

Package ‘polyglotSQL’

October 8, 2026

Title SQL Parsing, Analysis and Dialect Translation

Version 0.1.2

Description Parse, tokenize, validate, format, analyze and translate SQL between more than 30 dialects ('PostgreSQL', 'MySQL', 'BigQuery', 'Snowflake', 'DuckDB', 'T-SQL', and others) using the 'polyglot-sql' Rust crate <<https://github.com/tobilg/polyglot>>, a Rust port of the 'SQLGlot' 'Python' library. All processing happens locally in the R session; no database connection, 'Python' runtime or external service is required. Includes column-level lineage, structural query analysis, query optimization, 'AST' diffing and 'OpenLineage' facet generation.

License MIT + file LICENSE

URL <https://github.com/StrategicProjects/polyglot-sql-r>,
<https://strategicprojects.github.io/polyglot-sql-r/>

BugReports <https://github.com/StrategicProjects/polyglot-sql-r/issues>

Depends R (>= 4.2)

Imports cli, jsonlite

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/polyglotSQL/upstream 0.12.0

Config/rextendr/version 0.5.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

RoxygenNote 8.0.0

SystemRequirements Cargo (Rust's package manager), rustc >= 1.88.0, xz

NeedsCompilation yes

Author André Leite [aut, cre] (ORCID: <<https://orcid.org/0000-0002-4718-9766>>),
Marcos Wasiliew [aut] (ORCID: <<https://orcid.org/0009-0004-4694-3159>>),
Hugo Vasconcelos [aut] (ORCID: <<https://orcid.org/0000-0001-6249-0920>>),
Carlos Amorim [aut] (ORCID: <<https://orcid.org/0000-0001-6315-8305>>),
Diogo Bezerra [aut] (ORCID: <<https://orcid.org/0000-0002-1216-8674>>),
Júlia Nascimento Barreto [aut] (ORCID:
<<https://orcid.org/0009-0004-2851-7770>>),
Tobias Müller [cph] (Author of the bundled 'polyglot-sql' Rust crate
(Polyglot project)),
Toby Mao [cph] (Author of SQLGlot, from which Polyglot is derived),
The authors of the vendored Rust dependencies [cph] (see
inst/COPYRIGHTS)

Maintainer André Leite <leite@castlab.org>

Repository CRAN

Date/Publication 2026-10-08 16:50:02 UTC

Contents

as_polyglot_schema	2
polyglot_version	3
sql_analyze	4
sql_annotate_types	5
sql_dialects	5
sql_diff	6
sql_format	7
sql_generate	7
sql_lineage	8
sql_openlineage	9
sql_optimize	10
sql_parse	10
sql_source_tables	11
sql_tokenize	12
sql_transpile	12
sql_validate	13
Index	15

as_polyglot_schema	<i>Specify a table schema for schema-aware operations</i>
--------------------	---

Description

Several polyglotSQL functions ([sql_validate\(\)](#), [sql_lineage\(\)](#), [sql_analyze\(\)](#), [sql_optimize\(\)](#), [sql_annotate_types\(\)](#), [sql_openlineage\(\)](#)) accept an optional schema argument describing the tables referenced by the query. A schema enables column qualification, type inference, and existence checks.

Usage

```
as_polyglot_schema(schema)
```

Arguments

schema A schema specification (named list as described above), or NULL for no schema.

Details

A schema is a named list with one entry per table. Each entry is either:

- a *named* character vector mapping column names to SQL types, e.g. `c(id = "INT", name = "TEXT")`;
- an *unnamed* character vector of column names (types unknown), e.g. `c("id", "name")`.

Value

A JSON string in the upstream ValidationSchema format, or "" when schema is NULL. Mostly used internally; exported for advanced users who want to inspect the generated payload.

Examples

```
as_polyglot_schema(list(
  orders = c(o_id = "INT", o_total = "DECIMAL(10,2)"),
  users = c("id", "name")
))
```

polyglot_version	<i>Versions of polyglotSQL and its embedded Rust engine</i>
------------------	---

Description

Versions of polyglotSQL and its embedded Rust engine

Usage

```
polyglot_version()
```

Value

A named character vector with elements polyglotSQL (the R package version) and polyglot_sql (the version of the vendored **polyglot-sql** Rust crate the package was compiled against).

Examples

```
polyglot_version()
```

sql_analyze

Structural query analysis

Description

Extracts compact facts about a query: its shape, output projections, referenced relations, CTEs, set operations and star-projections.

Usage

```
sql_analyze(sql, dialect = "generic", schema = NULL)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.
schema	Optional schema specification (see as_polyglot_schema()); improves resolution of unqualified or ambiguous columns.

Value

A polyglot_analysis object — a list with (among others):

- shape — query shape (e.g. "select", "setOperation");
- projections — list of per-output-column facts (name, transform kind, upstream column references, type hints when a schema is given);
- relations — list of referenced relations with kind and alias;
- ctes, cteFacts — CTE names and per-CTE facts;
- setOperations, starProjections — when present.

Examples

```
a <- sql_analyze("WITH x AS (SELECT id FROM t) SELECT x.id, 2 AS two FROM x")
a$shape
vapply(a$projections, function(p) p$name, character(1))
```

sql_annotate_types	<i>Annotate a query with inferred data types</i>
--------------------	--

Description

Runs upstream type inference over the AST. With a schema, column references resolve to their declared types; without one, only types that can be inferred from literals, casts and function signatures are filled.

Usage

```
sql_annotate_types(sql, dialect = "generic", schema = NULL)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.
schema	Optional schema specification (see as_polyglot_schema()); improves resolution of unqualified or ambiguous columns.

Value

A polyglot_ast object whose nodes carry an inferred_type field where a type could be determined. Pass it to [sql_generate\(\)](#) to render, or inspect \$statements directly.

Examples

```
ast <- sql_annotate_types(  
  "SELECT id + 1 AS next_id FROM t",  
  schema = list(t = c(id = "INT"))  
)  
# the Add node now carries inferred_type INT
```

sql_dialects	<i>List supported SQL dialects</i>
--------------	------------------------------------

Description

List supported SQL dialects

Usage

```
sql_dialects(full = FALSE)
```

Arguments

full If FALSE (default), return a character vector of canonical dialect names. If TRUE, return a data frame with columns name, aliases (comma-separated accepted aliases) and description.

Details

Every function that takes a dialect, from or to argument accepts both the canonical names and the aliases (e.g. "postgresql" for "postgres", "mssql" or "sqlserver" for "tsql").

Value

A character vector, or a data frame when full = TRUE.

Examples

```
sql_dialects()
head(sql_dialects(full = TRUE))
```

sql_diff	<i>Structural diff between two SQL statements</i>
----------	---

Description

Compares the ASTs of two statements and reports inserted, removed, moved and updated nodes.

Usage

```
sql_diff(sql_from, sql_to, dialect = "generic", delta_only = TRUE)
```

Arguments

sql_from, sql_to Single SQL statements to compare.

dialect Dialect used for parsing.

delta_only If TRUE (default), omit unchanged ("keep") nodes.

Value

A data frame with columns op ("insert", "remove", "move", "update", "keep"), expression (SQL of the affected node) and target (for updates, the new SQL).

Examples

```
sql_diff("SELECT a FROM t", "SELECT a, b FROM t WHERE a > 1")
```

sql_format	<i>Format (pretty-print) SQL</i>
------------	----------------------------------

Description

Parses and re-renders SQL as canonically indented statements. Complexity guards protect against pathological inputs; exceeding a guard raises a `polyglot_guard_error`.

Usage

```
sql_format(  
  sql,  
  dialect = "generic",  
  max_input_bytes = NULL,  
  max_tokens = NULL,  
  max_ast_nodes = NULL,  
  max_set_op_chain = NULL  
)
```

Arguments

- `sql` A single character string with one or more SQL statements (separated by ;).
- `dialect` Dialect used for parsing and rendering.
- `max_input_bytes, max_tokens, max_ast_nodes, max_set_op_chain` Complexity guard limits. `NULL` (default) uses the upstream defaults (16 MiB input, 1e6 tokens, 1e6 AST nodes, 256 chained set operations); `Inf` disables a guard; a positive number sets an explicit limit.

Value

A character vector with one formatted statement per input statement.

Examples

```
cat(sql_format("select a,b from t where x=1 and y=2"))
```

sql_generate	<i>Generate SQL from a parsed AST</i>
--------------	---------------------------------------

Description

Renders a `sql_parse()` result back into SQL text using the target dialect's syntax rules (keywords, quoting, literals). Note this is plain *generation*: unlike `sql_transpile()`, it does not apply cross-dialect function rewrites (e.g. `IFNULL` is not converted to `COALESCE`). Use it to render programmatically-built or modified ASTs; use `sql_transpile()` for full dialect translation.

Usage

```
sql_generate(ast, dialect = "generic")
```

Arguments

- ast A polyglot_ast object from `sql_parse()` (or `sql_annotate_types()`).
- dialect Target dialect for rendering.

Value

A character vector with one element per statement in the AST.

Examples

```
ast <- sql_parse("SELECT a, b FROM t WHERE x = 1")
sql_generate(ast, dialect = "postgres")
```

sql_lineage	<i>Column-level lineage</i>
-------------	-----------------------------

Description

Traces each output column of a query back to the tables and expressions it is derived from, following CTEs, subqueries and set operations.

Usage

```
sql_lineage(sql, dialect = "generic", schema = NULL, column = NULL)
```

Arguments

- sql A single character string with one or more SQL statements (separated by ;).
- dialect Dialect used for parsing.
- schema Optional schema specification (see `as_polyglot_schema()`); improves resolution of unqualified or ambiguous columns.
- column Optional single column name. By default, lineage is computed for every output column of the query.

Value

A polyglot_lineage object: a list with one entry per column, each containing

- column — the output column name;
- sources — character vector of source tables feeding this column;
- tree — the lineage graph as nested lists with fields name, source_name, source_kind ("Table", "Cte", "DerivedTable", ...), expression (the SQL of the node's expression) and downstream.

Examples

```
sql_lineage("SELECT a + b AS total FROM t")
sql_lineage(
  "WITH base AS (SELECT id, amount FROM payments)
  SELECT id, amount * 2 AS doubled FROM base"
)
```

sql_openlineage	<i>OpenLineage column-lineage facet</i>
-----------------	---

Description

Produces an **OpenLineage**-compatible columnLineage facet plus inferred input/output datasets for a SQL statement, for integration with data catalogs and lineage backends.

Usage

```
sql_openlineage(
  sql,
  dialect = "generic",
  schema = NULL,
  namespace = NULL,
  job_name = NULL
)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.
schema	Optional schema specification (see as_polyglot_schema()); improves resolution of unqualified or ambiguous columns.
namespace	Optional dataset namespace applied to inferred datasets.
job_name	Optional job name recorded in the facet.

Value

A list with elements facet (the OpenLineage columnLineage facet), inputs, outputs (dataset descriptors) and warnings.

Examples

```
ol <- sql_openlineage(
  "INSERT INTO reports SELECT id, total FROM sales",
  namespace = "warehouse"
)
names(ol)
```

sql_optimize	<i>Optimize SQL</i>
--------------	---------------------

Description

Applies the upstream optimizer rule set (predicate pushdown, join reordering, CTE and subquery elimination, expression simplification, etc.) and returns the rewritten SQL.

Usage

```
sql_optimize(sql, dialect = "generic", schema = NULL)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.
schema	Optional schema specification (see as_polyglot_schema()); improves resolution of unqualified or ambiguous columns.

Value

A character vector with one optimized statement per input statement.

Examples

```
sql_optimize("SELECT * FROM (SELECT a FROM t) AS sub WHERE sub.a > 1")
```

sql_parse	<i>Parse SQL into an abstract syntax tree</i>
-----------	---

Description

Parse SQL into an abstract syntax tree

Usage

```
sql_parse(sql, dialect = "generic")
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.

Value

A `polyglot_ast` object: a list with elements

- `statements` — a list with one nested-list AST per statement;
- `sql` — the input SQL;
- `dialect` — the dialect used.

Each AST node is a named list; the name of the outer element gives the node kind (e.g. "select"). The structure follows the upstream `polyglot-sql` JSON AST format and round-trips through [sql_generate\(\)](#).

Examples

```
ast <- sql_parse("SELECT a, b FROM t WHERE x = 1")
ast
names(ast$statements[[1]])
```

<code>sql_source_tables</code>	<i>List source tables referenced by SQL</i>
--------------------------------	---

Description

Returns the physical tables a query reads from, across all statements. CTE names are not included (they are intermediate results, not sources).

Usage

```
sql_source_tables(sql, dialect = "generic")
```

Arguments

<code>sql</code>	A single character string with one or more SQL statements (separated by ;).
<code>dialect</code>	Dialect used for parsing.

Value

A character vector of table names, in order of first appearance.

Examples

```
sql_source_tables("SELECT * FROM a JOIN b ON a.id = b.id")
sql_source_tables("WITH x AS (SELECT 1 FROM t) SELECT * FROM x")
```

 sql_tokenize

Tokenize SQL

Description

Splits SQL into lexical tokens using the tokenizer of the given dialect.

Usage

```
sql_tokenize(sql, dialect = "generic")
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.

Value

A data frame with class `polyglot_tokens` and one row per token: `type` (token type, e.g. "Select", "Identifier", "Number"), `text` (raw token text), `line` and `column` (1-based position), `start` and `end` (byte offsets, end exclusive).

Examples

```
sql_tokenize("SELECT a FROM t")
```

 sql_transpile

Translate SQL between dialects

Description

Parses `sql` with the `from` dialect and regenerates it in the `to` dialect, rewriting functions, quoting, and constructs as needed (e.g. MySQL `IFNULL()` becomes PostgreSQL `COALESCE()`).

Usage

```
sql_transpile(
  sql,
  from,
  to,
  pretty = FALSE,
  unsupported = c("raise", "warn", "ignore")
)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
from	Source dialect name (see sql_dialects()).
to	Target dialect name (see sql_dialects()).
pretty	If TRUE, pretty-print the output.
unsupported	How to handle constructs that cannot be represented in the target dialect: "raise" (default) throws a <code>polyglot_transpile_error</code> ; "warn" and "ignore" continue and return the closest supported translation (with "warn", upstream collects diagnostics but still returns a result).

Value

A character vector with one element per input statement.

Semantic limitations

Transpilation is syntactic and best-effort: identical syntax can still behave differently across engines (implicit casts, collations, NULL ordering, integer division, time zone handling...). Always test the translated SQL against the target database before using it in production.

See Also

[sql_format\(\)](#), [sql_parse\(\)](#), [sql_validate\(\)](#)

Examples

```
sql_transpile("SELECT IFNULL(a, b) FROM t", from = "mysql", to = "postgres")
sql_transpile(
  "SELECT DATE_TRUNC('month', created_at) FROM events",
  from = "postgres", to = "duckdb"
)
```

sql_validate	<i>Validate SQL</i>
--------------	---------------------

Description

Checks that SQL parses in the given dialect and, optionally, applies stricter syntax rules, semantic lint warnings, and schema-aware checks.

Usage

```
sql_validate(sql, dialect = "generic", ...)
```

Arguments

sql	A single character string with one or more SQL statements (separated by ;).
dialect	Dialect used for parsing.
...	Additional validation options: • <code>strict_syntax</code> — reject non-canonical syntax the parser would accept for compatibility (e.g. trailing commas before FROM); default FALSE. • <code>semantic</code> — report query-quality warnings (W001–W004, e.g. <code>SELECT *</code> mixed with explicit columns) and semantic correctness errors (E230–E232: ungrouped columns, misplaced aggregates and misplaced window functions). The errors invalidate the statement even when <code>strict_syntax = FALSE</code>; default FALSE. • <code>schema</code> — a schema specification (see as_polyglot_schema()); enables unknown-table/column, type and reference checks. • <code>error</code> — if TRUE, raise a <code>polyglot_validation_error</code> instead of returning an invalid result; default FALSE.

Value

A `polyglot_validation` object: a list with `valid` (logical) and `errors` (data frame with columns `severity`, `code`, `message`, `line`, `column`).

Examples

```
sql_validate("SELECT a FROM t")
sql_validate("SELECT FROM WHERE")
v <- sql_validate("SELECT a, FROM t", strict_syntax = TRUE)
v$valid
```

Index

`as_polyglot_schema`, [2](#)
`as_polyglot_schema()`, [4](#), [5](#), [8–10](#), [14](#)

`polyglot_version`, [3](#)

`sql_analyze`, [4](#)
`sql_analyze()`, [2](#)
`sql_annotate_types`, [5](#)
`sql_annotate_types()`, [2](#), [8](#)
`sql_dialects`, [5](#)
`sql_dialects()`, [13](#)
`sql_diff`, [6](#)
`sql_format`, [7](#)
`sql_format()`, [13](#)
`sql_generate`, [7](#)
`sql_generate()`, [5](#), [11](#)
`sql_lineage`, [8](#)
`sql_lineage()`, [2](#)
`sql_openlineage`, [9](#)
`sql_openlineage()`, [2](#)
`sql_optimize`, [10](#)
`sql_optimize()`, [2](#)
`sql_parse`, [10](#)
`sql_parse()`, [7](#), [8](#), [13](#)
`sql_source_tables`, [11](#)
`sql_tokenize`, [12](#)
`sql_transpile`, [12](#)
`sql_transpile()`, [7](#)
`sql_validate`, [13](#)
`sql_validate()`, [2](#), [13](#)