denotes itself as a relation, and not as a scalar (domain value).

In contrast, most other programming languages use functor syntax to denote a function and/or (perhaps) a subroutine.

```
foo(a,b,c)
```

implies

- if a subroutine: an execution/call only
- if a function: an execution/call and substitution by the domain value that it produces.

Under these languages, arguments are passed into a function (`a`, `b`, and `c`) and are consumed. Here, the function syntax is operationally equivalent to the value it returns, and thus can be assigned to domain variables.

That delightql, like Prolog, assigns the concept of a predicate/relation to the functor does not mean the absence of functions.

In Prolog, functions re-use the predicate functor syntax via the designation of a known argument (called a mode) to be the output. Prolog has additional nuances about functions which we will sidestep.

Delightql provides a special syntax that remains reminiscent of a relation: a functor with a colon `:` between the identifier and the pair of parentheses, which we will call a function functor.

delightql

```
employee(*)
  |> %(title ~> count:(*) as n,
                max:(length:(last_name)) as longest,
                min:(substr:(last_name, 1, 3)) as prefix)
```

The colon asks us to read `substr of last_name and 1 and 3` or `length of last_name`.

1.1.2.3. Predicate vs Relation vs Table

The terms predicate, relation, and table lack any universal authority, yet each carries emphasis depending on whom you're talking to. This section exists not for philosophy but because Prolog, SQL, and therefore delightql all take different positions at different times on whether these are separate concepts, and how strongly to emphasize the chosen mental model.

A predicate/relation/table can be viewed as:

- An arbitrary subset of a Cartesian product (the mathematical view)
- A named storage entity for data queries (the database view)
- A named evaluator of truth in logical statements (the logic view)
- A generalization of functions (the relational view delightql emphasizes)

SQL treats relation and table as interchangeable, embracing the first two definitions. It reserves predicate for built-in truth evaluators over traditional domains like numbers and strings – as in `where age>20 and last_name like "%son"`, where two predicates compose into one.

Prolog favors predicate, with occasional references to relation and rare references to table. But it maps its term to all three definitions above. A Prolog predicate can be stored data, a logical assertion, or a computable relationship.

Delightql uses these terms interchangeably, with some deliberate emphasis. It borrows from Prolog in favoring predicate for tables (extensional data), views (intensional data derived from extensional data), and built-in domain predicates like `=`, `<`, `>`, and `like`.

Like Prolog, delightql recognizes an operational difference between predicates resolvable from ground truths (finitary – backed by actual stored tuples) and those expressing relations over infinities. To wit, there is no table enumerating all pairs where one number is less than another. Prolog restricts such non-Herbrand expressions via moding. Delightql similarly won't let you write the equivalent of `select left_hand from ">" where right_hand=3`.

There is one more definition worth surfacing:

- A predicate is a function of truth – given a tuple, it returns membership (true/false)

This is just a restatement of "named evaluator of truth," but the framing emphasizes the boolean nature of predicates. In such situations, we are asking not what data the predicate contains, but whether a given tuple belongs. Delightql utilizes this framing for what SQL calls semi-joins (`EXISTS` queries) and what Prolog expresses through provability contexts like negation-as-failure (`\+`). In these cases, we don't care what the predicate returns – only whether it succeeds or not.

1.1.2.4. Sigilization

Delightql has few keywords compared to SQL.

Throughout this reference, the term sigilization describes delightql's practice of representing operators with non-alphanumeric symbols rather than keywords.

For example, delightql sigilizes the DISTINCT relational operator with a `%` symbol followed by parentheses:

```delightql
invoice(*) |> %(billing_country)
// SQL: SELECT DISTINCT billing_country FROM Invoice
```

To continue this example, delightql recognizes that `GROUP BY` is simply `DISTINCT` extended with aggregation. The same `%` sigil handles both – separate grouping columns from aggregate functions with `~>` to get the equivalent of SQL's `GROUP BY`:

```delightql
invoice(*) |> %(billing_country ~> count:(*), sum:(total) as
total_by_country)
// SQL: SELECT billing_country, count(*), sum(Total) as total_by_country FROM
Invoice GROUP BY billing_country
```

2. DATA QUERY LANGUAGE (DQL)

The heart of both delightql and SQL is the query expression, also called a table expression. A query expression is a unit of code that returns exactly one table. Here, table is synonymous with predicate or relation, regardless of whether the data is persisted via CREATE TABLE. [1]

The majority of delightql lives in this section; mastering it is prerequisite to understanding DDL and DML.

A query's meaning is identical to the table it produces. This substitutability is key to composability: through subqueries and CTEs, query expressions become recursively inductive to any depth. Delightql encourages a particular style of composition – pipelining a relation through transformations, left to right, with consistent associativity and scoping.

1 More typically, these results are anonymous and ephemeral – the output of execution within the REPL.

2.1. BASICS: COMMENTS, STRINGS, AND SEMICOLONS

Unlike SQL, delightql uses the double forward slash `//` for comments. The double hyphen `--` is not a valid comment in delightql and will fail to parse.

delightql

```
// I am a valid comment
_(one@1)
```

Delightql only uses double quotes `"` to delimit a string in code. The single quote is not an optional syntax for strings. The backtick is used for stropping identifiers.

delightql

```
`employee`(*), last_name = "King"
```

The semicolon `;` is so pervasive in other programming languages that this reference must make this the first syntactic clarification: semicolons in delightql are NOT statement separators. Instead they are the special operator UNION CORRESPONDING and are covered in the section on set operators.

delightql

```
employee_2024(*) as x ;
  employee_2025(*) as y
// the above UNIONs the two tables
// with null padding for unshared
// columns
```

2.2. ALL RESERVED WORDS

Delightql has very few keywords. They are

- `as`
- `not`
- `and`
- `or`
- `in`
- `true`
- `false`
- `null`
- `of` – only for pivots
- `groups` – only for window functions
- `rows` – only for window functions
- `has` – only for tree group destructuring
- `asc` and `ascending` – only in order by
- `desc` and `descending` – only in order by

2.3. CASE INSENSITIVITY

Delightql is case-insensitive. [2] The following all refer to the same identifier:

- `employee_id`
- `Employee_Id`
- `EMPLOYEE_ID`

2 In contrast to Prolog, where capitalization distinguishes variables from atoms.

2.3.1. Stropping

Names containing characters outside the bare identifier shape (spaces, for instance) are delimited with backticks: `` `employee Id` ``. Keyword spellings do not require stropping in identifier positions: `from(*)` may name a table, and `as:(x)` may name a function. Keywords still perform their grammatical roles where an operator or literal is expected.

2.4. FROM

Every delightql query begins with a source: one or more tables from which to draw data. SQL calls this selection [3], but we will call this access or sourcing.

3 In contrast to Codd's original paper, where selection (σ) denoted row filtering–what SQL now calls WHERE.

2.4.1. All Columns

delightql

```
employee(*)
```

*

The glob `*` in argument position requests all columns from a table – equivalent to `SELECT * FROM employee`. This is the most common way to source data in delightql.

2.4.2. Argumentative Access

Arguments within the functor bring columns into scope by position.

delightql

```
employee(employee_id  , last_name ,
         first_name   , title     , reports_to,
         birth_date   , hire_date , address, city,
         state        , country   , postal_code,
         phone        , fax       , email )
```

This binds each identifier to the column at that ordinal position in the table definition. The arity must match exactly – if `employee` has 15 columns, the functor must have 15 arguments.

The identifiers you choose become the column names in scope of the position having the identifier.

delightql

```
employee(a, b,  c,  d,
         e,   f, g,   h, i,
         j,  k,   l, m,   n,  o )
```

This makes argumentative access error-prone for wide tables. When arity exceeds a handful of columns, prefer the glob `employee(*)` and use the columns by name (see All Columns).

2.4.3. Aliasing a Table

delightql

```
employee(*) as e
```

The `as` keyword assigns an alias to a table. Once aliased, columns must be accessed through the alias – the original table name leaves scope.

SQL

```
select * from employee as e;
```

Aliases are often a convenience, but become necessary in contexts like self-joins.

2.4.4. Table Namespacing

```delightql
mount!("etl.sqlite", "etl")(*)
etl.partner_sale_2025_07_31(*) as s
```

A dot-prefixed identifier namespaces the table. Here, `partner_sale_2025_07_31` lives within the namespace `etl`. What this namespace represents – schema in some databases, database in others – is implementation-dependent.

```sql
select * from etl.partner_sale_2025_07_31 as s;
```

The namespace is the entire syntax before the dot and may include nesting using `::`. Namespaces are nested like file-system folders.

```delightql
mount!("etl.sqlite", "client1::production::etl")(*)
client1::production::etl.partner_sale_2025_07_31(*) as s
```

In the above example, `client1::production::etl` is the namespace where `client1` contains `production` which contains `etl`.

Namespaces are elements of the delightql runtime. The delightql programmer chooses the hierarchy and maps these to source structures. For more information, see the namespacing section of DDL.

2.4.5. Anonymous Table

```delightql
_(1,2,3)
```

The underscore functor `_( )` declares a table inline, with literal values. Commas separate columns. This is equivalent to:

```sql
select 1,2,3;

select * from (values (1,2,3));
```

The underscore is the `FULL` sigil – a name for "no name." [4]

Note the two anonymous tables in play: the inline table declared with `_( )` and the result table returned by the query itself. The term anonymous table in this reference denotes the former – tables whose values are defined inline and whose names are discarded. [5]

4 The name `FULL` plays against `NULL`: where `NULL` matches nothing and has null potency, `FULL` matches everything and is full of potential.

5 An anonymous table is a query-mode construct, despite its syntactic resemblance to assertion-mode fact instantiation.

2.4.5.1. Stacked Notation and Multiple Rows

Multiple rows are expressed with the SEMI-OR sigil `;`, a disjunction operator:

```delightql
_(1,2;10,20)
```

SQL

```
select 1,2
  UNION ALL
select 10,20;
```

Semicolon binds looser than comma, so rows stack naturally without parentheses. This stacked notation appears throughout delightql – anywhere multiple clauses or rows would otherwise require repeating functor syntax. [6]

[6] Stacked notation is essentially syntactic sugar: it reduces redundant functor notation in both assertion mode and anonymous tables. See the glossary for examples.

2.4.5.2. Naming the Columns of Anonymous Tables

The columns in the examples so far have been positional – they have ordinal positions but no names. Delightql allows positional-only access, but also provides syntax for naming columns.

delightql

```
  _( first, second @ 1,2;10,20;100,200)
```

SQL

```
select 1 as first ,2 as second
  UNION ALL
select 10,20
  UNION ALL
select 100,200;
```

The ATOP sigil `@` separates column names (comma-delimited) from the stacked data that follows. This three-row, two-column table now has columns named **`first`** and **`second`**.

ATOP has an alternate form: three or more dashes. The following are all equivalent:

- `@`
- `---`
- `--------`
- `----------------------`

The dashed form enables formatted table literals:

delightql

```
_(
 id,first_name    , last_name , email
 ------------------------------------------------------
 0,"Gusti"        , "Parlor"  , "gparlor0@phoca.cz"  ;
 1,"Diane-marie" , "McHenry" , "dmchenry1@dot.gov"  ;
 2,"Ced"          , "Mainds"  , "cmainds2@goo.ne.jp" ;
 3,"Bren"         , "Berndsen", "bberndsen3@gr.com"
)
```

2.4.5.3. Sparse Anonymous Tables

When most columns in a wide anonymous table are NULL, data rows become verbose and error-prone. Sparse columns solve this: mark optional columns with **`?`** in the header, then fill only the ones you need per row.

```delightql
_(column, type, nullable?, default?, check?, primary_key?,
unique?

-----------------------------------------------------------------
  "id",      "INT",     _(primary_key @ "true") ;
  "name",    "TEXT" ;
  "email",   "TEXT",    _(unique @ "true") ;
  "salary", "DECIMAL",_(check @ "salary>0"))
```

Columns without `?` are positional – every row must supply them, in order. Columns with `?` are sparse – unfilled sparse columns default to NULL. If sparse columns are used, they must come after the required columns.

2.4.5.3.1. Sparse fills

A sparse fill uses anonymous table syntax `_(col @ val)` to assign a value to a named sparse column. Fills appear after the positional values in a data row:

```delightql
_(column, type, nullable?, primary_key?
  ------------------------------------
  // Single fill
  "id", "INT", _(primary_key @ "true") ;

  // Multiple separate fills
  "id", "INT", _(primary_key @ "true"), _(nullable @ "false") ;

  // Combined fill: multiple sparse columns in one expression
  "id", "INT", _(primary_key, nullable @ "true", "false"))
```

In a combined fill, column names and values are matched positionally: `primary_key` gets `"true"`, `nullable` gets `"false"`.

2.4.5.3.2. No fills

When a row supplies no fills, all sparse columns are NULL:

```delightql
_(a, b?, c?
  --------
  1 ;
  2 ;
  3)
```

This is equivalent to `_(a, b, c @ 1, null, null; 2, null, null; 3, null, null)`.

2.4.5.3.3. All sparse

A table may have no positional columns at all:

delightql

```
_(x?, y?
  ------
  _(x @ 1) ;
  _(y @ 2) ;
  _(x, y @ 3, 4))
```

2.4.6. Higher-Order Predicates

Delightql supports higher-order predicates–predicates that accept tables (or scalars) as parameters and return a table. SQL calls these table-valued functions. [7] Given this definition:

[7] Defining higher-order rules is covered in the DDL section.

delightql

```
clean_contacts(T(*))( * ) :-
  T(*)
    |> $(trim:())( last_name, first_name)
    |> $(lower:())( email)
    |> -( fax)
```

a query passes it a table:

delightql

```
clean_contacts(employee(*))(*)
```

Here, `clean_contacts` is a higher-order predicate that takes `employee(*)` as its parameter.

The transpiled SQL depends on how the predicate was defined. With the definition given, the query `clean_contacts(employee(*))(*)` produces:

SQL

```
select
    employee_id,
    trim(last_name)    as last_name,
    trim(first_name)   as first_name,
    title,
    reports_to,
    birth_date,
    hire_date,
    address,
    city,
    state,
    country,
    postal_code,
    phone,
    lower(email)       as email
from employee;
```

Higher-order predicates are structurally typed by the columns they reference. [8] In this example, any table with `last_name`, `first_name`, `email`, and `fax` qualifies:

[8] This resembles duck typing: delightql has no formal type layer, but detects which columns the predicate body requires. Any table providing those columns is a valid argument.

delightql

```
clean_contacts(customer(*))(*)
```

The pipeline form, explained in Higher-Order Pipes, is equivalent:

delightql

```
customer(*)
    |> clean_contacts(*)
```

2.5. ORDER BY

Delightql, like SQL, has order by. The ORDER-BY operator is an octothorpe-prefixed functor `#()` applied after a pipe:

delightql

```
employee(*)
    |>  #(first_name,last_name)
```

SQL

```
select
  *
from employee order by first_name, last_name;
```

Columns appear in the SQL in the order given. Collation modifiers work as in SQL–with keywords:

delightql

```
employee(*)
    |>  #(hire_date descending,last_name ascending)
```

SQL

```
select
  *
from employee order by hire_date desc, last_name asc;
```

> `Order By` has no meaning in pure relational algebra, where relations are unordered sets. SQL has never been true to theory; delightql is equally cavalier. To admit order-by as a first-class relational operator, delightql takes two positions:
>
> - Relations are ordered sequences of tuples, not sets. [9]
> - A parametric mechanism maps domain orderings onto tuple orderings. [10]

9 A **sequence** captures this notion.

10 We can imagine every tuple contains a hidden column `#`. Absent an explicit ordering, this column holds arbitrary values. Given `order by my_column`, the tuples reorder and `#` recalculates accordingly.

2.6. LIMIT

Limit the number of tuples returned using the TUPLE-ORDINAL sigil `#` in a predicate position:

delightql

```
customer(*) , # < 20
```

SQL

```sql
select * from customer limit 20;
```

Read this as: “all columns of customer where the implicit row ordinal is less than 20.”

Limit affects only cardinality, not schema.

Order of operations matters. Delightql evaluates left to right, so these two queries differ:

delightql

```
customer(*), invoice(*.(customer_id)), #<20
```

SQL

```sql
select
  *
from customer join invoice using(customer_id)
  limit 20;
```

delightql

```
customer(*), #<20, invoice(*.(customer_id))
```

SQL

```sql
select
  *
from (select * from customer limit 20)
  join invoice using(customer_id);
```

2.7. PROJECTION

Projection retains a subset of columns from a relation. In SQL, this is the list between `SELECT` and `FROM`.

2.7.1. Project Retain

delightql

```
employee(*)
  |>  (first_name , last_name)
```

SQL

```sql
select
  first_name,
  last_name
from employee;
```

The R-PIPE `|>` passes a relation to the PROJECT operator `( )`. Columns listed inside are retained; all others are discarded.

The pipe creates a scope barrier for the relation being projected: only its projected columns continue forward. Inside an interior relation, bindings from the enclosing scope remain accessible; they are not part of the local interface this projection replaces.

Projections can be chained:

delightql

```
employee(*)
  |>  (first_name , last_name)
  |>  (first_name )
```

SQL

```
select first_name from employee;
-- -- optimized from:
-- select first_name
--   from (
--     select
--       first_name,
--       last_name
--     from Employee);
```

Delightql (and SQL optimizers) will simplify redundant intermediate projections. But scope is enforced at each step–this will not work:

delightql

```
// Error: first_name not in scope
employee(*)
  |>  (last_name)
  |>  (first_name)
```

After line 2, only last_name exists in the piped relation.

2.7.2. Rename

Rename a column during projection with **as**:

delightql

```
employee(*)
  |>  (first_name as f, last_name)
```

SQL

```
select
  first_name as f,
  last_name
from employee;
```

2.7.3. Generalized Projection

Columns can be transformed during projection using domain functions:

delightql

```
track(*)
    |>  ( upper:(name) as n,
          upper:(composer) as composer,
          milliseconds / 1000 as seconds)
```

SQL

```
select
  upper(name) as n,
  upper(composer) as composer,
  milliseconds / 1000 as seconds
from track;
```

Note the colon in `upper:(name).` This distinguishes functions from relations – `foo(A,B)` is a relation; `foo:(A)` is a function.

> Aggregate functions are not permitted in projection. See the sections on `distinct` and `group by` for aggregate usage. Delightql will reject known aggregates, but cannot detect user-defined aggregates–these will transpile as if scalar.

Other functions – 'case', 'case select', concatenation, windowing/analytic functions, and operators – are covered in the function chapter of this reference.

2.7.4. Argumentative Projection

Columns can be projected by position using argumentative access. The FULL sigil `_` discards unwanted positions:

delightql

```
employee(employee_id , last_name , _ , _ ,
         _ , _ , _ , _ , _ ,
         _ , _ , _ , _ , _ , _ )
```

SQL

```
select
  employee_id,
  last_name
from employee;
```

Only the first two columns are retained; the rest are projected away. The identifiers `employee_id` and `last_name` name the columns in the result – matching the underlying column names here, though positional access can also rename by using a different identifier.

As with argumentative positional access, this notation is brittle for wide tables – prefer named projection when arity exceeds a handful of columns (see Argumentative Positional Access).

2.7.5. Project Out

-(◌)

The PROJECT-OUT operator subtracts columns from a relation:

delightql

```
employee(*)
  |> -(birth_date, email)
```

SQL

```
select
    employee_id,
    last_name,
    first_name,
    title,
    reports_to,
    -- birth_date, -- column projected out
    hire_date,
    address,
    city,
    state,
    country,
    postal_code,
    phone,
    fax
    -- Email -- column projected out
from employee;
```

All columns except `birth_date` and `email` are retained. This is particularly useful for wide tables where listing retained columns would be tedious.

2.7.6. Map Covering

The MAP-COVER operator `$( · )( · )` applies a function across specified columns while preserving all others:

delightql

```
employee(*)
  |> $(upper:())( last_name, first_name, title, city)
```

SQL

```
select
    employee_id,
    upper(last_name)  as  last_name,
    upper(first_name) as  first_name,
    upper(title)      as  title,
    reports_to,
    birth_date,
    hire_date,
    address,
    upper(city)       as  city,
    state,
    country,
    postal_code,
    phone,
    fax,
    email
from employee;
```

The first parentheses contain the function; the second lists the target columns. The function `upper:()` is written with its argument row open – the column value lands in the row's final argument, the same landing a function pipe takes. For a unary function the final argument is the only one; for `max:(0)` the column value lands after the written `0`, as `max(0, value)`. Write `@` where the value must land elsewhere.

Map covering:

1. Applies the function to each listed column
2. Renames results to their original column names
3. Passes through unlisted columns unchanged
4. Preserves column ordinality

Because unlisted columns pass through, transformations can be chained:

delightql

```
employee(*)
  |> $(upper:())( last_name, first_name, title, city)
  |> $(date:())( birth_date, hire_date)
```

Composing functions. When multiple functions apply to the same columns, three options exist:

Chained covers (repetitive but clear):

delightql

```
employee(*)
  |>  $(upper:())(first_name,last_name)
  |>  $(trim:())(first_name,last_name)
```

Containment composition using F-PARAM `@` as a placeholder:

delightql

```
employee(*)
  |>  $(trim:(upper:(@)) )(first_name,last_name)
```

Both produce:

SQL

```
select
    employee_id,
    trim(upper(last_name))  as  last_name,
    trim(upper(first_name)) as  first_name,
    title,
    reports_to,
    birth_date,
    hire_date,
    address,
    city,
    state,
    country,
    postal_code,
    phone,
    fax,
    email
from employee;
```

2.7.7. Basic Dimension Covering

The BASIC-COVER operator `$$(  )` transforms individual columns without the curried function syntax:

delightql

```
employee(*)
  |> $$( "--------" as phone, upper:(state) as state)
```

SQL

```sql
select
    employee_id,
    last_name,
    first_name,
    title,
    reports_to,
    birth_date,
    hire_date,
    address,
    city,
    upper(state) as state,
    country,
    postal_code,
    '--------' as Phone,
    fax,
    email
from employee;
```

Each transformed column requires an `as` modifier – this identifies which columns are being replaced. Unlisted columns pass through in their original ordinality. Referencing a nonexistent column is an error.

2.7.8. Conditional Covering (If-Only)

The IF-ONLY sigil `|` constrains which rows a cover applies to. Rows not matching the predicate pass through unchanged.

Map-cover with if-only:

delightql

```
employee(*)
  |> $(upper:())(last_name, first_name | title = "IT Staff")
```

SQL

```sql
SELECT
  employee_id,
  CASE
    WHEN title = 'IT Staff' THEN upper(
      last_name
    )
    ELSE last_name
  END AS last_name,
  CASE
    WHEN title = 'IT Staff' THEN upper(
      first_name
    )
    ELSE first_name
  END AS first_name,
  title,
  reports_to,
  birth_date,
  hire_date,
  address,
  city,
  state,
  country,
  postal_code,
  phone,
  fax,
  email
FROM employee;
```

The predicate follows the column list, separated by `|`. This mirrors the aggregate if-only `count:(col | pred)` – the `|` always sits between the operands and the condition.

Without if-only, the function applies to all rows. With if-only, the function applies only to matching rows; non-matching rows retain their original values.

Basic-cover with if-only:

delightql

```
employee(*)
  |> $$("REDACTED" as phone, "---" as fax | title = "IT Staff")
```

SQL

```
select
    employee_id,
    last_name,
    first_name,
    title,
    reports_to,
    birth_date,
    hire_date,
    address,
    city,
    state,
    country,
    postal_code,
    case when title = 'IT Staff'
         then 'REDACTED' else phone end as phone,
    case when title = 'IT Staff'
         then '---' else Fax end        as Fax,
    email
from employee;
```

The predicate goes at the end of the item list, after the last `as` target.

Composability. If-only composes with callable nesting and chaining:

delightql

```
customer(*)
  |> $(trim:(upper:(@)))(first_name, last_name | country =
"USA")
```

If-only is syntactic sugar over CASE expressions. The equivalent without if-only:

delightql

```
employee(*)
  |> $$( _:(title = "IT Staff" -> upper:(last_name);  _ ->
last_name)  as last_name,
         _:(title = "IT Staff" -> upper:(first_name); _ ->
first_name) as first_name)
```

2.7.9. Regular Expression Column Addressing

A regular expression can select columns by name pattern:

delightql

```
employee(*)
  |> ( /date/ )
```

SQL

```
select
    birth_date,    -- matches /date/
    hire_date      -- matches /date/
from employee;
```

The pattern matches anywhere in a column's name, and it ignores case, as a column reference does: `(Birth_Date)` and `/BIRTH/` both find `birth_date`. The regex applies only to column names, not namespaces or indexes.

To match case exactly, add the flag `c` after the closing slash:

delightql

```
employee(*)
  |> ( /date/c )
```

Every column in the example database is lowercase, so `/date/c` still finds `birth_date` and `hire_date`, while `/Date/c` finds nothing and refuses.

`c` is the only flag. `/date/i` refuses, because ignoring case is already the default. Ignoring case applies inside character classes too, so `/^[A-Z]/` matches `birth_date`, while `/^[A-Z]/c` matches nothing here.

The pattern language is a small subset of POSIX basic regular expressions: `^` and `$` anchor, `.` matches any character, `*` repeats, `[...]` is a character class, and `\+` and `\?` repeat one-or-more and zero-or-one. There are no groups and no alternation.

Positions. A regex stands wherever a list of columns is enumerated: PROJECT-IN `( )`, PROJECT-OUT `-( )`, the second parentheses of MAP-COVER, the source of a RENAME-COVER `*(/date/ as :"d_{@}")`, GROUP-MODULO keys, function arguments, and record members. A project-in that matches no column refuses; a project-out may match none.

It cannot:

- Be followed by `as`; rename the matched columns with a name template
- Stand as a single value: an operand, a function-pipe step, or a slot
- Add columns in EMBED `+( )`, since it names only columns that already exist

2.7.10. Rename Cover

The RENAME-COVER operator `*( )` renames specified columns while passing all others through:

delightql

```
track(*)
  |> *( name as track_name)
```

SQL

```
select
    track_id,
    name as track_name,
    album_id,
    media_type_id,
    genre_id,
    composer,
    milliseconds,
    bytes,
    unit_price
from track;
```

Rename cover preserves column count and column ordinality – only the names change.

2.7.11. Embedding

The EMBED operator `+( )` adds a new column to a relation, placed after existing columns:

delightql

```
employee(*)
  |> +(  strftime:("%Y",birth_date) - 2 as
two_years_before_birth )
```

SQL

```
select
    *,
    strftime('%Y',birth_date) - 2 as two_years_before_birth
from employee;
```

This is equivalent to `|> (*, expr as name)`. The embed syntax makes the intent explicit: the relation is unchanged except for the added column.

2.7.12. Map Embedding

The EMBED-MAP operator `+$( )( )` applies a function across columns and creates new columns from the results (rather than replacing the originals):

delightql

```
tidy:(s) :- trim:(coalesce:(s, ""))
label:(s) :- upper:(tidy:(s))
```

delightql

```
invoice(*)
  |> +$(tidy:() as :"tidy_{@}")( /^billing_/ )
  |> +$(label:() as :"label_{@}")( /^billing_/ )
```

```sql
SELECT
  invoice_id,
  customer_id,
  invoice_date,
  billing_address,
  billing_city,
  billing_state,
  billing_country,
  billing_postal_code,
  total,
  trim(coalesce(billing_address, '')) AS tidy_billing_address,
  trim(coalesce(billing_city, '')) AS tidy_billing_city,
  trim(coalesce(billing_state, '')) AS tidy_billing_state,
  trim(coalesce(billing_country, '')) AS tidy_billing_country,
  trim(coalesce(billing_postal_code, '')) AS
tidy_billing_postal_code,
  upper(trim(coalesce(billing_address, ''))) AS
label_billing_address,
  upper(trim(coalesce(billing_city, ''))) AS label_billing_city,
  upper(trim(coalesce(billing_state, ''))) AS
label_billing_state,
  upper(trim(coalesce(billing_country, ''))) AS
label_billing_country,
  upper(trim(coalesce(billing_postal_code, ''))) AS
label_billing_postal_code
FROM invoice;
```

The first parentheses contain the function and an as qualifier with an F-STRING. The F-PARAM sigil `@` stands in for the column name, generating `tidy_billing_address`, `tidy_billing_city`, etc. The second parentheses specify the target columns–here, all columns matching `/^billing_/`.

Unlike MAP-COVER, which replaces columns in place, EMBED-MAP preserves the originals and appends the transformed columns.

2.7.13. Projection Summary

The operators introduced so far are all unary – they transform a single relation. The following qualities characterize their behavior:

- Column Cardinality Preserved: The number of columns remains unchanged.
- Column Ordinality Preserved: Columns retain their relative order, with no intercalation of new columns among existing ones.
- Column Names Preserved: Existing columns keep their names (new columns do not affect this).
- Table Cardinality Preserved: The number of rows remains unchanged.
- Table Ordinality Preserved: Rows retain their order.

A quality is marked Y only if it holds under all circumstances; N if any case violates it.

	`( )`	`-( )`	`$( )( )`	`$$( )`
	Project	Project Out	Map Cover	Basic Map Cover
Column Ordinality Preserved	N	Y	Y	Y
Column Cardinality Preserved	N	N	Y	Y
Column Names Preserved	N	Y	Y	Y
Table Cardinality Preserved	Y	Y	Y	Y
Table Ordinality Preserved	Y	Y	Y	Y

Table 2: Preservation properties of Project, ProjectOut, MapCover, and BasicMapCover

	`*( )`	`#( )`	`%( )`
	Rename	Order By	Group Modulo
Column Ordinality Preserved	Y	Y	N
Column Cardinality Preserved	Y	Y	N
Column Names Preserved	N	Y	N
Table Cardinality Preserved	Y	Y	N
Table Ordinality Preserved	Y	N	N*

Table 3: Preservation properties of Rename, OrderBy, and GroupModulo

	`+( )`	`+$( )`
	Embed	Map Embed
Column Ordinality Preserved	Y	Y
Column Cardinality Preserved	N	N
Column Names Preserved	Y	Y
Table Cardinality Preserved	Y	Y
Table Ordinality Preserved	Y	Y

Table 4: Preservation properties of Embed and MapEmbed

One operator stands apart: GROUP-MODULO `%( )` preserves none of these qualities unconditionally. It captures both **`DISTINCT`** and **`GROUP BY`**, and is the subject of the next chapter.

2.8. DISTINCT AND GROUP BY

Distinct and group by are congruent operations. Their similarity is often a source of confusion:

SQL

```
select
  title
from employee
  group by title;

-- Produces the same result as:

select
  distinct title
from employee;
```

Delightql reflects this congruence with unified syntax.

2.8.1. Distinct

The GROUP-MODULO operator `%(  )` returns distinct combinations of the specified columns:

delightql

```
customer(*)
  |> %(country)
```

SQL

```
select
  distinct country
from customer;
```

Multiple columns return distinct combinations:

delightql

```
customer(*)
  |> %(country, state)
  |> #(country,state descending)
```

SQL

```
select
  distinct country, state
from customer
    order by country asc, state desc;
```

To deduplicate all columns – converting a multiset (bag) into a set:

delightql

```
genre_2025(*)
  |> %(*)   //returns unique rows and removes duplicates
```

2.8.2. Group By

`Group by` extends distinct with aggregation. The AGG-AND sigil `~>` separates grouping columns (left) from reduced columns (right):

delightql

```
invoice(*)
  |> %(billing_country ~>  count:(*) , sum:(total) )
```

SQL

```
select
  billing_country,   -- grouping column
  count(*),          -- reduced column
  sum(total)         -- reduced column
from invoice
  group by billing_country;
```

Grouping columns may be expressions:

delightql

```
invoice(*)
    |> %( total > 10  as high_low,
          upper:(billing_country) ~>
            count:(*) ,
            avg:(total) )
```

SQL

```
select
  total > 10 as high_low,  -- grouping column
  upper(billing_country),   -- grouping column
  count(*),     -- reduced column
  avg(total)    -- reduced column
from invoice
  group by upper(billing_country), (total > 10) ;
```

2.8.3. Group Delegates

A group delegate is a principled mechanism for choosing one value among many during group reduction. The group delegate generalizes the aggregate functions `max()` and `min()` by allowing the user to choose their own ordering criteria. The group delegate is a function in unfamiliar clothing.

The group delegate may be found in reduction place. A column parenthesisized and then postfixed by the F-OVER sigil `<~` followed by an explicit ordering criteria.

delightql

```
track(*)
  |> %(genre_id ~> count:(*) as n, (name) <~ #(milliseconds
desc))
```

With an empty ordering (bare `<~`) the delegate row is arbitrary:

delightql

```
track(*)
  |> %(genre_id
        ~> count:(*) as n,
        (name as longest) <~ #(milliseconds desc),
        (name as arbitrary)<~ )
```

Some Sql vendors implement this functionality directly with dedicated syntax – the `DISTINCT ON` syntax.

2.8.4. Whole-Group Aggregate Functions

To aggregate without grouping, omit the grouping columns:

delightql
```
invoice(*)
  |>  %( ~>  count:(*) , sum:(total) )
```

SQL
```
select
  count(*),
  sum(total)
from invoice;
```

The GROUP-PIPE `~>` provides a shorter form for a single aggregate:

delightql
```
invoice(*) ~>  count:(*)
```

Note. The GROUP-PIPE is different from the AGG-AND. This form replaces the `|>` pipe. It is pure sugar.

2.8.5. Internal Distinct

Some aggregates accept a distinct modifier on their input. The INNER-MODULO sigil `%` prefixes the column:

delightql
```
customer(*)
    |>  %( country ~>
           count:(%city) ,
           count:(%support_rep_id))
```

SQL
```
select
  country,
  count(distinct city),
  count(distinct support_rep_id)
from customer
  group by country;
```

2.8.6. Filter Clauses

The IF-ONLY sigil `|` constrains which values enter an aggregate:

delightql
```
customer(*)
  |>  %( country ~>
         count:(%city) ,
         count:(%support_rep_id),
         count:(last_name | length:(last_name) > 7)
            as long_lastname_count)
```

For dialects supporting `FILTER`:

SQL

```
select
  country,
  count(distinct city),
  count(distinct support_rep_id),
  count(last_name)
    filter
      (where length(last_name) > 7) as long_lastname_count
from customer
  group by country;
```

For dialects without `FILTER`, delightql emits a `CASE` expression:

SQL

```
select
  country,
  count(distinct city),
  count(distinct support_rep_id),
  count(case when length(last_name) > 7
             then last_name else null end) as long_lastname_count
from customer
  group by country;
```

2.8.7. Having

Filter on reduced columns by placing a predicate after the `group by`:

delightql

```
customer(*)
  |> %( country ~> count:(*) as customer_count)
      ,  customer_count > 4
```

Read this as: "group customers by country, count each group, then keep only groups with more than 4 rows."

SQL

```
select
  country,
  count(*) as customer_count
from customer
  group by country
    having count(*) > 4;
```

> Why does SQL have both WHERE and HAVING?
>
> SQL has an implicit order of operations. `WHERE` filters rows before grouping; `HAVING` filters groups after aggregation. The two keywords signal this distinction. [11]
>
> The abstraction is leaky – most programmers soon recognize that `HAVING` is equivalent to wrapping in a subquery and filtering with `WHERE`:

11 For a historical reflection on this issue, see [Dar24,].

SQL

```sql
SELECT country, count(*) AS customer_count
FROM customer
GROUP BY country
HAVING count(*) > 4;


-- equivalent to:

SELECT * FROM (
SELECT country, count(*) AS customer_count
FROM customer
GROUP BY country
) WHERE customer_count > 4;
```

Because delightql has explicit order of operations, no separate syntax is needed. The predicate simply follows the group by:

delightql

```
customer(*)
|> %(country ~> count:(*) as customer_count),
customer_count > 4
```

Placing the filter earlier would be an error, `customer_count` does not exist until after the aggregation.

2.9. WHERE

Selection (σ in Codd's relational algebra, `WHERE` in SQL) filters rows without changing schema. [12]

Delightql uses the comma (conjunction) to attach predicates to relations. This is the same syntax as joins and comes directly from Prolog.

12 SQL's use of "select" for projection is unfortunate – Codd used "select" for row filtering. Delightql follows Codd's terminology.

2.9.1. Domain Predicates

delightql

```
invoice(*), total > 10
```

SQL

```sql
select * from invoice where total > 10;
```

Multiple predicates conjoin naturally:

delightql

```
invoice(*), total > 10,
  trim:(lower:(billing_country))="usa"
```

SQL

```
select * from invoice
  where total > 10
    and trim(lower(billing_country))
      IS NOT DISTINCT FROM 'usa';
```

Scope restricts commutativity. Predicates can only reference columns already in scope. This is invalid:

delightql

```
// WONT WORK because total is not yet in scope
  customer(*),
    total > 10,
    trim:(lower:(billing_country))="usa",
    invoice(*)
```

But once columns are in scope, predicates may be reordered:

delightql

```
invoice(*),
  total > 10,
  trim:(lower:(billing_country))="usa"

// commutativity allowed when all LVars are in scope

invoice(*),
  trim:(lower:(billing_country))="usa",
  total > 10
```

Null-safe vs Null-dangerous equality. Delightql reserves the `=` sigil for the SQL comparison operator `IS NOT DISTINCT FROM`. To use the traditional (dangerous) equality in SQL, use `+sql_eq(x,y)`.

delightql

```
invoice(*), total > 10,
    trim:(lower:(billing_country))="usa",
    +sql_eq(billing_state,"CA")
```

SQL

```
select * from invoice
  where total > 10
    and trim(lower(billing_country))
      IS NOT DISTINCT FROM 'usa'
    and billing_state='CA';
```

> Three-Valued Logic
>
> Null has been with databases since the very beginning and so has the debate about its semantics and danger.
>
> SQL provides 'good enough' semantics for its usage in the set operations of distinct, grouping, union and intersect, but it can be a foot-gun in other circumstances.
>
> The simplest display of its behavior below:

SQL

```
select
    null=null,
    null is null,
    null is not null,
    1=null,
    1 is null,
    1 is not null,
    1 in (select null union all select 2),
    1 not in (select null union select 2),
    1 in (select null union all select 1),
    1 not in (select null union select 1)
;
```

shows many odd results

```
null=null                                 =  null
null is null                              =  1
null is not null                          =  0
1=null                                    =  null
1 is null                                 =  0
1 is not null                             =  1
1 in (select null union all select 2)     =  null
1 not in (select null union select 2)     =  null
1 in (select null union all select 1)     =  1
1 not in (select null union select 1)     =  0
```

Sigil	Name	SQL Equivalent
=	NULL-SAFE-GROUND-EQ	IS NOT DISTINCT FROM
==	does NOT exist	
>	GROUND-GT	>
<	GROUND-LT	<
>=	GROUND-GTE	>=
<=	GROUND-LTE	<=
!=	NULL-SAFE-NOT-EQ	IS DISTINCT FROM

Table 5: Infix domain predicates

If a programmer requires the use of the traditional Sql = they can use the named functor: `+sql_eq(left,right)`. Likewise, for SQL's `!=` there is `+sql_ne(left,right)`

> The row-correspondence exception.
>
> The table above describes local value tests – conditions comparing values within one row or against a ground term. An ordinary comparison matching two relation row occurrences

> uses SQL `=`, whether the matches feed a multiplying join, an EXISTS/NOT EXISTS, or a correlated scalar subquery.
>
> `IS NOT DISTINCT FROM` in correspondence would treat NULL as a matchable value and can explode cardinality. Suppressing the extra rows with EXISTS does not make that correspondence correct.
>
> Joins establish structural correspondence – "these rows belong together." NULL means absence, and absence does not make a correspondence. Filters test value equality, where null-safety matters because rows should not silently disappear.
>
> SQL clause placement is not the equality rule: a correlated EXISTS often has an inner WHERE, and still matches two row occurrences. A local `inner.x = null` beside its correlation stays null-safe.
>
> There is no `cardinality/nulljoin` gate. If an absent key has a real matchable meaning, first map both sides to an explicit non-NULL key and match those keys. The NULL itself never establishes correspondence.

2.9.2. Argumentative Grounding

When using argumentative functor notation, a ground term in argument position induces selection:

delightql

```
album(album_id,title,22)
```

SQL

```
select
  album_id,
  title
from album where artist_id IS NOT DISTINCT FROM 22;
```

All argumentative grounding uses null-safe equality.

The grounded column (`artist_id  IS NOT DISTINCT FROM  22`) filters rows and is excluded from projection. Multiple grounds compound:

delightql

```
album(album_id,"Coda",22)
```

SQL

```
SELECT album_id
FROM album
WHERE
  title IS NOT DISTINCT FROM 'Coda'
  AND artist_id IS NOT DISTINCT FROM 22;
```

Any domain expression that reduces to a ground term may also be used in argumentative position:

delightql

```
album(album_id,upper:("iv"),(20 + 2))
```

SQL

```
select
  album_id
from album where title IS NOT DISTINCT FROM upper('iv') and
artist_id IS NOT DISTINCT FROM (20+2);
```

The Prolog heritage is evident in this syntax and extends to joins – covered in a later section.

2.9.3. Semi-Joins and Anti-Joins

Semi-joins (∃ or ⋉) and anti-joins (∄ or ▷) test for existence without contributing columns. They ask "can you prove this?" rather than "give me this data."

The PROVE sigil `+` prefixes a semi-join:

delightql

```
employee(*) as e, +customer(, e.employee_id=support_rep_id)
```

SQL

```
SELECT *
FROM employee AS e
WHERE
  EXISTS (
    SELECT 1
    FROM customer
    WHERE
      support_rep_id = e.employee_id
  );
```

The DISPROVE sigil `\+` prefixes an anti-join: [13]

13 This syntax comes directly from Prolog's negation-as-failure.

delightql

```
employee(*) as e, \+ customer(,
e.employee_id=customer.support_rep_id)
```

SQL

```
select
  *
from employee e
  where not exists (select 1 from customer
                    where support_rep_id = e.employee_id);
```

The join condition(s) appears inside the parentheses – this is called interior notation. The relation is tested for provability, not joined for data. The correlated row match still has join equality: two NULL keys do not correspond. `+` and `\+` change how matching rows are observed, not how they match. A comparison confined to one interior row, such as `customer.support_rep_id = null`, remains a local null-safe test.

2.9.4. The **in** Predicate

```delightql
customer(*), +_(state@"MA";"TX";"CA";"ON")
```

Syntactic sugar provides the familiar form:

```delightql
customer(*), state in ("MA";"TX";"CA";"ON")
```

Both ask whether the customer's state corresponds to one candidate. With these non-NULL constants, the filtering SQL can be written:

```sql
select
  *
from customer where state in ('MA','TX','CA','ON');
```

The unsugared form generalizes to multi-column comparisons:

```delightql
customer(*), +_( country, state @
                 "USA","CA";
                 "USA","TX";
                 "Canada","ON")
```

```sql
SELECT *
FROM customer
WHERE
  ('USA' = country
  AND 'CA' = state)
  OR ('USA' = country
  AND 'TX' = state)
  OR ('Canada' = country
  AND 'ON' = state);
```

`+_` is an existential observer of the same anonymous table an unmarked comma member would join. When a column from one row is matched to a candidate row, NULL never establishes correspondence, even under `+`. For example, `_(a @ null), +_(a @ null)` has no answer. A wholly ground `null in (null)` is a local value test instead and remains true.

2.9.5. Relational **in**

The literal form tests candidates in a fixed list. The relational form tests candidate rows from a query. Both are existence observations over the ordinary row-correspondence match, not a promise to emit SQL's three-valued `IN (SELECT ...)`.

The right-hand side is any DQL relation (a table access, a pipe chain, or an anonymous table):

```delightql
customer(*), support_rep_id in employee(|> (employee_id))
```

SQL

```
SELECT * FROM customer
  WHERE EXISTS (SELECT 1 FROM employee AS e
                WHERE e.employee_id = customer.support_rep_id);
```

When the relation already has exactly one column, projection is unnecessary:

delightql

```
customer(*) |> %(support_rep_id) : support_rep
employee(*), employee_id in support_rep(*)
```

SQL

```
WITH support_rep AS (SELECT DISTINCT support_rep_id FROM
customer)
SELECT * FROM employee
  WHERE EXISTS (SELECT 1 FROM support_rep AS r
                WHERE r.support_rep_id = employee.employee_id);
```

2.9.5.1. Tuple relational **`in`**

Multi-column matching extends the tuple `in` syntax (`(x,y) in (1,2;3,4)`) to relations. The relation must produce exactly as many columns as the left-hand tuple:

delightql

```
customer(*), (city, country) in employee(|> (city, country))
```

SQL

```
SELECT * FROM customer
  WHERE EXISTS (SELECT 1 FROM employee AS e
                WHERE e.city = customer.city
                  AND e.country = customer.country);
```

2.9.5.1.1. Negation: **`not in`**

delightql

```
employee(*), employee_id not in employee(|> (reports_to))
```

SQL

```
SELECT * FROM employee
  WHERE NOT EXISTS (SELECT 1 FROM employee AS m
                    WHERE m.reports_to = employee.employee_id);
```

The negative form is anti-existence, not SQL `NOT IN`: a NULL candidate does not poison a nonmatching probe. A NULL employee key corresponds to no candidate key, including another NULL; positive `in` then fails and negative `not in` succeeds for that row. A wholly ground test such as `null in (null)` is a local null-safe value comparison and succeeds.

> Arity rule

> The relation must produce exactly as many columns as the left side has elements – one for a scalar, N for an N-tuple. A mismatch is a compile-time error.

> Relation to semi-joins
>
> Relational `in` is syntactic sugar over the semi-join notation introduced above. `col in R(|> (c))` desugars to `+R(, col = c)`; `col not in R(|> (c))` desugars to `\+R(, col = c)`.

2.9.6. Inverted **`In`**

The anonymous semi-join syntax permits an inversion – ground the header, vary the rows:

delightql

```
customer(*),
    +_("Dublin" @
      city;
      state;
      country)
```

SQL

```
select
  *
from customer
  where city = 'Dublin'
    or state = 'Dublin'
    or country = 'Dublin';
```

This asks: "does 'Dublin' appear in any of these columns?" The columns become the rows of the anonymous table; the constant becomes the match target. Because `'Dublin'` is non-NULL, SQL `=` is a result-equivalent filter here. Grounding a candidate's header to NULL is instead a local null-safe test.

> SQL supports Inverted In
>
> Though it might be a revelation to some – including the author! – the inverted in is standard SQL and is fully supported by all dialects:
>
> SQL
>
> ```
> select
>   *
> from customer
>   where 'Dublin' in
>     (city,state,country);
> ```

Similarly, to test if one column equals any of several others:

delightql

```
customer(*), +_(city @ state; country)
```

SQL

```
select
  *
from customer
  where city = state
    or city = country;
```

Here `city` is supplied by the customer row and each candidate is an anonymous-table row. `+` keeps or drops the customer depending on whether such a row corresponds; it does not turn NULL-to-NULL into a match. A wholly ground anonymous test still follows the local ground-value law.

2.9.7. Sigma Predicates

Predicates can be defined and reused. A sigma predicate is a rule that expands into selection criteria: [14]

14 Defining sigma predicates is covered in DDL.

delightql

```
no_data("NA";"N/A";"UNKNOWN")

empty(column) :- null=column
empty(column) :- trim:(column)=""
empty(column) :- +no_data(upper:(column))
```

Use with semi-join or anti-join syntax:[15]

15 Delightql applies De Morgan's laws, distributing negation across disjunctive clauses.

delightql

```
customer(*),
  \+empty(company),
  \+empty(state)
```

SQL

```
WITH no_data(value) AS (VALUES ('NA'), ('N/A'), ('UNKNOWN'))
SELECT *
FROM customer
WHERE
  (company IS NULL
    OR trim(company) IS NOT DISTINCT FROM ''
    OR EXISTS (SELECT 1 FROM no_data WHERE upper(company) =
no_data.value)
  ) IS NOT TRUE
  AND (state IS NULL
    OR trim(state) IS NOT DISTINCT FROM ''
    OR EXISTS (SELECT 1 FROM no_data WHERE upper(state) =
no_data.value)
  ) IS NOT TRUE;
```

2.9.7.1. *Like* and **Between**

SQL's `LIKE` and `BETWEEN` have special syntax. Delightql maps functor notation to these constructs:

delightql

```
track(*), +like(name,"%Love%"), \
+between(milliseconds,120000,600000)
```

The above delightql transpiles to the following Sql.

SQL

```
select
  *
from track
  where
    (name like '%Love%') is true and
    (milliseconds between 120000 and 600000) is not true;
```

To match a value against several patterns at once, `like_any` takes the value, then `&`, then the patterns as rows. Under `+` it keeps the rows whose value matches at least one pattern; under `\+`, the rows whose value matches none:

delightql

```
customer(*), \+like_any(email & "%.com"; "%.org")(*)
```

It is equivalent to:

SQL

```
select *
from customer
where not exists (
  select 1
  from (select '%.com' as pattern union all select '%.org') as p
  where email like p.pattern
);
```

The patterns can be any one-column relation — `+like_any(email, domains(*))(*)`. Without the `+` it joins instead of filtering: each row gains the `pattern` it matched, once per match. A NULL value matches no pattern.

2.9.7.2. Disjunction

Two syntaxes express `OR`:

Keyword form (recommended). The `or` keyword binds predicates within a sigma clause:

delightql

```
invoice(*)
  , trim:(lower:(billing_country)) = "usa"
       or total > 10
  , billing_state != "CA"
  |> %( billing_country
          ~>
       count:(*) as invoice_count,
       avg:(total) )
```

Sigil form. The SEMI-OR sigil `;` requires parentheses to capture scope:

delightql

```
invoice(*)
  , (trim:(lower:(billing_country)) = "usa"
       ; total > 10 )
  , billing_state != "CA"
  |> %( billing_country
          ~>
        count:(*) as invoice_count,
        avg:(total) )
```

Prefer the keyword form – it reads more clearly and avoids parenthesis errors. See “Precedence and Scoping” for details.

2.10. JOIN

Joins extend a relation’s schema by combining columns from multiple sources. Every join is a filtered cross product – the join condition determines which pairings survive.

Delightql evaluates joins left to right. Each table must be in scope before its columns can be referenced.

2.10.1. Cross Join

delightql

```
media_type(*), genre(*)
```

SQL

```
SELECT * FROM media_type CROSS JOIN genre;
```

Two relations joined with no condition produce a Cartesian product. The resulting cardinality is the product of both input cardinalities. This is rarely intended.

2.10.2. Inner Join

delightql

```
album(*), artist(*), album.artist_id = artist.artist_id
```

SQL

```
SELECT * FROM album
  JOIN artist ON album.artist_id = artist.artist_id;
```

The join condition follows the tables it correlates. Multiple conditions conjoin naturally:

delightql

```
customer(*), employee(*),
  customer.support_rep_id = employee.employee_id,
  customer.country = employee.country
```

Scope is left to right. This is an error:

delightql

```
// INVALID: artist not yet in scope
album(*), artist.artist_id = album.artist_id, artist(*)
```

2.10.3. **`USING`** Shorthand

The `.(cols)` operator specifies USING columns:

delightql

```
album(*), artist(*.(artist_id))
```

SQL

```
SELECT * FROM album JOIN artist USING (artist_id);
```

Multiple columns are comma-separated:

delightql

```
invoice_line(*), track(*.(track_id, unit_price))
```

SQL

```
SELECT * FROM invoice_line JOIN track USING (track_id,
unit_price);
```

`USING` and explicit `ON` differ operationally: `USING` retains one copy of the matched column; `ON` retains both.

`USING` may combine with explicit conditions:

delightql

```
track(*), album(*.(album_id)), track.name = album.title
```

2.10.4. ON vs WHERE Inference

For inner joins, the placement of conditions in `ON` versus `WHERE` is semantically equivalent. For outer joins, it differs: `ON` conditions govern the match while preserving nulls; `WHERE` conditions filter the result and eliminate nulls.

Delightql infers placement from column references:

- Condition references multiple tables ⟶ `ON`
- Condition references one table ⟶ `WHERE`

delightql

```
artist(*), album?(*),
  artist.artist_id = album.artist_id,    // two tables ⟶ ON
  album.title = "Greatest Hits"          // one table ⟶ WHERE
```

SQL

```
SELECT * FROM artist
  LEFT OUTER JOIN album
    ON artist.artist_id = album.artist_id
WHERE album.title = 'Greatest Hits';
```

The multi-table condition (`artist.artist_id = album.artist_id`) becomes the join's `ON` clause. The single-table condition (`album.title = 'Greatest Hits'`) becomes a `WHERE` filter – artists with no albums, or with no album of that title, are excluded.

To keep every artist while matching only albums of that title, constrain the albums before the join: filter them in a stage of their own and make that stage the optional member.

delightql
```
album(*), title = "Greatest Hits" : hits
artist(*), hits?(*.(artist_id))
```

2.10.5. Multi-Table Joins

Joins chain left to right. Each table enters scope upon appearance:

delightql
```
invoice(*),
  invoice_line(*.(invoice_id)),
  track(*.(track_id)),
  invoice.invoice_date > "2025-01-01"
```

SQL
```
SELECT * FROM invoice
  JOIN invoice_line USING (invoice_id)
  JOIN track USING (track_id)
WHERE invoice.invoice_date > '2025-01-01';
```

After the third table, columns from all three are in scope.

2.10.6. Self-Join

Aliases distinguish multiple references to the same table:

delightql
```
employee(*) as e, employee(*) as mgr, e.reports_to =
mgr.employee_id
  |> (e.last_name as employee, mgr.last_name as Manager)
```

SQL
```
SELECT e.last_name AS employee, mgr.last_name AS Manager
FROM employee e
  JOIN employee mgr ON e.reports_to = mgr.employee_id;
```

2.10.7. Argumentative Join

Shared identifiers across functors induce join conditions – Prolog-style unification:

delightql
```
album(album_id, title, artist_id), artist(artist_id, name)
```

SQL
```
SELECT album.album_id, album.title, album.artist_id, artist.name
FROM album
  JOIN artist ON album.artist_id = artist.artist_id;
```

The variable `artist_id` appears in both functors, unifying the columns.

Multi-table example:

delightql

```
artist(artist_id, name),
  similar_artist(artist_id, similar_id, score),
  artist(similar_id, similar_name),
  score > 0.85
  |> (name, similar_name)
```

Argumentative joins are idiomatic in Prolog. Delightql supports them but recommends `.(cols)` or explicit conditions for wide tables where positional notation becomes error-prone.

2.10.8. Outer Joins

The OUTER-IND sigil `?` marks a relation as optional – it may contribute nulls when no match exists.

Left outer (right table optional):

delightql

```
artist(*), album?(*.(artist_id))
```

SQL

```
SELECT * FROM artist LEFT OUTER JOIN album USING (artist_id);
```

Right outer (left table optional):

delightql

```
album?(*), artist(*.(artist_id))
```

Full outer (either optional):

delightql

```
album?(*), artist?(*.(artist_id))
```

Outer joins work with explicit conditions:

delightql

```
artist(*), album?(*), artist.artist_id = album.artist_id
```

SQL

```
SELECT * FROM artist
  LEFT OUTER JOIN album ON artist.artist_id = album.artist_id;
```

2.10.9. Lateral Joins

Correlated subqueries that return multiple columns use interior relation syntax. See Interior Relations.

2.10.10. ER-Context Joins

When join relationships are defined via ER-context-rules (see DDL):

```delightql
customer(*) & invoice(*) :-
  customer(*), invoice(*), customer.customer_id =
invoice.customer_id

invoice(*) & invoice_line(*) :-
  invoice(*), invoice_line(*), invoice.invoice_id =
invoice_line.invoice_id

invoice_line(*) & track(*) :-
  invoice_line(*), track(*), invoice_line.track_id = track.track_id
```

the `&` and `&&` operators provide concise join syntax:

```delightql
  customer(*) & invoice(*)
```

Equivalent to:

```delightql
customer(*), invoice(*), customer.customer_id = invoice.customer_id
```

The `&` operator performs direct lookup; `&&` finds a path through the ER-graph:

```delightql
customer(*) && track(*)
// Compiler finds: customer -> invoice -> invoice_line -> track
```

ER-context joins compose with all other features – filters, projections, aggregations, additional explicit joins.

For defining ER-rules and contexts, see DDL: ER-Context Rules.

2.11. DOMAIN FUNCTIONS

Functions on domains are ordinary functions–what most programming languages simply call "functions." The qualifier "on domains" distinguishes them from functions on relations (table-valued functions, covered elsewhere). Here, "domain" means data type: the addition function `+` in `SELECT Salary + bonus FROM employee` expects numeric operands.

SQL provides several syntaxes for functions: standard functor notation, infix operators, and `CASE` expressions. This section covers how to call domain functions in delightql. For defining functions, see Function Definition.

Domain functions can appear anywhere a column is valid – they are substitutable for the value they compute:

- During projection, as a transformation of a column
- During selection, as a transformation prior to comparison
- During grouping, as a transformation prior to grouping
- During aggregation, as a reduction of multiple values

Delightql's default function syntax incorporates a colon: `foo:(x)` rather than `foo(x)`. This functional functor notation distinguishes functions from relations. [16]

16 Delightql's syntax derives from Prolog, where `foo(x,y)` denotes a relation. Functional functor notation marks the subset of relations that are functions – mappings that return exactly one value per input.

2.11.1. Standard Function Invocation

delightql

```
employee(*),
  length:(last_name) > 5
  |> +( trim:(upper:(last_name)) as capitalized_last_name)
```

Three functions appear in two contexts:

- `length:()` in a predicate (sigma clause)
- `trim:()` and `upper:()` in an embed, using containment composition

The colon signals to both compiler and programmer that the functor returns a domain value, not a relation.

2.11.2. Function Pipe Composition

The F-PIPE sigil `>>` composes functions left to right:

delightql

```
employee(*), length:(last_name) > 5
  |> +( last_name >> upper:() >> trim:() as
capitalized_last_name)
```

The pipe begins with a domain expression (a column, literal, or function call). Each subsequent function is in curried form. The piped value fills the final argument unless an explicit `@` places it elsewhere.

When the value belongs in a different argument position, use the F-PARAM sigil `@`:

delightql

```
employee(*), length:(last_name) > 5
  |> +(  birth_date >> strftime:("%Y",@) as birth_year )
```

The LAMBDA sigil `:( )` creates an inline function. The `@` marks where the piped value is placed.

delightql

```
employee(*), length:(last_name) > 5
    |> +(  birth_date >> strftime:("%Y",@) >> :( @ + 2) as
birth_year_plus_two,
           birth_date >> strftime:("%Y",@) >> abs:() as
abs_of_birth_year)
```

A lambda has an alternate spelling where a programmer may name the parameter coming in:

delightql

```
employee(*), length:(last_name) > 5
    |> +(  birth_date >> strftime:("%Y") >> :(|bday| bday + 2)
as birth_year_plus_two,
           birth_date >> strftime:("%Y") >> abs:() as
abs_of_birth_year)
```

Use the LAMBDA-PARAM | | to capture and name the input variable as the named parameter.

2.11.3. Domain Operators

Delightql supports standard arithmetic operators:

Sigil	Name	Arity
*	OP-MULT	Binary
+	OP-PLUS	Binary
-	OP-MINUS	Binary
-	OP-NEGATIVE	Unary
/	OP-DIV	Binary
%	OP-MODULO	Binary

Table 6: Arithmetic domain operators

No implicit precedence. Delightql requires explicit parentheses when mixing operators. There is no PEMDAS. [17] This is a deliberate design choice favoring clarity over convention.

[17] See "Order of Operations" for the conventional rules delightql declines to adopt.

delightql

```
_(first,second,third @ 1.3,20,30)
    |> ( 1 + (first / third),
         -third / ((first * second) / 23.33),
         (third % 11) - 42)
```

SQL

```
select
  1+(first/third),
  -third / ((first*second)/23.33),
  (third % 11) - 42
from (select 1.3 as first, 20 as second, 30 as third);
-- 1.043333333333333|-26.9192307692308|-34
```

2.11.4. *Case* Simple

SQL's `CASE` expression serves as a switch statement. Delightql represents it as what it is: a function, and therefore a relation.

delightql

```
employee(*)
    |> +(  _:(title @
             "IT Manager" -> "tech";
             "IT Staff"   -> "tech";
             _            -> "other") as kind )
```

SQL

```sql
select
  *,
  case title
    when 'IT Manager' then 'tech'
    when 'IT Staff' then 'tech'
    else 'other'
  end as kind
from employee;
```

The ANON-FUNC sigil `_:(  )` creates an anonymous case function. The F-AND sigil `->` separates input patterns (left) from output values (right). The SEMI-OR sigil `;` separates cases. The header `title` `@` binds the input to the `title` column.

This is stacked notation applied to functions: the `->` acts as a special comma that declares a functional dependency – columns left of the arrow are inputs, columns right are outputs.

Named case functions. The same notation defines reusable functions in assertion mode:

delightql

```
title_kind(
  title        -> kind
  ------------------
  "IT Manager" -> "tech";
  "IT Staff"   -> "tech";
  _            -> "other"
)

?- _(title @ "IT Staff"; "Sales Manager")
  |> +(  title_kind:(title) as kind )
```

The predicate `title_kind` is both a table (two columns, three rows) and a function (input determines output). The `->` tells the compiler which column is the input when invoked as a function.

Without `->`, the predicate is valid but not callable as a function:

delightql

```
//WILL NOT WORK!! (at least if you want to use it for function calls)
//   .. it is a perfectly acceptable predicate
title_kind("IT Manager"    , "tech")
title_kind("IT Staff"      , "tech")
title_kind("Sales Manager" , "other")
```

[18]

18 Prolog calls these input/output declarations modes or adornments. Delightql's `->` serves the same purpose: columns left of the arrow must be instantiated inputs.

2.11.5. ***Case*** Search Function

Case search evaluates conditions rather than matching values. Two syntaxes exist.

Condition-first notation uses -> pointing to the return value:

delightql

```
invoice(*)
    |> %(  _:( total > 15              ->  "A";
            total > 10, total <= 15  ->  "B";
            total > 5,  total <= 10  ->  "C";
            total > 2,  total <= 5   ->  "D";
            _                        ->  "F") as tier
            ~> count:(*) )
    |>  #(tier)
```

Conditions can be conjoined with `,` (and). For disjunction, use the keyword `or`:

delightql

```
invoice(*)
    |> %(  _:( total > 15  or billing_country = "USA" -> "A";
               total > 10, total <= 15                -> "B";
               total > 5,  total <= 10                -> "C";
               total > 2,  total <= 5                 -> "D";
               _                                      -> "F") as
tier
            ~> count:(*) )
    |>  #(tier)
```

Like SQL's `CASE`, the first matching clause wins.

delightql

```
customer(*)
    |> (  first_name,
          last_name,
          country,
          _:(
              country in ("USA";"Canada"), state in ("CA";"BC") -> "north america
west";
              country in ("USA";"Canada")                       -> "north
america";
              country in ("Brazil";"Argentina";"Chile")         -> "south
america";
              country in ("Spain";"Portugal")                   -> "iberia"
          ) as region )
    |>  #(first_name,last_name)
```

2.11.6. Format Function and Strings

Format strings interpolate column values into text. The F-STRING sigil `:` prefixes a string literal:

delightql

```
1 invoice(*)
2   |> +( :"{billing_city}, {billing_country} billed ${total}" as
readable)
```

SQL

```sql
select
    invoice_id,
    customer_id,
    invoice_date,
    billing_address,
    billing_city,
    billing_state,
    billing_country,
    billing_postal_code,
    total,
    billing_city || ', ' || billing_country || ' billed $' ||
total as readable
from invoice;
```

Braces `{ }` enclose column names. Format strings also permit the usage of the following escape sequences:

escape sequence	meaning
\n	newline
\t	tab
\\	backslash
\q	single quote (')
\Q	double quote (")

Strings in delightql are only double-quoted. Any double-quoted string without a colon is a raw string and accepts neither interpolation nor escape sequences. Delightql also uses the triple-double-quotes to make for easier embedding of double quotes. These too may be used as string interpolators with a preceding colon.

delightql

```
_(1) |> ("""
  This is a banner and
  has no \n as a newline escape
  but the ones you typed are
  there because they are the bytes
  you typed
  """)
```

delightql

```
_(1) |> (:"""
  This is a banner and
  HAS the \n as a newline escape
  AND the ones you typed are
  there because they are the bytes
  you typed.  Also, you can type a
  double-quote as " or \Q and a
  single-quote as ' or \q
  """)
```

2.11.7. Citations and Symbols

Citations and symbols are thematically related. They roughly correspond to constant and symbols in other languages.

2.11.7.1. Citations

A citation is a special calling syntax for any function of zero arity.

delightql

```
version:() :  "v1.2.8"
employee(*)
  |> ( :version as vrs)
```

This calling syntax makes the invocation look like a global variable or a constant. The only reason for using this calling syntax is to emphasize the global nature of the citation. A function with zero arguments is usually effectively a constant.

2.11.7.2. Symbols

A symbol is special syntax for a string.

delightql

```
employee(*) |> +( ::ident as some_identifier )
```

Unlike a citation which requires a definition to use, a symbol is self-denoting and can be called wherever a string is used. When a symbol is written out it immediately treats the symbol as a string.

Symbols have one more special property: they only allow identifiers or functor syntax after the `::`

- `::identifier`
- `::functor(a,b)`

Delightql uses symbols wherever strings requiring special semantics are needed.

2.11.8. Window/Analytic Functions

Window functions combine aggregation with per-row results. They aggregate over a dynamic window but return one value per row – still scalar functions by definition. [19]

19 Array languages are the closest analog in other programming paradigms.

delightql

```
invoice(*)
    |> (invoice_id,
        customer_id,
        total,
        dense_rank:( <~ %(customer_id),#(total)) as ranking )
```

SQL

```sql
SELECT
  invoice_id,
  customer_id,
  total,
  dense_rank() OVER (
    PARTITION BY
      customer_id
    ORDER BY total
  ) AS ranking
FROM invoice;
```

The F-OVER sigil `<~` introduces the window specification. Everything before `<~` is passed to the function; everything after defines the window frame.

Window specification syntax. Comma-separated, all optional:

- `%( )` – partition clause (one allowed)
- `#( )` – order clause (one allowed)
- `rows(from, to)`, `range(from, to)`, or `groups(from, to)` – frame specification (one allowed)

Frame Bounds:

Syntax	Meaning
`.`	current row
`_`	unbounded
`+n`	n following
`-n`	n preceding

Table 8: Window frame bound syntax

Examples:

delightql

```
invoice(*)
    |> (invoice_id, customer_id, total,
        ntile:( 10  <~  %(customer_id),#(total desc),
groups(_,_))      as g_all,
        ntile:( 10  <~  %(customer_id),#(total desc),
groups(+1,_))     as g_after,
        ntile:( 10  <~  %(customer_id),#(total desc),
rows(-1,.))       as r_prev,
        ntile:( 10  <~  %(customer_id),#(total desc), rows(_,-
(2*2)))   as r_upto,
        ntile:( 10  <~  %(customer_id),#(total desc), range(.,
+(2*2)) ) as rg)
```

SQL

```
select invoice_id, customer_id, total,
  ntile(10) over
    ( partition by customer_id order by total desc
      groups between
        unbounded preceding and unbounded following) as g_all,
  ntile(10) over
    ( partition by customer_id order by total desc
      groups between 1 following and unbounded following) as g_after,
  ntile(10) over
    ( partition by customer_id order by total desc
      rows between 1 preceding and current row) as r_prev,
  ntile(10) over
    ( partition by customer_id order by total desc
      rows between unbounded preceding and (2*2) preceding) as r_upto,
  ntile(10) over
    ( partition by customer_id order by total desc
      range between current row and (2*2) following) as rg
from invoice;
```

Default window. For an empty window specification, use `<~` with nothing following:

delightql

```
employee(*)
  |> +(  row_number:( <~ ) as row_number )
```

SQL

```
select
  *,
  row_number() over () as row_number
from employee;
```

2.11.9. Scalar Subquery

Scalar subqueries return a single value (one row, one column) usable anywhere a column is valid. Delightql transforms a relation into a scalar subquery by postfixing its name with `:` and using interior notation.

Uncorrelated. The subquery is independent of the outer query:

delightql

```
invoice(*)
    |> (invoice_id,
        billing_country,
        total,
        invoice:( ~> avg:(total)) as avg_total)
```

SQL

```
select
  invoice_id,
  billing_country,
  total,
  (select avg(total) from invoice) as avg_total
from invoice;
```

The F-COLON sigil `:` after the relation name signals a scalar subquery. The interior notation–here `~> avg:(total)` – must produce exactly one row and one column.

Correlated. The subquery references values from the outer query. Use an explicit condition to correlate on a column:

delightql

```
invoice(*) as i
    |> (invoice_id,
        billing_country,
        total,
        invoice:( ~> avg:(total)) as avg_total,
        invoice:( , billing_country = i.billing_country ~> avg:(total)) as
avg_total_in_country)
```

SQL

```
select
  invoice_id,
  billing_country,
  total,
  (select avg(total) from invoice) as avg_total,
  (select avg(total) from invoice
      where billing_country = i.billing_country) as
avg_total_in_country
from invoice i;
```

2.11.10. Context-Aware Functions

A context-aware function closes over columns from its invocation context rather than receiving them as explicit parameters.

Context-aware functions are defined with `..` in their signature:

delightql

```
// only works within the context of a
// relation that has 'billing_city' and 'Total' columns

cost_of_living:( .. ) :-
  _:(  billing_city in ("Mountain View";"Boston";"New York")
          -> total*0.8;
       billing_city in ("Fort Worth"; "Tucson")
          -> total*1.45;
       _ -> total)
```

and called with it:

delightql

```
invoice(*)
  |> ( cost_of_living:(..) as col_adjusted_total)
```

The UP-CONTEXT sigil `..` signals that the function references columns from the surrounding scope. It is required – a reminder that the function is never truly nullary.

This function can be invoked on any relation with `billing_city` and `total` columns – the free variables in its body. These columns are bound at the call site, not passed explicitly.

Context-aware functions have no SQL counterpart; they are expanded inline during transpilation.

2.11.11. Boolean Expressions as Columns

Predicates – what delightql calls sigma clauses – can appear in column position, returning boolean values. [20]

20 Some SQL dialects lack a boolean type; these transpile to 1 and 0.

delightql

```
invoice(*)
    |> +( billing_country="USA"
            and billing_state!="CA"
                AS usa_outside_ca,
          billing_country="USA"
                AS usa,
          total > 15
            or invoice_date >= "2025-01-01"
                AS big_or_recent,
          billing_state!="CA"
                AS outside_ca)
```

SQL

```
select
  *,
  billing_country is not distinct from 'USA'
    and billing_state is distinct from 'CA' as usa_outside_ca,
  billing_country is not distinct from 'USA' as usa,
  total > 15 or invoice_date >= '2025-01-01' as big_or_recent,
  billing_state is distinct from 'CA' as outside_ca
from invoice;
```

Compound predicates must use keywords (`and`, `or`) rather than sigils (`,` , `;`) when appearing as column expressions.

Existence tests. Semi-joins and anti-joins also return booleans when used in column position:

delightql

```
employee(*)
 |> +( +customer(,
         customer.support_rep_id
          =employee.employee_id),
      \+ customer(,
         customer.support_rep_id
          =employee.employee_id),
     +between(hire_date,"2002-01-01","2002-12-31"))
```

SQL

```
select
  *,
  --
  exists (select 1 from customer
    where customer.support_rep_id=employee.employee_id),
  --
  not exists (select 1 from customer
    where customer.support_rep_id=employee.employee_id),
  --
  hire_date between '2002-01-01' and '2002-12-31'
from employee;
```

2.11.12. Compound Data Constructors

The enclyphs `{ }` and `[ ]` construct compound data–records and tuples. They are functions, though they look like syntax. Their behavior depends on context:

Position	`{ }`	`[ ]`
Scalar (non-reduction)	interior record	interior tuple
Aggregate (after ~>)	table of records	table of tuples

Table 9: Compound data constructor behavior by position

delightql

```
// scalar constructors
employee(*) |>
  ( [last_name,first_name] as interior_tuple )
employee(*) |>
  ( {last_name,first_name} as interior_record )

// aggregate constructors
employee(*)
  |> %( title
        ~>
      [last_name,first_name] as table_of_tuples )
employee(*)
  |> %( title
        ~>
      { last_name,first_name } as table_of_records )
```

These constructors transpile to JSON in most SQL dialects. [21] But the concept is not about JSON per se but about nested structure.

21 SQLite and Postgres both provide JSON as a data type with supporting functions. See SQLite JSON1.

> The compound data types introduced here provide groundwork for pivots, melts, and tree grouping (see Pivot and Melt and Tree Groups). Programmers needing arbitrary JSON manipulation can call SQL's JSON functions directly:
>
> `json_array:(last_name, first_name)`.

2.11.12.1. Scalar Interior Record

The INTERIOR-RECORD enclyph `{ }` creates a nested row addressable by name:

delightql

```
employee(*)
  |> (title , { last_name,first_name } as name  )
```

title	name
General Manager	`{"last_name":"Adams","first_name":"Andrew"}`

Sales Manager	`{"last_name":"Edwards","first_name":"Nancy"}`

Table 10: Scalar interior record result

Column names become keys. To specify different keys:

delightql

```
employee(*)
  |> (title ,
      { "first_name": first_name ,
        "last_name" : last_name} as name  )
```

Access nested fields with JSON-access notation (see next section).

delightql

```
employee(*)
  |> (title ,
      { "first_name": first_name ,
        "last_name" : last_name} as name  )
  |> ( title, name:{.first_name})
```

SQL

```
with
    _cpr0 as (
        select
          title,
          json_object('first_name',first_name,
                      'last_name' ,last_name) as name
        from employee)
    select
        title,
        name ->> "$.first_name" as first_name
    from _cpr0;
```

2.11.12.2. Aggregate Interior Record

In a reduction position, `{ }` collects multiple records into a table:

delightql

```
employee(*)
  |> %(title ~> { last_name,first_name } as name )
```

title	name
IT Staff	[{"last_name":"King","first_name":"Robert"}, {"last_name":"Callahan","first_name":"Laura"}]
Sales Support Agent	[{"last_name":"Peacock","first_name":"Jane"}, {"last_name":"Park","first_name":"Margaret"}, {"last_name":"Johnson","first_name":"Steve"}]

Table 11: Aggregate interior record result – grouped by `title`

The outer `[ ]` in the JSON represents multiplicity – a list of rows. The interior table has a uniform schema of named columns represented as objects.

2.11.12.3. Scalar Interior Tuple

The INTERIOR-TUPLE enclyph `[ ]` creates a nested row addressable by position:

delightql

```
employee(*)
  |> (title , [last_name,first_name] as name )
```

title	name
General Manager	["Adams","Andrew"]
Sales Manager	["Edwards","Nancy"]

Table 12: Scalar interior tuple result

Access elements by index:

delightql

```
employee(*)
  |> (title , [last_name,first_name] as name )
  |> ( title, name:{.1} as first_name)
```

2.11.12.4. Aggregate Interior Tuple

In a reduction position, `[ ]` collects multiple tuples into a table:

delightql

```
employee(*)
  |> %(title ~> [ last_name,first_name ] as name )
```

title	name
IT Staff	[["King","Robert"], ["Callahan","Laura"]]
Sales Support Agent	[["Peacock","Jane"], ["Park","Margaret"], ["Johnson","Steve"]]

Table 13: Aggregate interior tuple result – grouped by `title`

JSON's `[ ]` does double duty here: the outer brackets indicate multiple rows; the inner brackets indicate tuples. This is a syntactic limitation of JSON, not a semantic ambiguity. [22]

22 If JSON had a distinct tuple syntax – perhaps parentheses – this overloading would not exist.

2.11.12.5. Nesting

The power of these constructors emerges when nested:

delightql

```
customer(*)
  ~>  {  country ,
           "people_by_state":
             ~>{ state ,
                "people" : ~>{first_name, last_name} } }
              as people_by_state_within_country
```

This tree-structured output is covered in detail in the Tree Groups section.

2.11.13. Compound Data Pathing

Once compound data has been constructed, access its contents via pathing syntax.

Array access. Name the column, followed by a colon, the `{ }` enclyph, and a 0-indexed position:

delightql

```
employee(*)
  |> (title , [last_name,first_name] as name )
  |> ( title, name:{ .0 } as last_name)
```

Record access. Name the column, followed by a colon, the `{ }` enclyph, and a dot-prefixed key:

delightql

```
employee(*)
  |> (title ,
      { "first_name": first_name ,
        "last_name" : last_name} as name  )
  |> ( title, name:{.first_name})
```

Nested access. Chain steps with dots to descend into nested structures:

delightql

```
employee(*)
  |> ( {last_name, "hardcoded" : [ 1,"x"] } as packet)
  |> ( packet:{.hardcoded.1})
// returns "x" for as many records as there are employees
```

delightql

```
_(x @ [ 1 , 2 , {"hardcoded" : {"deeper": [ 2 , 3]}}]) as
named_anon_table
  |> (x:{.2.hardcoded.deeper.0})
```

Uniform dot notation. Unlike most languages, delightql does not alternate between `.key` and `[index]` when pathing. All steps – whether into a record or an array – use dot separation: `key.1.nested.0` rather than `key[1].nested[0]`.

2.12. ADDRESSING COLUMNS BY INDEX

Named columns are preferred for clarity, but some schemas resist naming–wide CSVs, auto-generated headers, legacy tables with hundreds of columns. For these, delightql provides index notation.

delightql

```
employee(*)
  |> ( |1|, |2|, |3| )
```

SQL

```
select
    employee_id,  -- at position 1
    last_name,    -- at position 2
    first_name    -- at position 3
from employee;
```

The INDEX enclyph `|  |` encloses an integer literal (no expressions). Indices are 1-based; negative indices count from the end.

delightql

```
employee(*)
  |> (|-3|, |-2|, |-1|)
```

SQL

```
SELECT
  phone,   -- position -3
  fax,     -- position -2
  email    -- position -1
FROM employee;
```

> Indexing conventions. Column indices are 1-based, following SQL's ORDER BY 1 convention. Array pathing is 0-based, following JSON/JavaScript convention. The first column is `|1|` the first array element is `[0]`.

2.12.1. Column Ranges

A COLUMN RANGE `|start:end|` selects a contiguous slice of columns by position. Both bounds are inclusive and 1-based.

delightql

```
customer(*)
  |> ( |1:3| )
```

SQL

```
SELECT customer_id, first_name, last_name
FROM customer;
```

Either bound may be omitted. An open start means "from the first column"; an open end means "through the last column":

delightql
```
customer(*)
  |> ( |:3| )            // first three columns
```

SQL
```
SELECT customer_id, first_name, last_name
FROM customer;
```

delightql
```
customer(*)
  |> ( |5:| )            // fifth column onward
```

SQL
```
SELECT address, city, state, country, postal_code, phone, fax,
email, support_rep_id
FROM customer;
```

Negative indices count from the end, following the same convention as single ordinals:

delightql
```
customer(*)
  |> ( |-3:-1| )         // last three columns
```

SQL
```
SELECT fax, email, support_rep_id
FROM customer;
```

delightql
```
customer(*)
  |> ( |:-2| )           // all but the last column
```

SQL
```
SELECT customer_id, first_name, last_name, company, address,
city, state, country,
       postal_code, phone, fax, email
FROM customer;
```

Ranges can be scoped to a table alias, just like single ordinals:

delightql
```
customer(*) as c
  |> ( c|1:3|, c|6:8| )
```

SQL
```
SELECT customer_id, first_name, last_name, city, state, country
FROM customer AS c;
```

Ranges compose with other operators. For example, EMBED-MAP can apply a function across a range of columns:

delightql
```
track(*)
  |> +$(:( @ + 100) as :"{@}_offset")(|7:9|)
```

Syntax	Meaning
`|1:3|`	Columns 1 through 3
`|5:|`	Column 5 through last
`|:3|`	First through column 3

Syntax	Meaning
`|-3:-1|`	Third-to-last through last
`|:-2|`	First through second-to-last
`c|1:3|`	Columns 1–3 of alias `c`

Table 14: Column range syntax summary

When ranges break down. The same caveats as single ordinals apply: schema changes silently shift what a range covers. Prefer named columns for stable queries; reserve ranges for exploration and hostile schemas.

Index notation works with PROJECT-OUT, RENAME-COVER, MAP-COVER, GROUP-MODULO, and other operators:

delightql

```
employee(*)
  |> -( |1|, |2| , |-2| )
```

Scoped Index Notation

In joins, indices can be scoped to a table alias:

delightql

```
customer(*) as c,
  employee(*) as e, e.employee_id=c.support_rep_id
  |> (  |12| as email,
        c|1| as customer_id,
        e|-1| as rep_email)
```

Unscoped indices refer to the total column order across all joined tables. The following addressing schemes are available:

Scheme	Example	Meaning
Total	`|14|`	14th column overall
Total reverse	`|-5|`	5th from end overall
Scoped	`e|1|`	1st column of `e`
Scoped reverse	`e|-1|`	Last column of `e`
Named	`e.email`	By name

Table 15: Index addressing schemes

When index notation breaks down. Total indexing across joins depends on column counts and join order. For this reason, it's probably wise to reserve index notation for exploration, and managing hostile schemas.

2.12.2. Reposition Operator

The REPOSITION operator `*[column as position]` moves columns to specific positions without removing any columns.

delightql
```
invoice_line(*) |> *[unit_price as 1]
```

SQL
```
SELECT unit_price, invoice_line_id, invoice_id, track_id,
quantity FROM invoice_line;
-- Before: (invoice_line_id, invoice_id, track_id, unit_price, Quantity)
-- After:  (unit_price, invoice_line_id, invoice_id, track_id, Quantity)
```

The column `unit_price` moves to position 1; all other columns shift to accommodate.

2.12.2.1. Positive and Negative Positions

Positions are 1-indexed. Negative positions count from the end:

Position	Meaning
1	First
2	Second
-1	Last
-2	Second-to-last

Table 16: Position meanings for the reposition operator

delightql
```
invoice_line(*) |> *[invoice_line_id as -1]
```

Moves `invoice_line_id` to the last position:

```
Before: (invoice_line_id, invoice_id, track_id, unit_price,
quantity)
After:  (invoice_id, track_id, unit_price, quantity,
invoice_line_id)
```

2.12.2.2. Multiple Repositions

Multiple columns can be repositioned in a single operation:

delightql
```
invoice_line(*) |> *[unit_price as 1, quantity as 2]
```

```
Before: (invoice_line_id, invoice_id, track_id, unit_price,
quantity)
After:  (unit_price, quantity, invoice_line_id, invoice_id,
track_id)
```

Columns are placed in the order specified; remaining columns fill the gaps.

2.13. COMMON EXPRESSIONS

Delightql has multiple forms of common expression: the traditional common table expression, and multiple varieties of delightql-specific common function expression (higher and lower order).

2.13.1. Common Table Expressions

Create a common table expression (CTE) by naming a functor with a glob, followed by a `:`, also known as SHADOW-NECK, followed by the query that is assigned to that name. This syntax is called pre-labeling.

Once, a common table expression is defined, it is sufficient to query from that as if it were a table.

delightql

```
big_invoices(*) : invoice(*), total > 15
big_invoices(*)
```

SQL

```
WITH "big_invoices" AS (
  SELECT *
  FROM "invoice"
  WHERE "total" > 15
)
SELECT *
FROM "big_invoices";
```

An alternate syntax, called post-labeling, allows the CTE to be named after the query by postfixing a valid query with the SHADOW-NECK `:` and a simple identifier:

delightql

```
invoice(*), total > 15 : big_invoices
big_invoices(*)
```

> Note: post-labeling can only be used on lower-order relational predicates. Common function expressions and higher-order CTEs must be pre-labeled.

These syntaxes may be intermixed:

delightql

```
us_customers(*): customer(*), country = "USA"
invoice(*), total > 15 : big_invoices
us_customers(*), big_invoices(*.(customer_id))
```

SQL

```
WITH "us_customers" AS (
  SELECT *
  FROM "customer"
  WHERE "country" IS NOT DISTINCT FROM 'USA'
),
"big_invoices" AS (
  SELECT *
  FROM "invoice"
  WHERE "total" > 15
)
SELECT *
FROM "us_customers"
JOIN "big_invoices" USING ("customer_id");
```

2.13.2. Common Function Expressions

Create common function expressions (CFEs) – functions whose name is created for the duration of the query – by pre-labeling with a functional functor. [23] The SHADOW-NECK separates the functional functor on the left from any valid domain expression on the right. CFEs may only be created by pre-labeling.

23 A functional functor is a functor with a colon separating the identifier from the opening parenthesis.

delightql

```
enweirden:(total) :
  total >> :(@ - 5) >> max:(0) >> min:(10)

invoice(*) |> (enweirden:(total) as silly, total)
```

SQL

```
SELECT
  min(max("total" - 5, 0), 10) AS "silly",
  "total" AS "total"
FROM "invoice";
```

CTEs and CFEs may be intermixed:

delightql

```
double:(x) : (x * 2)
invoice(*), total > 5 : over_five
triple:(y) : (y * 3)
mid_range(*): over_five(*), total < 10
mid_range(*)
  |> ( invoice_id,
       billing_city,
       total,
       double:(total) as doubled,
       total >> double:() >> double:() as quadrupled,
       triple:(total) as tripled,
       double:(triple:(total)) as sextupled)
```

SQL

```sql
WITH over_five AS (
    SELECT
        *
    FROM invoice
    WHERE total > 5
),
mid_range AS (
    SELECT
        *
    FROM over_five
    WHERE total < 10
)
SELECT
    invoice_id,
    billing_city,
    total,
    (total * 2) AS doubled,
    ((total * 2) * 2) AS quadrupled,
    (total * 3) AS tripled,
    ((total * 3) * 2) AS sextupled
FROM
    mid_range;
```

2.14. CLAUSE-LOCAL PARAMETERS

A value function's clauses bind actuals by argument position. Each clause chooses its own scalar names; arity, roles and context capture must agree. This applies to common functions and consulted functions alike.

delightql

```
f:(x | x > 1) : x * 10
f:(y) : y + 100
_(v @ 1; 3) |> (f:(v) as r)
```

The results are 101 and 30. The guard and computation in each clause use that clause's names. Scalar parameter metadata identifies argument positions.

2.14.1. Recursive Common Table Expressions

A common table expression becomes recursive when one of its defining clauses references the CTE being defined. Delightql detects this self-reference and emits WITH RECURSIVE.

Sequence generation:

delightql

```
_(n @ 1) : nums
nums(*), n < 100 |> (n + 1 as n) : nums
nums(*) ~> sum:(n)
```

```sql
WITH RECURSIVE nums(n) AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n + 1 FROM nums WHERE n < 100
)
SELECT sum(n) FROM nums;
```

The first clause seeds the CTE with 1. The second clause references nums and increments until the condition fails. This is the standard pattern: base case, then recursive case with termination condition.

Multiple clauses accumulate:

Non-recursive CTEs can have multiple clauses that combine via UNION ALL:

```delightql
customer(*) |> (first_name, last_name) : names
employee(*) |> (first_name, last_name) : names
names(*)
```

When any clause references the CTE name, the entire CTE becomes recursive:

```delightql
_(x @ 1) : nums
_(x @ 100) : nums
nums(*), x < 200 |> (x + 1 as x) : nums
nums(*)
```

Here, two anchor clauses (starting at 1 and 100) seed the recursion. Both sequences grow until reaching 200.

2.14.2. Common Higher-order Table Expressions

Higher-order expressions may be used as common expressions.

```delightql
above(amt,|(*))(*) : |(*), $.amt<total
below(amt,|(*))(*) : |(*), $.amt>total
mark_as(l,|(*))(*) : |(*) ~> count:(*) as count |> +($.l as
label)
invoice(*)
  |> above(1)(*)
  |> below(5)(*)
  : one_to_five
invoice(*)
  |> above(5)(*)
  |> below(10)(*)
  : five_to_ten
invoice(*)
  |> above(10)(*)
  : ten_up

one_to_five(|>mark_as("one_to_five")(*)) ;
ten_up(|>mark_as("ten_up")(*)) ;
five_to_ten(|>mark_as("five_to_ten")(*))
```

2.14.2.1. Recursion

A common higher-order expression may reference itself. Its head declares the name before any clause body is read, so a clause can call the expression it belongs to. A self-reference with the same arguments is a fixpoint, exactly as it is for the same rule written in a rules file: the clauses without a self-reference are the anchor, the others are applied to the previous iteration's rows until none are produced, and the clauses accumulate by `UNION ALL`.

```delightql
reports_to(boss)(employee_id) : employee(*), employee_id = $.boss
reports_to(boss)(employee_id) :
    employee(*) as e,
    reports_to($.boss)(*) as r,
    e.reports_to = r.employee_id
    |> (e.employee_id)
reports_to(2)(*)
```

The argument selects one fixpoint and stays fixed while it runs: `reports_to(2)` inside its own body reads the rows found so far for employee 2. The rules of recursion are the ones in the Recursion in Rules chapter, whichever form the definition is written in:

- A base clause comes first. A self-reference before any base clause refuses as `semantic/recursion/anchor_first`.
- A self-reference with a different argument refuses as `semantic/recursion/parameter-widening`. State that changes from step to step belongs in the relation's columns, as `v` does below.
- A clause references the expression at most once (`semantic/recursion/nonlinear`).

The argument may be any value, including a column of the caller's rows, and the expression may be handed to another rule as a rule value; it recurses the same way in every position.

```delightql
n:(x | (x % 2) = 0) : x / 2
n:(x) : (3 * x) + 1
collatz(s)(*) : _(v @ $.s)
collatz(s)(*) : collatz($.s)(*), v != 1 |> (n:(v) as v)
collatz(10)(*)
```

The badge `%` after the name makes the clauses accumulate by `UNION` instead, as it does on a rule: a row already found is not found again, so a traversal over a cycle ends. Every clause wears the same badge.

```delightql
edge(*) : _(src, dst @ "a", "b"; "b", "a"; "b", "c")
reach%(s)(node) : _(node @ $.s)
reach%(s)(node) :
    reach($.s)(*),
    edge(*) as e,
    e.src = node
    |> (e.dst as node)
reach("a")(*)
```

2.14.3. Common Sigma Expressions

Sigma predicates may be defined as common expressions for use in the query.

delightql

```
outside(x) : x < 0
outside(x) : x > 10
_(x @ -1; 5; 11), +outside(x) |> #(x)
```

Like rule-defined sigma predicates, the common sigma expressions are detected by the compiler via classification of the clause body – the expression defines a sigma expression when the body (inclusive of all clauses if applicable) is a scalar predicate.

SQL

```
SELECT
  anon_1.x AS x
FROM (
  SELECT -1 AS x
  UNION ALL
  SELECT 5
  UNION ALL
  SELECT 11
) AS anon_1
WHERE ((anon_1.x < 0 OR anon_1.x > 10)) IS TRUE;
```

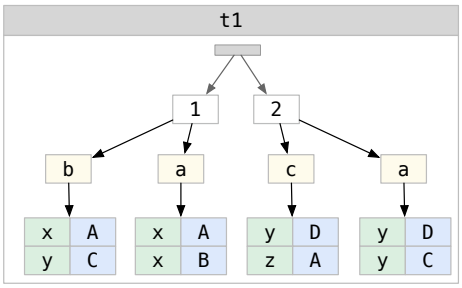
2.15. TREE GROUPS

Tree grouping transforms flat relations into nested JSON structures. Each nesting level corresponds to a `GROUP BY` – the tree's shape reflects the grouping hierarchy.

foo

c1	c2	c3	c4
1	a	x	A
1	b	x	A
1	a	x	B
1	b	y	C
2	a	y	D
2	a	y	D
2	c	y	D
2	c	z	A

tree group



Delightql provides this capability through compound data constructors (`{ }`, `[ ]`) used in reduction positions. The resulting JSON is not general-purpose – it maps relations to a tree normal form where each level represents a distinct grouping context.

Two forms exist:

- Data-oriented: produces arrays of objects; grouping columns become object fields

- Metadata-oriented: produces objects with data values as keys; a single column's values become the key names

Full JSON functionality remains available through the target SQL's native functions (`json_object`, `json_array`, etc.).

2.15.1. Compound Data Constructors (Recap)

Constructor	Scalar Position	Aggregate Position
`{ }`	Record (string-indexed)	Table of records
`[ ]`	Tuple (numeric-indexed)	Table of tuples

Table 17: Compound data constructors by position (recap)

2.15.2. Aggregation boundaries and computed members

After `~>`, `{…}` and `[…]` are collecting aggregates. Grouping keys to the left select their input groups; explicit keys do not turn a collector into a scalar summary record. Ordinary scalar expressions may occur in its members:

delightql
```
invoice(*) ~> { billing_country, "upper": upper:
(billing_country) } as report
```

An ordinary aggregate inside a collected row refuses, with or without keys:

delightql
```
invoice(*) ~> { billing_country, "revenue": sum:(total) }
```

delightql
```
invoice(*) |> %(billing_country ~> { "revenue": sum:(total) })
```

Write sibling aggregates, or explicitly stage the first result:

delightql
```
invoice(*) |> %(billing_country ~> { invoice_id, total } as items,
sum:(total) as revenue)
invoice(*) |> %(billing_country ~> sum:(total) as revenue) ~>
{ billing_country, revenue } as report
```

A window expression supplies one value per input row and therefore composes inside a collector, including through scalar expressions:

delightql
```
invoice(*) ~> { invoice_id, "running": sum:(total <~
%(billing_country), #(invoice_id)) } as report
```

Its meaning agrees with computing `running` in an earlier embed stage and then collecting `{invoice_id, running}`. Required SQL stages realize the authored window; they do not invent grouping keys. Nested tree grouping retains its explicit hierarchy, as described below.

2.15.3. Tree Group Syntax

Nested tree groups are created by nesting compound constructors with `~>` introducing each level:

delightql

```
employee(*)
  ~> { title,
       "people": ~> {first_name, last_name},
       city } as people_by_title_and_city
```

Reading the syntax. The `~>` marks tree group boundaries. Columns between a `~>` and either the next `~>` or a closing enclyph (`}`, `]`, `)`) belong to that level's group:

delightql

```
employee(*)
// level 1          level 2        L2 end   L1 end
// start            start
// ↳                ↳              ↱        ↱
   ~> { title, "people": ~> {first_name, last_name},  city }
```

- **`title`** and **`city`** belong to level 1 (the top-level tree group)
- **`first_name`** and **`last_name`** belong to level 2 (nested within level 1)

The grouping is hierarchical: level 2 groups are computed within each distinct combination of level 1 columns.

2.15.3.1. Terminology

- tree group: The set of columns whose distinct combinations form one level of the tree
- tree group variables: The columns belonging to a tree group
- nested tree group: A tree group inside another tree group
- tree group induction: Using a compound constructor in reduction position to create an interior table

2.15.4. Data-Oriented Tree Grouping

Data-oriented tree grouping uses `~>` followed by a compound constructor. Within each stated group it collects objects or tuples from the input rows. Nested boundaries state the grouping hierarchy; a flat collector does not deduplicate its members merely because their values agree.

Simple example:

delightql

```
employee(*)
  |> %( city ~> { title, state } as title_and_state)
```

The tree grouping variables above are **`city`** alone. Within each of these groups **`{title, state}`** do not require distinctness.

Simple example:

delightql

```
employee(*)
  ~> { title, state } as title_and_state
```

Returns one row containing an array of all `{title, state}` combinations. No grouping variables mean this is a whole-group tree group and the number of rows in the starting table will equal the number of rows in the array.

Nested example:

delightql

```
employee(*)
  ~> { title,
       "people": ~> {first_name, last_name},
       city } as people_by_title_and_city
```

Returns a single-row, single-column table:

people_by_title_and_city


```
[
 { "title": "General Manager",
   "city": "Edmonton",
   "people": [
     { "first_name": "Andrew", "last_name": "Adams" }
   ]
 },
 { "title": "IT Manager",
   "city": "Calgary",
   "people": [
     { "first_name": "Michael", "last_name": "Mitchell" }
   ]
 },
 { "title": "IT Staff",
   "city": "Lethbridge",
   "people": [
     { "first_name": "Robert", "last_name": "King" },
     { "first_name": "Laura", "last_name": "Callahan" }
   ]
 },
 { "title": "Sales Manager",
   "city": "Calgary",
   "people": [
     { "first_name": "Nancy", "last_name": "Edwards" }
   ]
 },
 { "title": "Sales Support Agent",
   "city": "Calgary",
   "people": [
     { "first_name": "Jane", "last_name": "Peacock" },
     { "first_name": "Margaret", "last_name": "Park" },
     { "first_name": "Steve", "last_name": "Johnson" }
   ]
 }
]
```


Table 18: {#tbl:array-tree-group}

Transpilation. Tree grouping uses JSON aggregation functions as intermediates:

SQL

```
SELECT
  json_group_array(
    json_object(
      'title', title,
      'city', city,
      'people', people
    )
  ) AS people_by_title_and_city
FROM (
  SELECT
    title,
    city,
    json_group_array(
      json_object('first_name', first_name, 'last_name',
last_name)
    ) AS people
  FROM employee
  GROUP BY title, city
);
```

The nested `GROUP BY` mirrors the nested `~>`. Each tree group level becomes a subquery with its own grouping and JSON aggregation. The JSON functions are implementation details – the result is a standard column containing structured data.

Three-level example:

delightql

```
customer(*)
  ~> { country,
       "customers_by_city":
         ~> { city,
              "customers": ~> {first_name, last_name} } }
    as customers_by_city_within_country
```

Groups first by `country`, then within each country by `city`, then collects customers within each city.

Sibling tree groups:

Multiple nested groups at the same level share their parent's context but are otherwise independent:

delightql

```
customer(*)
  ~> { country,
       "customers_by_city": ~> { city, "customers": ~>
{first_name, last_name} },
       "reps": ~> [support_rep_id] }
    as nested_with_siblings
```

The `customers_by_city` and `reps` tree groups are siblings – both nested within `country`, neither containing the other.

Sibling tree groups share their parent's context but aggregate independently. The relationship between siblings—which customer had which support rep – is not preserved. This is inherent to the structure: siblings represent independent projections of the grouped data. [24]

24 Trees with siblings satisfy TNF-G but not TNF-R; they cannot round-trip losslessly. (See Appendix A.)

2.15.5. Metadata-Oriented Tree Grouping

Metadata-oriented tree grouping elevates data values to JSON keys. A column's distinct values become the keys of a single object rather than elements of an array.

The syntax uses `:~>` after a bare identifier:

delightql
```
employee(*)
  ~> title: ~> {first_name, last_name} as people_by_title
```

The result is an interior record (one object), not an interior table (array of objects):

```
{
  "General Manager": [
    { "first_name": "Andrew", "last_name": "Adams" }
  ],
  "IT Manager": [
    { "first_name": "Michael", "last_name": "Mitchell" }
  ],
  "IT Staff": [
    { "first_name": "Robert", "last_name": "King" },
    { "first_name": "Laura", "last_name": "Callahan" }
  ],
  "Sales Manager": [
    { "first_name": "Nancy", "last_name": "Edwards" }
  ],
  "Sales Support Agent": [
    { "first_name": "Jane", "last_name": "Peacock" },
    { "first_name": "Margaret", "last_name": "Park" },
    { "first_name": "Steve", "last_name": "Johnson" }
  ]
}
```

Distinguishing syntax:

- Normal keys are quoted strings: `"people":`
- Metadata keys are bare identifiers followed by `:~>`: `title: ~>`

Restriction: Only one column can serve as a metadata key per level – the object can have only one set of keys. This constraint reflects JSON's structure: two metadata-keyed objects with the same key type would create ambiguous destructuring. Metadata-oriented trees satisfy TNF-M. (See Appendix A.)

Within a regular group by:

delightql
```
employee(*)
  |> %( city
         ~>
       title: ~> {first_name, last_name} as people_by_title )
```

Returns one row per city, each containing an object keyed by title.

2.15.6. Tree Distinction

Tree structures can serve as grouping columns, enabling aggregation alongside hierarchical output:

delightql

```
invoice(*)
  |> %( { billing_country,
          "invoices": ~> {invoice_id, invoice_date},
          billing_city } as invoices_by_country_and_city
          ~>
        sum:(total), count:(*) )
```

Other grouping columns may stand beside the tree. Each one splits the groups further, so every tree holds only its own group's rows – here, one tree per country, city and customer:

delightql

```
invoice(*)
  |> %( { billing_country, "invoices": ~> {invoice_id,
invoice_date}, billing_city } as tree,
        customer_id
          ~>
        sum:(total), count:(*) )
```

2.15.7. Tree Destructuring

Tree destructuring is the inverse of tree grouping – it flattens nested JSON back into rows.

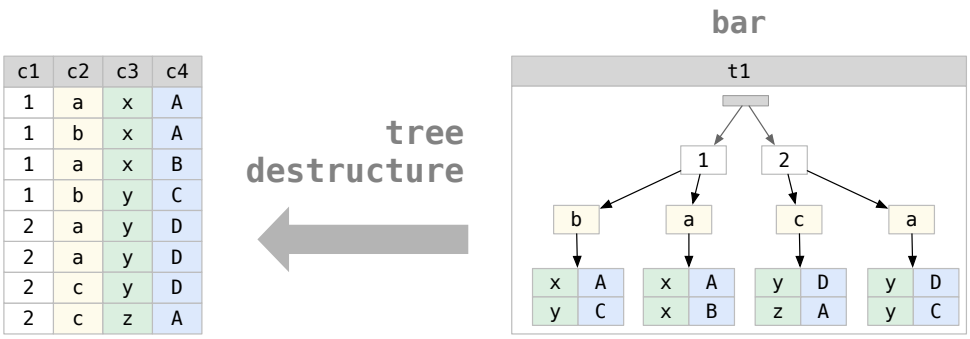


Figure 1: group and destructure

The TREE-UNIFY sigil `~=` matches a JSON column against a destructuring pattern:

delightql

```
partner_sale(*)
  , payload ~= { partner_order,
                 "buyer": { email, country },
                 "items": ~> { track_id, price, quantity } }
  |> -(payload)
```

The pattern syntax mirrors construction syntax. Each `~>` level multiplies rows – the result is the Cartesian product of all nested arrays.

Array vs object matching:

delightql

```
// Matches an ARRAY of objects -- multiplies rows by array length
partner_sale(*) |> (sale_id, payload:{.items} as items)
  , items ~= ~> { track_id, quantity }

// Matches a single OBJECT -- extracts fields, no multiplication
partner_sale(*) |> (sale_id, payload:{.buyer} as buyer)
  , buyer ~= { email, country }
```

The `~>` in destructuring means "iterate over this array," just as in construction it means "aggregate into this array."

Renaming during destructuring:

The string key matches the JSON; the identifier after `:` names the output column:

delightql

```
partner_sale(*)
  , payload ~= { partner_order,
                 "buyer": { email, country },
                 "items": lines }
  |> -(payload)
```

Here `"items"` matches the JSON key; `lines` becomes the column name. The `lines` column contains the nested array as-is, not destructured.

Staged destructuring:

Destructure incrementally by chaining `~=` operations:

delightql

```
partner_sale(*)
  , payload ~= {partner_order, "items": lines}
  , lines ~= ~> {track_id, price, quantity}
  |> -(payload)
```

The first `~=` extracts `partner_order` and keeps `lines` as a JSON array. The second destructures `lines` into individual rows. Stop at any level to preserve nested structure.

Metadata-oriented destructuring:

The `:~>` syntax works symmetrically – object keys become column values:

delightql

```
partner_sale(*) |> (sale_id, payload:{.payout} as payout)
  , payout ~= country: ~> _
  |> -(payout)
```

Given `payout`, an object keyed by country codes, this extracts the key into a `country` column; `_` disregards the amount under each key.

Binding semantics:

Column names in the pattern match JSON keys by name. If the pattern says `email` and the JSON has `"email"`, they bind. A mismatched name produces nulls – there is no compile-time validation against JSON structure.

2.15.8. Pathing in Tree Patterns

Destructuring patterns support direct pathing, eliminating the need to match intermediate structure. The pathing syntax (`.path.to.field`) reaches into nested JSON without declaring every level.

Basic pathing:

```delightql
_(json @ {"name": "app", "config": {"server": {"port": 3000}}})
  |> (json:{.config.server.port})
```

The path `.config.server.port` extracts the value directly.

Pathing in destructuring:

Instead of matching the full structure:

```delightql
partner_sale(*), payload ~= { partner_order, "buyer": { email,
country } }
```

Path directly to what you need:

```delightql
partner_sale(*), payload ~= {
  partner_order,
  .buyer.email,
  .buyer.country
}
```

Pathing with rename:

Combine pathing with `as` to name the output column:

```delightql
partner_sale(*) |> (sale_id, payload:{.items} as items)
  , items ~= ~> {
      track_id,
      .rights.territories as territories
    }
```

Mixed matching and pathing:

Structural matching and pathing can combine in a single pattern:

```delightql
partner_sale(*), payload ~= {
  partner_order,
  payout,
  .buyer.email,
  .buyer.country
}
```

Here `partner_order` and `payout` match top-level keys directly; the `.buyer.*` paths reach into nested structure.

Pathing in projection:

Pathing works outside destructuring patterns, in normal projection:

delightql

```
_(json @ {"name": "app", "scripts": {"dev": "next dev",
"build": "next build"}})
  |> ({
    "name": json:{.name},
    "scripts": json:{.scripts}
  })
```

2.15.9. Interior Drill-Down

Tree destructuring with `~=` requires the user to spell out the interior schema – every level of nesting must be declared in the pattern. When the schema is statically known (tree groups from a view, CTE, or inline query), `.column(*)` provides a shorter, self-documenting alternative.

Syntax. `.column_name(*)` as a suffix on any relation expression. The `(*)` means "all columns of the interior relation." Argumentative access is also supported: `.entities(name, type)` binds the interior relation's columns positionally, with the arity of the interior — the same Prolog-style relation access as `employees(id, name)`. It is relation access, never a projection list: to keep a subset of a wider interior, expand with `(*)` and project ordinarily, or use brace narrowing. The operator is chainable: `.entities(*).columns(*)`.

Context carry-forward. Outer columns remain available after a drill-down. `.entities(*)` produces entity-level columns plus all columns from the enclosing level, minus the exploded column itself. This is lateral-join semantics – each interior row inherits the context of its parent row.

Cardinality. Expansion is correlated: each parent row contributes one output row per row of ITS interior, so the total is the sum over parent rows of `cardinality(r.t)`. Duplicate interior rows are preserved. A NULL or empty interior IS empty — it contributes ZERO rows, in every expansion form; interior expansion is not an outer join, and a parent with no children vanishes rather than surviving as a row of NULL children. (This is the expansion half of the round-trip law: construction elides all-NULL contributor rows to `[]`, and `[]` expands to nothing.)

Post-pipe narrowing. The same parenthesized access in post-pipe position performs the SAME correlated expansion and retains only the interior heading:

delightql

```
// expand staff; keep City beside each child
employee(*) |> %(city ~> {last_name, title} as staff).staff(*)
// the same expansion; keep only staff's columns
employee(*) |> %(city ~> {last_name, title} as staff) |> .staff(*)
// argumentative narrowing (positional bind)
employee(*) |> %(city ~> {last_name, title} as staff) |
> .staff(name, title)
```

The two forms agree on row count, duplicate multiplicity, parent/child correspondence, and empty/NULL interiors; they differ only in retained context — `… |> .staff(*)` is exactly `….staff(*) |> (staff.*)`. Parenthesized access — postfix or post-pipe, glob or argumentative — requires a statically known tree-group interior; over external JSON every parenthesized form is refused, because the compiler cannot discover keys at runtime and plan a heading from them. External JSON narrows with braces, whose members are the programmer's static heading witness (declared fields become the planned columns; extra runtime keys never do; a missing declared key yields NULL without suppressing the row).

Example – CTE drill-down:

delightql
```
customer(*) |> %(country ~> {first_name, last_name} as people) :
by_country
by_country(*).people(*)
```

This produces one row per person, with `country` carried forward from the grouping level.

Example – chained drill-down:

delightql
```
main::(*).entities(*).columns(*)
  , entities.name = "employee"
```

Each `.name(*)` step explodes one level of nesting. Columns from all prior levels remain available for filtering.

Equivalence with `~=`. The same query written both ways:

delightql
```
// Drill-down form:
main::(*) |> (entities) .entities(*)

// Equivalent ~= form:
main::(*)
  , entities ~= ~> {name, type, doc, "columns": ~> {col_name,
col_type, col_pos}}
  |> -(entities)
```

The drill-down form does not require the user to know the interior schema.

2.15.10. Narrowing

The `~=` operator and interior drill-down both carry context forward – outer columns survive into the result. This is the correct default for relational composition, but it requires projecting out the intermediate columns when they are no longer needed:

delightql
```
partner_sale(*), payload ~= {.items} |> -(payload)
  , items ~= ~> {.track_id, .price, .quantity} |> -(items)
```

When the intent is to drill into a column, extract fields, and discard everything else, the `.column{...}` operator expresses this more efficiently:

```delightql
partner_sale(*)
  |> (payload:{.items} as items)
  |> .items{.track_id, .price, .quantity}
```

Each step replaces the current row with the destructured result.

When to use which.

Form	Carries context	Use case
`~= pattern`	Yes	General relational destructuring; join with outer columns
`.col(*)`	Yes	Drill-down when schema is known; outer columns needed
`|> .col(*)`	No	Same expansion as drill-down, interior heading only (schema-known)
`.col{...}`	No	Navigate into nested JSON; only interior fields matter — and the REQUIRED form for external JSON (static heading witness)

Example – partner sale items:

```delightql
partner_sale(*)
  |> (payload:{.items} as items)
  |> .items{.track_id, .price, .quantity}
```

The path extraction `payload:{.items}` pulls the items array out of the top-level object; then `.items{...}` iterates and extracts fields. The result is a flat table with `track_id`, `price`, and `quantity` columns – no intermediate columns to clean up.

2.15.10.1. Null Elision in Tree Groups

When an outer join feeds into a tree group, the join pads unmatched rows with NULLs. In a flat relation this is the only way to represent "no match" – NULL serves as a sentinel for absence. Trees have no such limitation: an empty array `[]` directly represents "no children."

Tree groups decode this flat-world encoding. When all value columns in a contributing row are NULL, the row is elided from the array. The result is `[]`, not `[{"col": null, ...}]`.

Example:

```delightql
parents(*) : _(id, name ---- 1, "Alice"; 2, "Bob"; 3, "Charlie")
children(*) : _(parent_id, toy ---- 1, "doll"; 1, "ball"; 2,
"car")

parents(*), children?(*), parents.id = children.parent_id
  |> %(name ~> {toy} as toys)
```

Charlie has no children. The outer join produces `(3, "Charlie", NULL)`. Without null elision, the tree group would produce:

name	toys
Alice	[{“toy”:“doll”},{“toy”:“ball”}]
Bob	[{“toy”:“car”}]
Charlie	[{“toy”:null}]

With null elision:

name	toys
Alice	[{“toy”:“doll”},{“toy”:“ball”}]
Bob	[{“toy”:“car”}]
Charlie	[]

Scope. Null elision applies to both forms:

- `~> {a, b, c}` – curly (object) tree groups
- `~> [a, b, c]` – bracket (tuple) tree groups

The elision rule. A row is elided when every value column in the constructor is NULL. If any value column is non-null, the row is preserved. This means a row like `{"name": "Alice", "age": null}` survives – only fully-null rows (the signature of outer-join padding) are dropped.

Parent entity preservation. Null elision never removes the parent entity. Charlie still appears in the result – only the contents of the nested array change (from `[{"toy":null}]` to `[]`). The GROUP BY produces a row for Charlie’s grouping key; the aggregate column becomes an empty array rather than an array containing a null-valued object.

Round-trip behavior. Destructuring an empty array produces zero rows. After null elision, destructuring Charlie’s `[]` eliminates Charlie from the flat result – which is the same behavior as an inner join.

2.16. SET OPERATORS

SQL’s set operators UNION, INTERSECT and EXCEPT all operate on union-compatible schemas with ordinality-based matching. This means that to use these operators in SQL you need to arrange an enumeration of columns from each table via position and type only, and that the number of columns must be the same.

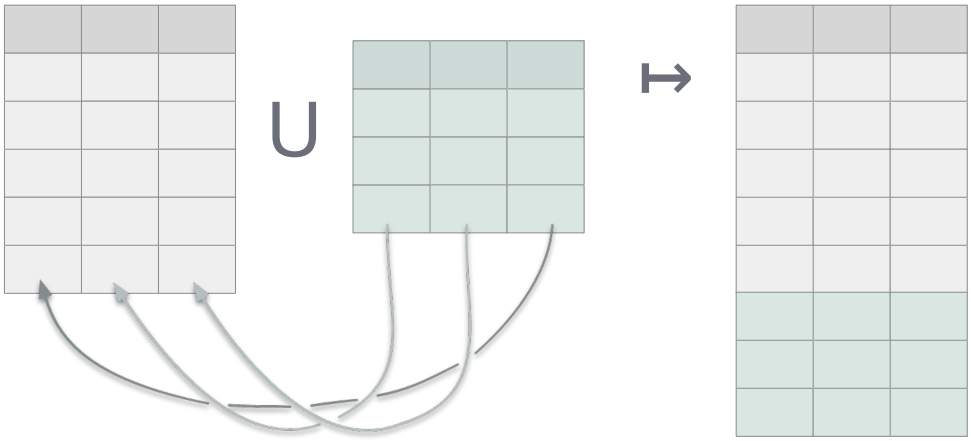


Figure 2: Union Compatability via Ordinal Alignment

It is a somewhat interesting corner of SQL as nowhere else (with the exception of the occasional `group by 2`) does SQL use the ordinality of a relation's columns.

Delightql, of course, supports all of these operators but opens up the door to other functionality by

1. relaxing the same number requirement and permitting ragged unions and intersects
2. having a preference for naming-based access

To make this more concrete, consider delightql's UNION CORRESPONDING:

```delightql
customer(*) ; employee(*)
```

The semicolon sigil `;` of UNION CORRESPONDING separates two tables much like a comma would for joining. With this operator columns are matched by their names and a super set of both tables' columns are synthesized.

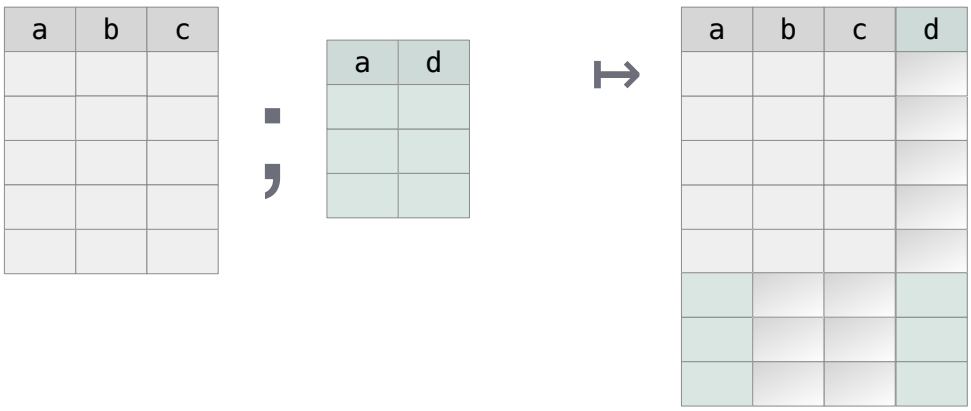


Figure 3: Union Compatability via Name Alignment With OUTER

Delightql re-introduces[25] this CORRESPONDING name-based alignment mode as an alternative to SQL's normal UNION-like ordinal alignment. Delightql prefers name-based alignment but still offers ordinal alignment for backwards compatibility.

25 The SQL standard actually defines a `CORRESPONDING BY` clause (SQL-92 (ISO/IEC 9075:1992)) that can be used with `INTERSECT`, `UNION`, and `EXCEPT`, though, as usual, few actually implemented it. It's even a reserved word.

Delightql's UNION-like operators are the following:

Mode	Sigil	Alignment	Schema requirement
Corresponding	`;`	By name	Any (NULL-padded)
Smart	`|;|`	By name	Identical names and same column count
Positional	`||`	By ordinal	Same column count

Table 22: Set operator alignment modes

2.16.1. Union All Corresponding (`;`)

Aligns by name, NULL-padding missing columns. Output schema: first relation's columns, then non-overlapping columns from the second.[26]

[26] This is closer in definition to OUTER UNION. The few SQLs that implement UNION ALL CORRESPONDING do so by outputting the intersection of the two column sets, instead of the union of the two column sets that delightql favors.

delightql

```
_( a,b,c
   -------
   1,2,3;
   4,5,6)
   ;
_( d,   a,b
   -------
   "foo",10,20;
   "bar",40,50)
```

a	b	c	d
1	2	3	NULL
4	5	6	NULL
10	20	NULL	foo
40	50	NULL	bar

Union All Corresponding is a ragged union.

2.16.2. Smart Union All (`|;|`)

Aligns by name, but requires both relations to have identical column count and names. Unlike SQL's `UNION/UNION-ALL` position is irrelevant. The resulting schema is adopted from the first relation:

delightql

```
genre_2024(*)
  |;|  genre_2025(*)
```

SQL

```
SELECT
  genre_id, name
FROM genre_2024
UNION ALL
SELECT
  genre_id, name  -- genre_2025 stores (name, genre_id)
FROM genre_2025;
```

2.16.3. Positional (Traditional) Union All (||)

Aligns by position. Requires identical column count. Useful for intentional realignment.[27]:

[27] The below example uses interior relations to shape each relation prior to the `UNION ALL`.

```delightql
invoice(|> (billing_address,billing_city,billing_country))
  ||
customer(|> (address,city,country))
```

2.16.4. Set Semantics vs Multiset Semantics

All of delightql set operators are actually multiset operators inasmuch as they preserve duplicates. In other words, all set operators are **`ALL`**-flavored.

If set semantics are required, use **`DISTINCT ALL`** via `|> %(*)`.

```delightql
genre_2024(*) |;| genre_2025(*) |> %(*)
```

```SQL
SELECT
  genre_id, name
FROM genre_2024
  UNION   --- NOT UNION ALL
SELECT
  genre_id, name
FROM genre_2025;
```

which is equivalent to

```SQL
SELECT DISTINCT * FROM
  (SELECT
    genre_id, name
  FROM genre_2024
    UNION ALL
  SELECT
    genre_id, name
  FROM genre_2025)
;
```

2.16.5. Intersects via correlation

Having introduced the operators above, one would assume that a new sigil for intersects is in the offing. Instead delightql chooses to reuse the same syntax for correlations to represent a statement of which columns to intersect on.

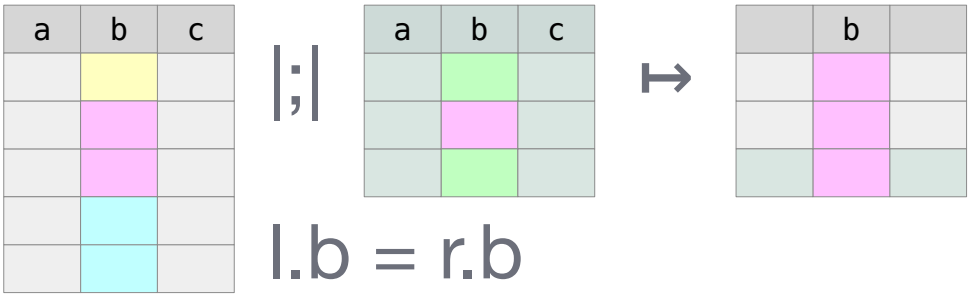


Figure 4: Intersect ON via correlation conditions

After any union-flavored multiset operator, conjoin a condition that correlates the previous relations together. From such a union an intersection results:

```delightql
genre_2024(*) as g1 |;|
  genre_2025(*) as g2,
  g1.genre_id = g2.genre_id
```

SQL's `INTERSECT` only matches on the entire tuple, it lacks an `INTERSECT ON/BY` parameterization. Delightql's correlation syntax lets you choose which columns to match on – a per-column intersection that SQL cannot express without rewriting the query as a pair of `EXISTS` subqueries.

2.16.5.0.1. Correlation syntax matches alignment mode

The correlation condition should use the same addressing schema as the alignment:

- Name-based modes (`;`, `|;|`) use name-based correlation: `x.col = y.col` or `x.* = y.*`
- Positional mode (`||`) uses positional correlation: `x|1| = y|1|` or `x|*| = y|*|`

The full-tuple shorthand `x.* = y.*` means "match on all column names that appear in both x and y." Columns present on only one side are ignored for matching. Under `|;|` this distinction is moot since the schemas are identical. Under `;` the schemas may be different, and matching on the intersection of names is the only natural reading.

The positional shorthand `x|*| = y|*|` means "match on all column positions."

DQL	Equivalent SQL concept

`delightql`
`employee_2024(*) as x ;`
`  employee_2025(*) as y,`
`  x.* = y.*`
	INTERSECT ALL CORRESPONDING

DQL	Equivalent SQL concept

delightql
`genre_2024(*) as x |;|`
`  genre_2025(*) as y,`
`  x.* = y.*`
	INTERSECT ALL
	(name safe)

delightql
`employee_2024(*) as x ||`
`  employee_2025(*) as y,`
`  x|*| = y|*|`
	INTERSECT ALL
	(positional, = SQL's `INTERSECT ALL`)

delightql
`employee_2024(*) as x ||`
`  employee_2025(*) as y,`
`  x|1| = y|1|`
	Per-column intersection (positional, no SQL equivalent)

delightql
`genre_2024(*) as x |;|`
`  genre_2025(*) as y,`
`  x.genre_id = y.genre_id`
	Per-column intersection (no SQL equivalent)

Table 24: Intersection as union with correlation

To belabor a point, intersection re-purposes correlation syntax that is seen most often with joins to be useful for intersection. To see the difference between a join and an intersection look at how the two tables prior are combined:

Correlation as JOIN ON	Correlation as INTERSECT ON

delightql
`genre_2024(*) as g1,`
`  genre_2025(*) as g2,`
`  g1.genre_id = g2.genre_id`
	delightql
	`genre_2024(*) as g1 |;|`
	`  genre_2025(*) as g2,`
	`  g1.genre_id = g2.genre_id`

`,` between the two tables produces a join (rows are paired).	`|;|` between the two tables produces an intersection (rows are filtered).

Table 25: Correlation as join versus correlation as intersect

> Equality and NULL-safety. The `=` in both columns above looks identical, but the compilation differs. In join position (left column), `=` compiles to SQL `=` – NULLs do not match, because NULL-to-NULL matching in a join can explode row counts . In set correlation position (right column), `=` compiles to `IS NOT DISTINCT FROM` – NULLs match. This is safe because set correlation filters via `EXISTS`, which tests for the presence of a matching row without multiplying output.

> How intersection is executed in SQL.

Example:

delightql

```
customer(*) as c ; employee(*) as e,
  c.city = e.city
```

SQL

```
SELECT
  customer_id, first_name, last_name, company, address,
city,
  state, country, postal_code, phone, fax, email,
support_rep_id,
  NULL, NULL, NULL, NULL, NULL
FROM customer AS c
  WHERE EXISTS (SELECT 1
    FROM (
      SELECT
        employee_id, last_name, first_name, title,
reports_to,
        birth_date, hire_date, address, city, state,
country,
        postal_code, phone, fax, email
      FROM employee AS e
    ) AS t1
    WHERE c.city IS NOT DISTINCT FROM t1.city)

UNION ALL

SELECT
  NULL, first_name, last_name, NULL, address, city,
  state, country, postal_code, phone, fax, email, NULL,
  employee_id, title, reports_to, birth_date, hire_date
FROM employee AS e
  WHERE EXISTS (SELECT 1
    FROM (
      SELECT
        customer_id, first_name, last_name, company,
address,
        city, state, country, postal_code, phone, fax,
email,
        support_rep_id
      FROM customer AS c
    ) AS t0
    WHERE t0.city IS NOT DISTINCT FROM e.city)
```

2.16.6. Minus (Except)

Minus returns rows from the first relation that have no match in the second. A single operator `-` aligns by name:

delightql

```
genre_2024(*) - genre_2025(*)
```

Rows in `genre_2024` with no corresponding row (by name) in `genre_2025`: Rock And Roll and Easy Listening, although `genre_2025`

stores its columns in the other order. Schemas must align – if column names differ, rename first:

```delightql
employee(|> (employee_id)) - customer(|> (support_rep_id as
employee_id))
```

2.17. INTERIOR RELATIONS AND LATERAL JOINS

Interior relations unify `EXISTS`, `NOT EXISTS`, scalar subqueries, and lateral joins under one syntax.

An interior relation is a query continuation inside a functor's parentheses:

```delightql
customer(|> (last_name,first_name))
```

> Query Continuation
>
> A query continuation extends a complete query rightward.
>
> `users(*)^ , age<50^ |> (department)^`
>
> After `users(*)`, the continuation `, age>50 |> (department)` is valid because `users(*)` alone is already meaningful. Likewise, `|> (department)` is valid because `users(*), age>50` is already meaningful.

Interior relations appear wherever tables are allowed. When uncorrelated, they're equivalent to exterior execution:

```delightql
customer(*) |> (last_name,first_name)
// equivalent to: Customer(|> (last_name,first_name))
```

Consider positional union all `||` where interiority crafts the proper alignment and projection:

```delightql
customer(|> (first_name,last_name,company))
  ||
employee(|> (first_name,last_name,title))
```

Interior relations are used in the following:

- scalar subqueries (regardless of correlation)
- `EXISTS` and `NOT EXISTS`
- simple shadowing subqueries
- correlated (non-scalar) subqueries – i.e. lateral joins

2.17.1. Scalar subqueries

Scalar subqueries use interior notation:

```delightql
invoice(*) as i
    |> (invoice_id,
        billing_country,
        total,
        invoice:( ~> avg:(total)) as avg_total,
        invoice:( , billing_country=i.billing_country
                  ~> avg:(total)) as avg_total_in_country)
```

In the above example, two query continuations started with `~>` and `,` execute an uncorrelated and correlated scalar subquery respectively.

2.17.2. `EXISTS` and `NOT EXISTS`

The `+` and `\+` prefixes with interior notation create `(NOT) EXISTS`:

```delightql
album(*), track(*),
  album.album_id = track.album_id,
  \+invoice_line(, track.track_id = invoice_line.track_id)
```

The `EXISTS` observer asks whether a row matched; it does not change the matching condition. A comparison between an inner and an enclosing row uses correspondence equality (`=` in SQL), including when the same interior is observed as a scalar subquery. A local test within the inner row keeps its ordinary null-safe meaning.

2.17.3. Simple Shadowing

Uncorrelated interior relations are simple shadowing – useful for reshaping before set operations:

```delightql
customer(|> (last_name,first_name))
```

but especially for set operators:

```delightql
customer(|> (first_name,last_name,company))
  ||
employee(|> (first_name,last_name,title))
```

```delightql
employee_2025(|> *(job_title as title,manager_id as reports_to))
  |;|
employee_2024(*)
```

```delightql
employee_2024(; employee_2025(*)) as staff,
  customer(*), staff.employee_id=customer.support_rep_id
  |> (staff.last_name,customer.email)
```

2.17.4. Correlated Table (Lateral Join)

Any table with interiority and correlation to other tables in the same query is a lateral join.

Lateral joins may be broken down into three sub-types:

- simple
- aggregate
- top-N

Simple Lateral. A join without aggregation or limits. Replaces multiple scalar subqueries; rarely advantageous over a regular join.

delightql

```
invoice(*) ,
  customer(, c.customer_id=invoice.customer_id
        |> (last_name,first_name,email)) as c
  |> (invoice.*,last_name,first_name,email)
```

SQL

```
SELECT invoice.*, last_name , first_name , email
  FROM invoice
  INNER JOIN (
    SELECT last_name , first_name , email ,
      customer_id   -- promote out of subquery for joining
    FROM customer
  ) AS c ON c.customer_id = invoice.customer_id;
```

Aggregate Lateral. Replaces multiple aggregate scalar subqueries. Advantageous when the aggregate key matches the join key.

delightql

```
customer(*) as c,
  invoice(, invoice.customer_id = c.customer_id |>
            %(customer_id
              ~> sum:(total) as total_spent,
                 count:(*) as invoice_count))
```

SQL

```
SELECT
  *
FROM customer AS c
  INNER JOIN (
    SELECT customer_id , sum(total) AS total_spent, count(*) AS
invoice_count
    FROM invoice
    GROUP BY customer_id
  ) AS invoice
ON invoice.customer_id = c.customer_id;
```

Top-N Lateral. Returns the top N correlated rows per outer row, avoiding explicit window functions.

delightql

```
customer(*) as c,
  invoice(, invoice.customer_id = c.customer_id |> #(total desc),
#<3)
```

SQL

```
SELECT *
FROM customer AS c
JOIN (SELECT
  invoice_id, customer_id, invoice_date, billing_address,
  billing_city, billing_state, billing_country,
  billing_postal_code, total
FROM (SELECT
  invoice_id, customer_id, invoice_date, billing_address,
  billing_city, billing_state, billing_country,
  billing_postal_code, total,
  ROW_NUMBER() OVER (
    PARTITION BY
      customer_id
    ORDER BY total DESC
  ) AS __dql_rn
FROM invoice) AS invoices_with_rn
WHERE
  invoices_with_rn.__dql_rn <= 3) AS invoice
  ON invoice.customer_id = c.customer_id;
```

Note how the windowing function above partitions by the correlation join condition.

2.17.5. Summary

The below diagram describes the hierarchy of all places where delightql uses interiority.

Only the tree labeled `interior relations` consists of expressions that are used as actual relations/tables.

```
Interiority:
├── interior relations:
│   ├── correlated (lateral):
│   │   ├── top-N
│   │   ├── simple/multi
│   │   └── aggregate
│   └── uncorrelated
├── (not) exists
└── scalar subqueries
    ├── correlated
    └── uncorrelated
```

2.18. HIGHER-ORDER PIPES

The R-PIPE `|>` passes a relation into a unary operator:

delightql

```
track(*), milliseconds > 600000
  |> ( name )
```

Delightql's built-in pipe unary operators – projection, distinct, group by – have dedicated syntax:

```delightql
customer(*)  |>   ( last_name )
customer(*)  |>  -( first_name, last_name )
customer(*)  |>  +( length:(last_name) as length_last_name )
customer(*)  |>  %( first_name, last_name )
customer(*)  |>  %( country ~> count:(*) )
```

Higher-order predicats are programmer-defined rules that can appear as the pipe target. Given this example definition in assertion mode:

```delightql
summarize(T(*))(*) :-
  T(*)
    |> %( ~>  count:(%composer)  as distinct_composer_count,
              count:(%genre_id)   as distinct_genre_count,
              count:(*)           as total_count,
              avg:(milliseconds) as average_milliseconds )
```

a pipe can target it:

```delightql
track(*)
  |> summarize(*)
```

The expression `track(*) |> summarize(*)` expands to:

```delightql
track(*)
    |> %( ~>  count:(%composer)  as distinct_composer_count,
              count:(%genre_id)   as distinct_genre_count,
              count:(*)           as total_count,
              avg:(milliseconds) as average_milliseconds )
```

2.18.1. Piped vs. Direct Invocation

Given this definition:

```delightql
clean_employees(T(*))(*) :-
  T(*)
    |> $(trim:())(last_name, first_name)
    |> $(date:())(birth_date, hire_date)
```

Higher-order predicates can be invoked directly, passing full functor expressions:

```delightql
clean_employees(main.employee(*))(*)
```

This is equivalent to:

```delightql
main.employee(*)
  |> clean_employees(*)
```

Direct invocation accepts any relation expression, including filters and projections:

```delightql
clean_employees(main.employee(*, hire_date > "2003-01-01"))(*)
```

Higher-order parameters are passed by reference: their invocation preserves the written relation without evaluation.

The piped form's advantage is composability with other pipe operators:

```delightql
main.employee(*), hire_date > "2003-01-01",
  title = "Sales Support Agent"
  |> clean_employees(*)
```

2.18.2. Multi-Parameter Piped Invocation

When the piped relation is not the last parameter, use `@` (the f-param placeholder) to mark where it goes – borrowing function-pipe syntax:

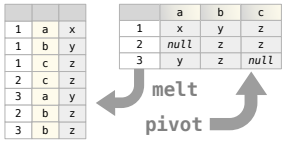
```delightql
// Definition: scalar second, table first
tagged(T(*),label)(*) : T(*) |> +($.label as tag)

// Piped with @:
customer(*) |> tagged(@,"vip")(*)
```

2.19. PIVOT AND MELT

Melting and pivoting are inverse transformations between two table shapes containing the same data. The "long skinny" table stores attributes as data – normalized, often resembling key-value pairs. The "short wide" table lifts attributes to metadata – they become column names. [28]

28



Both transformations are possible in pure SQL given:

1. Support for compound data (JSON objects, arrays)
2. Ability to join against compound data (`unnest`, `json_each`)
3. For pivoting: attribute values must be known at query-write time to become column names

2.19.1. Melt

Melting normalizes denormalized data. A common case: data transfers between organizations where a single row contains multiple relations.

Consider `customer` with three contact columns that should be normalized into separate rows:

delightql

```
customer(*), _( Contact, Channel, channel_number
                ------------------------------
                phone, "phone", 1;
                fax,   "fax",   2;
                email, "email", 3 )
  |> (customer_id, channel_number, Channel, Contact)
```

The anonymous table (lines 1–5) maps each source column set to a row. Joining it to `customer` with no condition produces one output row per contact channel per customer – three times the original cardinality. The projection (line 6) retains only the normalized columns.

The transpiled SQL:

SQL

```
SELECT
  customer_id,
  CASE arm.i WHEN 0 THEN 1       WHEN 1 THEN 2     WHEN 2 THEN 3
END AS channel_number,
  CASE arm.i WHEN 0 THEN 'phone' WHEN 1 THEN 'fax' WHEN 2 THEN
'email' END AS Channel,
  CASE arm.i WHEN 0 THEN phone   WHEN 1 THEN fax   WHEN 2 THEN
email   END AS Contact
FROM customer
CROSS JOIN (
  SELECT 0 AS i
  UNION ALL SELECT 1
  UNION ALL SELECT 2
) AS arm;
```

2.19.2. Pivot

Pivoting is a `GROUP BY` that rotates row-oriented data into columns. The group key defines the entity; an attribute column becomes column names; a value column fills them.

Given `invoice` totals summed per year and billing country (lines 3–4 of the query below):

year	billing_country	sales
2021	Brazil	37.62
2021	Canada	57.42
2021	France	35.64
2021	Germany	53.46
2021	USA	103.95
2022	Brazil	41.6
2022	Canada	76.26
2022	France	39.6
2022	Germany	25.74
2022	USA	102.98

Table 26: Sales per year and billing country (first two years)

A pivot on `billing_country` produces:

year	USA	Canada	France	Brazil	Germany
2021	103.95	57.42	35.64	37.62	53.46
2022	102.98	76.26	39.6	41.6	25.74
2023	103.01	55.44	42.61	19.8	48.57
2024	127.98	42.57	36.66	53.46	18.81
2025	85.14	72.27	40.59	37.62	9.9

Table 27: Pivoted result – countries become columns

delightql

```
1  invoice(*),
2    billing_country in ("USA"; "Canada"; "France"; "Brazil"; "Germany")
3    |> %(strftime:("%Y", invoice_date) as year, billing_country
4          ~> round:(sum:(total), 2) as sales)
5    |> %( year
6           ~>
7         sales of billing_country )
```

- Line 5: `year` defines the entity (group key), determining output cardinality
- Line 2: the `in` clause constrains which attribute values become columns
- Line 7: `sales of billing_country` rotates values into attribute-named columns

The `in` clause is required. Pivoting has compile-time semantics – the output schema is determined by the query, not the data. Without a fixed set of attribute values, the column names would be unknowable.

The transpiled SQL:

SQL

```
WITH _preagg_invoice AS (
  SELECT
    strftime('%Y', invoice_date) AS year,
    billing_country,
    round(sum(total), 2) AS sales
  FROM invoice
  WHERE billing_country IN ('USA', 'Canada', 'France', 'Brazil', 'Germany')
  GROUP BY strftime('%Y', invoice_date), billing_country
)
SELECT
  year,
  max(CASE WHEN billing_country = 'USA' THEN sales END) AS USA,
  max(CASE WHEN billing_country = 'Canada' THEN sales END) AS Canada,
  max(CASE WHEN billing_country = 'France' THEN sales END) AS France,
  max(CASE WHEN billing_country = 'Brazil' THEN sales END) AS Brazil,
  max(CASE WHEN billing_country = 'Germany' THEN sales END) AS Germany
FROM _preagg_invoice
GROUP BY year;
```

2.19.2.1. Multiple Value Columns

A second aggregate, `count:(*) as invoices`, counts each cell's invoices. Multiple `of` clauses pivot additional columns:

year	USA	Canada	...	USA_invoices	Canada_invoices	...
2021	103.95	57.42		17	10	
2022	102.98	76.26		18	12	
2023	103.01	55.44		19	11	
2024	127.98	42.57		21	9	
2025	85.14	72.27		16	14	

Table 28: Pivot with multiple value columns

delightql

```
1  invoice(*),
2    billing_country in ("USA"; "Canada"; "France"; "Brazil"; "Germany")
3    |> %(strftime:("%Y", invoice_date) as year, billing_country
4           ~> round:(sum:(total), 2) as sales, count:(*) as invoices)
5    |> %( year
6            ~>
7          sales of billing_country,
8          invoices of :"{billing_country}_invoices" )
```

Lines 7–8 introduce two pivot column sets. The second uses a format function to distinguish column names (`USA_invoices`, `Canada_invoices`, etc.). When pivoting multiple value columns, the attribute expression after `of` must differ – here, `billing_country` versus `:"{billing_country}_invoices"`.

2.19.3. Pivot Syntax

The `of` keyword rotates values into attribute-named columns. The grammar:

```
<value_column> of <attribute_column>
<value_column> of :<format_string>
```

The attribute column must be constrained by an `in` clause. When pivoting multiple value columns, each `of` expression must produce distinct column names – hence the format string option.

2.20. META-IZE OPERATOR

The meta-ize operator reifies a relation's schema as a relation – each column becomes a row. Where `*` inside a functor returns all data rows, `^` returns all columns as rows of metadata.

2.20.1. Schema as Relation (^)

```delightql
album(^)
```

This returns one row per column in `album`:

scope	column_name	ordinal
album	album_id	1
album	title	2
album	artist_id	3

Table 29: Output of `album(^)`

The `^` operator belongs to the unary continuation operator family – unary operators that transform table access:

Operator	Meaning
`*`	Qualify column names (data access)
`.*`	Unqualified columns (natural join candidate)
`.(cols)`	USING semantics on specific columns
`^`	Column metadata as rows

Table 30: Table continuation operators

These operators compose freely: `album(*.(artist_id))` means "qualified + USING on artist_id."

2.20.2. Postfix Form

`album(^)` is sugar for `album() ^`. The postfix form works on any relational expression, not just base tables:

```delightql
// schema of a projection (2 rows)
album(*) |> (title, artist_id) ^

// schema of a join
album(*), artist(*.(artist_id)) ^

// schema of an aggregation
album(*) |> %(artist_id ~> count:(*) as n) ^
```

The postfix `^` applies to the entire expression to its left, returning its schema as a relation.

2.20.3. Composability

Because `^` produces a regular relation, all DQL operations apply – filtering, projection, pipes, joins, and set operators:

delightql

```
// key columns only
track(^), +like(column_name, "%_id") |> (column_name)
```

delightql

```
// columns shared between two tables
customer(^) as x |;| employee(^) as y, x.column_name =
y.column_name
```

> The output of ^ is itself a relation with a fixed schema (scope, column_name, ordinal). Applying ^ to a ^ result would return the schema of the metadata relation – three rows describing scope, column_name, ordinal themselves.

3. LANGUAGE FUNDAMENTALS

Understanding delightql means internalizing several simple but fundamental choices that are both semantic and syntactic in nature (and often times both).

- Logic variables: delightql borrows Prolog's unification syntax and semantics, though with a subtle definition of a logic variable as being qualified
- Null semantics: delightql opts to provide null-safe equality in contrast to SQL's = operator
- SQL semantics: delightql defines its primary semantics via a direct mapping to SQL
- Evaluation: delightql evaluates left to right in a manner remeniscent of concatenative languages and eschewing parenthesization
- Column Scoping: delightql scopes left to right with well-known scope barriers that delimit what sections of the code may access which logic variables
- Entity Scoping: delightql has certain rules about what scope contexts may or may not shadow existing entities with the same name
- Column access: delightql makes strong choices about how a programmer may access columns by always providing ordinal access, and forbidding named access to ambiguous or generated columns
- Moding rules and currying with higher-order rules: delightql, like Prolog, has a mechanism to mark syntax as moded for input vs output

3.1. UNIFICATION AND LOGICAL VARIABLES

Delightql inherits from Prolog a simple rule: identifiers unify when their names match exactly. To unify means to insist on equality, i.e. the two variables at different locations must refer to the same datum. Unlike Prolog, however, delightql's identifiers are qualified by their table names.

Delightql also uses Prolog's argumentative unification syntax, where the position of an argument in a functor unifies whatever is written in this place: a ground, or another LVar with the dimension accessed at that location.

3.1.1. How Names Are Introduced

The way you access a table determines the names of its columns in scope:

Access Pattern	Columns Introduced
`users(id, name, status, _)`	`id`, `name`, `status`

Access Pattern	Columns Introduced
`users(*)`	`users.id`, `users.name`, `users.status`
`users(*) as u`	`u.id`, `u.name`, `u.status`

Table 31: Columns introduced by access pattern

Argumentative access introduces unqualified names – bare identifiers. Wildcard access introduces qualified names – prefixed by table name or alias.

3.1.2. Unification Creates Joins

When the same name appears in multiple places, unification creates a join condition:

```delightql
artist(artist_id, name), album(album_id, title, artist_id)
```

Both introduce `artist_id`. Unification produces:

```SQL
SELECT artist.artist_id, artist.name, album.album_id,
album.title
FROM artist, album
WHERE artist.artist_id = album.artist_id;
```

3.1.3. Wildcard Access and Qualification

```delightql
artist(*), album(*)
```

This introduces `artist.artist_id` and `album.artist_id` – different names. No unification occurs; the result is a cross join.

To join with wildcard access, use explicit conditions:

```delightql
artist(*), album(*), artist.artist_id = album.artist_id
```

Or use the USING operator `.(cols)`:

```delightql
artist(*), album(*.(artist_id))
```

3.1.4. Qualified References in Argumentative Access

Argumentative patterns can reference lvars from other tables:

```delightql
artist(*) as a, album(album_id, title, a.artist_id)
```

The `a.artist_id` in positional access matches the `a.artist_id` from `artist(*) as a`, creating unification. This mixes styles: wildcard for one table, positional for another, with explicit cross-reference.

A more elaborate example:

```delightql
playlist(*) as p,
track(*) as t,
playlist_track(p.playlist_id, t.track_id)
```

Here `playlist_track` unifies with `playlist` on `p.playlist_id` and with `track` on `t.track_id` – a three-way join through positional cross-references.

3.1.5. Literals and Constraints

Ground terms in positional access create `WHERE` conditions:

```delightql
invoice(invoice_id, customer_id, invoice_date, _, _, _, "Canada",
_, total)
```

The positional grounding filters rows where the seventh column equals `"Canada"`. The column `billing_country` has been unified with the ground value `"Canada"`.

```SQL
SELECT invoice_id, customer_id, invoice_date, total FROM invoice
WHERE billing_country IS NOT DISTINCT FROM 'Canada';
```

3.1.6. Self-Unification

The same name repeated in positional access forces equality:

```delightql
invoice(invoice_id, customer_id, invoice_date, _, billing_city,
billing_city, _, _, total)
```

Columns 5 and 6 both bind to `billing_city`. This filters to rows where those columns are equal:

```SQL
SELECT invoice_id, customer_id, invoice_date, billing_city,
total FROM invoice WHERE billing_city IS NOT DISTINCT FROM
billing_state;
```

3.1.7. Anonymous Tables and Unification

Anonymous tables participate in unification through their header names:

```delightql
invoice(invoice_id, customer_id, _, _, _, _, billing_country, _,
total),
_(billing_country @ "Canada"; "France"; "Germany")
```

The anonymous table introduces `billing_country`. This matches `billing_country` from `invoice`, creating:

SQL

```sql
SELECT invoice_id, customer_id, billing_country, total
FROM invoice
WHERE billing_country IN ('Canada', 'France', 'Germany');
```

With wildcard access, qualification is required:

delightql

```
invoice(*) as i,
_(i.billing_country @ "Canada"; "France"; "Germany")
```

Without the `i.` prefix, no unification occurs – the anonymous table's `billing_country` wouldn't match `i.billing_country`.

3.1.8. Lvars as Data in Anonymous Tables

Anonymous tables can use lvars as data values, not just in headers.

Constraint: An lvar cannot appear both in a header and in the data rows of the same anonymous table.

3.1.8.1. Inverted IN Pattern

delightql

```
employee(*) as e,
_(2 @ e.employee_id; e.reports_to)
```

Find employees where `2` appears in any of these columns:

SQL

```sql
SELECT * FROM employee e
WHERE 2 IN (e.employee_id, e.reports_to);
```

Or equivalently:

SQL

```sql
SELECT * FROM employee e
WHERE e.employee_id = 2
   OR e.reports_to = 2;
```

The anonymous table's header is a literal (`2`); the data rows are lvars from `employee`. This inverts the typical IN pattern.

3.1.8.2. EAV Transformation

delightql

```
employee(*) as e,
_(attribute, value @
  "name", e.last_name;
  "email", e.email;
  "title", e.title;
  "hired", e.hire_date)
```

This is the melt pattern.

3.1.8.3. Row-Wise Correspondence

delightql

```
customer(*) as c,
employee(*) as e,
_(c.country, e.employee_id @
  e.country, c.support_rep_id;
  e.country, 1;
  "USA", 2)
```

Each row in the anonymous table represents a valid combination. The result includes only rows where `(c.country, e.employee_id)` matches one of the specified pairs:

SQL

```
SELECT *
FROM customer c, employee e
WHERE (c.country = e.country AND e.employee_id =
c.support_rep_id)
   OR (c.country = e.country AND e.employee_id = 1)
   OR (c.country = 'USA' AND e.employee_id = 2);
```

3.1.9. Summary of Unification Rules

Pattern	Names Introduced	Unifies With
`t(a, b, c)`	`a`, `b`, `c`	Any `a`, `b`, `c`
`t(*)`	`t.a`, `t.b`, `t.c`	Only `t.a`, `t.b`, `t.c`
`t(*) as x`	`x.a`, `x.b`, `x.c`	Only `x.a`, `x.b`, `x.c`
`t(x.a, b, _)`	`x.a`, `b`	`x.a` from alias `x`; any `b`
`_("lit" @ v1; v2)`	(none – header is literal)	Filters where `lit` matches `v1` or `v2`
`_(col @ "a"; "b")`	`col`	Any `col`

Table 32: Summary of unification rules

3.2. IDENTIFIERS MIRROR CALL SITES

In many programming languages the identifier always matches a particular grammar wherein the identifier may stand for a scalar, a function, a data structure, etc.

Thus, for example, the identifier **`execute`** could be a variable that holds an entity from different abstraction levels. This is especially prevelant in functional programming languages.

Delightql makes the identifier structured. The main place in which this matters is in the parameter list of a higher-order rule.

3.3. NULL SEMANTICS

3.3.1. Equality expressions

When both sides of a delightql = expression come from two different relations, then the traditional SQL = is used during transpilation. This rule applies for regular joins and inner relations (EXISTS and scalar subqueries).

When the above condition is not met, then delightql transpiles = to SQL's `IS NOT DISTINCT FROM`.

3.3.2. Unification equality

The rules of equality expressions are created to use null-safe equality () in all places where join semantics are at play. This applies to Prolog-style unification as well:

delightql

```
playlist(playlist_id, name), playlist_track(playlist_id, track_id)
```

SQL

```
SELECT playlist.playlist_id, playlist.name,
playlist_track.track_id
FROM playlist, playlist_track
WHERE playlist.playlist_id = playlist_track.playlist_id;
```

delightql

```
employee(*) as e, employee(*) as m, e.reports_to = m.employee_id
```

SQL

```
SELECT *
FROM employee e, employee m
WHERE e.reports_to = m.employee_id;
```

3.3.3. Traditional SQL Equals

If you must transpile the regular SQL equals in non-join locations, you can use `+sql_eq(l,r)` which is included as part of `std::prelude`.

3.4. OPERATIONAL SEMANTICS OF QUERY EXPRESSIONS

Delightql operates by transforming valid delightql queries into valid SQL queries and submitting it to the relevant SQL engine.

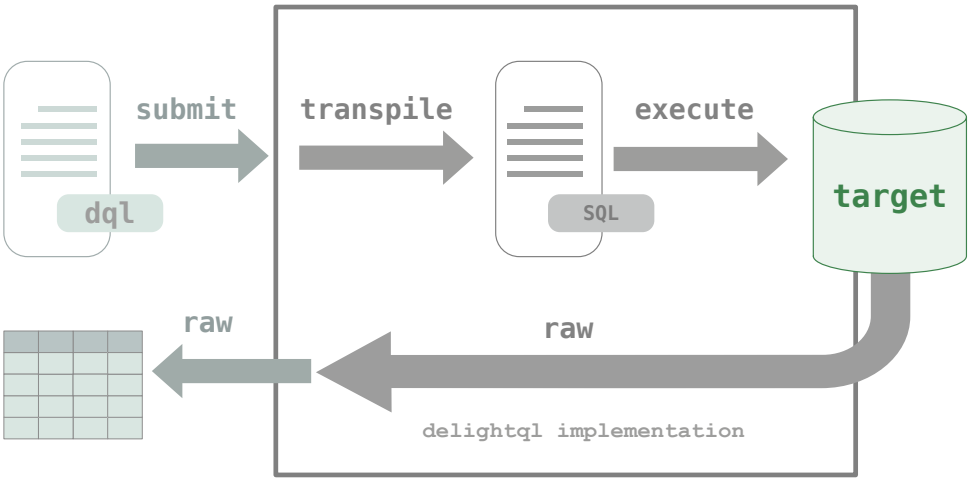


Figure 5: Inward Operations Only

Delightql makes a non-modification guarantee to the results that are returned – in other words, it passes back the raw bytes produced by the target engine.

By abstaining from any transformation to the data returned to the user, delightql ties the meaning of its code (query expressions) to the meaning of the SQL it produces – that is to say, meaning (i.e. semantics) is transitively mapped from a delightql string to a SQL string, and then to a resulting table.

Any questions about delightql's semantics are ultimately answered by the combination of its transpilation choices and what the targeted SQL engine returns.

Delightql may choose to arrange SQL in any of various forms if it can guarantee that the target SQL engine will return the exact same table. This choice affords flexibility in what SQL to produce that is orthogonal to performance or formatting.

3.5. OPERATIONAL SEMANTICS OF DIRECTIVES

Directives (i.e. effects) are the impure entities within the delightql language. They cannot have a semantic that is defined by a single query. In fact, with the lack of certain branching and looping primitives in the SQL standard for DDL, some primitives cannot have a semantic consisting of a sequence of DDL and DML.

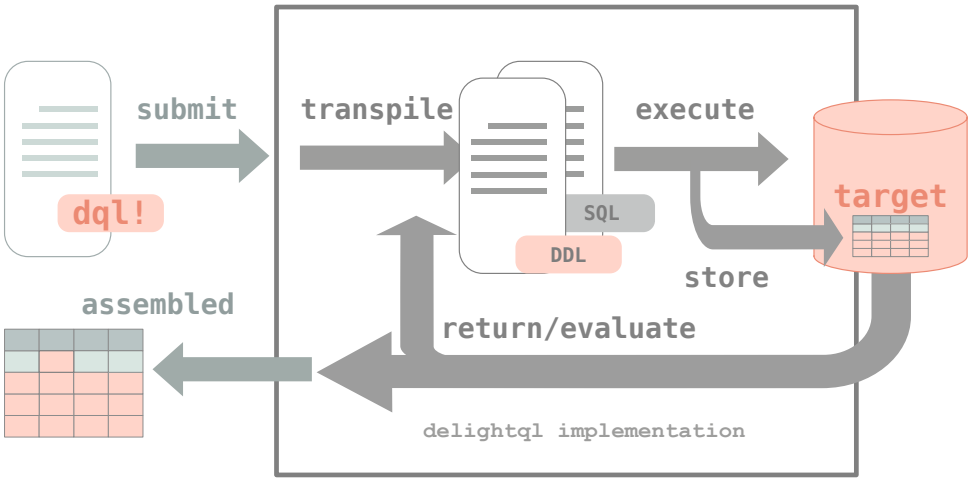


Figure 6: Inward Operations Only

Delightql therefore absolves directives as a category from the obligation to return back from the target whatever was returned. The delightql runtime must interpret and act accordingly on different conditions.

3.6. ENTITY SHADOWING AND UNIQUENESS

There are three axes for understanding the identity of an entity:

- name: the string identifier of the table, view, function or rule
- arity: the number of columns
- kind: whether the entity was defined as rule-first, function-first, sigma-condition-first or mounted from a source system

Delightql chooses a conservative identification scheme whereby name alone uniquely identifies an entity. This scheme prohibits the following in a single namespace:

- two entities with the same name but different arities
- two entities with the same name but different types (function vs rule vs sigma rule)

This does not prohibit the definition of shadowing entities, which may have different arities and/or kinds. Shadowing occurs when an entity is defined in a context/scope with higher precedence.

Some entities are always uniquely identified by a fully-qualified name – the concatentation of the namespace with a name. Referencing that entity with its fully-qualified name is guaranteed to access the entity in a way that will never be ambiguous. CTEs don't exist in a namespace and are always referenced without qualification. A relational alias is not an entity at all: it names one occurrence (see Aliases name occurrences).

Ambiguity can occur only at a tier that contains peer candidates, principally the enlisted-namespace tier. Enlistment is an ergonomic

choice that may make two distinct entities available under one bare name. It is legal to construct that state; the ambiguous name refuses only when mentioned.

It is up to the programmer to remove that ambiguity via qualification, aliasing, or renaming.

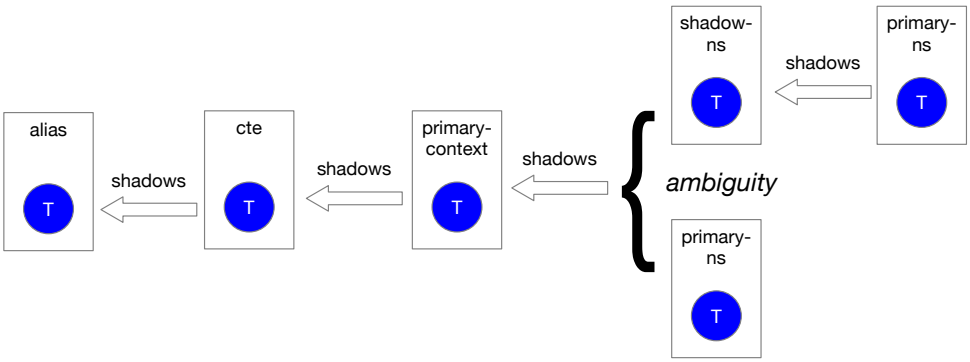


Figure 7: Entity Shadowing

3.6.1. Contexts

Contexts are the scopes into which an entity may be defined.

Entities can exist (be defined) in the following contexts:

- inside a data namespace (made visible by a `mount!`)
- inside a lib namespace (made visible by a `consult!`)
- inside a shadow namespace (created by a `temp_table!`, `temp_view!`)
- inside the primary context (`home` for REPL usage, and the primary namespace for file-based consulting)
- as a CTE
- within the target scope

A deep namespace path creates its missing prefixes as structural nodes. For example, consulting into `a::b::c` creates `a` and `a::b` if needed, but does not give them `lib` or `data` backing or entities. A later explicit `consult!` may claim an unbacked prefix as a library, or `mount!` may claim it as data. Either claim preserves existing children and refuses collisions or replacement of an already backed namespace. Backing belongs to each node independently: an empty library is still a library, and a child does not inherit its parent's kind. Enlisting a parent does not make its children's entities bare. This structural-prefix rule is ratified for alpha; implementation is pending.

Removing the last child does not silently remove an implicit parent. The empty structural node keeps its identity and can be claimed later. An explicit `unconsult!` of its exact path removes that node alone if it has no children or published dependents; it does not prune its parent.

3.6.2. Context Shadowing

We will use the notation `T[c]` to discuss the partial order that rules shadowing. It should be read aloud as "the entity `T` defined in context `c`" – for example `users[cte]` means "the entity with the name `users` defined as a CTE".

The context shadowing rule states that the following order always applies:

```
T[cte]
  >  T[primary-context]
  > { all other contexts }
```

Where the `>` indicates that the scope to the left supersedes and redefines all entity references to the right.

The "all other contexts" refers to the combination of separately enlisted namespaces, none of which has priority over another – which we denote with the `||` parallel symbol.

The first nonempty tier wins. A same-named entity in a lower tier is not a fallback if the selected entity has the wrong kind, arity, or heading.

```
{ all other contexts } =
  { T[shadow-NS1] > T[NS1] }
  ||
  { T[shadow-NS2] > T[NS2] }
  ||
  { T[NS3] }
```

As an example:

delightql

```
mount!("etl.sqlite","etl")(*)
enlist!("etl")(*)

partner_sale_2025_07_31(*), quantity > 0 |> temp_table!
(etl.partner_sale_2025_07_31(*))(*)

(~~ddl partner_sale_2025_07_31(*) :- _(sale_id @ 4; 5; 6) ~~)

invoice(*), total > 20 : partner_sale_2025_07_31
media_type(*) as partner_sale_2025_07_31 |>
(partner_sale_2025_07_31.name)
```

In the example above we assume the existence of a `partner_sale_2025_07_31` entity within the namespace `etl`.

- after line 2 an enlisted `partner_sale_2025_07_31` is available (from `etl`)
- after line 4 the temporary table `partner_sale_2025_07_31`, created for `etl`, shadows the original within the `etl` candidate. The mounted database shares the primary connection's temp schema, so its exact identity is `sys::shadow::main.partner_sale_2025_07_31`; no extra enlist edge is created, and a bare `temp_table!(partner_sale_2025_07_31(*))(*)` would instead have created a session object for `main`, the default write target.

- after line 6 the primary context definition of `partner_sale_2025_07_31` shadows the temporary table
- after line 8 where the CTE `partner_sale_2025_07_31` is defined, the query's use of `partner_sale_2025_07_31` will shadow all others
- at line 9 the alias `as partner_sale_2025_07_31` names the `media_type` occurrence, so `partner_sale_2025_07_31.name` addresses that occurrence's column. It does not rebind the name `partner_sale_2025_07_31`: a `partner_sale_2025_07_31(*)` read still selects the CTE.

> Warning: an alias names an occurrence, not an entity. In
>
> ```delightql
> invoice(*), total > 20 : partner_sale_2025_07_31
> media_type(*) as partner_sale_2025_07_31 |>
> (partner_sale_2025_07_31.name), partner_sale_2025_07_31(*)
> ```
>
> the `partner_sale_2025_07_31(*)` at the end of the query refers to the CTE. The alias never answers a relation name, and its column addressing ended at the scope barrier `|> (partner_sale_2025_07_31.name)`.

3.6.3. Aliases name occurrences

An alias (`as a`) names one read of a relation for column addressing. It creates no relation that a later mention can call:

```delightql
employee(*) as a, employee(*) as b      // two fresh reads of Employee: a self-join
invoice(*) as a, a.total > 20           // a.Total addresses the first occurrence
album(*) as artist, artist(*) as c      // Artist(*) selects by the shadowing order above
```

While the occurrence is live, a qualifier addresses its columns. Qualifiers address occurrences, never entities, so an entity that happens to be spelled `artist` does not compete with the alias. In relation position, `artist(*)` is an ordinary mention. It selects an independently defined `artist` by the shadowing order, or it fails; it never reopens the occurrence. A read whose scope would share the live name `artist` refuses, because two live scopes cannot share a name.

The same holds for a read of a CTE, of a mounted table, or of an aliased table expression (`R as u(a, b)`). A pipe ends the occurrence's addressing; to address a result further, name it (`|> (…) as v`).

3.6.4. Materialization Shadowing

Materialization shadowing is exactly the SQL DDL semantic available via `CREATE TEMPORARY TABLE` and `CREATE TEMPORARY VIEW`. It is the mechanism by which a session scoped entity may exist only for the duration of that session.

The order induced by the materialized shadow, inside a data namespace `D`, is:

```
T[session object recorded as overlaying D] > T[D]
```

The delightql directives `temp_table!` and `temp_view!` may be used to create these shadowing entities.

delightql

```
mount!("etl.sqlite","data::baz")(*)
enlist!("data::baz")(*)

partner_sale_2025_07_31(*) , quantity > 0 |> temp_table!
(data::baz.partner_sale_2025_07_31(*))(*)
```

The target names `data::baz` as the durable namespace this session object overlays. On SQLite, the mounted database shares the primary connection's temp schema, so the object's exact shadow path is `sys::shadow::main.partner_sale_2025_07_31`, not `sys::shadow::data::baz.partner_sale_2025_07_31`. Its recorded durable owner is still `data::baz`. Bare selection applies that overlay to the `data::baz` candidate before comparing it with peer enlisted namespaces; it does not apply it to `main` merely because `main` appears in the shadow path. A second durable owner on the same connection cannot create another physical temp `partner_sale_2025_07_31` at that path; the second creation refuses and names the holder.

delightql

```
mount!("etl.sqlite","data::baz")(*)
enlist!("data::baz")(*)
partner_sale_2025_07_31(*) , quantity > 0 |> temp_table!
(data::baz.partner_sale_2025_07_31(*))(*)

partner_sale_2025_07_31(*)                    // resolves from the local
shadow
data::baz.partner_sale_2025_07_31(*)          // exact durable entity
sys::shadow::main.partner_sale_2025_07_31(*)  // exact session entity on this
connection
```

The creation receipt reports both names: `target = data::baz.partner_sale_2025_07_31` is the resolved data-namespace target, while `created = sys::shadow::main.partner_sale_2025_07_31` is the exact session entity. For a durable `table!`, `target` and `created` coincide. A bare creation target uses the session's default data-write target (`main`); a qualified creation target must be data-backed. A source that cannot run on that target's connection is refused, not relocated. Without a database file, `main` is backed by the session's

in-memory primary: ordinary `table!` objects last for that database session, and temporary objects use its temp schema.

3.6.5. Addressing and enlistment

Qualification selects an exact route:

```delightql
(~~ddl:"reports" daily(*) :- invoice(*) |> %(invoice_date ~> sum:
(total) as total) ~~)

home::reports.daily(*)        // exact path
.::reports.daily(*)           // relative to the primary definition context
alias!("home::reports", "r")(*)
r.daily(*)                    // exact alias route
```

An unmarked qualifier is an exact top-level namespace or an explicit `alias!` shorthand. It does not search the children of enlisted namespaces. Enlisting a namespace makes that namespace's own entities bare; it does not make child namespaces implicitly addressable and is not transitive.

The session begins with these four enlist edges:

- `main`
- `std::prelude`
- `std::predicates`
- `sys::meta`

`home` is different: it is the top-level primary definition context, not a fifth auto-enlist. That is why its entities are bare and why they outrank enlisted entities. `sys::ns.enlisted_namespace(*)` reports enlist edges only; it does not report `home` merely for being the primary context.

Namespace aliases are qualifier shorthands. They do not grant bare entity names and, like relational aliases, occupy no tier of the shadowing order.

3.7. AMBIGUITY IN NON-SHADOWING CONTEXTS

Any context that is not a CTE or the primary context may be ambiguous with other such contexts. This is inclusive of materialized shadows which still attach to their original namespace.

```
{ T[shadow-NS1] > T[NS1] }
||
{ T[shadow-NS2] > T[NS2] }
||
{ T[NS3] } }
```

Multiple enlisted namespaces may export the same entity name. This overlap does not itself cause an error. A bare reference is ambiguous only when more than one distinct entity remains after applying the shadowing rules. Fully qualified references remain available. The bare

references that are shared from more than one enlisted namespaces and which have no shadows are ambiguous entities.

Using an ambiguous entity is an error and must be resolved by the programmer. Enlistment itself still succeeds. Its receipt may list names that have become ambiguous, but the list is advice, not a second resolution authority. The diagnostic at the mention lists the complete set of distinct candidate identities; enlist order and `main` do not break the tie.

Shadowing is by name before shape. A temporary `foo` may have a different kind, arity, or heading from the durable `foo` it shadows. If the selected temporary entity cannot serve a use, that use refuses; DelightQL does not silently fall through to the durable entity.

3.7.1. Binding follows lexical definition

Take a library `lib::a`, consulted from `a.dql`:

delightql

```
// a.dql
items(id, name
  ------------
  1, "pick"; 2, "tuner")
```

In the following example, the binding of `items` within the body of `report` is a reference to the un-qualified entity from `lib::a`.

delightql

```
consult!("a.dql", "lib::a")(*)
enlist!("lib::a")(*)

(~~ddl
report(*) :- items(*)
~~)

delist!("lib::a")(*)
report(*)
```

After a `delist!`, we retain the reference to `items` from its namespace because it was defined lexically.

The other option, dynamic scoping, where `items` is dynamically swapped in and out is not correct semantics.

The dependency retains its namespace identity, not a historical copy of its implementation. Thus, a successful `reconsult!` of `lib::a` changes what subsequent statements observe through that dependency.

`unconsult!` differs from `reconsult!`: it refuses if removing `lib::a` would leave a published definition such as `report` depending on it. The refusal names the blocking definition and namespace. A fully qualified reference may have been written before its target was consulted; once that target exists and the reference resolves, it too blocks removal. Change or remove the dependent definition first, or

rebuild the scripted session. This alpha removal rule is ratified; implementation is pending.

When no published definition blocks removal, `unconsult!` also removes the prompt's enlistments and namespace aliases pointing to the target. Its receipt names these removed edges. A scratch rule created with `(~~ddl … ~~)` is a definition, not one of those disposable prompt edges.

For an ordinary mounted data namespace, `unmount!` follows the same rule: a published definition reading its tables blocks removal, while a successful removal cleans prompt-owned enlistments and aliases and names them in the receipt. The special lifecycle of `main` remains a separate question.

3.7.2. Retracting a session definition

`retract!` removes one complete session-authored entity family:

```delightql
(~~ddl foo(*) :- _(x @ 1) ~~)
retract!(foo(*))(*)
```

The argument is resolved by the ordinary rules. At the prompt, bare `foo` therefore selects `home.foo` before an enlisted `foo`; the explicit spelling is `retract!(home.foo(*))(*)`. This is family removal, not Prolog's one-matching-clause retraction.

Only mutable definitions authored in `home` or another scratch namespace may be retracted. Data tables, system entities, consulted-source definitions, and CTEs refuse; an alias is never selected, so it is never a target. A dependent definition or grounding also blocks removal. The receipt names the identity removed; a later bare mention is resolved afresh and may therefore reveal a lower-tier candidate.

3.8. DEFAULT TRANSPILATION STRATEGY AND RESOLUTION

Delightql syntax makes a clear distinction between what is

- a table – `foo(*)`
- a higher-order table – `foo(bar)(*)`,
- a function – `foo:(x,y)`, or
- a sigma predicate – `+foo(bar)(*)`.

These different categories of entity have different rules for both resolution and transpilation. Importantly, functions and higher-order tables among these are permitted to be unknown and transpiled directly as SQL fragments. Tables and sigma predicates must be known.

3.8.1. Table Resolution and Transpilation

Calls of the form `foo(*)` may either be delightql rules or target tables.

delightql

```
artist(*)
```

In either case, delightql's resolution both ensures that the name (`artist`) exists and that its columns (dimensions) are named and in a certain order. For a data-backed table, the default transpilation rule places the entity after a `FROM` (e.g. `FROM artist`).

SQL

```
select * from artist;
```

Resolution for non-namespaced entities checks common-table expressions first followed by any in-scope entities. Only CTEs are allowed to shadow – no named entities sharing the same name are allowed to exist in the same scope.

3.8.2. Higher-order Resolution and Transpilation

delightql

```
foo(bar)(*)
```

Higher-order forms can be delightql rules or target table-value functions. If delightql finds the entity as a delightql-authored higher-order rules, it will use the rule as needed.

But if a delightql-authored higher-order rule is not found it will default to assuming that the entity is a table-value function and transpile as a function-call form after the `FROM`.

SQL

```
select * from foo(bar);
```

This default tranpsilation rule runs assuming imperfect knowledge as to whether the SQL target defines such an entity.

3.8.3. Function Resolution and Transpilation

Functions may be delightql-authored or functions hosted by the target SQL engine. If delightql finds an entity that is delightql-authored, it will use it first inclusive of shadowing.

Upon finding no delightql-authored function, delightql will emit a SQL fragment that transpiles the delightql function form into the SQL function form:

delightql

```
album(*) |> +( hex:(title) as title_hex)
```

SQL

```
select *, hex(title) as title_hex from album;
```

This default tranpsilation rules runs assuming imperfect knowledge as to whether the SQL target defines such an entity. That is to say, the `hex` in the above SQL may or may not exist as a function in the target SQL engine.

The SQLite that DelightQL ships is the same build everywhere it runs, the command line and the browser playground alike, so a target call answers or refuses the same way on both. That build includes SQLite's math functions (`sqrt`, `cos`, `floor`, `log`, …), its JSON functions, `soundex`, FTS and R-Tree; a call the engine does not define refuses with the engine's own message.

3.8.4. Sigma Predicate Resolution and Transpilation

All sigma predicates have no default transpilation rule. This is because SQL has no uniform syntax for row predication with such diverse syntax as `LIKE`, `BETWEEN`, in-fix operators, `IN`, and `EXSISTS`.

All delightql calls of the form

```delightql
track(*), +like(name, "%Love%")
```

will resolve to a delightql-authored entity (either built-in or programmer authored).

3.9. VARIABLE AND COLUMN SCOPING

Within a valid delightql expression the scope of logic variables (columns) is determined by a combination of a left-to-right evaluation strategy and per-operator rules.

The concepts of relational continuations and current pending relation are interrelated with this left-to-right evaluation strategy. Further scoping rules are given to interior relations which provide

3.9.1. Relational Continuation

Relational continuations are defined as all strings that - may be appended to a valid delightql relational expression (RELEX) to form a new valid delightql relational expression - and are not themselves valid delightql RELEXs

```delightql
customer(*), country = "Canada"
   |> +( :"{last_name}, {first_name}" as full_name)
```

Examples of continuations in the above valid delightql RELEX:

- `, country = "Canada"`
- `|> +( :"{last_name}, {first_name}" as full_name)`

- `, country = "Canada" |> +( :"{last_name}, {first_name}" as full_name)`

The continuation concept allows syntactic analysis of semantic concepts regarding scope and transformation.

3.9.1.1. Continuation Anchor

A continuation anchor is any location in a string where the substring to the left of the anchor is a RELEX and where a CONTINUATION may replace the substring to the right. This is inclusive of a zero width empty string at the end of the string.

delightql

```
album(*)                  // ①
    , artist_id = 22       // ②
    |> ( title )          // ③
```

The above example has three continuation anchors at the end of each line. These are the only three continuation anchors in this string.

3.9.1.2. Current Pending Relation

The current pending relation (CPR) is the relational value of the relational expression (RELEX) to the left of a continuation anchor. It is the ordered set of logic variables (columns) that are in scope.

delightql

```
album(*) as a
    // ① CPR = [ a.album_id, a.Title, a.artist_id ]
    , artist_id = 22
    // ② CPR = [ a.album_id, a.Title, a.artist_id ]
    |> ( title )
    // ③ CPR = [ Title ]
```

3.9.2. Left-to-Right Evaluation

Delightql uses a left-to-right evaluation strategy for determining both what variables are in scope and the state of the current pending relation (CPR).

Within a delightql relational expression, parentheses are not used at the relational level to force scope or evaluation – though they are permitted for grouping and binding domain expressions.

Delightql does have something like parentheses, but these are called interior relations and are accurately described as named contextual scopes.

3.9.2.1. Scope Introduction

Logic variables are brought into scope by ground relational expressions (GRELEXs) either directly or via joining or unioning. A GRELEX is inclusive of anonymous tables and literal references.

delightql

```
employee(*)
```

The `employee(*)` GRELEX literal reference introduces its logic variables (LVars) into scope.

The current pending relation (CPR) of each new continuation may grow or shrink or stay the same based on the category of relational operator that is applied in the continuation.

delightql

```
employee(*)   // ①
  ,_(a@3;39) // ②
```

The `employee(*)` GRELEX introduces logic variables into scope, followed by the `,_(a@3;39)` JOIN continuation which introduces even more logic variables (just `a`) into scope.

delightql

```
artist(*)
  ,album(*.(artist_id))
  |> ( name, title) // ③
```

The `|> ( name, title)` PROJECTION continuation removes logic variables and establishes a new scope barrier.

3.9.2.2. Scope Barrier

A scope barrier replaces the local relational interface: following continuations cannot recover its old logic variables unless republished. A pipe inside an interior relation does not also remove bindings from an enclosing scope. Those remain accessible under the ordinary naming rules, without automatically becoming columns of the interior's output.

delightql

```
album(*) as a
    //① CPR = [ a.album_id, a.Title, a.artist_id ]
    , artist_id = 22
    //② CPR = [ a.album_id, a.Title, a.artist_id ]
    |> ( title )
    //③ CPR = [ Title ]
    ,title="Coda"
    //④ no access to a.*. Only Title
```

After the third continuation which is a scope barrier, the fourth continuation does not have access to the logic variables `a.album_id`, `a.title`, or `a.artist_id`.

Scope barriers are most often post-pipe projection operators, but also include metaize ^ and witness +.

```delightql
employee(*) ^ // removes Employee's logical variables from scope
```

3.9.2.3. Non-commutativity of usage versus introduction

Because delightql uses a left-to-right evaluation scheme, a logic variable cannot be used unless it has been brought into scope to the left. This is in contrast to other "more declarative" languages where the usage and the introduction may be swapped.

```delightql
album(*), milliseconds < 20000, track(*)
```

The above is incorrect as the logic variable `milliseconds` has not yet been brought into scope via a left-to-right evaluation strategy.

3.9.3. Interior relations and scope

Interior relations create their own scope within the parentheses of a functor.

```delightql
track( , milliseconds<60000 |> (name))
```

An interior relation is constructed by authoring a relational continuation immediately after the opening (left) parenthesis of a named predicate and ending the continuation with the closing (right) parenthesis.

The interior relations' scope is delimited and private, but has access to the surrounding scope's logic variables. The final current pending relation (CPR) of the interior relation is published under the name of the functor.

```delightql
track(                      // ①
  , milliseconds<60000      // ②
  |> (name)                 // ③
  )
  //④ Track.Name
  // is the only logic variable
```

In the above example, a relational continuation is started at position one, and proceeds through to position three. The closing of the interior relation publishes the last CPR of the interior relation as the columns of the relation named `track`.

The term interior relation is used for multiple manifestations:

- simple subqueries
- lateral joins
- scalar subqueries

- EXISTS semi-joins

3.10. NOMINALIZATION

Delightql has strict rules about what columns may be addressed by name and how they may be addressed by name.

Delightql insists that a column may always be addressable by name or never addressable by name (minting).

All columns are always addressable by index ordinalization.

3.10.1. Functor rules

A delightql functor (independent of usage – i.e. table, function or semi-join) may be of three forms:

- inchoate
- argumentative
- interior

3.10.1.1. Inchoate

An inchoate functor is the literal reference to a ground relation GRELEX having nothing internal to its parentheses.

delightql

```
employee()
```

It names the relation without asking for any of its columns, so it selects zero rows. Its columns are opaque: none can be addressed by name, only by position. `employee(*)` activates them.

3.10.1.2. Argumentative

3.10.1.3. Interior

3.10.2. Minting

Any column that results from a function application or from a column ambiguity is automatically minted so that the column name may never be used again.

3.10.2.1. Function Application Minting

delightql

```
invoice_line(*)
  |> +( unit_price * quantity)
```

The `unit_price * quantity` column has a name generated for it that may never be used again. This means that absent naming the column at creation with `as` the only means of accessing this column is by ordinal index.

delightql

```
invoice_line(*)
  |> +( unit_price * quantity)
  |> ( |-1| )
```

3.10.2.2. Column Ambiguity

delightql

```
track(*), genre(*.(genre_id))
  |> (*)
  // Track.Name and Genre.Name receive
  // minted names
```

If `track.name` and `genre.name` are in-scope in the above example, then both are minted as neither are more real than the other.

Solve this by removing or renaming one of the columns:

delightql

```
track(*), genre(*.(genre_id))
  |> -( genre.name)
  // Name is addressable now
```

delightql

```
track(*), genre(*.(genre_id))
  |> *( genre.name as genre_name)
  // Name is addressable now
  // genre_name is addressable now
```

3.10.3. The Post-Fix Operators

- `+`
- `\+`
- `^`
- `*`
- `.(identifier)`
- `.*`

3.11. MODING AND CURRYING

Moding is the activity of marking a column as input only, output only, or both. Another way of saying input and output is supplied and returned.

The term mode comes from Prolog where syntax exists to declare certain dimensions to be input and others to be output. [29]

```
:- mode concatenate(+,+,-).
```

29 In ISO this syntax is documentation only. Some Prolog implementations enforce these annotations.

We must be aware of this Prolog heritage as we discuss where delightql matches versus diverges from it.

3.11.1. Prolog syntax

In Prolog, all predicates obey the same syntax (except for infix) and declare their dimensions.

```
foo(A,B,C).
```

But each column could be used in queries as output only, input only, or mixed (and possibly functional). Respectively:

All output:

```
foo(A,B,C).
```

All input:

```
foo(1,"bar",42).
```

Some input, some output:

```
foo(A,"bar",C).
```

In a very important way, the below is also an example of input only:

```
foo(A,"bar",C),A=1,C=42.
```

The logic variables `A` and `C` in the example above are still being input, as they have been instantiated with ground values.

3.11.2. Delightql's answer to moding

Delightql has semantic enforced definitions of which columns must be input only, versus those that can be mixed input or output.

Succinctly:

- All ordinary function parameters are input-only.
- Every position in an ordinary table or relational-rule schema is relational/bidirectional: it may be supplied or returned.
- All sigma-predicate parameters are input-only.
- A fact function with a _ default (TOTAL-CASE) arm:
  - positions left of `->` are input-only
  - and positions right of `->` are output-only.
- A fact function without the default arm (PARTIAL-CASE) is table-like: all columns may be input or output when accessed like a table (e.g. `foo(a,b,c)`)
- A fact function without the default arm (PARTIAL-CASE) can also be accessed like a function, in which case all ouput columns are elided and only the leftmost input columns are written (e.g. `foo:(a,b)` where `a,b -> c`).
- A fact-function call with exactly one output may stand directly in scalar/value composition; this implicitly selects its sole output. The output may still be selected explicitly by name.
- A fact-function call with several outputs must name the exact output selected whenever a scalar value is required..

- All parameters in the first parentheses of a higher-order/input-moded rule are input-only

All of the above semantics are parsable, that is to say, they are all different syntaxes that should be easily discernible.

3.11.3. Functions Column Modality

Any syntax of the form `foo:(x,y)` requires that its columns be existent and passed in:

```delightql
employee(*) |> ( upper:(last_name), upper:("literal") )
```

The above syntax might seem almost counter-intuitively obvious to anyone who has used a general-purpose programming.

But delightql allows other syntax that is still input-moded and which may be unfamiliar to those not used to Prolog:

```delightql
album(_, title, 22), upper:(title)="CODA"
```

Nevertheless, the fact that a form `foo:(a)` is discernible at all means that this column must be input only.

3.11.4. Tables Column Modality

Any syntax of the form `foo(x,y)` means that the columns may be input (supplied) or output (returned).

This matches the syntax and semantics of Prolog.

```delightql
genre(id,"Jazz")
```

In the above form, the `id` variable is output and the `"Jazz"` argument is input, producing a WHERE predication via grounding:

```SQL
select genre_id as id from genre where name = 'Jazz';
```

It is important to understand that in Prolog-like argumentatitve join syntax both columns are input at the same time :

```delightql
playlist_track(5,track),playlist_track(17,track)
```

In the above the variable `track` is input (or bidirectionally unified) to both predicates, creating a join.

The GLOB makes all columns into output columns:

```delightql
genre(*)
```

without having to enumerate all dimension positions.

3.11.5. Sigma Predicates Column Modality

All calls of the form `+foo(x,y)` or `\+ foo(x,y)` are EXISTS/NOT-EXISTS sigma-predicate invocations.

```delightql
track(*),
  +between(milliseconds,60000,180000),
  +like(name,"%Love%")
```

Delightql requires that all columns be instantiated as input.

Sigma predicates defined as re-usable rules, must only use forms in their body that pass the input moding to their implementation:

```delightql
no_data("NA"; "N/A"; "UNKNOWN")

empty(column) :- null = column
empty(column) :- trim:(column) = ""
empty(column) :- +no_data(upper:(column))
```

Note that all built-in predicates, like `=` and `<` and built-in functions, like `+` and `*` are also input-only moded.

Observing the definition of a rule consist of calls to input-moded predicates is enough to distinguish this rule from table rules.

The invocation of this rule will always be discernible, by virtue of requiring a `+` or `\+` EXISTS marker before the predicate name:

```delightql
track(*),
  +empty(composer)
```

3.11.6. Function Facts Column Modality

Function facts are DDL for defining named case statements. They utilitize the `->` (spoken F-AND) in their definitions to separate dimensions that are input from dimensions that are output:

```delightql
department_kind(
  department     -> kind
  ------------------
  "engineering"  -> "tech";
  "data science" -> "tech";
  _              -> "other"
)
```

In some respects, the invocation of the function fact is indistinguishable from a regular function call:

```delightql
employee(*) |> +( department_kind:(title) as kind)
```

but when function facts are not-total, i.e. they lack a default arm `_ -> "other"`,

```delightql
partial_department_kind(
  department     -> kind
  ------------------
  "engineering"  -> "tech";
  "data science" -> "tech"
)
```

then they may be called as a table:

```delightql
partial_department_kind(*)
```

and their columns may be input or output or both:

```delightql
partial_department_kind( "engineering", kind)
// or
partial_department_kind( e, k)
```

and still be available to be called as functions:

```delightql
employee(*)
  |> +( partial_department_kind:(title) as kind)
```

Function fact forms that return tuples must have their return columns accessed by name to project out the element of the tuple:

```delightql
shipping(zone, weight -> carrier, days
         --------------------------
         1, 5 -> "ground", 3;
         1, 50 -> "freight", 7;
         2, 5 -> "air", 1;
         _ -> "unrouted", 0)
```

```delightql
_(zone, weight
  ------------
  1, 5; 1, 50; 2, 5; 3, 20)
  |> (shipping:(zone, weight).carrier as carrier,
      shipping:(zone, weight).days as days)
```

This is the only place in delightql where the column is output-only and manifest.

3.11.7. Higher-Order Predicates Column Modality

The other name for higher-order predicates is input-moded predicates.

They are discernible in code by having multiple parentheses, or by being the target of a relational pipe:

```delightql
customer(*)
  |> at_least(50)(*)
```

```delightql
at_least(50, customer(*))(*)
```

The first parentheses (the ones to the left and closest to the name of the higher-order rule) contain only input-only columns. The columns in the second parentheses are treated like table columns which are input-or-output.

Again, a literal reference is not required to trigger input instantiation:

```delightql
_(val @ 1;2;3;4), iota(val)(*)
```

In the above example, the anonymous table's lone column `val` becomes the input to the `iota` parameter set.

3.12. GROUNDING HIGHER-ORDER LITERALS

4. DATA DEFINITION LANGUAGE (DDL)

The definition of SQL DDL is the set of features that create, modify, or delete schema entities.

Delightql's DDL encompasses SQL's DDL and extra features with no direct SQL mapping (higher-order predicates, ER-context rules). The thematic concern is reusability: definitions, tables, and data that can be authored, loaded, and referenced.

This section covers:

- Relational rules. Views, higher-order views, ER-context rules, and sigma predicates.
- Function rules. Reusable domain functions.
- Facts. Axiomatic ground data.
- Namespaces. Organization and visibility of all the above.

4.1. BASICS

> Assertion mode vs query mode. Delightql, like Prolog, distinguishes two programming modes:
>
> - Query mode. Expressions entered into a REPL that execute immediately. This was the subject of the first book in this reference.
> - Assertion mode. Files that contain definitions for later use.
>
> The features in this DDL section are assertion-mode constructs.

Assertion mode has two general syntactic forms: rules and facts.

4.1.1. Rules

The general form of a rule is

```
<HEAD>  <NECK>  <BODY>
```

delightql

```
short_tracks(*)
  :- track(*), milliseconds < 60000
```

In the above example:

- `short_tracks(*)` is the `HEAD`
- `:-` is the `NECK`
- `track(*), milliseconds < 60000` is the `BODY`. The body may use any DQL feature–the entire previous book applies here.

4.1.2. Facts

Facts are functor forms with grounded data.

```delightql
parent("Abraham", "Isaac")
```

This looks like a query. The syntax is context dependent. In assertion mode, i.e. in a file with rules and definitions, this syntax defines extensional data – axiomatic truths with no derivation.

4.2. THE TWO NECKS

The neck separates a rule's head from its body. Delightql has two necks, each defining a different scope and extent:

Neck	Name
`:`	Shadow neck
`:-`	Rule neck

Table 33: The two neck operators

4.2.1. Extent vs Scope

There is a difference between when and where a rule definition is available. The terminology is known by the names extent and scope.

Scope refers to the spatial visibility of a definition. It asks where an abstraction – a name – is available for use.

Extent is temporal lifetime. It asks when an abstraction is available for use. In contrast to regular programming languages, extent matters more for databases where tables and views outlive a process.

The necks in delightql map to different types of scope and extent:

- Query-extent (`:`) – Exists only for the duration of one query. These are CTEs. These also have a scope limited by a query.
- Session-extent (`:-`) – Exists for the duration of the connection. These are temporary views, tables or inlinable definitions. Their scope is determined by namespacing – the subject of a later chapter.
- Permanent-extent – Persists after disconnection. These are tables and views.

4.2.2. Shadow Neck `:`

The shadow neck defines a momentary definition with limited scope. The definition exists only for the single query in which it appears:

```delightql
short(name) : track(*), milliseconds < 60000
short(*)
```

A shadow-neck definition may shadow an existing table or view of the same name for the duration of the query. This scoping behavior is unique to the shadow neck.

Shadow-neck definitions are not DDL – they are reviewed here for syntactic similarity to rules.

4.3. RULE NECK (`:-`)

```delightql
short_tracks(*) :- audio_tracks(*), milliseconds < 60000
audio_tracks(*) :- track(*), media_type_id != 3
```

The rule neck (`:-`) creates a definition that is equivalent to SQL's `CREATE TEMP VIEW`. Delightql may choose to create temporary views to implement this abstraction, but may also choose to do expression rewriting.

4.4. RULES

The syntax of rules obey the general form:

```
<HEAD>  <NECK>  <BODY>
```

In delightql these rules may be used for any of the following:

- Views
- Tables
- Higher-order views
- ER-contexts
- Sigma predicates
- Functions

All of these use the `:-` neck. The first five define relations; functions define the special subset of relations that are functions.

4.4.1. Arity and Naming

A rule's arity is the number of arguments in its head.

```delightql
inc:(x) :- x + 1          // arity 1
add:(x, y) :- x + y       // arity 2


contact(customer_id, email) :- customer(*) // arity 2
band(*) :- artist(*)   // arity determined by body
```

All definitions – functions, sigma predicates, higher-order rulesviews, tables, facts – must have the same fixed arity.

The following are forbidden:

delightql

```
add:(x) :- x + 1          // arity 1
add:(x, y) :- x + y       // arity 2
```

delightql

```
contact(customer_id, email) :- customer(*)
contact(customer_id, email, phone)  :- customer(*)
```

If you can write `foo(*)`, all `foo` definitions must agree on arity. With argumentative heads, agreement is stricter: same arity AND same names at each position.

This rule is stricter than most languages where a name can be overloaded by arity. Delightql insists that there can only be one named entity per namespace and that that entity always have the same arity. The only place where this rule is relaxed is common expressions which can shadow tables, views, and functions.

4.5. HEAD SEMANTICS

The head of a rule names the entity and declares what it exposes. How the head is written determines the output schema and how multi-clause entities combine.

4.5.1. Two Head Forms

4.5.1.1. Glob head

delightql

```
young(*)
  :- employee(*), birth_date > "1970-01-01"
```

The glob `*` passes through whatever the body produces. The entity inherits its schema from the body. This is the permissive form – the head makes no claim about column names or count.

4.5.1.2. Argumentative head

delightql

```
young(last_name, birth_date)
  :- employee(*), birth_date > "1970-01-01"
```

The head declares the entity's output schema: exactly `last_name` and `birth_date`, in that order. Every clause must satisfy this contract –

the body must produce columns with those names (among possibly others).

The body may be wider than the head. The head projects from the body by name, so wide source tables need not be narrowed in the body before the head can name the subset. If the body does not produce a column named in the head, then it is an error.

4.5.2. Ground Terms in the Head

A head position can hold a ground term (a literal) instead of a free variable (a column name).

delightql

```
bracket("old" as tag, last_name, first_name)
  :- employee(*), birth_date < "1960-01-01"
```

Ground terms in the head inject constants into the output and provide choice semantics for multi-clause disjunctive rules.

4.5.3. Multi-Clause Rules

delightql

```
bracket("old", last_name, first_name)
  :- employee(*), birth_date < "1960-01-01"
bracket("young", last_name, first_name)
  :- employee(*), birth_date > "1970-01-01"
bracket(title, last_name, first_name)
  :- employee(*)
```

Multi-headed rules with the exact same named and shaped head are called disjunctive rules. The disjunctive in disjunctive rules is a reference to the logical `OR` and manifests in different ways depending on the context:

- sum types in algebraic data types
- tagged unions and/or variants in many programming languages
- union in set theory (and therefore SQL)
- choice in grammar rules
- `OR` in logic

The meaning of multi-headed clauses is exactly UNION ALL.

4.5.3.1. Disjunctive Rules: Consistency of Head Forms

All clauses of the same entity must use the same head form – either all glob or all argumentative. Mixing is an error:

delightql

```
// OK: all glob
place(*) :- customer(*) |> (city, country)
place(*) :- employee(*) |> (city, country)
```

delightql

```
// OK: all argumentative
place(city, country) :- customer(*)
place(city, country) :- employee(*)
```

delightql

```
// ERROR: mixed head forms
place(*) :- customer(*) |> (city, country)
place(city, country) :- employee(*)
```

4.5.4. Disjunctive Rules: Union Semantics by head form

Glob heads and argumentative heads use the exact same union strategy. Both syntaxes must publish exactly the same heading: identical column names in identical order. The programmer must explicitly reconcile disagreement across multiple bodies.

4.5.4.1. Column naming with ground terms

Argumentative heads require strict agreement: every clause must have the same number of positions, and free variables at each position must use the same name across all clauses. If clause 1 has `(name , age)` and clause 2 has `(age , name)` then this is an error.

Ground terms (constants) are nameless – they do not contribute a column name for a position. Free variables provide names. The rules:

- If a position has free variables in one or more clauses, all those free variables must use the same name. Constants in other clauses are compatible (they provide a value but no name).
- Conflicting names at the same position is an error.

delightql

```
// OK: title names position 1; constants in other clauses are compatible
bracket("old", last_name, first_name)
  :- employee(*), birth_date < "1960-01-01"
bracket("young", last_name, first_name)
  :- employee(*), birth_date > "1970-01-01"
bracket(title, last_name, first_name)
  :- employee(*)
```

delightql

```
// ERROR: position 1 named title in one clause, city in another
bracket("old", last_name, first_name)
  :- employee(*), birth_date < "1960-01-01"
bracket(title, last_name, first_name)
  :- employee(*)
bracket(city, last_name, first_name)
  :- employee(*)
```

4.6. HIGHER-ORDER RULES

Higher-order rules are rules that accept tables or scalars as parameters – inputs that further instantiate a templated relation. SQL calls these table-valued functions; Prolog would call them higher-order predicates.

All higher-order rules return a table value. It is this specific quality that permits us calling this abstraction "higher-order" and not the fact that the inputs may be tables themselves.

A programmer can create their own higher-order rules and use them in all places where a table is allowed. With a certain kind of definition – one in which the final parameter is itself a table – a higher-order rule may also be used after a pipe. Given this definition:

delightql

```
clean_employees(T(*))(*) :-
  T(*)
    |> $(trim:())(last_name, first_name)
    |> $(date:())(birth_date, hire_date)
```

`clean_employees` can follow a pipe:

delightql

```
employee(*)
  |> clean_employees(*)
```

Higher-order rules are also called input-moded rules.

4.6.1. Definition Syntax

A higher-order rule has two sets of parentheses: the first for input parameters the second for output (and occasionally input – via unification) columns.

delightql

```
genre_track_count(T(*), G(*))(genre, track_count) :-
  T(*), G(*.(genre_id))
    |> %(G.name as genre ~> count:(*) as track_count)
```

In the above example, the parameters `T(*)` and `G(*)` are glob parameter functors and denote that two tables (or lower-order relations) are expected as inputs. The `(*)` in the parameter functor name signals to the compiler that the body will reference these tables' columns by name.

The term higher-order rules is the standard usage throughout this reference, but an equally valid term is input-moded rules.

delightql

```
long_tracks(n, T(*), G(genre_id, genre))(genre, track_name) :-
  T(*), G(*.(genre_id)), milliseconds > $.n
    |> (genre, name as track_name)
```

The first set of parentheses, the ones closer to the name of the rule, should not be understood as a suite of dimensions that may be tables – although this is true – but as parameters that must be instantiated and passed in. In prolog, this sort of declaration is called a mode and is used to indicate which dimension needs to be instantiated vs which ones may be output only. In delightql input-moded rules require the parameters of the first parentheses to be input, and may allow the columns of the second set of parentheses to be input – though this is called unification and or grounding.

4.6.2. Direct Invocation

Tables can be passed in as parameter arguments. Given this definition:

```delightql
genre_track_count(T(*), G(*))(genre, track_count) :-
  T(*), G(*.(genre_id))
    |> %(G.name as genre ~> count:(*) as track_count)
```

a call passes two tables:

```delightql
genre_track_count(track(*), genre(*))(*)
```

The call site mirrors the definition head: each table parameter in this example is a full functor expression.

Because call-site arguments are relation expressions, they can compose:

```delightql
genre_track_count(
  track(*, milliseconds > 300000),
  genre(*)
)(*)
```

Here the first argument is a filtered relation.

4.6.3. Piped Invocation

Pipes can be used on any higher-order predicate that takes a table-valued parameter:

```delightql
clean_employees(T(*))(*) :-
  T(*)
    |> $(trim:())(last_name, first_name)
    |> $(date:())(birth_date, hire_date)
```

```delightql
employee(*)
  |> clean_employees(*)
```

The piped relation fills the last parameter. The `(*)` after the rule name is the output schema.

Chaining is possible:

```delightql
mask_phone(mask_value,T(*))(*) :-
  T(*) |> $$($.mask_value as phone)
```

```delightql
employee_2024(*)
  |;| employee_2025(|> *(job_title as title, manager_id as
reports_to))
  |> clean_employees(*)
  |> mask_phone("+1 (***) ***-****")(*)
```

Note. As with function pipes, the relation is piped into the last parameter of the higher-order predicate. If the higher-order predicate has multiple parameters, the other values must be set.

Multi-parameter piped invocation. When the piped relation is not the last parameter, use `@` (the f-param placeholder) to mark where it goes – the same syntax as function pipes:

```delightql
// Definition: table first, scalar second
tagged(T(*),label)(*) : T(*) |> +($.label as tag)

// Direct invocation (always works):
tagged(customer(*),"vip")(*)

// A `:` definition lasts one query, so it is restated:
tagged(T(*),label)(*) : T(*) |> +($.label as tag)

// Piped invocation with @:
customer(*) |> tagged(@,"vip")(*)
```

The `@` tells the compiler which parameter receives the piped relation. Without `@`, the piped relation fills the last parameter by default – which fails when the last parameter is a scalar.

4.6.4. Parameters to Higher-order rules

Higher-order parameters come in five flavors, distinguished by syntax in the definition head:

Form	Kind	Name
`T(*)`	table, nominally accessed	glob parameter functor
`T(a, b)`	table, positionally accessed	argumentative parameter functor
`n`	scalar value, read as `$.n`	scalar parameter
`f:()`	function value	function parameter
`P(... T(*))(*)`	higher-order residual rule	rule-valued parameter

The syntax alone tells the language what each parameter accepts. Capitalization is conventional – programmers may uppercase table

parameters and lowercase scalars for readability, but the language does not require it.

4.6.4.1. Scalar Parameters

A bare identifier without parentheses in the head declares a scalar parameter. The body reads it as `$.name`: the sigil `$.` glued to the parameter's name. A bare name in the body is always a column.

```delightql
big_markets(T(*), total_floor, min_count)(*) :-
  T(*), total > $.total_floor,
    customer(*.(customer_id))
    |> %(country ~> count:(*) as invoice_count),
    invoice_count > $.min_count
```

```delightql
big_markets(invoice(*), 10, 4)(*)
```

4.6.4.1.1. Parameters and columns do not compete

Because a parameter is always marked, a parameter and a column may share a name. Adding a column to an input can never redirect a reference to the parameter, and renaming a parameter never changes which column a bare name reads:

```delightql
above(total, T(*))(*) : T(*), total > $.total
above(20, invoice(*))(*)          // invoices whose own Total column exceeds
20
```

A pipe changes which columns are in view, never what `$.total` reads:

```delightql
h(total, T(*))(*) :-
  T(*) |> ((unit_price * quantity) as total) |> (total, $.total
as supplied)
```

Here the final `total` is the column the previous stage created, and `$.total` is the parameter.

4.6.4.1.2. Every named parameter is read

A family must use every scalar parameter it names. A position counts as used when some clause reads it as `$.name`, or when some clause grounds it with a constant in its head. Otherwise the definition refuses where it is declared (`semantic/ddl/head/unused_scalar`):

```delightql
h(total, T(*))(*) :- T(*), total > 20       // refused: `Total` here is a
column, and
                                            // the parameter `Total` is
read by no clause
```

4.6.4.1.3. Clause-local names

Scalar formal names are local to each clause. For example, `h(k)(*)` with body `_(v @ $.k)` and `h(j)(*)` with body `_(v @ $.j + 100)` bind the same argument position. Calling `h(7)(*)` returns 7 and 107 in either clause order. The same positional binding applies to query-local rules and effect rules. Family metadata labels this input `argument 1`; neither local name is the family's parameter name. Public output-head agreement is unchanged.

4.6.4.1.4. Where `$.name` stands

`$.name` is a value. It stands wherever a value stands, including:

- a slot, where it constrains the column rather than binding one: `T($.k, v)`;
- an argument forwarded to another rule: `inner($.k, T)(*)`;
- a bound or an ordinal, which take the parameter's literal actual: `T(*)`, `#< $.n` and `|$.n|`.

It never names or addresses a column, so it cannot follow `as`, stand in a selector, or appear in a head.

4.6.4.1.5. Nested definitions

A definition written inside a higher-order body may read the enclosing rule's parameters. `$.name` searches the enclosing higher-order clauses, nearest first. A nested higher-order rule's own parameter of the same name shadows the outer one. Each invocation of the outer rule hands its nested definitions its own actuals:

```delightql
outer(limit)(*) :-
    upto(s)(v) : _(v @ $.s)
    upto(s)(v) : upto($.s)(*), v < $.limit |> (v + 1 as v)
    upto(1)(*)
```

Value functions, contextual functions, truth rules and lambdas keep bare parameter names, and they do not interrupt this search: a nested value function may combine its own bare `x` with an enclosing `$.limit`. A `$.name` that no enclosing higher-order clause declares refuses where it is written (`semantic/resolution/parameter`).

4.6.4.2. Glob Parameter Functors

An glob parameter functor `T(*)` is structurally/duck typed: the body references columns by name, and any table that has those columns is accepted regardless of extra columns.

```delightql
clean_employees(T(*))(*) :-
  T(*)
    |> $(trim:())(last_name, first_name)
    |> $(date:())(birth_date, hire_date)
```

The parameter `T(*)` accepts any table with `last_name`, `first_name`, `birth_date`, and `hire_date` columns.

4.6.4.3. Argumentative Parameter Functors

An argumentative paramter functor `T(a, b)` is positionally typed: the input must have exactly two columns, and they are renamed to `a` and `b` inside the body. The caller's original column names are overwritten.

```delightql
foo(T(label, value))(*) :-
  T(*), value > 10 |> (label)
```

The names `label` and `value` are column aliases available in the body. The definition simultaneously declares the arity (two columns) and provides names for positional access.

The advantage of the argumentative functor is in the calling convention, called scalar lifting. Because the definition declares a positional contract, a call site may pass bare scalars instead of a table:

```delightql
foo("first", 12)(*)
```

The scalars are positionally matched to the declared columns `label` and `value` and lifted into a one-row table. This cascades to stacked notation:

```delightql
foo("first", 2; "second", 20)(*)
```

which sugars explicit anonymous tables:

```delightql
foo(_("first", 2; "second", 20))(*)
```

but still allows pipe invocation:

```delightql
// genre_2025 is (name, genre_id): name binds label, genre_id binds value
genre_2025(*)
  |> foo(*)
```

Or explicit functor invocation:

```delightql
foo(genre_2025(*))(*)
```

Scalar lifting requires a positional contract – a glob parameter functor cannot accept inline scalars because there is no declared arity to match against.

4.6.4.3.1. The **&** Rule

& is required only when using scalar lifting with an argumentative functor alongside other parameters.

When every table argument is passed as a functor expression, the parentheses disambiguate each argument. Commas separate parameters as usual – no **&** needed. **&** is the cost of the scalar-lifting shorthand: when bare scalars fill an argumentative functor, **&** marks where one argument ends and the next begins.

4.6.4.4. Higher-order Parameter Chart

Definition	Functor call site	&?
`f(T(*), V(*))`	`f(users(*), orders(*))(*)`	no
`f(T(*), V(*))`	`users(*) |> f(orders(*))(*)`	no
`f(T(*), n)`	`f(users(*), 10)(*)`	no
`f(T(*), V(*), n)`	`f(users(*), orders(*), 10)(*)`	no
`f(T(x, y))`	`f(data(col1, col2))(*)`	no
`f(T(x, y))`	`f(_(1, 2; 10, 20))(*)`	no
`f(T(x, y))`	`_(1, 2; 10, 20) |> f(*)`	no
`f(T(*), V(x, y))`	`f(users(*), _(1, 2))(*)`	no
`f(T(*), V(x, y))`	`f(users(*), data(col1, col2))(*)`	no
`f(T(*), V(x, y))`	`users(*) |> f(_(1, 2))(*)`	no
`f(T(*), V(x, y), n)`	`f(users(*), _(1, 2), 10)(*)`	no
`f(T(*), V(x, y), n)`	`f(users(*), _(1, 2; 10, 20), 10)(*)`	no
`f(::ns, n, V(x, y))`	`f(data::prod, "t", _(1, 2))(*)`	no
Scalar lifting (shorthand)		
`f(T(x, y))` (single)	`f(1, 2)(*)`	no
`f(T(x, y))`	`f(1, 2; 10, 20)(*)`	no
`f(T(*), V(x, y))`	`f(users(*) & 1, 2)(*)`	yes
`f(T(*), V(x, y))`	`f(users(*) & 1, 2; 10, 20)(*)`	yes
`f(T(x, y), n)`	`f("a", "b" & 10)(*)`	yes
`f(T(x, y), V(a, b))`	`f(1, 2 & 3, 4)(*)`	yes
`f(::ns, n, V(x, y))`	`f(data::prod & "t" & 1, 2)(*)`	yes

The table is divided into two regions. In the top region, every table argument uses functor syntax – `&` is never needed. In the bottom region (scalar lifting), bare scalars fill argumentative functors and `&` marks the boundaries.

Functor syntax is always available and always unambiguous. Scalar lifting is an optional shorthand for inline data – use `&` when you use it, or wrap in `_()` to avoid it.

4.7. SIGMA RULES

Sigma rules encapsulate reusable boolean logic:

delightql
```
is_high_value(amount) :- amount > 1000
```

delightql
```
track(*),
  +is_high_value(milliseconds / 1000),
  +like(name, "%Pt. 1")
```

Disjunctive Clauses

Multiple clauses with the same head are OR-ed together:

delightql
```
no_data("NA"; "N/A"; "UNKNOWN")

empty(column) :- null = column
empty(column) :- trim:(column) = ""
empty(column) :- +no_data(upper:(column))
```

delightql
```
customer(*),
  +empty(company),
  +empty(state)
```

SQL
```
SELECT *
FROM customer
WHERE (company IS NULL
       OR trim(company) = ''
       OR upper(company)
        IN ('NA', 'N/A', 'UNKNOWN'))
  AND (state IS NULL
       OR trim(state) = ''
       OR upper(state)
        IN ('NA', 'N/A', 'UNKNOWN'));
```

Requirements.

To create a sigma rule:

- The head is a relational functor with arguments
- The neck is `:-`
- The body consists of conjoined sigma predicates
- Each parameter must appear at least once in the body
- In disjunctive form, each clause must reference at least one parameter

> Sigma predicates include:
>
> - Infix comparisons: `age < 20`, `last_name = 'Johnson'`
> - Functor predicates: `+like(description, 'ipod%')`, `+between(Salary, 50000, 100000)`

- `in` statements: `state in ("MA"; "TX"; "CA")`
- Existence tests: `+other_table(...)`, `\+other_table(...)`

4.8. ER-CONTEXT RULES

ER-context rules allow a programmer to pre-define entity relationships via join conditions.

ER-context rules define the join relationships once. The `&` and `&&` operators reference them concisely.

4.8.1. Defining Relationships

An ER-rule declares how two tables join. The head uses `&` between table names; the body is the join expression:

```delightql
customer(*) & invoice(*) :-
  customer(*), invoice(*), customer.customer_id =
invoice.customer_id

invoice(*) & invoice_line(*) :-
  invoice(*), invoice_line(*), invoice.invoice_id =
invoice_line.invoice_id

invoice_line(*) & track(*) :-
  invoice_line(*), track(*), invoice_line.track_id = track.track_id
```

The `&` alone assigns this join to the default context `::normal`.

4.8.2. Multiple Contexts

The same table pair can have different join semantics in different contexts:

```delightql
customer(*) &(::support) employee(*) :-
  customer(*), employee(*), customer.support_rep_id =
employee.employee_id

customer(*) &(::local) employee(*) :-
  customer(*), employee(*), customer.city = employee.city
```

The context name is a symbol, i.e. a `::` followed by a valid identifier. The lack of a symbol means `::normal`. The following are the same:

```delightql
customer(*) & invoice(*)
customer(*) &(::normal) invoice(*)
```

4.8.3. Using Contexts

Calling the join mirrors the way in which the rule was defined:

```delightql
customer(*) & invoice(*)

customer(*) &(::local) employee(*)
```

4.8.4. Direct Join (`&`)

The `&` operator performs a direct lookup in the written context, or `::normal` when omitted. It does not search every context:

```delightql
customer(*) & invoice(*)
```

Equivalent to:

```delightql
customer(*), invoice(*), customer.customer_id = invoice.customer_id
```

Multiple `&` operators chain left to right. Each consecutive pair must have a defined ER-rule:

```delightql
customer(*) & invoice(*) & invoice_line(*)
```

Compiles to:

```delightql
customer(*), invoice(*), invoice_line(*),
  customer.customer_id = invoice.customer_id,
  invoice.invoice_id = invoice_line.invoice_id
```

4.8.5. Transitive Join (`&&`)

The `&&` operator finds a path through the ER-graph:

```delightql
customer(*) && track(*)
```

No direct `customer(*)&track(*)` rule exists, but the path does: `customer -> invoice -> invoice_line -> track`.

Ambiguity is an error. If multiple paths exist, the query fails:

```delightql
employee(*) &(::sales) customer(*) :-
  employee(*), customer(*), employee.employee_id =
customer.support_rep_id
customer(*) &(::sales) invoice(*) :-
  customer(*), invoice(*), customer.customer_id =
invoice.customer_id
employee(*) &(::sales) invoice(*) :-    // creates a cycle
  employee(*), invoice(*), employee.city = invoice.billing_city
```

```delightql
employee(*) &&(::sales) invoice(*)
// Error: Ambiguous: 2 paths from 'employee(*)' to 'invoice(*)':
//   employee(*) -> invoice(*)
//   employee(*) -> customer(*) -> invoice(*)
```

4.9. RECURSION IN RULES

Recursion in delightql emerges from self-reference. When a predicate's definition includes a clause that references the predicate itself, the definition is recursive. When a common table expression includes a clause that references the CTE itself, the CTE is recursive. Both transpile to SQL's `WITH RECURSIVE` construct.

This chapter covers the semantics of recursion in delightql, how it maps to SQL's execution model, and the constraints that model imposes.

4.9.1. Two Forms of Recursion

Delightql supports recursion in two contexts:

Recursive rules are defined in assertion mode and persist as reusable predicates:

```delightql
superior(employee, boss) :-
  employee(*), reports_to != null
    |> (employee_id as employee, reports_to as boss)
superior(employee, boss) :-
  employee(*) as e, superior(*) as s, e.reports_to = s.employee
    |> (e.employee_id as employee, s.boss)
```

Recursive CTEs are defined inline in query mode, scoped to a single query:

```delightql
superior(*) : employee(*), reports_to != null |> (employee_id as
employee, reports_to as boss)
superior(*) : employee(*) as e, superior(*) as s, e.reports_to =
s.employee
    |> (e.employee_id as employee, s.boss)
superior(*)
```

Both forms transpile to `WITH RECURSIVE`. The choice depends on whether the recursive logic is reusable (rule) or ad-hoc (CTE).

A parameterized rule may be written in either place: in a rules file with `:-`, or in the query itself with `:` as a common higher-order expression. The two recurse identically; see Higher-Order Recursive Predicates below.

4.9.2. The Anatomy of Recursion

Every recursive definition has two components:

Base clauses provide initial rows without self-reference. These are SQL's "anchor members":

```
delightql
_(n @ 1) : counter                                    // literal base
case
counter(*)

employee(*), title = "General Manager" : mgmt         // filtered base
case
mgmt(*)

employee(*), reports_to != null                       // projected
base case
    |> (reports_to as origin, employee_id as dest) : reachable
reachable(*)
```

Recursive clauses reference the predicate or CTE being defined. These are SQL's "recursive members":

```
delightql
_(n @ 1) : counter                                    // base case
counter(*), n < 100 |> (n + 1 as n) : counter
counter(*)

employee(*), title = "General Manager" : mgmt         // base case
mgmt(*) as m, employee(*) as o, o.reports_to = m.employee_id |>
(o.*) : mgmt
mgmt(*)

employee(*), reports_to != null                       // base case
    |> (reports_to as origin, employee_id as dest) : reachable
reachable(*) as r, employee(*) as e, r.dest = e.reports_to
    |> (r.origin, e.employee_id as dest) : reachable
reachable(*)
```

Base clauses must precede recursive clauses in source order: names resolve strictly left to right, and a recursive clause reads a name the base clause establishes. In a recursive CTE the base clause creates the name, so writing the recursive clause first refuses as an unresolved name ("Table not found"). A rule, or a parameterized rule written in the query, declares its name in its head, so the name can see itself; there a self-reference before any base clause refuses as `semantic/recursion/anchor_first`.

4.9.3. Evaluation Model

SQL's recursive CTEs evaluate using a working table algorithm:

1. Execute all base clauses; their results form the initial working table
2. Execute the recursive clause with the working table as input
3. The output becomes the new working table
4. Repeat until the working table is empty

5 Return the union of all iterations

This is bottom-up or co-recursive evaluation: starting from known facts it derives new facts and repeats until a fixed point is reached. It resembles dynamic programming more than classical recursion.

The critical implication: the recursive clause sees only the previous iteration's rows, not the full accumulated result. This is why certain operations are prohibited – they would require access to rows that haven't been computed yet or have already been consumed.

4.9.4. What Recursion Can Express

SQL's recursive model handles a well-defined class of problems:

Hierarchical traversal – org charts, bill of materials, folder structures:

```delightql
employee(*), reports_to = null |> (employee_id, last_name, 0 as
depth) : tree
employee(*) as e, tree(*) as t, e.reports_to = t.employee_id
    |> (e.employee_id, e.last_name, t.depth + 1 as depth) : tree
tree(*)
```

Transitive closure – reachability, ancestry, dependency graphs:

```delightql
employee(*), reports_to != null |> (reports_to as origin,
employee_id as dest) : reachable
reachable(*) as r, employee(*) as e, r.dest = e.reports_to
    |> (r.origin, e.employee_id as dest) : reachable
reachable(*) |> %(*)  // deduplicate
```

Sequence generation – numeric ranges, date series, iteration:

```delightql
_(d @ date:("2024-01-01")) : dates
dates(*), d < date:("2024-12-31")
    |> (date:(d, "+1 day") as d) : dates
dates(*)
```

Iterative computation – any algorithm expressible as "given previous state, compute next state":

```delightql
_(iter, x, target @ 0, 1.0, 2.0) : newton
newton(*), abs:((x * x) - target) > 0.0001, iter < 100
    |> (iter + 1 as iter, (x + (target / x)) / 2.0 as x,
target) : newton
newton(*) |> %(target ~> (x as sqrt) <~ #(iter desc))
```

4.9.5. What Recursion Cannot Express

The working-table model imposes fundamental limitations. These are not arbitrary restrictions – they follow from the evaluation semantics.

4.9.5.1. No Aggregation in Recursive Clauses

Aggregation requires access to multiple rows. The recursive clause sees only the working table (previous iteration), not the full accumulated result.

```delightql
// INVALID -- cannot aggregate within recursion
similar_artist(*), artist_id = 22      // Led Zeppelin
    |> (similar_id as artist_id, score as strength) : reach
reach(*) as r, similar_artist(*) as s, s.artist_id = r.artist_id,
r.strength > 0.5
    |> (s.similar_id as artist_id, r.strength * s.score as
strength) : reach  // seems ok?

// But this fails:
reach(*) as r, similar_artist(*) as s, s.artist_id = r.artist_id
    |> %(s.similar_id as artist_id ~> sum:(r.strength * s.score)
as strength) : reach  // aggregation -- NOT ALLOWED
reach(*)
```

4.9.5.2. No Subqueries Referencing the Recursive Target

A subquery inside the recursive clause cannot reference the CTE being defined:

```delightql
// INVALID -- subquery references 'paths'
employee(*), reports_to != null |> (reports_to as origin,
employee_id as dest) : paths
employee(*) as e, paths(*) as p, e.reports_to = p.dest,
    \+ paths(*, e.employee_id = dest)  // "dest not already reached" --
NOT ALLOWED
    |> (p.origin, e.employee_id as dest) : paths
paths(*)
```

The subquery `paths(*, ...)` would need to see all accumulated rows, which aren't available.

4.9.5.3. No Mutual Recursion

Two predicates cannot reference each other:

```delightql
// INVALID -- mutual recursion
even(n) :- _(n @ 0)
even(n) :- odd(m) |> (m + 1 as n)
odd(n) :- even(m) |> (m + 1 as n)
```

SQL's `WITH RECURSIVE` processes one CTE at a time. There's no mechanism for two CTEs to co-evolve.

The refusal is `semantic/recursion/mutual`, and its message names the whole cycle (`home.even -> home.odd -> home.even`). A definition that returns to itself through any other definition — a rule that uses a view whose body uses the rule — refuses the same way.

4.9.5.4. Single Self-Reference

The recursive clause may reference the target exactly once:

```delightql
// INVALID -- two self-references
employee(*), reports_to != null |> (reports_to as origin,
employee_id as dest) : paths
paths(*) as p1, paths(*) as p2, p1.dest = p2.origin
    |> (p1.origin, p2.dest) : paths
paths(*)
```

This would require joining the working table against itself, which SQL doesn't support in recursive CTEs.

4.9.5.5. Relations Only

Recursion is over relations. A value function that calls itself — with the same argument, a different one, or through other value functions — refuses as `semantic/recursion/function-form`. A sigma rule that cites itself, directly or through other sigma rules, refuses as `semantic/recursion/truth-form`, even with a base clause:

```delightql
// INVALID -- a value function has no fixpoint to re-enter
fib:(n) :- fib:(n - 1) + fib:(n - 2)
```

Define the relation and ground the argument instead: `fib(k, v)` as a recursive relation, `fib(5, v)` to read one point.

4.9.6. Termination

Recursive CTEs terminate when the recursive clause produces no new rows. This happens when:

- A `WHERE` condition filters out all candidates
- A join finds no matches
- The depth limit (`#`) is reached
- The data is exhausted (finite traversal)

Ensuring termination:

For sequence generation, always include a bound:

```delightql
_(n @ 1) : nums
nums(*), n < 1000 |> (n + 1 as n) : nums  // terminates at 1000
nums(*)
```

For graph traversal over potentially cyclic data, track visited nodes:

```delightql
similar_artist(*)
    |> (artist_id as origin, similar_id as dest, :",{artist_id},
{similar_id}," as visited) : paths
similar_artist(*) as e, paths(*) as p, p.dest = e.artist_id,
    \+like(p.visited, :"%,{e.similar_id},%")     // not already on the
path
    |> (p.origin, e.similar_id as dest, :"{p.visited}
{e.similar_id}," as visited) : paths
paths(*), origin = 22     // from Led Zeppelin: ,22,58,12, is not extended back to
22
```

For unknown depth, use `#` as a safety limit:

```delightql
employee(*), reports_to = null |> (employee_id as id, 0 as
depth) : tree
tree(*) as t, employee(*) as n, n.reports_to = t.id, # < 100
    |> (n.employee_id as id, t.depth + 1 as depth) : tree
tree(*)
```

4.9.7. UNION Badge

By default, delightql emits `UNION ALL` – duplicates across iterations are preserved. This is efficient and correct for most traversals.

For graph traversals that require the usage of `UNION` over `UNION ALL` for correctness use the UNION-BADGE where the badge `%` is postfixed to the name of the rule.

```delightql
artist_edge(*) :-
  similar_artist(*)
    |> (artist_id as src, similar_id as dst)

artist_edge(*) :-
  similar_artist(*)
    |> (similar_id as src, artist_id as dst)

connected%(*) :-
  artist(*), name = "Led Zeppelin"
    |> (artist_id)

connected%(*) :-
  connected(*) as c,
    artist_edge(*) as e,
    c.artist_id = e.src
    |> (e.dst as artist_id)
```

This applies to CTEs as well:

delightql

```
// Construct an undirected artist graph.
similar_artist(*)
  |> (artist_id as src, similar_id as dst)
  : edge

similar_artist(*)
  |> (similar_id as src, artist_id as dst)
  : edge

// Begin at Led Zeppelin.
artist(*), name = "Led Zeppelin"
  |> (artist_id)
  : connected%

// Follow every edge. UNION prevents previously reached artists
// from returning to the top of the stack
connected(*) as c,
  edge(*) as e,
  c.artist_id = e.src
  |> (e.dst as artist_id)
  : connected%

// Display the connected component.
connected(*) as c,
  artist(*) as a,
  c.artist_id = a.artist_id
  |> (a.artist_id, a.name)
  |> #(artist_id)
```

A parameterized head wears the badge in the same place, after the name and before the parameter row, in a rules file or in the query:

delightql

```
connected_to%(root)(*) :-
  artist(*), name = $.root
    |> (artist_id)

connected_to%(root)(*) :-
  connected_to($.root)(*) as c,
    similar_artist(*) as s,
    c.artist_id = s.artist_id
    |> (s.similar_id as artist_id)

connected_to%(root)(*) :-
  connected_to($.root)(*) as c,
    similar_artist(*) as s,
    c.artist_id = s.similar_id
    |> (s.artist_id as artist_id)
```

delightql

```
connected_to("Gilberto Gil")(*) as c,
  artist(*) as a,
  c.artist_id = a.artist_id
  |> (a.artist_id, a.name)
  |> #(artist_id)
```

Each argument selects its own fixpoint, and the badge removes duplicates within that fixpoint only. When the argument comes from

a caller's rows, two caller rows with the same argument each receive the whole result.

Every clause of a definition wears the same badge. A mixed set refuses as `semantic/recursion/mixed_badge`. A badge on a definition that never references itself refuses as `semantic/recursion/false_fixpoint`: there is no fixpoint for it to describe. To remove duplicates from an ordinary definition, write `|> %(*)` in the body.

4.9.8. Higher-Order Recursive Predicates

Recursive rules can be parameterized, deferring the base case:

```delightql
reports_to(boss)(employee_id, last_name) :- employee(*), last_name
= $.boss
reports_to(boss)(employee_id, last_name) :-
    employee(*) as e,
    reports_to($.boss)(*) as r,
    e.reports_to = r.employee_id
    |> (e.employee_id, e.last_name)
```

Each invocation monomorphizes to a concrete `WITH RECURSIVE`:

```delightql
reports_to("Edwards")(*)    // who reports to Edwards?
reports_to("Mitchell")(*)   // who reports to Mitchell?
```

The higher-order parameter `boss` is inlined into the anchor clause at query time. The recursive structure itself doesn't change – only the starting point.

The argument selects one fixpoint, and it stays fixed while that fixpoint runs. The self-reference `reports_to(boss)(*)` passes the same argument, so it reads the rows found so far. A self-reference that changes an argument refuses as `semantic/recursion/parameter-widening`; state that changes from one step to the next belongs in the relation's columns.

The same rule may be written in the query itself, as a common higher-order expression, with `:` in place of `:-`:

```delightql
reports_to(boss)(employee_id, last_name) : employee(*), last_name =
$.boss
reports_to(boss)(employee_id, last_name) :
    employee(*) as e,
    reports_to($.boss)(*) as r,
    e.reports_to = r.employee_id
    |> (e.employee_id, e.last_name)
reports_to("Edwards")(*)
```

It returns the same rows and refuses the same shapes as the rule, in every position: with a literal argument, with an argument read from the caller's rows, and handed to another rule as a rule value.

4.9.9. Example: Mandelbrot Set

This example demonstrates sequence generation, computational iteration, and post-recursion aggregation working together:

delightql

```
_(x@-2.0)                                    : xaxis
xaxis(*), x < 1.2
 |> (x + 0.05 as x)                          : xaxis
_(y@-1.0)                                    : yaxis
yaxis(*), y < 1.0
 |> (y + 0.1 as y)                           : yaxis
sq:(x):
  x * x
xaxis(*), yaxis(*)
 |> (0 as iter,
     x as cx,
     y as cy,
     0.0 as x,
     0.0 as y)                               : m
m(*), (sq:(x) + sq:(y)) < 4.0,
  iter < 28
 |> (iter + 1 as iter,
     cx as cx,
     cy as cy,
     (sq:(x) - sq:(y)) + cx as x,
     ((2.0 * x) * y) + cy as y)              : m
m(*)
 |> %(cx,cy ~> max:(iter) as iter )          : m2
m2(*)
 |> %(cy
        ~>
      group_concat:(substr:(" .+*#", 1+min:(iter/7,4), 1), "") as t)
                                             : a
a(*)
 ~> group_concat:(rtrim:(t),char:(0x0a))
```

The query generates a coordinate grid, runs the escape-time algorithm via recursive iteration, then aggregates the results into ASCII art – all in a single delightql expression.

4.9.10. Delightql Recursive Limits

A true fixed-point engine – like those in Datalog systems – would maintain the full set of derived facts and allow each iteration to query against it. SQL chose a simpler model. The restrictions on aggregation, subqueries, and mutual recursion all follow from this choice.

Delightql inherits these limitations because it transpiles to SQL and the semantics remain bound by the target. Where SQL's recursive CTEs fall short – self-similar tree construction, recursive aggregation, shortest-path computation – delightql falls short as well.

4.10. FACTS

Facts are body-less rules that define ground data. In Prolog terms, they represent the extensional (axiomatic) portion of a program – truths asserted without derivation.

4.10.1. Standard Facts

The notation matches Prolog (minus the terminating period):

```delightql
person(0, "Gusti", "Parlor", "gparlor0@phoca.cz")
person(1, "Diane-marie", "McHenry", "dmchenry1@dot.gov")
person(2, "Ced", "Mainds", "cmainds2@goo.ne.jp")
person(3, "Bren", "Berndsen", "bberndsen3@goodreads.com")
```

Standard facts sharing the same functor name must be co-located – no other definitions may appear between them.

4.10.2. Stacked Facts

Define tabular data with headers:

```delightql
staff(
  employee_id , first_name     , last_name
  -------------------
  0  , "Gusti"       , "Parlor" ;
  1  , "Diane-marie" , "McHenry" ;
  2  , "Ced"         , "Mainds"
)
```

The syntax mirrors anonymous tables, but anonymous tables are query-mode constructs (inline data) while stacked facts are assertion-mode constructs.

```delightql
// Anonymous table (query mode)
_(first_name, last_name
  -------------
  "Gusti"       , "Parlor" ;
  "Diane-marie" , "McHenry"
)
```

```delightql
// Stacked fact (assertion mode)
names(first_name, last_name
  -------------
  "Gusti"       , "Parlor" ;
  "Diane-marie" , "McHenry"
)
```

4.10.3. Default Implementation as Views

Delightql implements facts as views by default, not tables:

delightql

```
staff(
  employee_id , first_name     , last_name
  -------------------
  0  , "Gusti"       , "Parlor" ;
  1  , "Diane-marie" , "McHenry" ;
  2  , "Ced"         , "Mainds"
)
```

SQL

```
CREATE TEMP VIEW staff AS
  SELECT 0 AS employee_id, 'Gusti' AS first_name, 'Parlor' AS
last_name
  UNION ALL SELECT 1, 'Diane-marie', 'McHenry'
  UNION ALL SELECT 2, 'Ced', 'Mainds';
```

This seems counterintuitive – facts are data, so why not tables? The justification: typical delightql files contain only a handful of facts (test fixtures, configuration, lookup tables). For small datasets, the difference between a view over literal values and a table with inserted rows is negligible. Views avoid the overhead of table creation and cleanup.

4.10.4. Sparse Stacked Facts

Stacked facts support the same sparse column syntax as anonymous tables. Mark optional columns with `?` in the header, then use `_(col @ val)` fills in data rows:

delightql

```
config(
  key, value, description?, deprecated?
  ------------------------------------
  "timeout",  30 ;
  "retries",  3,   _(description @ "max retry count") ;
  "legacy",   1,   _(deprecated @ "true")
)
```

This is equivalent to the fully-expanded form:

delightql

```
config(
  key, value, description, deprecated
  ----------------------------------
  "timeout",  30,   null,               null ;
  "retries",  3,    "max retry count",  null ;
  "legacy",   1,    null,               "true"
)
```

Sparse columns reduce noise in metadata definitions and configuration facts where most rows only set a few optional properties.

4.11. CITATION RULES

A citation is the shorthand invocation of a zero-argument value function: `:greeting` reads the function `greeting`, whether it was defined as `greeting:() :- "hello"` or as the paren-less `greeting :- "hello"`. It is the same application as `greeting:()`; nothing distinguishes them after parsing.

delightql

```
esc :- char:(27)
hue :- 31
red:(s) :- :"{:esc}[{:hue}m{s}{:esc}[0m"
```

A citation composes wherever a value does — in arithmetic, in a template hole, as an argument. Qualified, the mark belongs to the leaf, after the namespace and its dot: `ansi.:esc`, the same function as `ansi.esc:()`. The mark never precedes the path: `:ansi.esc` is not a spelling.

A citation is a reference to a defined function and can alias another citation's value; contrast the symbol `::name`, which is a self-valued constant needing no definition.

4.12. FUNCTION RULES

Functions are relations with a functional dependency between input and output columns. Where a relation may have many outputs for a given input, a function has exactly one. Delightql supports several syntactic forms for defining functions, each suited to different use cases.

4.12.1. Stacked Notation (Named Case)

The stacked form defines functions as lookup tables with explicit input-output mappings:

delightql

```
title_kind(
  title          -> kind
  -----------------
  "IT Manager"   -> "tech";
  "IT Staff"     -> "tech";
  _              -> "other"
)
```

The `->` separates inputs (left) from outputs (right). The header row names the columns; subsequent rows provide the mappings. The `_` matches any input not explicitly listed.

Despite the visual similarity to anonymous table stacked notation, this is an assertion-mode construct – it defines a reusable function, not inline data.

As in anonymous tables, the dashed separator and `@` are interchangeable, so a short mapping fits on one line:

delightql
```
region(country -> region @ "USA" -> "home"; "Canada" -> "home"; _
-> "abroad")
```

delightql
```
customer(*) |> (last_name, country, region:(country) as region)
```

A function is defined in one form only: stacked notation and rule clauses (`region:(c) :- …`) cannot be mixed for the same name.

Invocation:

delightql
```
employee(*) |> +(title_kind:(title) as kind)
```

SQL
```
SELECT *,
  CASE title
    WHEN 'IT Manager' THEN 'tech'
    WHEN 'IT Staff' THEN 'tech'
    ELSE 'other'
  END AS kind
FROM employee;
```

Multi-column inputs:

delightql
```
tax_rate(
  country, state -> rate
  --------------------------
  "USA", "CA"    -> 0.0725;
  "USA", "TX"    -> 0.0625;
  "Canada", "ON" -> 0.13;
  "Canada", "AB" -> 0.05;
  _              -> 0.0
)
```

delightql
```
invoice(*) |> +(tax_rate:(billing_country, billing_state) as tax)
```

4.12.2. Rule Form

For computed functions, use the rule form:

delightql
```
plus_two:(x) :- x + 2
```

delightql
```
invoice(*) |> +(plus_two:(total) as incremented)
```

SQL
```
SELECT *, total + 2 AS incremented FROM invoice;
```

The body is any domain expression. The function returns its evaluation.

4.12.2.1. Zero-Argument Functions and Citations

A function with no parameters is defined with an empty head or, as shorthand, with no parentheses at all. The two spellings define the same kind of value function:

delightql

```
greeting:() :- "hello"
tab :- char:(9)
```

Either is invoked as an ordinary application or, as shorthand, as a citation — the name marked with a leading `:`:

delightql

```
artist(*) |> +(greeting:() as g, :tab as sep)
```

The body is any domain expression, not only a literal. The usual arity and clause laws apply: `greeting:("x")` refuses, and defining `greeting` twice without guards refuses whichever spelling each definition used.

A qualified citation puts the mark on the leaf, where the application form puts its own mark: `lib::text.:greeting` invokes the same function as `lib::text.greeting:()`. `:lib::text.greeting` is not a spelling.

4.12.2.2. Disjunctive Clauses

Multiple clauses create conditional functions. Clauses are evaluated top-to-bottom; first match wins:

delightql

```
fizzbuzz:(n | (n % 15) = 0) :- "fizzbuzz"
fizzbuzz:(n | (n % 3) = 0)  :- "fizz"
fizzbuzz:(n | (n % 5) = 0)  :- "buzz"
fizzbuzz:(n)              :- n
```

The guard condition follows `|` in the head. If the guard fails, the next clause is tried.

delightql

```
_(n @ 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15) |>
(fizzbuzz:(n) as result)
```

SQL

```
SELECT
  CASE
    WHEN n % 15 IS NOT DISTINCT FROM 0 THEN 'fizzbuzz'
    WHEN n % 3 IS NOT DISTINCT FROM 0 THEN 'fizz'
    WHEN n % 5 IS NOT DISTINCT FROM 0 THEN 'buzz'
    ELSE n
  END AS result
FROM (SELECT 1 AS n UNION ALL SELECT 2 UNION ALL SELECT 3
      UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6
      UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9
      UNION ALL SELECT 10 UNION ALL SELECT 11 UNION ALL SELECT
12
      UNION ALL SELECT 13 UNION ALL SELECT 14 UNION ALL SELECT
15);
```

Hailstone sequence example:

delightql

```
next_hailstone:(x | (x % 2) = 0) :- x / 2
next_hailstone:(x)               :- (x * 3) + 1
```

The `@` marks where the piped value is inserted when the function takes multiple arguments.

4.12.3. Higher-Order Functions

Functions are inherently higher-order: any function can accept other functions as parameters. Mark function parameters with colon-functor syntax `f:()` in the signature to distinguish them from scalar parameters:

delightql

```
apply:(f:(), x) :- f:(x)
```

The `f:()` declares that the first parameter is a function. The body calls whatever function was passed in. Scalar parameters are bare names as usual.

Invocation:

delightql

```
employee(*) |> (apply:(upper:(), first_name) as formatted)
```

The call site passes `upper:()` (a curried function) and `first_name` (a column) as two arguments. Arity matching works the same as regular functions: `apply` has arity 2, and the call provides 2 arguments.

Multiple function parameters:

delightql

```
chain:(f:(), g:(), x) :- x >> f:() >> g:()
```

delightql

```
employee(*) |> (chain:(upper:(), trim:(), first_name) as cleaned)
```

Lambda as function argument:

delightql
```
apply_twice:(f:(), x) :- x >> f:() >> f:()
```

delightql
```
invoice(*) |> (apply_twice:(:(@ * 2), total) as quadrupled)
```

Mixed function and scalar parameters:

delightql
```
transform_and_compute:(f:(), g:(), value, multiplier) :-
  f:(value) >> g:() >> :(@ * multiplier)
```

With conditional logic:

delightql
```
apply_if_long:(f:(), value) :-
  _:(length:(value) > 5 -> f:(value); _ -> value)
```

No double parentheses. Unlike higher-order views, higher-order functions use a single set of parentheses. Views need double parens because they operate on two modal categories – input-only parameters (tables) and bidirectional columns. Functions have no such distinction: everything is a value in, scalar out. See Higher-Order Rules for the full rationale.

4.12.4. Contextual Functions

The `..` sigil indicates a function that captures variables from its invocation context:

delightql
```
bitrate:(..) :-
  (bytes * 8.0)
    >> :(@ / milliseconds)
    >> round:(@, 1)
```

delightql
```
track(*) |> (name, bitrate:(..) as kbps)
```

The function analyzes its body for free variables (`bytes`, `milliseconds`) and expects them from the calling relation. This is structural typing for functions – any relation with those columns can use the function.

Mixed parameters:

Combine context capture with explicit arguments:

delightql
```
line_amount:(.., discount) :-
  (unit_price * quantity)
    >> :(@ * (1 - discount))
    >> round:(@, 2)
```

delightql
```
invoice_line(*) |> (
  line_amount:(.., 0) as full_price,
  line_amount:(.., 0.25) as discounted
)
```

Named context:

Explicitly declare captured variables:

```delightql
net_amount:(..{unit_price, quantity}, discount) :-
  (unit_price * quantity)
    >> :(@ * (1 - discount))
    >> round:(@, 2)
```

This makes dependencies visible in the signature and allows overriding context with explicit values:

```delightql
invoice_line(*) |> (
  net_amount:(.., 0) as from_context,
  net_amount:(1.99, quantity, 0) as explicit
)
```

4.13. NAMESPACES

Namespaces organize definitions and data. They provide isolation, qualification, and a mechanism for code reuse across databases.

4.13.1. What Namespaces Are

A namespace is a container for entities – tables, views, rules, functions. Every entity lives in exactly one namespace.

Namespaces are hierarchical, separated by `::`:

```
data::production
lib::analytics
scripts::etl
```

Entities within a namespace are accessed with the period `.`:

```
data::production.users(*)
lib::string.clean_name:(text)
scripts::etl.daily_load!(*)
```

The `::` separates namespace parts. The `.` separates namespace from entity.

4.13.2. Namespace Types

Namespaces fall into four categories based on what they contain and how they're used.

4.13.2.1. Pure Rules Namespaces

Contain functions, sigma predicates, transpilation rules, and higher-order views with no external references. Portable – they don't depend on any database – but may depend on other pure namespaces.

```delightql
// In lib::string
clean_name:(text) :- text >> trim:() >> upper:()
format_email:(name, domain) :- (name ++ "@") ++ domain
```

Pure namespaces can be used anywhere. They have no data dependencies to resolve.

4.13.2.2. Derived Rules Namespaces

Contain rules that reference external tables. These namespaces come in two forms:

Groundable – has free variables (unqualified table references):

```delightql
// In lib::analytics (groundable)
long_tracks(*) :- track(*), milliseconds > 600000    // 'track' is a free variable
```

The reference to `track` must be resolved before use. See Grounding.

Pre-grounded – all references are qualified:

```delightql
// In lib::analytics (pre-grounded)
long_tracks(*) :- main.track(*), milliseconds > 600000
```

No free variables. Ready to use immediately, but tied to a specific data namespace.

4.13.2.3. Data Namespaces

Map to physical database connections. Contain tables and views introspected from the database.

```delightql
mount!("example.sqlite", "data::sales")(*)
// Now: data::sales.invoice(*), data::sales.customer(*)
```

Data namespaces are the ground truth – they hold actual data.

When a qualified mount names missing ancestors, `mount!` creates those path segments as structural `container` namespaces. For example, mounting `data::sales` creates a queryable `data::(*)` parent and a `data::sales` data leaf. The parent has no database source of its own; mount lifecycle operations apply to the leaf.

4.13.3. Namespace Classification

A namespace's type is determined by its contents, not its path:

If it contains...	It's classified as...
Only functions, sigma predicates, pure HO-views	Pure

If it contains...	It's classified as...
Any rule referencing external tables	Derived

Table 36: Namespace classification by contents

Data namespaces are separate – they're created by `mount!` and contain database tables, not rules. They're the target of grounding, not the subject.

4.13.4. Conventional Prefixes

By convention, namespace paths indicate their type:

Prefix	Intended for	Typically created by
`data::`	Database connections	`mount!`
`lib::`	Pure and derived rules	`consult!`
`main`	Default working namespace	Implicit

Table 37: Conventional namespace path prefixes

These are conventions only, not constraints. The system determines namespace type by analyzing contents, independent of path.

4.13.5. Built-in Namespaces

Several namespaces exist automatically.

4.13.6. main

The default working data namespace for the REPL. When you use the REPL interactively, you're operating with the entities of `main` auto-enlisted:

```delightql
usa_customers(*) :- customer(*), country = "USA"
// Equivalent to: main.usa_customers(*)
```

4.13.7. home

The `home` namespace is the user's default library namespace. Like `main` its entities are auto-enlisted. In-line DDL establishes its entities into `home`.

```delightql
(~~ddl times_two:(x) :- x*2 ~~) _(a@2;3;5;7) |> +( times_two:
(a) as twice)
```

When you `enlist!` a namespace in the REPL, you're making its entities available in `home` without qualification.

More generally, every execution context has a working namespace. In the REPL, it is `home`. During `consult!("file.dql", "lib::foo")(*)`, the working context is `lib::foo` – the definitions in that file go into `lib::foo`. During `run!("file.dql")(*)`, the working context inherits from the caller.

4.13.7.1. lib::std::prelude

Core pseudo-predicates, universally available. This is a partial list; the catalog call `std::prelude::(*)` lists the complete set.

Pseudo-predicate	Purpose
`mount!`	Load database connection
`consult!`	Load DQL rules file
`enlist!`	Enable unqualified access
`delist!`	Remove enlisted namespace
`run!`	Execute query file

Table 38: Core pseudo-predicates in lib::std::prelude

The DML directives (`insert!`, `update!`, `delete!`) are covered in Data Manipulation Language (DML); multi-step effect programs in Effects and Directives.

No explicit enlist needed – these are available everywhere.

[30]

30 The pseudo-predicates that load and inspect namespaces are themselves defined in a namespace. This circularity is intentional – the system is self-describing. You can query `sys::entities.entity(*)` to see all built-in entities, including these pseudo-predicates.

4.13.7.2. lib::std::predicates

Built-in sigma predicates, universally available:

```delightql
track(*), +like(name, "A%"), +between(milliseconds, 180000, 240000)
```

The catalog call `std::predicates::(*)` lists the complete set.

4.13.7.3. sys::* (Introspection)

Metadata namespaces for system introspection:

Namespace	Contains
`sys::ns`	Namespaces, enlisted relationships, activated entities
`sys::entities`	Entities, types, references
`sys::cartridges`	Cartridges, languages, source types
`sys::execution`	Compilations, recursion stack (diagnostics)

Table 39: System introspection namespaces

Not auto-enlisted. Query explicitly when needed:

```delightql
sys::ns.namespace(*)
sys::entities.entity(*), type = 10    // database tables
sys::cartridges.cartridge(*)
```

4.13.8. Pseudo-predicates and "attaching" context

There are several functor forms ending with an exclamation point that are used to bring rules, facts, and data into scope and within a namespace.

4.13.8.1. mount! – Database Connections

Opens a database and introspects its tables:

```delightql
mount!("example.sqlite", "data::sales")(*)
```

Side effects:

1. Creates namespace `data::sales`
2. Introspects tables and views
3. Registers entities in namespace

After mounting, tables are accessible:

```delightql
mount!("example.sqlite", "data::sales")(*)
data::sales.invoice(*)
data::sales.customer(*)
```

4.13.8.2. consult! – DQL Rules

Loads a `.dql` file containing rules:

```delightql
consult!("analytics.dql", "lib::analytics")(*)
```

The file contains rule definitions. These are pre-grounded – every table they read is qualified – so they can be used as soon as they are loaded:

```delightql
// analytics.dql
long_tracks(*) :- main.track(*), milliseconds > 600000
big_invoices(*) :- main.invoice(*), total > 20
```

Side effects:

1. Creates namespace `lib::analytics`
2. Parses file
3. Creates session views/functions
4. Registers entities in namespace

Rules are now accessible (qualified or via enlist):

```delightql
consult!("analytics.dql", "lib::analytics")(*)
lib::analytics.long_tracks(*)
```

A groundable namespace is loaded the same way, but its rules refuse until `ground!` binds their free names (see Grounding).

4.13.8.3. enlist! – Unqualified Access

Makes a namespace's entities available without qualification:

```delightql
consult!("analytics.dql", "lib::analytics")(*)
enlist!("lib::analytics")(*)

// Now can write:
long_tracks(*)
// Instead of:
lib::analytics.long_tracks(*)
```

Enlisting doesn't load anything – the namespace must already exist.

4.13.8.4. delist! – Remove Enlisted Namespace

Removes a namespace from enlisted scope:

```delightql
consult!("analytics.dql", "lib::analytics")(*)
enlist!("lib::analytics")(*)
delist!("lib::analytics")(*)

lib::analytics.long_tracks(*)          // still works (qualified)
```

The bare name is gone:

```delightql
consult!("analytics.dql", "lib::analytics")(*)
enlist!("lib::analytics")(*)
delist!("lib::analytics")(*)

long_tracks(*)                         // Error: not found
```

4.13.9. Grounding

Groundable namespaces have free variables – references to tables that aren't defined in the namespace. Grounding binds those variables to a data namespace.

4.13.9.1. The Problem Again

```delightql
// tracks.dql, consulted into lib::analytics (groundable)
long_tracks(*) :- track(*), milliseconds > 600000
```

`track` is referenced but not defined. This namespace can't be used until `track` is bound to an actual table.

4.13.9.2. Grounding

For a permanent binding, use `ground!`:

```delightql
mount!("example.sqlite", "data::production")(*)
consult!("tracks.dql", "lib::analytics")(*)
ground!("data::production", "lib::analytics",
"lib::analytics_prod")(*)
lib::analytics_prod.long_tracks(*)
```

All three arguments are required. The first two are namespace paths; the third is a string literal naming the new namespace.

This:

1. Validates all entities in `lib::analytics` against `data::production` (strict validation). If any entity has an unresolved table reference, the entire operation fails – nothing is created.
2. Creates a new namespace `lib::analytics_prod`
3. Copies all entities with free variables bound to `data::production`

The result is a new namespace, not a mutation of the original.

4.13.9.3. Constraints

No intersection. The ground namespace and groundable namespace cannot share entity names. If both define `track`, grounding is ambiguous and fails.

Same database technology. Cross-database grounding (e.g., SQLite namespace against PostgreSQL namespace) is not supported.

4.13.10. Imprinting

Session-scoped entities (created with `:-`) disappear when the session ends. Imprinting makes them permanent.

More so than most of what we've discussed before: this is where actual SQL DDL will be generated.

4.13.10.1. Imprinting syntax

`imprint!` creates exactly what the library's manifest lists. The manifest is an `imprinting` rule in the library's `_internal` companion namespace, naming each entity and whether it becomes a view or a table:

delightql

```
// schema.dql
long_tracks(*) :- track(*), milliseconds > 600000

(~~ddl:"_internal"
  imprinting(*) :- _(entity, materialization, extent @
"long_tracks", "view", "permanent")
~~)
```

delightql

```
mount!("example.sqlite", "data::production")(*)
consult!("schema.dql", "lib::schema")(*)
imprint!("lib::schema", "data::production")(*)
data::production.long_tracks(*)
```

Imprinting:

1. Validates that all entities can resolve against the target (strict validation)
2. Generates DDL (`CREATE VIEW`, `CREATE TABLE`)
3. Executes DDL on the target database
4. Entities now exist permanently in the data namespace
5. Consumes the source: `lib::schema` is vacated, and its definitions are archived beside the target as `data::production::_0_blueprint`

4.13.10.2. Grounding and Imprinting

Grounding and imprinting are highly related: where grounding proves compatibility, imprinting makes it permanent.

If `data::production` grounded into `lib::analytics` is valid grounding, then `imprint!("lib::analytics", "data::production")(*)` is valid imprinting. The grounding operation proves that the derived namespace can bind against the data namespace. Imprinting persists that binding.

4.14. IN-LINE DDL

A special notation allows DDL to be inlined in two different contexts:

- within a query context (most often at the REPL)
- within a file to establish a sub-namespace

The former use-case allows for easy additions of entities that become immediately auto-enlisted into the evaluation context:

```
dql> (~~ddl
outside(x) :- x < 10
outside(x) :- x > 15 ~~)  _(x @ -1; 5; 11;253), +outside(x)
  ->
┌─────┐
│  x  │
├─────┤
│ -1  │
│ 5   │
```

```
┌─────┐
│ 253 │
└─────┘
dql> iota(20)(*), +outside(value)
  ->
┌─────────┐
│ value   │
├─────────┤
│ 1       │
│ 2       │
│ 3       │
│ 4       │
│ 5       │
│ 6       │
│ 7       │
│ 8       │
│ 9       │
│ 16      │
│ 17      │
│ 18      │
│ 19      │
│ 20      │
└─────────┘
```

The latter allows inline ddl within consulted files to create a sub-namespace:

delightql

```
// example.dql

// plus_two is defined in a file
// and what namespace it will belong to
// is a function of how consult! is called
plus_two:(x) :- x + 2

(~~ddl:"tmp"
  // times_three is in a sub-namespace
  // whose complete namespace will be known
  // once the file is consulted
  times_three:(x) :- x * 3

~~)
```

If the above file is consulted into `lib::ops`, then `times_three` as a function is namespaced as `lib::ops::tmp.times_three`.

5. DATA MANIPULATION LANGUAGE (DML)

Delightql supports SQL's tree mutation operations through three destructively sigilized pipe targets:

- `update!(T(*))(*)` – modify existing rows
- `insert!(T(*))(*)` – add new rows
- `delete!(T(*))(*)` – remove rows

The `T(*)` is the mutation target – a functor expression identifying which relation to mutate.

5.1. THE !! MARKER

For `update!` and `delete!`, the source relation is also the mutation target – the rows being sourced are the rows being mutated. Mark the source with `!!` to make this explicit:

delightql

```
employee!!(*)                          // !! = "these rows will be mutated"
  , title = "IT Staff"
  |> delete!(employee(*))(*)
```

The `!!` marker is required when the source is the mutation target. The compiler verifies that the `!!`-marked relation matches the terminal target.

For `insert!`, the source rows are read-only input – even when the source table happens to be the same as the target. Do not use `!!` on insert sources:

delightql

```
employee(*)   // no !! -- these rows are read-only
  , title = "IT Staff" |> (employee_id + 100 as employee_id,
                           last_name,
                           first_name,
                           title,
                           reports_to) |> insert!(employee(*))(*)
```

Terminal	Source has !!?	Reason
`update!`	Yes	Source rows are modified in place
`delete!`	Yes	Source rows are removed
`insert!`	No	Source rows are read-only input

5.2. INSERT

Use the `insert!` pseudo-predicate to insert rows. The relation entering the `insert!` pipe must contain a subset of the schema of the mutation target. Any extra or erroneously named columns are an error.

delightql
```
_(last_name, first_name, title @ "Eklund", "Daniel", "IT
Staff")
  |> insert!(employee(*))(*)
```

SQL
```
INSERT INTO employee (last_name, first_name, title)
VALUES ('Eklund', 'Daniel', 'IT Staff');
```

You may union tables and predicate their tuples to provide input tuples:

delightql
```
mount!("etl.sqlite", "etl")(*)
// a new target: an empty table shaped like the loads
etl.partner_sale_2025_06_30(*), #<0 |> table!(etl.partner_sale_all(*))
(*)

etl.partner_sale_2025_06_30(*)
  |;| etl.partner_sale_2025_07_31(, received_at >= "2025-07-01")
  |> insert!(etl.partner_sale_all(*))(*)
```

The target may already exist; the input names a subset of its columns:

delightql
```
mount!("etl.sqlite", "etl")(*)
etl.partner_sale_2025_07_31(*),
  quantity > 0 |> (sale_id, line_no, track_id, unit_price,
quantity)
  |> insert!(etl.partner_sale_line(*))(*)
```

SQL
```
INSERT INTO etl.partner_sale_line (sale_id, line_no, track_id,
unit_price, quantity)
SELECT sale_id, line_no, track_id, unit_price, quantity
FROM etl.partner_sale_2025_07_31
WHERE quantity > 0;
```

5.3. UPDATE

To update a table, pipe a matching schema into `update!` pseudo-predicate. The name of the table to be updated must be the higher-order parameter – this is the mutation target. [31] The mutation target must be the source of the data as well.

31 Note that this implies that the DML pseudo-predicate accepts two higher-order parameters: the contents of the query before the pipe and the table targeted. The trailing `(*)` is RECEIPT ACCESS – ordinary relation access over the directive's return table – never an input group.

Every DML directive returns a receipt relation of cardinality zero or one: one row (`success` first, then `operation`, then parameter echoes) answers YES – at least one row was affected; the empty relation answers NO. There is no `success = false` row, and failure aborts rather than encoding itself as data. DML receipts declare no `returned` payload, so unwrapping one (`!>` or `.returned(*)`) is a category error – the receipt continues through `|>`.

delightql

```
employee!!(*)
  , title = "IT Staff"
  |> $$("-------" as phone)
  |> update!(employee(*))(*)
```

SQL

```
UPDATE employee
SET phone = '-------'
WHERE title = 'IT Staff';
```

5.4. DELETE

To delete from a table, use predication to select the rows that should be removed. The mutation target must also be the source table and the schemas must match.

delightql

```
employee!!(*)
  , title = "IT Staff"
  |> delete!(employee(*))(*)
```

SQL

```
DELETE FROM employee
WHERE title = 'IT Staff';
```

Without filters, all rows are deleted:

delightql

```
invoice_line!!(*) |> delete!(invoice_line(*))(*)
```

SQL

```
DELETE FROM invoice_line;
```

To keep only some rows, invert the predicate and delete the complement:

delightql

```
invoice_line!!(*)
  , unit_price != 0.99
  |> delete!(invoice_line(*))(*)
```

6. EFFECTS AND DIRECTIVES

A directive is a functor whose name ends in an `!` exclamation point. It an instruction to the system to change something: rows in user tables, database objects, the session itself, or any mutative script, i.e. it is called for its effect. A directive is still a predicate. It is a relation whose evaluation returns a table called a receipt.

success	operation	...	...
1	mount!	...	...

Examples of calling directives:

delightql

```
mount!("etl.sqlite", "etl")(*)

enlist!("etl")(*)

_(last_name, first_name @ "Eklund", "Daniel") |> insert!
(employee(*))(*)
```

Directives may be built-in or user authored. In either case, the following are all true:

- all directives always end with a `!`
- all directives always return a single row or no row at all

6.1. RECEIPTS

A receipt is a single row table that is the result of calling a directive. A receipt also includes the concept of not being returned, so the more accurate statement is that a receipt is a zero or one row table that is the result of calling a directive.

The presence of a single row is meant to be understood as a successful judgement of a proposition – a “YES”, that the directive was successfull. The lack of a row, a zero cardinality table, indicates “NO”.

This cardinality and semantics allows receipts to be used in composed queries, where the NO short-circuits the rest of the query, simulating an early return. The presence of a YES row merely adds columns to the relation but never explodes the cardinality of the query in which it participates.

6.2. RECEIPT SCHEMA

Every receipt GUARANTEES exactly two things:

- the first column is `success`, value `1`;
- the second column is `operation`, the name of the directive that produced the receipt, as written (`'temp_table!'`, `'insert!'`),

Every remaining column is a declared addition: a typed column recorded in the directive's contract. A creation receipt (`table!`, `temp_table!`, `temp_view!`) declares `target`, the fully qualified data namespace target the designator selected, and `created`, the exact catalog path of the object created; they coincide for `table!`:

success	operation	target	created
1	temp_table!	main.staged	sys::shadow::main.staged

success	operation	...	...
1	mount!	...	...

Some receipts from certain directives embed a table within a table (a cheat that allows richer data return, but with the same zero-or-one cardinality that receipts demand).

Those receipts that embed a table within a table assigned to the name `returned` operate with the TEE-PIPE `!>` a syntactic sugar for exploding out the table within and removing the receipts columns.

```delightql
employee(*), title = "IT Staff" !> assert!(count_is(2), "two IT
staff")(*)
```

is exactly equivalent to the interior-drill form:

```delightql
employee(*), title = "IT Staff" |> assert!(count_is(2), "two IT
staff")(*) |> .returned(*)
```

6.3. EFFECT RULES

An effect rule is a rule where the functor name has a `!` in the head. It can be either lower or higher-order.

Effect rules allow the programmer to genereate their own reusable effects.

delightql
```
?- mount!("etl.sqlite", "etl")(*)

quarantine!(Bad(*))(*) :-
    Bad(*) |> insert!(etl.partner_sale_quarantine(*))(*)

stage!(*) :-
    etl.partner_sale_2025_07_31(*) |> temp_table!(staged(*))(*)

load!(Good(*))(*) :-
    Good(*), +customer(, email = buyer_email), quantity > 0
      |> insert!(etl.partner_sale_line(*))(*)
```

An effect rule must

1. Have an `!` exclamation point as the last character in the rule head name
2. End with a directive call (either built-in or a user authored effect rule)

This second requirement guarantees that every directive maintain the contractual semantic of returning a receipt.

6.4. THE UNWRAP PIPE

The UNWRAP-PIPE `!>` is a special pipe syntax used often with certain built-in directives.

delightql
```
employee(*), title = "IT Staff" !> assert!(count_is(2), "two IT
staff")(*)
```

The above notation is sugar for

delightql
```
employee(*), title = "IT Staff" |> assert!(count_is(2), "two IT
staff")(*) |> .returned(*)
```

For certain primitive built-in directives this notation permits a complete relation to be returned from a directive efficiently.

6.5. ETL AND SCRIPTING

Scripting is the act of utilizing effect rules – built-in or authored – to affect some change to a system.

Delightql provides the following semantic guarantees that makes scripting principled:

- all directive invocations return a receipt – zero or one rules
- all authored effect rules must end in another directive
- the COMMA `,` short-circuits effects after any other effect that returns a zero row receipt

- certain built-in directives – run! chief among them – have a well-known protocol

delightql

```
?- mount!("etl.sqlite", "etl")(*)
?- enlist!("main")(*)      // track, customer, and staged are read bare

landed(*) :-
    etl.partner_sale_2025_06_30(*)
      |;| etl.partner_sale_2025_07_31(*)
      |> %(*)

quarantine!(Bad(*))(*) :-
    Bad(*) |> insert!(etl.partner_sale_quarantine(*))(*)

stage!(*) :-
    landed(*)
      |> $$(lower:(trim:(buyer_email)) as buyer_email)
      |> temp_table!(staged(*))(*)

load!(Good(*))(*) :-
    Good(*), +track(*.(track_id)), +customer(, email =
buyer_email), quantity > 0
      |> insert!(etl.partner_sale_line(*))(*)

main!(*) :-
    stage!(*) : s!
    staged(*), (\+track(*.(track_id)) or \+customer(, email =
buyer_email))
      |> quarantine!(*) : q!
    staged(*) |> load!(*) : l!

    s!(*) ; q!(*) ; l!(*)
```

6.6. BUILT-IN DIRECTIVES

The built-in directives fall into five categories, by what effects they produce:

category	directives	direct
session directives	`consult!`, `consult_tree!`, `reconsult!`, `unconsult!`, `mount!`, `mount_new!`, `mount_tree!`, `unmount!`, `refresh!`, `ground!`, `enlist!`, `delist!`, `alias!`, `expose!`, `doc!`	the session's namespace tree
DDL directives	`temp_table!`, `table!`, `temp_view!`, `imprint!`, `imprint_replace!`	database objects
DML directives	`insert!`, `update!`, `delete!`	rows in user tables
execution directives	`run!`, `run_namespace!`	starting runs

category	directives	direct
utility directives	`exit!`, `returning!`, `returning_other!`, `stdout!`	the run itself: stop, return, sequence, print

6.6.1. The assert! directive

The assert! directive takes three arguments:

- a HO-judgment rule
- a human readable label
- the input relation

and runs the input relation through the judgment rule. If the judgement returns any rows then the assertion has passed. If the judgement fails to return rows, then assert! aborts the script or one REPL invocation and prints the human-readable label.

Example Passing

delightql

```
_(a@10;22;42)
  !> assert!(
       same_bag(
       _(a@10;22;42)),
       "Anon tables should match")(*)
  |> ( a*2 as post_assert)
```

post_assert
20
44
84

Example Failing

delightql

```
_(a@10;22;42)
  !> assert!(
       same_bag(
       _(a@ 99999999 )),
       "Anon tables should match")(*)
  |> ( a*2 as post_assert)
```

Error: [delightql-error://authored/abort] Permission: Anon tables should match

6.6.1.1. Judgment Rules

A judgment rule is any higher-order rule that takes a table parameter as its last parameter. Structurally, this is denoted in higher-order parameter syntax as `P(... T(*))(*)`.

delightql

```
at_least(n,T(*))(*) :-
    T(*) ~> count:(*) as count,
    count >= $.n

is_greater(n, T(*))(*) :-
    T(*) ~> count:(*) as count,
    count > $.n

not_empty(T(*))(*) :-
    T(*) ~> count:(*) as count,
    count > 0

greater_than(n, T(*))(*) :-
    T(*) ~> count:(*) as count,
    count > $.n

count_between(lo, hi, T(*))(*) :-
    T(*) ~> count:(*) as count,
    count >= $.lo,
    count <= $.hi
```

A judgment rule should encode some simple logic about when a condition is met or not and return zero or one rows: zero to cause the system to abort, and one to allow the chain to continue.

6.6.1.2. Denotational Assert! Definition

The assert rule is assumed to obey the following denotation:

delightql

```
assert!(P(... T(*))(*), label, I(*))(*) :-
    I(*) |> P(*) : witness
    witness(*) + : judgment
    judgment(*), met=1 |> returning_other!(I(*))(*) : pass!
    judgment(*), met!=1, abort!($.label) : fail!

    fail!(*); pass!(*)
```

Roughly this shows three higher-order parameters:

- the judgment rule (`P(...T(*))(*)`) that structurally still requires one last table-value parameter
- the `label` to be printed on failure and
- the input relation `I(*)`

The invocation `I(*) |> P(*)` is the heart of the asserts.

6.6.2. Consult

6.6.2.1. consult!

`consult!(path, ns)` reads the definitions file at `path` into a new namespace `ns` and executes the file's liminal space as it loads. The definitions are addressable by their qualified names; `enlist!` makes them available without qualification.

Consultation and name visibility are distinct from grounding: a groundable library's free table references are bound by `ground!` (see Grounding). The rule below instead names its data dependency explicitly as `main.track`.

A relative path resolves against the session's base directory, not the directory of the file that names it.

delightql

```
// mylib.dql
long_tracks(*) :- main.track(*), milliseconds > 600000
```

delightql

```
consult!("mylib.dql", "mylib")(*)
enlist!("mylib")(*)
long_tracks(*) |> (name, milliseconds)
```

One consulted source owns one namespace. Consulting into a name that already holds a consulted namespace refuses rather than merging; reload it with `reconsult!` or remove it with `unconsult!`.

6.6.2.2. reconsult!

`reconsult!(ns)` reads the namespace's file again and replaces the namespace whole: the next statement sees the new definitions, while a statement already running keeps the ones it began with. `reconsult!(ns, path)` reads a different file into the same namespace.

delightql

```
consult!("mylib.dql", "mylib")(*)
reconsult!("mylib")(*)
```

6.6.2.3. unconsult!

`unconsult!(ns)` removes a consulted namespace. It refuses, naming the blockers, while a definition outside the namespace still depends on it or another namespace stands beneath it.

delightql

```
consult!("mylib.dql", "mylib")(*)
unconsult!("mylib")(*)
```

6.6.2.4. consult_tree!

`consult_tree!(path, ns)` consults every `.dql` file under the directory `path`, one namespace per file beneath `ns`: `consult_tree!("lib", "lib")(*)` makes `lib/top.dql` the namespace `lib::top` and `lib/sub/speed.dql` the namespace `lib::sub::speed`. The receipt's `returned` column lists each file with the namespace it became.

6.6.3. Mount

Mounting is the process of importing knowledge of tables and views into a delightql namespace. No table, view, function or higher-order table can be accessed without it having been mounted into a namespace.

During the mounting process, delightql will scan the target connection for a list of all entities at that location and store them into internal delightql system tables. This effectively creates a delightql-side information schema that becomes the system of record

6.6.3.1. mount!

delightql

```
mount!("etl.sqlite", "data::prod")(*)
```

success	operation	path	namespace
1	mount!	etl.sqlite	data::prod

The directive `mount!` is a higher-order directive. It takes a pair of higher-order parameters both of which are strings:

- the target specific connection string
- the namespace into which this data will become visibile

The namespace must not exist prior to the mount call.

6.6.3.2. unmount!

delightql

```
mount!("etl.sqlite", "data::prod")(*)
unmount!("data::prod")(*)
```

success	operation	namespace
1	unmount!	data::prod

6.6.3.3. refresh!

The `refresh!` directive asks delightql to re-connect to the connection associated with the data namespace and to discover new or

changed entities. Use this when other programs may modify the database entities underneath you.

delightql

```
mount!("etl.sqlite", "data::prod")(*)
refresh!("data::prod")(*)
```

success	operation	namespace
1	refresh!	data::prod

6.6.3.4. mount_new!

Use **`mount_new!`** when the connection refers to a non-existent database. Delightql will create an empty namespace connected to the target's own empty namespace.

delightql

```
mount_new!("test.db", "data::test")(*)
```

success	operation	path	namespace
1	mount_new!	test.db	data::test

6.6.3.5. mount_tree!

Use **`mount_tree!`** when the connection has an internal hierarchy that can be turned into sub-namespaces on the delightql side.

7. SETTLED, KNOWN LIMITATION, UNSPECIFIED

- Rulings: semantics undertsood and supported
- Known Limitations: semantics understood but there is too much current cost to implement
- Unspecified: semantics not understood

This reference is an attempt to enumerate all semantics and the syntax that manifests those semantics. These we may call rulings.

The enumeration of these rulings may be incomplete. A language that is designed of composable units may have compositions that create unsettled semantics.

To better specify delightql, the enumeration of known limitations and/or unspecified behavior creates both an acknowledgment of boundaries and an inventory for future work.

This section attempts the enumeration of the negative space where delightql has knowingly refused to support something or has thrown up its hands and said “this is too hard for the moment.”

7.1. UNSPECIFIED BEHAVIOUR

Categorizing functionality or code as “unspecified behavior” leaves a clear warning to the programmer.

The contents of this category are places where delightql or SQL or both have no clear idea as to what the proper semantics should be.

This does not preclude a latter enlightment or clarification.

7.2. LANGUAGE LIMITATIONS

Delightql has limitations on its expressive power and its expressive guarantees.

The source of these limitations can be any of the following:

- a missing feature in SQL itself
- a feature that is inconsistently implemented across SQL targets
- a feature from logic that has no expressive abstraction at all in SQL
- a feature that is possible in SQL but requires an admixture of DDL and DML and querying
- any feature that does not yet have a satisfactory semantics

The last point expresses the fact that delightql may choose to not support a feature at the current time while leaving the door open for

its support upon future semantic clarity. This is a statement that can apply to all SQL targets or to selective SQL targets.

The current section aims to enumerate all known limitations.

7.2.1. Higher-Order Moding Limitations

Because there is no clear standard in SQL with regards to higher-order tables, delightql has two paths whenever a higher-order abstraction is used or invoked:

- it can lower the abstraction directly to the SQL target via a default transpilation rules
- it can rewrite with a detail intended semantics into a composition of normal relational values

The former states that if delightql sees:

```delightql
json_tree("[1, 2]")(*)
```

it will lower to:

```SQL
select * from json_tree('[1, 2]');
```

which is useful for well-known table-value functions like `json_each`.

The latter is exclusively the concern of the transpiler when the higher-order rule is programmer authored.

7.2.1.1. Programmer authored

7.2.2. Non-deterministic functions

> Delightql makes NO guarantee about impure function-level referential transparency or delimited referential transparency. The rest of this section details the challenges that will need to be overcome to permit any future guarantees.
>
> True referential transparency is a strong guarantee: that a function with the exact same arguments may be substituted with the one value it produces in all situations.
>
> Delimited referential transparency is a looser guarantee: that within one calling context, a function form with the exact same arguments may be substituted with the one value it produces.

Non-deterministic functions like certain vendors' `random()` may not have clearly defined semantics around delimited referential transparency.

To motivate, in SQLite SQL, a random value across many rows:

SQL

```
with
  _x as ( VALUES (1), (2), (3))
select random() as r from _x;
```

r
7906891248351165381
-8883546532970986158
-8728216203524003274

feels correct, as does when calling random twice in the same expression:

SQL

```
with
  _x as ( VALUES (1), (2), (3))
select random() as r1, random() as r2 from _x;
```

r1	r2
-3037396489422509295	4469562710814311768
-4848443231643320494	-713724063462585786
4226627867170606774	2825360043875777743

When we desire our expected flavor of delimited referential transparency, the random function "materialized" once seems to work like we think it should:

SQL

```
with
  _random as (
    select random() as r
      union all
    select random() as r)
select
    r,
    r+1 as r_plus_one
  from _random;
```

SQL

```
select
    r,
    r+1 as r_plus_one
  from
  (
    select random() as r
      union all
    select random() as r)
```

r	r_plus_one
-7433712953842994257	-7433712953842994256
-8813118922048968177	-8813118922048968176

but when SQLite's rules about flattening mark the expression differently, we may not get what we think we are going to get:

SQL

```
WITH input(n) AS (
  VALUES (1), (2), (3)
)
SELECT r AS first_read, r AS second_read
FROM (
  SELECT random() AS r
  FROM input
);
```

first_read	second_read
1747592606472746190	-5537291275306372640
-8262209585999104522	6666755709977980449
6154489567650109788	-8039489511224155566

SQLite does, however, adhere to a certain form of referential transparency for those non-deterministic functions that are part of the SQL standard: `CURRENT_TIME`,`CURRENT_DATE`,`CURRENT_TIMESTAMP`.

The difference in this behaviour can be understood by the taxonomic Postgres terminology:

- IMMUTABLE: a function is guaranteed to be referentially transparent in all situations
- STABLE: a function is guaranteed to be referentially transparent within one context
- VOLATILE: a function is not guaranteed to be referentially transparent

> NOTE: the term "context" may mean different things between different SQL targets: for some, at the unit of a query, for others at the unit of a transaction.

7.2.2.1. Delightql's challenge

Delightql has several challenges in this corner of the SQL world:

- is there a semantics that can be guaranteed equivalent across SQL targets?
- can the standard semantic of "one delightql query expression = one SQL query expression" be maintained?
- is the IMMUTABLE/STABLE/VOLATILE taxonomy a suitable standard for coloring SQL functions?
- can rewrite equivalences match any expected semantics?

7.2.2.2. Cross-target

The SQLite example above shows that its implementations of the functions `random()` and `current_timestamp()` are VOLATILE and STABLE respectively. That the function `current_timestamp` is part of the SQL standard implies that SQL vendors should maintain STABLE semantics for these.

A matrix of implementation strategies across vendors shows that even the STABLE semantic comes with some vendor-specific caveats:

SQL-standard family	PostgreSQL	DuckDB	MySQL	SQL Server
`CURRENT_DATE`	Yes; transaction start	Yes; transaction start	Yes; query start	SQL Server 2025+/ Azure
`CURRENT_TIME`	Yes; transaction start	Yes; transaction start	Yes; query start	No ordinary T-SQL spelling
`CURRENT_TIMESTAMP`	Yes; transaction start	Yes; transaction start	Yes; query start	Yes; classified nondeterministic
`LOCALTIME`	Yes; transaction start	Yes; transaction start	Accepted as a `NOW()` synonym, not the standard `TIME` distinction	No
`LOCALTIMESTAMP`	Yes; transaction start	Yes; transaction start	Accepted as a `NOW()` synonym	No
`NEXT VALUE FOR`	No; uses `nextval()`	No; uses `nextval()`	No sequences	Yes
`PREVIOUS VALUE FOR`	No; uses `currval()`	No; uses `currval()`	No	No
Routine volatility declaration	`IMMUTABLE` / `STABLE` / `VOLATILE`	UDF `side_effects` metadata	`DETERMINISTIC` / `NOT DETERMINISTIC`, author-asserted	Engine-derived determinism and schema binding

The challenge therefore may be answered with vendor-specific transpilation using correct transactional boundaries.

7.2.2.3. Query-only Semantics and Materialization

The primary semantic that delightql guarantees is that all pure queries are equivalent to the SQL string that they produce. That delightql does not have a direct mapping between query string to query string when directives are utilized is not a contradiction: directives are impure and are marked as such (`!`).

Therefore if delightql wishes to support support referential transparency for functions syntactically marked as pure:

delightql

```
_(a@1;2;3) as anon
    |> +(random:() as r)
    |> (r, r+1)
```

it cannot result in an admixture of DDL and SQL:

SQL

```
BEGIN;

CREATE TEMPORARY TABLE __dql_stage AS
WITH input(a) AS (
  VALUES (1), (2), (3), (3)
)
SELECT
  input.a,
  random() AS r
FROM input;

SELECT
  a,
  r,
  r + 1
FROM __dql_stage;

DROP TABLE __dql_stage;

COMMIT;
```

If delightql eventually does allow for these impure functions, it will require marking them as impure:

delightql

```
_(a@1;2;3) as anon
    |> +(random!:() as r)
    |> (r, r+1)
```

which will

- give a syntactic indication to the programmer that both the primary semantic is void and that a new (to-be-determined) referential semantic is now in force
- instruct the delightql compiler to create an effect plan of DDL and transactions

This remains a challenge under the current epistemic regime whereby unknown functions are default transpiled into the target SQL:

delightql

```
employee(*) |> +( do_it:(last_name) as di)
```

SQL

```
select *, do_it(last_name) as di from employee;
```

7.2.2.4. The IMMUTABLE/STABLE/VOLATILE

7.2.2.5. Relational Equational Rewriting

7.2.3. Recursion Limitations

7.2.4. Autogenerated Names

7.2.5. JSON-awareness and cataloging

```delightql
partner_sale(*)
  |> (sale_id, payload:{.buyer} as buyer)
  |> %(sale_id ~> max:(buyer) as buyer)
  |> (sale_id, {"buyer": buyer} as packet)
```

Today, without the max step, the buyer re-enters the packet as an object (sale 1):

```
{"buyer":{"email":"luisg@embraer.com.br","country":"BR"}}
```

With the max step it re-enters as a string:

```
{"buyer":"{\"email\":\"luisg@embraer.com.br\",\"country\":
\"BR\"}"}
```

This is a known limitation.

7.2.6. Lateral Join Limitations

An inner-relation can only reference an outer relation prior to a scope barrier.

8. THE DELIGHTQL RUNTIME

The delightql runtime is a set of stateful data and contracts that anyone running a query or script may depend upon.

The delightql runtime guarantees

- the existence of certain namespaces
- special namespaces for CLI, REPL and library usages
- special rules for namespace removal and addition
- special namespaces deemed auto-enlisted
- the existence of certain entities: rules, functions, sigma predicates, etc.
- the existence of certain directives

The runtime namespacing mechanism is introspectable as tables by delightql itself. It also doubles as a code module system and data import mechanism.

This blend of introspection and user-driven namespacing has a precedent in images, a style of always-on programming promulgated by Smalltalk.

8.1. NAMESPACE CONTRACTS

8.1.1. Built-in Namespaces

Delightql provides the following namespaces:

- `main` – where data is mounted when a database is provided at start of process
- `home` – where the REPL's definitions (`.ddl` input, inline ddl) are written
- `std` – no entities, just a container for other namespaces
- `std::prelude` – auto-enlisted; the directives and core pseudo-predicates
- `std::string` – string functions
- `std::constants` – named constants such as `nl` and `tab`
- `std::predicates` – for `like` and `between`
- `std::help` – `operators(*)`, the operator sigils as a relation
- `std::info` – `tables(*)`, the tables visible in the session
- `std::json`, `std::math`, `std::reshape` – JSON, aggregate and reshaping (pivot) helpers

`std::(*)` lists the `std` namespaces.

> While introspectable system tables may be seen and used (located at all namespaces with a `sys` prefix) they are not to be depended upon.

This is a hard choice to allow the slow adoption of guranteed introspection and metadata predicates into `std` while their lower-level and version-based siblings at `sys` may be churned as dictated by functionality and refactoring.

Delightql is already exceptionally well introspectable, but has not yet decided its public facing contract.

To repeat: all namespaces and entities prefixed by **`sys`** do not guarantee stability to a programmer over version changes.

8.1.2. Namespace metadata

8.1.2.1. Namespace catalog functor

Every namespace may be called directly:

```
namespace::subnamespace::(*)
```

8.1.3. Auto-enlist namespaces

8.1.4. Special Protected Namespaces

A programmer has wide latitude to create their own tree of namespaces anywhere in the image inclusive of entities within those namespaces.

Delightql sets aside the following namespaces as protected classes:

- `std`-prefixed namespaces may not receive new entities nor add or drop namespaces
- `sys`-prefixed namespaces may not receive new entities nor add or drop namespaces

8.2. REPL VS CLI VS CORE

8.3. SYSTEM TABLES – A TOUR

All namespaces and entities prefixed by **`sys`** do not guarantee stability to a programmer over version changes.

Nevertheless, the current version of delightql has many candidates for promotion to `std` that there might come a time when

8.3.1. Information Schema Namespaces

The following namespaces contain entities that allow introspection of what entities are knowable at runtime. These namespaces contain multiple table or view entities that are created, modified or deleted whenever a programmer runs `consult!`, `mount!`, etc.

- `sys::ns`
- `sys::cartridges`
- `sys::entities`
- `sys::entities::ho`
- `sys::entities::interior`

8.3.2. Transpilation Rules Tables

8.4. OVERRIDE TABLES

9. ANNOTATIONS

Annotations appear at continuation points in a query pipeline, and permit language directives to operate as inline syntactic elements.

9.1. THE ANNOTATION FRAMEWORK

Annotations are distinguished by matching PARENOTATES `(~~ ~~)` followed immediately (no space) by an `<identifier>`.

```
(~~<identifier> body ~~)              // annotation with body
(~~<identifier>:instance body ~~)     // annotation with instance
name
```

The identifier after the `(~~` is the ANNOTATION-TYPE.

How the body is parsed within an annotation is a function of the specific ANNOTATION-TYPE. Each recognized type has its own grammar rule:

- `(~~error`: no body – just an optional URI which matches with well-known errors
- `(~~danger`: a URI naming the acknowledged danger

`(~~docs` shares the delimiters but is not an annotation. It is the dedicated definition-document form, and its body is raw text.

A colon form `identifier:instance` names a specific annotation instance and is also a function of the particular annotation type in question.

delightql

```
invoice(*), total > 0
  (~~config://generation/rule/inlining/view ~~)
```

Only the annotation types listed above are recognized by the grammar. Unknown annotation names produce a parse error.

9.2. PLACEMENT

Annotations may appear at any continuation point – before a pipe, before a comma, or at the end of an expression:

delightql

```
invoice(*), total > 20
  (~~config://generation/rule/inlining/view ~~)
  |> (invoice_date, billing_city)
```

Annotations are usually transparent to SQL generation and so the pipeline above produces identical SQL to:

```delightql
invoice(*), total > 20 |> (invoice_date, billing_city)
```

Multiple annotations at the same point are permitted. They appear as siblings in the CST and are processed in order.

9.3. ANNOTATION TYPES

- `error` – expects compilation to fail, matches the error URI (see Error Assertions)
- `danger` – acknowledges a named safety gate for the current query (see Danger Gates) (see Options)

The dedicated `docs` form attaches documentation to a definition; see Docs.

9.4. ERROR ASSERTIONS

Error assertions verify that a query fails compilation or resolution with a specific error. They use a separate annotation with the reserved name `error`:

`(~~error://uri/path ~~)`

The URI identifies the expected error category using a hierarchical path. The pipeline attempts to compile the query; if compilation fails and the actual error matches the URI, the assertion passes: the statement returns no rows. A statement that compiles anyway is refused (`runtime/expectation`).

```delightql
// should fail: table does not exist
nonexistent_table(*) (~~error://semantic/resolution/table ~~)

// should fail: column not in scope
customer(*) |> (no_such_column) (~~error://semantic/resolution/column
~~)

// should fail: any semantic error (prefix match)
invoice(*), total in (1,2,3) (~~error://semantic ~~)
```

A bare error annotation with no URI matches any error:

```delightql
// should fail with some error, don't care which
bad_query(*) (~~error ~~)
```

Errors are rarely needed for end users.

9.4.1. URI Prefix Matching

The URI is matched as a prefix against the actual error's canonical URI. `error://semantic/resolution` matches `semantic/resolution/table`, `semantic/resolution/column`, and any future `semantic/resolution/*` error. `error://semantic/arity` matches only `semantic/arity` and its sub-paths.

9.4.2. Error URI Categories

Each `DelightQLError` variant maps to a canonical URI path. The URI is a stable identifier for the error category, reusable in documentation, tooling, and diagnostics.

URI	Meaning
`authored`	A termination the program itself demanded.
`client`	An incident of the interactive client itself.
`configuration`	A host's boot settings are invalid.
`dml`	A data-modification query violated DML shape rules.
`imprint`	A blueprint or manifest lifecycle refusal.
`internal`	A defect in DelightQL itself.
`namespace`	A namespace-creation policy refusal.
`operational`	This session refuses to run a valid query.
`parse`	The source text is structurally invalid.
`runtime`	An execution-time failure.
`semantic`	The query is well-formed but semantically invalid.
`target`	The foreign engine rejected or failed the query.

Table 55: Top-level error URI families (`dql explain delightql-error://<family>` lists each family's members)

9.4.3. Coexistence with Data Assertions

Error assertions and data assertions can appear in the same file, documenting both correct and incorrect forms:

delightql

```
// correct: an ordinary assertion effect checks the established relation
invoice(*), total > 0
  !> assert!(exists(*), "a positive-total row exists")(*)

// incorrect: commas produce a multi-column single-row relation
invoice(*), total in (1,2,3) (~~error://semantic/membership ~~)
```

9.4.4. Scope

Error assertions are primarily a language development tool. They assert contracts about the compiler's behavior. End users writing queries against a database should have no use for expected failures. [32]

32 I think

9.5. DANGER GATES

Certain behaviors are safe in most contexts but dangerous in others. Rather than forbid them outright, delightql gates them behind danger URIs: named safety boundaries that are closed by default and opened explicitly per-query.

9.5.1. Syntax

A danger gate is a `danger://` URI inside annotation delimiters:

```delightql
genre_2024(*) as x (~~danger://semantics/min_multiplicity~~)
  |;| genre_2025(*) as y,
  x.* = y.*
```

The annotation attaches at a continuation point (after a relation). The URI identifies the specific danger. The annotation takes the URI alone: writing it opens that gate for this query, and nothing else opens one: a state word such as `ON` has no place in it.

A state word inside the annotation is an error:

```delightql
// INVALID: the annotation takes no state word
genre_2024(*) as x (~~danger://semantics/min_multiplicity ON~~)
  |;| genre_2025(*) as y,
  x.* = y.*
```

9.5.2. Scoping

A danger gate opens for one query and auto-closes at query end. It does not leak into subsequent queries:

```delightql
// gate is open for this query: 23 rows, Jazz once
genre_2024(*) as x (~~danger://semantics/min_multiplicity~~)
  |;| genre_2025(*) as y,
  x.* = y.*

// gate is closed again: 47 rows, every matching row of both arms
genre_2024(*) as x |;| genre_2025(*) as y,
  x.* = y.*
```

A `min_multiplicity` gate must be spent: written on a statement where no correlated union uses it, it refuses (`semantic/setop/min_multiplicity/unspent`), and over a correlation that compares only some of the columns the arms share it refuses too (`semantic/setop/min_multiplicity/partial`) — correlate the whole row (`x.* = y.*`).

An unaliased self-join is not a danger to acknowledge: two live scopes sharing a name always refuse (`semantic/scope/duplicate`). Alias one side.

9.5.3. Session Baseline

The program starts with every danger OFF, and only an annotation in the source text opens one, for its query.

Per-query annotations override the session baseline. At query end, the danger reverts to the enclosing scope:

9.5.4. Danger URI Reference

The known dangers are:

URI	What it gates
`delightql-danger://semantics/min_multiplicity`	A correlated union pairs duplicate copies by minimum multiplicity (SQL's `INTERSECT ALL`) instead of keeping every matching row of both arms; inline only

9.6. DOCS

Definitions may carry structured documentation between the neck and the body. The docs block uses the annotation delimiters with the `docs` identifier:

```
<HEAD>  <NECK>  (~~docs ... ~~)  <BODY>
```

9.6.1. Syntax

delightql

```
long_tracks(*) :-
  (~~docs
    Tracks longer than ten minutes.

    Returns:
      columns: inherited from track
      cardinality: variable
  ~~)
  track(*), milliseconds > 600000
```

The docs block is a `(~~docs ... ~~)` annotation. The body is raw text – no DQL parsing is applied. Line breaks, indentation, and blank lines are preserved as written.

The block must appear immediately after the neck, before the first expression of the body. Only one docs block per definition is permitted.

9.6.2. Applicability

The docs block is valid on any rule-form definition:

Views:

```delightql
usa_customers(*) :-
  (~~docs
    Customers whose country is USA.
  ~~)
  customer(*), country = "USA"
```

Functions:

```delightql
tax_amount:(price, rate) :-
  (~~docs
    Computes tax as price times rate, rounded to two decimal
places.

    Returns:
      type: numeric
  ~~)
  round:(price * rate, 2)
```

Higher-order rules:

```delightql
same_schema(T(*), V(*))(*) :-
  (~~docs
    Compares T's and V's schema for
    equality. Equality is reached if the
    column names match exactly and are
    in the same ordinal position.

    Returns:
      column: pass
      column type: boolean
      cardinality: 1
  ~~)
  first_md(*)  : T(^)
  second_md(*) : V(^)
  together(*)  :
    second_md(*),
      first_md(*.(column_name, ordinal))
  together(~> count:() as a),
    first_md(~> count:() as b),
    second_md(~> count:() as c)
      |> ( (a = b) and (b = c) as pass)
```

Sigma predicates:

delightql

```
is_recent(invoice_date, threshold) :-
  (~~docs
    Filters to rows where invoice_date is
    within threshold days of today.
  ~~)
  invoice_date > date:("now", ("-" ++ threshold) ++ " days")
```

The docs block is not valid on facts (which have no neck) or on shadow-neck definitions (which are query-scoped and ephemeral).

9.6.3. Storage

When a definition is loaded via `consult!`, the docs text is extracted at parse time and stored alongside the entity in the system catalog.

The docs are queryable through the system metadata:

delightql

```
sys::entities.entity(*)
  |> ( name, doc )
```

10. APPENDIX: TREE NORMAL FORM

Tree grouping bridges two worlds: the relational (flat, tabular, join-oriented) and the hierarchical (nested, tree-structured, document-oriented). Not all JSON has a sensible relational interpretation – arbitrary nesting can be too irregular to map cleanly. But some trees do, and understanding which ones helps clarify what tree grouping actually computes.

This appendix defines tree normal forms – a vocabulary for describing JSON structure and its relational interpretation. Unlike database normal forms (which form a linear hierarchy), tree normal forms are organized as a graph. Nodes are forms; edges are constraints or interpretations. Some edges restrict structure; others assign meaning to nesting.

10.0.1. The Graph

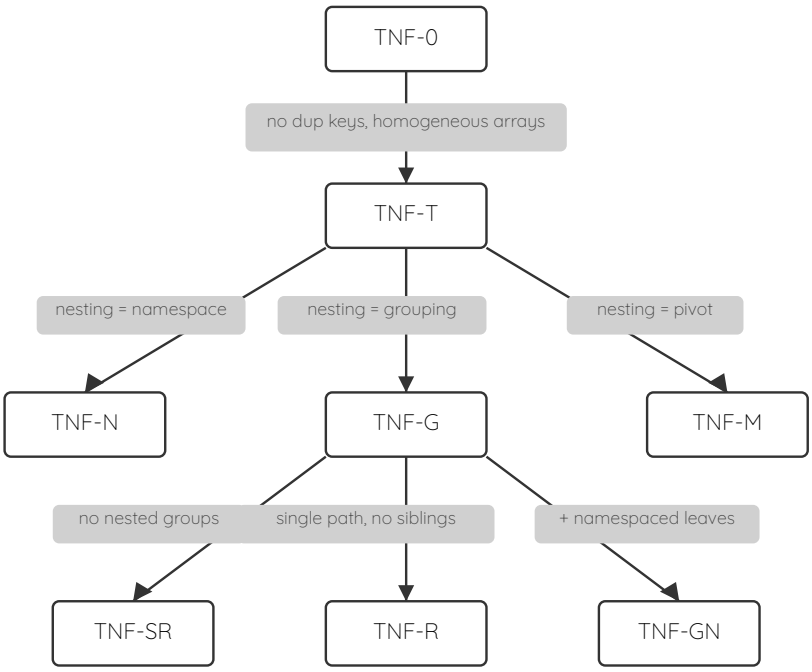


Figure 8: Mind Map

Edges represent:

Edge	Type	Meaning
TNF-0 ⟶ TNF-T	Restriction	Structural hygiene
TNF-T ⟶ TNF-N	Interpretation	Nesting means namespacing

Edge	Type	Meaning
TNF-T ⟶ TNF-G	Interpretation	Nesting means grouping
TNF-T ⟶ TNF-M	Interpretation	Nesting means pivot
TNF-G ⟶ TNF-SR	Restriction	No nested groups (flat)
TNF-G ⟶ TNF-R	Restriction	Single path, no siblings
TNF-G ⟶ TNF-GN	Combination	Grouping with namespaced leaves

Table 57: Tree normal form edge types

The graph admits extension. New forms slot in by defining their edges.

10.1. THE FORMS

10.1.1. TNF-0: Valid JSON

The baseline: any valid JSON per RFC 7159.

- Keys may be duplicated
- Arrays may be heterogeneous
- Top-level value may be a scalar
- Objects may be empty

```
42
```

```
[ 1, [], "a string", true, {},
  { "a": [2, 3], "a": [2, 3] },
  [ 2, 3, [ 4, 5 ] ]
]
```

Relational interpretation: None guaranteed. This is raw material. But it's still queryable – pathing and `json_each` work on any valid JSON.

10.1.2. TNF-T: Well-Typed JSON

Restriction from TNF-0: no duplicate keys, homogeneous arrays, non-empty objects, array or object at top level.

```
{
  "name": "Alice",
  "scores": [95, 87, 91]
}
```

Relational interpretation: Arrays can be interpreted as collections; objects as records. Pathing is unambiguous. But nesting semantics are not yet defined – is a nested object a namespace? A grouped row? A pivot?

TNF-T is the foundation for the interpretive forms that follow.

10.1.3. TNF-N: Namespaced

Interpretation from TNF-T: nesting means semantic organization.

Nested objects group related fields – structure, not data rows.

```
{
  "last_name": "eklund",
  "address": {
    "city": "boston",
    "state": "MA"
  }
}
```

The `address` object is a namespace. The tree is semantically equivalent to:

```
{
  "last_name": "eklund",
  "address_City": "boston",
  "address_State": "MA"
}
```

Relational interpretation: Namespaced trees flatten to a single row. Pathing (`.address.city`) navigates the namespace.

Trade-off: More expressive (preserves semantic grouping) but less directly relational (requires flattening).

10.1.4. TNF-G: Grouped

Interpretation from TNF-T: nesting means aggregation.

Arrays represent grouped rows – the result of `GROUP BY`.

```
[
  { "title": "Engineer",
    "people": [
      { "first_name": "Alice", "last_name": "Smith" },
      { "first_name": "Bob", "last_name": "Jones" }
    ]
  },
  { "title": "Manager",
    "people": [
      { "first_name": "Carol", "last_name": "White" }
    ]
  }
]
```

Each nesting level is a grouping context. The outer array groups by `title`; the inner `people` array collects rows within each title.

Relational interpretation: Direct correspondence to `GROUP BY`. Construction compresses cardinality; destructuring expands it.

10.1.5. TNF-M: Metadata-Keyed

Interpretation from TNF-T: nesting means pivot.

Data values become object keys.

```
{
  "Engineer": [
    { "first_name": "Alice", "last_name": "Smith" }
  ],
  "Manager": [
    { "first_name": "Carol", "last_name": "White" }
  ]
}
```

The keys (`Engineer`, `Manager`) are data values lifted to metadata.

Relational interpretation: Keys map to a column; values map to grouped rows. Destructuring recovers the key as a column value.

Trade-off: Convenient for lookup but constrained – only one column can serve as keys per level. Two metadata-keyed objects with the same key type create ambiguous destructuring.

10.1.6. TNF-SR: Simply Relational

Restriction from TNF-G: no nested groups.

A flat array of homogeneous objects – the simplest grouped form.

```
[
  { "title": "Engineer", "first_name": "Alice", "last_name":
"Smith" },
  { "title": "Engineer", "first_name": "Bob", "last_name":
"Jones" },
  { "title": "Manager", "first_name": "Carol", "last_name":
"White" }
]
```

No nested arrays. Each object is a row; the array is a table.

Relational interpretation: Direct. The JSON is a table in array-of-objects form. No grouping, no hierarchy – just rows.

10.1.7. TNF-R: Round-Trippable

Restriction from TNF-G: single path from root to deepest leaf, no sibling groups.

```
[
  { "title": "Engineer",
    "state": "CA",
    "people": [
      { "first_name": "Alice", "last_name": "Smith" }
    ]
  }
]
```

Relational interpretation: Lossless. `relation → tree → relation` recovers the original data (modulo column order).

Why siblings break round-tripping:

```delightql
customer(*) ~> { country,
                 "people": ~> {first_name, last_name},
                 "cities": ~> [city] }
```

Siblings aggregate independently. The join – which person was in which city – is not preserved. Destructuring recovers each path independently:

- `country`, `first_name`, `last_name` (via `people`)
- `country`, `city` (via `cities`)

But not the original four-column row. This is TNF-G but not TNF-R.

10.1.8. TNF-GN: Grouped with Namespaced Leaves

Combination of TNF-G and TNF-N: grouping structure with namespaced leaf objects.

```
[
  { "title": "Engineer",
    "people": [
      { "name": { "first": "Alice", "last": "Smith" },
        "contact": { "email": "alice@x.com", "phone":
"555-1234" }
      }
    ]
  }
]
```

The outer structure is grouped (array of objects with nested arrays). The leaf objects use namespacing (`name`, `contact`).

Relational interpretation: Destructure the grouping levels; flatten the namespaced leaves. The result has columns `title`, `name_first`, `name_last`, `contact_email`, `contact_phone`.

10.2. MIXING FORMS

Real trees often combine forms at different levels. The graph shows which combinations make sense:

```
{
  "metadata": {
    "generated": "2024-01-15",
    "version": "1.0"
  },
  "data": [
    { "title": "Engineer",
```

```
      "people": [
        { "first_name": "Alice", "last_name": "Smith" }
      ]
    }
  ]
}
```

- `metadata` is TNF-N (namespacing)
- `data` is TNF-G (grouping)

The relational interpretation:

- Flatten `metadata.generated`, `metadata.version` to columns
- Destructure `data` → `data.people` to rows
- Result: one row per person, metadata fields repeated

Understanding which form applies where clarifies what operations make sense.

10.3. EDGE TYPES

The graph has two kinds of edges:

Restriction edges add structural constraints: - TNF-0 → TNF-T: hygiene (no dup keys, homogeneous arrays) - TNF-G → TNF-SR: flatness (no nested groups) - TNF-G → TNF-R: single-path (no siblings)

Interpretation edges assign meaning to structure: - TNF-T → TNF-N: nesting is namespacing - TNF-T → TNF-G: nesting is grouping - TNF-T → TNF-M: nesting is pivot

Restriction edges constrain what trees are valid. Interpretation edges determine how to read them relationally.

10.4. SUMMARY

Form	Key Property	Relational Interpretation
TNF-0	Valid JSON	None guaranteed; queryable via pathing
TNF-T	Well-typed	Arrays are collections; objects are records
TNF-N	Namespaced	Flatten to single row
TNF-G	Grouped	GROUP BY; destructure to rows
TNF-M	Metadata-keyed	Pivot; keys become column values
TNF-SR	Simply relational	Direct table (array of flat objects)
TNF-R	Round-trippable	Lossless construction/destruction

Form	Key Property	Relational Interpretation
TNF-GN	Grouped + name-spaced	Destructure groups, flatten namespaces

Table 58: Summary of tree normal forms

The forms answer different questions:

- TNF-0 / TNF-T: Is this JSON structurally sound?
- TNF-N / TNF-G / TNF-M: What does nesting mean here?
- TNF-SR / TNF-R: How constrained is the grouping?
- TNF-GN: Can I mix interpretations?

Tree normal forms are not prescriptive – TNF-0 is sometimes exactly what you need. They are a vocabulary for understanding what your tree structure means relationally, and what operations it supports.

10.5. EXTENDING THE GRAPH

The graph admits new forms by defining edges. Examples:

- TNF-MR (metadata-keyed, round-trippable): TNF-M + single-path constraint
- TNF-SN (simply namespaced): TNF-N + flat (no nested namespaces)
- TNF-GM (grouped + metadata): grouping with metadata-keyed intermediate levels

Each new form names a useful combination.

10.5.1. Guiding Principles

1 Trees and tables mix. Trees have a valid relational interpretation under certain structural constraints.

2 Start with relations. The cleanest trees arise from grouping relations, not from arbitrary JSON. Construction informs understanding.

3 Arrays are rows; objects are records. Arrays represent homogeneous collections (multiple rows of the same shape). Objects represent heterogeneous structure (named fields, like columns).

4 Grouping compresses; destructuring expands. Construction decreases cardinality (many rows ⟶ fewer rows with nested arrays). Destructuring increases cardinality (nested arrays ⟶ many rows).

5 Siblings lose information. Sibling tree groups aggregate independently. The relationship between siblings – which person was in which city – is not preserved. This is inherent, not a bug.

6 Metadata-oriented trees are pivots. When data values become object keys, the structure resembles a pivot table. This helps with the object-relational impedance mismatch but introduces constraints.

7 Zeroth normal form has its place. Arbitrary JSON can still be queried via `json_each` and pathing. Tree normal forms define what's cleanly relational, not what's queryable at all.

10.5.1 BIBLIOGRAPHY

Darwen, Hugh. “HAVING A Blunderful Time.” Accessed December 26, 2024. https://www.dcs.warwick.ac.uk/~hugh/TTM/HAVING-A-Blunderful-Time.html.

10.5.1 FIGURES

Table 1 SQL and logic programming terminology equivalences 10
Table 2 Preservation properties of Project, ProjectOut, MapCover, and BasicMapCover 34
Table 3 Preservation properties of Rename, OrderBy, and GroupModulo 34
Table 4 Preservation properties of Embed and MapEmbed 34
Table 5 Infix domain predicates 41
Table 6 Arithmetic domain operators 55
Table 7 58
Table 8 Window frame bound syntax 60
Table 9 Compound data constructor behavior by position 64
Table 10 Scalar interior record result 64
Table 11 Aggregate interior record result – grouped by **`title`** 65
Table 12 Scalar interior tuple result 66
Table 13 Aggregate interior tuple result – grouped by **`title`** 66
Table 14 Column range syntax summary 69
Table 15 Index addressing schemes 70
Table 16 Position meanings for the reposition operator 71
Table 17 Compound data constructors by position (recap) 78
Table 18 {#tbl:array-tree-group} 80
Figure 1 group and destructure 83
Table 19 88
Table 20 89
Table 21 89
Figure 2 Union Compatability via Ordinal Alignment 90
Figure 3 Union Compatability via Name Alignment With OUTER 90
Table 22 Set operator alignment modes 91
Table 23 91
Figure 4 Intersect ON via correlation conditions 93
Table 24 Intersection as union with correlation 93
Table 25 Correlation as join versus correlation as intersect 94

Table 26 Sales per year and billing country (first two years) 102
Table 27 Pivoted result – countries become columns 103
Table 28 Pivot with multiple value columns 104
Table 29 Output of **`album(^)`** 105
Table 30 Table continuation operators 105
Table 31 Columns introduced by access pattern 107
Table 32 Summary of unification rules 111
Figure 5 Inward Operations Only 113
Figure 6 Inward Operations Only 114
Figure 7 Entity Shadowing 115
Table 33 The two neck operators 136
Table 34 143
Table 35 148
Table 36 Namespace classification by contents 169
Table 37 Conventional namespace path prefixes 170
Table 38 Core pseudo-predicates in lib::std::prelude 171
Table 39 System introspection namespaces 171
Table 40 177
Table 41 181
Table 42 182
Table 43 182
Table 44 184
Table 45 185
Table 46 188
Table 47 188
Table 48 189
Table 49 189
Table 50 193
Table 51 193
Table 52 193
Table 53 194
Table 54 195
Table 55 Top-level error URI families (**`dql explain delightql-error://<family>`** lists each family's members) .. 205
Table 56 207
Figure 8 Mind Map 211
Table 57 Tree normal form edge types 211
Table 58 Summary of tree normal forms 216