Package 'dbplyr'

June 17, 2026

Type Package

Title A 'dplyr' Back End for Databases

Version 2.6.0

Description A 'dplyr' back end for databases that allows you to work with remote database tables as if they are in-memory data frames. Basic features work with any database that has a 'DBI' back end; more advanced features require 'SQL' translation to be provided by the package author.

License MIT + file LICENSE

URL https://dbplyr.tidyverse.org/, https://github.com/tidyverse/dbplyr

BugReports https://github.com/tidyverse/dbplyr/issues

Depends R (>= 4.1)

Imports blob (>= 1.2.0), cli (>= 3.6.1), DBI (>= 1.1.3), dplyr (>= 1.1.2), glue (>= 1.6.2), lifecycle (>= 1.0.3), magrittr, methods, pillar (>= 1.9.0), purrr (>= 1.0.1), R6 (>= 2.2.2), rlang (>= 1.1.1), tibble (>= 3.2.1), tidyr (>= 1.3.0), tidyselect (>= 1.2.1), utils, vctrs (>= 0.6.3), withr (>= 2.5.0)

Suggests adbcdrivermanager, adbcsqlite, adbi, bit64, covr, knitr, Lahman, nycflights13, odbc (>= 1.4.2), RJDBC, RMariaDB (>= 1.2.2), rmarkdown, RPostgres (>= 1.4.5), RPostgreSQL, RSQLite (>= 2.3.8), testthat (>= 3.1.10)

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel TRUE

Encoding UTF-8

Language en-gb

Collate 'verb-copy-inline.R' 'verb-copy-to.R' 'db-sql.R' 'db.R' 'utils-check.R' 'import-standalone-types-check.R'

'import-standalone-obj-type.R' 'utils.R' 'sql.R' 'escape.R'
'translate-sql-cut.R' 'translate-sql-string.R'
'translate-sql-aggregate.R' 'translate-sql-scalar.R'
'translate-sql-helpers.R' 'translate-sql-window.R'
'translate-sql-conditional.R' 'backend-.R' 'backend-access.R'
'backend-adbc.R' 'backend-db2.R' 'backend-hana.R'
'backend-hive.R' 'backend-impala.R' 'backend-jdbc.R'
'backend-mssql.R' 'backend-mysql.R' 'backend-odbc.R'
'backend-oracle.R' 'backend-postgres.R'
'backend-postgres-old.R' 'backend-redshift.R'
'backend-snowflake.R' 'backend-spark-sql.R' 'backend-sqlite.R'
'backend-teradata.R' 'backward-compatibility.R'
'bind-queries.R' 'data-cache.R' 'data-lahman.R'
'data-nycflights13.R' 'db-io.R' 'dbplyr.R' 'ident.R'
'import-standalone-s3-register.R' 'join-by-compat.R'
'join-cols-compat.R' 'lazy-ops.R' 'memdb.R' 'optimise-utils.R'
'progress.R' 'query-base.R' 'query-join.R' 'query-rf-join.R'
'query-select.R' 'query-semi-join.R' 'query-set-op.R'
'query-union.R' 'query.R' 'remote.R' 'rows.R' 'schema.R'
'sql-build.R' 'sql-clause.R' 'sql-dialect.R' 'sql-glue.R'
'sql-quote.R' 'sql-superseded.R' 'src-sql.R' 'src_dbi.R'
'table-name.R' 'tbl-lazy.R' 'tbl-sql.R' 'tidyeval-across.R'
'tidyeval.R' 'translate-sql.R' 'utils-format.R'
'verb-arrange.R' 'verb-collapse.R' 'verb-collect.R'
'verb-compute.R' 'verb-count.R' 'verb-distinct.R'
'verb-do-query.R' 'verb-do.R' 'verb-expand.R' 'verb-explain.R'
'verb-fill.R' 'verb-filter.R' 'verb-group_by.R' 'verb-head.R'
'verb-joins.R' 'verb-mutate.R' 'verb-pivot-longer.R'
'verb-pivot-wider.R' 'verb-pull.R' 'verb-select.R'
'verb-set-ops.R' 'verb-slice.R' 'verb-summarise.R'
'verb-uncount.R' 'verb-window.R' 'with-dialect.R' 'zzz.R'

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Hadley Wickham [aut, cre],
Maximilian Girlich [aut],
Edgar Ruiz [aut],
Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2026-06-17 21:50:07 UTC

Contents

.sql 4
arrange.tbl_lazy 5
backend-access 6

backend-adbc . . . 6
backend-ansi . . . 7
backend-db2 . . . 8
backend-hana . . . 9
backend-hive . . . 9
backend-impala . . . 10
backend-jdbc . . . 11
backend-mssql . . . 12
backend-mysql . . . 13
backend-odbc . . . 14
backend-oracle . . . 15
backend-postgres . . . 16
backend-redshift . . . 16
backend-snowflake . . . 17
backend-spark-sql . . . 18
backend-sqlite . . . 18
backend-teradata . . . 19
bind_queries . . . 20
collapse.tbl_sql . . . 20
collect.tbl_sql . . . 21
complete.tbl_lazy . . . 22
compute.tbl_sql . . . 23
copy_inline . . . 24
copy_to.src_sql . . . 25
count.tbl_lazy . . . 26
dbplyr-slice . . . 27
dbplyr_uncount . . . 29
distinct.tbl_lazy . . . 30
do.tbl_sql . . . 30
escape . . . 31
expand.tbl_lazy . . . 32
fill.tbl_lazy . . . 34
filter.tbl_lazy . . . 35
get_returned_rows . . . 36
group_by.tbl_lazy . . . 37
head.tbl_lazy . . . 38
intersect.tbl_lazy . . . 39
join.tbl_sql . . . 40
last_sql . . . 44
memdb . . . 45
mutate.tbl_lazy . . . 46
pivot_longer.tbl_lazy . . . 48
pivot_wider.tbl_lazy . . . 50
pull.tbl_sql . . . 53
remote_name . . . 54
replace_na.tbl_lazy . . . 55
rows_insert.tbl_lazy . . . 56
select.tbl_lazy . . . 59

show_query 60
sql 61
sql_dialect 61
sql_glue 63
sql_options 64
sql_query_insert 65
sql_translation_agg 68
sql_translation_scalar 69
sql_translation_string 70
sql_translation_window 71
sql_variant 72
summarise.tbl_lazy 74
tbl.src_dbi 75
translate_sql 76
window_order 78
with_dialect 79

Index **80**

`.sql` *Flag SQL function usage*

Description

Use `.sql$foo(x, y)` to make it clear that you're calling the SQL `foo()` function, not the R `foo()` function. This also makes it easier to reduce `R CMD check` notes in packages; just import `.sql` from dbplyr with e.g. `@importFrom dbplyr .sql`.

Note that `.sql` itself does nothing and is just `NULL`; it is automatically removed when dbplyr translates your R code to SQL.

Usage

```
.sql
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- lazy_frame(x = 1, y = 2)
db |> mutate(z = .sql$CUMULATIVE_SUM(x, 1))
```

`arrange.tbl_lazy` *Arrange rows by column values*

Description

This is an method for the dplyr `arrange()` generic. It generates the `ORDER BY` clause of the SQL query. It also affects the `window_order()` of windowed expressions in `mutate.tbl_lazy()`.

Note that `ORDER BY` clauses can not generally appear in subqueries, which means that you should `arrange()` as late as possible in your pipelines.

Usage

```
## S3 method for class 'tbl_lazy'
arrange(.data, ..., .by_group = FALSE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Missing values

Unlike R, most databases sorts `NA` (`NULL`s) at the front. You can can override this behaviour by explicitly sorting on `is.na(x)`.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db |> arrange(a) |> show_query()

# Note that NAs are sorted first
db |> arrange(b)
# override by sorting on is.na() first
db |> arrange(is.na(b), b)
```

backend-access *MS Access backend*

Description

This backend supports Microsoft Access databases, typically accessed via odbc. Use `dialect_access()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- `SELECT` uses `TOP`, not `LIMIT`
- Non-standard types and mathematical functions
- String concatenation uses &
- No `ANALYZE` equivalent
- `TRUE` and `FALSE` converted to 1 and 0

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_access()

simulate_access()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)
lf <- lazy_frame(x = 1, y = 2, z = "a", con = dialect_access())

lf |> head()
lf |> mutate(y = as.numeric(y), z = sqrt(x^2 + 10))
lf |> mutate(a = paste0(z, " times"))
```

backend-adbc *ADBC backend*

Description

This backend supports databases accessed via `AdbiConnection` created by `adbi::adbi()` and `DBI::dbConnect()`. dbplyr automatically detects the underlying database type by querying the ADBC driver's vendor name and uses the appropriate SQL dialect.

The following vendors are recognized:

- PostgreSQL and PostgreSQL-compatible systems (CrateDB, CockroachDB, Citus, Neon, ParadeDB, TimescaleDB, Yellowbrick, YugabyteDB, CedarDB): `dialect_postgres()`

- MySQL and MySQL-compatible systems (MariaDB, TiDB, Vitess): the appropriate dialect (`dialect_mariadb()` for MariaDB, `dialect_mysql()` otherwise)
- SQLite: `dialect_sqlite()`
- Microsoft SQL Server: `dialect_mssql()`
- Snowflake: `dialect_snowflake()`
- Amazon Redshift: `dialect_redshift()`

If your database is not recognized, dbplyr will fall back to a generic ODBC dialect. In this case, or if dbplyr guesses wrong, you can use `with_dialect()` to choose a specific dialect.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Examples

```
# ADBC connections require the adbi package and an ADBC driver.
# Once connected, dbplyr automatically detects the database type:
#
# library(adbi)
# library(dplyr, warn.conflicts = FALSE)
#
# con <- DBI::dbConnect(adbi::adbi("adbcsqlite"), uri = ":memory:")
# tbl(con, "my_table") |> filter(x > 1)
```

---

backend-ansi *ANSI SQL backend*

Description

This is the base dialect for ANSI compliant SQL, forming the foundation of all other dialects. Use `dialect_ansi()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

See `vignette("translation-function")` for a list of functions that are translated.

Usage

```
dialect_ansi()

simulate_dbi(class = character(), ...)
```

Arguments

`class, ...`	No longer used.

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_ansi())
lf |> transmute(x = mean(b, na.rm = TRUE))
lf |> transmute(x = log(b), y = log(b, base = 2))
```

backend-db2 *DB2 backend*

Description

This backend supports IBM DB2 databases, typically accessed via ODBC. Use `dialect_db2()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Uses `FETCH FIRST n ROWS ONLY` instead of `LIMIT n`
- Uses double quotes for identifier quoting
- `paste()` uses `||`
- DB2-specific data type names for casts (e.g. `VARCHAR(255)`, `DOUBLE`)
- Date component extraction via `YEAR()`, `MONTH()`, ..., `DAYOFYEAR()`, `DAYOFWEEK()`, `QUARTER()`, `WEEK()`
- `str_flatten()` uses `LISTAGG`
- Statistical summaries `sd()`, `var()`, `cor()`, `cov()`
- `runif()` translates to `RAND()`
- Regular expression functions (DB2 11.1+)

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_db2()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_db2())
lf |> head()
lf |> transmute(x = paste0(d, " times"))
lf |> summarise(x = sd(b, na.rm = TRUE))
```

backend-hana *SAP HANA backend*

Description

This backend supports SAP HANA databases, typically accessed via `HDBConnection` created by `DBI::dbConnect()`. Use `dialect_hana()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Temporary tables get # prefix and use `LOCAL TEMPORARY COLUMN`.
- No table analysis performed in `copy_to()`.
- `paste()` uses `||`
- Note that you can't create new boolean columns from logical expressions; you need to wrap with explicit `ifelse`: `ifelse(x > y, TRUE, FALSE)`.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_hana()

simulate_hana()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_hana())
lf |> transmute(x = paste0(d, " times"))
```

backend-hive *Hive backend*

Description

This backend supports Apache Hive, typically accessed via odbc. Use `dialect_hive()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are a scattering of custom translations provided by users.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_hive()

simulate_hive()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = dialect_hive())
lf |> transmute(x = cot(b))
lf |> transmute(x = bitwShiftL(c, 1L))
lf |> transmute(x = str_replace_all(c, "a", "b"))

lf |> summarise(x = median(d, na.rm = TRUE))
lf |> summarise(x = var(c, na.rm = TRUE))
```

---

backend-impala *Impala backend*

Description

This backend supports Apache Impala, typically accessed via odbc. Use `dialect_impala()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are a scattering of custom translations provided by users, mostly focussed on bitwise operations.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_impala()

simulate_impala()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_impala())
lf |> transmute(X = bitwNot(bitwOr(b, c)))
```

backend-jdbc *JDBC backend*

Description

This backend supports databases accessed via `JDBCConnection` created by `RJDBC::JDBC()` and `DBI::dbConnect()`. dbplyr automatically detects the underlying database type from the JDBC connection class and uses the appropriate SQL dialect.

The following databases are recognized via their JDBC connection class:

- PostgreSQL (`org.postgresql.*`): `dialect_postgres()`
- MySQL (`com.mysql.*`): `dialect_mysql()`
- MariaDB (`org.mariadb.*`): `dialect_mariadb()`
- SQLite (`org.sqlite.*`): `dialect_sqlite()`
- Oracle (`oracle.*`): `dialect_oracle()`
- SQL Server (`com.microsoft.sqlserver.*`): `dialect_mssql()`
- IBM DB2 (`com.ibm.db2.*`): `dialect_db2()`
- SAP HANA (`com.sap.db.*`): `dialect_hana()`
- Teradata (`com.teradata.*`): `dialect_teradata()`
- Apache Hive (`org.apache.hive.*`): `dialect_hive()`
- Apache Spark (`org.apache.spark.*` or `com.simba.spark.*`): `dialect_spark_sql()`
- Snowflake (`net.snowflake.*`): `dialect_snowflake()`
- Impala (`com.cloudera.impala.*`): `dialect_impala()`
- Amazon Redshift (`com.amazon.redshift.*`): `dialect_redshift()`

If your database is not recognized, dbplyr will fall back to a generic ODBC dialect. Please file an issue if you'd like support for additional databases.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

See Also

`with_dialect()` to use a different dialect if dbplyr guesses incorrectly, or a more specific translation is available.

Examples

```
# JDBC connections require the RJDBC package and a JDBC driver JAR file.
# Once connected, dbplyr automatically detects the database type:
#
# library(RJDBC)
# library(dplyr, warn.conflicts = FALSE)
#
# drv <- JDBC("org.postgresql.Driver", "postgresql.jar")
# con <- dbConnect(drv, "jdbc:postgresql://localhost/mydb", "user", "password")
# tbl(con, "my_table") |> filter(x > 1)
```

backend-mssql *SQL Server backend*

Description

This backend supports Microsoft SQL Server, typically accessed via odbc. Use `dialect_mssql()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- `SELECT` uses `TOP` not `LIMIT`
- Automatically prefixes # to create temporary tables. Add the prefix yourself to avoid the message.
- String basics: `paste()`, `substr()`, `nchar()`
- Custom types for `as.*` functions
- Lubridate extraction functions, `year()`, `month()`, `day()` etc
- Semi-automated bit <-> boolean translation (see below)
- stringr functions `str_detect()`, `str_starts()`, `str_ends()` with `fixed()` patterns work on all versions; regular expression patterns require SQL Server 2025+ (version 17.0)
- stringr functions `str_replace()`, `str_replace_all()`, `str_remove()`, `str_remove_all()`, `str_extract()`, and `str_count()` require SQL Server 2025+ (version 17.0)

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_mssql(version = "15.0")

simulate_mssql(version = "15.0")
```

Arguments

`version`	Version of MS SQL to simulate. Currently, 11.0 and above will use `TRY_CAST()` instead of `CAST()`, and 17.0 and above will support regular expression patterns in stringr functions.

Bit vs boolean

SQL server uses two incompatible types to represent `TRUE` and `FALSE` values:

- The `BOOLEAN` type is the result of logical comparisons (e.g. `x > y`) and can be used `WHERE` but not to create new columns in `SELECT`. https://learn.microsoft.com/en-us/sql/t-sql/language-elements/comparison-operators-transact-sql
- The `BIT` type is a special type of numeric column used to store `TRUE` and `FALSE` values, but can't be used in `WHERE` clauses. https://learn.microsoft.com/en-us/sql/t-sql/data-types/bit-transact-sql?view=sql-server-ver15

dbplyr does its best to automatically create the correct type when needed, but can't do it 100% correctly because it does not have a full type inference system. This means that you many need to manually do conversions from time to time.

- To convert from bit to boolean use `x == 1`
- To convert from boolean to bit use `as.logical(if(x, 0, 1))`

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_mssql())
lf |> head()
lf |> transmute(x = paste(b, c, d))

# Can use boolean as is:
lf |> filter(c > d)
# Need to convert from boolean to bit:
lf |> transmute(x = c > d)
# Can use boolean as is:
lf |> transmute(x = ifelse(c > d, "c", "d"))
```

---

backend-mysql *MySQL/MariaDB backend*

Description

This backend supports MySQL and MariaDB databases, typically accessed via `MySQLConnection` or `MariaDBConnection` created by `DBI::dbConnect()`. Use `dialect_mysql()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- `paste()` uses `CONCAT_WS()`
- String translations for `str_detect()`, `str_locate()`, and `str_replace_all()`
- Clear error message for unsupported full joins

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_mariadb()

dialect_mysql()

simulate_mysql()

simulate_mariadb()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_mysql())
lf |> transmute(x = paste0(d, " times"))
```

---

backend-odbc *ODBC backend*

Description

This backend supports databases accessed via `OdbcConnection` created by `DBI::dbConnect()`. Use `dialect_odbc()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are minor translations for common data types.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_odbc()

simulate_odbc()
```

See Also

`with_dialect()` to use a different dialect if dbplyr guesses incorrectly, or a more specific translation is available.

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = dialect_odbc())
lf |> transmute(x = as.numeric(b))
lf |> transmute(x = as.integer(b))
lf |> transmute(x = as.character(b))
```

backend-oracle *Oracle backend*

Description

This backend supports Oracle databases, typically accessed via `OraConnection` created by `DBI::dbConnect()`. Use `dialect_oracle()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Use `FETCH FIRST` instead of `LIMIT`
- Custom types
- `paste()` uses `||`
- Custom subquery generation (no `AS`)
- `setdiff()` uses `MINUS` instead of `EXCEPT`

Note that versions of Oracle prior to 23c have limited supported for `TRUE` and `FALSE` and you may need to use `1` and `0` instead. See https://oracle-base.com/articles/23/boolean-data-type-23 for more details.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_oracle()

simulate_oracle()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_oracle())
lf |> transmute(x = paste0(c, " times"))
lf |> setdiff(lf)
```

backend-postgres *PostgreSQL backend*

Description

This backend supports PostgreSQL databases, typically accessed via a `PqConnection` created by `DBI::dbConnect()`. Use `dialect_postgres()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Many stringr functions
- lubridate date-time extraction functions
- More standard statistical summaries

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_postgres()

simulate_postgres()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_postgres())
lf |> summarise(x = sd(b, na.rm = TRUE))
lf |> summarise(y = cor(b, c), z = cov(b, c))
```

backend-redshift *Redshift backend*

Description

This backend supports Amazon Redshift databases, typically accessed via a `RedshiftConnection` created by `DBI::dbConnect()`. Use `dialect_redshift()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Base translations come from PostgreSQL backend. There are generally few differences, apart from string manipulation.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_redshift()

simulate_redshift()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_redshift())
lf |> transmute(x = paste(c, " times"))
lf |> transmute(x = substr(c, 2, 3))
lf |> transmute(x = str_replace_all(c, "a", "z"))
```

---

backend-snowflake *Snowflake backend*

Description

This backend supports Snowflake databases, typically accessed via odbc. Use `dialect_snowflake()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_snowflake()

simulate_snowflake()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_snowflake())
lf |> transmute(x = paste0(d, " times"))
```

backend-spark-sql *Databricks Spark SQL backend*

Description

This backend supports Databricks Spark SQL, typically accessed via the Databricks ODBC or JDBC connector. Use `dialect_spark_sql()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are better translation of statistical aggregate functions (e.g. `var()`, `median()`) and use of temporary views instead of temporary tables when copying data.

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_spark_sql()

simulate_spark_sql()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = dialect_spark_sql())

lf |> summarise(x = median(d, na.rm = TRUE))
lf |> summarise(x = var(c, na.rm = TRUE), .by = d)

lf |> mutate(x = first(c))
lf |> mutate(x = first(c), .by = d)
```

backend-sqlite *SQLite backend*

Description

This backend supports SQLite databases, typically accessed via a `SQLiteConnection` created by `DBI::dbConnect()`. Use `dialect_sqlite()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Uses non-standard `LOG()` function
- Date-time extraction functions from lubridate
- Custom median translation
- Right and full joins are simulated using left joins

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_sqlite()

simulate_sqlite()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_sqlite())
lf |> transmute(x = paste(c, " times"))
lf |> transmute(x = log(b), y = log(b, base = 2))
```

---

backend-teradata *Teradata backend*

Description

This backend supports Teradata databases, typically accessed via odbc. Use `dialect_teradata()` with `lazy_frame()` to see simulated SQL without connecting to a live database.

Key differences for this backend are:

- Uses `TOP` instead of `LIMIT`
- Selection of user supplied translations

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

Usage

```
dialect_teradata()

simulate_teradata()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = dialect_teradata())
lf |> head()
```

bind_queries *Combine multiple lazy queries*

Description

Combine multiple lazy queries into a single query using `UNION ALL`. This is a convenient wrapper around `purrr::reduce(tables, union_all)`. Like `dplyr::bind_rows()` (and unlike `UNION ALL`), `bind_queries()` will automatically align columns based on their name, and fill in any missing columns with missing values.

Usage

```
bind_queries(...)
```

Arguments

`...`	<dynamic-dots> Lazy tables to combine.

Value

A lazy query.

Examples

```
lf1 <- lazy_frame(x = 1, y = "a")
lf2 <- lazy_frame(x = 2, y = "b")
bind_queries(lf1, lf2)

lf3 <- lazy_frame(y = "c", x = 3, z = 10)
bind_queries(lf2, lf3)

# If you already have a list, you can use splice operator
queries <- list(lf1, lf2)
bind_queries(!!!queries)
```

collapse.tbl_sql *Collapse a query into a subquery*

Description

`collapse()` forces computation of a lazy query by wrapping it in a subquery. This is not generally needed, but can be useful if you need to work around database/dbplyr limitations.

Usage

```
## S3 method for class 'tbl_sql'
collapse(x, ...)
```

Arguments

x	A lazy data frame backed by a database query.
...	Ignored.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db |> filter(a <= 2) |> show_query()
db |> filter(a <= 2) |> collapse() |> show_query()
```

---

collect.tbl_sql *Collect results into a local data frame*

Description

`collect()` executes the query and retrieves the results into a local tibble. This brings all the data from the database into R's memory, which is useful once you've done as much as possible in the database, and now need to use R functions.

Usage

```
## S3 method for class 'tbl_sql'
collect(
  x,
  ...,
  n = Inf,
  warn_incomplete = TRUE,
  sql_options = NULL,
  cte = deprecated()
)
```

Arguments

x	A lazy data frame backed by a database query.
...	Ignored.
n	Number of rows to fetch. Defaults to `Inf`, meaning all rows.
warn_incomplete	Warn if `n` is less than the number of result rows?
sql_options	SQL rendering options generated by `sql_options()`.
cte	**[Deprecated]** Use the `sql_options` argument instead.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db |> filter(a <= 2) |> collect()
```

`complete.tbl_lazy` *Complete a SQL table with missing combinations of data*

Description

Turns implicit missing values into explicit missing values. This is a method for the `tidyr::complete()` generic.

Usage

```
## S3 method for class 'tbl_lazy'
complete(data, ..., fill = list())
```

Arguments

`data`	A lazy data frame backed by a database query.
`...`	Specification of columns to expand. See tidyr::expand for more details.
`fill`	A named list that for each variable supplies a single value to use instead of NA for missing combinations.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
df <- memdb_frame(
  group = c(1:2, 1),
  item_id = c(1:2, 2),
  item_name = c("a", "b", "b"),
  value1 = 1:3,
  value2 = 4:6
)

df |> tidyr::complete(group, nesting(item_id, item_name))

# You can also choose to fill in missing values
df |> tidyr::complete(group, nesting(item_id, item_name), fill = list(value1 = 0))
```

`compute.tbl_sql` *Save results into a new remote table*

Description

`compute()` executes the query and stores the results in a new remote table. This is useful when you want to cache intermediate results for reuse or to improve performance by avoiding repeated computation of complex queries.

Usage

```
## S3 method for class 'tbl_sql'
compute(
  x,
  name = NULL,
  temporary = TRUE,
  overwrite = FALSE,
  unique_indexes = list(),
  indexes = list(),
  analyze = TRUE,
  ...,
  sql_options = NULL,
  cte = deprecated()
)
```

Arguments

`x`	A lazy data frame backed by a database query.
`name`	Name of new remote table. Use a string to create the table in the current catalog/schema. Use `I()` to create the table in a specific catalog/schema, e.g. `I("schema.table")`.
`temporary`	if `TRUE`, will create a temporary table that is local to this connection and will be automatically deleted when the connection expires
`overwrite`	If `TRUE`, will overwrite an existing table with name `name`. If `FALSE`, will throw an error if `name` already exists.
`unique_indexes`	a list of character vectors. Each element of the list will create a new unique index over the specified column(s). Duplicate rows will result in failure.
`indexes`	a list of character vectors. Each element of the list will create a new index.
`analyze`	if `TRUE` (the default), will automatically ANALYZE the new table so that the query optimiser has useful information.
`...`	Ignored.
`sql_options`	SQL rendering options generated by `sql_options()`.
`cte`	**[Deprecated]** Use the `sql_options` argument instead.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db |> filter(a <= 2) |> show_query()
db |> filter(a <= 2) |> compute() |> show_query()
```

`copy_inline` *Use a local data frame in a dbplyr query*

Description

This is an alternative to `copy_to()` that does not need write access and is faster for small data.

Usage

```
copy_inline(con, df, types = NULL)
```

Arguments

`con`	A database connection.
`df`	A local data frame. The data is written directly in the SQL query so it should be small.
`types`	A named character vector of SQL data types to use for the columns. The data types are backend specific. For example for Postgres this could be `c(id = "bigint", created_at = "timestamp", values = "integer[]")`. If `NULL`, the default, the types are determined from `df`.

Details

It writes the data directly in the SQL query via the `VALUES` clause.

Value

A `tbl_lazy`.

See Also

`copy_to()` to copy the data into a new database table.

Examples

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

copy_inline(con, df)

copy_inline(con, df) |> dplyr::show_query()
```

copy_to.src_sql *Copy a local data frame to a remote database*

Description

This is an implementation of the dplyr copy_to() generic and it mostly a wrapper around DBI::dbWriteTable().

It is useful for copying small amounts of data to a database for examples, experiments, and joins. By default, it creates temporary tables which are only visible within the current connection to the database.

Usage

```
## S3 method for class 'src_sql'
copy_to(
  dest,
  df,
  name = deparse(substitute(df)),
  overwrite = FALSE,
  types = NULL,
  temporary = TRUE,
  unique_indexes = NULL,
  indexes = NULL,
  analyze = TRUE,
  ...,
  in_transaction = TRUE
)
```

Arguments

dest	remote data source
df	A local data frame, a tbl_sql from same source, or a tbl_sql from another source. If from another source, all data must transition through R in one pass, so it is only suitable for transferring small amounts of data.
name	Name of new remote table. Use a string to create the table in the current catalog/schema. Use I() to create the table in a specific catalog/schema, e.g. I("schema.table").
overwrite	If TRUE, will overwrite an existing table with name name. If FALSE, will throw an error if name already exists.
types	a character vector giving variable types to use for the columns. See https://www.sqlite.org/datatype3.html for available types.
temporary	if TRUE, will create a temporary table that is local to this connection and will be automatically deleted when the connection expires
unique_indexes	a list of character vectors. Each element of the list will create a new unique index over the specified column(s). Duplicate rows will result in failure.

indexes	a list of character vectors. Each element of the list will create a new index.
analyze	if TRUE (the default), will automatically ANALYZE the new table so that the query optimiser has useful information.
...	other parameters passed to methods.
in_transaction	Should the table creation be wrapped in a transaction? This typically makes things faster, but you may want to suppress if the database doesn't support transactions, or you're wrapping in a transaction higher up (and your database doesn't support nested transactions.)

Value

Another tbl_lazy. Use show_query() to see the generated query, and use collect() to execute the query and return data to R.

See Also

copy_inline() to use small data in an SQL query without actually writing to a table.

Examples

```
library(dplyr, warn.conflicts = FALSE)

df <- data.frame(x = 1:5, y = letters[5:1])
db <- copy_to(memdb(), df)
db

df2 <- data.frame(y = c("a", "d"), fruit = c("apple", "date"))
# copy_to() is called automatically if you set copy = TRUE
# in the join functions
db |> left_join(df2, copy = TRUE)
```

---

count.tbl_lazy *Count observations by group*

Description

These are methods for the dplyr dplyr::count() and tally() generics. They wrap up group_by.tbl_lazy(), summarise.tbl_lazy() and, optionally, arrange.tbl_lazy().

Usage

```
## S3 method for class 'tbl_lazy'
count(x, ..., wt = NULL, sort = FALSE, name = NULL)

## S3 method for class 'tbl_lazy'
add_count(x, ..., wt = NULL, sort = FALSE, name = NULL, .drop = NULL)

## S3 method for class 'tbl_lazy'
tally(x, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

x	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
...	<data-masking> Variables, or functions of variables. Use desc() to sort a variable in descending order.
wt	<data-masking> Frequency weights. Can be NULL or a variable: • If NULL (the default), counts the number of rows in each group. • If a variable, computes sum(wt) for each group.
sort	If TRUE, will show the largest groups at the top.
name	The name of the new column in the output. If omitted, it will default to n. If there's already a column called n, it will use nn. If there's a column called n and nn, it'll use nnn, and so on, adding ns until it gets a new name.
.drop	Not supported for lazy tables.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db |> count(g) |> show_query()
db |> count(g, wt = x) |> show_query()
db |> count(g, wt = x, sort = TRUE) |> show_query()
```

---

dbplyr-slice *Subset rows using their positions*

Description

These are methods for the dplyr generics slice_min(), slice_max(), and slice_sample(). They are translated to SQL using dplyr::filter() and window functions (ROWNUMBER, MIN_RANK, or CUME_DIST depending on arguments). slice(), slice_head(), and slice_tail() are not supported since database tables have no intrinsic order.

If data is grouped, the operation will be performed on each group so that (e.g.) slice_min(db, x, n = 3) will select the three rows with the smallest value of x in each group.

Usage

```
## S3 method for class 'tbl_lazy'
slice_min(
  .data,
  order_by,
  ...,
  n,
  prop,
```

```
  by = NULL,
  with_ties = TRUE,
  na_rm = TRUE
)

## S3 method for class 'tbl_lazy'
slice_max(
  .data,
  order_by,
  ...,
  n,
  by = NULL,
  prop,
  with_ties = TRUE,
  na_rm = TRUE
)

## S3 method for class 'tbl_lazy'
slice_sample(.data, ..., n, prop, by = NULL, weight_by = NULL, replace = FALSE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`order_by`	Variable or function of variables to order by.
`...`	Not used.
`n, prop`	Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. If the proportion of a group size is not an integer, it is rounded down.
`by`	<tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`with_ties`	Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use FALSE to ignore ties, and return the first n rows.
`na_rm`	Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.
`weight_by, replace`	Not supported for database backends.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:3, y = c(1, 1, 2))
db |> slice_min(x) |> show_query()
```

```
db |> slice_max(x) |> show_query()
db |> slice_sample() |> show_query()

db |> group_by(y) |> slice_min(x) |> show_query()

# By default, ties are includes so you may get more rows
# than you expect
db |> slice_min(y, n = 1)
db |> slice_min(y, n = 1, with_ties = FALSE)

# Non-integer group sizes are rounded down
db |> slice_min(x, prop = 0.5)
```

---

dbplyr_uncount *"Uncount" a database table*

Description

This is a method for the tidyr `uncount()` generic. It uses a temporary table, so your database user needs permissions to create one.

Usage

```
dbplyr_uncount(data, weights, .remove = TRUE, .id = NULL)
```

Arguments

`data`	A lazy data frame backed by a database query.
`weights`	A vector of weights. Evaluated in the context of `data`; supports quasiquotation.
`.remove`	If `TRUE`, and `weights` is the name of a column in `data`, then this column is removed.
`.id`	Supply a string to create a new variable which gives a unique identifier for each created row.

Examples

```
df <- memdb_frame(x = c("a", "b"), n = c(1, 2))
dbplyr_uncount(df, n)
dbplyr_uncount(df, n, .id = "id")

# You can also use constants
dbplyr_uncount(df, 2)

# Or expressions
dbplyr_uncount(df, 2 / n)
```

distinct.tbl_lazy *Subset distinct/unique rows*

Description

This is a method for the dplyr `distinct()` generic. It adds the `DISTINCT` clause to the SQL query.

Usage

```
## S3 method for class 'tbl_lazy'
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.keep_all`	If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = c(1, 1, 2, 2), y = c(1, 2, 1, 1))
db |> distinct() |> show_query()
db |> distinct(x) |> show_query()
```

do.tbl_sql *Perform arbitrary computation on remote backend*

Description

[Deprecated]

`do()` is deprecated. Instead of `do()` you should use `collect()` and then your favourite combination of purrr and dplyr functions.

Usage

```
## S3 method for class 'tbl_sql'
do(.data, ..., .chunk_size = 10000L)
```

Arguments

.data	a tbl
...	Expressions to apply to each group. If named, results will be stored in a new column. If unnamed, must return a data frame. You can use . to refer to the current group. You can not mix named and unnamed arguments.
.chunk_size	The size of each chunk to pull into R. If this number is too big, the process will be slow because R has to allocate and free a lot of memory. If it's too small, it will be slow, because of the overhead of talking to the database.

---

escape *Escape/quote a value*

Description

escape() turns R values into SQL literals. It implements double dispatch via two sets of generics: first escape() dispatches on the class of x, then that method calls sql_escape_ident(), sql_escape_logical(), etc, which dispatch on con.

These generics translate individual values into SQL. The core generics are DBI::dbQuoteIdentifier() and DBI::dbQuoteString() for quoting identifiers and strings, but dbplyr needs additional tools for inserting logical, date, date-time, and raw values into queries.

Usage

```
escape(x, parens = NA, collapse = " ", con = NULL)

sql_escape_ident(con, x)

sql_escape_logical(con, x)

sql_escape_date(con, x)

sql_escape_datetime(con, x)

sql_escape_string(con, x)

sql_escape_raw(con, x)

sql_vector(x, parens = NA, collapse = " ", con = NULL)
```

Arguments

x	An object to escape. Existing sql vectors will be left as is, character vectors are escaped with single quotes, numeric vectors have trailing .0 added if they're whole numbers, identifiers are escaped with double quotes.

parens, collapse
: Controls behaviour when multiple values are supplied. parens should be a logical flag, or if NA, will wrap in parens if length > 1.

  Default behaviour: lists are always wrapped in parens and separated by commas, identifiers are separated by commas and never wrapped, atomic vectors are separated by spaces and wrapped in parens if needed.

con
: A sql_dialect object or database connection. Connections are supported for backward compatibility.

Value

A sql vector.

See Also

Other generic: db-sql, db_connection_describe(), db_copy_to()

Examples

```
con <- dialect_ansi()

# Doubles vs. integers
escape(1:5, con = con)
escape(c(1, 5.4), con = con)

# String vs known sql vs. sql identifier
escape("X", con = con)
escape(sql("X"), con = con)
escape(ident("X"), con = con)

# Escaping is idempotent
escape("X", con = con)
escape(escape("X", con = con), con = con)

# Database specific generics
sql_escape_logical(con, c(TRUE, FALSE, NA))
sql_escape_date(con, Sys.Date())
sql_escape_date(con, Sys.time())
sql_escape_raw(con, charToRaw("hi"))
```

---

expand.tbl_lazy *Expand SQL tables to include all possible combinations of values*

Description

This is a method for the tidyr::expand generics. It doesn't sort the result explicitly, so the order might be different to what expand() returns for data frames.

Usage

```
## S3 method for class 'tbl_lazy'
expand(data, ..., .name_repair = "check_unique")
```

Arguments

`data`	A lazy data frame backed by a database query.
`...`	Specification of columns to expand. See tidyr::expand for more details.
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic • `"unique_quiet"`: Same as `"unique"`, but "quiet" • `"universal_quiet"`: Same as `"universal"`, but "quiet" • a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R). • A purrr-style anonymous function, see `rlang::as_function()` This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
fruits <- memdb_frame(
  type   = c("apple", "orange", "apple", "orange", "orange", "orange"),
  year   = c(2010, 2010, 2012, 2010, 2010, 2012),
  size = c("XS", "S",  "M", "S", "S", "M"),
  weights = rnorm(6)
)

# All possible combinations ---------------------------------------
fruits |> tidyr::expand(type)
fruits |> tidyr::expand(type, size)

# Only combinations that already appear in the data ---------------
fruits |> tidyr::expand(nesting(type, size))
```

`fill.tbl_lazy` *Fill in missing values with previous or next value*

Description

Fill in missing values with previous or next value

Usage

```
## S3 method for class 'tbl_lazy'
fill(.data, ..., .direction = c("down", "up", "updown", "downup"))
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	Columns to fill.
`.direction`	Direction in which to fill missing values. Currently either "down" (the default) or "up". Note that "up" does not work when `.data` is sorted by non-numeric columns. As a workaround revert the order yourself beforehand; for example replace `arrange(x, desc(y))` by `arrange(desc(x), y)`.

Examples

```
library(dplyr, warn.conflicts = FALSE)

squirrels <- tibble::tribble(
  ~group,    ~name,     ~role,     ~n_squirrels, ~ n_squirrels2,
  1,      "Sam",    "Observer",   NA,                1,
  1,     "Mara", "Scorekeeper",    8,               NA,
  1,    "Jesse",    "Observer",   NA,               NA,
  1,      "Tom",    "Observer",   NA,                4,
  2,     "Mike",    "Observer",   NA,               NA,
  2,  "Rachael",    "Observer",   NA,                6,
  2,  "Sydekea", "Scorekeeper",   14,               NA,
  2, "Gabriela",    "Observer",   NA,               NA,
  3,  "Derrick",    "Observer",   NA,               NA,
  3,     "Kara", "Scorekeeper",    9,                10,
  3,    "Emily",    "Observer",   NA,               NA,
  3, "Danielle",    "Observer",   NA,               NA
)
squirrels$id <- 1:12

squirrels_db <- copy_to(memdb(), squirrels)
squirrels_db |>
  window_order(id) |>
  tidyr::fill(n_squirrels, n_squirrels2)
```

`filter.tbl_lazy` *Keep or drop rows that match a condition*

Description

These are methods for the dplyr `dplyr::filter()` and `dplyr::filter_out()` generics. They generate the `WHERE` clause of the SQL query.

`filter()` is translated directly to `WHERE`, which already matches dplyr's behaviour of treating `NA` like `FALSE` (SQL's three-valued logic drops `NULL` rows from `WHERE`).

`filter_out()` requires an additional step, where the combined condition is wrapped in `is_distinct_from(., TRUE)`, which is then translated using the backend (e.g. to `IS DISTINCT FROM` on PostgreSQL, `IS NOT` on SQLite). This ensures that the SQL translation matches dplyr's semantics.

Usage

```
## S3 method for class 'tbl_lazy'
filter(.data, ..., .by = NULL, .preserve = FALSE)

## S3 method for class 'tbl_lazy'
filter_out(.data, ..., .by = NULL, .preserve = FALSE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.preserve`	Not supported by this method.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = c(2, NA, 5, NA, 10), y = 1:5)
db |> filter(x < 5) |> show_query()
db |> filter_out(x < 5) |> show_query()
db |> filter(is.na(x)) |> show_query()
```

get_returned_rows *Extract and check the* RETURNING *rows*

Description

[Experimental]

`get_returned_rows()` extracts the RETURNING rows produced by `rows_insert()`, `rows_append()`, `rows_update()`, `rows_upsert()`, or `rows_delete()` if these are called with the `returning` argument. An error is raised if this information is not available.

`has_returned_rows()` checks if `x` has stored RETURNING rows produced by `rows_insert()`, `rows_append()`, `rows_update()`, `rows_upsert()`, or `rows_delete()`.

Usage

```
get_returned_rows(x)

has_returned_rows(x)
```

Arguments

x	A lazy tbl.

Value

For `get_returned_rows()`, a tibble.

For `has_returned_rows()`, a scalar logical.

Examples

```
library(dplyr)

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbExecute(con, "CREATE TABLE Info (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   number INTEGER
)")
info <- tbl(con, "Info")

rows1 <- copy_inline(con, data.frame(number = c(1, 5)))
rows_insert(info, rows1, conflict = "ignore", in_place = TRUE)
info

# If the table has an auto incrementing primary key, you can use
# the returning argument + `get_returned_rows()` its value
rows2 <- copy_inline(con, data.frame(number = c(13, 27)))
info <- rows_insert(
  info,
  rows2,
```

```
  conflict = "ignore",
  in_place = TRUE,
  returning = id
)
info
get_returned_rows(info)
```

`group_by.tbl_lazy` *Group by one or more variables*

Description

This is a method for the dplyr `group_by()` generic. It is translated to the `GROUP BY` clause of the SQL query when used with `summarise()` and to the `PARTITION BY` clause of window functions when used with `mutate()`.

Usage

```
## S3 method for class 'tbl_lazy'
group_by(.data, ..., .add = FALSE, .drop = TRUE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<data-masking> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`.
`.drop`	Not supported by this method.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db |>
  group_by(g) |>
  summarise(n()) |>
  show_query()

db |>
  group_by(g) |>
  mutate(x2 = x / sum(x, na.rm = TRUE)) |>
  show_query()
```

`head.tbl_lazy` *Subset the first rows*

Description

This is a method for the `head()` generic. It is usually translated to the `LIMIT` clause of the SQL query. Because `LIMIT` is not an official part of the SQL specification, some database use other clauses like `TOP` or `FETCH ROWS`.

Note that databases don't really have a sense of row order, so what "first" means is subject to interpretation. Most databases will respect ordering performed with `arrange()`, but it's not guaranteed. `tail()` is not supported at all because the situation is even murkier for the "last" rows. Additionally, `LIMIT` clauses can not generally appear in subqueries, which means that you should use `head()` as late as possible in your pipelines.

Usage

```
## S3 method for class 'tbl_lazy'
head(x, n = 6L, ...)
```

Arguments

`x`	A lazy data frame backed by a database query.
`n`	Number of rows to return
`...`	Not used.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:100)
db |> head() |> show_query()

# Pretend we have data in a SQL server database
db2 <- lazy_frame(x = 1:100, con = dialect_mssql())
db2 |> head() |> show_query()
```

`intersect.tbl_lazy` *SQL set operations*

Description

These are methods for the dplyr generics `dplyr::intersect()`, `dplyr::union()`, and `dplyr::setdiff()`. They are translated to `INTERSECT`, `UNION`, and `EXCEPT` respectively.

Usage

```
## S3 method for class 'tbl_lazy'
intersect(x, y, copy = "none", ..., all = FALSE)

## S3 method for class 'tbl_lazy'
union(x, y, copy = "none", ..., all = FALSE)

## S3 method for class 'tbl_lazy'
union_all(x, y, copy = "none", ...)

## S3 method for class 'tbl_lazy'
setdiff(x, y, copy = "none", ..., all = FALSE)
```

Arguments

`x`, `y`	A pair of lazy data frames backed by database queries.
`copy`	If `x` and `y` are not from the same data source, `copy` controls how `y` is copied into the same source as `x`. There are three options: • `"none"`, the default, will error if `y` needs to be copied. This ensures that you don't accidentally copy large datasets from R to the database. • `"temp-table"`: copies `y` into a temporary table in the same database as `x`. `*_join()` will automatically run `ANALYZE` on the created table in the hope that this will make your queries as efficient as possible by giving more data to the query planner. • `"inline"`: `y` will be inlined into the query using `copy_inline()`. This should be faster for small datasets and doesn't require write access. `TRUE` (`"temp-table"`) and `FALSE` (`"none"`) are also accepted for backward compatibility.
`...`	Must be empty.
`all`	If `TRUE`, includes all matches in output, not just unique rows.

`join.tbl_sql` *Join SQL tables*

Description

These are methods for the dplyr join generics. They are translated to the following SQL queries:

- `inner_join(x, y)`: `SELECT * FROM x JOIN y ON x.a = y.a`
- `left_join(x, y)`: `SELECT * FROM x LEFT JOIN y ON x.a = y.a`
- `right_join(x, y)`: `SELECT * FROM x RIGHT JOIN y ON x.a = y.a`
- `full_join(x, y)`: `SELECT * FROM x FULL JOIN y ON x.a = y.a`
- `semi_join(x, y)`: `SELECT * FROM x WHERE EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`
- `anti_join(x, y)`: `SELECT * FROM x WHERE NOT EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`

Usage

```
## S3 method for class 'tbl_lazy'
inner_join(
  x,
  y,
  by = NULL,
  copy = "none",
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
left_join(
  x,
  y,
  by = NULL,
  copy = "none",
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
```

```
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
right_join(
  x,
  y,
  by = NULL,
  copy = "none",
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
full_join(
  x,
  y,
  by = NULL,
  copy = "none",
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
cross_join(
  x,
```

```
  y,
  ...,
  copy = "none",
  suffix = c(".x", ".y"),
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
semi_join(
  x,
  y,
  by = NULL,
  copy = "none",
  ...,
  na_matches = c("never", "na"),
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
anti_join(
  x,
  y,
  by = NULL,
  copy = "none",
  ...,
  na_matches = c("never", "na"),
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)
```

Arguments

x, y	A pair of lazy data frames backed by database queries.
by	A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and

x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

copy — If x and y are not from the same data source, copy controls how y is copied into the same source as x. There are three options:

- "none", the default, will error if y needs to be copied. This ensures that you don't accidentally copy large datasets from R to the database.
- "temp-table": copies y into a temporary table in the same database as x. *_join() will automatically run ANALYZE on the created table in the hope that this will make your queries as efficient as possible by giving more data to the query planner.
- "inline": y will be inlined into the query using copy_inline(). This should be faster for small datasets and doesn't require write access.

TRUE ("temp-table") and FALSE ("none") are also accepted for backward compatibility.

suffix — If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

... — Other parameters passed onto methods.

keep — Should the join keys from both x and y be preserved in the output?

- If NULL, the default, joins on equality retain only the keys from x, while joins on inequality retain the keys from both inputs.
- If TRUE, all keys from both inputs are retained.
- If FALSE, only keys from x are retained. For right and full joins, the data in key columns corresponding to rows that only exist in y are merged into the key columns from x. Can't be used when joining on inequality conditions.

na_matches — Should NA (NULL) values match one another? The default, "never", is how databases usually work. "na" makes the joins behave like the dplyr join functions, merge(), base::match(), and %in%.

multiple, unmatched — Unsupported in database backends. As a workaround for multiple use a unique key and for unmatched a foreign key constraint.

relationship — Unsupported in database backends.

sql_on — A custom join predicate as an SQL expression. Usually joins use column equality, but you can perform more complex queries by supplying sql_on which should be a SQL expression that uses LHS and RHS aliases to refer to the left-hand side or right-hand side of the join respectively.

auto_index — if copy is TRUE, automatically create indices for the variables in by. This may speed up the join if there are matching indexes in x.

x_as, y_as — Alias to use for x resp. y. Defaults to "LHS" resp. "RHS"

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

band_db <- copy_to(memdb(), dplyr::band_members)
instrument_db <- copy_to(memdb(), dplyr::band_instruments)
band_db |> left_join(instrument_db) |> show_query()

# Can join with local data frames by setting copy = TRUE
band_db |>
  left_join(dplyr::band_instruments, copy = TRUE)

# Unlike R, joins in SQL don't usually match NAs (NULLs)
db <- memdb_frame(x = c(1, 2, NA))
label <- memdb_frame(x = c(1, NA), label = c("one", "missing"))
db |> left_join(label, by = "x")
# But you can activate R's usual behaviour with the na_matches argument
db |> left_join(label, by = "x", na_matches = "na")

# By default, joins are equijoins, but you can use `sql_on` to
# express richer relationships
db1 <- memdb_frame(x = 1:5)
db2 <- memdb_frame(x = 1:3, y = letters[1:3])
db1 |> left_join(db2) |> show_query()
db1 |> left_join(db2, sql_on = "LHS.x < RHS.x") |> show_query()
```

---

last_sql *Retrieve the last SQL query generated*

Description

This is a helper function that retrieves the most recent SQL query generated by dbplyr, which can be useful for debugging.

Usage

```
last_sql()
```

Value

A SQL string, or `NULL` if no query has been generated yet.

Examples

```
library(dplyr, warn.conflicts = FALSE)

df <- lazy_frame(x = 1:3)
df |> filter(x > 1)

last_sql()
```

---

memdb *A temporary in-memory database*

Description

`memdb()` creates a temporary in-memory database that disappears when the R session ends. It's a convenient way to learn about and experiment with dbplyr without having to connect to a "real" database.

`memdb_frame()` works like `tibble::tibble()`, but instead of creating a new data frame in R, it creates a table in `memdb()`. `local_memdb_frame()` is like `memdb_frame()` but the table will be automatically deleted when the current scope ends. It's useful for tests. But beware: this function will overwrite an existing table of the same name.

Usage

```
memdb()

memdb_frame(.name = unique_table_name(), ...)

local_memdb_frame(.name = unique_table_name(), ..., frame = caller_env())
```

Arguments

`.name`	Name of table in database: defaults to a random name that's unlikely to conflict with an existing table.
`...`	<dynamic-dots> A set of name-value pairs. These arguments are processed with `rlang::quos()` and support unquote via `rlang::!!` and unquote-splice via `rlang::!!!`. Use `:=` to create columns that start with a dot. Arguments are evaluated sequentially. You can refer to previously created elements directly or using the rlang::.data pronoun. To refer explicitly to objects in the calling environment, use `rlang::!!` or rlang::.env, e.g. `!!.data` or `.env$.data` for the special case of an object named `.data`.
`frame`	The created table is bound to this execution frame and will be deleted when it ends. For expert use only.

Examples

```
library(dplyr)

# use memdb_frame() to create a new database table
df <- memdb_frame(x = runif(100), y = runif(100))
df |> arrange(x)
df |> arrange(x) |> show_query()

# Use memdb() + copy_to() to copy an existing data frame
iris_db <- copy_to(memdb(), iris)
iris_db
```

`mutate.tbl_lazy` *Create, modify, and delete columns*

Description

These are methods for the dplyr `mutate()` and `transmute()` generics. They are translated to computed expressions in the `SELECT` clause of the SQL query.

Usage

```
## S3 method for class 'tbl_lazy'
mutate(
  .data,
  ...,
  .by = NULL,
  .order = NULL,
  .frame = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.order`	<`data-masking`> A selection of columns to control ordering for window functions within this `mutate()` call. Use `c()` to order by multiple columns, e.g. `.order = c(x, y)`. Each column can be wrapped in `desc()` to specify descending order. Equivalent to calling `window_order()` before and clearing it after the `mutate()`.

`.frame` A length-2 numeric vector specifying the bounds for window function frames. The first element is the lower bound (use `-Inf` for "unbounded preceding") and the second is the upper bound (use `Inf` for "unbounded following", `0` for "current row"). Equivalent to calling `window_frame()` before and clearing it after the `mutate()`.

`.keep` Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept.

- `"all"` retains all columns from `.data`. This is the default.
- `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side.
- `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them.
- `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept.

`.before, .after` <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:5, y = 5:1)
db |>
  mutate(a = (x + y) / 2, b = sqrt(x^2L + y^2L)) |>
  show_query()

# dbplyr automatically creates subqueries as needed
db |>
  mutate(x1 = x + 1, x2 = x1 * 2) |>
  show_query()

# `.order` and `.frame` control window functions
db <- memdb_frame(g = c(1, 1, 2, 2, 2), x = c(5, 3, 1, 4, 2))
db |>
  mutate(rolling_sum = sum(x), .by = g, .order = x, .frame = c(-2, 2)) |>
  show_query()
```

`pivot_longer.tbl_lazy` *Pivot data from wide to long*

Description

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `tidyr::pivot_wider()`.

Learn more in `vignette("pivot", "tidyr")`.

While most functionality is identical there are some differences to `pivot_longer()` on local data frames:

- the output is sorted differently/not explicitly,
- the coercion of mixed column types is left to the database,
- `values_ptypes` NOT supported.

Note that `build_longer_spec()` and `pivot_longer_spec()` do not work with remote tables.

Usage

```
## S3 method for class 'tbl_lazy'
pivot_longer(
  data,
  cols,
  ...,
  cols_vary,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = NULL,
  names_transform = NULL,
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes,
  values_transform = NULL
)
```

Arguments

`data`	A data frame to pivot.
`cols`	Columns to pivot into longer format.
`...`	Additional arguments passed on to methods.
`cols_vary`	Unsupported; included for compatibility with the generic.

names_to	A string specifying the name of the column to create from the data stored in the column names of data.
names_prefix	A regular expression used to remove matching text from the start of each variable name.
names_sep, names_pattern	If names_to contains multiple values, these arguments control how the column name is broken up.
names_ptypes	A list of column name-prototype pairs.
names_transform, values_transform	A list of column name-function pairs.
names_repair	What happens if the output has invalid column names?
values_to	A string specifying the name of the column to create from the data stored in cell values. If names_to is a character containing the special .value sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.
values_drop_na	If TRUE, will drop rows that contain only NAs in the value_to column.
values_ptypes	Not supported.

Details

The SQL translation basically works as follows:

1. split the specification by its key columns i.e. by variables crammed into the column names.
2. for each part in the split specification transmute() data into the following columns

- id columns i.e. columns that are not pivotted
- key columns
- value columns i.e. columns that are pivotted

1. combine all the parts with union_all()

Examples

```
# See vignette("pivot") for examples and explanation

# Simplest case where column names are character data
memdb_frame(
  id = c("a", "b"),
  x = 1:2,
  y = 3:4
) |>
  tidyr::pivot_longer(-id)
```

`pivot_wider.tbl_lazy` *Pivot data from long to wide*

Description

`pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`. Learn more in `vignette("pivot", "tidyr")`.

`pivot_wider()` on database tables comes with some caveats, please make sure to read below for details.

Usage

```
## S3 method for class 'tbl_lazy'
pivot_wider(
  data,
  ...,
  id_cols = NULL,
  id_expand = FALSE,
  names_from = name,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_vary = "fastest",
  names_expand = FALSE,
  names_repair = "check_unique",
  values_from = value,
  values_fill = NULL,
  values_fn = ~max(.x, na.rm = TRUE),
  unused_fn = NULL
)

dbplyr_pivot_wider_spec(
  data,
  spec,
  ...,
  names_repair = "check_unique",
  id_cols = NULL,
  id_expand = FALSE,
  values_fill = NULL,
  values_fn = ~max(.x, na.rm = TRUE),
  unused_fn = NULL,
  error_call = current_env()
)
```

Arguments

data	A lazy data frame backed by a database query.
...	Unused; included for compatibility with generic.
id_cols	A set of columns that uniquely identifies each observation.
id_expand	Unused; included for compatibility with the generic.
names_from, values_from	A pair of arguments describing which column (or columns) to get the name of the output column (names_from), and which column (or columns) to get the cell values from (values_from). If values_from contains multiple values, the value will be added to the front of the output column.
names_prefix	String added to the start of every variable name.
names_sep	If names_from or values_from contains multiple variables, this will be used to join their values together into a single string to use as a column name.
names_glue	Instead of names_sep and names_prefix, you can supply a glue specification that uses the names_from columns (and special .value) to create custom column names.
names_sort	Should the column names be sorted? If FALSE, the default, column names are ordered by first appearance.
names_vary	When names_from identifies a column (or columns) with multiple unique values, and multiple values_from columns are provided, in what order should the resulting column names be combined? • "fastest" varies names_from values fastest, resulting in a column naming scheme of the form: value1_name1, value1_name2, value2_name1, value2_name2. This is the default. • "slowest" varies names_from values slowest, resulting in a column naming scheme of the form: value1_name1, value2_name1, value1_name2, value2_name2.
names_expand	Should the values in the names_from columns be expanded by expand() before pivoting? This results in more columns, the output will contain column names corresponding to a complete expansion of all possible values in names_from. Additionally, the column names will be sorted, identical to what names_sort would produce.
names_repair	What happens if the output has invalid column names?
values_fill	Optionally, a (scalar) value that specifies what each value should be filled in with when missing. Be careful when using this in combination with the default values_fn.
values_fn	A function, the default is max(), applied to the value in each cell in the output. In contrast to local data frames it must not be NULL.
unused_fn	Optionally, a function applied to summarize the values from the unused columns (i.e. columns not identified by id_cols, names_from, or values_from). The default drops all unused columns from the result. This can be a named list if you want to apply different aggregations to different unused columns.

id_cols must be supplied for unused_fn to be useful, since otherwise all unspecified columns will be considered id_cols.

This is similar to grouping by the id_cols then summarizing the unused columns using unused_fn.

spec — A specification data frame. This is useful for more complex pivots because it gives you greater control on how metadata stored in the columns become column names in the result.

Must be a data frame containing character .name and .value columns. Additional columns in spec should be named to match columns in the long format of the dataset and contain values corresponding to columns pivoted from the wide format. The special .seq variable is used to disambiguate rows internally; it is automatically removed after pivoting.

error_call — The execution environment of a currently running function, e.g. caller_env(). The function will be mentioned in error messages as the source of the error. See the call argument of abort() for more information.

Caveats

pivot_wider() is eager:

Note that pivot_wider() cannot be lazy because we need to look at the data to figure out what the new column names will be. If you have a long-running query you have two options:

- Temporarily store the result of the query via compute().
- Create a spec before and use dbplyr_pivot_wider_spec() - dbplyr's version of tidyr::pivot_wider_spec().

You must supply values_fn:

The big difference to pivot_wider() for local data frames is that values_fn must not be NULL. By default it is max() which yields the same results as for local data frames if three conditions are true:

1. The combination of id_cols and value uniquely identify an observation.
2. The column has a comparable type (e.g. numeric, date-time, or (for most databases) string).
3. values_fill is NULL.

If either the second or third condition is not met, you must supply a custom values_fn. Unfortunately there is no generally available alternative and you'll need to look for something database specific, like FIRST() or ANY_VALUE().

How does it work?

The translation to SQL code basically works as follows:

1. Get unique keys in names_from column.
2. For each key value generate an expression of the form:

```
values_fn(
  CASE WHEN (`names from column` == `key value`) THEN (`value column`) END
) AS `output column`
```

3. Group data by id columns.
4. Summarise the grouped data with the expressions from step 2.

Examples

```
memdb_frame(
  id = 1,
  key = c("x", "y"),
  value = 1:2
) |>
  tidyr::pivot_wider(
    id_cols = id,
    names_from = key,
    values_from = value
  )
```

`pull.tbl_sql` *Extract a single column*

Description

This is a method for the dplyr `pull()` generic. It evaluates the query retrieving just the specified column.

Usage

```
## S3 method for class 'tbl_sql'
pull(.data, var = -1, name = NULL, ...)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`var`	A variable specified as: • a literal variable name • a positive integer, giving the position counting from the left • a negative integer, giving the position counting from the right. The default returns the last column (on the assumption that's the column you've created most recently). This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).
`name`	An optional parameter that specifies the column to be used as names for a named vector. Specified in a similar manner as `var`.
`...`	<data-masking> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.

Value

A vector of data.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:5, y = 5:1)
db |>
  mutate(z = x + y * 2) |>
  pull()
```

`remote_name` *Metadata about a remote table*

Description

`remote_name()` gives the unescaped name of the remote table, or `NULL` if it is a query (created by `sql()`) or already escape (created by `ident_q()`). `remote_table()` gives the remote table or the query. `remote_query()` gives the text of the query, and `remote_query_plan()` the query plan (as computed by the remote database). `remote_src()` and `remote_con()` give the dplyr source and DBI connection respectively.

Usage

```
remote_name(x, null_if_local = TRUE)

remote_table(x, null_if_local = TRUE)

remote_src(x)

remote_con(x)

remote_query(x, sql_options = NULL, cte = deprecated())

remote_query_plan(x, ...)
```

Arguments

`x`	Remote table, currently must be a `tbl_sql`.
`null_if_local`	Return `NULL` if the remote table is created via `tbl_lazy()` or `lazy_frame()`?
`sql_options`	SQL rendering options generated by `sql_options()`.
`cte`	**[Deprecated]** Use the `sql_options` argument instead.
`...`	Additional arguments passed on to methods.

Value

- `remote_name()` returns a string, or `NULL` if not a remote table, or not applicable. For example, computed queries do not have a "name".

- `remote_table()` returns a `sql()` object containing the (correctly quoted) table identifier, or the custom query if the table was created from `sql()`. Returns `NULL` if applied to a lazy query that no longer corresponds to a single table (e.g. after a `filter()` or a join), or if applied to a local lazy table.

Examples

```
mf <- memdb_frame(x = 1:5, y = 5:1, .name = "blorp")
remote_name(mf)
remote_src(mf)
remote_con(mf)
remote_query(mf)

mf2 <- dplyr::filter(mf, x > 3)
remote_name(mf2)
remote_src(mf2)
remote_con(mf2)
remote_query(mf2)
```

---

replace_na.tbl_lazy *Replace NAs with specified values*

Description

This is a method for the `tidyr::replace_na()` generic.

Usage

```
## S3 method for class 'tbl_lazy'
replace_na(data, replace = list(), ...)
```

Arguments

`data`	A pair of lazy data frame backed by database queries.
`replace`	A named list of values, with one value for each column that has NA values to be replaced.
`...`	Unused; included for compatibility with generic.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
df <- memdb_frame(x = c(1, 2, NA), y = c("a", NA, "b"))
df |> tidyr::replace_na(list(x = 0, y = "unknown"))
```

`rows_insert.tbl_lazy` *Edit individual rows in the underlying database table*

Description

These are methods for the dplyr `rows_insert()`, `rows_append()`, `rows_update()`, `rows_patch()`, `rows_upsert()`, and `rows_delete()` generics.

When `in_place = TRUE` these verbs do not generate `SELECT` queries, but instead directly modify the underlying data using `INSERT`, `UPDATE`, or `DELETE` operators. This will require that you have write access to the database: the connection needs permission to insert, modify or delete rows, but not to alter the structure of the table.

The default, `in_place = FALSE`, generates equivalent lazy tables (using `SELECT` queries) that allow previewing the result without actually modifying the underlying table on the database.

Usage

```
## S3 method for class 'tbl_lazy'
rows_insert(
  x,
  y,
  by = NULL,
  ...,
  conflict = c("error", "ignore"),
  copy = "none",
  in_place = FALSE,
  returning = NULL,
  method = NULL
)

## S3 method for class 'tbl_lazy'
rows_append(x, y, ..., copy = "none", in_place = FALSE, returning = NULL)

## S3 method for class 'tbl_lazy'
rows_update(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = "none",
  in_place = FALSE,
  returning = NULL
)

## S3 method for class 'tbl_lazy'
rows_patch(
```

```
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = "none",
  in_place = FALSE,
  returning = NULL
)

## S3 method for class 'tbl_lazy'
rows_upsert(
  x,
  y,
  by = NULL,
  ...,
  copy = "none",
  in_place = FALSE,
  returning = NULL,
  method = NULL
)

## S3 method for class 'tbl_lazy'
rows_delete(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = "none",
  in_place = FALSE,
  returning = NULL
)
```

Arguments

x	A lazy table. For `in_place = TRUE`, this must be a table instantiated with `tbl()` or `compute()`, not to a lazy query. The `remote_name()` function is used to determine the name of the table to be updated.
y	A lazy table, data frame, or data frame extensions (e.g. a tibble).
by	An unnamed character vector giving the key columns. The key columns must exist in both `x` and `y`. Keys typically uniquely identify each row, but this is only enforced for the key values of `y` when `rows_update()`, `rows_patch()`, or `rows_upsert()` are used. By default, we use the first column in `y`, since the first column is a reasonable place to put an identifier variable.
...	Other parameters passed onto methods.

conflict — For rows_insert(), how should keys in y that conflict with keys in x be handled? A conflict arises if there is a key in y that already exists in x.

One of:

- "error", the default, is not supported for database tables. To get the same behaviour add a unique index on the by columns and use rows_append().
- "ignore" will ignore rows in y with keys that conflict with keys in x.

copy — If x and y are not from the same data source, copy controls how y is copied into the same source as x. There are three options:

- "none", the default, will error if y needs to be copied. This ensures that you don't accidentally copy large datasets from R to the database.
- "temp-table": copies y into a temporary table in the same database as x. *_join() will automatically run ANALYZE on the created table in the hope that this will make your queries as efficient as possible by giving more data to the query planner.
- "inline": y will be inlined into the query using copy_inline(). This should be faster for small datasets and doesn't require write access.

TRUE ("temp-table") and FALSE ("none") are also accepted for backward compatibility.

in_place — Should x be modified in place? If FALSE will generate a SELECT query that returns the modified table; if TRUE will modify the underlying table using a DML operation (INSERT, UPDATE, DELETE or similar).

returning — Columns to return. See get_returned_rows() for details.

method — A string specifying the method to use. This is only relevant for in_place = TRUE.

unmatched — For rows_update(), rows_patch(), and rows_delete(), how should keys in y that are unmatched by the keys in x be handled?

One of:

- "error", the default, is not supported for database tables. Add a foreign key constraint on the by columns of y to let the database check this behaviour for you.
- "ignore" will ignore rows in y with keys that are unmatched by the keys in x.

Value

A new tbl_lazy of the modified data. With in_place = FALSE, the result is a lazy query that prints visibly, because the purpose of this operation is to preview the results. With in_place = TRUE, x is returned invisibly, because the purpose of this operation is the side effect of modifying rows in the table behind x.

Examples

```
library(dplyr)

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
```

```
DBI::dbExecute(con, "CREATE TABLE Ponies (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   name TEXT,
   cutie_mark TEXT
)")

ponies <- tbl(con, "Ponies")

applejack <- copy_inline(con, data.frame(
  name = "Apple Jack",
  cutie_mark = "three apples"
))

# The default behavior is to generate a SELECT query
rows_insert(ponies, applejack, conflict = "ignore")
# And the original table is left unchanged:
ponies

# You can also choose to modify the table with in_place = TRUE:
rows_insert(ponies, applejack, conflict = "ignore", in_place = TRUE)
# In this case `rows_insert()` returns nothing and the underlying
# data is modified
ponies
```

`select.tbl_lazy` *Subset, rename, and reorder columns using their names*

Description

These are methods for the dplyr `select()`, `rename()`, and `relocate()` generics. They generate the `SELECT` clause of the SQL query.

These functions do not support predicate functions, i.e. you can not use `where(is.numeric)` to select all numeric variables.

Usage

```
## S3 method for class 'tbl_lazy'
select(.data, ...)

## S3 method for class 'tbl_lazy'
rename(.data, ...)

## S3 method for class 'tbl_lazy'
rename_with(.data, .fn, .cols = everything(), ...)

## S3 method for class 'tbl_lazy'
relocate(.data, ..., .before = NULL, .after = NULL)
```

Arguments

.data	A lazy data frame backed by a database query.
...	<data-masking> Variables, or functions of variables. Use desc() to sort a variable in descending order.
.fn	A function used to transform the selected .cols. Should return a character vector the same length as the input.
.cols	<tidy-select> Columns to rename; defaults to all columns.
.before, .after	<tidy-select> Destination of columns selected by ... Supplying neither will move columns to the left-hand side; specifying both is an error.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1, y = 2, z = 3)
db |> select(-y) |> show_query()
db |> relocate(z) |> show_query()
db |> rename(first = x, last = z) |> show_query()
```

---

show_query *Show generated SQL and query plan*

Description

show_sql() displays the SQL query that will be dispatched to the database; explain() displays both the SQL query and the query plan.

Usage

```
## S3 method for class 'tbl_lazy'
show_query(x, ..., use_colour = TRUE, sql_options = NULL, cte = deprecated())

## S3 method for class 'tbl_sql'
explain(x, ...)
```

Arguments

x	An object to explain
...	For explain(), further arguments to remote_query_plan(). For show_query(), ignored.
use_colour	Should the output be coloured?
sql_options	SQL rendering options generated by sql_options().
cte	**[Deprecated]** Use the sql_options argument instead.

sql *Literal SQL escaping*

Description

Use `sql()` to declare that a string is literal SQL and should be used as is, without quoting.

Usage

```
sql(...)

is.sql(x)
```

Arguments

`...`	Character vectors that will be combined into a single SQL vector.
`x`	Object to check if it is an sql object.

Examples

```
library(dplyr, warn.conflicts = FALSE)
# sql() just adds a class
sql("x + 1")
is.sql(sql("x + 1"))

# You can use it when you need to insert some literal SQL in a query
db <- memdb_frame(x = 1:3)
db |> mutate(y = sql("CAST(x as VARCHAR)"))
```

sql_dialect *SQL dialects*

Description

The dialect system allows multiple database connection classes to share SQL generation code. A dialect object encapsulates the SQL syntax rules for a particular database, independent of the connection mechanism.

- `sql_dialect()` returns the dialect for a connection. For connections that haven't implemented a dialect method, returns the connection itself for backward compatibility.
- `new_sql_dialect()` creates a new dialect object. This is primarily intended for dbplyr backend authors.

Usage

```
sql_dialect(con)

new_sql_dialect(
  dialect,
  quote_identifier,
  has_window_clause = FALSE,
  has_table_alias_with_as = TRUE,
  has_star_table_prefix = FALSE
)
```

Arguments

con	A database connection.
dialect	A string giving the dialect name (e.g., "postgres", "mysql").
quote_identifier	A function that quotes identifiers. Should accept a character vector and return a sql vector.
has_window_clause	Does the backend support named window definitions (the WINDOW clause)?
has_table_alias_with_as	Does the backend support using AS when aliasing a table in a subquery?
has_star_table_prefix	Does the backend require table prefixes when selecting all columns in single-table queries (e.g., "table".* vs *)?

Value

- sql_dialect() returns a dialect object (class sql_dialect) or the connection itself for backward compatibility.
- new_sql_dialect() returns a dialect object with class c("sql_dialect_{name}", "sql_dialect").

Dispatching on dialect

For backward compatibility, all sql_ generics (and a handful of others) call sql_dialect() on the con argument in order to dispatch further on the dialect object, if possible:

```
sql_generic <- function(con, arg1, arg2, ...) {
  UseMethod("sql_generic", sql_dialect(con))
}
```

Unfortunately, due to the way that UseMethod() works, this uses sql_dialect(con) to control which method is selected, but still passes the original con to the method. This means that if you are implementing a method for a dialect and need to access dialect properties, you must call sql_dialect(con) again inside the method.

Examples

```
# Create a custom dialect
my_dialect <- new_sql_dialect(
  "custom",
  quote_identifier = function(x) sql_quote(x, "`"),
  has_window_clause = TRUE
)
class(my_dialect)
```

`sql_glue` *Build SQL strings with glue syntax*

Description

`sql_glue()` and `sql_glue2()` are designed to help dbplyr extenders generate custom SQL. They differ only in whether or not they require a connection. `sql_glue()` retrieves the ambient connection, making it suitable for use inside `sql_translation()` methods; `sql_glue2()` requires a connection, making it suitable for use inside all other `sql_` methods.

As the name suggests, these functions use **glue** syntax to make it easy to mix fixed SQL with varying user inputs. The default glue syntax, `{x}`, will escape `x` using the database connection. If there are multiple values in `x`, they'll be collapsed into a single string with `,`. If you want them to be wrapped in `()`, use a `*` suffix, e.g. `{x*}`.

You can also use type markers to control how the value is treated:

- `{.sql x}`: `x` is literal SQL that should be interpolated as is, without additional escaping. `x` must be a string.
- `{.tbl x}`: `x` is a table identifier like a string, `I()`, or one of the older forms like `DBI::Id()` or `in_schema()`.
- `{.id x}`: `x` is a generic identifier, e.g. for a column or index. `x` must be a character vector

Usage

```
sql_glue(sql, envir = parent.frame())

sql_glue2(con, sql, envir = parent.frame())
```

Arguments

`sql`	A string to interpolate.
`envir`	Environment to evaluate `sql` in.
`con`	A sql_dialect object or database connection. Connections are supported for backward compatibility.

Value

An SQL string.

Examples

```
con <- dialect_ansi()

tbl <- "my_table"
sql_glue2(con, "SELECT * FROM {.tbl tbl}")

# Values are properly escaped
name <- "Robert'); DROP TABLE students;--"
sql_glue2(con, "INSERT INTO students (name) VALUES ({name})")

# Control wrapping with *
x <- c("name", "age", "grade")
sql_glue2(con, "SELECT {.id x} FROM students")
sql_glue2(con, "SELECT * WHERE variable IN {x*}")
```

`sql_options` *Options for generating SQL*

Description

Customise SQL generation with these three options.

Usage

```
sql_options(cte = FALSE, use_star = TRUE, qualify_all_columns = FALSE)
```

Arguments

`cte`	If `FALSE`, the default, subqueries are used. If `TRUE` common table expressions are used.
`use_star`	If `TRUE`, the default, `*` is used to select all columns of a table. If `FALSE` all columns are explicitly selected.
`qualify_all_columns`	If `FALSE`, the default, columns are only qualified with the table they come from if the same column name appears in multiple tables.

Value

A <dbplyr_sql_options> object.

Examples

```
library(dplyr, warn.conflicts = FALSE)
lf1 <- lazy_frame(key = 1, a = 1, b = 2)
lf2 <- lazy_frame(key = 1, a = 1, c = 3)

result <- left_join(lf1, lf2, by = "key") |>
  filter(c >= 3)
```

```
show_query(result)
sql_options <- sql_options(cte = TRUE, qualify_all_columns = TRUE)
show_query(result, sql_options = sql_options)
```

`sql_query_insert` *Generate SQL for Insert, Update, Upsert, and Delete*

Description

These functions generate the SQL used in `rows_*(in_place = TRUE)`.

Usage

```
sql_query_insert(
  con,
  table,
  from,
  insert_cols,
  by,
  ...,
  conflict = c("error", "ignore"),
  returning_cols = NULL,
  method = NULL
)

sql_query_append(con, table, from, insert_cols, ..., returning_cols = NULL)

sql_query_update_from(
  con,
  table,
  from,
  by,
  update_values,
  ...,
  returning_cols = NULL
)

sql_query_upsert(
  con,
  table,
  from,
  by,
  update_cols,
  ...,
  returning_cols = NULL,
  method = NULL
```

```
)

sql_query_delete(con, table, from, by, ..., returning_cols = NULL)
```

Arguments

con	A sql_dialect object or database connection. Connections are supported for backward compatibility.
table	Table to update. Must be a table identifier. Use a string to refer to tables in the current schema/catalog or `I()` to refer to tables in other schemas/catalogs.
from	Table or query that contains the new data. Either a table identifier or SQL.
insert_cols	Names of columns to insert.
by	An unnamed character vector giving the key columns. The key columns must exist in both `x` and `y`. Keys typically uniquely identify each row, but this is only enforced for the key values of `y` when `rows_update()`, `rows_patch()`, or `rows_upsert()` are used. By default, we use the first column in `y`, since the first column is a reasonable place to put an identifier variable.
...	Other parameters passed onto methods.
conflict	For `rows_insert()`, how should keys in `y` that conflict with keys in `x` be handled? A conflict arises if there is a key in `y` that already exists in `x`. One of: • `"error"`, the default, will error if there are any keys in `y` that conflict with keys in `x`. • `"ignore"` will ignore rows in `y` with keys that conflict with keys in `x`.
returning_cols	Optional. Names of columns to return.
method	Optional. The method to use.
update_values	A named SQL vector that specify how to update the columns.
update_cols	Names of columns to update.

Details

Insert Methods

`"where_not_exists"`:

The default for most databases.

```
INSERT INTO x_name
SELECT *
FROM y
WHERE NOT EXISTS <match on by columns>
```

`"on_conflict"`:

Supported by:

- Postgres

- SQLite

This method uses the `ON CONFLICT` clause and therefore requires a unique index on the columns specified in `by`.

Upsert Methods

`"merge"`:

The upsert method according to the SQL standard. It uses the `MERGE` statement

```
MERGE INTO x_name
USING y
  ON <match on by columns>
WHEN MATCHED THEN
  UPDATE SET ...
WHEN NOT MATCHED THEN
  INSERT ...
```

`"on_conflict"`:

Supported by:

- Postgres
- SQLite

This method uses the `ON CONFLICT` clause and therefore requires a unique index on the columns specified in `by`.

`"cte_update"`:

Supported by:

- Postgres
- SQLite
- Oracle

The classical way to upsert in Postgres and SQLite before support for `ON CONFLICT` was added. The update is done in a CTE clause and the unmatched values are then inserted outside of the CTE.

Value

A SQL query.

Examples

```
sql_query_upsert(
  con = dialect_postgres(),
  table = "airlines",
  from = "df",
  by = "carrier",
  update_cols = "name"
)
```

sql_translation_agg *SQL helpers for aggregate functions*

Description

These functions help you create custom aggregate SQL translations when implementing a new backend. They are typically used within sql_translator() to define how R aggregate functions should be translated to SQL.

- sql_aggregate() creates a SQL aggregate function translator for functions with a single argument and an optional na.rm parameter (e.g., SUM(), AVG()).
- sql_aggregate_2() creates a SQL aggregate function translator for functions with exactly two arguments (e.g., CORR(), COVAR_SAMP()).
- sql_aggregate_n() creates a SQL aggregate function translator for functions with any number of arguments and an optional na.rm parameter (e.g., LEAST(), GREATEST()).
- sql_not_supported() creates a function that throws an informative error when a function is not supported in SQL.
- sql_check_na_rm() is a helper that you can use in aggregate functions to direct the user towards setting na.rm = TRUE.

Usage

```
sql_aggregate(f, f_r = f)

sql_aggregate_2(f)

sql_aggregate_n(f, f_r = f)

sql_check_na_rm(na.rm)

sql_not_supported(f)
```

Arguments

f	The name of the SQL function as a string.
f_r	The name of the R function being translated as a string.
na.rm	Logical indicating whether missing values should be removed. In SQL, missing values are always removed in aggregate functions, so this function will warn if na.rm is not TRUE.

See Also

Other SQL translation helpers: sql_translation_scalar, sql_translation_string, sql_translation_window, sql_variant()

sql_translation_scalar *SQL helpers for scalar functions*

Description

These functions help you create custom scalar SQL translations when implementing a new backend. They are typically used within `sql_translator()` to define how R functions should be translated to SQL.

- `sql_infix()` creates SQL infix operators like `+`, `-`, `*`, `/`.
- `sql_prefix()` creates SQL prefix functions like `ABS()`, `SQRT()`.
- `sql_cast()` creates SQL cast expressions like `CAST(x AS type)`.
- `sql_try_cast()` creates SQL try_cast expressions (for safe casting).
- `sql_log()` creates a SQL logarithm function with optional base.
- `sql_cot()` creates a SQL cotangent function (as `1 / TAN(x)`).
- `sql_runif()` creates a SQL expression for generating uniform random numbers.

Usage

```
sql_infix(f, pad = TRUE)

sql_prefix(f, n = NULL)

sql_cast(type)

sql_try_cast(type)

sql_log()

sql_cot()

sql_runif(rand_expr, n = n(), min = 0, max = 1)
```

Arguments

`f`	The name of the SQL function as a string.
`pad`	If `TRUE`, the default, pad the infix operator with spaces.
`n`	For `sql_prefix()`, an optional number of arguments to expect. Will signal error if not correct.
`type`	SQL type name as a string.
`rand_expr`	A string giving an SQL expression that generates a random number between 0 and 1, e.g. `"RANDOM()"`.
`min, max`	Range of random values.

See Also

Other SQL translation helpers: sql_translation_agg, sql_translation_string, sql_translation_window, sql_variant()

`sql_translation_string` *SQL helpers for string functions*

Description

These functions help you create custom string SQL translations when implementing a new backend. They are typically used within sql_translator() to define how R string functions should be translated to SQL.

- `sql_substr()` creates a SQL substring function translator that converts R's `substr(x, start, stop)` to SQL's `SUBSTR(x, start, length)`.
- `sql_str_sub()` creates a SQL substring function translator that handles stringr's `str_sub()` with support for negative indices.
- `sql_paste()` creates a SQL paste function using `CONCAT_WS()` or similar.
- `sql_paste_infix()` creates a SQL paste function using an infix operator like `||`.

Usage

```
sql_substr(f = "SUBSTR")

sql_str_sub(subset_f = "SUBSTR", length_f = "LENGTH", optional_length = TRUE)

sql_paste(default_sep, f = "CONCAT_WS")

sql_paste_infix(default_sep, op, cast = sql_cast("text"))
```

Arguments

`f`	The name of the SQL function as a string.
`subset_f`	The name of the SQL substring function.
`length_f`	The name of the SQL string length function.
`optional_length`	Whether the length argument is optional in the SQL substring function.
`default_sep`	The default separator for paste operations.
`op`	The SQL operator to use for infix paste operations.
`cast`	A function to cast values to strings.

See Also

Other SQL translation helpers: sql_translation_agg, sql_translation_scalar, sql_translation_window, sql_variant()

sql_translation_window *SQL helpers for window functions*

Description

These functions help you create custom window SQL translations when implementing a new backend. They are typically used within `sql_translator()` to define how R window functions should be translated to SQL.

- `win_over()` makes it easy to generate the window function specification.
- `win_absent()`, `win_rank()`, `win_aggregate()`, and `win_cumulative()` provide helpers for constructing common types of window functions.
- `win_current_group()` and `win_current_order()` allow you to access the grouping and order context set up by `group_by()` and `arrange()`.

Usage

```
win_over(
  expr,
  partition = NULL,
  order = NULL,
  frame = NULL,
  con = sql_current_con()
)

win_rank(f, empty_order = FALSE)

win_aggregate(f)

win_aggregate_2(f)

win_cumulative(f)

win_absent(f)

win_current_group()

win_current_order()

win_current_frame()
```

Arguments

`expr`	The window expression.
`partition`	Variables to partition over.

order	Variables to order by.
frame	A numeric vector of length two defining the frame.
con	A sql_dialect object or database connection. Connections are supported for backward compatibility.
f	The name of an SQL function as a string.
empty_order	A logical value indicating whether to order by NULL if order is not specified.

See Also

Other SQL translation helpers: sql_translation_agg, sql_translation_scalar, sql_translation_string, sql_variant()

Examples

```
con <- dialect_ansi()

win_over(sql("avg(x)"), con = con)
win_over(sql("avg(x)"), "y", con = con)
win_over(sql("avg(x)"), order = "y", con = con)
win_over(sql("avg(x)"), order = c("x", "y"), con = con)
win_over(sql("avg(x)"), frame = c(-Inf, 0), order = "y", con = con)
```

---

sql_variant *Create an SQL translator*

Description

sql_variant() creates a SQL variant, a list of translators for scalar, aggregate, and window functions. sql_translator() creates a translator, an environment containing R to SQL translations. When creating a backend, you'll use these functions to customize how R functions are converted to SQL.

Learn more in vignette("new-backend").

Usage

```
sql_variant(
  scalar = sql_translator(),
  aggregate = sql_translator(),
  window = sql_translator()
)

sql_translator(..., .funs = list(), .parent = new.env(parent = emptyenv()))

base_scalar

base_agg
```

base_win

base_no_win

base_odbc_scalar

base_odbc_agg

base_odbc_win

Arguments

scalar, aggregate, window
: The three families of functions that an SQL variant can supply.

..., .funs
: Named functions, used to add custom converters from standard R functions to SQL functions. Specify individually in ..., or provide a list of .funs.

.parent
: The SQL variant that this variant should inherit from. Defaults to base_agg which provides a standard set of mappings for the most common operators and functions.

Base translators

dbplyr provides the following base translators that implement standard SQL semantics:

- base_scalar - scalar functions and operators
- base_agg - aggregate functions
- base_win - window functions
- base_no_win - versions of window functions that throw errors

See Also

Other SQL translation helpers: sql_translation_agg, sql_translation_scalar, sql_translation_string, sql_translation_window

Examples

```
# An example of adding some mappings for the statistical functions that
# postgresql provides: http://bit.ly/K5EdTn
postgres_agg <- sql_translator(.parent = base_agg,
  cor = sql_aggregate_2("CORR"),
  cov = sql_aggregate_2("COVAR_SAMP"),
  sd =  sql_aggregate("STDDEV_SAMP", "sd"),
  var = sql_aggregate("VAR_SAMP", "var")
)

# Next we have to simulate a connection that uses this variant
con <- new_sql_dialect("test", quote = \(x) sql_quote(x, '"'))
sql_translation.sql_dialect_test <- function(x) {
```

```
  sql_variant(
    base_scalar,
    postgres_agg,
    base_no_win
  )
}

translate_sql(cor(x, y), con = con, window = FALSE)
translate_sql(sd(income / years, na.rm = TRUE), con = con, window = FALSE)
```

`summarise.tbl_lazy` *Summarise each group to one row*

Description

This is a method for the dplyr `summarise()` generic. It generates the `SELECT` clause of the SQL query, and generally needs to be combined with `group_by()`.

Usage

```
## S3 method for class 'tbl_lazy'
summarise(.data, ..., .by = NULL, .groups = NULL)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.groups`	**[Experimental]** Grouping structure of the result. • "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0. • "drop": All levels of grouping are dropped. • "keep": Same grouping structure as `.data`. When `.groups` is not specified, it defaults to "drop_last". In addition, a message informs you of that choice, unless the result is ungrouped, the option "dplyr.summarise.inform" is set to `FALSE`, or when `summarise()` is called from a function in a package.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db |>
  summarise(n()) |>
  show_query()

db |>
  group_by(g) |>
  summarise(n()) |>
  show_query()
```

---

tbl.src_dbi *Create a lazy query backed by a database*

Description

Use `tbl()` to create a SQL query backed by a database. Manipulating this object with dplyr verbs then builds up a SQL query that will only be executed when you explicitly ask for it, either by printing the object, calling `collect()` to bring the data back to R or calling `compute()` to create a new table in the database. You can see the query without executing it with `show_query()`.

Learn more in `vignette("dbplyr")`.

Usage

```
## S3 method for class 'src_dbi'
tbl(src, from, vars = NULL, ...)
```

Arguments

`src`	A `DBIConnection` object produced by `DBI::dbConnect()`.
`from`	Either a table identifier or a literal `sql()` string. Use a string to identify a table in the current schema/catalog or `I()` for a table elsewhere, e.g. `I("schema.table")` or `I("catalog.schema.table")`. For backward compatibility, you can also use `in_schema()`/`in_catalog()` or `DBI::Id()`.
`vars`	Optionally, provide a character vector of column names. If not supplied, will be retrieved from the database by running a simple query. This argument is mainly useful for better performance when creating many `tbl`s with known variables.
`...`	Passed on to `tbl_sql()`

Examples

```
library(dplyr)

# Connect to a temporary in-memory SQLite database and add some data
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
copy_to(con, mtcars)

# To retrieve a single table from a source, use `tbl()`
mtcars_db <- con |> tbl("mtcars")
mtcars_db

# Use `I()` for qualified table names
con |> tbl(I("temp.mtcars")) |> head(1)

# You can also pass raw SQL if you want a more sophisticated query
con |> tbl(sql("SELECT * FROM mtcars WHERE cyl = 8")) |> head(1)

# But in most cases, you'll rely on dbplyr to construct the SQL:
mtcars_db |>
  filter(vs == 1) |>
  summarise(mpg = mean(mpg, na.rm = TRUE), .by = cyl) |>
  show_query()
```

---

translate_sql *Translate an expression to SQL*

Description

dbplyr translates commonly used base functions including logical (`!`, `&`, `|`), arithmetic (`^`), and comparison (`!=`) operators, as well as common summary (`mean()`, `var()`), and transformation (`log()`) functions. All other functions will be preserved as is. R's infix functions (e.g. `%like%`) will be converted to their SQL equivalents (e.g. `LIKE`).

Learn more in `vignette("translation-function")`.

Usage

```
translate_sql(
  ...,
  con,
  vars_group = NULL,
  vars_order = NULL,
  vars_frame = NULL,
  window = TRUE
)

translate_sql_(
  dots,
```

```
  con,
  vars_group = NULL,
  vars_order = NULL,
  vars_frame = NULL,
  window = TRUE,
  context = list()
)
```

Arguments

..., dots	Expressions to translate. `translate_sql()` automatically quotes them for you. `translate_sql_()` expects a list of expression objects.
con	A sql_dialect object or database connection. Connections are supported for backward compatibility.
vars_group, vars_order, vars_frame	Parameters used in the `OVER` expression of windowed functions.
window	Use `FALSE` to suppress generation of the `OVER` statement used for window functions. This is necessary when generating SQL for a grouped summary.
context	Use to carry information for special translation cases. For example, MS SQL needs a different conversion for is.na() in WHERE vs. SELECT clauses. Expects a list.

Examples

```
con <- dialect_ansi()

# Regular maths is translated in a very straightforward way
translate_sql(x + 1, con = con)
translate_sql(sin(x) + tan(y), con = con)

# Note that all variable names are escaped
translate_sql(like == "x", con = con)
# In ANSI SQL: "" quotes variable _names_, '' quotes strings

# Logical operators are converted to their sql equivalents
translate_sql(x < 5 & !(y >= 5), con = con)
# xor() doesn't have a direct SQL equivalent
translate_sql(xor(x, y), con = con)

# If is translated into case when
translate_sql(if (x > 5) "big" else "small", con = con)

# Infix functions are passed onto SQL with % removed
translate_sql(first %like% "Had%", con = con)
translate_sql(first %is% NA, con = con)
translate_sql(first %in% c("John", "Roger", "Robert"), con = con)

# And be careful if you really want integers
translate_sql(x == 1, con = con)
translate_sql(x == 1L, con = con)
```

```
# If you have an already quoted object, use translate_sql_:
x <- quote(y + 1 / sin(t))
translate_sql_(list(x), con = dialect_ansi())

# Windowed translation ----------------------------------------
# Known window functions automatically get OVER()
translate_sql(mpg > mean(mpg), con = con)

# Suppress this with window = FALSE
translate_sql(mpg > mean(mpg), window = FALSE, con = con)

# vars_group controls partition:
translate_sql(mpg > mean(mpg), vars_group = "cyl", con = con)

# and vars_order controls ordering for those functions that need it
translate_sql(cumsum(mpg), con = con)
translate_sql(cumsum(mpg), vars_order = "mpg", con = con)
```

`window_order` *Override window order and frame*

Description

These allow you to override the `PARTITION BY` and `ORDER BY` clauses of window functions generated by grouped mutates.

Usage

```
window_order(.data, ...)

window_frame(.data, from = -Inf, to = Inf)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	Variables to order by
`from, to`	Bounds of the frame.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = rep(1:2, each = 5), y = runif(10), z = 1:10)
db |>
  window_order(y) |>
  mutate(z = cumsum(y)) |>
  show_query()
```

```
db |>
  group_by(g) |>
  window_frame(-3, 0) |>
  window_order(z) |>
  mutate(z = sum(y)) |>
  show_query()
```

`with_dialect` *Override the SQL dialect for a connection*

Description

`with_dialect()` overrides the default dialect assigned to a connection. This is useful when dbplyr guesses the dialect incorrectly, which is most likely to occur with ODBC/JDBC/ADBC backends.

Usage

```
with_dialect(con, dialect)
```

Arguments

`con`	A database connection (class `DBIConnection`).
`dialect`	A dialect object created by a `dialect_*()` function (e.g., `dialect_postgres()`, `dialect_sqlite()`).

Value

A connection object that uses the specified dialect for SQL generation.

Examples

```
# Wrap an in-memory SQLite connection to use Postgres dialect
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
pg_con <- with_dialect(con, dialect_postgres())

# SQL generation uses Postgres syntax
lf <- lazy_frame(x = 1, con = pg_con)
lf |> dplyr::mutate(y = sd(x))

DBI::dbDisconnect(con)
```

Index

∗ **SQL translation helpers**
    sql_translation_agg, 68
    sql_translation_scalar, 69
    sql_translation_string, 70
    sql_translation_window, 71
    sql_variant, 72
∗ **generic**
    escape, 31
.sql, 4
?dplyr_by, *28*, *35*, *46*, *74*
?join_by, *43*

abort(), *52*
add_count.tbl_lazy *(*count.tbl_lazy*)*, 26
anti_join.tbl_lazy *(*join.tbl_sql*)*, 40
arrange(), *5*, *28*, *71*
arrange.tbl_lazy, 5
arrange.tbl_lazy(), *26*

backend-access, 6
backend-adbc, 6
backend-ansi, 7
backend-db2, 8
backend-hana, 9
backend-hive, 9
backend-impala, 10
backend-jdbc, 11
backend-mssql, 12
backend-mysql, 13
backend-odbc, 14
backend-oracle, 15
backend-postgres, 16
backend-redshift, 16
backend-snowflake, 17
backend-spark-sql, 18
backend-sqlite, 18
backend-teradata, 19
base::match(), *43*
base_agg *(*sql_variant*)*, 72
base_no_win *(*sql_variant*)*, 72
base_odbc_agg *(*sql_variant*)*, 72
base_odbc_scalar *(*sql_variant*)*, 72
base_odbc_win *(*sql_variant*)*, 72
base_scalar *(*sql_variant*)*, 72
base_win *(*sql_variant*)*, 72
bind_queries, 20

collapse.tbl_sql, 20
collect(), *5*, *22*, *26*, *30*, *33*, *35*, *38*, *44*, *47*, *55*, *74*, *75*
collect.tbl_sql, 21
complete.tbl_lazy, 22
compute(), *57*, *75*
compute.tbl_sql, 23
copy_inline, 24
copy_inline(), *26*, *39*, *43*, *58*
copy_to(), *9*, *24*, *25*
copy_to.src_sql, 25
count.tbl_lazy, 26
cross_join(), *43*
cross_join.tbl_lazy *(*join.tbl_sql*)*, 40

db-quote *(*escape*)*, 31
db_connection_describe(), *32*
db_copy_to(), *32*
DBI::dbConnect(), *6*, *9*, *11*, *13–16*, *18*, *75*
DBI::dbQuoteIdentifier(), *31*
DBI::dbQuoteString(), *31*
DBI::dbWriteTable(), *25*
DBI::Id(), *75*
dbplyr-slice, 27
dbplyr_pivot_wider_spec *(*pivot_wider.tbl_lazy*)*, 50
dbplyr_uncount, 29
desc(), *5*, *27*, *30*, *35*, *37*, *46*, *53*, *60*, *74*
dialect_access *(*backend-access*)*, 6
dialect_ansi *(*backend-ansi*)*, 7
dialect_db2 *(*backend-db2*)*, 8
dialect_db2(), *11*
dialect_hana *(*backend-hana*)*, 9

dialect_hana(), *11*
dialect_hive *(backend-hive)*, 9
dialect_hive(), *11*
dialect_impala *(backend-impala)*, 10
dialect_impala(), *11*
dialect_mariadb *(backend-mysql)*, 13
dialect_mariadb(), *7*, *11*
dialect_mssql *(backend-mssql)*, 12
dialect_mssql(), *7*, *11*
dialect_mysql *(backend-mysql)*, 13
dialect_mysql(), *7*, *11*
dialect_odbc *(backend-odbc)*, 14
dialect_oracle *(backend-oracle)*, 15
dialect_oracle(), *11*
dialect_postgres *(backend-postgres)*, 16
dialect_postgres(), *6*, *11*, *79*
dialect_redshift *(backend-redshift)*, 16
dialect_redshift(), *7*, *11*
dialect_snowflake *(backend-snowflake)*, 17
dialect_snowflake(), *7*, *11*
dialect_spark_sql *(backend-spark-sql)*, 18
dialect_spark_sql(), *11*
dialect_sqlite *(backend-sqlite)*, 18
dialect_sqlite(), *7*, *11*, *79*
dialect_teradata *(backend-teradata)*, 19
dialect_teradata(), *11*
distinct(), *30*
distinct.tbl_lazy, 30
do.tbl_sql, 30
dplyr::bind_rows(), *20*
dplyr::count(), *26*
dplyr::filter(), *27*, *35*
dplyr::filter_out(), *35*
dynamic-dots, *20*

escape, 31
expand(), *51*
expand.tbl_lazy, 32
explain.tbl_sql *(show_query)*, 60

fill.tbl_lazy, 34
filter.tbl_lazy, 35
filter_out.tbl_lazy *(filter.tbl_lazy)*, 35
full_join.tbl_lazy *(join.tbl_sql)*, 40

get_returned_rows, 36
get_returned_rows(), *58*
group_by(), *28*, *35*, *37*, *46*, *71*, *74*
group_by.tbl_lazy, 37
group_by.tbl_lazy(), *26*

has_returned_rows *(get_returned_rows)*, 36
head(), *38*
head.tbl_lazy, 38

in_catalog(), *75*
in_schema(), *75*
inner_join.tbl_lazy *(join.tbl_sql)*, 40
intersect.tbl_lazy, 39
is.sql *(sql)*, 61

join, *40*
join.tbl_sql, 40
join_by(), *42*, *43*

last_sql, 44
lazy_frame(), *54*
left_join.tbl_lazy *(join.tbl_sql)*, 40
local_memdb_frame *(memdb)*, 45

memdb, 45
memdb_frame *(memdb)*, 45
merge(), *43*
mutate(), *37*, *46*, *47*
mutate.tbl_lazy, 46
mutate.tbl_lazy(), *5*

new_sql_dialect *(sql_dialect)*, 61

pivot_longer.tbl_lazy, 48
pivot_wider.tbl_lazy, 50
PostgreSQL backend, *16*
pull(), *53*
pull.tbl_sql, 53

quasiquotation, *53*

relocate(), *47*, *59*
relocate.tbl_lazy *(select.tbl_lazy)*, 59
remote_con *(remote_name)*, 54
remote_name, 54
remote_name(), *57*
remote_query *(remote_name)*, 54
remote_query_plan *(remote_name)*, 54
remote_query_plan(), *60*

remote_src (remote_name), 54
remote_table (remote_name), 54
rename(), *59*
rename.tbl_lazy (select.tbl_lazy), 59
rename_with.tbl_lazy (select.tbl_lazy), 59
replace_na.tbl_lazy, 55
right_join.tbl_lazy (join.tbl_sql), 40
rlang::.data, *45*
rlang::.env, *45*
rlang::as_function(), *33*
rlang::quos(), *45*
rows_append(), *36*, *56*
rows_append.tbl_lazy (rows_insert.tbl_lazy), 56
rows_delete(), *36*, *56*
rows_delete.tbl_lazy (rows_insert.tbl_lazy), 56
rows_insert(), *36*, *56*
rows_insert.tbl_lazy, 56
rows_patch(), *56*
rows_patch.tbl_lazy (rows_insert.tbl_lazy), 56
rows_update(), *36*, *56*
rows_update.tbl_lazy (rows_insert.tbl_lazy), 56
rows_upsert(), *36*, *56*
rows_upsert.tbl_lazy (rows_insert.tbl_lazy), 56

select(), *59*
select.tbl_lazy, 59
semi_join.tbl_lazy (join.tbl_sql), 40
setdiff.tbl_lazy (intersect.tbl_lazy), 39
show_query, 60
show_query(), *5*, *22*, *26*, *30*, *33*, *35*, *38*, *44*, *47*, *55*, *74*, *75*
simulate_access (backend-access), 6
simulate_dbi (backend-ansi), 7
simulate_hana (backend-hana), 9
simulate_hive (backend-hive), 9
simulate_impala (backend-impala), 10
simulate_mariadb (backend-mysql), 13
simulate_mssql (backend-mssql), 12
simulate_mysql (backend-mysql), 13
simulate_odbc (backend-odbc), 14
simulate_oracle (backend-oracle), 15
simulate_postgres (backend-postgres), 16
simulate_redshift (backend-redshift), 16
simulate_snowflake (backend-snowflake), 17
simulate_spark_sql (backend-spark-sql), 18
simulate_sqlite (backend-sqlite), 18
simulate_teradata (backend-teradata), 19
slice_max(), *27*
slice_max.tbl_lazy (dbplyr-slice), 27
slice_min(), *27*
slice_min.tbl_lazy (dbplyr-slice), 27
slice_sample(), *27*
slice_sample.tbl_lazy (dbplyr-slice), 27
sql, *32*, 61, *62*
sql(), *55*, *75*
sql_aggregate (sql_translation_agg), 68
sql_aggregate_2 (sql_translation_agg), 68
sql_aggregate_n (sql_translation_agg), 68
sql_cast (sql_translation_scalar), 69
sql_check_na_rm (sql_translation_agg), 68
sql_cot (sql_translation_scalar), 69
sql_dialect, *32*, 61, *63*, *66*, *72*, *77*
sql_escape_date (escape), 31
sql_escape_datetime (escape), 31
sql_escape_ident (escape), 31
sql_escape_logical (escape), 31
sql_escape_raw (escape), 31
sql_escape_string (escape), 31
sql_glue, 63
sql_glue2 (sql_glue), 63
sql_infix (sql_translation_scalar), 69
sql_log (sql_translation_scalar), 69
sql_not_supported (sql_translation_agg), 68
sql_options, 64
sql_options(), *21*, *23*, *54*, *60*
sql_paste (sql_translation_string), 70
sql_paste_infix (sql_translation_string), 70
sql_prefix (sql_translation_scalar), 69
sql_query_append (sql_query_insert), 65
sql_query_delete (sql_query_insert), 65
sql_query_insert, 65
sql_query_update_from (sql_query_insert), 65

sql_query_upsert *(sql_query_insert)*, 65
sql_runif *(sql_translation_scalar)*, 69
sql_str_sub *(sql_translation_string)*, 70
sql_substr *(sql_translation_string)*, 70
sql_translation(), *63*
sql_translation_agg, 68, *70*, *72, 73*
sql_translation_scalar, *68*, 69, *70*, *72, 73*
sql_translation_string, *68*, *70*, 70, *72, 73*
sql_translation_window, *68*, *70*, 71, *73*
sql_translator *(sql_variant)*, 72
sql_translator(), *68–71*
sql_try_cast *(sql_translation_scalar)*, 69
sql_variant, 72
sql_variant(), *68*, *70*, *72*
sql_vector *(escape)*, 31
summarise(), *37*, *74*
summarise.tbl_lazy, 74
summarise.tbl_lazy(), *26*

tally(), *26*
tally.tbl_lazy *(count.tbl_lazy)*, 26
tbl(), *57*
tbl.src_dbi, 75
tbl_dbi *(tbl.src_dbi)*, 75
tbl_lazy(), *54*
tbl_sql, *54*
tbl_sql(), *75*
tibble::tibble(), *45*
tidyr::complete(), *22*
tidyr::expand, *22*, *32, 33*
tidyr::pivot_wider(), *48*
tidyr::replace_na(), *55*
translate_sql, 76
translate_sql_ *(translate_sql)*, 76
transmute(), *46*

union.tbl_lazy *(intersect.tbl_lazy)*, 39
union_all.tbl_lazy *(intersect.tbl_lazy)*, 39

vctrs::vec_as_names(), *33*

win_absent *(sql_translation_window)*, 71
win_aggregate *(sql_translation_window)*, 71
win_aggregate_2 *(sql_translation_window)*, 71
win_cumulative *(sql_translation_window)*, 71
win_current_frame *(sql_translation_window)*, 71
win_current_group *(sql_translation_window)*, 71
win_current_order *(sql_translation_window)*, 71
win_over *(sql_translation_window)*, 71
win_rank *(sql_translation_window)*, 71
win_recycled *(sql_translation_window)*, 71
window_frame *(window_order)*, 78
window_frame(), *47*
window_order, 78
window_order(), *5*, *46*
with_dialect, 79
with_dialect(), *7*, *11*, *14*