Package 'lupa'

November 12, 2026

Title Data Profiling and Quality Measurement

Version 0.1.1

Description Profiles tabular data and measures its quality. Discovers structural patterns, implicit types, mixed date formats and reusable findings about potential data issues; defines quality frameworks with user-declared dimensions, reusable metrics with explicit granularity and typed aggregation; keeps versioned histories and detects structural drift; writes self-contained HyperText Markup Language reports; and produces editable cleaning plans that apply only the actions the user selects, with an audit log. The general model follows Batini and Scannapieco (2016) <doi:10.1007/978-3-319-24106-7>. The package implements version 1.6 of the data quality management framework for digital government of the Uruguayan Agency for Electronic Government and Information and Knowledge Society, AGESIC (2020) <https://www.gub.uy/agencia-gobierno-electronico-sociedad-informacion-conocimiento/comunicacion/publicaciones/marco-trabajo-para-gestion-calidad-datos>; an optional taxonomy of the fifteen characteristics of the standard 25012 of the International Organization for Standardization and the International Electrotechnical Commission, ISO/IEC (2008) <https://www.iso.org/standard/35736.html>; and the four levels and nineteen principles of the national quality assurance framework of the United Nations (2019) <https://unstats.un.org/unsd/methodology/dataquality/>, as adapted for Latin America and the Caribbean by the Statistical Conference of the Americas of the Economic Commission for Latin America and the Caribbean, CEA-CEPAL (2022) <https://repositorio.cepal.org/handle/11362/47464>. AGESIC, CEA and CEPAL are the Spanish acronyms of these institutions.

License GPL-3

Copyright Parts of the encoding-repair engine are translated from ftfy 6.3.1, copyright 2023 Robyn Speer (Apache-2.0); the missing-value sentinels come from naniar 1.1.0, copyright 2016-2018 Nicholas Tierney (MIT). All other code is copyright 2026 Sebastián Lucas. See file inst/COPYRIGHTS.

URL https://github.com/sebollin/lupa, https://sebollin.github.io/lupa/

BugReports https://github.com/sebollin/lupa/issues

Encoding UTF-8

Language es

LazyData true

Depends R (>= 4.2.0)

Imports cli (>= 3.0.0), data.table

Suggests bit64 (>= 4.0.5), covr, DBI, duckdb, jsonlite, knitr, odbc, rJava, rmarkdown, RPostgres, RSQLite, sf, stringdist, stringi, testthat (>= 3.1.7), tibble, units

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Sebastián Lucas [aut, cre, cph] (ORCID: <https://orcid.org/0009-0009-9068-0276>), Robyn Speer [ctb, cph] (byte tables, character categories, regular expressions and repair functions of the encoding-repair engine, translated from ftfy 6.3.1 (Apache-2.0)), Nicholas Tierney [ctb, cph] (missing-value sentinel values from naniar 1.1.0 (MIT))

Maintainer Sebastián Lucas <sebalucas@gmail.com>

Repository CRAN

Date/Publication 2026-11-12 15:50:02 UTC

Contents

accesores_perfil 4
agregar 5
analizar 8
analizar_tiempo 12
catalogo_agesic 13
clasificar_variables 15
cobertura_analisis 16
coleccion 18
comparar_equivalencia 20
comparar_evaluaciones 22
comparar_perfiles 23
contratos_medicion 26
datos_administrativos 27
datos_operativos 28
descubrir_patrones 29
detectar_asociaciones 31
detectar_claves 33

detectar_dependencias . . . 34
detectar_deriva_calidad . . . 37
detectar_discordancias . . . 38
detectar_duplicados_aproximados . . . 40
detectar_formatos_fecha . . . 47
detectar_relaciones . . . 48
distribucion_valores . . . 51
elegir_clave . . . 53
estadisticos_estimacion . . . 54
estimar_costo . . . 54
estimar_costo_coleccion . . . 59
evaluar . . . 60
granularidades . . . 61
guardar_historico . . . 62
guiar_limpieza . . . 63
historico_calidad . . . 64
indice_calidad . . . 66
inferir_tipo . . . 68
marco_calidad . . . 69
medicion_desde_estimaciones . . . 71
medir . . . 73
metricas_referencial . . . 75
modelo_calidad . . . 76
modelo_desde_propuesta . . . 83
normalizacion . . . 84
organizacion . . . 85
pack_validadores . . . 87
perfilar . . . 88
perfilar_coleccion . . . 124
perfilar_dbi . . . 128
perfilar_por . . . 155
persistir_analisis . . . 158
planificar_limpieza . . . 159
plan_perfilado_dbi . . . 166
propiedades_regla . . . 174
proponer_modelo . . . 175
referencial . . . 178
reglas_evaluacion . . . 179
relaciones_coleccion . . . 181
reportar . . . 185
requisitos_motor . . . 187
senal_redundante . . . 188
sentinelas_naniar . . . 189
sugerir_clave . . . 190
tablero_calidad . . . 192
validadores_formato . . . 194
validadores_uy . . . 196

Index **198**

accesores_perfil *Leer un perfil sin depender de su forma*

Description

perfilar(), perfilar_dbi() y perfilar_coleccion() devuelven objetos con formas distintas, y eso convertía la lectura en un acertijo: perfil$general$filas funciona sobre la salida en memoria y devuelve NULL sobre la salida DBI, donde el conteo vive en resumen_tabla$meta$filas. Un NULL silencioso en un guion de medición es la peor forma de fallar: no avisa, y lo que sigue calcula sobre nada.

Usage

```
hallazgos(x, ...)

columnas(x, ...)

cobertura(x, ...)

n_filas(x, ...)

sql_perfil(x, ...)
```

Arguments

x	Objeto de perfilar(), analizar(), perfilar_dbi() o perfilar_coleccion().
...	Sin uso, para compatibilidad de métodos.

Details

Estos accesores no agregan información: le ponen un solo nombre a la que ya está. La forma concreta de cada objeto sigue disponible y documentada.

Lo que no hay, no se inventa. Un perfilar_dbi() sin muestra leída no tiene hallazgos por fila, y hallazgos() devuelve una tabla vacía **con su aviso**, no una tabla que aparente que se midió y no había nada.

Value

hallazgos() devuelve un data.frame con una fila por hallazgo —sobre una colección, los de cada perfil conservado con la identidad de su tabla adelante (tabla, esquema, identificador), como sql_perfil(); la columna-lista trazabilidad conserva, en hallazgos que involucran varias columnas, el vector original columnas; columna sigue siendo el texto legible que se muestra y se puede guardar. si la colección no conservó los perfiles, una tabla vacía con su aviso—; columnas(), un data.frame con una fila por columna de la tabla perfilada y sus métricas y diagnósticos; cobertura() devuelve la tabla de diagnósticos no evaluados —y, sobre un perfil_dbi, la de **métricas SQL**, que es la cobertura de ese objeto: sus diagnósticos no evaluados viajan en x$perfil_muestra$cobertura_diagnosticos, con el esquema de perfilar(), y la impresión

del perfil los declara—; n_filas() devuelve el conteo de filas del alcance, o un vector nombrado con un conteo por tabla para una colección —los nombres son los de resumen_coleccion$tabla, en el orden de la tabla—. Devuelve NA con su motivo cuando el objeto no conoce el conteo. El significado de cada campo de columnas() se explica por familias en los detalles de perfilar(). sql_perfil() devuelve la tabla de consultas emitidas; sobre una colección incluye la identidad de la tabla y conserva las consultas aunque no se retengan los perfiles completos.

See Also

perfilar(), perfilar_dbi(), perfilar_coleccion()

Examples

```
perfil <- perfilar(datos_administrativos, analizar_dependencias = FALSE)
nrow(hallazgos(perfil))
n_filas(perfil)
nrow(cobertura(perfil))
```

---

agregar *Agregar medidas entre granularidades*

Description

Aplica exactamente una de las cuatro agregaciones del marco: ratio, ratio_umbral, promedio o promedio_ponderado. La transición se valida contra el grafo de transiciones_granularidad().

Usage

```
agregar(
  medidas,
  destino,
  funcion = c("ratio", "ratio_umbral", "promedio", "promedio_ponderado"),
  umbral = NULL,
  pesos = NULL,
  coleccion = NULL,
  colecciones = NULL,
  organizacion = NULL,
  organizaciones = NULL
)
```

Arguments

medidas	Data frame producido por medir() o una agregación anterior. Debe contener una sola métrica específica, una corrida y una granularidad.
destino	Granularidad de destino.
funcion	Una de "ratio", "ratio_umbral", "promedio" o "promedio_ponderado".
umbral	Umbral en [0, 1] requerido por ratio_umbral.

`pesos` Vector numérico requerido por `promedio_ponderado`, con una entrada por fila de `medidas`. Si trae nombres, se emparejan con `objeto_medible` o con el nombre de la parte —el que declaró la frontera, o el del objeto que la lista renombró— y se falla nombrando lo que sobra o falta —igual que en `indice_calidad()`, también cuando un nombre sólo difiere en su forma Unicode—, así que la misma declaración escrita en otro orden da el mismo número. Si los nombres casan con los dos vocabularios y reparten distinto, se rechaza: no se puede saber a cuáles se refieren. Sin nombres se leen por posición. Con otra `funcion` se rechaza, como `umbral` fuera de `ratio_umbral`: no pesarían nada y se publicarían igual.

`coleccion` Frontera declarada, exigida cuando `destino` es `"coleccion"`: el objeto de `coleccion()` o el perfil de `perfilar_coleccion()`. Sin ella no se sabe sobre qué tablas se está agregando, y el número resultante no describiría nada.

`colecciones` Lista nombrada de objetos de `coleccion()` o `perfilar_coleccion()`, exigida cuando `destino` es `"conjuntoColecciones"`. Los nombres declaran la identidad y la frontera del conjunto, y mandan sobre el del objeto, que la sigue reconociendo; no se agregan organizaciones ni otros alcances implícitos.

`organizacion` Frontera institucional declarada con `organizacion()`, exigida cuando `destino` es `"organizacion"`. Qué bases pertenecen a un organismo **no está en los datos**, así que lo declara quien lo sabe.

`organizaciones` Lista de objetos de `organizacion()`, exigida cuando `destino` es `"conjuntoOrganizaciones"`.

Los dos niveles institucionales son **opcionales**: un análisis de calidad no siempre tiene una organización detrás —una entrega suelta, un archivo que alguien mandó, una base sin dueño declarado—, y nada obliga a pasar por ellos. Existen para quien los necesita, y sin declaración `agregar()` se niega y explica cómo declararla, que es distinto de inventar una frontera que nadie nombró.

Details

`ratio` sólo acepta medidas booleanas. `ratio_umbral` sólo acepta medidas reales. Los promedios aceptan **esos dos tipos y ningún otro**, y siempre producen resultado real en `[0, 1]`. Los tres tipos no acotados —`numero_real`, `entero` y `duracion`— se rechazan nombrando la métrica que los declara: el promedio de una duración en días es un número de días, y publicarlo como resultado real en `[0, 1]` lo presentaría como una proporción. La guarda es por **tipo declarado**, no por el valor observado: una duración de `0,25` y `0,75` días se rechaza igual que una de `1,5` días, así que el mismo modelo se comporta igual con cualquier dato que le toque. Para el promedio ponderado, los pesos deben estar en `[0, 1]` y sumar uno dentro de cada objeto de destino. La columna `orientacion` se conserva sin invertir el resultado: un ratio de una métrica de defecto sigue siendo la proporción de defectos.

Value

Objeto `medicion` agregado, con una fila por objeto de destino.

Lo que el agregado conserva de la medición

Una agregación hereda lo que sus partes declararon: `configuracion_modelo`, `configuracion_aplicabilidad`, `cobertura_metricas`, `marco_calidad` y `fecha_declarada` viajan en el objeto. Lo último no es

decorativo: sin fecha_declarada, la guarda que impide comparar dos entregas en orden invertido queda desactivada y comparar_evaluaciones() publicaría el delta con el signo al revés.

alcance_medidas —«midió tres celdas de cuatro»— también viaja, pero **reexpresado en la clave del agregado**: la tabla que publica medir() está indexada por metrica_instanciada (Formato@t.cod) y el agregado renombra la métrica (agregada:ratio:Formato), así que arrastrarla tal cual dejaría una declaración que no se podría atribuir a ninguna fila. Los conteos se **suman** por objeto de destino, y sólo cuando todas las partes declaran la misma unidad; si no, la fila declara la mezcla en lugar de publicar un total que no estaría en ninguna unidad. Las partes **completas** también suman —una columna de 6 de 6 junto a una de 5 de 6 da 11 de 12—, también cuando se agregaron por separado y se unieron con rbind(); y en un destino por fila (instanciaEntidad) cada fila cuenta sus propias celdas: se declaran sólo las incompletas, con su número de fila en el motivo.

Una medida repetida —el mismo id_medida dos veces, como deja una medición unida consigo misma— se rechaza, como en evaluar(). El id_medida de cada fila agregada nombra lo que agrega —destino, función, métrica y objeto, como M-agg-atributo-ratio-NoNulo@t.x—, así que dos partes agregadas por separado se pueden unir y subir de nivel, y el mismo agregado repetido sigue repitiendo su identificador. Cada nombre lleva escapados los signos que lo separan del siguiente —el punto, la coma—, y desde la colección para arriba el identificador nombra también las partes que reúne: dos conjuntos distintos, o dos colecciones sin nombre, no comparten identificador. Un valor que no se puede agregar se rechaza nombrando la métrica y la causa: un tipo no acotado, una medida ausente o suprimida, o un valor fuera de [0, 1]. Cuando el origen es una instancia de entidad y el destino es una entidad, el nombre de la métrica agregada conserva también la instancia completa, incluida la entidad y sus atributos: agregada:ratio:ClaveDuplicada@t.a.b. Eso mantiene separadas dos claves distintas de la misma tabla. umbral sólo se acepta con ratio_umbral, y el borde se compara con tolerancia de redondeo: 1 - 0.9 alcanza el umbral 0.1.

Las coberturas de frontera (cobertura_coleccion y sus hermanas) **no** se arrastran: cada agregación calcula la de su propio destino con las partes que entraron a ese número, y hereda las de sus partes en cobertura_de_partes. Dos partes agregadas por separado y unidas con rbind() conservan cada una la suya, en cualquier orden de la unión, y el tablero, el índice, el informe, evaluar() e historico_calidad() las publican todas. Unidas dos corridas distintas, cada cobertura lleva además su id_medicion, y el histórico registra la parte no medida en la corrida a la que le faltó. Una colección lleva en cobertura_metricas sólo la de sus propias tablas.

Los pesos se publican

Cuando hay pesos, el objeto los publica en pesos_declarados, con el nombre de la parte que recibió cada uno, de modo que el número se pueda rehacer con lo que el objeto trae. Y si alguna parte entró con peso cero, se declara en partes_con_peso_cero: entró al número sin aportarle nada, queda contada en la identidad de la fila y leerlo sin saberlo es leer otra cosa. La declaración depende de que **haya** un peso cero, no del destino: va como atributo en todos, también en conjuntoEntidades, y en los cuatro destinos que tienen cobertura de frontera entra además en esa cobertura, que es la que se imprime.

Cuando el destino es conjuntoEntidades, promedio combina las partes sin pesos porque las medidas de nivel entidad no llevan su cantidad de filas. El resultado lo declara en advertencia_agregacion: es un promedio sin pesos y agregar() no conoce el alcance de cada parte. En coleccion, la frontera se declara y por eso sólo se admite promedio_ponderado: sin pesos no se puede decidir cuanto debe aportar cada tabla de esa coleccion.

No existe una transición hacia factor, dimensión o modelo: esos campos son taxonómicos y esta función no calcula un índice global.

Examples

```
nucleo <- metricas_nucleo()
especifica <- especializar(nucleo$NoNulo)
instancia <- instanciar(especifica, "personas", "edad")
medidas <- medir(modelo(instancia), data.frame(edad = c(20, NA, 35)))
agregar(medidas, "atributo", "ratio")
```

`analizar` *Ejecutar el análisis descriptivo completo*

Description

Es la puerta de entrada al recorrido de `lupa`. Construye el perfil, las distribuciones, asociaciones, diagnóstico temporal, clasificación confirmable de variables, propuesta de modelo, medición agregada, tablero, cobertura conceptual y plan de limpieza. No modifica los datos.

Usage

```
analizar(
  datos,
  nombre = .nombre_de_los_datos(substitute(datos)),
  fecha = Sys.time(),
  argumentos_perfil = list(),
  metadatos_variables = NULL,
  modelo_confirmado = NULL,
  propuesta_confirmada = NULL,
  marco = NULL,
  perfil_evaluacion = NULL,
  id_medicion = NULL,
  medir_propuesta = TRUE,
  conservar_detalle_medicion = FALSE,
  muestra = 1e+05,
  muestra_asociacion = 10000,
  max_valores = 20L,
  probabilidades = c(0, 0.25, 0.5, 0.75, 1),
  umbral_asociacion = 0.3,
  max_columnas_asociacion = 50L,
  max_niveles_asociacion = 50L,
  max_pares_asociacion = 500L,
  metodo_asociacion_numerica = c("pearson", "spearman"),
  calendario = 1:7,
  frecuencia_dias = NULL,
  max_huecos = 20L,
  max_columnas_temporales = 50L,
```

```
  conservar_datos = FALSE,
  proteger_datos_personales = FALSE,
  ...
)
```

Arguments

datos	Tabla que se desea analizar.
nombre	Nombre de la entrega.
fecha	Fecha y hora reproducible de la corrida. Se lee una vez, como en perfilar() y medir(): en UTC, también un texto como "2026-03-01". Si se omite, el perfil y la medición lo registran igual que esas dos funciones (fecha_declarada = FALSE), y comparar_perfiles() no les exige el orden temporal: la hora de la corrida no dice nada sobre las entregas.
argumentos_perfil	Lista de argumentos adicionales para perfilar(). Los nombres se completan como los completa perfilar() —aplic es aplicabilidad— antes de leerlos, así que un nombre abreviado gobierna el perfil y la medición por igual. sentinelas_numericos también gobierna las dependencias del perfil y las asociaciones: las filas que contienen un valor declarado no se evalúan en esos pares.
metadatos_variables	Declaraciones para clasificar_variables().
modelo_confirmado	Modelo creado por modelo() o NULL.
propuesta_confirmada	Propuesta editada por el usuario o NULL.
marco	Taxonomía opcional creada por marco_calidad(). Si se omite, usa la asociada a modelo_confirmado y, en último término, marco_agesic().
perfil_evaluacion	Perfil explícito para evaluar() o NULL.
id_medicion	Identificador opcional enviado a medir(). Si argumentos_perfil declara aplicabilidad, esa misma declaración viaja a la medición: el universo se declara una vez y gobierna todo el análisis, así que el perfil y el tablero cuentan el mismo dato de la misma manera. distribuciones recibe esa declaración y la de sentinelas_numericos: lo que queda fuera del universo no entra en sus frecuencias ni en sus cuantiles, y un centinela declarado sale de los cuantiles como sale de minimo y maximo del perfil.
medir_propuesta	Si se mide automáticamente la propuesta con estado "lista" e incluir = TRUE cuando no se recibe un modelo o una propuesta confirmada. Con FALSE, analizar() describe y no mide.
conservar_detalle_medicion	Si se retienen las medidas fila a fila. Es FALSE por omisión: el tablero y la medición agregada permanecen, y la medición se hace métrica por métrica, agregando cada una antes de pasar a la siguiente, así que el detalle entero no existe nunca a la vez. Con TRUE el detalle es parte del resultado y su memoria crece

con las filas por las métricas: una tabla de 19 columnas producía 21 filas de medida por fila de datos. Si ninguna métrica del modelo produce medidas, el tablero sale vacío y lleva el motivo de cada una en `cobertura_metricas`, en lugar de abortar el análisis.

`muestra` Límite de filas para perfil, distribuciones y enumeración de niveles observados.

`muestra_asociacion`
: Límite común de filas para asociaciones.

`max_valores` Máximo de valores por tabla de frecuencias.

`probabilidades` Cuantiles solicitados.

`umbral_asociacion`
: Asociación mínima informada.

`max_columnas_asociacion`
: Máximo de columnas para asociaciones.

`max_niveles_asociacion`
: Máximo de niveles categóricos.

`max_pares_asociacion`
: Máximo de pares devueltos.

`metodo_asociacion_numerica`
: Medida entre columnas numéricas que usa `detectar_asociaciones()`: `"pearson"` por omisión, o `"spearman"` para asociación monótona sobre los rangos, que no supone linealidad.

`calendario` Días ISO usados por el análisis temporal.

`frecuencia_dias`
: Frecuencia temporal conocida o `NULL` para proponerla.

`max_huecos` Máximo de grupos de huecos por columna.

`max_columnas_temporales`
: Máximo de columnas temporales analizadas.

`conservar_datos`
: Si el objeto retiene una copia de la entrada. Es `FALSE` por omisión para limitar tamaño y exposición. Con protección activa, las columnas personales de esa copia también se enmascaran, y en las demás columnas se tapa el valor protegido que identifica —el piso de `perfilar()`—: la cédula citada en una observación, `"pago CI 3.738.538-0"`, o guardada como número en otra columna. En todo lugar donde un `data.frame` guarda un valor: las celdas de cualquier clase —una copia `POSIXct` de la fecha con hora protegida queda en `NA`, como la tapa `perfilar()`—, las columnas de lista elemento por elemento, los atributos de la tabla y de cada columna —también `levels` en lo que no es un factor y `units`, y el nombre de cada atributo, además de su valor—, los nombres de columna de una tabla anidada en una columna lista, y los nombres de fila, que vuelven a ser los números de fila si alguno lleva un valor protegido. Una columna `POSIXlt` se tapa como la `POSIXct`, con sus atributos —`comment`, `label`—, y cualquier lista con clase —un registro de `vctrs`, un `blob`— por sus valores; un BLOB (`raw`) se lee como el texto que guarda y, si lleva un valor protegido, la copia guarda los bytes del texto tapado; un entorno no se modifica —es del usuario—: si guarda un valor protegido, en su contenido o en sus atributos —valor y nombre, también `name`, que `print()` muestra—, la copia lleva `"[valor protegido]"` en su

lugar, también si tiene un atributo `name`; y también si no se puede leer entero —una promesa o una ligadura activa que aborta al leerse—, porque lo que no se lee no se sabe limpio; sólo el global, el de un paquete y los de la ruta de búsqueda se dejan como están. Una llamada, una fórmula o una función se leen como el texto que `print()` escribe —con sus atributos y, si guarda su fuente (`srcref`), la fuente con sus comentarios—, y también por sus atributos y, la función, por su entorno; si algo de eso lleva un valor protegido, la copia lleva `"[valor protegido]"` en su lugar. Una columna protegida conserva sólo los atributos de su clase (`class`, `tzone`, `units`, `dim`), y `tzone` y `units` pasan por el piso: las etiquetas de valor de un `haven_labelled` se quitan. Para conservar sus valores debe usarse `proteger_datos_personales = FALSE` al crear el analisis y al guardarlo. La opcion del guardado no recupera valores que el analisis ya enmascaro.

`proteger_datos_personales`
: Si perfiles y resúmenes ocultan valores de columnas cuya clasificación activa protección automática, incluidos estadísticos de orden, cuantiles y rangos temporales. `FALSE` por omisión.

`...`
: Argumentos con nombre enviados a `perfilar()`. Es una alternativa concisa a `argumentos_perfil`.

Details

Por omisión mide las sugerencias de la propuesta cuyo estado es `"lista"` y cuya marca `incluir` es `TRUE`; los patrones y dominios observados que `proponer_modelo()` deja inactivos no llegan al tablero sin confirmación. Declara que esa selección fue realizada por `lupa` sin confirmación. Agrega inmediatamente el detalle y conserva `tablero_calidad()`; las medidas fila a fila sólo quedan en el resultado si `conservar_detalle_medicion = TRUE`. La evaluación, cuando se solicita, usa la medición agregada. Los hallazgos `casi_clave` del perfil se reiteran en `advertencias`, con su columna, valores en colisión, frecuencias y criterio observado, para que no queden ocultos dentro del recorrido integral.

Value

Objeto S3 `analisis` con todos los componentes y su cobertura.

Tres diferencias respecto de llamar a las funciones sueltas, y las tres son deliberadas. La evaluación usa la medición **agregada**, no las medidas por fila: `evaluar(medir(...), perfil)` aplica la regla a cada medida y puede dar otra cifra que `analizar(perfil_evaluacion = perfil)` sobre los mismos datos —una proporción de medidas contra una proporción de métricas—, y sobre medidas de `0` o `1` los tres `perfiles_madurez()` no se distinguen. Para reproducirla con las funciones sueltas, `agregar()` antes de `evaluar()`. La entrada puede ser una matriz de dos dimensiones: se la pasa a `perfilar()` sin convertir, para que sea él quien la convierta y lo declare en `meta$entrada_convertida`. Y cuando una columna temporal pertenece a una columna protegida, `temporal$resumen` gana la columna `proteccion_temporal` con el texto `[rangos y huecos protegidos]`: la protección se aplica y se declara en la misma tabla donde faltan los valores. La llamada directa a `analizar_tiempo()` respeta la misma declaración cuando recibe ese `perfil`.

See Also

`guardar_analisis()`, `reportar()`, `cobertura_analisis()`

Examples

```
# Cuatro columnas de la tabla de ejemplo, para que corra en pocos segundos.
columnas <- c("fecha_nacimiento", "sexo", "ingreso", "departamento")
resultado <- analizar(
  datos_administrativos[, columnas], analizar_dependencias = FALSE,
  fecha = as.POSIXct("2026-01-15", tz = "UTC")
)
resultado
```

`analizar_tiempo` *Examinar regularidad y cobertura temporal*

Description

Para cada columna temporal propone una frecuencia en días. La confianza es el mínimo entre la contigüidad de las fechas sobre la grilla propuesta y la cobertura del período: una coincidencia breve dentro de una serie muy dispersa no puede producir confianza alta. Ambas componentes se devuelven para que la propuesta sea auditable. La propuesta nunca queda confirmada automáticamente. `calendario` usa días ISO: 1 es lunes y 7 domingo; así una oficina puede declarar `1:5` sin que la ausencia de fines de semana se interprete como hueco. Las fechas-hora se llevan a fecha civil en UTC para que el resultado no dependa de la zona del equipo que ejecuta el análisis.

Usage

```
analizar_tiempo(
  datos,
  perfil = NULL,
  columnas = NULL,
  calendario = 1:7,
  frecuencia_dias = NULL,
  max_huecos = 20L,
  max_columnas = 50L,
  proteger_datos_personales = FALSE,
  columnas_personales = character()
)
```

Arguments

`datos`	Tabla que se desea examinar.
`perfil`	Perfil opcional de los mismos datos.
`columnas`	Columnas temporales; `NULL` usa clases e inferencia.
`calendario`	Días de semana esperados, enteros entre 1 y 7.
`frecuencia_dias`	Frecuencia entera conocida en días. Si es `NULL`, se propone la moda de los intervalos positivos.
`max_huecos`	Máximo de grupos de huecos devueltos por columna.

`max_columnas` Máximo de columnas temporales analizadas.

`proteger_datos_personales`
: Si se ocultan los rangos y los huecos de las columnas temporales que son dato personal. `FALSE` por omisión.

`columnas_personales`
: Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`. Se suman a las que protege el `perfil` o, sin él, a las que se clasifican aquí. Sólo tiene efecto con `proteger_datos_personales = TRUE`.

Value

Objeto `analisis_temporal` con `resumen`, `dias_semana`, `huecos` y `propuestas`. El recorte de huecos queda en `resumen`; el de columnas, en atributos del objeto. `resumen` agrega tres campos sobre el alcance del resumen, con el mismo vocabulario que usa `perfilar()` sobre la misma columna: `n_fechas_excluidas_parseo` cuenta los valores **presentes** que ningún formato confirmado pudo convertir y que por eso quedaron fuera; `n_fechas_excluidas_granularidad`, cuántos de ésos son períodos de mes — `2024-02` y sus formas—, que no se convierten a propósito porque no nombran un día; y `estado_resumen` vale `"calculados_sobre_dias"` cuando hubo períodos de mes, `"calculados_sobre_fechas_parseadas"` cuando sólo hubo valores ilegibles, y `"calculados"` cuando no quedó nada afuera. Los tres se informan siempre, haya muestreo o no, porque el descarte tampoco depende del muestreo. Un `NA` no entra en esta cuenta: es una ausencia declarada y se informa como faltante, no como valor que no se pudo leer.

Una columna mixta —días y meses— **sí** se resume, sobre sus fechas completas. Sólo cuando ningún formato confirmado nombra un día no hay serie diaria que construir; entonces la columna no aparece en `resumen`, y se declara en los atributos `columnas_omitidas` y `columnas_sin_serie_diaria` en vez de desaparecer. Si se recibe un `perfil` que declara proteccion para una columna temporal, sus rangos y huecos se protegen tambien en esta llamada directa, con el mismo texto que usa `analizar()`. Sin `perfil`, la columna temporal que es un dato personal —una fecha de nacimiento— se clasifica aquí por léxico y por forma, como en `clasificar_variables()`, y sus rangos y huecos se protegen igual; lo que sólo quien conoce los datos sabe personal entra por `columnas_personales`.

See Also

`detectar_formatos_fecha()`, `analizar()`

Examples

```
fechas <- as.Date("2026-01-01") + c(0:4, 20:24)
analizar_tiempo(data.frame(fecha = fechas))
```

---

catalogo_agesic *Correspondencia con el catálogo de métricas de AGESIC*

Description

Devuelve las 49 entradas del catálogo del marco y explicita qué parte está disponible en `lupa`. Los ratios no se duplican como métricas: aparecen con estado `"via_agregacion"` y con la llamada que los materializa.

Usage

```
catalogo_agesic()
```

Details

`estado` responde qué disponibilidad tiene cada entrada:

- `implementada`: existe un motor ejecutable, completo o parcial;
- `via_agregacion`: se obtiene agregando una métrica base;
- `pendiente`: es automatizable, pero el motor necesario aún no existe;
- `fuera_de_alcance`: se decidió no implementarla en el alcance tabular de esta versión, por ejemplo métricas geográficas o de imágenes.

`motivo` separa la causa o el matiz: `semantica_completa`, `semantica_parcial`, `agregacion`, `requiere_referencial`, `requiere_configuracion`, `motor_pendiente` o `decision_alcance`. Así una métrica ejecutable que necesita un padrón externo no se confunde con una métrica cuyo motor falta. `observacion` explica la situación concreta de cada una de las 49 entradas y nunca queda vacía.

`Escala` se clasifica como implementada con configuración experta mediante `escala()`, no como referencial. `DesactualizacionPorFecha`, `DesactualizacionPorCambios` y las oportunidades de entidad requieren un contrato `vigencia()`. `ErrorEstandar` sigue la semántica literal de la tabla 16.5 y devuelve desviación estándar, aunque su nombre pueda sugerir el error estándar de la media.

La implementación de `ReglaIntegridadInterEntidad` se declara parcial: calcula cobertura PK/FK, pero todavía no materializa la regla booleana de inclusión o expresión condicional con granularidad `conjuntoEntidades` que define el marco. `OportunidadAtributo*`, en cambio, sigue las tablas 16.29 y 16.30 con resultado booleano. Las variantes continuas del curso CPAP se conservan como `GradoOportunidadAtributo*` y no figuran como entradas del catálogo. `RatioDensidadPonderada` usa `ratio_umbral`, porque su medida base es real y `ratio` sólo es válido para medidas booleanas.

`implementacion` publica una receta, no una expresión que se evalúe sola: en las entradas `via_agregacion`, `m` es la medición ya tomada de la métrica base que nombra `metrica_lupa` y `u` el umbral que elige quien agrega. Son los dos únicos nombres libres que aparecen, y ninguno se inventa aquí: `m` sale de `medir()` y `u` es una decisión de política —cuánta densidad se considera suficiente— que el paquete no toma por nadie.

Value

Data frame con una fila por entrada y las columnas `numero`, `dimension`, `factor`, `metrica_agesic`, `clase_catalogo`, `estado`, `motivo`, `metrica_lupa`, `implementacion` y `observacion`. `estado` y `motivo` son factores.

References

AGESIC (2020). *Marco de trabajo para la Gestión de la Calidad de Datos en Gobierno Digital*, versión 1.6, capítulo 16, Presidencia de la República, Uruguay.

See Also

`metricas_nucleo()`, `metricas_referencial()`, `agregar()`

Examples

```
catalogo <- catalogo_agesic()
subset(catalogo, estado == "via_agregacion")
table(catalogo$estado, catalogo$motivo)
```

`clasificar_variables` *Proponer escalas y roles de las variables*

Description

Separa el tipo de almacenamiento, el tipo implícito y la escala de medición. Una propuesta basada sólo en valores nunca queda confirmada: `1, 2, 3` puede ser conteo, código o nivel. Las clases `ordered`, `factor`, `logical`, las clases temporales y el metadato `measure` sí constituyen declaraciones. `measure = "scale"` se publica como `continua`, la lectura que SPSS da a ese nivel. Una unidad declarada —el atributo `units`— establece una magnitud, pero no si es discreta o continua: confirma `continua` cuando los valores traen decimales, y con valores enteros la propuesta queda sin confirmar, porque pueden ser una cuenta o una medida redondeada.

Usage

```
clasificar_variables(
  datos,
  perfil = NULL,
  metadatos = NULL,
  max_niveles = 100L,
  muestra = 1e+05,
  proteger_datos_personales = FALSE,
  columnas_personales = character()
)
```

Arguments

`datos`	Tabla que se desea clasificar.
`perfil`	Perfil opcional de los mismos datos.
`metadatos`	Declaraciones opcionales por columna.
`max_niveles`	Máximo de niveles guardados en cada lista.
`muestra`	Máximo de valores usados para enumerar niveles no declarados; los niveles ausentes de factores y metadatos se verifican sobre toda la columna, y `n_niveles_observados` cuenta los valores distintos de la columna entera, con o sin `perfil`. `niveles_muestreados` dice si la lista de niveles salió de una muestra.
`proteger_datos_personales`	Si se ocultan niveles concretos de columnas cuya clasificación activa protección automática. Véase `perfilar()`. Con `perfil`, se respeta la clasificación de ese perfil; sin `perfil`, se rehace aquí por léxico y por forma, y lo declarado entra por `columnas_personales`.

`columnas_personales`
: Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`: nombres de columna, o un vector con nombre donde el nombre es la columna y el valor es el tipo. **Es el único camino cuando no se pasa** `perfil`: sin perfil, la clasificación corre por léxico y por forma, así que una columna que sólo quien la conoce sabe personal —una edad, un legajo interno— se publicaría entera. Sólo tiene efecto con `proteger_datos_personales = TRUE`.

Details

Con `perfil`, una columna declarada en su `columnas_codigo` es un código y no una magnitud: en lugar de `discreta` o `continua` con rol `medida`, sale `nominal`, con rol `categoria` y confirmada, porque la declaración es del usuario. `metadatos` la puede corregir igual.

`metadatos` permite confirmar o corregir la propuesta con una tabla editable. Debe contener `columna` y puede incluir `escala`, `rol`, `confianza`, `confirmada`, `unidad` y una columna de lista `niveles`. Las escalas válidas son nominal, ordinal, discreta, continua, binaria, temporal y desconocida.

Los niveles declarados y observados se conservan como columnas de lista. Los niveles ausentes son una observación, no prueba de error. Si la evidencia de dato personal activa la protección, los niveles concretos se protegen, y los valores de esas columnas escritos en los niveles de OTRA columna —una observación que cita la cédula— se tapan con el mismo piso que aplica `perfilar()`. Sólo las escalas nominal, ordinal y binaria guardan niveles observados; en las demás, `n_niveles_observados` es `NA`, porque la cifra no aplica.

Una columna numérica sin ningún valor finito —sólo `Inf` o `-Inf`— queda `desconocida`: `Inf` no es un entero ni un estado.

Value

Data frame S3 `clasificacion_variables`, editable y filtrable.

See Also

`analizar()`, `distribucion_valores()`, `proponer_modelo()`

Examples

```
d <- data.frame(
  prioridad = ordered(c("baja", "alta"), levels = c("baja", "media", "alta")),
  cantidad = c(1L, 2L)
)
clasificar_variables(d)
```

`cobertura_analisis` *Informar la cobertura conceptual de un análisis*

Description

Devuelve una fila por dimensión y factor del `marco_calidad()` elegido. Usa `marco_agesic()` por omisión, pero acepta cualquier taxonomía declarada. Distingue lo efectivamente medido de lo que no fue declarado, lo que no aplica a los tipos presentes y lo que queda fuera del alcance actual. La tabla evita que la ausencia de un hallazgo se interprete como evidencia de calidad.

Usage

```
cobertura_analisis(perfil, medicion = NULL, modelo = marco_agesic())
```

Arguments

`perfil`	**Primer argumento.** Objeto creado por `perfilar()` o `perfilar_dbi()`. En el segundo caso se analiza `perfil_muestra` cuando existe; un perfil por agregados usa el resumen de la tabla y cubre la tabla completa. Cuando corresponde, cada motivo indica que la cobertura es la de la muestra y no la de la tabla completa.
`medicion`	**Segundo argumento, opcional.** Objeto creado por `medir()`, con métricas ejecutadas que pueden completar esa cobertura. No es un perfil creado por `perfilar()`. **La correspondencia entre los dos objetos es suya, no del paquete.** La cobertura aparea por dimensión y factor, y no verifica que la medición provenga de la tabla que se perfiló: si le pasa una medición de otra tabla, el factor va a figurar como medido por esa corrida. El paquete no puede comprobarlo porque un perfil no guarda una identidad de tabla comparable con la entidad de la medición.
`modelo`	**Tercer argumento.** Objeto creado por `marco_calidad()` que actúa como referencia conceptual. El nombre enfatiza que no es el modelo operativo creado por `modelo()` que recibe `medir()`.

Details

En el marco incluido, el profiling automático mide densidad y no duplicación. Un marco propio puede marcar otros factores mediante la columna `perfil_mide`. Los demás sólo pasan a `"medida"` cuando `medicion` contiene una métrica del factor; descubrir un patrón o una dependencia no los convierte por sí solo en un requisito confirmado.

Para AGESIC, `como_resolverlo` se deriva de las entradas implementadas o agregables de `catalogo_agesic()`, no de una lista independiente de factores. Por ejemplo, `No-contradicción` recomienda instanciar `EntidadContradictoria` sobre un atributo; no requiere una clave y, cuando compara más de un valor distinto, necesita el paquete opcional `stringdist`.

Value

Data frame con `marco`, `dimension`, `factor`, `estado`, `motivo` y `como_resolverlo`. `marco` identifica explícitamente la taxonomía contra la que se calculó la tabla. `estado` es un factor con niveles `"medida"`, `"no_declarada"`, `"no_aplica"` y `"fuera_de_alcance"`.

El `motivo` publica el alcance cuando el perfil resumió sólo parte de las columnas. Si ninguna se pudo resumir, el `estado` es `"no_aplica"`; si se resumieron todas, el motivo lo dice; y si se resumieron algunas, el `estado` sigue siendo `"medida"` —el factor sí se midió, en las columnas que se pudieron resumir— y el motivo empieza con `"Parcial: "` y nombra cuántas de cuántas.

Esas filas se encuentran con `grepl("^Parcial: ", cobertura$motivo)`. El `estado` no distingue el caso parcial a propósito: es un factor de cuatro niveles documentados que `tablero_calidad()` valida, y el grado —1 de 100 no es 99 de 100— no cabe en un nivel. El alcance viaja en el motivo, y `tablero_calidad()` lo conserva al cruzar la cobertura con su medición.

See Also

`marco_calidad()`, `perfilar()`, `medir()`, `vigencia()`, `escala()`, `reportar()`

Examples

```
perfil <- perfilar(datos_administrativos, analizar_dependencias = FALSE)
cobertura_analisis(perfil)
```

`coleccion` *Declarar la frontera de una colección*

Description

Una colección es una base de datos entera: el séptimo nivel de granularidad del marco. `lupa` no la descubre recorriendo el catálogo — **la frontera la declara quien conoce la base**, igual que los pesos de `indice_calidad()` o el marco de `cobertura_analisis()`. Explorar el catálogo convertiría un error de permisos en un resultado, y en bases reales de más de mil tablas recorrerlas todas no es viable.

Usage

```
coleccion(conexion, tablas, nombre = NULL)
```

Arguments

`conexion`	Conexión abierta compatible con DBI.
`tablas`	Nombres de las tablas que componen la colección: un data frame con columnas `esquema`, `tabla` y opcionalmente `tipo`; un `DBI::Id` o una lista de `DBI::Id`; o un vector de texto `"esquema.tabla"`.
`nombre`	Etiqueta de la colección.

Details

Esta función no consulta nada: sólo declara. Lo que se mide viene después, con `perfilar_coleccion()`.

El **esquema es parte de la identidad de la tabla**, y el **catálogo también** cuando existe: `catalogo.esquema.tabla` es el nombre de tres partes que usan SQL Server y otros motores. Se declara con la columna `catalogo`, y donde no hay catálogo se omite —o va `NA`— y la identidad queda igual que siempre. Dos tablas con el mismo nombre y esquema en catálogos distintos son **dos tablas distintas**, y se cuentan como dos. Una tercera columna `tipo` permite declarar qué es cada objeto —`"tabla"`, `"vista"`, `"temporal"`—, porque el conteo bruto de un catálogo mezcla tablas base con vistas, índices y secuencias, y no todas se perfilan igual.

Value

Objeto de clase `coleccion_lupa`: una lista con `nombre`, la `conexion` declarada, `tablas` -un data frame con una fila por tabla declarada y sus columnas de esquema y nombre-, `motor` con la clase del controlador DBI, y `n_declaradas` con la cantidad de tablas. No abre consultas ni lee datos: sólo declara qué tablas forman la colección.

Cómo declarar el nombre

Hay tres formas, y **no son equivalentes**:

`data.frame(esquema =, tabla =)` **La forma recomendada para cualquier nombre no trivial.** Los nombres viajan literales: puntos, espacios y comillas incluidos. No hay parseo y por lo tanto no hay nada que se pueda parsear mal.

`DBI::Id`**, suelto o en una lista** La forma canónica de DBI. Tampoco se parsea. Se admiten hasta tres componentes —catálogo, esquema y tabla—. Con cuatro o más se rechaza **nombrando la causa**: por encima del catálogo no hay un nivel que la colección sepa declarar, y devolver ese error como si fuera un problema de permisos mandaría a pedir un acceso que ya se tiene.

Texto `"esquema.tabla"` Atajo cómodo para el caso simple. El texto **se parte en el punto**, respetando el entrecomillado del motor: un nombre entrecomillado con un punto adentro queda entero, como una sola tabla. Un nombre de tres o más partes, un punto al principio o al final, o una comilla sin cerrar se **rechazan acá**, con el motivo real. No se aceptan para fallar más tarde como si fueran un problema de permisos.

El atajo de texto no puede resolver una ambigüedad genuina: `"informe.2024"` puede ser la tabla `2024` del esquema `informe` o una tabla llamada `informe.2024`. `lupa` elige la primera lectura y lo deja anotado, de modo que si después no encuentra la tabla dice que el nombre se partió. Para el caso ambiguo use el data frame.

See Also

`perfilar_coleccion()`, `perfilar_dbi()`, `granularidades()`

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "personas", data.frame(id = 1:3, nombre = letters[1:3]))
  coleccion(con, "personas", nombre = "padron")
  # La forma literal, sin parseo, para nombres con puntos:
  coleccion(con, data.frame(esquema = NA, tabla = "personas"))
  DBI::dbDisconnect(con)
}
```

`comparar_equivalencia` *Comparar la equivalencia de dos resúmenes de perfiles*

Description

Compara por intersección los campos registrados de dos perfiles y devuelve una fila por cada par de columna y campo. Los campos que no tienen un eje registrado no se comparan y quedan declarados en `campos_no_comparables`, y lo mismo pasa con un campo registrado que uno de los dos lados no mide: se declara ahi, con el motivo `campo_solo_en_anterior` o `campo_solo_en_actual` en `detalle_campos_no_comparables` y una fila en `cobertura_diagnosticos`. Los campos bajo protección tampoco se comparan: se declaran por columna, campo y lado en `campos_protegidos`.

Usage

```
comparar_equivalencia(anterior, actual, tolerancia)
```

Arguments

`anterior`, `actual`
: Un objeto `perfil` de `perfilar()`, un objeto `perfil_dbi` de `perfilar_dbi()` o directamente un frame `columnas`.

`tolerancia`
: Número escalar no negativo y finito, declarado por quien llama. No tiene valor por omisión y se publica en cada fila.

  La tolerancia es mixta, no relativa. La diferencia se divide por el mayor de `1`, `|anterior|` y `|actual|`, que es la misma regla que usa la detección de relaciones aritméticas: por debajo de magnitud 1 el divisor es 1 y la comparación pasa a ser absoluta, porque dividir por casi cero convierte el ruido en un cambio del cien por ciento. Eso tiene una consecuencia que conviene tener presente: una media que pasa de `0,001` a `0,4` queda dentro de una tolerancia de `0,5`. Cuando la escala usada fue la unidad y no el valor, el `motivo` de esa fila lo dice (`dentro_de_tolerancia_escala_unidad`, `fuera_de_tolerancia_escala_unidad`) y la columna publicada se llama `diferencia_normalizada`, no «relativa».

Details

El registro fijo asigna tolerancia sólo a `media`, `mediana`, `desvio` y `longitud_media`. Los conteos, proporciones de conteos y extremos por selección se comparan en el eje `exacto`; las fechas canónicas en `fecha` y `moda` y `centinela_valor` en `valor`. Los ejes `exacto`, `fecha` y `valor` son binarios por construcción.

El comparador devuelve datos, no decisiones: no alimenta hallazgos, severidades ni puntajes. La tolerancia es del llamador y jamás entra en una regla del paquete; sólo se aplica al eje flotante finito y queda publicada para que el llamador decida cómo usarla.

Si una columna es temporal en exactamente uno de los perfiles, los campos de magnitud numérica se omiten por el cambio de esquema y su motivo queda en `detalle_campos_no_comparables` como `tipo_cambiado:temporal_vs_no_temporal`. Así no se comparan duraciones en segundos

contra magnitudes numéricas sin unidad común. Cuando ambos lados son temporales, `desvio` sí se compara en flotante porque ambas puertas lo expresan en segundos.

Si un lado es temporal y el otro **no publica su tipo** —el frame de `perfilar_dbi()` no trae ningún campo de tipo—, no se afirma un cambio de tipo: los mismos campos se omiten con el motivo `tipo_desconocido:temporal_vs_sin_tipo`, porque tampoco se puede garantizar una unidad común. Una fecha guardada como número en el motor tiene su `desvio` en días de ese lado y en segundos del otro, y compararlos publicaría como diferencia del dato una diferencia de unidad.

Si una columna guarda caracteres en exactamente uno de los perfiles, los campos calculados sobre la representación —longitudes, variantes unicode y números escritos como texto— se omiten y su motivo queda en `detalle_campos_no_comparables` como `tipo_cambiado:texto_vs_no_texto`. Esos campos cambian con sólo cambiar el almacenamiento, aunque el dato sea el mismo, de modo que compararlos publicaría el síntoma y callaría la causa. `factor` cuenta como almacenamiento de caracteres. Este guarda lee el tipo *declarado* y no el inferido: si el texto contiene números o fechas, la inferencia borra justo la diferencia que hay que ver. Cuando el tipo declarado viene de un vocabulario ajeno al de memoria —un tipo SQL, por ejemplo— no se afirma nada y la comparación sigue como siempre.

Por la misma razón, si una columna lleva zona horaria en exactamente uno de los perfiles, los campos que sólo ese almacenamiento puede medir se omiten con el motivo `tipo_cambiado:con_zona_vs_sin_zona`. Del otro lado no dan un valor distinto: no dan ninguno, porque no hay zona que comparar contra UTC. Los extremos de fecha **sí** se siguen comparando: al guardarse como texto la columna pierde la zona y los instantes que denota son realmente otros, que es lo más importante que hay para informar en ese caso.

Value

Un frame de clase `equivalencia_perfiles` con `columna`, `campo`, `valor_anterior`, `valor_actual`, `diferencia_normalizada`, `veredicto`, `motivo`, `tipo_eje` y `tolerancia`. `veredicto` es un factor ordenado con niveles `identico < equivalente < materialmente_distinto`. Los atributos `campos_no_comparables`, `detalle_campos_no_comparables`, `columnas_no_comparables`, `cobertura_diagnosticos`, `campos_protegidos` y `resumen` declaran, respectivamente, los campos omitidos, los motivos estructurales de esos campos, las columnas presentes en un solo lado o con tipos incompatibles, los diagnósticos que no se pudieron evaluar, los campos omitidos por protección y el conteo de cada veredicto. Un campo sin valor medible en ambos lados, o con valor en un solo lado, queda en la tabla con veredicto `no_comparable` y no suma como acuerdo. `campos_protegidos` es un data frame con las columnas `columna`, `campo` y `lado`; este último toma los valores `anterior` y `actual`.

See Also

`comparar_perfiles()`, `perfilar()`, `perfilar_dbi()`

Examples

```
anterior <- data.frame(
  columna = "monto", media = 10, minimo = 1, moda = "a",
  stringsAsFactors = FALSE
)
actual <- data.frame(
  columna = "monto", media = 10.00000000001, minimo = 2, moda = "b",
```

```
  stringsAsFactors = FALSE
)
comparar_equivalencia(anterior, actual, tolerancia = 1e-9)
```

`comparar_evaluaciones` *Comparar evaluaciones de perfil*

Description

Calcula el cambio de `EvaluacionPerfil` entre dos corridas. Cada objeto debe contener una sola `id_medicion`; no persiste los resultados. Para una serie de N corridas use `historico_calidad()` y `detectar_deriva_calidad()`.

Usage

```
comparar_evaluaciones(anterior, actual)
```

Arguments

`anterior`, `actual`
: Objetos creados por `evaluar()`.

Value

Data frame con resultados anterior y actual, `delta` y `comparacion`. El delta es el que publica `detectar_deriva_calidad()` sobre el histórico de las dos evaluaciones, con la misma regla: queda en `NA` cuando las corridas miden tablas distintas, cambia el marco o los tipos de resultado, cambia la parte medida de la frontera o un resultado no se evaluó, y `comparacion` dice por qué con la palabra de la deriva. El orden lo dan los argumentos, no la fecha: `anterior` es siempre el primero.

Examples

```
nucleo <- metricas_nucleo()
instancia <- instanciar(especializar(nucleo$NoNulo), "personas", "edad")
perfil <- perfil_evaluacion(
  "Completa", regla_evaluacion("Al menos 90%", function(x) x >= 0.9)
)
medir_en <- function(edades, id, fecha) {
  medir(modelo(instancia), data.frame(edad = edades), id_medicion = id,
        fecha = as.POSIXct(fecha, tz = "UTC"))
}
enero <- evaluar(medir_en(c(20, NA, 35), "enero", "2026-01-31"), perfil)
febrero <- evaluar(medir_en(c(20, 41, 35), "febrero", "2026-02-28"), perfil)
comparar_evaluaciones(enero, febrero)
```

`comparar_perfiles` *Comparar dos perfiles y detectar deriva estructural*

Description

Compara entregas sin exigir que tengan las mismas columnas. Devuelve cambios de esquema, tipos, faltantes, cardinalidad, rango, patrones y hallazgos como un objeto de datos filtrable.

Usage

```
comparar_perfiles(anterior, actual, umbral_cambio = 0.05, umbral_error = 0.2)
```

Arguments

`anterior`, `actual`	Objetos producidos por `perfilar()`.
`umbral_cambio`	Diferencia mínima para considerar significativo un cambio de proporción, cardinalidad o rango relativo. Cinco puntos porcentuales evita elevar variaciones pequeñas a hallazgo.
`umbral_error`	Diferencia a partir de la cual un aumento de faltantes o un cambio de patrón —uno que aparece, desaparece o cambia de proporción— se clasifica como `error`.

Details

Los patrones se comparan sobre el resumen acotado que conserva cada perfil, no sobre los valores originales ni una distribución completa. Si las dos corridas usaron configuraciones de patrones diferentes, se informa un error de comparabilidad y esa parte de la comparación se omite.

Un perfil DBI con `meta$origen_dbi$muestreo` o con `meta$alcance$universo_id = "muestra_motor"` describe una muestra de la tabla fuente. Si uno de los perfiles es de ese tipo, las filas se comparan con `filas_totales_fuente` cuando está disponible, pero rangos, cardinalidades, patrones y hallazgos no se interpretan como cambios de la tabla completa. La proporción de faltantes sí se estima sobre las filas observadas y se compara; los extremos, las cardinalidades y los conteos absolutos que dependen del universo completo quedan fuera. La salida agrega una fila `muestra / no_comparable` y deja fuera esos aspectos; una muestra distinta no se convierte en un cambio ni en un hallazgo resuelto.

Los nombres de columnas y las configuraciones que contienen texto del usuario se comparan por sus bytes, no por la marca de codificación ni por la intercalación del locale.

Eso alcanza mientras los bytes lleguen intactos, y hay un caso en que no llegan y **no depende de este paquete**: el formato RDS 3 —el de `saveRDS()` por omisión— anota la codificación nativa de quien escribe, y al leer *traduce* desde ella el texto que no declara la suya. Un perfil guardado bajo `LC_CTYPE=C` y releído bajo otro locale puede volver con el texto cambiado; en Windows el cambio es silencioso, porque ahí la conversión no falla.

Cuando eso pasa, el perfil releído **está** alterado, y conviene saber qué alcanza a ver esta función. Compara las **propiedades medidas** de cada columna, los hallazgos, el alcance del resumen y las

configuraciones declaradas —no los valores uno por uno—: la columna `aspecto` del resultado nombra, fila por fila, qué se comparó, y es la enumeración viva. **Un cambio de texto que no altera ninguna propiedad medida no aparece**: por ejemplo, si la moda de una columna pasa de `"Basico"` a `"Casico"` —mismo largo, misma forma de patrón, mismos conteos—, la comparación devuelve cero filas. Por eso el remedio ante una relectura sospechosa no es confiar en la comparación para detectarla.

El remedio es una línea, `saveRDS(perfil, archivo, version = 2)`, porque ese formato no anota codificación nativa y por lo tanto no traduce nada. La persistencia propia del paquete —`guardar_historico()` y `guardar_analisis()`— ya lo usa. Las claves internas que agrupan hallazgos también se normalizan por bytes; por eso dos perfiles idénticos no emiten avisos espurios bajo C.

Que la **moda** no se compare es una **decisión**, no un pendiente: la moda es un valor, no una propiedad medida, y cambia con cualquier corrimiento normal de los datos, así que compararla llenaría la deriva de filas que no son deterioro. Quien necesite detectar una relectura corrompida tiene el remedio de una línea que está dos párrafos más abajo, que la evita en lugar de detectarla.

Entre las propiedades comparadas está la **unidad** en la que la columna publica sus cifras, con la misma severidad que un cambio de tipo: dos entregas con los mismos valores y la unidad cambiada de `m` a `km` son un cambio real —la cifra dejó de significar lo mismo— y la comparación lo informa en una fila. Un perfil guardado por una versión anterior del paquete, que no registra la unidad, no hace mentir la comparación: esa parte se declara **no comparable**, como el resto de los campos ausentes.

El **rango** se compara por los valores de sus extremos, no por su texto impreso: un cambio chico sale como fila `ok` y uno que alcanza `umbral_cambio` —medido contra el ancho del rango anterior— como `sospechoso`, sin que `options(scipen)` ni `digits` muevan el veredicto. Cuando el perfil conserva los extremos exactos de un entero que no entra en un doble —un `integer64` por encima de 2^53, o un entero escrito como texto—, esos extremos se comparan con aritmética exacta. Los números se publican con la escritura más corta que vuelve al valor, así que dos extremos distintos nunca se imprimen iguales; la marca decimal sigue a `OutDec`, y con coma decimal los extremos se separan con punto y coma (`[1,5; 3]`).

Una columna **protegida** no publica su rango, y la comparación tampoco lo compara: lo declara en el atributo `campos_protegidos`, con la misma regla que `comparar_equivalencia()`. Si la protección de una columna cambia entre las corridas con la misma política —por ejemplo, porque la proporción de documentos válidos cruzó el umbral de verificación—, el cambio se informa con una fila propia, `aspecto = "proteccion"` y severidad `sospechoso`, en lugar de un cambio de rango que no ocurrió en los datos. Si lo que cambió es la política entera, ya lo dice la fila `configuracion_proteccion` y no se repite por columna.

Además de los patrones que aparecen y desaparecen, se informa el **cambio de proporción** de un patrón que sigue en el resumen cuando alcanza `umbral_cambio`: `sospechoso`, o `error` desde `umbral_error`, en las dos direcciones. Que los cambios menores no generen fila es una **decisión**, a diferencia de los faltantes: serían una fila por patrón en cada entrega. Si el resumen de un lado estaba **lleno** —la columna tenía más formas que `max_patrones`—, un patrón que falta en él pudo quedar por debajo del corte, y no se afirma que apareció o desapareció: la fila dice `no_comparable`, con severidad `ok`, salvo que el cambio que sí se puede garantizar —contra la menor proporción que guardaba ese resumen— alcance el umbral; entonces dice `modificado` y `delta` es esa cota.

La **cardinalidad** no tiene una dirección buena: que caiga la tasa de valores distintos de un identificador es un deterioro, y que suba la de una columna de categorías también. Por **decisión**, un cambio significativo de cardinalidad es `sospechoso` en los dos sentidos. Los faltantes sí la tienen: que bajen es una mejora y queda `ok`.

Las columnas que aparecen o desaparecen generan cambios estructurales de severidad `error`, pero no impiden comparar las columnas compartidas. Un hallazgo de una columna retirada no se presenta como resuelto.

Un hallazgo que ya no aparece se informa como `resuelto` sólo si el diagnóstico volvió a evaluarse. Si el perfil nuevo lo declinó —y lo dice en su `cobertura_diagnosticos`—, el cambio se informa como `no_evaluado` con severidad `sospechoso`, porque no se sabe si el hallazgo sigue: dejar de mirar no es lo mismo que arreglar. Lo mismo si el perfil nuevo apagó el diagnóstico con su argumento o retiró la declaración que lo habilita, cosa que se lee en `meta`: `casi_duplicados_vocabulario = FALSE`, `ausencia_estructural = FALSE`, `variantes_equifrecuentes_vocabulario = FALSE`, una `clave` retirada o declarada sobre otras columnas, una columna retirada de `columnas_sin_ceros`, `columnas_no_negativas`, `columnas_personales` o `aplicabilidad`, y `duplicados_aproximados` apagado o corrido sobre otras columnas —lo que corrió queda en `perfil$duplicados_aproximados`; si corrió sobre las mismas columnas con otros parámetros, la fila `configuracion_duplicados_aproximados` lo declara—; la descripción nombra el argumento. Al revés, un hallazgo que aparece porque el perfil nuevo encendió o declaró lo que lo habilita sigue saliendo `aparecido`, y su descripción dice que el anterior no lo evaluaba. Un hallazgo de varias columnas —un grupo de columnas duplicadas, una relación entre columnas— se compara por el contenido de sus columnas, no por sus posiciones en la tabla: reordenar columnas no hace aparecer ni desaparecer un grupo. Sólo no se presenta como resuelto si alguna de sus columnas desapareció. Un grupo de columnas duplicadas que se separa sólo porque una de sus columnas cambió de tipo declarado —un entero que llega como doble— se publica `no_evaluado`, con severidad `sospechoso`: el diagnóstico sólo iguala columnas del mismo tipo y no volvió a compararlas, salvo que el perfil nuevo ya muestre valores distintos (otros faltantes, otra cantidad de distintos, otro mínimo o máximo); y uno que aparece sólo por un cambio de tipo sale `aparecido` con esa causa en la descripción. Si el tope de un diagnóstico deja un hallazgo fuera del alcance nuevo, se publica `no_evaluado` con el motivo de cobertura. La proximidad de vocabulario también puede dejar pares con frecuencias parejas bajo su piso de asimetría sin emitir un hallazgo; esa cobertura parcial se conserva como `no_evaluado` al comparar, aunque el diagnóstico haya recorrido todos sus valores. Cada argumento de `perfilar()` tiene su lugar en la comparación: habilita un diagnóstico, es una vara —su cambio se declara en la fila `configuracion_umbrales`, que incluye los umbrales de las relaciones de orden y aritméticas entre columnas, los del vocabulario y `datos_personales_permitidos`, `validadores_personales`, `umbral_documento_verificado` y `muestra_validadores`—, tiene su propia fila de configuración, acota el trabajo y lo declara en `cobertura_diagnosticos`, o no cambia los hallazgos. Una vara que uno de los dos perfiles no registra —un perfil de una versión anterior— no se compara.

Los hallazgos que cuentan filas de la tabla y no pertenecen a una columna concreta tampoco se declaran resueltos si aparecieron o desaparecieron columnas entre las corridas: cambio el universo que define esas filas. En ese caso se publica `no_comparable`, con severidad `error`, y se nombran las columnas cambiadas. Con las mismas columnas, el veredicto `resuelto` se conserva.

Value

Data frame `deriva_perfil`. `severidad` usa el factor ordenado `ok < sospechoso < error`; los cambios menores permanecen como filas `ok` para que la serie sea exportable sin ocultar diferencias. El atributo `campos_protegidos` declara lo que no se comparó porque la protección de datos personales lo tapa: un data frame con `columna`, `campo` y `lado` —`anterior` o `actual`—, la misma forma que usa `comparar_equivalencia()`. El único campo que puede figurar es `rango`. Está siempre, vacío si no hay nada que declarar, y `rbind()` une los de todas las derivas.

See Also

perfilar(), detectar_deriva_calidad(), reportar(), comparar_equivalencia()

Examples

```
anterior <- perfilar(data.frame(codigo = c("AA1", "AA2")),
                     fecha = as.POSIXct("2026-01-01", tz = "UTC"))
actual <- perfilar(data.frame(codigo = c("AA1", "B-2"), nueva = 1:2),
                   fecha = as.POSIXct("2026-02-01", tz = "UTC"))
comparar_perfiles(anterior, actual)
```

---

contratos_medicion *Declarar vigencia y escala de medición*

Description

vigencia() reúne el contrato temporal que las métricas de actualidad y oportunidad no pueden inferir de los datos: columna de actualización, fecha de acceso, último cambio conocido, fecha límite, intervalo y frecuencia esperada. Cada métrica valida los campos que necesita y se abstiene si faltan: declara en cobertura_metricas cuál falta y cómo declararlo, en lugar de abortar la medición de las demás métricas.

Usage

```
vigencia(
  columna_actualizacion,
  fecha_acceso = Sys.time(),
  fecha_ultimo_cambio = NULL,
  fecha_limite = NULL,
  inicio_intervalo = NULL,
  fin_intervalo = NULL,
  frecuencia_cambio = NULL
)

escala(error, tipo = c("absoluto", "relativo"))
```

Arguments

columna_actualizacion
: Nombre de la columna Date o POSIXt que registra la última actualización de cada fila.

fecha_acceso
: Momento de acceso usado para estimar actualidad. De la misma clase que la columna de actualización: con una columna Date, Sys.Date() en lugar del Sys.time() por omisión.

fecha_ultimo_cambio
: Fecha conocida del último cambio en el mundo real; puede ser escalar o tener una entrada por fila.

`fecha_limite` Fecha límite escalar o por fila para oportunidad.

`inicio_intervalo, fin_intervalo`
Extremos del intervalo de vigencia.

`frecuencia_cambio`
Frecuencia esperada como `difftime` o número de días.

`error` Error no negativo escalar, vectorial o función del valor.

`tipo` Interpretación `"absoluto"` o `"relativo"` del error.

Details

Las dos puntas de cada comparación tienen que ser de la **misma clase**. Un `Date` se ancla a la medianoche UTC y un `POSIXct` vale por su instante, así que mezclarlos puede hacer contar como tarde una actualización del día del límite, y si el `POSIXct` se leyó sin huso el resultado depende de la sesión. Cuando una métrica compara clases distintas lo avisa. Como `fecha_acceso` vale `Sys.time()` por omisión, con una columna `Date` conviene declararlo también como `Date`: `fecha_acceso = Sys.Date()`.

`escala()` declara el error de un instrumento o de otra escala experta. Con error absoluto, `Escala` calcula `1 - error / abs(valor)` y acota el resultado a `[0, 1]`; con error relativo calcula `1 - error`. No se aprende el error de la distribución observada.

Value

`vigencia()` devuelve un objeto `vigencia_datos`; `escala()` devuelve un objeto `escala_medicion`. Ambos son contratos de configuración y no examinan datos.

See Also

`metricas_nucleo()`, `especializar()`, `cobertura_analisis()`

Examples

```
contrato <- vigencia(
  "actualizado", fecha_limite = as.Date("2026-02-01"),
  frecuencia_cambio = 30, fecha_acceso = as.Date("2026-02-15")
)
instrumento <- escala(error = 0.5, tipo = "absoluto")
```

---

datos_administrativos *Datos administrativos sintéticos con problemas sembrados*

Description

Conjunto enteramente sintético para demostrar el motor de profiling. Incluye cédulas con formatos heterogéneos, fechas mezcladas, faltantes disfrazados, una columna constante, valores extremos, una fila duplicada y registros con el mismo identificador pero contenidos distintos. No representa personas ni registros de ningún organismo.

Usage

```
datos_administrativos
```

Format

Un data frame con 13 filas y 10 variables:

id_persona Identificador interno, con una repetición contradictoria.

cedula Documento sintético con formatos correctos e incorrectos.

fecha_nacimiento Fechas en varios formatos y un faltante disfrazado.

sexo Categoría sintética con un faltante disfrazado.

ingreso Importes con sentinelas numéricos y un valor extremo.

departamento Categoría administrativa.

pais Columna constante.

correo Direcciones sintéticas y un patrón anómalo.

id_copia Copia redundante del identificador interno.

id_tramite Identificador sintético de alta unicidad.

Source

Generación sintética incluida con el paquete.

See Also

`perfilar()`, `planificar_limpieza()`

Examples

```
data(datos_administrativos)
perfil <- perfilar(datos_administrativos, analizar_dependencias = FALSE)
perfil$columnas[, c("columna", "tipo_inferido", "prop_faltantes_totales")]
```

`datos_operativos` *Datos operativos sintéticos y neutrales*

Description

Conjunto enteramente sintético, sin vocabulario ni convenciones de un país, para demostrar el recorrido general de `lupa`. Siembra los mismos tipos de problemas que el ejemplo administrativo: formatos heterogéneos, fechas mezcladas, ausentes disfrazados, espacios y mayúsculas inconsistentes, valores extremos, una columna constante, una columna duplicada, una fila duplicada y un identificador repetido con datos contradictorios. No representa personas, operaciones ni registros reales.

Usage

```
datos_operativos
```

Format

Un data frame con 13 filas y 10 variables:

id_registro Identificador interno, con una repetición contradictoria.

codigo_usuario Código sintético con formatos heterogéneos y un ausente disfrazado.

fecha_evento Fechas en varios formatos y un ausente disfrazado.

canal Categoría con variantes de capitalización, espacios y ausentes disfrazados.

monto Importes con sentinelas, cero, un negativo y un valor extremo.

zona Categoría operativa ficticia.

sistema Columna constante.

contacto Direcciones sintéticas y un patrón anómalo.

id_copia Copia redundante del identificador interno.

id_evento Identificador sintético de alta unicidad.

Source

Generación sintética incluida en `data-raw/datos_operativos.R`.

See Also

datos_administrativos, `analizar()`, `perfilar()`

Examples

```
data(datos_operativos)
perfil <- perfilar(datos_operativos, analizar_dependencias = FALSE)
perfil$hallazgos[, c("columna", "tipo_hallazgo", "severidad")]
```

`descubrir_patrones` *Descubrir patrones de formato*

Description

Generaliza un vector de texto mediante la convención del *Pattern Finder* de DataCleaner: `9` representa un dígito, `a` una letra minúscula y `A` una letra mayúscula. Los símbolos y espacios se conservan literalmente. Las tres clases valen en cualquier escritura: un dígito arábigo-índico es `9` y una é es `a`, de modo que el patrón no publica letras del valor; una letra sin mayúscula ni minúscula —árabe, CJK— cuenta como `a`.

Usage

```
descubrir_patrones(
  x,
  distinguir_mayusculas = TRUE,
  expandir = FALSE,
  max_patrones = 20,
  na.rm = TRUE,
  muestra = 1e+05,
  umbral_raro = 0.05,
  proteger_datos_personales = FALSE
)
```

Arguments

x	Vector que se convertirá a texto.
distinguir_mayusculas	Si es TRUE, distingue a de A.
expandir	Si es FALSE, colapsa tokens repetidos (9999 a 9+).
max_patrones	Número máximo de patrones que se muestran: un entero positivo, o Inf para mostrarlos todos.
na.rm	Si es TRUE, excluye los valores ausentes.
muestra	Máximo de valores que se analizan.
umbral_raro	Frecuencia máxima de un patrón raro, usada para conservar un resumen acotado de patrones raros para los hallazgos. La frontera es inclusiva: un patrón no dominante con exactamente esa proporción es raro.
proteger_datos_personales	Si los ejemplos se enmascaran cuando la forma de los valores alcanza para clasificarlos como dato personal. FALSE por omisión. perfilar() lo pasa en FALSE porque protege el perfil entero después, respetando ahí la declaración del usuario.

Details

El cálculo aplica reemplazos vectorizados sobre el vector completo. Si el vector supera muestra, usa una muestra sistemática reproducible y registra esa decisión en los atributos del resultado.

Value

Un data frame de clase patrones con patrón, frecuencia, proporción y ejemplos. Los **ejemplos no se publican** cuando la forma de los valores alcanza por sí sola para clasificarlos como dato personal —un correo, por ejemplo, aunque los correos no sean la mayoría del vector—: salen como [valor protegido], los de todos los patrones. Acá llega un vector suelto, sin nombre de columna, así que la vía por nombre no está disponible; un número de ocho dígitos **sí** se publica, porque su forma sola no alcanza para afirmar que es un documento. Los atributos total, analizados, filas_analizadas y muestreado describen el posible muestreo; filas_analizadas es un alias explícito de analizados para mantener el alcance visible junto a otros diagnósticos. n_evaluados es el denominador de proporcion: los valores analizados, sin los ausentes cuando na.rm = TRUE.

resumen_patrones conserva sólo el patrón dominante y hasta seis patrones raros para presentacion; nunca guarda la distribucion completa. patrones_raros_trazabilidad conserva solo los nombres de los patrones raros, hasta 5.000, para que la trazabilidad pueda enumerar filas sin retener frecuencias ni ejemplos. Las proporciones siempre estan en [0, 1]. n_patrones_distintos registra el total antes de truncar la tabla para informar omisiones sin retenerla. n_patrones_raros y n_patrones_raros_trazabilidad registran cuantos patrones raros habia antes de sus respectivos limites. n_filas_patrones_no_dominantes_excluidos registra cuantas filas pertenecen a patrones no dominantes cuya proporcion no es rara y que por eso quedan fuera del hallazgo.

See Also

perfilar(), inferir_tipo(), detectar_formatos_fecha()

Examples

```
descubrir_patrones(
  c("2020-01-31", "2021-12-01", "31/01/2020"),
  expandir = TRUE
)
```

---

detectar_asociaciones *Detectar asociaciones entre columnas*

Description

Calcula Pearson —o Spearman, si se pide— entre numéricas, V de Cramér entre categóricas y eta cuadrado entre una categórica y una numérica. Las medidas se informan en [0, 1]: la correlación usa su valor absoluto. La tabla declara el método, su supuesto, el soporte y el posible muestreo; no presenta significancia estadística.

Usage

```
detectar_asociaciones(
  datos,
  dependencias = NULL,
  umbral = 0.3,
  muestra = 10000,
  max_columnas = 50L,
  max_niveles = 50L,
  max_pares = 500L,
  metodo_numerico = c("pearson", "spearman"),
  sentinelas_numericos = NULL
)
```

Arguments

datos	Tabla que se desea examinar.
dependencias	Resultado opcional de detectar_dependencias().
umbral	Valor mínimo en [0, 1] que se informa.
muestra	Máximo común de filas; Inf desactiva el muestreo.
max_columnas	Máximo de columnas analizables.
max_niveles	Máximo de niveles para tratar una columna como categórica.
max_pares	Máximo de asociaciones devueltas después de ordenar.
metodo_numerico	Medida entre columnas numéricas: "pearson" por omisión, o "spearman" para asociación monótona sobre los rangos.
sentinelas_numericos	Vector de valores declarados como ausencia. La fila de un par que contiene un valor declarado no se usa para medirlo, igual que una fila con NA; por eso n_pares cuenta sólo las filas completas restantes. La lista por omisión no se considera una declaración.

Details

metodo_numerico = "spearman" mide asociación **monótona** sobre los rangos y no supone linealidad, así que reconoce una relación creciente aunque sea curva. Pearson sigue siendo el valor por omisión porque es lo que la mayoría espera de una correlación, y el método elegido viaja en la columna metodo de la salida para que ninguna lectura dependa de recordar cuál se pidió.

Se descartan constantes, fechas, listas, columnas compuestas —matrices o arreglos de más de una dimensión—, categóricas de cardinalidad alta y columnas posteriores a max_columnas antes de construir pares. Las dependencias funcionales exactas recibidas en dependencias no se repiten como asociaciones.

Un par en el que una columna toma un solo valor en las filas completas no se puede medir —ni Cramér ni eta cuadrado están definidos con una sola categoría—, aunque la columna no sea constante en la tabla entera: va a pares_omitidos_medicion y no se publica como asociación cero.

Value

Data frame S3 asociaciones_columnas. Sus atributos declaran filas, columnas y pares examinados, omisiones por dependencia y truncamiento. pares_omitidos_medicion conserva los pares que no se pudieron medir, junto con el motivo y la cantidad de filas completas disponible. Su atributo cobertura_diagnosticos tiene una fila por par omitido. Cuando analizar() integra el resultado, esas filas se agregan a la cobertura del perfil.

See Also

detectar_dependencias(), analizar()

Examples

```
d <- data.frame(x = 1:20, y = 2 * (1:20), grupo = rep(c("A", "B"), 10))
detectar_asociaciones(d, umbral = 0)
```

`detectar_claves` *Detectar claves candidatas*

Description

Busca primero claves simples y luego combinaciones mínimas de dos o tres columnas. No prueba una combinación si ya contiene una clave candidata más pequeña. Una clave exige ausencia de `NA` y unicidad en todas las filas. Además informa columnas simples casi-clave cuando tienen al menos 100 filas, al menos el 90 % de sus valores son distintos y un único valor concentra al menos la mitad de los duplicados excedentes. La concentración evita confundir texto libre de alta cardinalidad, con muchas colisiones dispersas, con una clave dañada. Las variables con rol propuesto `fecha`, incluidas fecha-hora, no se consideran casi-claves. Los faltantes disfrazados —`"SIN DATO"`, `-999`— no cuentan como colisiones: se reconocen con la misma máscara que `perfilar()`, con su política por omisión o con la del `perfil` que se pase, y la casi-clave se mide sin ellos —sus distintos, exactos y normalizados, y sus colisiones—. Una clave sana con treinta documentos en `"SIN DATO"` no es una clave rota: le faltan treinta documentos. `n_filas` sigue siendo el de la tabla. La unicidad normalizada compara cada valor de texto por su clave de bytes, la misma con que `perfilar()` cuenta `n_distintos_normalizados`: un valor con marca `bytes` entra con cada byte alto escrito `\xHH` y no se pliega con el mismo texto marcado UTF-8; `latin1` y UTF-8 del mismo texto entran igual. Los vectores `double` sólo son candidatos si ninguno de sus valores finitos tiene parte fraccionaria. Esto conserva identificadores enteros importados desde archivos de texto y excluye importes, coordenadas y otras medidas. Los vectores `integer64` se tratan como enteros semánticos.

Usage

```
detectar_claves(datos, max_combinacion = 3, normalizar = NULL, perfil = NULL)
```

Arguments

`datos`	Objeto que hereda de `data.frame`.
`max_combinacion`	Máximo de columnas por combinación, entre 1 y 3.
`normalizar`	Perfil de comparación. `NULL` hereda el perfil de `perfil`, pero las claves se siguen descubriendo por identidad exacta.
`perfil`	Perfil producido por `perfilar()` para heredar la comparación y la política de faltantes disfrazados —`sentinelas_numericos` y `cadenas_ausencia`—.

Details

Dos claves simples se marcan como redundantes cuando sus contenidos son idénticos —incluidas clase, atributos, ausencias y representación exacta—, aunque tengan nombres distintos. Las columnas compuestas —matrices o arreglos de más de una dimensión— y las de lista no se interpretan como claves. Los pares también quedan en el atributo `claves_redundantes`.

Una columna cuyo nombre se repite en la tabla no se analiza: una clave se publica por el nombre de sus columnas, y ese nombre no diría cuál de las dos identifica. Se avisa, y los nombres quedan en el atributo `columnas_nombre_repetido`.

Value

Data frame de claves candidatas y casi-claves con las columnas combinadas, cantidad de columnas, marcas de redundancia, `casi_clave`, `unicidad_exacta` y `unicidad_normalizada`. Las columnas de colisiones publican sus valores, frecuencias y la concentración observada. Las claves exactas conservan `casi_clave = FALSE`; una fila con `casi_clave = TRUE` es un diagnóstico que requiere corregir o confirmar, no una clave válida.

See Also

`detectar_dependencias()`, `detectar_relaciones()`

Examples

```
detectar_claves(data.frame(id = 1:4, grupo = c("a", "a", "b", "b")))
```

---

detectar_dependencias *Detectar dependencias funcionales entre columnas*

Description

Busca pares ordenados `determinante -> dependiente`. El cumplimiento es la proporción de filas que coincide con el valor modal del dependiente para cada valor del determinante; por eso una dependencia aproximada señala directamente las filas minoritarias que pueden ser errores de carga.

Usage

```
detectar_dependencias(
  datos,
  umbral = 0.995,
  muestra = 1e+05,
  max_columnas = 100L,
  umbral_casi_constante = 0.95,
  umbral_casi_clave = 0.8,
  incluir_claves = FALSE,
  min_observaciones = 10L,
  max_ejemplos = 5L,
  max_comparaciones = 200000L,
  max_trabajo = 1e+08,
  proteger_datos_personales = FALSE,
  columnas_personales = character(),
  sentinelas_numericos = NULL
)
```

Arguments

`datos`	Tabla que se desea examinar.
`umbral`	Cumplimiento mínimo en `[0, 1]`.
`muestra`	Máximo de filas; `Inf` desactiva el muestreo.
`max_columnas`	Máximo de columnas analizadas.
`umbral_casi_constante`	Proporción modal a partir de la cual un determinante se descarta por casi constante.
`umbral_casi_clave`	Tasa de valores distintos a partir de la cual un determinante se descarta por casi clave, salvo que `incluir_claves` sea verdadero.
`incluir_claves`	Si se incluyen determinantes únicos, que satisfacen dependencias de forma trivial.
`min_observaciones`	Mínimo de pares presentes para informar una dependencia.
`max_ejemplos`	Máximo de contradicciones concretas en `evidencia`.
`max_comparaciones`	Máximo de pares determinante-dependiente que se comparan. `Inf` desactiva el presupuesto.
`max_trabajo`	Máximo de unidades fila-par estimadas. `Inf` desactiva el presupuesto que escala con las filas; se combina con `max_comparaciones` y se aplica el límite más restrictivo.
`proteger_datos_personales`	Si los valores de las columnas clasificadas como dato personal se ocultan en `evidencia`. `FALSE` por omisión; los nombres de columna y los conteos se conservan igual. La clasificación se hace aquí por léxico y por forma —esta función no recibe un perfil—, así que lo que sólo quien conoce los datos sabe personal entra por `columnas_personales`. `perfilar()` la llama con `FALSE` porque protege el perfil entero después.
`columnas_personales`	Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`: nombres de columna, o un vector con nombre donde el nombre es la columna y el valor es el tipo. Sin ella, una columna que el léxico no reconoce —un legajo interno— publica sus valores en `evidencia`. Sólo tiene efecto con `proteger_datos_personales = TRUE`.
`sentinelas_numericos`	Vector de valores declarados como ausencia. Las celdas que contienen un valor declarado no participan en la dependencia, igual que una celda `NA`; por eso `n_evaluados` cuenta sólo las filas con ambas columnas presentes. La lista por omisión no se considera una declaración.

Details

Para evitar resultados vacíos o triviales, se omiten por defecto las claves únicas, los determinantes cuya tasa de valores distintos alcanza `umbral_casi_clave`, los determinantes cuya moda alcanza

umbral_casi_constante y los dependientes constantes. Tampoco se informa un par cuyo determinante no acierta más filas que la moda del dependiente sin agrupar: con un dependiente casi constante —un estado que vale "ACTIVO" en el 99,7 % de las filas— cualquier columna alcanzaría el umbral sin aportar nada. El valor predeterminado de umbral = 0.995 exige que como máximo 5 de cada 1.000 filas contradigan la relación. Los ausentes de cualquiera de las dos columnas no integran el cálculo. El descarte ocurre antes de construir agrupaciones. El valor predeterminado umbral_casi_clave = 0.8 excluye determinantes con menos de 1,25 filas por valor distinto en promedio: aun si cumplen, suelen describir una casi-clave y no una regla reutilizable. Las columnas compuestas —matrices o arreglos de más de una dimensión— no se toman como determinantes ni dependientes: no son valores escalares por fila.

El costo crece aproximadamente como columnas^2 * filas. max_columnas conserva las primeras columnas analizables, max_comparaciones limita el número de pares columna a columna y muestra aplica una única muestra de filas enteras a toda la tabla, de modo que las relaciones entre filas no se rompen. La muestra parte la tabla en tantos tramos consecutivos como filas pide y toma una de cada tramo, en una posición que sale de un hash determinista del número de tramo: es la misma en cada corrida, no usa el generador de números aleatorios ni su estado, y no se alinea con datos periódicos. max_trabajo acota el producto entre pares posibles y filas analizadas. Es un presupuesto estimado, no una medición de segundos: permite que el tope efectivo de pares baje cuando la tabla tiene muchas filas. Los atributos del resultado declaran todos los recortes.

Una muestra conserva cada fila entera, pero no la cantidad de filas por valor: sobre datos agrupados en filas consecutivas, la muestra toma a lo sumo una fila de cada tramo y el determinante parece una clave. Por eso los descartes por casi-clave y casi-constante se verifican sobre la columna completa; si sólo la muestra los produjo, el motivo es casi_clave_solo_en_muestra o casi_constante_solo_en_muestra, la columna sigue fuera —evaluarla sobre esa muestra daría dependencias exactas sin evidencia— y perfilar() lo declara en cobertura_diagnosticos.

Lo que la muestra encuentra se recalcula sobre la tabla completa: cada dependencia publicada sale con el cumplimiento, los grupos, las violaciones y la evidencia de todas las filas, y la que ya no alcanza el criterio no se publica. Una muestra puede mostrar exacta una relación que la tabla contradice —una muestra sistemática, a paso fijo, se alineaba con un panel mensual cuyo tamaño hacía que el paso cayera justo en el período, y tomaba siempre los mismos meses—. El recálculo cuesta una pasada por dependencia publicada y entra en max_trabajo; las que no entran conservan las cifras de la muestra, y perfilar() lo declara.

Value

Data frame de clase dependencias_funcionales, ordenado por cumplimiento y soporte. evidencia cita contradicciones concretas, y **no publica los valores de una columna clasificada como dato personal**: la evidencia de un par que nombra una columna personal sale entera como [evidencia protegida] —lo mismo que en el perfil— y, en las demás, lo que repite un valor de una columna personal de la tabla —una observación que cita la cédula— se tapa con el mismo piso que aplica perfilar(). Los atributos muestreado, filas_analizadas, columnas_analizadas, columnas_omitidas, columnas_descartadas y truncado documentan el alcance efectivo. n_pares_posibles, n_pares_comparados, n_pares_sin_comparar y max_comparaciones documentan el presupuesto de comparaciones. trabajo_estimado, trabajo_comparado, trabajo_sin_comparar, unidad_trabajo y max_trabajo documentan el presupuesto por filas. Con muestra, verificadas_tabla_completa dice, fila por fila, si las cifras se recalcularon sobre la tabla completa, y n_descartadas_tabla_completa cuenta las que la muestra mostraba y la tabla no sostiene. columnas_descartadas es un data frame que explica por qué una columna no se usó como determinante. Una columna cuyo nombre se repite en la tabla no entra en ninguna

dependencia, de ningún lado, y se descarta con el motivo `nombre_repetido`: la fila diría `k -> k` sin decir cuál columna es cuál.

See Also

`detectar_claves()`, `proponer_modelo()`, `perfilar()`

Examples

```
datos <- data.frame(
  codigo = rep(1:3, each = 4),
  descripcion = rep(c("A", "B", "C"), each = 4),
  valor = seq_len(12)
)
detectar_dependencias(datos, min_observaciones = 4)
```

---

detectar_deriva_calidad

Detectar deriva en una serie de evaluaciones

Description

Compara corridas consecutivas, ordenadas por fecha dentro de cada perfil o regla, y marca cambios significativos en la escala `[0, 1]`. Los empates de fecha se resuelven por los bytes de `id_medicion`, de modo que la misma secuencia persiste igual bajo cualquier locale.

Usage

```
detectar_deriva_calidad(historico, nivel = c("perfil", "regla"), umbral = 0.05)
```

Arguments

`historico`	Objeto creado por `historico_calidad()`.
`nivel`	`"perfil"` o `"regla"`.
`umbral`	Cambio absoluto mínimo considerado significativo. El valor predeterminado de `0.05` representa cinco puntos porcentuales: evita tratar como deriva diferencias de redondeo, pero sigue siendo sensible a cambios operativamente visibles.

Value

Data frame `deriva_calidad` con una fila por par de corridas consecutivas. Una mejora significativa conserva severidad `ok`; un deterioro de al menos un umbral es `sospechoso` y uno de al menos dos umbrales es `error`. `identidad_tabla` separa series de tablas distintas y `aspecto` marca el resultado o un cambio de configuracion; `cambio` usa `no_comparable` cuando una configuracion del modelo impide comparar las corridas —cambió el marco o los tipos de resultado, y entonces la fila de resultado no se publica—. Cualquier otro cambio de modelo se declara en su fila `configuracion_modelo` sin esa etiqueta: la comparación se mantiene, y la fila de resultado del mismo par conserva su veredicto. Limitación conocida: una corrida guardada por una versión

anterior del paquete cuya configuración lleva un número que con 15 cifras no vuelve a su valor —`maximo = 1/3`, `0.1 + 0.2`, un umbral calculado con `quantile()`— o un instante con fracción de segundo declara **una vez** esa fila, con severidad `error`, contra la primera corrida nueva: esas corridas guardan el número con 15 cifras (`0.333333333333333`) y el instante cortado al segundo, y las nuevas con las cifras que los devuelven a su valor (`0.3333333333333333`). La fila de resultado de ese par conserva su veredicto, y desde ahí la serie aparea.

Un par que no se puede comparar no recibe veredicto. Si alguna de las dos corridas no evaluó su resultado, `delta`, `cambio_absoluto`, `significativo`, `direccion` y `severidad` quedan en `NA` —no en `estable` ni en `ok`— y `descripcion` nombra de qué lado falta el resultado. Filtrar por `severidad != "ok"` deja fuera esas filas a propósito: no son filas sanas, son filas sin medición, y se encuentran con `is.na(significativo)`. La comparación es entre corridas **consecutivas**: una corrida sin evaluar en medio de la serie deja sin veredicto sus dos pares y la deriva no compara por encima de ella. Para comparar las corridas de los dos lados, quitar la del medio del histórico —`h[h$id_medicion != "B", ]`—. Lo mismo cuando a una de las dos corridas le falta su configuración —una tabla armada a mano, o un CSV exportado sin las columnas `identidad_tabla` y `configuracion_*`—: sin ella no se sabe si las dos miden la misma tabla con el mismo marco, así que `cambio` es `no_comparable`, el veredicto queda en `NA` y el atributo `cobertura_diagnosticos` lo declara.

Un cambio de frontera tampoco se compara. Si las dos corridas no dejan afuera las mismas partes declaradas —el nivel `parte_no_medida` del histórico: una tabla de la colección que entró o salió del número—, sus números no cubren lo mismo: la fila de resultado no se publica y una fila `aspecto = "cobertura_frontera"`, `cambio = "no_comparable"`, lo declara, con las partes sin medir de cada lado en `evidencia`. Dos corridas a las que les falta la misma parte sí se comparan.

Examples

```
nucleo <- metricas_nucleo()
instancia <- instanciar(especializar(nucleo$NoNulo), "personas", "edad")
perfil <- perfil_evaluacion(
  "Completa", regla_evaluacion("Al menos 90%", function(x) x >= 0.9)
)
corrida <- function(edades, id, fecha) {
  medidas <- medir(modelo(instancia), data.frame(edad = edades),
                   id_medicion = id, fecha = as.POSIXct(fecha, tz = "UTC"))
  historico_calidad(medidas, evaluar(medidas, perfil))
}
historico <- rbind(
  corrida(c(20, NA, 35, NA), "enero", "2026-01-31"),
  corrida(c(20, 41, 35, 52), "febrero", "2026-02-28")
)
deriva <- detectar_deriva_calidad(historico)
deriva[, c("perfil", "resultado_anterior", "resultado_actual", "direccion")]
```

`detectar_discordancias` *Detectar filas donde señales redundantes se contradicen*

Description

Recorre las señales declaradas con `senal_redundante()` e informa las filas donde las columnas que deberían codificar el mismo hecho no concuerdan dentro de la ventana declarada.

Usage

```
detectar_discordancias(
  datos,
  senales,
  max_ejemplos = 5L,
  proteger_datos_personales = FALSE,
  columnas_personales = character()
)
```

Arguments

`datos`	Tabla que se desea examinar.
`senales`	Una señal creada por `senal_redundante()` o una lista de señales.
`max_ejemplos`	Máximo de filas concretas que se citan como evidencia: un entero no negativo; `0` no cita ninguna.
`proteger_datos_personales`	Si los valores de las columnas clasificadas como dato personal se enmascaran en `evidencia`. `FALSE` por omisión; el número de fila y el nombre de la columna se conservan igual. La clasificación se hace aquí por léxico y por forma —esta función no recibe un perfil—, así que lo que sólo quien conoce los datos sabe personal entra por `columnas_personales`.
`columnas_personales`	Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`: nombres de columna, o un vector con nombre donde el nombre es la columna y el valor es el tipo. Sin ella, una columna que el léxico no reconoce —un legajo interno— publica sus valores en `evidencia`. Sólo tiene efecto con `proteger_datos_personales = TRUE`.

Details

El valor de este diagnóstico es que **ninguna columna por separado lo muestra**: los tres años de una fila pueden ser todos plausibles y aun así contradecirse entre sí.

Una fila donde alguna de las columnas comparadas está ausente **no se cuenta como discordante ni como concordante**: se excluye del universo evaluado, y `n_evaluadas` lo declara. Ausencia no es desacuerdo.

Una columna de la señal que no tiene un valor comparable por fila —una compuesta (matriz, tabla anidada, valores que no se escriben como texto), una lista de objetos o de geometrías, o una magnitud con unidad (`units`), cuyo número sin la unidad no es su valor— no entra: la fila de la señal sale con `n_evaluadas = 0`, `n_discordantes = NA` y la evidencia dice qué columna y por qué. Con una `transformacion` que la lleve a un valor por fila —por ejemplo `units::drop_units()` después de llevarla a una unidad—, se compara.

Una fila con el mismo valor en todas las columnas concuerda, también si ese valor es infinito: `Inf` contra `Inf` es coincidencia exacta.

Una fila citada no muestra iguales dos valores que el veredicto separó. Si la escritura corriente los junta —dos dobles que coinciden en 15 cifras, dos instantes del mismo segundo, dos textos que se ven iguales con distintos puntos de código—, esa fila los cita con la escritura que los distingue: el doble con las cifras que lo devuelven a su valor, el instante con su fracción y el texto con cada carácter fuera del ASCII imprimible escrito `<U+XXXX>`, con una nota que lo dice.

Value

Data frame con una fila por señal: `senal`, `columnas`, `n_filas`, `n_evaluadas`, `n_discordantes`, `proporcion`, `ventana` y `evidencia`.

`evidencia` cita filas concretas, y **no publica los valores de una columna clasificada como dato personal** —salvo que se pida lo contrario con `proteger_datos_personales = FALSE`—: esos salen como `[valor protegido]`. El número de fila y el nombre de la columna se conservan, que es lo que hace falta para ir a corregirla. Las columnas que no son personales se citan enteras, salvo lo que en ellas repite un valor de una columna personal de la tabla —una observación que cita la cédula—: eso se tapa con el mismo piso que aplica `perfilar()`, aunque la columna personal no esté en la señal.

See Also

`senal_redundante()`, `detectar_dependencias()`, `detectar_relaciones()`

Examples

```
d <- data.frame(
  anio_fiscal = c(2023L, 2023L, 2022L),
  anio_archivo = c(2023L, 2023L, 2023L)
)
detectar_discordancias(d, senal_redundante(c("anio_fiscal", "anio_archivo")))
```

`detectar_duplicados_aproximados`

Detectar pares de filas con similitud aproximada

Description

Compara filas seleccionadas de una tabla con `stringdist` y devuelve pares cuya distancia esta bajo el umbral. El resultado describe similitud, distancia, medida y alcance; nunca afirma que dos filas representen la misma entidad. Por omision se combinan como maximo dos columnas de texto o factores, despues de excluir nombres que parecen identificadores (`id`, `codigo`, `uuid`, entre otros). Si quedan mas de dos columnas, la funcion pide indicar `columnas` explicitamente en vez de mezclar campos que pueden diluir la similitud. La medida predeterminada es Jaro–Winkler (`"jw"`), con `p = 0.1`: Winkler favorece coincidencias con el mismo prefijo y, por eso, premia una errata al final mas que una al comienzo. El umbral predeterminado es `0.10`, que apunta a una precisión prudente y no es una calibración fina. Ambos argumentos se pueden cambiar.

Usage

```
detectar_duplicados_aproximados(
  datos,
  columnas = NULL,
  metodo = "jw",
  umbral = 0.1,
  p = 0.1,
  muestra = Inf,
  max_pares = 50000000L,
  max_resultados = 100L,
  normalizar = NULL,
  perfil = NULL,
  proteger_datos_personales = FALSE,
  bloque = 1000L,
  estrategia = "auto",
  lsh_bandas = 12L,
  lsh_filas = 3L,
  lsh_q = 3L,
  lsh_max_cubeta = 1000L,
  lsh_muestra_estimacion = 400000L,
  presupuesto_pares = Inf,
  bloquear_por = NULL,
  lotes = FALSE,
  tamano_lote = 1000L,
  directorio_lotes = NULL,
  nucleos = getOption("lupa.nucleos", 2L),
  max_largo_valor = .MAX_LARGO_VALOR_CASI_DUPLICADOS
)
```

Arguments

datos	Tabla con una fila por entidad observada.
columnas	Columnas atomicas a combinar. NULL aplica la seleccion automatica descrita arriba; no se incluyen matrices ni listas. **El orden forma parte de la comparación**: los valores de cada fila se miden concatenados en el orden declarado, así que c("nombre", "domicilio") y c("domicilio", "nombre") son dos comparaciones distintas y pueden publicar pares distintos con el mismo umbral —medido: 6 pares contra 10 sobre las mismas cinco filas—. Con NULL, el orden es el que tienen las columnas en datos. El objeto publica en columnas el vector que usó, en ese orden, así que el resultado se puede rehacer; declararlo a mano lo vuelve independiente de cómo estén ordenadas las columnas del archivo. **Cómo se concatena, para poder rehacer la distancia.** Los valores se unen con " | "; dentro de cada valor, cada barra invertida \ se escribe \\ y cada | se escribe \\\|. Los escapes no son cosméticos: sin el de |, c1 = "x | y", c2 = "z" y c1 = "x", c2 = "y | z" producirían la misma cadena y dos filas que **no comparten ningún valor** saldrían exacto_normalizado a distancia 0; el de \ es el que deja leer ese \\\| de una sola manera, y el que distingue un valor declarado bytes del literal de su escape. Con los dos, la distancia publicada de

una comparación de varias columnas se rehace así —y la de una sola, con el mismo `escapar()` y sin pegar—:

```
escapar <- function(v) {
 gsub("|", "\\|", gsub("\\", "\\\\", v, fixed = TRUE), fixed = TRUE)
}
fila <- function(i) paste(escapar(datos$c1[i]), escapar(datos$c2[i]), sep = " | ")
stringdist::stringdist(fila(1), fila(2), method = "jw", p = 0.1)
```

Fuera de esos dos escapes, el texto corriente se compara tal cual. Un valor con marca `bytes` entra con cada byte alto escrito `\xHH` —su clave de bytes; esas barras son del escape y no se duplican—, y un mismo texto en `latin1` y en UTF-8 entra igual. Un doble entra con la escritura más corta que vuelve a su valor: la de `as.character()` cuando sus 15 cifras alcanzan —`0.1` es `"0.1"` y `1e15`, `"1e+15"`— y 16 o 17 cuando no —`1e15 + 1` es `"1000000000000001"` y `0.1 + 0.2`, `"0.30000000000000004"`—, así que dos números que R distingue no se comparan como el mismo texto. La evidencia los cita con esa misma escritura.

Si dos columnas de texto **tienen el mismo nombre**, la comparación no corre: identifica las columnas por nombre y no podría decir cuál compara. No se compara un subconjunto —sacar una columna cambia el significado de la comparación—: el objeto sale sin pares, con el motivo en `razon`.

`metodo` Medida admitida por `stringdist::stringdistmatrix()`. Por defecto, `"jw"`.

`umbral` Distancia maxima para informar un par. Por defecto `0.10`.

`p` Factor de prefijo de Jaro–Winkler, entre 0 y 0.25. Por defecto `0.1`; sólo tiene efecto con `metodo = "jw"`.

`muestra` Máximo de filas candidatas. En el camino exacto queda sujeto a `max_pares`; con LSH, `Inf` usa todas las filas.

`max_pares` Máximo de pares comparados en el camino exacto. Por defecto `50000000`, que permite recorrer exhaustivamente hasta 10.000 filas con el método y el bloque predeterminados; se puede reducir para limitar el tiempo. En LSH el alcance se expresa con candidatos y cubetas, por lo que este límite no se usa para recortar filas; el resultado lo marca explícitamente.

`max_resultados` Maximo de pares devueltos. Por defecto `100`. Se conservan los mas cercanos; entre pares empatados en distancia, el desempate usa el orden canonico de los valores y no la posicion de las filas, de modo que reordenar la tabla no cambia que pares sobreviven. Un corte que cae dentro de un empate deja afuera pares igual de cercanos, y eso se declara en `alcance$corte_en_empate`.

`normalizar` Perfil de comparación. `TRUE` conserva el perfil predeterminado, `FALSE` desactiva sus pasos configurables, `"amplio"` activa puntuación, ligaduras y ancho, y `normalizacion()` permite declarar cada paso. Una lista nombrada puede resolver perfiles por columna. `NULL` hereda el perfil guardado en `perfil`; si no se recibe uno, usa `TRUE`. La normalización cambia sólo la representación usada para comparar, no los datos guardados. El umbral se aplica sobre esa cadena normalizada. La **descomposición canónica no es un paso configurable**: corre también con `FALSE`, y es lo que hace que dos escrituras del mismo texto —`café` precompuesto y `café` con acento combinante— sean el mismo texto.

Como la distancia se mide sobre esa forma, un acento cuenta como un carácter aparte: con `normalizar = FALSE`, `café` y `cafe` distan `0.04` —y entran en un umbral de `0.1`—, mientras la misma distancia sobre las cadenas tal como se guardaron daría `0.117`. Para reproducir un número publicado hay que descomponer primero.

Hasta dónde llega esa descomposición. La tabla que usa `lupa` cubre el **subconjunto latino** —el mismo límite que declara `normalizacion()`—, así que fuera de él dos escrituras canónicamente equivalentes **no** colapsan: medido, el mismo nombre griego en NFC y en NFD sale `aproximado` a `0.124` donde `café` precompuesto y descompuesto salen `exacto_normalizado` a `0`. No es un descuido de la comparación: es el alcance de la tabla, y se dice acá porque afecta la receta de reproducción. Para un texto fuera del subconjunto latino, el número publicado se rehace **sin** descomponer, sobre los valores tal como están guardados. Si necesita que colapsen, normalice la entrada a una sola forma antes de comparar —por ejemplo con `stringi::stri_trans_nfc()`—. El informe de fusiones sólo se calcula cuando algún paso configurable está activo; con `FALSE` se omite. Si se entrega `perfil`, se reutiliza su informe ya calculado.

`perfil` Perfil de los mismos datos para reutilizar su clasificacion de datos personales y no volver a inferirla.

`proteger_datos_personales`
Si la evidencia de columnas protegidas se reemplaza por `[valor protegido]`. La supresion queda indicada en cada par. `FALSE` por omision.

`bloque` Cantidad de filas por tesela de comparación. Por defecto `1000`; controla la memoria temporal, no el número de pares comparados.

`estrategia` Estrategia de comparación: `"auto"` (por omisión), `"teselas"`, `"muestra"` o `"lsh"`. Con `"auto"`, MinHash/LSH se activa exactamente cuando hay más de 10.000 filas, `muestra = Inf` y `max_pares` es `Inf` o es mayor o igual que 50.000.000. Si falla cualquiera de esas condiciones, se conserva el camino exacto por teselas o el muestreado que corresponda; `"lsh"` lo fuerza.

`lsh_bandas` Número de bandas del esquema LSH. Por defecto, 12.

`lsh_filas` Número de filas de firma por banda. Por defecto, 3.

`lsh_q` Longitud de los q-gramas usados para MinHash. Por defecto, 3.

`lsh_max_cubeta` Umbral a partir del cual una cubeta se considera grande y se procesa por el mismo troceo acotado del camino exhaustivo. No se descartan pares por este umbral; el alcance informa cuántas cubetas y cuántos pares se procesaron de esta forma. Por defecto, 1000.

`lsh_muestra_estimacion`
Cantidad máxima de pares de filas usados para estimar la proporción de candidatos, el tiempo del camino LSH y, si hay `bloquear_por`, la pérdida de candidatos del bloqueo. La muestra es interna, reproducible y su tamaño efectivo queda en `alcance`. Por defecto se intentan 400.000 pares.

`presupuesto_pares`
Presupuesto de pares candidatos. Por defecto es `Inf`; si la estimación previa lo supera, una sesión no interactiva aborta antes del recorrido y una interactiva pregunta si se continúa. También limita la comparación exacta: allí el número de pares se conoce antes de empezar.

`bloquear_por` Nombre de una columna declarada por el usuario para restringir la comparación a filas con la misma clave. La clave no tiene significado incorporado en `lupa`; sus tamaños, ausentes y pares que quedan fuera se registran en `alcance`. Los `NA` forman un bloque propio.

`lotes` Si es `TRUE`, procesa la comparación exacta por pares de grupos de filas y guarda cada resultado parcial en RDS. Por omisión es `FALSE`; el camino LSH ya administra sus cubetas en memoria y no admite este modo.

`tamano_lote` Cantidad de filas por grupo de trabajo cuando `lotes` es `TRUE`. Por defecto, `1000`.

`directorio_lotes`
Directorio base elegido por el usuario para los parciales. Si es `NULL`, se crea un subdirectorio dentro de `tempdir()`; nunca se escribe en el directorio de trabajo por omisión. Los parciales no son reanudables y se conservan al terminar para auditoría; si el directorio base fue elegido por el usuario, éste debe eliminarlos cuando ya no los necesite. Su ruta, cantidad y tamaño quedan en `resultado$lotes`.

`nucleos` Cantidad máxima de hilos que `stringdist` puede usar. Por defecto es `getOption("lupa.nucleos", 2L)`; `NULL` usa esa misma opción y un valor mayor que los núcleos disponibles se limita de forma segura. El resultado no depende de esta cantidad, pero el tiempo sí. El valor efectivo queda declarado en `alcance$nucleos_usados`.

`max_largo_valor`
Maximo de caracteres permitido en cada valor de las columnas combinadas, **medido sobre la cadena que de verdad se compara**: las columnas ya unidas y ya normalizadas. Las dos mitades importan. Dos columnas de 9.000 caracteres estan las dos por debajo de un tope de 10.000 y llegan a 18.003 una vez unidas; y la normalizacion `amplio` expande ligaduras tipograficas -la de f-f-l es un solo caracter que se convierte en tres-, asi que un valor puede estar por debajo del tope guardado y por encima del comparado. Por defecto es `10000`, umbral elegido porque la distancia normalizada deja de distinguir de forma estable una diferencia local de muchas diferencias en textos largos. Si una columna supera el tope, la combinacion completa se declara fuera de alcance en `alcance$columnas_excluidas_largo`, y `alcance$largo_maximo` publica el largo comparado; no se recortan valores en silencio. Con el tope en `Inf` no se mide ningun largo y `largo_maximo` vale `NA`, no cero. `Inf` quita el tope.

Details

`tipo_par = "exacto"` indica que los textos guardados son iguales en las columnas comparadas; una celda ausente (`NA`) y una cadena vacía cuentan como el mismo texto vacío, que es como entran a la comparación, así que esa diferencia no se atribuye a la normalización. Dos dobles son iguales si lo son sus valores, no su escritura con 15 cifras. `tipo_par = "exacto_normalizado"` indica que coinciden despues de la normalizacion declarada, aunque los textos guardados difieran. Los restantes son `"aproximado"`. La columna logica `igualo_normalizar` es `TRUE` unicamente para el segundo caso. La clasificacion no depende de que una medida de distancia devuelva cero: por ejemplo, `soundex` puede dar distancia cero para textos distintos. Ningun par demuestra identidad.

Como la normalizacion por omision iguala mayusculas, espacios, acentos y comillas, el objeto conserva esta distincion tanto en `pares` como en los hallazgos. `duplicados_exactos_normalizados` es un tipo de hallazgo propio: no afirma que los valores guardados sean iguales y tampoco oculta la causa de la coincidencia bajo `duplicados_aproximados`.

Una fila cuyos valores en las columnas comparadas son **todos** ausentes, cadena vacía o solo blancos —espacios, tabulaciones, saltos de línea; lo que `perfilar()` cuenta en `n_blancos`— no entra en la comparación, con o sin normalización: no hay texto que medir, y dos filas vacías no son un duplicado sino dos ausencias. `alcance$n_filas_validas` cuenta las que sí entraron —contra `n_filas_total`— y, si no quedan dos, `razon` lo dice. Una cadena vacía **sí** es una clave válida en `bloquear_por`: ahí no se compara su texto, se agrupa por él.

En `alcance`, `n_pares_exactos` cuenta solo textos guardados iguales, `n_pares_exactos_normalizados` cuenta coincidencias producidas por la normalizacion y `n_pares_aproximados` cuenta los pares restantes. Son conteos disjuntos y explican `n_pares_hallados`.

La comparacion usa teselas de `bloque` filas: cada matriz temporal se descarta antes de continuar, por lo que la memoria no crece con el tamaño de la tabla. El recorrido es exacto para las filas seleccionadas. `muestra` y `max_pares` recortan el camino exacto; en el camino LSH, `muestra = Inf` conserva todas las filas y `max_pares` no se aplica a las comparaciones. En ese camino `alcance$limite_pares` es `NA`, mientras `limite_pares_configurado` conserva el valor pedido como referencia y `limite_pares_aplica` permite distinguir ambos casos sin inferirlo. El objeto informa cuantos pares eran posibles, cuantos se compararon, el modo, el tamaño de las teselas o los parámetros LSH y los que quedaron fuera. Solo se muestran `max_resultados` coincidencias; el truncamiento tambien queda declarado.

El recorte conserva los pares mas cercanos y, entre los empatados en distancia, desempata por el **rango canonico del valor** con una clave simetrica. Es la misma regla que gobierna el recorte del vocabulario, y existe por lo mismo: desempatar por posición de fila haría que los pares que sobreviven dependieran de cómo viene ordenado el archivo. Con la clave simétrica, el mismo archivo en cualquier orden de filas conserva los mismos pares.

Lo que ningun orden evita es que un corte dentro de un empate deje afuera pares igual de cercanos, asi que se declara. `alcance` publica `distancia_corte` —la distancia donde cayo el corte—, `n_en_distancia_corte` —cuantos de los conservados la comparten— y `corte_en_empate`, que **no** es `truncado` con otro nombre: vale `FALSE` cuando el corte cae en una distancia unica. Si vale `TRUE`, subir `max_resultados` por encima del empate devuelve todos los pares de esa distancia.

Queda un limite que ningun orden puede sacar: si varias filas tienen el mismo valor en la columna comparada, sus pares empatan tambien en el rango, y ahi el desempate cae en la posicion. Medido: con cuatro filas y solo dos valores distintos, el conjunto de **pares de valores** es identico en los cinco ordenes, pero **cuales instancias de fila** los representan cambia. Es irreducible: dos filas con el mismo valor son indistinguibles en esa columna, y su unica identidad es la posicion, que es justamente lo que varia al reordenar. Si importa que instancia se informa, hace falta una clave que las distinga.

En el camino LSH el conjunto de **candidatos** depende del orden de las filas, porque el vocabulario de q-gramas se numera por orden de primera aparicion y esa numeracion alimenta las firmas. Eso ocurre dentro de la garantia declarada en `lsh_garantia_jaccard_*`: medido sobre 1.200 filas, de los pares que cambian al barajar ninguno supera un Jaccard de q-gramas de 0,8, donde el recall declarado es 0,9998. Conviene saberlo antes de comparar dos corridas sobre el mismo contenido exportado en distinto orden.

`stringdist` es una dependencia opcional. Si no esta instalado, la funcion devuelve un objeto con `disponible = FALSE`, una tabla vacia y el motivo explicito; no falla ni presenta silencio como si se hubieran comparado todos los pares. En ese caso —y cuando la comparación no corre por un valor que supera `max_largo_valor`, por dos columnas con el mismo nombre o por no haber columnas comparables— los conteos de `alcance` que salen de comparar filas (`n_filas_validas`,

n_pares_hallados, n_pares_exactos, n_pares_exactos_normalizados y n_pares_aproximados) son NA, no cero: nadie los midió. Si las filas se recorrieron y no quedaron dos válidas, cero pares sí es lo medido.

Las comparaciones que delegan en stringdist usan nucleos hilos como máximo. El valor por omisión es getOption("lupa.nucleos", 2L), se limita a los núcleos disponibles y se publica como alcance$nucleos_usados. Cambiar la cantidad de hilos no cambia los pares ni los hallazgos, aunque sí puede cambiar el tiempo de ejecución. p es el factor de prefijo de Jaro–Winkler y admite valores entre 0 y 0.25; sólo tiene efecto cuando metodo = "jw". Para los demás métodos se valida pero stringdist no lo utiliza. El aviso interactivo señala esta perilla: subir nucleos puede acortar la etapa de comparación, pero la ganancia depende de la máquina y de los datos.

estrategia = "auto" usa MinHash con bandas LSH sobre todas las filas exactamente cuando se cumplen estas tres condiciones: hay más de 10.000 filas, muestra = Inf y max_pares es Inf o es mayor o igual que 50.000.000. Si falla cualquiera de ellas, conserva el camino exacto por teselas o el muestreado que corresponda. La salida declara las bandas, las filas por banda, las cubetas descartadas y la probabilidad teorica de colision. Esa probabilidad se refiere al Jaccard de los q-gramas, no a la medida final (metodo). estrategia = "lsh" fuerza este camino; "teselas" y "muestra" fuerzan el camino exacto por teselas o el muestreado. Las cubetas que superan lsh_max_cubeta se procesan igualmente por lotes; el parámetro identifica cubetas potencialmente costosas, pero no descarta sus pares. El alcance separa lsh_teselas_cubetas_grandes (matrices del primer tramo) de lsh_lotes_cubetas_grandes (lotes de filas de las bandas posteriores), pues no son la misma unidad. Los pares aceptados por ambos recorridos entran al resumen de Jaccard. Si alguna implementación futura descarta una cubeta, la garantía se devuelve como NA y lsh_garantia_estado lo deja explícito. La familia MinHash usa un hash entero biyectivo y determinista del vocabulario, con una semilla distinta por función de hash. No usa el generador de números aleatorios de R ni toca su estado: el resultado no depende de set.seed() ni de RNGkind(), y la muestra de la estimación sale del mismo hash. La tabla de consulta de cada hash evita repetir trabajo para cada celda de la matriz de q-gramas y hace que el resultado no dependa de cómo se numeraron esos q-gramas. Antes de recorrer las bandas se toma una muestra interna y se publica una estimación reproducible de candidatos. El cronómetro de la medida se ejecuta durante al menos 50 ms y queda en resultado$estimacion, separado de alcance y marcado como no determinista. Es una referencia de la medida aislada y no estima la firma, las cubetas ni el troceo; sólo se muestra como aviso en una sesión interactiva. Fuera de una sesión interactiva se señala una condición silenciosa de clase lupa_tiempo_lsh, sólo en el camino LSH: no se imprime en stdout ni stderr. Se puede capturar con withCallingHandlers(resultado <- detectar_duplicados_aproximados(...), lupa_tiempo_lsh = function(c) { ... }). Como hereda de message, un tryCatch(..., message = ...) puede interrumpir la corrida y devolver el valor del manejador; para observar sin interrumpir use withCallingHandlers y, si corresponde, invokeRestart("muffleMessage"). presupuesto_pares permite rechazar el recorrido antes de iniciarlo; sólo se pregunta en una sesión interactiva. El diagnóstico de Jaccard puede quedar limitado a los primeros pares del recorrido determinista; lsh_jaccard_alcance lo dice literalmente y no presenta ese prefijo como una muestra representativa. Con bloquear_por, alcance separa los pares estructuralmente fuera del bloqueo de una estimación de los candidatos que se habrían informado y que quedaron fuera según una muestra determinista. La segunda cantidad es una estimación, no una cuenta exacta.

Value

Lista de clase duplicados_aproximados con pares, hallazgos, alcance, columnas, metodo, umbral, disponible, razon y estimacion. alcance es reproducible; en el camino LSH, estimacion$tiempo_determinis es FALSE y reúne la velocidad, duración y tiempo de referencia medidos en esa corrida. El campo estimacion$tiempo_estimado_etapa indica que esa referencia cubre sólo la comparación stringdist, no la firma, las cubetas ni el troceo. En el camino exacto o cuando no se puede comparar, estimacion es NULL. Si lotes = TRUE, se agrega lotes con el directorio, los archivos RDS, sus tamaños, el estado de completitud y reanudable = FALSE. El loteo cruza todos los grupos, por lo que no pierde pares; su resultado de pares, hallazgos y alcance es el mismo que el recorrido exacto sin lotes. Cada fila de hallazgos declara n_evaluados, n_afectados y unidad_conteo; los conteos desconocidos son NA, nunca cero.

References

Broder, A. Z. (1997) doi:10.1109/SEQUEN.1997.666900. On the resemblance and containment of documents. En *Compression and Complexity of Sequences*, 21–29. Leskovec, J., Rajaraman, A. y Ullman, J. D. (2020) doi:10.1017/9781108684163. *Mining of Massive Datasets* (3.ª ed.), capítulo 3.

See Also

perfilar(), reportar(), planificar_limpieza()

Examples

```
datos <- data.frame(
  nombre = c("Ana Perez", "Ana Peres", "Luis Diaz"),
  domicilio = c("Calle 1", "Calle 1", "Calle 9")
)
pares <- detectar_duplicados_aproximados(datos)
if (!pares$disponible) pares$razon
```

---

detectar_formatos_fecha

Detectar formatos de fecha

Description

Reconoce formatos de fecha y fecha-hora sin escoger arbitrariamente entre día/mes y mes/día. Cuando todos los valores con barra, guion o punto son ambiguos, devuelve todas sus lecturas posibles con estado "candidato": día/mes y mes/día y, con guion y dos dígitos en cada parte (01-02-03), también año/mes/día. Esas lecturas son una sola representación. El atributo formatos_mixtos indica si hay evidencia de dos o más representaciones en la columna. El nombre May es a la vez el mes completo en inglés y la abreviatura de mayo: se cuenta como %B o %b según cómo escriba el mes el resto de la columna, y no como una representación aparte. Se aceptan días y meses con uno o dos dígitos. Los años de dos dígitos se detectan, pero siempre quedan como candidatos y se señalan en la columna anio_dos_digitos: el siglo no se interpreta en silencio. También reconoce

meses escritos en español rioplatense (`setiembre` y `set`) y en inglés, con la estructura completa de una fecha o como mes y año. La tabla interna de nombres no usa `LC_TIME`, por lo que el resultado es independiente del locale del proceso; los nombres de mes dentro de una oración no se reconocen. Un mes escrito desambigua día/mes, pero un año de dos dígitos sigue siendo candidato. Los formatos escritos de mes y año, igual que las fechas compactas, exigen un año entre 1800 y 2100. El formato compacto `%Y%m%d` exige un año entre 1800 y 2100 para evitar que identificadores de ocho dígitos se clasifiquen parcialmente como fechas. Una hora con segundo 60 no se acepta: R la leería como el minuto siguiente. Tampoco la hora `24:00`: R la leería como la medianoche del día siguiente, y el perfil publicaría una fecha que nadie escribió. El separador `T` y la zona `Z` de una fecha-hora se reconocen también en minúscula (`2020-01-31t23:30:00z`), como lo admite RFC 3339 (sección 5.6). Un `POSIXlt` se lee como el `POSIXct` del mismo instante: el atributo `n_filas_fecha_civil_distinta_utc` cuenta igual en las dos clases.

Usage

```
detectar_formatos_fecha(x, muestra = 1e+05)
```

Arguments

`x`	Vector de texto, fechas o fechas-hora.
`muestra`	Máximo de valores que se analizan.

Value

Data frame con formato, frecuencia, proporción, estado, granularidad (`"dia"` o `"mes"`) y conteos de casos inequívocos y ambiguos. Los atributos informan el muestreo, la cantidad de valores compatibles y la presencia de formatos mixtos.

See Also

`inferir_tipo()`, `perfilar()`

Examples

```
detectar_formatos_fecha(c("2020-01-31", "31/01/2020"))
detectar_formatos_fecha(c("01/02/2020", "02/03/2020"))
```

`detectar_relaciones` *Detectar relaciones entre dos tablas*

Description

Examina los pares de columnas declarados y describe su cardinalidad a partir de la unicidad completa de cada lado. La cobertura `tabla1_en_tabla2` es la proporción de valores no ausentes de la primera columna que existe en la segunda; la cobertura inversa se informa de forma simétrica. Así se puede escoger la dirección PK/FK sin imponerla de antemano.

Usage

```
detectar_relaciones(
  tabla1,
  tabla2,
  muestra = 1e+05,
  columnas_candidatas = NULL,
  umbral_cobertura = 0.9,
  podar = FALSE,
  tope_memoria_mb = Inf,
  .rangos = NULL,
  .rellenos_finales = NULL
)
```

Arguments

tabla1, tabla2	Objetos que heredan de data.frame.
muestra	Se conserva por compatibilidad y se valida como antes, pero ya no limita nada: las coberturas se calculan sobre todas las filas.
columnas_candidatas	Lista de dos vectores de nombres, para tabla1 y tabla2, que declara las columnas que pueden participar. NULL conserva la exploración completa por compatibilidad.
umbral_cobertura	Umbral usado por la poda de cardinalidades imposibles.
podar	Si se aplican las podas que cambiarían lo informado —tipos incompatibles y cardinalidades imposibles—. FALSE por omisión: sólo se aplica la poda cierta, que no cambia ninguna fila.
tope_memoria_mb	Presupuesto de memoria para las filas comparadas, en megabytes. Las combinaciones pendientes se declaran como podas cuando se alcanza; Inf no limita el procesamiento.
.rangos	Uso interno de relaciones_coleccion(). Lista nombrada por tabla1 y tabla2, con rangos y su origen para cada columna. La poda por rangos solo acepta rangos de la columna completa o del universo DBI. Una columna cuya lectura no permite comparar exacto lo declara en no_comparable, y sus pares salen sin_comparar con ese motivo.
.rellenos_finales	Uso interno de relaciones_coleccion(). Declara las columnas cuyo controlador compara ignorando espacios finales. Se aplica solo al par que tiene una columna declarada, para conservar la semantica de comparacion del motor sin convertir todas las columnas de texto.

Details

Los valores se comparan por su identidad. Los números, por su valor; los instantes y las fechas también, con su fracción: dos instantes del mismo segundo no coinciden, y un instante sin fracción se escribe al segundo —2020-01-01 10:00:00— para que un texto con esa escritura lo encuentre.

Un texto con bytes que no son UTF-8 válido se compara por sus bytes, como `%in%`: no es un ausente, y dos valores así distintos no coinciden.

`columnas_candidatas` permite evitar la exploración de columnas que el usuario sabe que no pueden participar. El costo crece con el producto de anchos: dos tablas de treinta columnas son novecientas combinaciones por par de tablas, y declarar cuáles pueden participar es lo que lo vuelve manejable.

Hay dos clases de poda y el paquete no las trata igual. Dos columnas de la misma familia con rangos numéricos disjuntos, medidos sobre la columna completa o sobre el universo DBI, no comparten ningún valor, y eso se sabe sin comparar: la fila sale como siempre —`sin_coincidencias`, con cobertura cero— y la comparación se ahorra. Esa poda está siempre activa porque no cambia lo que el objeto informa.

Las otras dos sí lo cambiarían. Familias distintas parece decisivo y no lo es: una columna de texto puede guardar `"2020-01-05"` y coincidir con una de fecha. Y una cardinalidad imposible no dice que no haya coincidencias, dice que no alcanzan `umbral_cobertura`, que es otra cosa. La cota es por filas, como la cobertura: con `k` valores distintos en `tabla1`, las filas cubiertas de `tabla2` no superan las de sus `k` valores más frecuentes. Por eso van detrás de `podar = TRUE`, y cuando se aplican **el par no desaparece**: sale con `cardinalidad = "sin_comparar"`, coberturas `NA` y su motivo en `motivo_poda`. Un par que no se evaluó no es un par sin relación.

Dos pares no se comparan nunca, con `podar` o sin él, porque la comparación no daría una respuesta sino una falsa. Si una tabla tiene dos columnas con el mismo nombre, el nombre no dice cuál de las dos es: sus pares salen `sin_comparar` con `motivo_poda = "nombre_repetido"`. Y una fecha contra una fecha-hora no comparte escritura —`2020-01-01` contra `2020-01-01 00:00:00`—, así que ningún valor coincidiría aunque fueran los mismos días: el par sale con `motivo_poda = "fecha_contra_instante"`. Convertir una de las dos a la clase de la otra, con `tz` explícito, y volver a comparar. Tampoco entra una columna que no tiene un valor comparable por fila: una compuesta —matriz, tabla anidada, valores que no se escriben como texto, como una `vctrs_rcrd`—, una lista de objetos —de vectores, de geometrías, de `NULL`— o una magnitud con unidad (`units`), cuyo número sin la unidad no es su valor. Sus pares salen `sin_comparar` con `motivo_poda = "columna_no_comparable"`, y el detalle en `podas` dice cuál de las tres es; la función sigue con las demás columnas.

Todas las podas, de las dos clases, quedan además en el atributo `podas` con su motivo y su detalle.

La cardinalidad, la cantidad de valores comunes y las dos coberturas se calculan con ambas columnas completas: la cobertura cuenta las filas de cada valor distinto, así que sale exacta por el costo de una pasada más por columna, sin una muestra que pueda alinearse con los datos.

Value

Data frame con `columna_tabla1`, `columna_tabla2`, `cardinalidad` —`1:1`, `1:m`, `m:1`, `m:m`, `sin_coincidencias` o `sin_comparar`—, `n_valores_comunes`, coberturas de integridad referencial en ambas direcciones y `motivo_poda`, que sólo tiene valor en los pares no comparados. Los atributos `filas_totales`, `filas_analizadas` y `muestreado` documentan el alcance: `filas_analizadas` son todas las filas y `muestreado` vale `FALSE`; `podas`, `n_pares_totales`, `n_pares_comparados` y `n_pares_podados` documentan qué se comparó y qué no.

See Also

`detectar_claves()`, `referencial()`, `proponer_modelo()`

Examples

```
personas <- data.frame(id = 1:3)
tramites <- data.frame(persona_id = c(1, 1, 3, 4))
detectar_relaciones(personas, tramites)
```

`distribucion_valores` *Distribuciones de valores y cuantiles por columna*

Description

Resume frecuencias sin conservar una tabla completa de alta cardinalidad. Cada columna se limita a `max_valores`; `alcance` declara cuántos valores se analizaron, cuántos distintos se observaron y si hubo muestreo o truncamiento. Los cuantiles se calculan sólo para números ordinarios finitos.

Usage

```
distribucion_valores(
  datos,
  perfil = NULL,
  max_valores = 20L,
  probabilidades = c(0, 0.25, 0.5, 0.75, 1),
  muestra = 1e+05,
  proteger_datos_personales = FALSE,
  columnas_personales = character(),
  .universo = NULL
)
```

Arguments

`datos`	Tabla que se desea examinar.
`perfil`	Perfil opcional de los mismos datos; evita repetir la clasificación de posibles datos personales. Si el perfil declaró `columnas_codigo`, esas columnas publican en `cuantiles` sólo los extremos —probabilidad 0 y 1, el tramo de códigos—; los intermedios quedan en `NA` con `estado = "codigo_declarado"`, como `mediana` en el perfil.
`max_valores`	Máximo de valores mostrados por columna.
`probabilidades`	Probabilidades de los cuantiles, en `[0, 1]`.
`muestra`	Máximo de filas por columna; `Inf` desactiva el muestreo.
`proteger_datos_personales`	Si se ocultan valores de columnas cuya clasificación activa protección automática. `FALSE` por omisión; véase `perfilar()`. Con `perfil`, se respeta la clasificación de ese perfil —incluido lo que su usuario declaró—; sin `perfil`, la clasificación se rehace aquí por léxico y por forma, y lo declarado entra por `columnas_personales`.

`columnas_personales`
: Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`: nombres de columna, o un vector con nombre donde el nombre es la columna y el valor es el tipo. **Es el único camino cuando no se pasa** `perfil`: sin perfil, la clasificación corre por léxico y por forma, así que una columna que sólo quien la conoce sabe personal —una edad, un legajo interno— se publicaría entera. Sólo tiene efecto con `proteger_datos_personales = TRUE`.

`.universo`
: Uso interno de `analizar()`: el universo que declara su `argumentos_perfil` —`aplicabilidad`, `columnas_opcionales` y `sentinelas_numericos`—. Con él, lo que queda fuera del universo no entra en `frecuencias` ni en `cuantiles`, y un centinela declarado sale de los `cuantiles` como sale de `minimo`, `mediana` y `maximo` del perfil; en `frecuencias` sigue contando, como en `n_distintos` y `moda`.

Details

Las proporciones de `frecuencias` **se calculan sobre los valores que se pudieron analizar, no sobre la columna entera**, y `alcance` dice cuántos fueron: `n_total` contra `n_analizados`. Cuando alguno no se pudo comparar —bytes inválidos, sobre todo— el `estado` de esa columna es `calculada_parcial` en vez de `calculada`, para que un resultado parcial no se lea como completo. Un `NA` no cuenta como descarte: es una ausencia declarada, no un valor que no se pudo analizar. Los otros estados posibles son `sin_valores`, `sin_valores_analizables` y `tipo_no_comparable`. La entrada es la de `perfilar()`: un `POSIXlt` se cuenta por su instante, como el `POSIXct` equivalente, y el nivel `NA` de un factor es un faltante.

Cuando una columna tiene evidencia suficiente para activar la protección de datos personales, sus frecuencias y niveles se conservan pero el valor concreto se reemplaza. Los cuantiles mantienen sus filas y probabilidades, pero `valor` queda en `NA` y `estado` informa `"valor_protegido"`: un cuantil, especialmente en tablas pequeñas, puede coincidir exactamente con una observación. Esta protección es independiente de la usada al construir el perfil.

Hay clases que R declara numéricas y cuyo cuantil interpolado no se puede calcular, porque su aritmética rechaza el producto por una fracción —`lubridate::Period` es una—. Esas columnas **no interrumpen el análisis**: conservan sus filas con `valor` en `NA` y el `estado` `"no_interpolable:<clase>"`, que nombra la clase que no se pudo interpolar. La condición se mide intentando el cálculo, no enumerando clases, así que una clase nueva con la misma limitación queda cubierta sin tocar el código.

Las frecuencias se cuentan **por valor, no por su escritura**: dos dobles que coinciden en sus primeras 15 cifras —`0.1 + 0.2` y `0.3`, `1e15` y `1e15 + 1`— son dos filas, como los cuenta `perfilar()`, y cada una se escribe con las cifras que la devuelven a su valor. Un instante se escribe en UTC y con el rótulo `UTC`, como los resúmenes de `perfilar()`: la etiqueta no depende de la zona de la sesión. A igual frecuencia, las filas se ordenan por valor —el texto por sus bytes, los números como números—, el mismo desempate que la moda de `perfilar()`: así, el recorte de `max_valores` no depende del orden de las filas de la tabla.

Value

Objeto `distribuciones_perfil`, una lista con data frames `frecuencias`, `cuantiles` y `alcance`. Todas las proporciones están en `[0, 1]`.

See Also

perfilar(), analizar(), clasificar_variables()

Examples

```
d <- data.frame(grupo = c("A", "A", "B"), valor = c(1, 2, 10))
distribucion_valores(d)
```

---

elegir_clave *Elegir la clave entre las sugeridas*

Description

Muestra las candidatas de sugerir_clave() numeradas, con el motivo de cada una, y devuelve la que se elija para pasarla al argumento clave de perfilar(). La última opción permite escribir un nombre que no esté en la lista, o varios separados por coma para una clave compuesta. Si un nombre real contiene una coma, el texto no se puede desambiguar: en ese caso hay que pasar un vector de nombres, por ejemplo c("a,b"), al argumento clave.

Usage

```
elegir_clave(datos, maximo = 5L, umbral_casi = 0.95)
```

Arguments

datos	Tabla a examinar.
maximo	Cuántas sugerencias devolver como máximo: un entero positivo.
umbral_casi	Proporción de valores distintos a partir de la cual una columna que repite se informa igual, como candidata con duplicados.

Details

En una sesión no interactiva no pregunta: devuelve NULL y avisa qué habría ofrecido. Un guion que corre solo no puede quedarse esperando una respuesta, y elegir por su cuenta sería exactamente lo que esta función existe para no hacer.

Value

El nombre —o los nombres— de la clave elegida, o NULL si no se eligió ninguna o la sesión no es interactiva.

See Also

sugerir_clave(), y el argumento clave de perfilar().

Examples

```
personas <- data.frame(id_persona = 1:3, edad = c(30, 41, 25))
# En una sesion interactiva pregunta; aqui devuelve NULL y dice que ofreceria.
elegir_clave(personas)
```

`estadisticos_estimacion` *Catálogo de estadísticos de estimación reconocidos*

Description

Los siete estadísticos que `medicion_desde_estimaciones()` sabe llevar al contrato de `medir()`, con la métrica a la que corresponden, su tipo, su unidad y su **orientación**: si un valor alto es conformidad o defecto. Sin la orientación el número no se puede evaluar, porque un coeficiente de variación de `0,30` y un tamaño de muestra de `0,30` no se leen igual.

Usage

```
estadisticos_estimacion()
```

Value

Data frame con `estadistico`, `metrica`, `tipo_resultado`, `orientacion` y `unidad`.

See Also

`medicion_desde_estimaciones()`, `evaluar()`

Examples

```
estadisticos_estimacion()
```

`estimar_costo` *Estimar el costo de una comparación de duplicados*

Description

Construye las firmas y calcula el pronóstico de candidatos sin recorrer las cubetas ni comparar los pares. Es una operación deliberada: la medición del reloj queda en el resultado de esta función y no es necesaria para obtener `alcance` reproducible en `detectar_duplicados_aproximados()`.

Usage

```
estimar_costo(
  datos,
  columnas = NULL,
  metodo = "jw",
  umbral = 0.1,
  p = 0.1,
  muestra = Inf,
  max_pares = 50000000L,
  max_resultados = 100L,
  normalizar = NULL,
  perfil = NULL,
  proteger_datos_personales = FALSE,
  bloque = 1000L,
  estrategia = "auto",
  lsh_bandas = 12L,
  lsh_filas = 3L,
  lsh_q = 3L,
  lsh_max_cubeta = 1000L,
  lsh_muestra_estimacion = 400000L,
  presupuesto_pares = Inf,
  bloquear_por = NULL,
  lotes = FALSE,
  tamano_lote = 1000L,
  directorio_lotes = NULL,
  nucleos = getOption("lupa.nucleos", 2L),
  max_largo_valor = .MAX_LARGO_VALOR_CASI_DUPLICADOS
)
```

Arguments

`datos`	Tabla con una fila por entidad observada.
`columnas`	Columnas atomicas a combinar. `NULL` aplica la seleccion automatica descrita arriba; no se incluyen matrices ni listas. **El orden forma parte de la comparación**: los valores de cada fila se miden concatenados en el orden declarado, así que `c("nombre", "domicilio")` y `c("domicilio", "nombre")` son dos comparaciones distintas y pueden publicar pares distintos con el mismo umbral —medido: 6 pares contra 10 sobre las mismas cinco filas—. Con `NULL`, el orden es el que tienen las columnas en `datos`. El objeto publica en `columnas` el vector que usó, en ese orden, así que el resultado se puede rehacer; declararlo a mano lo vuelve independiente de cómo estén ordenadas las columnas del archivo. **Cómo se concatena, para poder rehacer la distancia.** Los valores se unen con `" | "`; dentro de cada valor, cada barra invertida `\` se escribe `\\` y cada `|` se escribe `\\\|`. Los escapes no son cosméticos: sin el de `|`, `c1 = "x | y"`, `c2 = "z"` y `c1 = "x"`, `c2 = "y | z"` producirían la misma cadena y dos filas que **no comparten ningún valor** saldrían `exacto_normalizado` a distancia `0`; el de `\` es el que deja leer ese `|` de una sola manera, y el que distingue un valor declarado `bytes` del literal de su escape. Con los dos, la distancia publicada de

una comparación de varias columnas se rehace así —y la de una sola, con el mismo `escapar()` y sin pegar—:

```
escapar <- function(v) {
 gsub("|", "\\|", gsub("\\", "\\\\", v, fixed = TRUE), fixed = TRUE)
}
fila <- function(i) paste(escapar(datos$c1[i]), escapar(datos$c2[i]), sep = " | ")
stringdist::stringdist(fila(1), fila(2), method = "jw", p = 0.1)
```

Fuera de esos dos escapes, el texto corriente se compara tal cual. Un valor con marca `bytes` entra con cada byte alto escrito `\xHH` —su clave de bytes; esas barras son del escape y no se duplican—, y un mismo texto en `latin1` y en UTF-8 entra igual. Un doble entra con la escritura más corta que vuelve a su valor: la de `as.character()` cuando sus 15 cifras alcanzan —`0.1` es `"0.1"` y `1e15`, `"1e+15"`— y 16 o 17 cuando no —`1e15 + 1` es `"1000000000000001"` y `0.1 + 0.2`, `"0.30000000000000004"`—, así que dos números que R distingue no se comparan como el mismo texto. La evidencia los cita con esa misma escritura.

Si dos columnas de texto **tienen el mismo nombre**, la comparación no corre: identifica las columnas por nombre y no podría decir cuál compara. No se compara un subconjunto —sacar una columna cambia el significado de la comparación—: el objeto sale sin pares, con el motivo en `razon`.

`metodo` Medida admitida por `stringdist::stringdistmatrix()`. Por defecto, `"jw"`.

`umbral` Distancia maxima para informar un par. Por defecto `0.10`.

`p` Factor de prefijo de Jaro–Winkler, entre 0 y 0.25. Por defecto `0.1`; sólo tiene efecto con `metodo = "jw"`.

`muestra` Máximo de filas candidatas. En el camino exacto queda sujeto a `max_pares`; con LSH, `Inf` usa todas las filas.

`max_pares` Máximo de pares comparados en el camino exacto. Por defecto `50000000`, que permite recorrer exhaustivamente hasta 10.000 filas con el método y el bloque predeterminados; se puede reducir para limitar el tiempo. En LSH el alcance se expresa con candidatos y cubetas, por lo que este límite no se usa para recortar filas; el resultado lo marca explícitamente.

`max_resultados` Maximo de pares devueltos. Por defecto `100`. Se conservan los mas cercanos; entre pares empatados en distancia, el desempate usa el orden canonico de los valores y no la posicion de las filas, de modo que reordenar la tabla no cambia que pares sobreviven. Un corte que cae dentro de un empate deja afuera pares igual de cercanos, y eso se declara en `alcance$corte_en_empate`.

`normalizar` Perfil de comparación. `TRUE` conserva el perfil predeterminado, `FALSE` desactiva sus pasos configurables, `"amplio"` activa puntuación, ligaduras y ancho, y `normalizacion()` permite declarar cada paso. Una lista nombrada puede resolver perfiles por columna. `NULL` hereda el perfil guardado en `perfil`; si no se recibe uno, usa `TRUE`. La normalización cambia sólo la representación usada para comparar, no los datos guardados. El umbral se aplica sobre esa cadena normalizada. La **descomposición canónica no es un paso configurable**: corre también con `FALSE`, y es lo que hace que dos escrituras del mismo texto —`café` precompuesto y `café` con acento combinante— sean el mismo texto.

Como la distancia se mide sobre esa forma, un acento cuenta como un carácter aparte: con `normalizar = FALSE`, `café` y `cafe` distan `0.04` —y entran en un umbral de `0.1`—, mientras la misma distancia sobre las cadenas tal como se guardaron daría `0.117`. Para reproducir un número publicado hay que descomponer primero.

Hasta dónde llega esa descomposición. La tabla que usa `lupa` cubre el **subconjunto latino** —el mismo límite que declara `normalizacion()`—, así que fuera de él dos escrituras canónicamente equivalentes **no** colapsan: medido, el mismo nombre griego en NFC y en NFD sale `aproximado` a `0.124` donde `café` precompuesto y descompuesto salen `exacto_normalizado` a `0`. No es un descuido de la comparación: es el alcance de la tabla, y se dice acá porque afecta la receta de reproducción. Para un texto fuera del subconjunto latino, el número publicado se rehace **sin** descomponer, sobre los valores tal como están guardados. Si necesita que colapsen, normalice la entrada a una sola forma antes de comparar —por ejemplo con `stringi::stri_trans_nfc()`—. El informe de fusiones sólo se calcula cuando algún paso configurable está activo; con `FALSE` se omite. Si se entrega `perfil`, se reutiliza su informe ya calculado.

`perfil` Perfil de los mismos datos para reutilizar su clasificacion de datos personales y no volver a inferirla.

`proteger_datos_personales`
Si la evidencia de columnas protegidas se reemplaza por `[valor protegido]`. La supresion queda indicada en cada par. `FALSE` por omision.

`bloque` Cantidad de filas por tesela de comparación. Por defecto `1000`; controla la memoria temporal, no el número de pares comparados.

`estrategia` Estrategia de comparación: `"auto"` (por omisión), `"teselas"`, `"muestra"` o `"lsh"`. Con `"auto"`, MinHash/LSH se activa exactamente cuando hay más de 10.000 filas, `muestra = Inf` y `max_pares` es `Inf` o es mayor o igual que 50.000.000. Si falla cualquiera de esas condiciones, se conserva el camino exacto por teselas o el muestreado que corresponda; `"lsh"` lo fuerza.

`lsh_bandas` Número de bandas del esquema LSH. Por defecto, 12.

`lsh_filas` Número de filas de firma por banda. Por defecto, 3.

`lsh_q` Longitud de los q-gramas usados para MinHash. Por defecto, 3.

`lsh_max_cubeta` Umbral a partir del cual una cubeta se considera grande y se procesa por el mismo troceo acotado del camino exhaustivo. No se descartan pares por este umbral; el alcance informa cuántas cubetas y cuántos pares se procesaron de esta forma. Por defecto, 1000.

`lsh_muestra_estimacion`
Cantidad máxima de pares de filas usados para estimar la proporción de candidatos, el tiempo del camino LSH y, si hay `bloquear_por`, la pérdida de candidatos del bloqueo. La muestra es interna, reproducible y su tamaño efectivo queda en `alcance`. Por defecto se intentan 400.000 pares.

`presupuesto_pares`
Se acepta por simetría de la firma y **no aborta aquí**: estimar el costo es justamente lo que se hace antes de fijar un presupuesto, así que interrumpir la estimación por superarlo dejaría sin respuesta la pregunta que motivó la llamada. Quien aborta es `detectar_duplicados_aproximados()`, con el número que devuelve esta función.

`bloquear_por` Nombre de una columna declarada por el usuario para restringir la comparación a filas con la misma clave. La clave no tiene significado incorporado en `lupa`; sus tamaños, ausentes y pares que quedan fuera se registran en `alcance`. Los `NA` forman un bloque propio.

`lotes` Se acepta por simetría de la firma, pero la estimación no escribe parciales ni modifica el directorio indicado.

`tamano_lote` Se acepta por simetría; no cambia el pronóstico.

`directorio_lotes`
Se acepta por simetría y no se crea ni se usa al estimar.

`nucleos` Cantidad máxima de hilos que `stringdist` puede usar. Por defecto es `getOption("lupa.nucleos", 2L)`; `NULL` usa esa misma opción y un valor mayor que los núcleos disponibles se limita de forma segura. El resultado no depende de esta cantidad, pero el tiempo sí. El valor efectivo queda declarado en `alcance$nucleos_usados`.

`max_largo_valor`
Maximo de caracteres permitido en cada valor de las columnas combinadas, **medido sobre la cadena que de verdad se compara**: las columnas ya unidas y ya normalizadas. Las dos mitades importan. Dos columnas de 9.000 caracteres estan las dos por debajo de un tope de 10.000 y llegan a 18.003 una vez unidas; y la normalizacion `amplio` expande ligaduras tipograficas -la de f-f-l es un solo caracter que se convierte en tres-, asi que un valor puede estar por debajo del tope guardado y por encima del comparado. Por defecto es `10000`, umbral elegido porque la distancia normalizada deja de distinguir de forma estable una diferencia local de muchas diferencias en textos largos. Si una columna supera el tope, la combinacion completa se declara fuera de alcance en `alcance$columnas_excluidas_largo`, y `alcance$largo_maximo` publica el largo comparado; no se recortan valores en silencio. Con el tope en `Inf` no se mide ningun largo y `largo_maximo` vale `NA`, no cero. `Inf` quita el tope.

Details

En el camino LSH, `candidatos_previstos` es una estimación reproducible a partir de una muestra de firmas. `tiempo_estimado_segundos` es una referencia de la medida aislada y no incluye firmas, cubetas ni troceo; sus campos de reloj tienen `tiempo_determinista = FALSE`. Con `bloquear_por`, los pares entre bloques no entran en el pronóstico y la pérdida estructural y estimada queda en `alcance`. En el camino exacto, `candidatos_previstos` es la cantidad de pares que se compararán, sujeta a `muestra`, `max_pares` y `bloquear_por`.

La función no escribe archivos ni modifica el estado del generador de R.

Value

Lista de clase `estimacion_costo_lupa` con los campos de la estimación, `alcance`, `disponible` y `razon`. En el camino exacto, `alcance` declara además **la corrida que se está estimando**: `modo_comparacion` es el mismo que publicará `detectar_duplicados_aproximados()` con esos argumentos —`exhaustiva_por_bloques` o `muestreada_por_bloques`—, y `filas_previstas`, `filas_totales` y `estrategia_prevista` dicen sobre cuántas filas se va a comparar y con qué selección. Una estimación que anunciara un recorrido exhaustivo sobre una corrida que va a muestrear no serviría para decidir.

See Also

detectar_duplicados_aproximados()

Examples

```
datos <- data.frame(
  nombre = c("Ana Perez", "Ana Peres", "Luis Diaz"),
  grupo = c("A", "A", "B")
)
if (requireNamespace("stringdist", quietly = TRUE)) {
  costo <- estimar_costo(datos, columnas = "nombre", estrategia = "lsh")
  costo$candidatos_previstos
}
```

---

estimar_costo_coleccion

Estimar el costo de buscar relaciones en una colección

Description

Cuenta cuántas comparaciones de columnas implicaría buscar claves foráneas entre los pares indicados, **antes** de hacerlas. El número que importa no es la cantidad de tablas sino la de pares de columnas: dos tablas de cincuenta columnas son dos mil quinientas comparaciones.

Usage

```
estimar_costo_coleccion(coleccion, pares = NULL, columnas_candidatas = NULL)
```

Arguments

coleccion	Objeto creado por coleccion().
pares	Data frame con columnas tabla_1 y tabla_2. Un data frame con cero filas es una declaración válida de «ningún par». Sin pares —NULL—, todos los pares dirigidos.
columnas_candidatas	Lista nombrada por identificador de tabla. Cada entrada declara las columnas que pueden participar de una relación; las tablas no nombradas conservan todas sus columnas. La declaración reduce tanto la lectura como el número de comparaciones.

Details

Sin pares estima sobre todos los pares dirigidos de la colección, que es justamente lo que suele mostrar por qué hay que declararlos. Ese caso **no materializa los pares**: la suma de c_i * c_j sobre los pares dirigidos es sum(c)^2 - sum(c^2), así que el trabajo es lineal en el número de tablas y no cuadrático. Estimar el costo no puede costar más que medirlo.

Sólo se consulta el ancho de las tablas que los pares nombran, por metadatos (`DBI::dbListFields()`), no leyendo datos.

Una tabla cuyo esquema no se puede leer no se suma como cero. Queda declarada en `tablas_sin_esquema` y los pares que la involucran se cuentan en `n_pares_sin_estimar`: el número devuelto cubre `n_pares_estimados`, no todos los pares.

Value

Lista con `n_tablas`, `n_pares_dirigidos`, `n_pares_declarados`, `n_pares_estimados`, `n_pares_sin_estimar`, `n_comparaciones_columnas`, `n_tablas_sin_esquema`, `tablas_sin_esquema` y `alcance`. También publica `columnas_candidatas` y el alcance real de las comparaciones.

See Also

`relaciones_coleccion()`, `coleccion()`

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "personas", data.frame(id = 1:3, nombre = letters[1:3]))
  DBI::dbWriteTable(con, "visitas", data.frame(persona_id = 1:3, dia = 1:3))
  estimar_costo_coleccion(
    coleccion(con, c("personas", "visitas")),
    columnas_candidatas = list(personas = "id", visitas = "persona_id")
  )
  DBI::dbDisconnect(con)
}
```

`evaluar` *Evaluar medidas, reglas y perfiles*

Description

Ejecuta la cadena formal: condición por medida, proporción de medidas que cumplen cada regla y media aritmética simple, no ponderada, de las reglas del perfil. Los resultados por regla se conservan en `reglas`; el resumen de `perfiles` no sustituye esa distribución.

Usage

```
evaluar(medicion, perfil)
```

Arguments

medicion	**Primer argumento.** Data frame producido por medir(); puede reunir varias corridas si conserva sus id_medicion —con rbind(), que se niega a unir mediciones de modelos distintos: para eso está historico_calidad()—. Cada id_medida se cuenta una vez: una medida repetida, o una con resultado NA sin estar suprimida, se rechaza nombrándola. No es el perfil descriptivo que devuelve perfilar().
perfil	**Segundo argumento.** Objeto creado por perfil_evaluacion(), que reúne las reglas que se aplican a la medición. Es un perfil de evaluación, distinto del objeto perfil creado por perfilar().

Value

Objeto evaluacion_calidad con tres data frames filtrables: medidas, reglas y perfiles. Si alguna regla declara un desenlace, contiene además desenlaces, un plan que identifica las medidas incumplidas, el valor medido, el motivo y la regla que lo produjo. Cuando una métrica no pudo medirse, conserva cobertura_metricas y deja en NA el resumen afectado, en lugar de tratar la ausencia como éxito; también sobre un agregado, donde la regla nombra la métrica agregada —agregada:ratio:NoNulo— o su nombre con instancia, y la parte sin medir es una de sus instancias. La evaluación de un agregado conserva además la cobertura de su frontera —las tablas o colecciones declaradas que no entraron al número—, la imprime y la lleva a su informe y a historico_calidad(). Y cuando una regla **declara** una métrica que la medición no trae —ninguna medida de ella—, el veredicto cubre menos de lo que la regla dice: eso se avisa al evaluar y queda en el atributo cobertura_reglas, con la corrida, la métrica, el motivo y cómo resolverlo: se mira corrida por corrida, así que una corrida evaluada junto a otras avisa lo mismo que evaluada sola. Si **ninguna** de las métricas declaradas por una regla tiene medidas, evaluar() se niega y nombra las solicitadas y las disponibles. Conserva además, en atributos, la configuración del modelo, la aplicabilidad y el perfil de evaluación que produjo el resultado.

Examples

```
nucleo <- metricas_nucleo()
especifica <- especializar(nucleo$NoNulo)
instancia <- instanciar(especifica, "personas", "edad")
medidas <- medir(modelo(instancia), data.frame(edad = c(20, NA, 35)))
regla <- regla_evaluacion("Al menos 90%", function(x) x > 0.9)
evaluar(medidas, perfil_evaluacion("Avanzado", regla))
```

---

granularidades *Granularidades y transiciones de agregación*

Description

granularidades() declara los diez niveles del marco, y los diez se miden. La columna implementada significa que se puede instanciar una metrica y medir ese nivel. La columna agregable es distinta: indica que el grafo ofrece al menos una transicion de agregacion hacia o desde el nivel. Por eso conjuntoAtributos puede estar implementado para medir una metrica y a la vez no ser agregable

con las transiciones disponibles; `motivo_agregacion` deja esa razon escrita. Los cuatro de arriba —colección, conjunto de colecciones, organización y conjunto de organizaciones— sólo cuando el usuario **declara la frontera**: qué tablas componen una colección, qué bases un conjunto, qué colecciones una organización, qué organizaciones un conjunto. `lupa` no infiere ninguna de las cuatro, porque ninguna está en los datos.

Usage

```
granularidades()

transiciones_granularidad()
```

Details

Que estén implementadas no obliga a usarlas. Un análisis que no tiene una organización detrás se detiene donde corresponda; los niveles superiores existen para quien los necesita.

`transiciones_granularidad()` devuelve el grafo dirigido de agregaciones. La transición `instanciaAtributo` a `instanciaEntidad` se incorpora porque el propio marco la usa aunque no aparezca en su tabla no exhaustiva.

Value

Data frames con niveles o aristas del grafo de granularidad.

See Also

`modelo()`, `medir()`, `evaluar()`

Examples

```
granularidades()
transiciones_granularidad()
```

---

guardar_historico *Guardar y recuperar un histórico de calidad*

Description

Persiste el data frame versionado mediante RDS de base R. La escritura no reemplaza un archivo existente salvo consentimiento explícito.

Usage

```
guardar_historico(historico, archivo, sobrescribir = FALSE)

leer_historico(archivo)
```

Arguments

historico	Objeto creado por historico_calidad().
archivo	Ruta del archivo RDS. Una ruta que ya es un directorio se rechaza: el archivo va dentro, con su nombre.
sobrescribir	Si se permite reemplazar un archivo existente.

Value

guardar_historico() devuelve invisiblemente la ruta normalizada; leer_historico() devuelve un historico_calidad validado.

See Also

guardar_historico(), detectar_deriva_calidad()

historico_calidad(), comparar_evaluaciones()

Examples

```
archivo <- tempfile(fileext = ".rds")
guardar_historico(historico_calidad(), archivo)
leer_historico(archivo)
unlink(archivo)
```

---

guiar_limpieza *Revisar decisiones de limpieza paso a paso*

Description

Recorre los grupos pendientes y aquellos que contienen alternativas destructivas. Muestra evidencia calculada sobre datos, las estrategias y sus justificaciones, y devuelve el plan editado sin aplicarlo. En una sesión no interactiva retorna inmediatamente el plan sin cambios, salvo que se proporcione un selector explícito.

Usage

```
guiar_limpieza(
  plan,
  datos,
  selector = NULL,
  diccionarios = list(),
  max_ejemplos = 5L
)
```

Arguments

plan	Objeto plan_limpieza.
datos	Datos correspondientes al perfil que originó el plan.
selector	Función opcional que recibe una lista con grupo, acciones, elegibles, ejemplos y opciones. Debe devolver la posición —el número que imprime la lista, que cuenta sólo las opciones elegibles—, el identificador o el nombre de una estrategia, o 0 para no hacer nada. NA no es un valor válido: se rechaza con un error, para que una lectura fallida no quede registrada como una decisión de omitir.
diccionarios	Lista opcional con nombre de diccionarios por columna.
max_ejemplos	Máximo de ejemplos reales mostrados por grupo.

Details

No se representa "no hacer nada" como una acción ficticia. Elegirlo cambia decision_grupo a "omitida"; por contraste, un grupo aún no revisado conserva "pendiente". Cuando conservar los datos es la recomendación, esa opción también se muestra con la marca "(Recomendado)" y su justificación. Los diccionarios de capitalización se suministran como una lista con nombre de vectores con nombre.

Value

El mismo plan_limpieza recibido, con la columna de decisión sincronizada según lo elegido, y sin ejecutar ninguna acción sobre los datos: cambia qué acciones quedan marcadas para aplicar(), no la tabla. En una sesión no interactiva y sin selector, devuelve el plan sin cambios: **también un plan ya editado**, cuyo decision_grupo se conserva tal como lo recibió. La sincronización de decisiones corre en el camino interactivo y en aplicar(), no en la puerta de salida.

See Also

planificar_limpieza(), aplicar()

Examples

```
datos <- data.frame(zona = c("Norte", "NORTE", "sur"))
plan <- planificar_limpieza(perfilar(datos), datos)
guiado <- guiar_limpieza(plan, datos)
identical(plan, guiado) # TRUE en una sesión no interactiva
```

---

historico_calidad *Construir y ampliar un histórico de calidad*

Description

Crea un data frame plano y versionado con corridas producidas por medir() o evaluar(). acumular_historico() agrega objetos al mismo esquema y es idempotente cuando recibe otra vez registros idénticos, incluso si el RDS se guardó o se vuelve a leer bajo otro locale.

Usage

```
historico_calidad(..., detalle = c("resumen", "completo"))

acumular_historico(historico, ..., detalle = c("resumen", "completo"))
```

Arguments

`...`	Objetos `medicion`, `evaluacion_calidad` o `historico_calidad`. También puede darse una única lista que los contenga.
`detalle`	Para evaluaciones, `"resumen"` conserva los niveles de regla y perfil; `"completo"` conserva además cada evaluación de medida. Una `medicion` pasada explícitamente siempre se conserva completa.
`historico`	Objeto creado por `historico_calidad()`.

Details

El detalle predeterminado evita repetir una fila por celda y regla cuando el objetivo es monitorear la serie de evaluaciones. El objeto no guarda modelos, closures, datos originales ni perfiles de profiling. Esto mantiene la tabla exportable directamente con `write.csv()` o una herramienta de base de datos. Al volver a leerla, cada columna recupera el tipo del esquema —`read.csv()` lee como lógica una columna de texto que viene entera en `NA`— y las fechas de texto se leen en UTC, con su hora y con su desplazamiento cuando lo traen —`2026-01-31 10:00:00-03:00` son las 13:00 UTC; también `Z`, `+0100` o `-03`, como los escribe una columna `timestamptz` de una base—. Un texto que no se puede leer entero se rechaza: no se lee a medias.

`[`, `subset()` y `dplyr::filter()` conservan la configuración de las corridas que quedan; `rbind()` de dos históricos los acumula, como `acumular_historico()`.

El esquema largo mapea las cuatro tablas de la sección 9.5 del marco mediante `nivel`. Las columnas que no corresponden a un nivel quedan como `NA`.

Value

Data frame S3 `historico_calidad`. La columna `version_esquema` y el atributo del mismo nombre permiten migraciones futuras. `nivel` corresponde a `medida`, `evaluacion_medida`, `evaluacion_regla` o `evaluacion_perfil`; una métrica sin valores se conserva como `metrica_no_evaluada` con su motivo, siempre que la medición tenga al menos una medida; y una parte declarada en una frontera —una tabla de la colección, una colección de la organización— que no entró al número, como `parte_no_medida`, con la frontera en `id_registro`, la parte en `entidad` y el motivo en `objeto_medible` —también cuando un objeto reúne varias corridas con `rbind()`: cada parte va a la corrida a la que le faltó—. Una medida que una regla declaró `desenlace = "suprimir"` no publica su valor **tampoco aquí**: su fila deja `resultado` en `NA` y marca `objeto_medible` con `[valor suprimido]`, en los dos niveles donde esa medida aparece —`medida` y `evaluacion_medida`, en este último bajo cualquier perfil o regla que la evalúe, no solo el que la suprimió—, porque esta tabla está pensada para exportarse. Con el detalle resumido, las medidas suprimidas entran igual, enmascaradas en el nivel `evaluacion_medida`: así una medición acumulada **después** —también sobre un histórico guardado o leído de un CSV— queda tapada en las mismas medidas, sea cual sea el orden en que llegan los objetos. Una medición **enteramente** vacía —ninguna métrica pudo aplicarse— no se acumula: se rechaza citando el motivo que `medir()` declaró en

cobertura_metricas, porque no hay corrida que registrar. El atributo configuracion_evaluacion conserva, en una tabla plana separada, el modelo, su marco y sus tipos, la aplicabilidad, el perfil, la identidad de tabla y **la fecha** de cada corrida. Acumular una corrida con un id_medicion ya presente exige que todo eso coincida, la fecha incluida: dos entregas distintas son dos corridas, y el error nombra en qué difieren. Esa misma configuración viaja **en las filas** evaluacion_perfil, en las columnas identidad_tabla, configuracion_modelo, configuracion_marco, configuracion_tipos_resultado, configuracion_aplicabilidad y configuracion_perfil —en NA en los demás niveles—, porque el atributo no sobrevive a write.csv(): un histórico releído de un CSV la recupera de ahí. Son columnas agregadas al esquema 1: un histórico anterior que no las trae se lee igual, con ellas en NA.

References

AGESIC (2020). *Marco de trabajo para la Gestión de la Calidad de Datos en Gobierno Digital*, versión 1.6, sección 9.5, Presidencia de la República, Uruguay.

See Also

medir(), evaluar(), detectar_deriva_calidad(), reportar()

historico_calidad(), leer_historico()

Examples

```
nucleo <- metricas_nucleo()
instancia <- instanciar(especializar(nucleo$NoNulo), "personas", "edad")
medidas <- medir(
  modelo(instancia), data.frame(edad = c(20, NA)),
  id_medicion = "enero", fecha = as.POSIXct("2026-01-31", tz = "UTC")
)
evaluacion <- evaluar(
  medidas,
  perfil_evaluacion("Basico", regla_evaluacion("Presente", function(x) x > 0))
)
historico_calidad(medidas, evaluacion)
```

---

indice_calidad *Calcular un índice de calidad declarado por el usuario*

Description

Sin pesos, devuelve tablero_calidad() y nunca un puntaje. Con pesos nombrados por dimensión, transforma las métricas de defecto como 1 - valor, excluye las de orientación no_aplica y conserva cada paso.

Usage

```
indice_calidad(medidas, pesos, pesos_internos = NULL, ...)
```

Arguments

medidas	Medición, tablero o análisis de lupa.
pesos	Vector numérico nombrado por dimensión, en [0, 1] y con suma uno. Si se omite, se devuelve el tablero.
pesos_internos	Vector opcional nombrado por componente; es obligatorio para cada dimensión con más de una fila incluida.
...	Argumentos enviados a tablero_calidad() cuando medidas no es ya un tablero o análisis.

Details

Cuando una dimensión contiene varios componentes, pesos_internos debe declarar una ponderación completa que sume uno dentro de esa dimensión. No existe un promedio interno por omisión. El resultado conserva el tablero, ambas capas de pesos, las inversiones, las exclusiones, los universos y la cobertura del marco.

Una métrica sin dimensión no entra al índice: se excluye con su motivo, como las de orientación no_aplica. Un peso para una dimensión cuyas métricas se excluyeron se rechaza diciéndolo.

advertencia_universos **se calcula de los universos que el propio objeto publica**, no es un texto fijo. Con un solo universo lo nombra y dice que las unidades son comparables; con varios dice cuántos son y nombra, por universo, qué componentes caen en cada uno; sin componentes no afirma nada sobre universos. Así no afirma «salen de universos distintos» cuando los tres componentes publican universo = "celdas" —una afirmación que el mismo objeto desmentiría en su columna universo—, y nombra los componentes, que es justamente lo que promete declarar.

Value

Sin pesos, un tablero_calidad. Con pesos, un objeto S3 indice_calidad que nunca se imprime como un número aislado.

See Also

tablero_calidad()

Examples

```
nucleo <- metricas_nucleo()
instancias <- list(
  instanciar(especializar(nucleo$NoNulo), "padron", "codigo"),
  instanciar(especializar(nucleo$EntidadDuplicada), "padron")
)
medidas <- medir(modelo(instancias), data.frame(codigo = c("A", "B", "B")))
indice_calidad(medidas)
# Pesos propios de este ejemplo, no del paquete:
indice_calidad(
  medidas,
  pesos = c(Completitud = 0.6, Unicidad = 0.4)
)
```

`inferir_tipo` *Inferir el tipo implícito de un vector*

Description

Para columnas de texto distingue valores lógicos, enteros, dobles, fechas, fechas-hora, horas, identificadores y texto. Además de la etiqueta devuelve la proporción de valores compatibles, siempre en la escala `[0, 1]`. Los textos compuestos únicamente por `0` y `1` se consideran enteros; para inferir un lógico debe aparecer al menos un literal alfabético como `sí`, `no`, `true` o `false`. Una letra suelta (`T`, `F`, `S`, `N`) cuenta como literal sólo si aparece también su opuesto —`T` con `F` o `false`, `S` con `N` o `no`, y al revés—: sola es un código, y una columna de `F` y `M` no es lógica. Los literales se reconocen sin distinguir mayúsculas —`SÍ` es `sí`— con el pliegue de caja de `normalizacion()`, que no depende del locale. Un separador seguido de exactamente tres dígitos (`1.500`, `1,500`) cuenta como número sólo si la columna aporta evidencia de una sola convención decimal, con punto o con coma por igual. Un número con los miles agrupados —`1.234,56`, `1,234.56`, `12.500.000`— cuenta como número si agrupa los miles de a tres con la convención de la columna o, cuando la columna aporta evidencia de las dos, con la que trae el propio valor (`1.234,56` sólo admite la coma decimal): es como lo lee el resumen de `perfilar()`. Sin evidencia de convención no cuenta, y un primer grupo de miles que empieza con cero (`099.123.456`, la forma de un teléfono) no es un número en ninguna.

Usage

```
inferir_tipo(x, umbral = 0.8, muestra = 1e+05)
```

Arguments

`x`	Vector que se desea examinar. Una tabla —también la de una columna que da `tibble[, "f"]`— se rechaza: se pasa la columna, `tabla[["f"]]`.
`umbral`	Proporción mínima para asignar un tipo implícito.
`muestra`	Máximo de valores que se analizan.

Value

Lista de clase `inferencia_tipo` con `tipo`, `proporcion`, conteos, candidatos evaluados y, cuando corresponde, formatos de fecha. `n_analizados` y `muestreado` declaran cuántos valores sostienen la proporción. El resultado público no conserva cachés internos del perfilado.

See Also

`detectar_formatos_fecha()`, `descubrir_patrones()`, `perfilar()`

Examples

```
inferir_tipo(c("1", "2", "3"))
inferir_tipo(c("2020-01-01", "31/12/2020"))
```

`marco_calidad` *Declarar una taxonomía de calidad de datos*

Description

Un `marco_calidad` enumera las dimensiones y factores contra los que se interpreta la cobertura de un análisis. No contiene métricas instanciadas: esa función sigue correspondiendo a `modelo()`. Tampoco es un catálogo de métricas; `catalogo_agesic()` conserva esa correspondencia específica.

Usage

```
marco_calidad(nombre, factores)

marco_agesic()

marco_iso25012()

marco_cepal()
```

Arguments

`nombre`	Nombre del marco.
`factores`	Data frame con `dimension` y `factor`, o lista con nombres. El data frame puede añadir estos campos de contrato: • `como_resolverlo`: instrucción que muestra `cobertura_analisis()` cuando el factor todavía no fue medido; • `perfil_mide`: lógico que declara si el profiling por sí solo aporta una medición suficiente del factor. No ejecuta métricas ni se infiere del nombre; su valor predeterminado es `FALSE`; • `aplicabilidad`: `"siempre"`, `"temporal"` o `"geometria"`. Las dos últimas permiten informar `"no_aplica"` cuando el perfil no contiene columnas temporales o geometrías, respectivamente; • `disponibilidad`: `"disponible"` o `"fuera_de_alcance"`. Esta última declara una limitación del motor y no puede combinarse con `perfil_mide = TRUE`. Las columnas adicionales se conservan como metadatos y no cambian por sí solas la cobertura. Si se usa una lista, `como_resolverlo` recibe una instrucción genérica, `perfil_mide = FALSE`, `aplicabilidad = "siempre"` y `disponibilidad = "disponible"` para todas las filas.

Details

`marco_agesic()` devuelve la taxonomía incluida de fábrica. Un marco propio puede construirse con un data frame o con una lista cuyos nombres son dimensiones y cuyos valores son factores. La forma lista aplica a todas las filas los valores predeterminados descritos abajo; en particular, `perfil_mide = FALSE`.

En AGESIC, las instrucciones `como_resolverlo` de los factores se derivan de las métricas implementadas o agregables en `catalogo_agesic()`. En los marcos ISO/IEC 25012 y CEA/CEPAL no se infieren métricas por similitud de nombres: sus factores conservan la instrucción declarada por cada marco hasta que se les asocia una métrica con el mismo par dimensión-factor.

`marco_iso25012()` adapta las dos perspectivas de ISO/IEC 25012:2008 a la estructura dimensión-factor. Usa tres grupos disjuntos como dimensiones: características inherentes, características inherentes y dependientes del sistema, y características dependientes del sistema. Este agrupamiento es una representación operativa para `lupa`, no afirma que la norma defina una jerarquía dimensión-factor. Los nombres de las quince características y su clasificación siguen la norma; las descripciones son redacción propia. Las quince filas declaran `perfil_mide = FALSE`: a diferencia de la política incluida para dos factores de AGESIC, el profiling genérico no demuestra por sí solo que una característica ISO satisfaga el uso declarado.

`marco_cepal()` representa los cuatro niveles y los diecinueve principios del marco nacional de aseguramiento de la calidad de las Naciones Unidas, adoptados y adaptados para América Latina y el Caribe por la CEA/CEPAL. La columna `principio` conserva la numeración de la fuente. Los nombres de los niveles y principios son textuales; las descripciones y las instrucciones `como_resolverlo` de este paquete son redacción propia. Los principios 1 a 13 quedan fuera del alcance de una tabla: se refieren al sistema, al entorno institucional o al proceso estadístico. Los principios 14 a 19 describen productos estadísticos y quedan disponibles, aunque ninguno se considera medido por el profiling genérico.

Value

`marco_calidad()`, `marco_agesic()`, `marco_iso25012()` y `marco_cepal()` devuelven un objeto S3 `marco_calidad`. `as.data.frame()` devuelve su tabla de factores.

References

ISO/IEC (2008). *ISO/IEC 25012:2008 Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model*. https://www.iso.org/standard/35736.html.

Naciones Unidas (2019). *Manual del marco nacional de aseguramiento de calidad en las estadísticas oficiales*. Estudios en Métodos, serie M, N° 100 (ST/ESA/STAT/SER.M/100), Nueva York.

Grupo de Trabajo de la Conferencia Estadística de las Américas (CEA), coordinado por Colombia (DANE) y México (INEGI), Secretaría Técnica: División de Estadísticas de la CEPAL (2022). *Guía para la implementación del marco de aseguramiento de la calidad para procesos y productos estadísticos*. LC/CEA.11/19. Comisión Económica para América Latina y el Caribe (CEPAL), Naciones Unidas, Santiago.

See Also

`catalogo_agesic()`, `modelo()`, `cobertura_analisis()`

Examples

```
propio <- marco_calidad("Marco operativo", list(
  Trazabilidad = c("Origen documentado", "Linaje reproducible"),
  Pertinencia = "Adecuaci\u00f3n al uso"
```

```
))
propio
as.data.frame(propio)
marco_agesic()
iso <- marco_iso25012()
table(as.data.frame(iso)$dimension)
marco_cepal()
```

`medicion_desde_estimaciones` *Llevar estimaciones ya calculadas al contrato de medición*

Description

`lupa` **no estima**: eso necesita un diseño muestral, estimación de varianza y otra disciplina. Lo que sabe hacer es evaluar contra un marco declarado. Esta función recibe estimaciones calculadas por otra herramienta —`survey`, el paquete `calidad` del INE de Chile, o cualquier otra— y las convierte en una medición que `evaluar()` entiende.

Usage

```
medicion_desde_estimaciones(
  estimaciones,
  entidad,
  fuente,
  atributo = NULL,
  columnas = NULL,
  fecha = Sys.time()
)
```

Arguments

`estimaciones`	Data frame con una fila por estimación y una columna por estadístico. Los nombres reconocidos son los de `estadisticos_estimacion()`; se puede renombrar con `columnas`.
`entidad`	Nombre de la entidad estimada, por ejemplo el tabulado o la población de referencia.
`fuente`	Texto que declara quién calculó las estimaciones. Es obligatorio: sin él, el resultado no dice de dónde viene.
`atributo`	Columna opcional de `estimaciones` que nombra el atributo o la celda estimada. Cuando falta, se numeran las filas.
`columnas`	Vector con nombres para traducir columnas de `estimaciones` a estadísticos reconocidos, en la forma `c(cv = "coef_var")`.
`fecha`	Fecha y hora de la medición. Se lee como en `medir()`: se guarda en UTC y un texto —`"2026-03-01"`— se lee en UTC, no en el huso de la sesión, así que la misma llamada registra el mismo instante —y el mismo `id_medicion`— en cualquier máquina. Una sola fecha: `NA`, `NULL`, dos fechas o un texto que no se puede leer entero se rechazan.

Details

Cada estadístico se convierte en **una medida canónica propia**, con su métrica, su tipo y su orientación, porque los siete tienen unidades y dominios distintos: un coeficiente de variación y un tamaño de muestra no se evalúan con la misma regla. Los reconocidos están en `estadisticos_estimacion()`.

Cada medida declara su procedencia en `fuente`, de modo que nadie lea el resultado como si `lupa` lo hubiera calculado. Los estadísticos que la tabla no traiga simplemente no producen medidas: no se rellenan con ceros ni se estiman.

Una columna vacía tampoco produce medidas. Un estadístico cuya columna está presente pero no trae ningún valor utilizable no es una estimación: no se publica con `resultado = NA`, porque eso sería informar como medido lo que nadie midió —y además deja la medición inservible, porque `evaluar()` la rechaza entera—. Esos estadísticos se declaran en el atributo `estadisticos_sin_valores`, separados de `estadisticos_ausentes`: no es lo mismo no mandar la columna que mandarla sin datos, y la acción de quien la preparó es distinta en cada caso. Dentro de una columna que sí trae datos, la celda vacía se descarta por la misma razón, y el resto de las estimaciones se publica.

Una celda que no es una estimación también se descarta, y se declara. Un texto que no es un número, o un número fuera del dominio de su estadístico —un infinito, en cualquiera; un tamaño de muestra que no es un entero no negativo; un error estándar, un coeficiente de variación, unos grados de libertad, un efecto de diseño o un tamaño efectivo negativos— no se publica: se avisa y queda en el atributo `celdas_descartadas`, con la métrica, la celda, el valor recibido y el motivo (`no_numerica` o `fuera_de_dominio`). Una columna `factor` se lee por el texto de sus niveles, no por sus códigos.

Value

Data frame de clase `medicion_calidad` y `medicion` con el contrato de `medir()`, más las columnas `fuente` y `unidad`: se acumula con `historico_calidad()` y se une con `rbind()` como cualquier medición. Su `id_medida` nombra la métrica y la celda estimada —`objeto_medible`— y no el número de fila, así que dos tabulados de la misma entidad y la misma fecha se pueden unir y evaluar juntos; la misma celda medida dos veces sí repite su identificador, y `evaluar()` lo rechaza.

See Also

`estadisticos_estimacion()`, `evaluar()`, `marco_cepal()`

Examples

```
estimaciones <- data.frame(
  celda = c("Montevideo", "Interior"),
  stat = c(0.42, 0.38),
  cv = c(0.08, 0.34),
  n = c(1200L, 90L)
)
medicion_desde_estimaciones(
  estimaciones, entidad = "ech2024", atributo = "celda",
  fuente = "survey 4.4, diseno complejo declarado por el equipo"
)
```

`medir` *Medir un modelo de calidad*

Description

Ejecuta todas las métricas instanciadas de un `modelo_calidad`. Cada fila es una medida reutilizable y conserva el identificador y la fecha de la corrida.

Usage

```
medir(
  modelo,
  datos,
  id_medicion = NULL,
  fecha = Sys.time(),
  aplicabilidad = NULL,
  proteger_datos_personales = FALSE,
  columnas_personales = character(),
  validadores_personales = NULL
)
```

Arguments

`modelo`	**Primer argumento.** Objeto operativo creado por `modelo()`; reúne las métricas instanciadas que se van a ejecutar. No es el objeto `perfil` que devuelve `perfilar()` ni el marco conceptual de `marco_calidad()`.
`datos`	**Segundo argumento.** Data frame para una sola entidad o lista con nombre de data frames para varias entidades. Es la tabla que se mide, no un perfil ni un modelo. Con una lista, tienen que estar las tablas de **todas** las entidades que el modelo mide; si falta alguna se rechaza antes de medir, nombrando todas las que faltan y cuáles se recibieron. Una tabla que el modelo no pide **se ignora**: no hace falta que la lista coincida exactamente con las entidades, sólo que no falte ninguna.
`id_medicion`	Identificador de corrida. Si se omite, se genera uno. Cada medida lleva un `id_medida` que lo antepone a su métrica instanciada y la numera dentro de ella —`M-NoNulo@t.x-000001`—: una corrida armada por partes con el mismo `id_medicion` y unida con `rbind()` no repite identificadores, y la misma métrica medida dos veces en la corrida sí, que es lo que `evaluar()` y `historico_calidad()` rechazan.
`fecha`	Fecha y hora de la corrida. Se guarda en UTC; un texto —`"2026-03-01"` o `"2026-03-01 09:30:00"`— se lee en UTC, no en el huso de la sesión, así que la misma llamada registra el mismo instante en cualquier máquina. Un desplazamiento escrito en el texto —`"2026-03-01 09:30:00-03:00"`, `Z`, `+0100`— se respeta, y un texto que no se puede leer entero se rechaza.

`aplicabilidad` Lista con nombre por columna, donde cada elemento es una fórmula que dice en qué filas esa columna corresponde —por ejemplo `list(marca_auto = ~ tiene_auto == "Si")`—. Las filas fuera de ese universo salen de la medición en vez de contarse como ausencia. Una métrica de varias columnas se mide donde corresponden todas. Las medidas citan la fila de la tabla original —`fila` y `objeto_medible`—, no su posición en el universo.

Es la misma declaración que recibe `perfilar()`, y hace falta porque una métrica de completitud sobre una columna condicionada mide lo que no corresponde: con universo de 300 filas sobre 1.000 y 30 vacías de verdad, sin declararlo da `0,270` y declarándolo `0,900`. El histórico y la deriva consumen mediciones, así que heredan el número que salga de acá.

Sin declaración toda la tabla aplica y el resultado es el de siempre.

`proteger_datos_personales`
Si se enmascaran los candidatos de proximidad que corresponden a columnas personales. Por omisión `FALSE`. También gobierna lo que escribe un método propio: si alguna tabla recibida tiene columnas personales, su etiqueta `objeto` se reemplaza por la canónica del paquete (`tabla$columna[fila]`), y los valores de esas columnas se enmascaran en el motivo de una métrica `no_medible`. La protección alcanza a lo que `medir()` recibe: un valor que un método trae de otro lado —una tabla que no se le pasó a `medir()`— no se puede reconocer, y publicarlo es responsabilidad de quien escribe el método.

`columnas_personales`
Columnas que traen datos personales, declaradas con la misma forma que acepta `perfilar()`: nombres de columna, o un vector con nombre donde el nombre es la columna y el valor es el tipo. Se protegen aunque el léxico por omisión no las reconozca —un nombre propio de la organización, un identificador interno—. Sólo tiene efecto con `proteger_datos_personales = TRUE`.

`validadores_personales`
Pack o lista nombrada de funciones de validación, con la misma forma que acepta `perfilar()`. Sin esto, la medición usa el léxico por omisión.

Value

Data frame S3 de clase `medicion`, con una fila por objeto medido. Los booleanos se almacenan como `0` y `1` en la columna común `resultado`. `orientacion` conserva si un valor alto expresa conformidad, si un valor alto expresa defecto o si esa lectura no aplica. Algunas métricas que trabajan con un vocabulario o un alcance parcial agregan un atributo `alcance_metricas` con sus conteos y límites. Si una métrica no puede medirse, no crea filas ni ceros: deja el motivo y cómo resolverlo en el atributo `cobertura_metricas`, con un estado que distingue tres causas: `sin_valores` cuando su universo no tiene valores, `contrato_incompleto` cuando su contrato no trae un campo que necesita, y `no_medible` cuando su método falló sobre estos datos —una columna sin dos valores para `ErrorEstandar`, un atributo que la tabla no trae, una regla que devuelve `NA`—, devolvió una salida que no cumple el contrato de `metodo`, o no devolvió ninguna medida sobre un universo que sí tiene valores. Cuando el método falló, el motivo conserva su mensaje, con los valores de las columnas personales enmascarados si `proteger_datos_personales = TRUE`, y `medir()` **avisa**: la falla no se lleva la medición de las demás métricas, pero tampoco queda muda. Y cuando **sí** pudo medirse pero sobre **menos** elementos de los que hay en su universo aplicable

—una métrica por celda no mide la celda vacía, que no produce medida ni cuenta como incumplimiento—, el atributo `alcance_medidas` publica cuántos midió de cuántos, con su unidad: sin ese número, el agregado de tres celdas de cuatro no se distingue del de cuatro. Viaja al tablero, se imprime con la medición y se publica en el informe. También conserva `configuracion_modelo` y `configuracion_aplicabilidad`, descripciones de la política usada para que una deriva posterior pueda distinguir modelo de datos. Si el modelo declara un marco, la medición conserva tambien `marco_calidad`; la configuracion del modelo registra su nombre, sus pares dimension-factor y el `tipo_resultado` de cada metrica.

Esa descripcion es **insensible al orden de la declaracion**: declarar las mismas metricas, o los mismos pares del marco, en otro orden no cambia el modelo, porque un modelo es un conjunto de instrumentos y un marco un conjunto de pares. El orden de los `atributos` de una instancia **sí** es parte del modelo, porque el metodo recibe las columnas en ese orden. Y el `metodo` de medicion entra en la descripcion solo si lo declaro quien llama: el metodo por omision de una metrica es del paquete y su identidad ya viaja en el nombre de la metrica, de modo que un cambio interno de lupa no se lee como un cambio del modelo del usuario.

Examples

```
nucleo <- metricas_nucleo()
especifica <- especializar(nucleo$NoNulo, nombre_especifico = "NoNuloEdad")
instancia <- instanciar(especifica, "personas", "edad")
medir(modelo(instancia), data.frame(edad = c(20, NA, 35)))
```

`metricas_referencial` *Métricas que consumen un referencial tabular*

Description

Devuelve las tres métricas base que pueden medirse con el contrato de `referencial()`. `CorrectitudSemFuerte` verifica que la identificación exista; `CorrectitudSemDebil` comprueba el par identificación–valor; `RatioCobertura` mide qué proporción del universo completo de claves aparece en la entidad. Los ratios de correctitud se obtienen mediante `agregar()` con `"ratio"`. Las tres métricas aceptan `normalizar`, `proximidad`, `metodo`, `p`, `umbral`, `max_pares` y `nucleos`. `normalizar = NULL` hereda el perfil declarado por `referencial()`. La normalización sólo cambia la representación usada para emparejar: no modifica los datos. La proximidad es evidencia para los valores ausentes y nunca cambia su veredicto; si el paquete opcional stringdist no está instalado, se declara que no se calculó. Se calcula una sola vez por valor fallido distinto y la evidencia se reparte a las filas repetidas. El alcance conserva por separado `n_fallos` (filas), `n_valores_fallidos_distintos`, `n_valores_fallidos_comparados` y los pares comparados; así el límite no depende del orden ni de la frecuencia de las filas.

Usage

```
metricas_referencial()
```

Details

Los números se emparejan por su valor. El mismo número guardado como entero en una tabla y como doble en la otra coincide —`100000L` y `1e5`—, y dos dobles distintos no coinciden aunque se escriban igual con 15 cifras —`0.1 + 0.2` y `0.3`—. Lo mismo vale para `metricas_nucleo()` cuando compara filas por sus atributos, como en `EntidadDuplicada`. Los instantes y las fechas se emparejan también por su valor, con su fracción: dos instantes del mismo segundo no coinciden.

Un texto con bytes que no son UTF-8 válido —lo que deja un archivo latin1 leído sin declarar su codificación— se empareja por sus bytes, como `%in%`: dos valores así distintos no coinciden entre sí. La proximidad no los compara, porque una distancia de texto no se mide sobre bytes que no son texto, y su `motivo` declara cuántos quedaron afuera.

Los valores ausentes no generan medidas de correctitud: corresponden a la dimensión Completitud. La cobertura ignora claves ausentes en el objetivo y no permite que duplicados inflen el resultado. El universo se cuenta con la misma identidad con que se empareja: dos claves del padrón que la normalización funde —`A` y `a`— son una sola clave del universo, y `n_referencial` cuenta las claves de ese universo. Sobre un referencial sin claves, `RatioCobertura` no se mide: la cobertura de un universo vacío no es una proporción, y la métrica queda `sin_valores` con ese motivo, igual que cuando la entidad no tiene filas o ningún valor en los atributos ligados: lo que no se pudo medir no se publica como una cobertura cero.

Value

Lista con tres objetos `metrica_generica` —`CorrectitudSemFuerte`, `CorrectitudSemDebil` y `RatioCobertura`—, listos para instanciar contra un `referencial`. Son definiciones: no miden hasta que se las instancia.

See Also

`referencial()`, `metricas_nucleo()`, `agregar()`

Examples

```
ref <- referencial(
  data.frame(id = 1:3, nombre = c("Ana", "Bruno", "Carla")),
  "id", "nombre", completo = TRUE, alcance = "padr\u00f3n de ejemplo"
)
m <- metricas_referencial()
fuerte <- instanciar(especializar(m$CorrectitudSemFuerte),
  "personas", "id", referencial = ref)
medir(modelo(fuerte), data.frame(id = c(1, 4)))
```

`modelo_calidad` *Construir métricas y modelos de calidad*

Description

`metrica()` declara una métrica genérica y devuelve una fábrica de closures. `especializar()` fija sus propiedades de configuración; la closure específica resultante puede instanciarse sobre distintas columnas o tablas. `instanciar()` liga la métrica a objetos concretos y materializa el método de medición. `modelo()` reúne métricas instanciadas sin calcular un índice global.

Usage

```
metrica(
  nombre,
  semantica,
  granularidad,
  tipo_resultado,
  propiedades = character(),
  dimension = NA_character_,
  factor = NA_character_,
  metodo = NULL,
  validar_propiedades = NULL,
  orientacion = if (tipo_resultado %in% c("booleano", "real")) "conformidad" else
    "no_aplica"
)

especializar(metrica, nombre_especifico = NULL, ...)

instanciar(
  metrica_especifica,
  entidad,
  atributos = character(),
  nombre_instancia = NULL,
  metodo = NULL,
  referencial = NULL
)

propiedades_metrica(x)

modelo(..., marco = NULL)

metricas_nucleo()
```

Arguments

`nombre`	Nombre estable y legible.
`semantica`	Descripción de lo que mide la métrica.
`granularidad`	Uno de los niveles de ontología devueltos por `granularidades()` o su alias en la columna `relacional`. Los aliases se normalizan al nombre de ontología antes de guardarse en la métrica.

`tipo_resultado` `"booleano"`, `"real"` en `[0, 1]`, `"numero_real"`, `"entero"` no negativo o `"duracion"` no negativa. Los tres últimos conservan resultados no acotados del catálogo y no admiten las cuatro agregaciones normalizadas.

`propiedades` Nombres de las propiedades que fija `especializar()`.

`dimension`, `factor`
Metadatos taxonómicos; no se usan para calcular puntuaciones. Cada uno es una cadena no vacía o `NA`.

`metodo` Método predeterminado opcional. Es una función de `tablas` e `instancia` que cumple el contrato descrito en **Contrato de** `metodo`.

`validar_propiedades`
Función opcional que recibe la lista con nombre enviada a `especializar()` y debe devolver otra lista con nombre formada sólo por propiedades declaradas. Puede validar alternativas, completar valores predeterminados y normalizar la configuración. Si es `NULL`, todas las propiedades declaradas son obligatorias y no se admiten otras.

`orientacion` Sentido de lectura del resultado: `"conformidad"` indica que un valor mayor es mejor; `"defecto"`, que un valor menor es mejor; y `"no_aplica"`, que la métrica no es una proporción interpretable en esos términos. Es un vocabulario cerrado. Las métricas booleanas deben usar una de las dos primeras y los resultados no acotados deben usar la última.

`metrica` Objeto de clase `metrica_generica`.

`nombre_especifico`
Nombre de la especialización. Si se omite, conserva el nombre genérico.

`...` En `especializar()`, propiedades con nombre de las declaradas en `metrica(propiedades = )`; consúltelas con `propiedades_metrica()`. En `modelo()`, métricas instanciadas o una única lista que las contenga.

`metrica_especifica`
Objeto de clase `metrica_especifica`.

`entidad` Nombres de las tablas ligadas, en el orden que espera el método.

`atributos` Nombres de las columnas ligadas, en el mismo orden. Son los nombres que tiene la tabla: una columna llamada `""` o `NA` se liga con ese nombre, que es el que publica `perfilar()`. Un nombre que la tabla no tiene no se rechaza al instanciar; `medir()` declara la métrica `no_medible` con el atributo que no encontró.

`nombre_instancia`
Nombre de la instancia. Si se omite, se deriva de la especialización y los objetos ligados.

`referencial` Objeto opcional creado por `referencial()`. Se conserva en la instancia sin modificarlo: `instanciar()` no supone que toda métrica lo use. El `metodo` debe leerlo y validar el contrato que necesite; las métricas de `metricas_referencial()` hacen esa validación al medir.

`x` Métrica genérica, específica o instanciada.

`marco` Objeto opcional creado por `marco_calidad()`. Cuando se provee, todas las métricas instanciadas deben pertenecer a uno de sus pares dimensión-factor.

Details

`metricas_nucleo()` devuelve veintidós métricas automatizables una vez declaradas sus propiedades; `escala()` y `vigencia()` hacen explícitos los insumos expertos que algunas necesitan. `metricas_referencial()` aporta por separado las tres métricas que consumen un padrón tabular. Consulte `catalogo_agesic()` para la correspondencia completa.

En `ReglaIntegridadInterEntidad`, `entidad` y `atributos` se ligan como `c(referencia, dependiente)` y `c(clave_primaria, clave_foranea)`. Esta implementación calcula cobertura PK/FK como resultado real y sólo cubre una parte de la genérica del marco, que declara granularidad `conjuntoEntidades`, resultado booleano y admite además una expresión condicional. `catalogo_agesic()` deja visible esa cobertura parcial. Pese a su nombre, `ErrorEstandar` sigue literalmente la semántica de la tabla 16.5 del marco y devuelve la desviación estándar muestral sin normalizar; exige al menos dos valores numéricos válidos y ninguno infinito —con uno, la métrica se declara no medible, como `Escala`—. Por eso declara `tipo_resultado = "numero_real"` y no admite `agregar()`.

`Formato` acepta exactamente una de las propiedades `expresion_regular`, `diccionario` o `validador`. Esta última permite conectar validadores externos sin incorporarlos como dependencias. Tanto `Formato` como `ValoresPosiblesPorExtension` omiten los valores `NA`: un ausente no genera una medida en esas métricas y, por lo tanto, tampoco integra el denominador de sus agregaciones. La completitud se mide por separado con `NoNulo`; así un mismo ausente no se penaliza en dos factores. `NoNulo` acepta el vector opcional `valores_nulos` para aplicar de forma deliberada el diccionario de nulos que contempla el marco; sin configurarlo, sólo considera los `NA` reales. Con `opcional = TRUE` mide sobre las celdas presentes: la ausencia no es defecto y sólo cuentan como faltantes los `valores_nulos`. Es como `proponer_modelo()` propone la `NoNulo` de una columna declarada en `columnas_opcionales` de `perfilar()`, para medir sobre el mismo universo que el perfil: con un `NA` y tres `-999` en cuarenta celdas, 36/39 = 0,923 y no 36/40. Cuando `valores_nulos` contiene texto, su comparación usa la misma normalización textual que `perfilar()`: ignora los blancos de los bordes y no distingue mayúsculas. Así, `""`, `" "` y un tabulador describen la misma ausencia cuando se declara `valores_nulos = ""`. Los marcadores que `perfilar()` reconoce por su cuenta como faltantes disfrazados (`"NA"`, `"null"`, `-999`) sólo cuentan para `NoNulo` si se declaran en `valores_nulos`: la métrica mide lo que el modelo declara. `ValoresPosiblesPorComprension` sigue la misma convención y acepta un `predicado` o un rango definido por `minimo`, `maximo` e `inclusivo`.

Las métricas de duplicación marcan **todas** las apariciones que participan en un grupo repetido, no sólo la segunda y siguientes. `AtributoDuplicado` omite ausentes. `ConjuntoAtributosDuplicado` compara las columnas ligadas y `EntidadDuplicada` compara la fila completa cuando se instancia sin atributos. Si se ligan atributos de clave, sigue la semántica del marco: marca filas con la misma clave cuyos demás valores son iguales o ausentes en alguna de las dos. En las comparaciones exactas, los `NA` forman parte de la combinación comparada.

`EntidadContradictoria` compara los valores distintos de un atributo con la misma normalización y medida de similitud que `detectar_duplicados_aproximados()`. Marca las filas cuyo valor tiene un vecino por debajo de `umbral`, pero no modifica datos ni propone una limpieza. Su alcance se conserva en el atributo `alcance_metricas` del resultado de `medir()`: informa los valores distintos comparados, los pares evaluados y los que quedaron bajo el umbral. `max_valores` permite acotar el vocabulario; cuando se alcanza, el alcance declara el prefijo evaluado y los valores que quedaron fuera.

`DesactualizacionPorFormato` devuelve `TRUE` cuando el valor **no** cumple el formato vigente. Conforme a las tablas 16.29 y 16.30 del marco, `OportunidadAtributoPorFecha` indica si la fecha es

anterior o igual a fecha_limite, y OportunidadAtributoPorIntervalo si pertenece al intervalo cerrado [inicio_vigencia, fin_vigencia]; ambas son booleanas.

GradoOportunidadAtributoPorFecha y GradoOportunidadAtributoPorIntervalo son extensiones propias basadas en el curso CPAP, no entradas adicionales del catálogo AGESIC. Conservan la fórmula continua max(0, min(1, 1 - (t1 - t2) / (t3 - t2))) para expresar cuánto margen de utilidad queda. Exigen que t3 sea posterior a t2; un intervalo de duración cero se rechaza porque no define el cociente. Todas estas métricas omiten los valores NA: su ausencia corresponde a completitud y no genera una segunda medida de incumplimiento.

Toda métrica que compara la columna con un valor temporal declarado avisa cuando las dos puntas no son de la misma clase. Un Date se ancla a la medianoche UTC y un POSIXct vale por su instante, así que mezclarlos cambia el número sin que se vea: la fila que cae exactamente en un límite cambia de veredicto con el huso de la sesión, y en las comparaciones por conjunto —valores_nulos de NoNulo, diccionario de Formato, el dominio de ValoresPosiblesPorExtension, el rango de ValoresPosiblesPorComprension y la clave de un referencial()— ningún valor coincide con ninguno: la métrica publicaría un incumplimiento que no existe, o —peor, en NoNulo— una completitud perfecta sobre una columna llena de ausencias disfrazadas. Declarar las dos puntas en la misma clase deja el número estable.

Desviación documentada del marco: en DensidadPonderada, un atributo más crítico recibe un coeficiente mayor y, si falta, produce una penalización mayor. El texto del marco indica acercar a cero el coeficiente de mayor gravedad, lo que penalizaría menos el ausente crítico y contradice el sentido de la ponderación. Los coeficientes deben estar en [0, 1] y sumar 1.

tipo_resultado es el contrato canónico que consultan las agregaciones. Las unidades no forman parte de este núcleo porque el marco presenta ambas nociones de forma inconsistente y sólo el tipo permite validar las fórmulas. El argumento referencial se conserva sin transformación dentro de la instancia; metricas_referencial() declara y valida el contrato específico de correctitud semántica y cobertura.

Value

metrica() y especializar() devuelven closures S3; instanciar() devuelve una metrica_instanciada; modelo() devuelve un modelo_calidad; metricas_nucleo() devuelve una lista de métricas genéricas. propiedades_metrica() devuelve un data frame con las propiedades declaradas y si ya fueron configuradas.

Contrato de metodo

El método tiene la firma function(tablas, instancia). tablas es una lista con nombre de data frames, incluso cuando medir() recibió una sola tabla. instancia expone entidad, atributos, configuracion, referencial y declaracion; el método decide cómo interpretar esos vínculos.

Debe devolver un data frame con exactamente una observación por objeto medido y, como mínimo, estas columnas:

- resultado: valor medido. Debe respetar tipo_resultado: lógicos sin NA para "booleano"; números finitos en [0, 1] para "real"; números finitos para "numero_real"; enteros o duraciones no negativos para los tipos homónimos;
- entidad: nombre de la tabla a la que corresponde la medida;
- atributo: nombre de la columna o NA_character_ cuando no corresponde;

- `fila`: posición de la fila o `NA_integer_` para resultados agregados;
- `objeto`: etiqueta legible y estable del objeto medido.

`medir()` hace cumplir el contrato: una salida sin esas columnas, con un `resultado` fuera de su tipo, con más de una observación para el mismo objeto, con una `entidad` que no está ligada a la instancia, con un `atributo` que no es una columna de las tablas que recibió el método —o varias unidas con `+`—, con una `fila` que no es una posición entera o que falta en una métrica por celda o por fila, o con una `fila` fuera del universo que la métrica declara con su propiedad `aplicable` —una métrica con `aplicable` mide sólo ese universo, y `alcance_medidas` cuenta sobre él—, deja la métrica `no_medible` en `cobertura_metricas`, igual que un método que **aborta**: `medir()` avisa y las demás métricas se miden igual.

`fila` es la posición en la tabla que **recibe** el método. Con `aplicabilidad`, `medir()` le pasa la tabla recortada al universo aplicable, y traduce las `fila` de un método propio a la posición en la tabla original; la etiqueta `objeto` queda como la escribió el método.

Las columnas adicionales se descartan. Un `metodo` pasado a `instanciar()` reemplaza el predeterminado sólo para esa instancia. El ejemplo ejecutable muestra la cadena genérica → específica → instanciada completa.

Contrato de propiedades

`propiedades` declara los nombres que puede recibir `especializar()`. Sin `validar_propiedades`, todas son obligatorias. Con un validador propio, éste recibe la lista `configuracion`, debe rechazar combinaciones inválidas y puede devolver sólo el subconjunto activo o añadir valores predeterminados, pero no propiedades ajenas a la declaración. `propiedades_metrica()` permite consultar los nombres aceptados sin inspeccionar atributos internos de las closures.

References

AGESIC (2020). *Marco de trabajo para la Gestión de la Calidad de Datos en Gobierno Digital*, versión 1.6, Presidencia de la República, Uruguay.

Curso CPAP, material *Evaluación de Calidad*: fórmula continua de oportunidad implementada bajo los nombres `GradoOportunidadAtributo*`.

See Also

`referencial()`, `agregar()`, `medir()`, `proponer_modelo()`

Examples

```
nucleo <- metricas_nucleo()
no_nulo <- especializar(
  nucleo$NoNulo, nombre_especifico = "NoNuloEdad"
)
instancia <- instanciar(no_nulo, entidad = "personas", atributos = "edad")
modelo_calidad <- modelo(instancia)
medir(modelo_calidad, data.frame(edad = c(20, NA, 35)))

# Las fábricas también se pueden encadenar: genérica() -> específica().
instancia_directa <- nucleo$NoNulo()(
```

```
  entidad = "personas", atributos = "edad"
)
propiedades_metrica(nucleo$Formato)

# Métrica propia: las dos llamadas encadenadas son especializar e instanciar.
metodo_origen <- function(tablas, instancia) {
  x <- tablas[[instancia$entidad]][[instancia$atributos]]
  filas <- seq_along(x)
  data.frame(
    resultado = !is.na(x) & nzchar(x),
    entidad = instancia$entidad,
    atributo = instancia$atributos,
    fila = filas,
    objeto = paste0(instancia$entidad, "$", instancia$atributos,
                    "[", filas, "]")
  )
}
OrigenDeclarado <- metrica(
  "OrigenDeclarado", "Indica si se declar\u00f3 el origen del registro.",
  "instanciaAtributo", "booleano", orientacion = "conformidad",
  dimension = "Trazabilidad", factor = "Origen documentado",
  metodo = metodo_origen
)
origen <- OrigenDeclarado()(
  entidad = "entrega", atributos = "origen"
)
medir(
  modelo(origen),
  data.frame(origen = c("sistema_a", "", NA), stringsAsFactors = FALSE)
)

# Especialización oficial de teléfono fijo según el formato vigente del PNN.
telefono_pnn <- especializar(
  nucleo$DesactualizacionPorFormato,
  nombre_especifico = "TelefonoFijoPNN",
  expresion_regular = "^[0-9]{8}$"
)

# La métrica oficial es booleana y recibe la fecha límite Tf.
a_tiempo <- especializar(
  nucleo$OportunidadAtributoPorFecha,
  nombre_especifico = "EntregaATiempo",
  fecha_limite = as.Date("2026-06-30")
)
medir(
  modelo(instanciar(a_tiempo, "entregas", "fecha")),
  data.frame(fecha = as.Date(c("2026-06-29", "2026-07-01")))
)

# La extensión continua conserva cuánto margen de utilidad queda.
grado <- especializar(
  nucleo$GradoOportunidadAtributoPorFecha,
  fecha_solicitud = as.Date("2026-06-01"),
```

```
  fecha_fin_utilidad = as.Date("2026-07-01")
)

# Formato(NumeroDocumento, DNIC) se obtiene conectando el validador incluido:
cedula_dnic <- especializar(
  nucleo$Formato, nombre_especifico = "NumeroDocumentoDNIC",
  validador = validar_ci_uy
)
```

`modelo_desde_propuesta` *Materializar una propuesta de modelo de calidad*

Description

Convierte las filas con `incluir == TRUE` en métricas instanciadas y las reúne mediante `modelo()`. No vuelve a examinar los datos ni ejecuta una medición.

Usage

```
modelo_desde_propuesta(propuesta)
```

Arguments

`propuesta`	Objeto creado por `proponer_modelo()`.

Value

Objeto `modelo_calidad` con las métricas que la propuesta declaró, ya instanciadas y reunidas por `modelo()`, listo para `medir()`. No vuelve a examinar los datos ni ejecuta ninguna medición.

See Also

`proponer_modelo()`, `modelo()`, `medir()`

Examples

```
datos <- data.frame(x = c(1, NA, 3))
propuesta <- proponer_modelo(perfilar(datos), datos)
modelo_propuesto <- modelo_desde_propuesta(propuesta)
medir(modelo_propuesto, datos)
```

`normalizacion` *Perfiles de normalizacion para comparar valores*

Description

La normalizacion afecta unicamente la representacion usada para comparar; nunca modifica los datos de entrada. La descomposicion canonica y el orden de sus marcas son siempre activos para que NFC y NFD sean equivalentes en el subconjunto latino cubierto por lupa: las letras latinas precompuestas, las marcas combinantes de U+0300 a U+0332 ordenadas por su clase combinante canónica, y los signos Kelvin y Ångström, que Unicode declara iguales a la `K` y a la `Å`. Las tablas se generan desde Unicode (`data-raw/normalizacion_unicode.R`). Los pasos optativos de ligaduras y ancho completo tambien se aplican por punto de codigo, despues de declarar como UTF-8 los bytes validos, por lo que no dependen de `LC_CTYPE`. Bajar a minusculas tampoco depende del locale: usa un mapa explicito para el latin acentuado, el griego, el cirilico, el armenio y el ancho completo, y lo que queda fuera de ese mapa se conserva. Para comparar, la sigma final griega se iguala a la sigma, como en el plegado de caja de Unicode: una palabra griega en mayúsculas y la misma en minúsculas, escrita con su sigma final, son el mismo valor.

Usage

```
normalizacion(
  minusculas = TRUE,
  espacios = TRUE,
  acentos = TRUE,
  comillas = TRUE,
  puntuacion = FALSE,
  ligaduras = FALSE,
  ancho = FALSE,
  proteger = c("ñ", "ü", intToUtf8(c(103L, 771L)))
)
```

Arguments

`minusculas`, `espacios`, `acentos`, `comillas`, `puntuacion`, `ligaduras`, `ancho`
: Activan el paso correspondiente.

`proteger`
: Grafemas cuyas marcas deben conservarse al quitar acentos. Puede incluir una base seguida de una o más marcas combinantes, como la secuencia `g` seguida por una tilde combinante para la letra guaraní. Se declaran en minúscula y valen también en mayúscula: la eñe de `PEÑA` se conserva igual que la de `peña`.

Details

El paso `espacios` lleva primero al espacio común la tabulación, el salto de línea, el retorno de carro, el avance de página, la tabulación vertical y los espacios Unicode, y borra los controles de transporte; al final colapsa y recorta los blancos. Como va antes que `comillas`, una comilla junto a una tabulación o un salto de línea se trata igual que junto a un espacio.

El paso comillas quita una comilla de apertura cuando abre una palabra —la precede un blanco o el inicio y la sigue una letra— y una de cierre o un apóstrofo cuando la cierra —la precede una letra y la sigue un blanco o el final—; los apóstrofos tipográficos que quedan se escriben '. La comilla simple ASCII cierra pero no abre, porque también es el apóstrofo de una elisión: 'Piedras' se compara como 'piedras, mientras que la misma palabra entre comillas simples tipográficas se compara como piedras.

Value

Objeto de clase normalizacion_lupa: una lista con un elemento logico por paso -minusculas, espacios, acentos, comillas, puntuacion, ligaduras, ancho- y el vector proteger con los grafemas cuyas marcas se conservan. Es una declaracion de como comparar: no toca los datos, y la descomposicion canonica se aplica siempre.

Examples

```
# El perfil por omisión: minúsculas, espacios, acentos y comillas.
normalizacion()

# Comparar sin quitar acentos, para que "papa" y "papá" no se fusionen. La
# eñe y la diéresis ya están protegidas en el perfil por omisión: "pena" y
# "peña" no se fusionan nunca.
perfil <- normalizacion(acentos = FALSE)
perfil$acentos

# La normalización sólo afecta la representación usada para COMPARAR; no
# modifica los datos ni los conteos. Estas tres formas siguen siendo tres
# valores distintos, y así se informan: lo que la normalización habilita es
# que se reconozcan como variantes de la misma cosa al buscar duplicados.
datos <- data.frame(ciudad = c("Montevideo", "MONTEVIDEO", "montevideo"))
salida <- perfilar(datos, normalizar = normalizacion())
salida$columnas$n_distintos  # 3, no 1
```

---

organizacion *Declarar una organización y las colecciones que le pertenecen*

Description

Las granularidades novena y décima del marco —una organización y un conjunto de organizaciones— necesitan un dato que **no está en los datos**: qué bases pertenecen a qué organismo. lupa no lo adivina, así que lo declara quien lo sabe, con el mismo mecanismo que ya usa el conjunto de colecciones.

Usage

```
organizacion(nombre, colecciones)
```

Arguments

nombre	Nombre de la organización. Es su identidad dentro de un conjunto de organizaciones.
colecciones	Colecciones que le pertenecen: un vector de nombres, o una lista de objetos de coleccion() o perfilar_coleccion(). Si la lista tiene nombres, esos nombres mandan sobre el del objeto.

Details

Es opcional. Un análisis de calidad no siempre tiene una organización detrás, y nada obliga a pasar por estos niveles: existen para quien los necesita. Sin declaración, agregar() a "organizacion" se niega y explica cómo declararla, que es distinto de inventar una frontera que nadie nombró.

No pide una conexión. Una colección es una cosa viva —tablas de un motor—, pero una organización es un enunciado *sobre* colecciones, y puede declarar colecciones que viven en motores distintos. Para **agregar**, en cambio, las medidas de sus colecciones tienen que venir de una misma corrida —un id_medicion y una fecha—, porque un número describe un momento: las tablas de motores distintos se miden juntas, y colecciones medidas en momentos distintos se siguen cada una con historico_calidad(). agregar() lo dice así cuando recibe partes de dos corridas.

Value

Objeto S3 organizacion_lupa con nombre, declaradas, n_declaradas y alias: el nombre del objeto de cada colección, con el que agregar() la sigue reconociendo cuando la lista la renombró.

See Also

agregar(), coleccion(), granularidades()

Examples

```
organismo <- organizacion("Organismo A", c("padron", "tramites"))
organismo
organismo$declaradas

# Cuando los nombres siguen una convencion, la agrupacion se deriva de ella
# y despues se declara. `lupa` no aplica la regla por su cuenta -deducir el
# dueno de una base a partir de su nombre es una inferencia, y la frontera
# se declara-, pero tampoco obliga a escribir la lista a mano: la regla la
# pone quien conoce la convencion, y con la forma que quiera.
esquemas <- c("org_a_padron", "org_a_tramites", "org_b_registro", "cruce")
sigla <- ifelse(grepl("^org_", esquemas), sub("_[^_]+$", "", esquemas), NA)
partes <- split(esquemas[!is.na(sigla)], sigla[!is.na(sigla)])
organismos <- Map(organizacion, names(partes), partes)

# Y lo que la convencion no alcanza -aca, una base de un integrador, que no
# tiene dueno unico- se declara aparte, en vez de quedar afuera en silencio.
organismos[["integrador"]] <- organizacion("integrador", esquemas[is.na(sigla)])
vapply(organismos, function(o) o$n_declaradas, integer(1))
```

pack_validadores *Crear y consultar packs de validadores*

Description

Un pack es una lista con nombres de funciones vectorizadas. No se registra en estado global: puede definirse en otro paquete o en un script y conectarse directamente con la propiedad `validador` de `Formato` o con `validadores_personales` en `perfilar()`. Esta forma permite agregar países o dominios sin modificar el núcleo de `lupa`; el pack uruguayo predeterminado de `perfilar()` usa exactamente esta misma puerta.

Usage

```
pack_validadores(nombre, validadores, pais = NULL, descripcion = NULL)

validadores_internacionales()

validadores_uruguay()
```

Arguments

`nombre`	Nombre del pack.
`validadores`	Lista con nombres de funciones que aceptan un vector y devuelven un vector lógico de igual longitud.
`pais`	Código ISO 3166 alpha-2 opcional del país al que pertenece el pack. `NULL` representa un pack internacional o no territorial.
`descripcion`	Descripción breve opcional.

Value

`pack_validadores()` devuelve un objeto S3 `pack_validadores`. `validadores_internacionales()` y `validadores_uruguay()` devuelven packs preparados para usar.

See Also

`especializar()`, `validar_iso3166()`, `validar_ci_uy()`

Examples

```
internacionales <- validadores_internacionales()
internacionales$correo(c("persona@example.org", "incorrecto"))

# Un pack de otro país se construye sin registrar ni cambiar el núcleo.
digito_rut_cl <- function(x) {
  uno <- function(valor) {
    z <- toupper(gsub("[.-]", "", valor))
    if (!grepl("^[0-9]{7,8}[0-9K]$", z)) return(FALSE)
```

```
    cuerpo <- as.integer(strsplit(substr(z, 1, nchar(z) - 1), "")[[1]])
    suma <- sum(rev(cuerpo) * rep(2:7, length.out = length(cuerpo)))
    esperado <- 11L - suma %% 11L  # entero: su texto no sigue a `scipen`
    esperado <- if (esperado == 11) "0" else if (esperado == 10) "K" else
      as.character(esperado)
    identical(substr(z, nchar(z), nchar(z)), esperado)
  }
  ifelse(is.na(x), NA, vapply(as.character(x), uno, logical(1)))
}
chile <- pack_validadores(
  "Chile", list(rut = digito_rut_cl), pais = "CL",
  descripcion = "Validadores mantenidos por el proyecto consumidor."
)
chile$rut(c("12.345.678-5", "12.345.678-4"))
```

`perfilar` *Perfilar un conjunto de datos*

Description

Examina un `data.frame`, `tibble` o `data.table` y devuelve estadísticas generales, métricas por columna, patrones, formatos de fecha y hallazgos accionables. Todas las proporciones se expresan en `[0, 1]`.

Usage

```
perfilar(
  datos,
  nombre = .nombre_de_los_datos(substitute(datos)),
  fecha = Sys.time(),
  muestra = 1e+05,
  max_patrones = 20,
  distinguir_mayusculas = TRUE,
  expandir = FALSE,
  umbral_alta_cardinalidad = 0.5,
  umbral_faltantes_sospechoso = 0.1,
  clave = NULL,
  umbral_faltantes_error = 0.4,
  umbral_patron_raro = 0.05,
  umbral_patron_dominante = 0.5,
  columnas_sin_ceros = character(),
  columnas_no_negativas = character(),
  columnas_opcionales = character(),
  columnas_codigo = character(),
  aplicabilidad = NULL,
  ausencia_estructural = TRUE,
  sentinelas_numericos = c(-9, -99, -999, -9999, 999),
  cadenas_ausencia = NULL,
```

```
  analizar_dependencias = TRUE,
  umbral_dependencia = 0.995,
  umbral_casi_clave_dependencia = 0.8,
  max_columnas_dependencias = 100L,
  datos_personales_permitidos = TRUE,
  proteger_datos_personales = FALSE,
  columnas_personales = character(),
  validadores_personales = NULL,
  umbral_documento_verificado = 0.9,
  muestra_validadores = 1000L,
  duplicados_aproximados = FALSE,
  normalizar = TRUE,
  max_filas_hallazgo = 1000L,
  umbral_orden_columnas = 0.95,
  max_columnas_orden = 20L,
  umbral_solapamiento_orden = 0.1,
  umbral_aritmetica = 0.9,
  min_filas_aritmetica = 3L,
  tolerancia_aritmetica = 1e-08,
  max_columnas_aritmetica = 20L,
  casi_duplicados_vocabulario = TRUE,
  max_proporcion_grupo_vocabulario = 0.5,
  umbral_variante_rara_vocabulario = 0.05,
  min_asimetria_vocabulario_corto = 10,
  min_asimetria_vocabulario = 2,
  min_participacion_dominante_vocabulario_corto = 0.5,
  variantes_equifrecuentes_vocabulario = FALSE,
  max_asimetria_equifrecuente_vocabulario = 2,
  max_comparaciones_dependencias = 200000L,
  max_trabajo_vocabulario = 2e+10,
  max_trabajo_dependencias = 1e+08,
  max_largo_valor_vocabulario = .MAX_LARGO_VALOR_CASI_DUPLICADOS,
  max_celdas_muestra = .MAX_CELDAS_MUESTRA,
  max_bytes_muestra = .MAX_BYTES_MUESTRA,
  avisar_costo_tabla_ancha = TRUE,
  umbral_celdas_aviso_tabla_ancha = .UMBRAL_CELDAS_AVISO_TABLA_ANCHA
)
```

Arguments

datos	Objeto que hereda de `data.frame` —`tibble` y `data.table` entran por ahí— o una matriz de dos dimensiones, que se convierte con `as.data.frame()`. La conversión queda declarada en `meta$entrada_convertida` para que el perfil no aparente haber recibido lo que no recibió.
nombre	Nombre descriptivo del objeto: una cadena no vacía.
fecha	Fecha y hora de la corrida. Se puede fijar para construir series reproducibles; se normaliza a UTC.

`muestra` Máximo de filas usadas para patrones, inferencia de tipos, formatos de fecha y dependencias funcionales. Use `Inf` para analizar todas las filas en esos análisis.

`max_patrones` Máximo de patrones mostrados por columna.

`distinguir_mayusculas`
Si se distinguen mayúsculas y minúsculas.

`expandir` Si se emite un token por carácter en los patrones.

`umbral_alta_cardinalidad`
Umbral sobre la tasa de valores distintos de una columna categórica. No alcanza por sí solo: el hallazgo exige además al menos diez valores distintos, porque con pocos la tasa está dominada por el tamaño de la tabla —dos valores en tres filas dan 0,67 y superan cualquier umbral razonable— y una columna de dos valores no puede tener cardinalidad alta.

`umbral_faltantes_sospechoso`
Umbral inferior de faltantes. El hallazgo se activa al superarlo en sentido estricto.

`clave` Nombres de las columnas que identifican una fila. Cuando se declaran, la trazabilidad de cada hallazgo trae además el valor de esas columnas para las filas señaladas, de modo que el caso se pueda verificar en el sistema de origen sin abrir la tabla. El índice de fila se conserva siempre. La clave declarada se trata como sensible: si la protección de datos personales está activa y alguna de esas columnas se clasifica como personal, sus valores salen enmascarados igual que la evidencia. La comprobación de una clave tiene dos ejes independientes: `meta$clave$unicidad` comprueba si sus combinaciones son únicas con la semántica de R, y `meta$clave$ausencia_nulos` comprueba que sus componentes no tengan valores ausentes. Cada eje declara `verificada`, `refutada` o `no_verificada`; una clave con ambos ejes verificados conserva exactamente el objeto histórico y no agrega metadatos. Cuando un eje falla o no se puede comprobar, `meta$clave$trazabilidad` explica que la localización agrupa con la semántica de R, incluso si el motor SQL trata dos `NULL` como distintos. `hallazgos` separa esos ejes: `clave_con_ausentes` enumera las filas que impiden la garantía `NOT NULL`, mientras `clave_no_unica` sólo informa repeticiones entre filas con la clave completa, el mismo universo que `meta$clave$unicidad`.

`umbral_faltantes_error`
Umbral por encima del cual los faltantes son un error; la igualdad conserva la severidad sospechosa.

`umbral_patron_raro`
Máxima frecuencia de un patrón raro. La frontera es inclusiva: un patrón no dominante con exactamente esa proporción es raro, y sólo los que la superan quedan excluidos.

`umbral_patron_dominante`
Frecuencia mínima del patrón dominante.

`columnas_sin_ceros`
Nombres de columnas donde cero no es admisible.

`columnas_no_negativas`
Nombres de columnas que deben ser no negativas.

columnas_opcionales

Nombres de columnas donde la ausencia no es un defecto. Su universo de completitud son las celdas presentes, y cobertura_diagnosticos declara el recorte. Sirve para el vacío por diseño: un historial con vigencia abierta, una columna que sólo corresponde a algunas filas. La declaración queda en meta$columnas_opcionales —comparar_perfiles() declara su cambio entre dos corridas con la fila configuracion_columnas_opc porque los faltantes de esas columnas pasan a contarse sobre otro universo—, y la NoNulo que proponer_modelo() propone para la columna —cuando tiene faltantes disfrazados— se mide sobre el mismo universo (opcional = TRUE).

columnas_codigo

Nombres de columnas que son **códigos**, no magnitudes: un código de catálogo, un departamento numerado, una clave foránea. El paquete no los distingue por su forma de una magnitud entera que llenó su rango —una edad de 18 a 90 repite valores igual que un código de 1 a 284—, así que sin la declaración los mide como magnitudes. Una columna declarada se perfila por sus valores y no recibe lo que sólo tiene sentido sobre una magnitud: ni límites de Tukey (n_outliers en NA, sin hallazgo outliers ni posible_centinela_numerico), ni la ley de Benford, ni relaciones aritméticas con otras columnas (relacion_aritmetica_columnas), ni media, mediana y desvio, que quedan en NA con estado_resumen_cuantitativo = "omitidos_codigo". Lo que no se evaluó se declara en cobertura_diagnosticos con el motivo. Conserva lo que describe sus valores: minimo y maximo —el tramo de códigos observado, donde aparece un código fuera de catálogo—, moda, n_distintos, los conteos y los faltantes disfrazados. Vale igual por todas las vías que aceptan las declaraciones de perfilar(): perfilar_dbi() —también en su resumen_tabla—, perfilar_por(), perfilar_coleccion() y analizar() por argumentos_perfil. Vale también para el código guardado como texto o como factor, que el paquete lee como número —y ese código no recibe los hallazgos que lo leen como una cantidad escrita como texto y piden convertirlo: ni tipo_declarado_distinto, ni numero_como_texto (un rubro "1.1", un "10,5"), ni unidades_mixtas o monedas_mixtas; tampoco planificar_limpieza() le propone una acción que lo convierta en número—; el resumen de una columna sin resumen numérico —texto que no se lee como número, fechas— no cambia, porque no tiene esas cifras. Un nombre que la tabla no tiene es un error. Dentro de analizar() la declaración llega también a variables —escala nominal, rol categoria, confirmada— y a asociaciones, donde el código se asocia como categoría (eta cuadrado o V de Cramér) y no con una correlación que supone una magnitud. En perfilar_dbi(), la corroboración entre resumen_tabla y perfil_muestra no compara la media, la mediana ni el desvío de una columna declarada: no se calculan. En perfilar_coleccion() la declaración vale igual para todas las tablas: una tabla que no tiene la columna declarada no se perfila (limitación conocida; ver la ayuda de esa función).

aplicabilidad

Lista con nombre por columna, donde cada elemento es una fórmula de un solo lado evaluada sobre datos — por ejemplo list(marca_auto = ~ tiene_auto == "Si"). Las filas donde el predicado no se cumple salen del universo de esa columna: no cuentan como ausencia. Las filas donde el predicado no se puede determinar se declaran aparte, sin contarse ni como aplicables ni como no aplicables. Un valor presente fuera del universo produce el hallazgo valor_fuera_de_aplicabilidad, que es el error simétrico y no tiene otra forma de aparecer. Las filas fuera del

universo —y las indeterminadas— salen también de la ley de Benford y de las relaciones aritméticas, con la misma máscara que usa el resumen.

`lupa` no infiere el universo: si nadie lo declara, toda la columna aplica y el resultado es el de siempre. Declararlo es lo que distingue el vacío por diseño del vacío por error, y sin esa distinción una tabla sana puede informar completitud baja siendo completa.

Hasta dónde llega el universo. Gobierna todo el perfilado: los conteos, las proporciones, los hallazgos y las acciones que propone `planificar_limpieza()`. Medido sobre una columna condicionada de mil filas con universo de trescientas, treinta de ellas vacías: declarándolo, la proporción de faltantes es `0,100`, no hay hallazgo y el plan no propone nada; sin declararlo son `0,730`, sale `faltantes` y el plan propone dos acciones.

El universo tambien llega a la limpieza. El perfil guarda la regla, el plan la lleva y `aplicar()` no toca las celdas que quedan fuera: una `S/D` en una fila donde la columna no corresponde no se convierte en ausencia, y las acciones que marcan o eliminan filas solo consideran ese universo. Si la regla da un valor indeterminado, la celda tampoco se toca. Las conversiones de tipo son la excepción, porque una columna tiene un solo tipo y no se puede convertir a medias: el plan declara en `n_afectadas` y en la justificación que cambia la columna entera. Si el plan se aplica a otros datos donde la regla no se puede evaluar, la acción falla y lo dice, en vez de operar sobre toda la columna.

La medición contra un marco recibe la misma declaración: `medir()` acepta `aplicabilidad` y recorta las filas antes de medir, así que el histórico y la deriva —que consumen mediciones, no perfiles— heredan el número correcto. El `aplicabilidad` de `marco_calidad()` es otra cosa: dice si un factor aplica a datos temporales o geométricos, no qué filas entran al universo.

`ausencia_estructural`

Si se busca evidencia de que la ausencia de una columna es por diseño. `lupa` no infiere el universo ni lo cambia por su cuenta, pero declarar `aplicabilidad` exige saber que existe: quien perfila una tabla con columnas condicionadas sin declarar nada recibe el mismo informe engañoso que si el vacío fuera un defecto. Con `TRUE`, cuando el valor de otra columna decide qué filas tienen ésta, o cuando dos columnas se reparten las filas sin pisarse, se emite `posible_ausencia_estructural` con severidad `ok`, la evidencia medida y la línea exacta que habría que escribir para declararlo. Un umbral se busca primero sobre un corte cada medio por ciento de la columna y después se refina entre todos los valores observados de ese tramo, así que el corte publicado es el primer valor observado del lado donde la columna corresponde. La línea no depende de la sesión: un instante se escribe con su zona (`tz`) —la de la columna, o UTC si no declara ninguna— y un número, con punto decimal y las cifras que hacen falta para volver al valor, cualquiera sea `OutDec`. `n_evaluados` cuenta las ausencias de la columna y `n_afectados` las que la regla explica: las que caen donde la columna no corresponde, sin contar las filas donde el determinante falta. Sugiere; no decide. Las columnas ya declaradas quedan fuera del examen. La regla tiene que cumplirse en al menos el 99 % de las filas: una que un solo caso rompe por debajo de ese corte no se publica, porque ofrecer la línea para declararla sería afirmar de más.

Con el mismo argumento viaja `regla_silencia_ausencia`, también `ok`: avisa cuando una columna declarada opcional o con universo propio sigue casi vacía

dentro de ese universo. La declaración funcionó y por eso el perfil salió limpio; el aviso existe para que eso sea una decisión y no un efecto.

`sentinelas_numericos`

Vector completo de valores numéricos que se interpretan como ausencia. Se normaliza como un conjunto numérico, por lo que el orden, el tipo entero o doble y los duplicados no cambian el resultado. `numeric()` los desactiva; las cadenas de ausencia se siguen evaluando por separado.

La lista por omisión es una sospecha del paquete y se aplica con una reserva: un valor de esa lista que aparece **una sola vez** y cae **dentro** del rango de los demás valores de la columna no se señala, porque una aparición única es un número y no una convención de ausencia. Un `999` suelto dentro de un catálogo de `1` a `3000` no se acusa; el mismo `999` repetido, o un `-9` fuera del rango, sí.

La reserva **sólo** rige para la lista por omisión. Si usted declara sus propios valores —o pide sentinelas_naniar, que es optar por una heurística más ancha—, se cuentan todos, incluida la aparición única. Ésa es también la forma de recuperar el caso que la reserva calla: declarar el valor lo vuelve a contar y además lo saca de los estadísticos del resumen, de la ley de Benford, de las relaciones aritméticas, de las dependencias funcionales y de las asociaciones que se calculen con `analizar()`. En estas relaciones la fila que contiene el valor declarado no se evalúa.

En una columna de texto, una celda es el centinela si el número que escribe, leído con la convención de la columna (signo, ceros a la izquierda, miles, marca decimal, exponente), es exactamente el valor declarado, o si tiene a lo sumo 17 cifras significativas y R lo lee como ese valor: `"-999"`, `"-999.0"` y `"-9.99e2"` son el mismo centinela en toda plataforma. Un texto con más cifras de las que distingue un doble sólo es el centinela si lo escribe exacto.

`cadenas_ausencia`

Cadenas que el usuario declara como ausencia, además de la lista incorporada. Lo declarado atraviesa la guarda del vocabulario: `sd`, `nc` y `nd` sólo cuentan solas en columnas de pocos valores distintos —para no acusar a un código de dos letras—, y declararlas las hace contar siempre. Es el equivalente textual de `sentinelas_numericos`. Como la lista incorporada, se comparan sin los blancos de los bordes y sin distinguir mayúsculas, con el pliegue de caja de `normalizacion()`: `SIN INFORMACIÓN` es `sin información` en cualquier locale.

`analizar_dependencias`

Si se buscan dependencias funcionales entre pares de columnas. Se aplica una sola muestra común a toda la tabla.

`umbral_dependencia`

Cumplimiento mínimo para informar una dependencia.

`umbral_casi_clave_dependencia`

Tasa de valores distintos a partir de la cual un determinante se descarta como casi-clave antes de agrupar.

`max_columnas_dependencias`

Máximo de columnas que intervienen en la búsqueda, cuyo costo crece cuadráticamente.

`datos_personales_permitidos`
: Si la entrega admite datos personales. El valor predeterminado no juzga su presencia: la clasificación se informa con severidad `"ok"`. Use `FALSE` sólo cuando el contrato de la entrega declare que no deben existir.

`proteger_datos_personales`
: Si se reemplazan modas, ejemplos, evidencia y estadísticos de orden concretos cuando `poder_discriminante` es medio, alto o verificado. Las clasificaciones débiles se conservan como aviso pero no suprimen estadísticos, ni la magnitud de Benford, ni la evidencia de sus hallazgos: en ella se tapan sólo sus valores. Es `FALSE` por omisión: la protección es un complemento que se pide, y publicar o no los valores de una base es decisión de quien la usa. La clasificación de posibles datos personales corre igual —es un diagnóstico— y lo que se pide es el enmascarado. `reportar()` tiene su propia protección, que también se pide. El enmascarado es **por columna**: se reemplaza lo que describe a una columna protegida —su moda, sus estadísticos, sus ejemplos y la evidencia de sus hallazgos—, no todas las apariciones de ese texto en la salida. Un valor puede ser vocabulario compartido: `"S/D"` es un centinela de una cédula y a la vez el «sin dato» de `sexo`, y publicarlo en `sexo` no revela nada de la cédula. Lo que **no** tiene columna a la que atribuirse sí se enmascara en toda la salida: un hallazgo de filas duplicadas muestra filas enteras, y los componentes que describen la tabla —`general`, la cobertura, los formatos de fecha— no son de ninguna columna en particular. Los **nombres de columna** nunca se enmascaran, aunque un valor protegido aparezca dentro de uno: un nombre es estructura, existía antes y aparte del dato, y el paquete lo necesita para cruzar el perfil con los datos. Si se enmascararan, `fecha_nacimiento` saldría publicada como `fecha_[valor protegido]`, y `planificar_limpieza()` y `analizar()` abortarían porque los nombres del perfil no coincidirían con los de los datos. Un campo se reconoce como de nombres por su **contenido** —todos sus valores son nombres de la entrada, solos o unidos—, así que la protección de los valores no se afloja: si un valor protegido es igual a un nombre de columna —una planilla pivoteada con una columna de notas por docente y la docente protegida—, su moda sigue tapada, y también los ejemplos de sus patrones, sus niveles observados y la configuración propuesta: un campo hecho de nombres que son valores protegidos es un campo de **valores**. Salvo donde el campo refiere columnas —`columna`, `columna_1` y `columna_2` de las columnas duplicadas, `eliminar`, `determinante`, `dependiente`, `nombres_propuestos` y los demás de su clase—: ahí el nombre es estructura y el plan lo necesita para ejecutar, también cuando el campo lo mezcla con la tabla —`objeto` dice `(tabla)` en las medidas que no son de una columna— o lo escribe detrás de un prefijo del paquete —el `origen` `dependencia_funcional:curso->turno` de la propuesta de `analizar()`—. Un nombre suelto en un campo que también lleva valores —el mapa de una imputación— se sigue tapando, aunque todos los valores del mapa sean nombres de columna: sus `determinante` y `dependiente` son valores de las dos columnas, y `aplicar()` lo resuelve sobre los datos que recibe. El texto del usuario que llega a un campo de vocabulario —la `unidad` de una columna, que es su atributo `units`— no se ampara en el vocabulario: un valor protegido entre límites de palabra se tapa, y una palabra del paquete —`faltantes`, `segundos`— sigue siendo vocabulario aunque un valor protegido

coincida con ella. Y un texto que nombra una columna entre comillas —la prosa del paquete, la SQL que guarda `perfilar_dbi()`— la conserva entera: con «Segundo» protegido, `segundo_nombre` se publica tal cual —y no como `[valor protegido]_nombre`— en los hallazgos sin acción del plan y en la SQL. Lo mismo los **nombres que el paquete deriva** de un nombre de columna: el que propone el plan en `nombres_propuestos` —con `make.names()` y en snake_case— y las marcas `.ausente_<nombre>` y `.outlier_<nombre>` que crean sus acciones. Barridos como texto de los datos, con «Segundo» protegido en `Primer Nombre` la evidencia diría `"Segundo Nombre" -> "[valor protegido].Nombre"`, el plan propondría ese nombre y `aplicar()` se lo pondría a la columna en los datos del usuario. Un nombre que es él mismo un valor protegido, con otra caja, otros separadores, tildes o codificación, no se exceptúa —salvo citado en la SQL que guarda `perfilar_dbi()`, donde es un identificador y se publica igual en `columnas$columna`: tapado, taparía las sentencias enteras del lote—. Y un valor protegido que **contiene** un nombre de columna —el correo `juan_perez@empresa.com.uy` con una columna `juan_perez`— se busca entero, con cualquier caja, antes de apartar el nombre: se tapa el correo y no su dominio ni el correo de otra persona, y los nombres que siguen en el texto quedan cada uno en su lugar. Lo mismo vale para el **vocabulario**: un tipo de hallazgo, una severidad, una estrategia, el nombre de un diagnóstico o de una métrica, o el de un factor de un marco que declaró el usuario, no se enmascaran aunque coincidan con un valor protegido o lo contengan. El principio es el de los nombres de columna: son estructura, no salen de las celdas, y publicarlos no dice nada de ninguna fila —el tipo `constante` aparece porque otra columna es constante, no porque alguien se apellide así—. Si el vocabulario se enmascarara, un nombre de persona contenido en `faltantes_disfrazados` dejaría los hallazgos con el tipo `[valor protegido]` y el plan sin ninguna acción para ellos. La regla es por campo y por contenido: un campo cuyos valores son todos vocabulario del propio objeto queda afuera, uno que mezcla no. Ese alcance tiene un **piso**: un valor de una columna protegida que identifique —seis caracteres o más, el mismo corte con el que la batería de fugas decide qué cuenta como filtración— no se publica en ninguna parte, tenga o no columna a la que atribuirse, y también en los campos numéricos. Sin ese piso, una columna que el clasificador no marcó publicaría los documentos de la protegida con sólo repetirlos: una copia con nombre neutro, un texto libre que los contuviera, o los estadísticos de orden de una columna clasificada con poder discriminante `debil`. Una columna que **copia los documentos protegidos sin su verificador** —la cédula `4.685.467-9` de cada fila guardada como el entero `4685467`, o como texto, en una columna de nombre neutro; también el texto con los decimales que deja una planilla, `"4685467.0"` o `"4.685.467,00"`, con signo o en notación científica (`"+4685467.0"`, `"4685467e0"`), que se compara por el número que el paquete lee en él, con la misma lectura que el resumen, o con espacios de miles y una parte decimal de ceros (`"4 685 467.0"`); un texto con centavos que no son cero no es un documento— se protege entera, como una columna personal: se clasifica `documento_identidad` con poder `alto` cuando al menos tres de sus valores distintos son el documento sin su verificador de un valor protegido —con la regla de los textos, ver abajo; también la cédula de seis cifras, `854321` de `"854.321-2"`: sin ella, una sola entre las de siete dejaría pub-

licar la columna—, son el 90 % de sus distintos —el umbral de `verificado`, que tolera tipeos aislados— y tantos no coinciden por azar con la densidad de esos documentos en el rango de la columna —una probabilidad binomial menor que una en un millón—. Sin protegerla, publicaría sus dos extremos, que son dos cédulas. La regla es por **columna** y no por valor: una columna que coincide por azar —uno de cada cinco importes de siete cifras coincide con un documento sin verificador frente a un millón de cédulas, y hasta tres de cada cuatro frente a cuatro millones— publica sus extremos aunque coincidan. Medido en `data-raw/medir_tapado_de_mas.R` sobre 400 columnas por familia, con uno y con cuatro millones de cédulas protegidas: no se protege ninguna columna de importes de siete cifras —tampoco redondeados a miles, con 10, 50 o 500 valores distintos, aunque por azar coincidan todos— ni de enteros de ocho cifras, y sí todas las copias de 50 valores distintos o más; con el 10 % de tipeos, todas frente a un millón y la mitad frente a cuatro. Una copia de pocos valores distintos frente a muchos documentos —tres, o diez frente a cuatro millones— no se distingue del azar y publica sus extremos; tampoco una copia con más del 10 % de otros valores. Y la regla compara contra los documentos protegidos: sin la columna de la que copia —la tabla de pagos perfilada sin la de personas, fuera de una colección con perfiles conservados— una columna de siete cifras no se distingue de un importe y publica sus extremos. El vocabulario corto no entra en el piso: `"S/D"` sigue publicándose donde describe a una columna que no es personal. Tampoco los **rellenos** de una columna protegida, que ocupan el lugar de un dato y no son el de nadie: el que no tiene letras ni dígitos (`"------"`), y el dígito repetido que se repite cinco veces o más en una columna que sin él es casi una clave —cuarenta `"99999999"` entre cédulas que no se repiten—. En el piso taparían la moda, el mínimo, la mediana y el máximo de un `monto` con su propio `99999999`. Donde el documento se repite —una tabla de movimientos— el dígito repetido sigue en el piso: `2.222.222-2` es una cédula válida. Fuera del piso no quiere decir publicados en la columna protegida: su moda, sus ejemplos y los parámetros del plan sobre ella los siguen tapando, y un centinela que el usuario declaró en `sentinelas_numericos` y aparece en una columna protegida —también como su relleno— sigue en el piso. Sin los datos, el declarado entra siempre, porque no se sabe de qué columna es; con ellos, el que no aparece en ninguna protegida no entra: si entrara, se taparía a sí mismo en `meta` y la acción que lo convierte en un `monto` quedaría sin efecto sin proteger nada. El piso tiene **una excepción declarada**, y es la única: `perfilar_por()` agrupando *por* una columna personal publica sus valores como etiquetas de grupo, porque la etiqueta es el eje del resultado y sin ella los grupos no se distinguen. Lo mismo cuando la columna de agrupación no es personal pero alguna etiqueta contiene un valor protegido de otra columna. No ocurre en silencio —avisa al ejecutar y lo declara en `etiquetas_personales`— y el remedio es agrupar por una columna seudonimizada. El piso alcanza además la **forma sin separadores**: `"771.771-01"` es el mismo documento que `"77177101"`, y la celda entera se enmascara. En los dígitos se busca **el documento en cada número del texto**, con cualquier agrupación: un número une grupos de dígitos con espacios, guiones o puntos —también con un espacio a cada lado—, y con barras, comas sin espacio, apóstrofos, signos más, puntos medios o paréntesis pegados, y se tapa la celda si grupos contiguos forman un documento protegido —`"2901 1234"`, `"099 12`

`34 56"`, `"(2) 487 1234"`, `"4111 1111 1111 1111"`, `"01.23.45.67.89"`—, también con ceros a la izquierda, con los guiones, espacios e invisibles de Unicode —las marcas combinantes, los ignorables y los caracteres de control también—, con los parecidos más comunes del punto, la coma, la barra y el apóstrofo —el acento agudo que el teclado en español pone por apóstrofo—, con la raya o el apóstrofo de un CSV de Windows leído como `latin1`, o escrito con dígitos arábigo-índicos, persas o devanagari. El **teléfono uruguayo con su prefijo de país** —`"598"` o `"00598"` y ocho cifras que empiezan con 2, 4 o 9: `"+598 2901 1234"`, `"00598 99 123 456"` o pegado, `"+59899123456"`, como lo escriben los enlaces de WhatsApp y las exportaciones de un CRM— se prueba por su parte nacional, como si estuviera escrito solo, y entero; el prefijo no se une a otro grupo. También cuando va pegado a otro número por una barra o un guion —`"+59899123456/+59899765432"`, `"29011234/+59899123456"`—: cada número con esa forma se prueba por su parte nacional. Y el celular protegido guardado con el prefijo se reconoce escrito sin él, también guardado como número de planilla —`"59899123456.0"`, `"59899123456,0"` o `"5.9899123456E+10"`, la misma cifra, con cualquier cantidad de ceros —`"59899123456.0000"`, como lo deja un `DECIMAL(15,4)`—, con cualquier exponente —`"5.9899123456e+010"`, como lo escribe el `printf` de Windows, también con un espacio antes, `"5.9899123456 e+10"`— y con separador de miles, como lo muestra el formato "Número" de una planilla —`"59.899.123.456"`, `"59,899,123,456.00"`, `"59 899 123 456,00"`, `"59'899'123'456"`, con el punto medio o sus parecidos —el operador viñeta, el punto medio katakana: cualquier signo que une los grupos de un número— o con el agrupado indio, `"59,89,91,23,456"`: todas las escrituras dan la misma cifra entera—, y al revés: escrito así en un texto se lee por su parte nacional, que sale de esa cifra y no de los grupos como vienen. La cifra escrita así es el documento **entero**, no el parcial: el importe redondo `"5460537.00"` no se lee como un documento sin su verificador. No unen los parecidos raros: la prima de una coordenada sexagesimal, el acento grave, la tilde. Los dos puntos y la barra vertical **no** unen: una razón `"494:2714"` o una lista `"12|345|6789"` no es un documento, y un documento escrito así no se reconoce. Una fecha con hora protegida se reconoce escrita con `T`, `Z`, uno a tres espacios, coma, punto y coma, guion o guion doble, guion bajo, barra, barra vertical, arroba, paréntesis o corchete entre la fecha y la hora, con día y mes en cualquier orden, con la hora separada por puntos, guiones o `h` y `m` (`"21h24m25s"`), en la forma compacta `"20240627T212425"` o con la hora compacta tras la fecha con separadores (`"2024-06-27T212425"`); no en palabras, con a.m./p.m., con cuatro espacios o más ni con el año en dos cifras —también con las formas chicas o de ancho completo de esos signos, o con la barra partida por barra vertical—. El documento **parcial**, de siete cifras o más, se busca con más cuidado porque es el que coincide por azar: sin su verificador, fuera de la parte de un decimal, de un número con forma de dirección IPv4 —con o sin máscara CIDR— y de un importe con miles de cuatro grupos o más (`"3.919.024.166"`), que se prueban enteros —el importe, también sin su primer grupo: es el prefijo de país de un celular agrupado de a tres, `"598.099.123.456"`, que de otro modo se publicaría— (`"4.123.456"` frente a `"4.123.456-7"`, el verificador mal tipeado, el documento seguido de un año), y sin su primer dígito sólo detrás de un asterisco o una equis (`"*.123.456-7"`, `"X.012.345-8"`, `"X2345678"`) —también los asteriscos de Unicode, todo signo

que Unicode llama asterisco salvo el combinante de abajo, que el pliegue borra como a una tilde, y el asterisco pegado a una letra, `"CI*123.456-7"`—; la equis seguida de un espacio no es comodín, porque `"92 x 6931234"` es una multiplicación; un número redondo (`1.000.000`) no cuenta como parcial. Es un **decimal** el número cuyo último signo es un punto o una coma con otra cantidad de cifras que tres, y antes un entero o miles separados con otro signo: `"4.123.456,7"` es un importe, `"4.123.456.7"` no; su parte decimal no se pega al resto cuando es la de un importe con miles o la de un decimal suelto —una coordenada, un par lat lon—, pero sí cuando el número es otra cosa, como `"(02) 901.1234"`. Una fecha con un año entre 1800 y 2100 —también con espacios y el año al final, `"16 01 2002"`; con el año adelante no, porque `"2012 11 05"` es también un teléfono escrito de a pares—, una hora —con su fracción de segundo: hasta siete cifras o nueve tras un punto, nunca ocho, y no seguida de un guion y una cifra, que sería un verificador—, un ISBN y una lista separada por comas no son documentos. El ISBN-10 —con guiones, espacios o puntos— sólo si su dígito de control es válido o lo precede `ISBN`: un NIT `"900-123-456-7"` tiene su forma, y uno de cada once documentos de diez cifras escritos así valida por azar y no se reconoce. Una fracción de nueve cifras tras un punto se lee como nanosegundos: un documento con un cero adelante pegado así a una hora no se reconoce. En la prosa del paquete los números son conteos suyos, y ahí la regla mira sólo lo que la prosa cita entre comillas y el código que sugiere entre comillas invertidas, y tapa la cita o el número, no la frase. **El parcial tiene un costo, medido**: un número entero de siete cifras que coincide con un documento protegido sin su verificador no se distingue de él, y se tapa. Con un millón de cédulas, 300.000 teléfonos fijos y 300.000 celulares protegidos, eso le pasa a uno de cada cinco números de siete cifras en ejemplos, modas y evidencia —en cualquier escritura: con puntos o apóstrofos de miles, con un guion, tras un numeral o tras una equis seguida de un espacio (`"92 x 6931234"`); tras un asterisco, o tras una equis pegada (`"serie X6276963"`, que es como se escribe el documento recortado sin formato, `"X2345678"`), a uno de cada cuatro—, y a uno o dos de cada cien enteros de ocho cifras, que coinciden con un documento entero —también a los importes de ocho cifras con centavos, `"52317219.09"`, 1,8 % sobre 4.000 con las mismas cifras enteras: su parte entera es la de un documento entero; y a los escritos con separador de miles o en notación científica, con o sin un espacio antes del exponente —`"52.317.219,00"`, `"5.2317219 e+07"`—, también 1,8 %: son la misma cifra entera; y a tres de cada cien de los que empiezan con 9 —entre 90 y 99,9 millones: un importe, un código—, en cualquier escritura (`"98276375"`, `"98.276.375,00"`: 3,3 % sobre 4.000), porque tienen además la forma de un celular protegido sin su cero—; las coordenadas, los p-valores, los importes de siete cifras con decimales —también los redondos, `"5460537.00"`, `"5460537,00"` o dentro de una frase; con un solo cero, `"5460537.0"`, dos de cada cien, que se leen también como la cédula con su verificador tras un punto—, las horas, los ISBN —también el ISBN-10 con espacios o puntos—, las fechas con espacios, las listas separadas por comas con espacio o por barras verticales, las razones no se tapan; los importes con miles de cuatro grupos o más, uno o dos de cada mil —su cola sin el primer grupo coincide con un documento cuando empieza con un cero—; una dirección IP —también con su máscara CIDR, `"10.4.123.234/24"`, o su

puerto, y en un rango de redes `"10.4.1.0/24-10.4.9.0/24"`— sólo cuando coincide entera, menos de una de cada cien; y una coordenada escrita con un cero adelante (`"-056.092164"`), que tiene la forma de un celular con punto (`"099.123456"`), casi una de cada cien. Lo mismo vale, por la misma razón, para lo que se une como el documento: una lista separada por espacios, ocho de cada cien; un rango de años `"2004-2017"` o con la raya de un CSV de Windows, dos números en dos líneas, una lista con tabuladores o con comas sin espacio —también la coma ideográfica—, entre una y tres de cada cien; o una versión `"v4.16.4302"`. Un teléfono fijo que no es de nadie, tres de cada cien —2,8 %, sobre 4.000 fijos—, escrito de cualquier forma: con espacios o guiones (`"4656 4570"`), con su prefijo de país (`"+598 4499 0917"`) o pegado, y como número de planilla con su prefijo (`"59.822.877.006"`): 3,2 % sobre otros 4.000, lo mismo que pegados (`"+59822877006"`). Coincide con un documento o con un teléfono protegido, y como se prueba su parte nacional, las mismas cifras se tapan en todas las escrituras, salvo la de planilla con los miles y los decimales en cero o con espacios (`"59.822.877.006,00"`, `"59 822 877 006"`), 4,4 %: como en el celular, sus primeras ocho cifras tienen además la forma de una cédula entera. Las familias del reproductor que sortean 400 fijos por escritura dan entre 1,5 y 5,5 %: es el sorteo, no la escritura. El prefijo no se une al primer grupo: `"598"` y ese grupo formarían una cédula de siete cifras sin su verificador. Y una lista de dos teléfonos unidos por una barra —`"+59894364556/+59849290458"`—, entre cinco y ocho de cada cien, como la misma lista con `" / "`: cada uno se prueba por su parte nacional. Un celular que no es de nadie, entre dos y tres de cada cien —2,0 a 2,5 % con 400 por escritura, pegado con su prefijo, con espacios o sin el prefijo; 3,0 % sobre 4.000 con las mismas cifras, pegado (`"+59895257283"`) o como número de planilla con separador de miles (`"59.895.257.283"`)—: coincide con uno protegido. Con los miles y los decimales en cero, o con espacios (`"59 895 257 283"`), 4,4 %: sus primeras ocho cifras tienen además la forma de una cédula entera. Con el punto medio como separador de miles, 5,3 % sobre otros 4.000 que pegados dan 4,0 %, y 7,8 % con los decimales en cero: como con espacios, sus grupos se prueban también sueltos. Con el agrupado indio (`"59,89,52,57,283"`), uno de cada cinco —20,3 %, y 32,0 % con `".00"`—: sus grupos de dos forman números de siete cifras que coinciden con un documento sin su verificador. Y, por el mismo parcial, una versión de cuatro partes (`"v4.62.660.4361"`), trece de cada cien, la de cinco partes (`"v2.76.527.6327.3"`), quince de cada cien, y un expediente `"exp 2023-6-880041"`, uno de cada seis: su tramo `"660.4361"` o `"6-880041"` es un número de siete cifras con un signo que une, y no se distingue del documento escrito así. Y por la regla de la hora, una fracción de ocho cifras (`"18:54:14.80786977"`) no es fracción —así no se traga un documento pegado— y se tapan cuatro o cinco de cada cien; lo mismo un número de diez cifras con la forma de un ISBN-10 y su dígito de control inválido. La medición se rehace con `data-raw/medir_tapado_de_mas.R`. Las tildes, la caja y la escritura también son cosméticas: `juan.perez` se enmascara frente al nombre protegido «Juan Pérez», y el nombre pegado a otras letras dentro de un correo o un alias, también. La comparación usa un pliegue generado de Unicode —caja de todo alfabeto, letras con y sin diacríticos, ligaduras, ancho completo y medio ancho, letras matemáticas, y las letras modificadoras que hacen de acento o de apóstrofo

pegadas a una letra, como el agudo U+02CA, el circunflejo bajo U+A788, la tilde vertical U+2E2F, la prima U+02B9 o el apóstrofo U+02BC escritos en lugar de la tilde—, compara el texto publicado también con los escapes del paquete deshechos (un espacio duro sale `<U+00A0>`) —el valor protegido nunca se desescapa: una barra literal es parte del valor—, y lee lo que no es UTF-8 válido de las dos maneras, byte por byte como CP1252 conservando las secuencias válidas y entero. El **correo escrito con palabras** —`juan(at)gmail.com`, `[at]`, `{at}`, `_at_`, `(arroba)`, `(a)`, `juan arroba gmail punto com`— se lee como correo: la palabra es la arroba sólo si la sigue un dominio con su punto, y `dot` o `punto` son el punto sólo en un texto que ya tiene arroba —también la arroba escrita, `juan@gmail punto com`—. Se lee con los espacios de Unicode como espacios —el duro, el ideográfico, el salto de línea de C1 (U+0085)— y los signos de ancho completo como los de ASCII —los paréntesis o la arroba anchos, también los paréntesis blancos anchos (U+FF5F, U+FF60)—, y con todas las demás formas de compatibilidad de un signo ASCII —cada carácter que la normalización NFKC de Unicode lleva a un solo signo ASCII—: el punto ideográfico y el de medio ancho (U+3002, U+FF61), las variantes chicas de la coma, el punto, los paréntesis o la arroba (U+FE50 a U+FE6B), la barra partida, los paréntesis superíndice y subíndice (U+207D, U+207E, U+208D, U+208E), las formas de presentación vertical (U+FE10 a U+FE48) y el punto de guía (U+2024). Se toma la normalización entera y no una lista de las formas conocidas: con una lista, el correo escrito con los paréntesis superíndice alrededor de `at` se publicaría. Las letras de otro alfabeto que se parecen —una «a» cirílica (U+0430) en lugar de la latina— no son el mismo signo y no se leen, ni una letra encerrada en un círculo (U+24D0): es otra forma de la letra, no un signo. Y en cualquier codificación: un texto `latin1` con una tilde se lee como el mismo texto en UTF-8. Medido sobre 3.000 correos protegidos: los no protegidos escritos así se tapan lo mismo que con la arroba común —cuatro de cada mil, por los separadores cosméticos—, y ninguna de 3.015 frases con esas palabras, mil de ellas con una arroba y el punto en palabras. **No** se leen otras ofuscaciones: el punto entre corchetes o paréntesis (`[at]...[.]`, `(.)`), `<at>`, `"at"`, `-at-`, `/at/`, el punto con espacios alrededor (`empresa . com`) ni un dominio con guion bajo; un correo protegido escrito así se publica. Las marcas del paquete se apartan del texto, y un valor protegido que contiene una se busca entero antes de apartarla. **Una fecha de calendario sola no entra en el piso**, tampoco escrita como `AAAAMMDD` o con el año en dos cifras en una columna de esas fechas, ni como fecha-hora ISO a medianoche: en una tabla de miles de personas casi todo día es el cumpleaños de alguien, y con la fecha de nacimiento protegida se taparían los estadísticos de todas las demás columnas de fechas. La columna de fechas protegida se sigue protegiendo entera; una fecha con hora sí cuenta. Se decide **por la columna**: en una columna protegida que no es de fechas —un teléfono fijo `"2012-11-05"` entre otros teléfonos— el valor con forma de fecha sí entra en el piso, escrito como está guardado (la regla de dígitos borra las fechas antes de buscar, así que `"2012/11/05"` no lo encuentra); una columna es de fechas si es `Date` o `POSIXt`, o si la mitad o más de sus valores distintos tienen forma de fecha —con el año en cuatro cifras o en dos, o compacta `AAAAMMDD`—. Es una sola regla, en memoria y en `perfilar_dbi()`: una columna de nacimientos con el 60 % en `dd/mm/aa` es de fechas, y sus valores no tapan la moda ni los

ejemplos de las fechas de otra columna. En la columna de fechas, lo que no tiene forma de fecha —un relleno— sigue entrando al piso. En la **prosa del paquete** —descripción, sugerencia, motivo, cómo resolverlo, justificación— se exige además que la variante no tenga una letra pegada antes o después: con los nombres de millones de personas como valores protegidos, alguno aparecería cruzando palabras de esas frases y las taparía enteras, también en columnas no personales. Lo que esa prosa cita entre comillas dobles —el nivel de un determinante en la sugerencia de `posible_ausencia_estructural`— no es prosa sino un valor: se compara con la regla de los valores y se tapa sólo la cita, para que la sugerencia se siga leyendo. Y si el determinante es una columna protegida, la sugerencia no reproduce el criterio. **Las palabras y las marcas del propio paquete no se tapan**: con columnas de nombre y de apellido separadas, los valores protegidos son palabras sueltas, y algunas lo son también del paquete —`Blanco` frente a la marca `<blanco>`, `Máximo` frente a la clave `grupo_maximo`, `Patrón` en «el patrón dominante»—. El léxico se calcula al instalar, desde las cadenas del código del paquete: sus marcas no se comparan en ningún campo, y sus palabras no se tapan en su prosa salvo dentro de una cita. El nombre completo en el texto de otra columna se sigue tapando. Lo que no es del paquete, también: con nombre y apellido protegidos en columnas separadas, un lugar que comparte una palabra de seis letras o más con uno de ellos se tapa —el departamento `"Maldonado"` entre los ejemplos, la calle `"Av. Rivera 1343"`, el `"Instituto Rivera"`; seis de los diecinueve departamentos de Uruguay son apellidos así, y ciudades como Mercedes o Dolores son nombres de pila—. Medido sobre 5.000 personas con esos nombres y apellidos, se tapan trece campos de cuatro columnas que no son personales. Es el costo del piso y no se afloja: sin él, «llamó la Sra. Rivera» publicaría el apellido. Y lo aplican también `analizar()`, `distribucion_valores()`, `clasificar_variables()` y `detectar_discordancias()`, así que no publican lo que `perfilar()` tapa sobre la misma tabla.

`columnas_personales`

Columnas que traen datos personales, declaradas por quien conoce el dato. `c("cod_benef", "apodo")` las nombra sin decir de qué tipo son; `c(cod_benef = "documento_identidad")` además lo dice. Lo declarado gana sobre lo inferido y no se vuelve a examinar.

Existe porque el léxico de nombres de columna **no puede ser completo**: una columna con documentos se puede llamar de cualquier manera, y ninguna lista de nombres frecuentes la va a reconocer. Es el mismo patrón que `columnas_opcionales`, aplicado a la otra decisión que el paquete no puede tomar solo.

`validadores_personales`

Pack o lista nombrada de funciones que reciben un vector de texto y devuelven un lógico de igual longitud. `NULL` usa `validadores_uruguay()` por compatibilidad; `FALSE` o `numeric()` desactiva la verificación de documentos. El nombre del mejor validador queda en el fundamento de la clasificación.

`umbral_documento_verificado`

Proporción mínima de valores que debe aceptar un validador para clasificar una forma de documento como `verificado`. Por defecto es `0.9`.

`muestra_validadores`

Máximo de valores usados en el filtro preliminar de cada validador. Si la pro-

porción preliminar ya queda bajo el umbral no se valida la columna completa; use `Inf` para revisar todos desde el inicio.

`duplicados_aproximados`
: `FALSE` por omisión. Use `TRUE` o una lista de argumentos para ejecutar `detectar_duplicados_aproximad` y añadir sus pares y hallazgos al perfil. Es un análisis acotado y opcional porque no afirma identidad ni debe encarecer todas las corridas.

`normalizar`
: Perfil de comparación que se conserva en `meta$normalizacion` y que heredan los análisis de duplicados y claves cuando no reciben uno explícito. Cambia sólo la representación usada para comparar.

`max_filas_hallazgo`
: Tope de índices de fila que conserva cada trazabilidad disponible. Por defecto es `1000`; cuando se supera, el estado queda como `truncada` y el total se conserva. Use `Inf` sólo si necesita desactivar explícitamente el tope.

`umbral_orden_columnas`
: Cumplimiento mínimo de una relación de orden entre columnas comparables. Se usa `0.95` por omisión; con menos de 20 filas comparables se permite una sola inversión para no descartar tablas pequeñas. El alcance efectivo queda en `meta$orden_columnas`.

`max_columnas_orden`
: Máximo de columnas numéricas o temporales que se comparan entre sí para detectar relaciones de orden. Las columnas que exceden el límite se conservan en `meta$orden_columnas$columnas_omitidas` y el diagnóstico se declara `no_evaluado` en `cobertura_diagnosticos`. `comparar_perfiles()` no presenta como `resuelto` un hallazgo anterior que quedó fuera de ese alcance.

`umbral_solapamiento_orden`
: Solapamiento mínimo de los rangos intercuartiles para considerar que dos columnas representan magnitudes comparables. Por defecto es `0.1`: al menos una décima parte del rango intercuartílico más ancho debe ser común a ambos. Esto evita interpretar como restricción fila a fila un orden explicado sólo por escalas separadas. Si no hay ese solapamiento, una brecha con IQR exactamente cero conserva el par porque la mitad central sostiene el mismo desplazamiento fila a fila. No se aplica una tolerancia oculta. Use `0` para desactivar el filtro de magnitud. Ambos criterios se publican en la evidencia. Los pares descartados se cuentan en `meta$orden_columnas$pares_descartados_magnitud` y los recuperados en `meta$orden_columnas$pares_rescatados_brecha_estable`. Los pares que involucran una columna cuyo tipo o formato se descubrió sobre una muestra y cuya conversión dejó valores afuera no se comparan; el alcance los conserva en `columnas_conversion_parcial`, `n_valores_excluidos_conversion` y `pares_descartados_conver`

`umbral_aritmetica`
: Proporción mínima de filas comparables que deben satisfacer una identidad dentro de `tolerancia_aritmetica` para reconocer una regularidad aritmética entre columnas numéricas. El valor por omisión es `0.9`. Una vez reconocida la relación se informan todas sus discrepancias, sin un segundo filtro por su cantidad absoluta. La proporción y el criterio efectivos se publican en cada evidencia.

`min_filas_aritmetica`
: Mínimo de filas comparables necesario para evaluar una candidata aritmética. Por omisión es `3`.

`tolerancia_aritmetica`
: Tolerancia numérica relativa escalada usada al comparar un valor observado y uno esperado. Por omisión es `1e-8`; el criterio completo y el valor efectivo se declaran en cada evidencia. Como es relativa a la escala del total, con montos de `1e9` vale entre 10 y 20 unidades, y una columna que vive por debajo de eso completaría cualquier suma entre dos columnas casi iguales. Por eso una identidad que se cumple igual sin uno de sus sumandos no se publica, salvo que se cumpla exacta —con la única diferencia del redondeo del tipo double, `4 * .Machine$double.eps` relativo— en la proporción de `umbral_aritmetica`; la evidencia lo dice, y las descartadas se cuentan en `meta$aritmetica_columnas$identidades_descarta` junto a la resolución usada en `resolucion_exacta`. Es el mismo criterio que el filtro de magnitud de las relaciones de orden. Con el valor por omisión, una identidad aditiva cuyas tres columnas guardan decimales fijos —todos sus valores son múltiplos de un mismo cuanto decimal, como los centavos o los enteros— acota la tolerancia a la mitad de ese cuanto: entre esos valores una diferencia es un múltiplo del cuanto, y la tolerancia relativa tragaba centavos en los importes grandes (un peso entero sobre 1e8). La cota se usa si el total guarda exactamente los decimales de la suma o si la identidad se cumple exacta, salvo el redondeo del tipo double, en la mayoría de las filas, y sólo si los valores tienen menos de 15 cifras significativas en ese cuanto —con centavos, hasta importes de 1e12; con enteros, hasta 1e14—; por encima la evidencia dice que la tolerancia no se acota y cuánto vale. Una identidad que la tolerancia relativa reconoce y la cota no se descarta y se cuenta en `meta$aritmetica_columnas$identidades_descartadas_cuanto`.
En una suma la tolerancia se mide también sobre los sumandos —`max(1, |observado|, |esperado|, |` porque el error de sumar en doble es relativo a ellos; si el total queda por debajo de esa tolerancia, la identidad se publica sólo si es exacta. Una tolerancia declarada no se acota. En una proporcionalidad, con el valor por omisión, una columna con decimales fijos cuyo redondeo la tolerancia relativa supera en alguna fila —o cuya relación la relativa ya reconoce acusando filas— se lee como la respuesta redondeada. Con respuestas más chicas frente a su cuanto, donde la relativa no reconoce la relación, se lee igual si hay al menos 20 bases y 20 respuestas distintas y la relación explica más filas que la moda de la respuesta: entre columnas independientes chicas, sin esas dos condiciones, la mitad del cuanto publicaba proporcionalidades espurias. En la lectura, la mitad de su cuanto —la resolución de su redondeo, como 0,05, 0,25 o 5, y no la potencia de diez de sus decimales: primero la de todos sus valores y, si con esa no alcanza el criterio, la de al menos el 90 % de las filas, medidas sobre los valores distintos y sin el cero —con la cota de azar contada en registros distintos: un registro repetido es uno, y cien importes que el redondeo junta en cinco valores son cien—; un valor fuera de ella cuenta como discrepante, y unas pocas filas con más decimales que las demás no la hacen pasar por cifras significativas—, inclusiva, es la tolerancia de todas las filas, y `k` es la constante en la que cada cifra explica al menos una fila más: la de más filas dentro menos cifras —cinco cifras más que solo se tragan dos filas rotas de importes grandes no se pagan, y una que explica una fila sana sí—, entre las que explican al menos `umbral` de las filas de la mejor y con al menos una fila por cifra. La recíproca corta también es una constante corta: `y ~= x / 7` cuesta la cifra del 7, como `k` cuesta las suyas, y le gana a `0.143`, que dejaba afuera filas sanas; con costos iguales gana la forma directa, y una recíproca que termina se escribe como ese decimal (`1/0.4`

es `2.5`). En un empate gana la más larga solo si cada fila que agrega está en el cuarto de mayor magnitud de las que explica o a menos de dos medios cuantos de la corta, no pierde ninguna de las de la corta que le quede a menos de esa distancia, y la corta no explica exactas las suyas: un truncado falla primero las filas grandes y deja afuera otras por un pelo, y una fila rota en el medio no compra una cifra. Una que explica más filas que otra igual de corta o más y pierde algunas de las de esa paga dos veces cada fila que pierde, salvo que alguna de las que solo explica ella le quede cerca a la otra. Y una corta que explica exactas todas sus filas no la gana ninguna más larga que las contiene: con `y = 0.18 * x` exacto, `1/5.555` se tragaba tres rotas grandes. Entre los decimales del mismo largo, el que deja más filas justo en el empate del redondeo y después el más cercano a la constante de mínimos cuadrados (`0.0725` y no `0.0724` en `round(cant * 0.0725, 2)`). Gana otra más larga si explica casi todas las filas que esa deja a más de dos medios cuantos —todas salvo `1 - umbral`, y al menos dos— sin perder ninguna de las que esa explica, como un tipo de cambio largo contra su truncado. Si las dos columnas admiten esa lectura, con soporte las dos y la tolerancia de una el doble o más que la de la otra en la misma columna, se comparan con la fina —una fila que solo cae adentro con la gruesa no cuenta— y la gruesa gana solo si explica más filas y paga sus cifras, contadas en la fina —las de su forma corta, directa o recíproca—; si no, la de la constante claramente más corta (`1.22` y no `0.8196...`), la que tiene soporte —con cotas parecidas, contado en las filas que la gruesa explica con la fina—, la que explica más filas pagando sus cifras y la de la resolución más fina, en ese orden. Con cotas parecidas, cada una cuenta las filas que explica con su propia tolerancia, salvo la franja entre las dos: ahí la gruesa cuenta sólo si su redondeo la explica mejor que las rotas que caen justo afuera de su ventana, medidas con la constante de más cifras de las dos —en `y = round(x / 1.105)` en enteros la franja es redondeo de `y`, uno de cada diez residuos, y `x ~= y * 1.105` la acusaba; en `y = round(1.22 * x)` con rotas por una unidad, la fila de la franja es una rota más—; las filas fuera de la retícula de alguna de las dos no cuentan para ninguna, y en la gruesa cuentan como explicadas las que caen justo en su medio cuanto, un empate del redondeo: `y = round(0.9 * x)` se lee `y ~= x * 0.9` sin acusar las filas que R redondea al par; con una tolerancia el doble de la otra esos empates no cuentan, porque ahí cae una fila corrida un cuanto. Con todo igual, la de menos cifras, la que acusa más y la directa (`y ~= x * 3.75` y no `x ~= y / 3.75`). La que la tolerancia relativa reconoce y ninguna resolución explica se descarta, se cuenta en `meta$aritmetica_columnas$proporcionalidades_descartadas_cuanto` y deja una fila en `cobertura_diagnosticos` con la relación y el motivo; si la resolución de alguna de las dos columnas es compatible con la relación, se publica con la tolerancia relativa. Lo mismo vale para una identidad descartada por la cota del cuanto. En cualquier camino, una proporcionalidad se sostiene en las filas que distinguen `k`: tiene que alcanzar el criterio en las filas con la base distinta de cero, donde el par de valores más repetido pesa a lo sumo `1 - umbral_aritmetica` de ellas, y con al menos `min_filas_aritmetica` filas fuera de ese par cumpliendo —180 filas en cero, o en un mismo par, no hacen una relación con las otras 20—. Las filas en cero y las del par más repetido no cuentan como cumplimiento más allá de ese tope, así que una relación real con muchas de ellas y más de una fila de cada diez rota entre las demás no se publica.

Una que alcanza el criterio sin ese soporte no deja fila de cobertura —en una tabla rala serían una por par de columnas— y se cuenta en `meta$aritmetica_columnas$proporcionali` también cuando una identidad entre las mismas columnas la vuelve redundante. Una columna que varía menos que la tolerancia relativa de su magnitud se evalúa con la cota del cuanto cuando la identidad la admite; si no, una fila de `cobertura_diagnosticos` por columna dice qué relación reconocía la tolerancia y por qué no se publicó. `meta$aritmetica_columnas` se publica cuando hay una relación, un recorte o una relación descartada —por escala, por la cota del cuanto o por falta de soporte—.

`max_columnas_aritmetica`
: Máximo de columnas numéricas que intervienen en la búsqueda aritmética, cuyo costo crece cúbicamente. Por omisión es `20`. Si se omiten columnas, `cobertura_diagnosticos` declara el recorte y `meta$aritmetica_columnas` conserva los conteos de combinaciones.

`casi_duplicados_vocabulario`
: Lógico que activa el diagnóstico de variantes casi duplicadas dentro del vocabulario de cada columna de texto. Por defecto es `TRUE`; `FALSE` lo omite sin afectar los demás hallazgos. La distancia es una señal heurística, no una prueba de identidad: Jaro–Winkler puede agrupar nombres de calles o códigos que sólo comparten un prefijo o un sufijo. En vocabularios heterogéneos revise la evidencia como sospechosa, declare una regla de dominio o use `FALSE` para desactivar este diagnóstico.

`max_proporcion_grupo_vocabulario`
: Proporción máxima del vocabulario que puede abarcar el grupo mayor para entregar grupos de variantes. Por defecto es `0.5`; si se supera, el alcance declara que el diagnóstico no aplica en vez de entregar un bloque que abarque casi toda la columna. El diagnóstico se ejecuta por columna y no deduplica columnas con el mismo vocabulario: cada una conserva su propia fila de cobertura y sus propios hallazgos. El costo de comparar vocabularios idénticos puede repetirse.

`umbral_variante_rara_vocabulario`
: Proporcion maxima de la columna que puede ocupar una variante breve para abrir la comparacion por una edicion.

`min_asimetria_vocabulario_corto`
: Razon minima entre la frecuencia de una forma dominante y una variante breve para abrir la comparacion por una edicion.

`min_asimetria_vocabulario`
: Razón mínima entre la frecuencia de la forma dominante y la de la variante para abrir un grupo por la vía general de distancia. Por omisión `2`. Una asimetría de `1,5` significa que las dos formas son casi igual de comunes, que es evidencia muy floja de una errata: medido sobre tablas limpias y sobre erratas sembradas, los falsos positivos están entre `1,0` y `1,5` y las erratas reales desde `9,0`.

`min_participacion_dominante_vocabulario_corto`
: Proporcion minima de la columna que debe ocupar la forma dominante en la comparacion por una edicion.

`variantes_equifrecuentes_vocabulario`
: Si se informa el diagnóstico `variantes_equifrecuentes_vocabulario`, que señala dos formas cercanas que se reparten la columna sin que ninguna sea

dominante —el error sistemático de dos operadores, una plantilla rota o una migración parcial—.

Está apagado por omisión y la razón está medida: sobre la batería de 31 tablas limpias produce un grupo sospechoso donde no hay defecto. Lo que lo dispara no es el tamaño de la tabla sino el reparto: dos formas parecidas con frecuencias del mismo orden lo activan igual con cuatro filas que con quinientas, y en una tabla chica ese reparto sale por casualidad más seguido. Es aditivo: encenderlo no cambia ni pierde ninguna detección de `casi_duplicados_vocabulario`.

El límite que deja de cubrir se declara igual, encendido o apagado, en `n_grupos_sin_variante_rara`.

Cada par cercano se declara una sola vez: si `casi_duplicados_vocabulario` ya publicó sus dos formas en un mismo grupo, el par no se cuenta ahí ni se publica otra vez como equifrecuente, aunque otra forma cercana cuelgue de ese grupo. Pasa con una clase exacta de normalización unida por distancia a su vecina, y cuando la asimetría llega a `min_asimetria_vocabulario` sin pasar `max_asimetria_equifrecuente_vocabulario`. La forma que comparten se nombra en los dos grupos y cada uno declara sus pares: con `Montevideo` (100), `Montevido` (50) y `Montividoo` (50), `casi_duplicados_vocabulario` publica `Montevideo` y `Montevido`, y el grupo equifrecuente, `Montevido` y `Montividoo`, con un solo par. Lo que el piso de asimetría deja afuera sí se cuenta.

`max_asimetria_equifrecuente_vocabulario`

Razón máxima entre la frecuencia mayor y la menor para considerar que dos formas se reparten la columna. Con `2`, `40` contra `5` no entra y `5` contra `5` sí. Las grafías que la normalización junta cuentan juntas: `MONTEVIDEO` (10) y `Montevideo` (40) contra `Montevido` (50) es un empate, el grupo publica las tres formas con sus frecuencias y su asimetría es 1. El tope es del grupo, no de cada par: `Montevideo` (25), `Montevido` (20) y `Montividoo` (10) están a 1,25 y 2 de a pares, pero a 2,5 de punta a punta, y no se agrupan. Sus pares se declaran aparte en la misma fila de `cobertura_diagnosticos`, como pares cercanos que forman cadenas sobre el tope.

`max_comparaciones_dependencias`

Tope de pares determinante-dependiente que se comparan. El costo de las dependencias es del orden de `columnas^2 x filas` y empeora con determinantes casi únicos; cuando el presupuesto se agota, lo comparado se informa y lo que quedó sin comparar se declara en `cobertura_diagnosticos`, nunca como cero.

`max_trabajo_vocabulario`

Tope de comparaciones de carácter para las comparaciones de distancia del vocabulario. Comparar dos valores cuesta del orden del producto de sus largos, así que el trabajo es la suma de ese producto sobre todos los pares. Contar pares por longitud media subestima las cadenas largas: medido así, el mismo presupuesto rinde 5,3 millones de unidades por segundo con valores de 900 caracteres y 44 millones con valores de 40. Se combina con el límite interno de pares del detector y manda el más restrictivo; ese límite interno no es un argumento de `perfilar()`. El recorte declara valores, pares y trabajo sin comparar. `Inf` lo desactiva.

`max_trabajo_dependencias`

Tope predeterminado de unidades fila-par para las dependencias. Se combina con `max_comparaciones_dependencias`; el límite efectivo baja cuando hay muchas filas. `Inf` desactiva este tope, pero no el de pares.

`max_largo_valor_vocabulario`
: Maximo de bytes o caracteres permitido en cada valor que entra en los diagnosticos textuales costosos. Por defecto es `10000`, elegido porque las expresiones regulares, la normalizacion y la comparacion de textos largos dejan de ser una medida razonable. Los valores que superan el tope quedan fuera de esos diagnosticos, se cuentan completos para las medidas de la columna y se declaran en `cobertura_diagnosticos`. `Inf` quita el tope. El limite especifico de casi duplicados conserva este mismo valor para no ejecutar una distancia sobre cadenas sin presupuesto. La inferencia de tipo, los conteos baratos, las dependencias y los estadisticos se calculan sobre el universo completo que resulte aplicable. Cada diagnostico que omita los valores largos agrega su propia fila de cobertura, con `dependencia = "texto_largo_no_evaluado"`.

`max_celdas_muestra`
: Maximo de celdas que puede contener la muestra comun de los diagnosticos que muestrean filas. Por defecto es `1000000`; se calcula como filas efectivas por columnas de la tabla. Si reduce la muestra, `cobertura_diagnosticos` informa las filas y celdas solicitadas, el umbral y el nuevo alcance. `Inf` desactiva este tope.

`max_bytes_muestra`
: Maximo de bytes de la muestra materializada que alimenta los diagnosticos que muestrean filas. Por defecto es `512 MiB`. El tamaño se estima sobre una sonda de hasta cien filas y se comprueba sobre la muestra efectiva; si la cota reduce el alcance, `cobertura_diagnosticos` informa los bytes observados y el umbral. `Inf` desactiva este tope. Se mide la tabla como llega, con sus nombres de fila: son parte de lo que se materializa —y el paquete los trata como datos, por ejemplo al proteger—, así que la misma tabla con nombres de fila de texto ocupa más y, con el mismo tope, se muestrea en menos filas.

`avisar_costo_tabla_ancha`
: Si es `TRUE`, avisa en sesiones interactivas cuando las celdas proyectadas superan `umbral_celdas_aviso_tabla_ancha`. El aviso es una estimacion y queda en silencio en scripts no interactivos.

`umbral_celdas_aviso_tabla_ancha`
: Cantidad de celdas a partir de la cual se emite el aviso de tabla ancha. Por defecto, `100000`; `Inf` lo desactiva explicitamente.

Details

Los umbrales de faltantes se aplican a la suma de ausentes reales y faltantes disfrazados y son estrictos: la proporción debe superar el umbral para generar el nivel correspondiente. La lista de cadenas está congelada con referencia a naniar::common_na_strings 1.1.0 y suma extensiones habituales en datos administrativos uruguayos. Las entradas que naniar expresa como patrones escapados se adaptan a los signos literales de interrogación, asterisco y punto porque aquí se comparan por igualdad. La lista no depende de la versión instalada. Los sentinelas numéricos predeterminados son `-9`, `-99`, `-999`, `-9999` y `999`. La lista es deliberadamente más corta que naniar::common_na_numbers 1.1.0: `66`, `77`, `88` y `9999` también pueden ser edades, códigos o años legítimos. `sentinelas_numericos` representa la política completa, no una lista que se agrega silenciosamente: use `numeric()` para desactivar todos los sentinelas numéricos, o `sentinelas_naniar` para solicitar explícitamente la lista de naniar. La política se normaliza como un conjunto numérico: el orden, el tipo entero o doble, los duplicados y los `NA` no cambian su interpretación.

`muestra` limita el descubrimiento de patrones, la inferencia de tipos, la detección de formatos de fecha y la búsqueda de dependencias funcionales. Las demás métricas y hallazgos se calculan sobre todas las filas. En los resúmenes que convierten texto con el tipo o formato descubierto en esa muestra, los valores presentes que no se pueden convertir no se ocultan: `n_fechas_excluidas_granularidad` los cuenta para fechas y `n_valores_excluidos_resumen` para números, y el estado deja de ser un cálculo completo. `n_valores_excluidos_resumen` cuenta también los valores **no finitos** que el resumen deja afuera, y lo hace en cualquier clase: una columna de fechas con un `Inf` publica `n_valores_excluidos_resumen = 1`, `n_fechas_resumidas` sin contarlo y el estado `"calculados_sobre_valores"`. Y para las longitudes de texto, `n_longitudes_resumidas` declara sobre cuántos valores se calcularon `longitud_minima`, `longitud_maxima` y `longitud_media`: `nchar()` no puede medir un valor cuyos bytes no son UTF-8 válido, y ése queda fuera del promedio. La causa viaja aparte, en `n_codificacion_invalida`. Por eso `meta$filas_analizadas` describe el máximo usado por los análisis muestreados, no el alcance del perfil completo. En cada fila de `columnas`, `n_filas_analizadas_tipo` y `muestreado_tipo_inferido` declaran el alcance concreto de `proporcion_tipo_inferido`; no debe interpretarse esa proporción como si hubiera usado necesariamente toda la columna.

El tipo se **infiere** de los valores sólo en las columnas de texto: en las demás es el tipo con que R las guarda —lógico, entero, doble, fecha—, que no depende de los valores. Por eso una columna no textual sin ningún valor conserva su tipo de almacenamiento, con `proporcion_tipo_inferido` en `NA` porque ningún valor lo midió, y una columna de texto sin valores queda `"desconocido"`, porque ahí no hay nada de qué inferirlo.

Cuando el tipo inferido es `"fecha"` o `"fecha-hora"`, `estado_tipo_inferido` declara además cómo quedó establecida esa lectura: `"confirmado"` si todo formato con casamientos es inequívoco, y `"candidato"` si el veredicto se apoya en casamientos ambiguos — el caso de una columna donde `proporcion_tipo_inferido` llega a 1 sin un solo valor inequívoco. En los demás tipos la columna queda `NA`. El resumen impreso anota `(candidato)` junto al tipo cuando corresponde; la conversión, el rango temporal y la remediación sólo usan formatos confirmados.

En una columna temporal, `minimo`, `maximo`, `media` y `mediana` quedan en `NA` y su valor viaja en `minimo_fecha`, `maximo_fecha`, `media_fecha` y `mediana_fecha`, que son texto legible. `desvio` es la excepción y conviene saberlo: no es un momento sino una duración, así que se informa como número, y ese número está **en segundos** tanto para `"fecha"` como para `"fecha-hora"`, porque las dos clases se unifican en esa unidad antes de resumirlas. Un desvío de `136610.4` sobre una columna de fechas son 1,6 días.

Ese detalle importa al comparar las dos puertas. `perfilar_dbi()` no clasifica tipos: informa el que declara el motor, y un motor que no preserva `DATE` ni `BOOLEAN` —SQLite guarda ambos como número— hace que esas columnas se midan como números. La misma columna de fechas da entonces `desvio` en días por la puerta DBI y en segundos por ésta, y su `moda` sale como el entero crudo del motor en vez de la fecha formateada. No es una discrepancia de cálculo: cada puerta describe lo que tiene delante, y lo que tiene delante es distinto.

Una columna cuyo año se expresa con dos dígitos se informa con su `tipo_inferido` —`"fecha"` o `"fecha-hora"`— pero deja `minimo_fecha`, `maximo_fecha`, `media_fecha` y `mediana_fecha` en `NA`. No es una omisión: `23` puede ser 1923 o 2023, y elegir el siglo para calcular un rango sería inventarlo. El hallazgo `anio_de_dos_digitos` señala esas columnas, y el rango aparece una vez que el usuario resuelve la ambigüedad. Si sólo algunos valores traen el año con dos dígitos, la columna conserva su rango: esos valores quedan fuera, como cualquier valor con más de una lectura, y lo declaran `n_valores_excluidos_resumen` y `"calculados_sobre_valores"`. Lo mismo vale para un valor que dos formatos confirmados de la misma columna leen distinto: en una columna que tiene `13/06/2020` y `06/30/2020`, el valor `01/12/2020` puede ser el 1 de diciembre o el 12

de enero, y no se decide por la mayoría. Queda fuera del rango, y `planificar_limpieza()` bloquea la conversión de la columna mientras queden valores así. El resumen de fechas usa la misma regla de validez que la detección: un `99991231` que el formato `%Y%m%d` no admite tampoco llega a `maximo_fecha`. Cuando no sobrevive ningún valor utilizable, el estado distingue **dos afirmaciones que no son la misma**: `"sin_valores"` dice que la columna no tenía ningún valor presente **en el universo que el resumen mide** —vacía, todo `NA`, o con la aplicabilidad declarada dejando ese universo sin valores— y `"sin_valores_utilizables"` dice que sí los tenía y ninguno servía para el resumen. La relación que se cumple con la fila es `n_aplicables - n_faltantes`, no `n - n_faltantes`: con aplicabilidad declarada los valores de afuera del universo se declaran en `n_presentes_fuera_de_aplicabilidad` y no cuentan para este estado. Una columna de treinta `Inf` es el segundo caso y no el primero: publica `n_faltantes = 0` y `n_distintos = 1`, así que decir «sin valores» de ella contradiría a su propia fila. La otra puerta del segundo caso son los centinelas que se llevan todos los valores. La presencia se define igual que en `n_faltantes`, que cuenta el `NaN` como ausente: una columna de treinta `NaN` es `"sin_valores"`.

La `media` y el `desvio` son los dobles más cercanos de la media y del desvío exactos (el desvío con `n - 1`); la `mediana` es exacta. La media se acumula por bloques con una suma exacta —cada bloque se parte en sumandos que se suman sin redondeo—, así que no depende de cómo se agrupen las filas, de que los valores se cancelen ni de la precisión con que la plataforma acumula `sum()`; también es la más cercana cuando es subnormal y cuando la columna junta valores de más de 2^900 (8,5e270) con otros diminutos, que se suman aparte para que no se pierdan. El desvío sale de la suma de los cuadrados, exacta o con una cota de error demostrada, y se elige comparando con aritmética exacta contra los puntos medios entre dobles vecinos; si la cota no alcanza para decidir, la columna se recorre otra vez con la suma exacta. Vale con cuadrados que desbordan (1e200) o que caen en los subnormales. La única excepción: en una columna con un valor 2^885 veces menor que el mayor, el cuadrado de ese valor entra con una cota, y el desvío puede quedar a un ulp sólo si el exacto cae a menos de 2^-1700 relativos de un punto medio (un empate, en la práctica). La mediana se calcula con `median()` sobre el vector entero y no como el cuantil 0,5. `mean()` y `sd()` no son la referencia: pueden quedar a varios ulps. Medido sobre `rnorm(2e6)` con `set.seed(1)`: `mean()` a 208 ulps de la media exacta y la `media` en la exacta; sobre `c(-2^51, 1, 2, 2^51), 0.75`; sobre `((1:25) %% 7 - 3) * 0.3`, `sd()` y el desvío en doble a un ulp del exacto. Quien rehaga `mean()` o `sd()` sobre una columna y vea otro último dígito está viendo esto y no un error.

Value

Objeto S3 de clase `perfil`. Cada fila de hallazgos incluye n_evaluados, n_afectados y unidad_conteo: son conteos de las unidades declaradas (por ejemplo fila, columna, formato o par). En `mayusculas_inconsistentes`, `normalizacion_unicode` y `casi_duplicados_vocabulario`, la unidad es `valor_distinto`: `n_evaluados` cuenta los valores distintos evaluados y `n_afectados` los valores distintos que participan en la colisión. Su `trazabilidad` sigue siendo por fila y enumera todas las filas que contienen esos valores, no sólo las filas defectuosas; por eso su total puede ser mayor que `n_afectados`. En `casi_duplicados_vocabulario`, la traza incluye todas las filas cuyos valores pertenecen al grupo elegido, incluida la forma dominante. La distancia es una senal heuristica, no una prueba de que cada fila deba corregirse; la evidencia declara cuantas filas mostradas pertenecen a las formas variantes y cuantas a las formas dominantes. En `filas_duplicadas`, el conteo y la traza incluyen todas las filas participantes de los grupos; el numero de excedentes queda en la evidencia. `columnas_duplicadas` publica un hallazgo por grupo de columnas con el mismo contenido, no uno por par: `columna` es la primera del grupo en el orden de los datos, la evidencia nombra a todas (`A = B = C`) y `n_afectados` las cuenta a todas. `general$columnas_duplicadas` tiene una fila por

cada columna repetida, con la primera de su grupo en columna_1 y la repetida en columna_2. Dos columnas son iguales si lo son en lo que el perfil lee como contenido: un factor con nivel NA y el mismo sin ese nivel, un POSIXlt y su POSIXct, o dos factores sin orden con los mismos niveles en otro orden son la misma columna; un entero y un doble, que el perfil declara de tipos distintos, no. Si algún nombre del grupo no se puede leer suelto —vacío, repetido en la tabla, con espacios al borde o con =, ;, , o comillas—, la evidencia cita todos los nombres entre comillas, y el que se repite lleva su posición: "X" (columna 1) = "X" (columna 2). Sin filas no hay hallazgo ni filas en general: la cobertura declara que no se compararon. Cuando el camino no puede conocer un conteo, informa NA, nunca cero. La columna de lista trazabilidad distingue disponible, truncada, no_aplica y no_disponible; cuando corresponde conserva índices de fila acotados por max_filas_hallazgo, el total conocido y el alcance. En clave_con_ausentes, cuenta las filas con al menos un componente ausente y su traza enumera esas filas. En clave_no_unica, cuenta las filas que participan en colisiones entre claves completas y su traza excluye las filas incompletas; ambas unidades son fila. Así, una colisión entre ausentes no aparece como una repetición que refute la unicidad. casi_duplicados_vocabulario, donde la traza mezcla filas de formas variantes con filas de la forma dominante, conserva además n_filas_formas_variantes y n_filas_formas_dominantes con el reparto completo, y mostrados_formas_variantes y mostrados_formas_dominantes con el reparto de lo que sobrevivió al truncado. En los tres diagnósticos de unidad valor_distinto las filas de las formas variantes se entregan primero —en mayusculas_inconsistentes y normalizacion_unicode, la forma dominante de cada grupo es la guardada más veces—, de modo que el truncado no se lleve lo accionable. Para patron_raro, el alcance puede ser completo, muestra_patrones, patrones_parciales o muestra_patrones+patrones_parciales. El resumen y el texto de evidencia de patron_raro muestran como maximo seis patrones, pero la trazabilidad conserva los nombres de todos los patrones raros hasta un limite de 5.000; patrones_parciales indica que se alcanzo ese limite, no que se haya alcanzado el tope de presentacion. Cuando se emite un hallazgo patron_raro, su evidencia incluye la proporcion del patron dominante y cuantas filas pertenecen a patrones no dominantes que superan umbral_patron_raro y por eso quedan excluidos. Su n_evaluados es el universo de esas proporciones: las filas analizadas que tienen valor —un ausente no tiene patrón—. Si el patron dominante no alcanza umbral_patron_dominante, no se emite el hallazgo: cobertura_diagnosticos declara la no medicion, su proporcion observada y el argumento que se puede ajustar. Si el conteo y la traza no coinciden, conserva el hallazgo y emite una advertencia de clase lupa_trazabilidad_incoherente. La guarda compara el total previo al truncado y respeta la unidad declarada. Una columna compuesta no analizada conserva en la traza todas sus filas. Si una columna de listas se reconoce como constante pero no se puede contar su frecuencia, el conteo afectado queda en NA y cobertura_diagnosticos explica la no evaluación. Los índices no contienen valores. Usarlos para extraer filas de los datos originales puede volver a exponer datos personales; el paquete no realiza esa extracción y la protección de salidas no sustituye el control de acceso a los datos de entrada. En la evidencia de casi_duplicados_vocabulario, clase_diferencia puede ser normalizacion_exacta (la coincidencia aparece después de normalizar), dentro_de_palabra (la diferencia está dentro de un token), token_completo (cambian tokens completos), token_unico (ambos valores son un solo token y esa distinción estructural no aplica), mixta (el grupo reúne aristas de más de una clase) o indeterminada (no hubo aristas clasificables). Son categorías de evidencia, no veredictos de identidad. cobertura_diagnosticos es una tabla hermana de hallazgos, con una fila por diagnóstico que no pudo evaluarse o cuya enumeración quedó parcial y las columnas diagnostico, columna, motivo, como_resolverlo y dependencia. Incluye la falta de stringdist, stringi o sf y las zonas horarias POSIXt sin declarar. Sin bit64 una columna integer64 no deja una fila acá: el perfil se detiene con lupa_error_integer64. Los patrones de frecuencia intermedia

no se consideran desvios del patron dominante: `patron_raro` es completo respecto de su criterio de rareza cuando no hay recorte de trazabilidad. Si el conjunto de nombres raros supera 5.000, `cobertura_diagnosticos` declara el recorte y su limite. Si `muestra` deja filas fuera del descubrimiento de patrones y la muestra no trae ningún patrón raro, `cobertura_diagnosticos` también lo declara: la ausencia del hallazgo vale sólo para la muestra. Quien decida automáticamente sobre un perfil debe revisar `nrow(perfil$cobertura_diagnosticos)` además de las severidades: un perfil sin hallazgos y con diagnósticos no evaluados no es un perfil limpio. Cuando una clave declarada no queda plenamente verificada, `meta$clave` conserva los estados de unicidad y ausencia de nulos, sus conteos y la semántica usada por la trazabilidad. Los tres responden preguntas distintas y no comparten universo:

- `unicidad` se evalúa **sólo entre las filas con la clave completa** (`semantica = "claves_completas"`), porque una repetición entre filas incompletas no viola la unicidad: en SQL dos `NULL` no son iguales. `filas_evaluadas` cuenta esas filas y `filas_totales` la tabla entera. Su estado es `"verificada"`, `"refutada"`, `"no_verificada"` cuando no se pudo comparar, o `"sin_casos_evaluables"` cuando **ninguna** fila tiene la clave completa: ahí la unicidad sería cierta sobre un conjunto vacío, que es cierto y engañoso a la vez, y por eso tiene estado propio.
- `ausencia_nulos` responde si todos los componentes están presentes. Su hallazgo asociado, `clave_con_ausentes`, cuenta las filas con al menos un componente ausente y conserva sus índices.
- `clave_no_unica` sólo se emite cuando hay valores repetidos entre las filas completas y usa `filas_evaluadas` como `n_evaluados`; no convierte una colisión entre ausentes en una violación de unicidad.
- `trazabilidad` conserva la semántica de R, que es la que localiza las filas, e informa en `colisiona_con_ausentes` si el localizador queda ambiguo porque dos filas con ausentes comparten representación.

La unidad viaja al lado de la cifra

El campo `unidad` dice **en qué unidad están las cifras que la fila publica** —`minimo`, `maximo`, `media`, `mediana`, `desvio`—, y queda en `NA` cuando la columna no declara ninguna. Es una sola pregunta, y por eso no se confunde con `numero_texto_unidad`, que es la unidad que un valor de texto traía pegada al número (`"12 kg"`), ni con la `unidad` de `clasificar_variables()`, que es la unidad **declarada** del dato.

Sin este campo, columnas medidas en unidades distintas publicarían filas idénticas: `units::set_units(c(1, 2, 3), "m")` y la misma en `"km"` dan las dos `media = 2` y `desvio = 1`, y un `Period` de dos días publicaría `172800` sin decir que son segundos. Por clase:

- `units` y `difftime`: la unidad declarada, en forma canónica (`km/h`, `m^2`, `kg*m/s^2`). `difftime` **publica sus estadísticos en esa unidad**: una columna de minutos publica `media = 75` con `unidad = "mins"`: sin la unidad al lado, un `75` no sería una medida. Los valores van en la unidad declarada por la columna, sin convertir a segundos: convertirlos publicaría un número que no está en la columna.
- `Period`, `Duration`, `Date` y `POSIXt`: `"segundos"`, que es la unidad en la que sale `desvio`.
- cualquier otra columna que **declare** un atributo `units` —eso circula como metadato en datos importados— publica lo declarado, porque sus cifras están en esa unidad.
- el resto: `NA`. El campo no inventa una unidad donde no hay ninguna.

Y se calla cuando la fila no publica ninguna cifra cuantitativa, o sea cuando `estado_resumen_cuantitativo` vale `"no_aplica"`: el atributo `units` lo puede llevar cualquier clase, y un `factor` con `attr(x, "units") <- "kg"` publicaría `unidad = "kg"` al lado de `media = NA`. Una unidad de cifras que no existen no contesta la pregunta del campo. El discriminador es ese estado y no una lista de clases, así que una columna `units` de puros `NA` **sí** declara su unidad: su clase se resume, y la unidad dice en qué estarían sus cifras.

Si una columna declara **más de una** unidad —un atributo `units` de largo mayor que uno, que ningún objeto `units` produce pero un dato importado sí—, el campo las publica todas separadas por coma. Ese valor no es una unidad: dice que la declaración está mal. Quedarse con la primera sería elegir una por el usuario, justo lo que este campo evita.

Y `comparar_perfiles()` compara este campo como un aspecto propio, con la severidad de un cambio de tipo: publicar la unidad sin compararla no alcanzaría, porque lo que importa es que **una entrega donde cambió la unidad muestre el cambio**.

Una columna de períodos expresados sólo como mes y año informa la `granularidad` `"mes"` en `formatos_fecha` y deja los resúmenes de fecha en `NA` con `estado_resumen_cuantitativo = "granularidad_incompleta"`: asignar el día 1 para obtener un mínimo o una media también sería inventar un dato. Si esos períodos son minoritarios dentro de una columna que también contiene fechas con día, los resúmenes se calculan sólo sobre las fechas completas y declaran `estado_resumen_cuantitativo = "calculados_sobre_dias"`, junto con `n_fechas_resumidas` y `n_fechas_excluidas_granularidad`. El mínimo y el máximo son entonces condicionales al subconjunto con día; no representan un rango de toda la columna.

Los dos conteos que acompañan a ese estado cuentan cosas distintas. `n_fechas_excluidas_granularidad` cuenta **sólo** las fechas que quedaron fuera por ser de granularidad mes, y queda en `NA` cuando los formatos se descubrieron sobre una muestra: ese número se deriva de los formatos detectados en la muestra mientras el resumen recorre la columna entera, así que atribuir la causa exigiría rehacer la detección sobre todo, que es el costo que el muestreo existe para evitar. `n_valores_excluidos_resumen` es el campo general —el mismo que usan las columnas numéricas— y cuenta **todo** valor presente que no sostiene el resumen, sea un período de mes o un texto que ningún formato pudo leer. Siempre que ese total sea mayor que cero, el perfil agrega una fila `resumen_cuantitativo` en `cobertura_diagnosticos`, haya muestreo o no.

Para números ordinarios, los estadísticos cuantitativos se calculan sólo con valores finitos; `n_nan`, `n_infinito_positivo` y `n_infinito_negativo` declaran lo excluido. En columnas `integer64` que exceden el entero máximo representable exactamente por `double`, `minimo` y `maximo` quedan en `NA` y los extremos exactos se conservan en `minimo_exacto` y `maximo_exacto`. La `mediana` depende sólo de sus una o dos centrales: se publica si cada una cabe exacta en un doble —valor absoluto menor que 2^53— y queda en `NA` si no. Lo mismo vale para una columna de texto con esos enteros, salvo con `sentinelas_numericos` declarados, que ese camino no excluye. Cuando la conversión de texto deja valores presentes afuera —haya muestreo o no, porque la conversión descarta lo que no puede leer en los dos casos—, `n_valores_excluidos_resumen` y `estado_resumen_cuantitativo = "calculados_sobre_valores"` declaran ese alcance parcial.

Los limites de Tukey se publican solo cuando hay al menos 20 valores finitos y el recorrido intercuartilico es positivo. Si falta alguna condicion, `n_outliers` queda en `NA` y `cobertura_diagnosticos` explica el motivo y como reunir evidencia suficiente; un IQR cero no convierte una categoria minoritaria en outlier.

El paquete distingue lo que el usuario **declara** de lo que él mismo **sospecha**, y esa distinción gobierna los resúmenes. Cuando se reemplaza `sentinelas_numericos`, esos valores son una

declaración de ausencia y salen de `media`, `mediana`, `minimo`, `maximo`, `desvio` y `n_outliers`, igual que salen los `NA` y las filas que `aplicabilidad` deja fuera del universo; `n_valores_excluidos_resumen` y `estado_resumen_cuantitativo` declaran ese alcance. Salen también de la ley de Benford y de las relaciones aritméticas, que leen la columna como magnitud, y las dependencias funcionales y las asociaciones no evalúan la fila que los tiene: la misma celda no puede ser ausencia en el resumen y valor en un diagnóstico. La lista por omisión —`-9`, `-99`, `-999`, `-9999` y `999`— es en cambio una conjetura del paquete: se informa como `faltantes_disfrazados` pero **no** se retira de los resúmenes, porque `999` también puede ser un dato legítimo y actuar sobre una sospecha cambiaría números que nadie pidió cambiar.

`moda`, `frecuencia_moda`, `n_distintos`, `tasa_distintos` y `longitud_*` describen la **representación almacenada** y siguen contando esos valores, del mismo modo que cuentan un `Inf` que los estadísticos excluyen o un `"N/A"` textual. Por eso `moda` puede quedar fuera del rango que informan `minimo` y `maximo`: no es una contradicción sino la diferencia entre describir lo guardado y resumir lo que vale como dato, y los conteos —`n_valores_excluidos_resumen`, `n_infinito_positivo`, `n_faltantes_disfrazados`— dicen cuánto separa a los dos. Una columna de listas intenta contar sus valores distintos; si la clase no admite comparación, informa `NA` en lugar de afirmar cero. Las columnas compuestas —matrices, arreglos de más de una dimensión o tablas anidadas (una columna `data.frame`, como la de un tibble empaquetado)— se conservan como una unidad por fila: `n` informa las filas de la tabla, pero los estadísticos por valor —`n_faltantes` incluido— quedan en `NA` y un hallazgo explica que deben separarse en columnas con semántica explícita. Entra por la misma puerta una columna cuyos valores no se pueden escribir como texto —una `vctrs_rcrd`, que guarda varios campos por fila, o un `vctrs_vctr` sin conversión declarada—: se pregunta por esa propiedad y no por la clase, y el hallazgo `tipo_compuesto_no_analizado` lo dice con su clase, en vez de abortar el perfil entero. Un vector de bytes (`raw`) se compara y se publica por su escritura hexadecimal —`moda`, `n_distintos`—, pero no pasa por las etapas que interpretan la escritura de una persona —patrones, números escritos como texto, unidades, espacios—: un byte no es un texto escrito. Un nivel `NA` de un factor —el que dejan `addNA()` o `factor(exclude = NULL)`— es un faltante: cuenta en `n_faltantes` y no en `n_distintos`, igual que en la medición, en `distribucion_valores()`, en `inferir_tipo()`, en las dependencias y en las claves —`detectar_claves()` y `sugerir_clave()` no la proponen—. Un `POSIXlt` —lo que deja `strptime()`— se perfila por su instante, como el `POSIXct` equivalente: la misma columna en las dos clases da la misma fila. Cuando todos los valores válidos aparecen una sola vez **no hay moda**, y `moda` queda en `NA`. Lo que se publicaría es el ganador de un desempate de tantas vías como valores haya, y ese desempate sigue el orden de ordenamiento, que depende de cómo esté guardada la columna: la misma columna `c(-5.5, -1, 0, 3.75)` daría `-5.5` como número y `-1` como texto. `frecuencia_moda` se conserva —vale 1, es cierto y es la evidencia de por qué la moda quedó callada—. Con un solo valor distinto sí hay moda, aunque su frecuencia sea 1: ahí no hay empate que resolver.

Una columna numérica emite `valor_concentrado` como señal `sospechoso` cuando tiene al menos 20 valores válidos y 10 valores distintos, y su moda tiene una frecuencia al menos cinco veces mayor que la del segundo valor más frecuente y representa al menos 0,15 de los valores válidos. La elegibilidad es parte de la señal: una columna con menos categorías no recibe una fila de `cobertura_diagnosticos`, porque allí la moda es la distribución. La evidencia publica el valor modal, ambas frecuencias, el cociente y la fracción. La regla se calibró sobre 114 columnas limpias: no produjo ningún falso positivo, con un margen medido de 2,2 veces. Es una sospecha, no una acusación: un valor legítimo puede dominar. Sus puntos ciegos medidos son las concentraciones por debajo de 15 % y los empates naturales en columnas enteras pequeñas, donde el cociente puede quedar por debajo de cinco.

La ley de Benford se evalúa sólo en columnas numéricas con al menos 50 valores finitos; las colum-

nas compuestas —matrices, arreglos de más de una dimensión o tablas anidadas— no son magnitudes por fila y quedan fuera del análisis. Antes de comparar exige variación, que la columna no parezca un identificador ni una secuencia correlativa, al menos 100 observaciones positivas utilizables, una proporción de positivos igual a 1 y tres órdenes de magnitud según `log10(max/min)`. Una secuencia correlativa es una columna de enteros que cubre al menos la mitad del tramo entre su mínimo y su máximo **sin repetirse** —al menos un 90 % de valores distintos—: una magnitud entera acotada también llena su tramo cuando se cargan filas suficientes, pero repitiendo valores, y la exclusión no puede depender de cuántas filas se cargaron. Es la regla de numeración de todo el perfil (ver `secuencia_entera_densa` más abajo), y una clave declarada excluye a Benford siempre, como una columna declarada en `columnas_codigo`. Si falla alguna precondición no emite un hallazgo: la enumera en `cobertura_diagnosticos`. Si aplica, `meta$benford$resultados` conserva la distribución observada y esperada por primer dígito, el chi-cuadrado de Pearson, ocho grados de libertad y el valor p; si entre los positivos hay valores que el perfil cuenta como faltantes disfrazados sin que estén declarados, `sin_faltantes_disfrazados` publica la misma prueba sin ellos, y el hallazgo dice si la desviación se mantiene o la producen ellos. `meta$benford$umbrales` publica todos los cortes. Un valor p menor que `0.01` genera `desviacion_benford` como señal descriptiva para revisar, no como evidencia de fraude o manipulación. Topes administrativos, redondeos, precios psicológicos y subsidios de monto fijo son explicaciones posibles.

Las relaciones aritméticas se buscan sólo entre columnas numéricas **sin clase declarada** y con variación: una columna que declara una clase propia —`Date`, `POSIXt`, `difftime`, `integer64`, `units`, `lubridate::Period` o cualquier otra— no participa, porque sumarla o dividirla no es la aritmética de un doble; el texto numérico y las columnas constantes tampoco. Las Las columnas que **publican cifras cuantitativas** y quedan fuera por su clase se declaran una por una en `cobertura_diagnosticos`: `integer64`, `units`, `lubridate::Period`, `haven_labelled` y `difftime`. El criterio **no** es `is.numeric()` —que responde `FALSE` sobre un `difftime`— sino si la fila publica `minimo`, `maximo` y `media` como números: ahí la ausencia de hallazgo se lee como conformidad y por eso hay que declararla. `difftime` está en esa categoría porque publica su resumen cuantitativo: sin declararlo quedaría mudo en los dos diagnósticos mientras `units` los declara —la misma columna, con las mismas cifras, tratada de dos maneras—.

Una **fecha** y una **hora** no entran en esa declaración, y eso es deliberado: no publican `media` como número pelado —su resumen sale en `media_fecha` y compañía—, así que su silencio se lee distinto, y no son magnitudes para estos diagnósticos. Quedan fuera del mismo modo que una columna de texto, sin fila propia. `meta$aritmetica_columnas` publica las clases medidas en esta tabla. La ley de Benford sigue el mismo criterio y la misma declaración. Cada relación requiere al menos tres filas con valores finitos en todas las columnas involucradas; los `NA`, `NaN` e infinitos quedan fuera del universo que publica la evidencia. Una columna `integer` se suma en doble: que dos enteros pasen `.Machine$integer.max` no apaga la relación. Para cada terna se prueban las tres orientaciones de una identidad aditiva; esto cubre sumas y sus restas equivalentes sin informar tres veces la misma igualdad. En pares proporcionales, `k` es la mediana de los cocientes finitos cuya base no es cero —salvo cuando la resolución de una columna con decimales fijos hace de tolerancia (ver `tolerancia_aritmetica`): ahí es la constante con menos cifras que deja casi tantas filas dentro de esa resolución como la que más—, pero el cumplimiento se evalúa después también en las filas con base cero. Si una identidad aditiva ya relaciona una terna, se omiten las proporcionalidades redundantes entre su total y sus sumandos; se conserva la proporcionalidad entre los dos sumandos. Una regularidad completa se informa con severidad `"ok"`; si alcanza el umbral pero tiene discrepancias, sigue el criterio de las relaciones de orden y es `"sospechoso"`. Todo esto describe evidencia observada: no declara una regla del dominio ni autoriza una corrección. Los valores de texto que no forman UTF-8 válido tampoco se convierten: se cuentan, se excluyen de los análisis

textuales y generan un hallazgo con sus posiciones. Los diagnósticos de invisibles incluyen controles C0/C1, espacios Unicode, marcas direccionales, BOM y otros caracteres de transporte. La evidencia los muestra como puntos de código; los espacios Unicode se detectan aunque sólo se normalizan mediante una acción explícita, y ZWJ/ZWNJ se informan pero se conservan porque pueden ser semánticos. La comparación de duplicados con `normalizar = TRUE` aplica estas mismas clases sin borrar ZWJ/ZWNJ. El resto del texto se cita con sus caracteres, y no con los escapes de punto de código de `encodeString()`, que dependen del locale: la evidencia es la misma bajo `C` y bajo un locale UTF-8, y el ejemplo que cita casa con el de la tabla de patrones.

Un valor `double` **subnormal** —distinto de cero y menor que `.Machine$double.xmin` en valor absoluto, unos 2,2e-308— casi siempre sale de reinterpretar un patrón de bits o de un desbordamiento por defecto. El hallazgo `valores_subnormales` los cuenta y da sus filas, con severidad `sospechoso`. El caso que motiva el diagnóstico es leer una tabla donde una columna de enteros grandes se escribió como doble reinterpretando los bits: los números salen del orden de 1e-314 y, al venir así desde el origen, ninguna comprobación cruzada los contradice.

No es concluyente, y el hallazgo no lo afirma. Un cálculo que desborda por defecto cae legítimamente en ese rango —`2^-1050` da 8,3e-317—, y una columna de p-valores o de verosimilitudes muy pequeñas puede dispararlo sin que haya nada roto. Lo que se publica es el hecho medido —cuántos valores son subnormales y en qué filas—, no un veredicto sobre su origen.

Los resúmenes de fecha-hora se expresan siempre en UTC y llevan el sufijo `UTC` en el texto para hacer visible la zona aplicada. El instante se conserva aunque la columna de entrada use otra zona horaria. Las columnas `POSIXt` declaran `zona_horaria_origen` y `n_filas_fecha_civil_distinta_utc`, que cuenta filas cuya fecha civil cambia al mostrar el instante en UTC. Cuando la fecha civil cambia se emite un hallazgo `zona_horaria_fecha_hora`; si la zona de origen no está declarada, el conteo queda en `NA` y `cobertura_diagnosticos` declara que no se evaluó. Una columna de texto cuyos valores llevan el desplazamiento horario escrito (`2020-01-31T23:30:00-03:00`) también publica su instante en UTC, y su fecha civil puede no ser la escrita: el conteo es de las columnas `POSIXt` y queda en `NA`, y `cobertura_diagnosticos` declara igual que no se evaluó. La zona se declara en la columna original, por ejemplo: `attr(x, "tzone") <- "America/Montevideo"`.

El diagnóstico de formas Unicode compara sin modificar el texto y puede usar el paquete opcional `stringi` para enriquecer la evidencia cuando existen caracteres no ASCII. `columnas$unicode_evaluado` declara si esa comprobación pudo ejecutarse; en texto no ASCII sin `stringi` queda `FALSE`, `n_variantes_unicode` queda en `NA` y `cobertura_diagnosticos` informa la dependencia ausente. Las columnas ASCII se evalúan siempre y conservan cero. El perfil de comparación es completamente R base.

Hay un segundo caso en que `unicode_evaluado` queda en `FALSE`, y no es una dependencia ausente: cuando **todos** los valores de la columna traen bytes que no forman UTF-8 válido, no queda texto que analizar. Ahí las pasadas que dependen de texto legible no llegan a correr, así que `n_codificacion_rota`, `n_codificacion_reparable`, `n_codificacion_reparable_parcialmente`, `n_codificacion_irreparable`, `n_codificacion_no_se_pudo` y `estado_codificacion_reparacion` quedan en `NA` en vez de en cero: un cero afirmaría no haber encontrado nada, y lo cierto es que no hubo nada que mirar. `n_codificacion_invalida` sigue contando esos valores y el hallazgo `codificacion_invalida`, de severidad `error`, sigue declarando que se excluyeron.

Esas cifras **no son una partición**. `n_codificacion_rota` cuenta cada valor afectado una sola vez; `n_codificacion_reparable` y `n_codificacion_reparable_parcialmente`, los que el motor reparó del todo o en parte; `n_codificacion_irreparable`, los que traen el carácter de reemplazo `U+FFFD`, cuyo original ya no está; y `n_codificacion_no_se_pudo`, los que el motor reconoce como mal convertidos y no pudo reparar. Un valor con `U+FFFD` suele contar en las dos últimas, así que sumarlas no da `n_codificacion_rota`.

Los **nombres de columna** que el perfil publica —en `columnas$columna` y en todo lo que nombre una columna: claves, relaciones, dependencias, hallazgos, patrones, el plan de remediación y el reporte— son los nombres de la tabla de entrada tal como llegaron, con sus bytes **y su marca de codificación**. Eso significa que sirven para indexar esa misma tabla —`datos[[nombre]]`— sea cual sea el `LC_CTYPE` de la sesión. El paquete deriva una representación de trabajo para comparar y desambiguar internamente, pero no la publica: dos nombres con bytes distintos siguen siendo dos columnas distintas. Al componer texto publicado —en particular las sugerencias de `posible_ausencia_estructural`— usa una copia marcada como UTF-8 del nombre, sin modificar el nombre conservado en la tabla. Por eso una sugerencia como `perfilar(datos, aplicabilidad = list(entero = ~ categoría == "categoría"))` conserva el nombre real también bajo `LC_CTYPE = "C"` y se puede copiar y pegar.

La identidad de la columna no depende de esa exclusión: `n_distintos`, `moda` y `frecuencia_moda` comparan la representación almacenada, que se puede distinguir sin decodificarla. Un valor ilegible cuenta como valor, y la moda que se publica es el valor original tal como llegó, sin reinterpretarlo. Cuando ese valor es numérico, su representación textual se formatea con notación fija, con la precisión completa —las cifras que hacen falta para volver al valor: `0.1 + 0.2` se publica `0.30000000000000004`, no `0.3`— e independiente de `scipen` y de `digits`, igual en `perfilar()` y en los caminos de `perfilar_dbi()`; se conserva `OutDec`, porque esa marca decimal sí pertenece a la preferencia de locale de quien usa el paquete.

Las columnas `sfc` declaran su CRS, los tipos concretos y la dimensión (`XY`, `XYZ`, `XYM` o `XYZM`), además de geometrías vacías, validez, dominio y caja envolvente. `POINT`/`MULTIPOINT`, `LINESTRING`/`MULTILINESTRING` y `POLYGON`/`MULTIPOLYGON` son una misma familia: el hallazgo de tipos mixtos aparece sólo al combinar familias o ante `GEOMETRYCOLLECTION`. La validez se calcula siempre con GEOS en el plano, sin CRS, para que el resultado no dependa de que `s2` esté instalado; `validez_criterio` publica `"planar"` y `n_validez_evaluados` publica su universo, que incluye las geometrías vacías porque GEOS sí devuelve su validez. Si hay una dimensión M, se aplica `sf::st_zm()` y se valida la topología XY; `validez_preprocesamiento = "st_zm(x)"` declara esa transformación. Si el CRS es geográfico —con SRID mezclados, si todos lo son—, un fallo es `sospechoso` y no un `error`, porque no afirma invalidez esférica. Las dimensiones Z y M no se evalúan como medidas: `dimensiones_no_evaluadas` las enumera y `cobertura_diagnosticos` deja constancia explícita incluso cuando la validez XY sí pudo calcularse. En un CRS geográfico el dominio exige longitudes en `[-180, 180]` y latitudes en `[-90, 90]`. Además, las coordenadas se comparan en longitud/latitud con la `BBOX` del área de uso incluida en el WKT del CRS. Este control puede detectar valores en unidades incompatibles —por ejemplo, grados declarados como metros—, pero no una zona UTM equivocada cuando esa interpretación cae dentro del área de la zona declarada. Una `BBOX` mundial es un no-op válido; si el WKT no permite extraer la caja, el dominio queda en `NA` y `cobertura_diagnosticos` lo declara en lugar de suponer el mundo entero. Si el CRS geográfico mide desde otro meridiano de origen —París, Ferro— o en otra unidad —gradianes—, ese rango se exige en sus propias unidades y la comparación con la `BBOX` se hace después de llevar las coordenadas a longitud/latitud WGS 84. Un componente vacío de una geometría que no es vacía —un `POINT EMPTY` dentro de una colección o de un `MULTIPOINT`— no es una coordenada y no cuenta como fuera de dominio. Las geometrías vacías se cuentan aparte y no integran el universo del dominio ni de la bbox. `n_dominio_evaluados` y `n_bbox_evaluados` publican ambos universos; la bbox se calcula sobre coordenadas crudas de geometrías no vacías, incluidas las que el dominio marque fuera, y `bbox_alcance` lo declara. Si todas son vacías, sus conteos evaluados y fuera de dominio son cero. `geometria_vacia` y `tipos_geometria_mixtos` cuentan sobre las geometrías medidas: un valor ausente de una columna de texto o un valor que no se pudo convertir no es una geometría y no entra en su denominador. Sin CRS, `n_fuera_de_dominio` queda en `NA` y se emite

crs_no_declarado: nunca se supone EPSG:4326. Si falta el paquete opcional sf, todos esos campos quedan en NA, no se emite un hallazgo geométrico y cobertura_diagnosticos registra la dependencia ausente. Si sf está instalado pero no se pudo cargar en la sesión —con LC_NUMERIC en coma no carga su dependencia units—, pasa lo mismo y el motivo lo dice: no pide instalar lo que ya está.

Una columna cuyos valores son data.frame o matriz —lo que devuelve un driver al leer un STRUCT— declara n en **filas**, como cualquier otra: tiene tantos valores como la tabla, y lo que no se puede es analizarlos como texto, que se declara aparte en cobertura_diagnosticos.

Con un universo aplicable declarado, los conteos y los índices de geometría se restringen a ese universo, igual que el resto del perfilado: una geometría que la regla dejó afuera no cuenta como vacía, ni como inválida, ni como fuera de dominio, y n_geometrias, n_dominio_evaluados y n_validez_evaluados describen el universo. Los dos evaluados cuentan sólo lo que de verdad se evaluó dentro de él: no entran las geometrías que el presupuesto no analizó, los valores ausentes de una columna de texto ni, para el dominio, las geometrías vacías. El presupuesto geométrico —los topes lupa.max_geometrias y lupa.max_vertices, que acotan la validez y el dominio— se gasta primero en el universo, con el mismo muestreo uniforme si el universo solo ya excede el tope, y lo que sobra va al resto de la columna. Si aun así quedaron geometrías del universo sin analizar, cobertura_diagnosticos dice cuántas, junto al recorte de la columna; si el recorte cayó entero fuera del universo, también lo dice. Tres campos son la excepción y conviene saber por qué: n_bbox_evaluados, n_geometrias_analizadas y n_vertices_analizados cuentan **el trabajo que se hizo** sobre la columna entera —el universo y lo que el presupuesto alcanzó del resto—, no filas de un universo: el análisis geométrico corre sobre la columna, porque necesita su CRS y sus tipos, así que se publican tal como se midieron. Recortarlos a ojo sería inventar un número: no hay conjunto de índices del que derivarlos.

Una columna de texto puede ser una geometría. Se reconoce como WKT o WKB (crudo o hexadecimal) cuando la mayoría de una muestra de 20 valores tiene esa forma; si después hay valores que no se pueden convertir, la pérdida se declara con su cifra en motivo_representacion y en cobertura_diagnosticos, y las métricas cubren el resto: un valor que no convierte no borra la medición de la columna. A sf sólo llega lo que tiene la forma de su representación —un WKT de los tipos que el perfil mide, con su dimensión Z o M y colecciones anidadas como las que escribe PostGIS, o un WKB de encabezado plausible—; el texto corrupto y las curvas se declaran como no convertidos. Si los valores declaran su SRID —SRID=n; en EWKT, o EWKB—, la columna se mide por SRID y no se acusa crs_no_declarado, aunque ningún valor se haya podido convertir. Esa columna se sigue perfilando como texto en lo que toca a su escritura —espacios, codificación, invisibles—, pero no recibe los diagnósticos que leen los valores como palabras o como claves —la proximidad de vocabulario, posible_identificador, patron_raro—: cobertura_diagnosticos declara que no aplican. El argumento normalizar declara el perfil de comparación que se conserva en meta$normalizacion; cambia sólo la representación usada para comparar, no el texto guardado. TRUE usa el perfil predeterminado, FALSE desactiva sus pasos configurables, "amplio" activa los tres pliegues optativos y normalizacion() permite declararlos. También admite una lista nombrada por columna. meta$normalizacion_fusiones informa, para cada paso activo, la diferencia entre los valores distintos con el perfil completo y los valores distintos con ese paso apagado y todo lo demás igual. Esa comparación responde cuánto aporta cada paso por separado; sus números no son aditivos porque dos pasos pueden fundir el mismo par. n_distintos y n_distintos_normalizados declaran el total antes y después del perfil completo. La descomposición canónica es siempre activa y forma parte de la línea base, no una fila configurable. Un valor con marca bytes entra con su clave de bytes —cada byte alto escrito \xHH—, como en detectar_claves() y detectar_duplicados_aproximados(): bytes

declara que eso no se interprete como texto, así que no se pliega con el mismo texto marcado UTF-8. Los tres cuentan igual `n_distintos_normalizados`. Con `normalizar = FALSE` no hay pasos configurables que medir y el informe de fusiones queda en `NULL`; en un perfil por columna sólo se incluyen las columnas con algún paso activo. Si `detectar_duplicados_aproximados()` recibe un perfil, puede reutilizar este informe ya calculado. El informe usa el vocabulario completo: `n_distintos` y `n_usados` son el número de valores distintos realmente comparados y el estado es `exacto`. Las fusiones son una propiedad de pares, por lo que muestrear valores aislados podría dejar fuera los dos miembros de cada par y convertir una fusión real en un cero falso. La normalización se vectoriza para que este alcance completo no dependa de la cardinalidad de la columna. Las columnas sin ausentes que tienen al menos 100 filas y 90 % de valores distintos producen un hallazgo `casi_clave` cuando el valor dominante concentra al menos la mitad de los duplicados excedentes. La concentración se calcula sobre las repeticiones posteriores a la primera de cada valor, no sólo sobre la tasa de distintos. La evidencia declara el mínimo de filas, ambos umbrales, los valores que colisionan y sus frecuencias. Las variables con rol propuesto `fecha`, incluidas fecha-hora, se excluyen. Así una colisión concentrada queda separada del texto libre de alta cardinalidad con repeticiones dispersas. Un vector `double` sólo participa si todos sus valores finitos son enteros; así se conservan identificadores importados con ese almacenamiento y se excluyen medidas con alguna parte fraccionaria, como importes o coordenadas. Los vectores `integer64` se tratan como enteros semánticos. La evidencia del hallazgo declara este criterio y el recuento observado. Además, si `casi_duplicados_vocabulario = TRUE`, el perfil busca variantes casi duplicadas en columnas de texto. Agrupa el vocabulario crudo mediante fusiones exactas de la normalización y estrellas de distancia centradas en un valor de frecuencia estrictamente mayor y único; los empates no se fuerzan. No cierra cadenas transitivamente ni elige una forma canónica. La unidad es el valor distinto, no la fila, y cada variante conserva su frecuencia. Cada grupo declara sus distancias mínima y máxima. El límite `max_proporcion_grupo_vocabulario` evita presentar un grupo que abarque casi toda la columna como un diagnóstico útil: en ese caso el alcance dice que el diagnóstico no aplica. Ese límite se activa desde 20 valores distintos o cuando el grupo mayor ya tiene 10 variantes; sólo suprime el grupo si además ocupa una fracción mayor que el umbral. Así un grupo de tres en cuatro valores se entrega, pero quince variantes que ocupan toda una columna no se presentan como una sola familia. Un grupo grande dentro de un vocabulario mucho mayor puede seguir pasando si su proporción es pequeña. El alcance expone ambos cortes. El argumento permite apagar el detector cuando no corresponde a la tabla. Si hay pares cercanos pero todas las frecuencias empatan, el alcance declara que no hubo asimetría para formar una estrella y sugiere `detectar_duplicados_aproximados()` para comparar filas. El alcance clasifica cada grupo como `normalizacion_exacta`, `dentro_de_palabra`, `token_completo`, `token_unico`, `mixta` o `indeterminada`. La primera indica una coincidencia tras normalizar; `dentro_de_palabra`, una diferencia dentro de un token; `token_completo`, diferencias en tokens completos; `token_unico`, que ambos valores son un único token y la clase estructural no aplica; `mixta`, aristas de más de una clase; e `indeterminada`, que no hubo aristas clasificables. Son evidencia descriptiva, no una decisión sobre identidad. El agrupamiento no cambia por esa etiqueta. El alcance también descarta aristas de distancia cuyos números no coinciden. Se comparan las secuencias numéricas, quitando ceros de relleno y separadores de miles; una diferencia numérica se trata como otra entidad. Esto puede dejar sin agrupar una errata dentro de un número, porque no hay evidencia para distinguirla de dos entidades reales. El alcance declara los valores y pares comparados, los recortes por cardinalidad y si `stringdist` no estuvo disponible; también informa cuántas aristas se descartaron por esa regla y el tamaño compatible después del filtro. El tamaño máximo usado para el límite conserva el componente potencial antes del filtro numérico, para que una familia de entidades numeradas no vuelva a presentarse como una sola variante; el tamaño compatible se muestra aparte. Ese límite de-

cide sobre los grupos que contienen formas con números. Un grupo sin números, que el filtro nunca une con una forma que los tenga, se conserva además si el mismo límite medido sobre los grupos sin números lo admite: una familia numerada no calla una errata de texto de la misma columna. El denominador sigue siendo el vocabulario entero de la columna, con las formas con números, así que el veredicto es el que la columna daría si la familia no se encadenara: una cadena sin números que ocupa una fracción chica del vocabulario se publica igual junto a la familia que junto a códigos que no se encadenan o a cualquier otro texto. Entonces la cobertura dice que el diagnóstico no aplica a las formas con números y la evidencia agrega `grupo_maximo sin numeros`. Para mantener acotado el perfil, por omisión se evalúan hasta 5.000 valores distintos y 2.000.000 de pares de unidades normalizadas; si se alcanza un límite, la evidencia lo declara y no presenta el resultado como universo completo. Las fusiones exactas se informan aun sin ese paquete opcional. Antes de formar esos grupos se retiran los valores que el mismo perfil ya informó como `faltantes_disfrazados`. El diagnóstico fuerte de ausencia tiene precedencia: un centinela no se presenta también como posible variante de un valor válido. El alcance declara cuántas observaciones retiró este filtro.

La corazonada de centinelas numéricos se apaga sobre una **numeración limpia**, para no llamar ausencia a un código válido. `densa` responde a la cobertura del rango —`densidad_secuencia_entera` por encima de su umbral— y a que los valores **casi no se repitan**: al menos un 90 % de valores distintos, descontando el excedente de la moda. Cubrir el rango sola lo cumple también una magnitud entera acotada cuando se cargan filas suficientes, y con esa condición sola el veredicto dependería de cuántas filas se leyeron; una numeración que se repite —un código de catálogo, una clave foránea— no se distingue por su forma de esa magnitud y no se trata como numeración. Es la misma regla que deciden `posible_identificador`, la guarda de los límites de Tukey, la de los patrones raros y la de Benford. Por eso un código de catálogo que se repite —un `EstCod` de 1 a 284— recibe límites de Tukey y, con tres órdenes de magnitud, Benford, aunque se lo pase como texto o como factor: el paquete infiere número. Para eso está `columnas_codigo`. `moda_sobresale_secuencia_entera` es una señal independiente y no cambia los tres escudos de forma que dependen de la numeración. En una columna `integer64`, `bit64` se carga de manera diferida para registrar sus métodos antes de medir. Desde `bit64` 4.0.5, la versión más antigua medida, el perfil es el mismo que con la actual: el paquete empareja los valores por su texto, sin depender del método `mtfrm.integer64` que `bit64` registra desde la 4.6.0. Si `bit64` no está instalado, `perfilar()` no perfila la columna: se niega con la condición `lupa_error_integer64`, porque sin él R lee los valores como los bits de un doble e iguala entre sí todos los negativos, y los distintos, la moda y la mediana saldrían mal sin aviso. Si la columna tiene valores por encima de 2^53, en un doble se redondean y la densidad no se mide sobre números redondeados: las cinco medidas públicas de la secuencia quedan en `NA` y `cobertura_diagnosticos` declara que `secuencia_entera` no se evaluó, en lugar de darla como «no densa». La guarda vuelve a abrirse si un candidato presente queda fuera del rango de los valores restantes de la numeración, o si la frecuencia del candidato es la moda sobresaliente. Esta segunda señal se mide por el **salto entre frecuencias consecutivas**, ordenadas de mayor a menor y mirando sólo las primeras posiciones. Es el mismo idioma que `salto_de_escala_secuencia_entera` usa con los huecos: lo que delata a un centinela no es que su frecuencia sea grande, sino que haya un **acantilado** entre el grupo que sobresale y el resto de la distribución.

Comparar contra un valor concreto no alcanza, y cada intento lo rompió un caso distinto: contra el **segundo** valor, un señuelo legítimo y frecuente infla el segundo puesto; contra la frecuencia **típica** exigiendo forma de numeración, un centinela masivo sobre una clave foránea nunca la tiene; y con las dos juntas, **varios centinelas empatados se cubren entre sí** —tres valores con la misma frecuencia dejan «máximo igual al segundo» y la comparación vacía—. El acantilado ve los tres porque no le importa quién es el segundo sino dónde se corta la distribución.

Se miran sólo las primeras posiciones porque el grupo que sobresale está arriba: en la cola, una

caída de dos apariciones a una da un salto que no dice nada de la columna. Medido sobre las diez columnas numéricas del banco real y dos formas de clave foránea, ninguna pasa de 1,71; los casos que hay que atrapar van de 4,75 a 249. Perder la condición de numeración no genera un hallazgo por sí solo: sólo devuelve la columna a la mirada de la lista de centinelas. En una numeración compacta, un candidato fuera de rango también abre la guarda aunque su frecuencia no forme un acantilado.

El factor de frecuencia sólo califica la señal `moda_sobresale`: una caída menor no abre esa vía, pero tampoco bloquea la vía independiente del rango. Así, un candidato fuera de rango se informa aunque aparezca cuatro veces y no forme un acantilado. Lo que el usuario declara con `sentinelas_numericos` atraviesa la guarda y se informa con severidad `error`, que es la regla general del paquete: excluye lo que se declara e incluye lo que sospecha.

La clasificación de posibles datos personales es más amplia que la protección. Cada clasificación declara `poder_discriminante` y `proteger`:

- `debil`: una forma genérica, como siete a doce dígitos, coincide también con importes, facturas y códigos; se informa y no se ocultan su moda, sus ejemplos ni sus estadísticos. En la evidencia de sus hallazgos se tapan sus valores de seis caracteres o más —la colisión de una casi clave—, no la evidencia entera, así que la que no cita ningún valor se publica (`"4939 valores distintos de 5000 (0.988)"`, `"3 valores"`); sin la tabla —un perfil que se vuelve a proteger al guardarlo— no hay valores con que barrer y se tapa entera;
- `medio`: el nombre de la columna expresa una categoría personal (por ejemplo `telefono`, `fecha_nacimiento` o `fecha_fallecimiento`); se protege aunque sus valores no se puedan validar. El nombre tiene prioridad sobre una forma numérica genérica y también determina la etiqueta de tipo. Cada palabra vale también abreviada como se la escribe en una base —`tel`, `telef`, `fono`, `cel`, `dir`, `domic`, `nom`, `ape`, `doc`, `ced`, `e_mail`—, en plural —también la abreviatura: `tels`, `cels`, `fonos`, `noms`, `dirs`—, con un número pegado (`telefono2`, `apellido1`), en camelCase (`NombreCliente`, `nroDoc`), en las formas pegadas de los sistemas heredados, enumeradas (`nrotel`, `TELCEL`, `nomape`, `NOMBRECOMPLETO`, `nrodoc`, `dirpart`), y en la nomenclatura GIK de GeneXus, con la fecha abreviada `Fch` (`PerFchNac`, `CliFchNac`, `FchNac`, `PerFchDef`, `PerFchFall`; también `FNac`); `mov` no, que es también un movimiento, ni `FchFalla` o `FchDefinitiva`, que son otras fechas. La sigla del documento vale también escrita con un punto o un guion bajo tras cada letra, entera —`C.I.`, como se encabeza la cédula en una planilla, `C.I`, `C. I.`, `Nro. C.I.`, `C.I. Titular`, `D.N.I.`, y `c_i` o `nro_c_i`, como los deja `janitor::clean_names()`—: con `C.I.`, como con `CI`, las cédulas sin dígito verificador se protegen. `C.I.I.U.`, la actividad económica, no es un documento. Una palabra de nombre —`nombre`, `apellido`— la expresa en cualquier lugar del nombre: `nombre_cliente` es un nombre, y también `nombre_jefe_grupo` o `nombre_grupo`; una columna de números o de fechas no lo es (`cantidad_nombres`), y con un lugar —`nombre_departamento`, `nom_localidad`, la categoría de cualquier base pública— decide la señal de los valores, como la palabra de persona con un lugar. Un documento por el nombre tiene que tener algún valor con seis cifras o más: `cedula_valida` con `"S"`/`"N"` o `cod_documento` con `1`, `2`, `9` no lo son, si son al menos veinte valores de dos distintos o más; sin valores —una columna rala que la muestra de `perfilar_dbi()` no alcanzó—, con pocos o con un solo valor repetido —el `0` o el `"S/D"` del que no tiene— decide el nombre, porque el resumen de la tabla entera publicaría sus documentos. Y la dirección de una red o de un organismo no es un domicilio (`dir_ip`, `dir_web`, `direccion_general` con `"Direccion de Salud"`). Una palabra de **persona** —`cliente`, `socio`, `usuario`, `persona`, `paciente`, `titular`— dice de quién habla la columna, no que guarde su nombre: la clasifica como `nombre` sólo si ninguna otra palabra del

nombre la clasificó —`direccion_cliente` es un domicilio—, si el resto del nombre no dice que es otra cosa —una clave (`cliente_id`, `id_usuario`, `cod_socio`), un tipo, un estado o una categoría (`tipo_cliente`, `estado_socio`), un medio o una forma (`medio_pago_cliente`, `forma_pago_cliente`), un lugar (`pais_cliente`, `nacionalidad_socio`), una fecha (`cliente_desde`, `fecha_alta_socio`), una cantidad o un importe— y si sus valores pueden ser nombres: una columna de números, fechas o lógicos no lo es. No basta la palabra: si bastara, esas columnas se protegerían como nombres, perderían su moda y sus extremos, y sus categorías —`"mayorista"`, `"empresa"`— se taparían también en las demás columnas. La misma regla vale para las demás palabras: `tipo_documento` es una categoría y no un documento, `fecha_contacto` y `medio_contacto` no son teléfonos, `estado_correo` no es un correo y `tipo_domicilio` no es un domicilio; sí lo siguen siendo `nro_documento`, `documento_id`, `numero_celular`, `telefono_contacto`, `barrio_id`, `segundo_nombre` o una fecha de nacimiento guardada como número. Y una palabra que el resto del nombre califica se decide con sus **valores**, con el tipo que ellos muestran y no el de la palabra, en el 80 % de sus valores distintos: un correo —una `@`, también el de una intranet: `correo_alta` con `"jperez@intranet"`—; un teléfono de siete a quince cifras escrito como teléfono —con `+`, con el prefijo `00598` o `598` pegado, con un cero inicial, con separadores (también puntos de a dos cifras, `"099.12.34.56"`, o el espacio duro y la raya de una planilla), con el `.0` de una planilla, con un interno o en una lista: `responsable_canal` con `"+598 99 123 456"`—, en una columna que no es de fechas; con la palabra del teléfono, también las cifras sueltas (`celular_estado`), salvo que el resto del nombre diga que son un importe (`importe_telefono`) o un plazo (`telefono_plazo_dias`); el celular escrito de a tres sin su cero (`"99.123.456"`) con esa palabra, y el `598` pegado aunque la columna sea de números. No son teléfonos un importe —negativo, con centavos o con cuatro decimales (`"-12345.10"`, `"1234.5000"`), con separadores de miles (`"1.250.000"`, `"12 345.50"`, `"1.456.651,89"`)—, un número redondo, un código con un año o con un grupo rellenado con ceros (`"2024-00001"`, `"10-0012345"`) ni un documento escrito como documento; un importe con miles no empieza con cero y separa los decimales con otro signo que los miles, y el celular escrito de a tres con su cero (`"099.123.456"`, `"099 123.456"`, `"098.765.000"`) o el fijo `"4.732.1234"` son teléfonos. Un domicilio con una marca de calle —`Av.`, `calle`, `Bvar.`, `Br.`, `Cno.`, `Rbla.`, `Pje.`, `Gral.`, `ruta`, `esq.`, `apto`, `Dpto.`, `Mza.`, `km` seguido de un número— en al menos el 30 % de sus distintos y sin empezar con una palabra de categoría (`cliente_tipo` con `"Av. Italia 2345"`, también `"Rivera entre Soca y Ellauri 1234"`; `"En ruta"`, `"Radio 5 km"` no lo son). O un nombre con un nombre de pila conocido —también con iniciales, `"Juan A. Perez"`, `"Juan M. A. Perez"`, `"Juan A Perez"`, o un `O'`—, sin cifras, sin una palabra que no va en un nombre —`sin`, `en`, `por`—, sin empezar como un lugar —`San`, `Villa`— y que no sea una frase que empieza con un nombre de pila que es también una palabra —`Dolores`, `Victoria`— seguido de minúsculas (`"Dolores musculares agudos"`, `"Victoria local"`), aunque sean dos personas (`responsable_grupo` con `"Perez Joaquin"`, `contacto_alta` con `"Ramon Acosta"`); con otro nombre de pila la minúscula no cuenta, y `"Juan perez"` o `"Juan carlos Perez"` —lo que deja el autocapitalizado de un formulario— son nombres. Los acentos se leen también en `latin1` sin marca, como lo deja un CSV de Windows bajo `C`, también en el nombre de la columna: el encabezado `Dirección`, `Teléfono` o `Cédula` de un CSV de Windows leído sin declarar la codificación —`read.csv(check.names = FALSE)`, `data.table::fread()`— se reconoce en cualquier sesión. Y una fecha de nacimiento o de defunción calificada como categoría (`nacimiento_estado`) es esa fecha si sus valores son fechas; su lugar, su causa o una parte no (`pais_nacimiento`, `causa_defuncion`, `edad_fallecimiento`, `anio_nacimiento`). La forma sola no alcanza: `"Gran Empresa"`, `"Cerro Largo"`, `"Arroz Premium"` u `"Oficina 101"` siguen siendo cate-

gorías, y los nombres de pila que también son palabras de una categoría —Pilar, Marco, Norma, Franco, Domingo, Julio, Blanca, Celeste, Rosa, Max— no cuentan ("Primer Pilar", "Convenio Marco"). Y la palabra de quien actúa en el registro —usuario, funcionario, responsable, solicitante— con una de auditoría —alta, baja— dice quién dio el alta o la baja: ahí basta un texto que no sea una fecha ni un estado —un login "jperez", también con la inicial pegada por un guion bajo, un guion o un punto, delante o detrás ("a_perez", "o-silva", "perez.a": la a, la o o la u pegadas son una inicial, no la preposición ni la conjunción), un apellido solo, "A. Rodriguez"; no "Sin baja", "Baja definitiva" ni un estado o un motivo de una lista cerrada —"Cancelada", "Rechazada", "Renuncia", "Jubilacion", "Despido", "Fallecimiento", "Traslado", "Cese", "Fin de contrato", también en mayúsculas y en plural—; un login o un apellido que termina como un estado —"lmachado", "MACHADO, L.", "j.machado", "Machado"—, un nombre autocapitalizado —"Juan perez"— y un usuario del sistema —"SISTEMA", "MIGRACION"— cuentan como personas: junto a tres logins, uno solo que pasara por estado dejaría la columna en el 75 % y publicaría los logins de todos—; la de quien es objeto del registro pide la señal de los valores, y cliente_desde con "Sin fecha" o paciente_alta con "Traslado" se publican. El nombre no decide solo: si decidiera, esas columnas publicarían nombres, celulares y domicilios. Limitación conocida: la lista de nombres de pila no es completa, y un nombre que falta —o que es también una palabra corriente, como Max o Pilar— deja la columna sin esa señal: con tres personas, una basta para que no llegue al 80 %; una localidad con nombre de persona —"Jose Ignacio", "Juan Lacaze"— se lee como un nombre; con una palabra de categoría o de clave, un login de sólo letras o un apellido solo (responsable_grupo con "Rodriguez", usuario_id con "jperez_01") no se distinguen de una categoría o de un código y se publican, como un domicilio sin marca de calle; en la regla de quien actúa, un estado o un motivo que no está en la lista —"Ingresos superiores", "Abandono de cargo" sí está— se toma por una persona y la columna se protege; una frase que empieza con otro nombre de pila —"Mercedes centro"— se lee como un nombre, y "Victoria perez" escrito así no; el fijo con punto cuyo último grupo termina en cero ("4732.1230") se lee como un importe con cuatro decimales y el que tiene un grupo que empieza con 00 ("2400-0010") como un código rellenado, y no cuentan para el 80 % de la columna; con check.names = TRUE bajo C, R cambia la letra con tilde de un encabezado de Windows por un punto (Direcci.n) y ese nombre no se reconoce; un nombre GIK todo en minúsculas —perfchnac, pernom, como lo deja un motor que pliega a minúsculas los identificadores creados sin comillas— no se separa en palabras y no se reconoce, ni la sigla con el número delante pasada por clean_names() (n_c_i), ni con la N. del número detrás (C.I. N., que se lee como una sigla de tres letras), ni FchNac1, PerFchNacim o FCHNAC; FchFall o FchDef se leen como la fecha de defunción aunque sean la de una falla o una fecha definitiva, y se protegen de más; los códigos con un dígito tras un guion ("170177-3", "025-082916-3", "10-0123456") en una columna calificada se leen como teléfonos y se protegen; una palabra de persona con otra que no es un calificador conocido —victoria_equipo_socio— se protege por el nombre, y dir es una dirección aunque la columna sea de puntos cardinales; y las escrituras raras de un número —coma decimal con exponente, "4.685.467,00e0"— no se leen. Una columna de personas con otro nombre se declara en columnas_personales. También es medio una columna que trae correos completos sin que sean la mayoría de sus valores: un correo es un correo aunque esté solo, y la columna se protege;

- alto: una forma muy específica, como un correo, o nombre y forma se apoyan mutuamente; se protege;
- verificado: al menos tres valores distintos y al menos el 90% cumple uno de los validadores

personales configurados; se protege incluso sin un nombre orientador. La proporción medida —la que se compara contra ese umbral— se publica en `proporcion_verificada`, para que una columna con el 90% no se lea igual que una con el 100%. El pack uruguayo es el predeterminado, pero puede reemplazarse por un `pack_validadores()` de otro país o desactivarse con `FALSE`. La tolerancia del 10% permite tipeos aislados sin convertir una columna real en una salida pública; el umbral es configurable.

La forma genérica de siete a doce dígitos tiene poder discriminante débil: también describe importes, teléfonos, facturas e identificadores. Las formas con separadores sólo se aceptan cuando tienen una estructura de documento reconocible (por ejemplo, una cédula con grupos y guion o un RUT con grupos de tres y cuatro dígitos); una fecha ISO, una fecha con puntos o guiones y separadores arbitrarios no se consideran documentos. Un validador de dígito que supera el umbral aporta evidencia verificable y eleva la clasificación; una forma sola nunca se trata como prueba de identidad.

Este criterio mide capacidad de discriminación, no juzga si la presencia del dato es correcta. La protección sustituye modas, ejemplos, evidencia y extremos o medianas que corresponden a observaciones reales. También los momentos: las medias en sus dos formas (`media` y `media_fecha`), porque la media de una columna personal puede reconstruir demasiado, y el desvío, que es una magnitud de sus valores. `detalle_proteccion_personal` hace visible la supresión y dice qué se tapó: los estadísticos de orden, la media, el desvío o los dos momentos. En fechas de nacimiento y fallecimiento, un hallazgo separado conserva el diagnóstico de valores anteriores a 1900 o posteriores a la corrida sin publicar las fechas. Los números escritos como texto reconocen tanto coma como punto decimal y sus separadores de miles simétricos. El resumen cuantitativo, los outliers y la secuencia entera los leen con la convención que la misma fila publica en `numero_texto_convencion`, por la misma normalización que la acción `convertir_numero_regional`: con `"decimal_coma"`, `1.500` es mil quinientos. Un separador seguido de exactamente tres dígitos —`1.500`, `1,500`— sólo se lee si la columna aporta evidencia de una sola convención; si no la aporta (`"ambigua"`) o aporta de las dos (`"mixta"`), ese valor no se lee como número, con punto o con coma por igual: una columna entera de esos valores queda `texto` y sin resumen, y si son pocos quedan fuera del resumen y lo declaran `n_valores_excluidos_resumen` y `"calculados_sobre_valores"`. El resumen lee sólo las formas que la inferencia cuenta como número, también en las filas que su muestra no miró: un primer grupo de miles con cero adelante —`099.123.456`, un teléfono—, un hexadecimal o un `5.` quedan afuera y se declaran igual. Un número con moneda o unidad (`USD 1,234.56`, `12 kg`) no se resume: es un número escrito como texto, y `numero_texto_moneda` y `numero_texto_unidad` publican lo que traía. Los prefijos de tres letras separados del número, incluso como sufijo, `U$S` y los símbolos monetarios se conservan como evidencia; `monedas_mixtas` informa sus frecuencias sin convertir ni suponer tasas de cambio. Una única moneda o un símbolo `$` aislado no produce ese hallazgo. Un sufijo de unidad se reconoce sólo si es `%` o una abreviatura alfabética en minúsculas; por eso `12 kg` y `13500 g` son unidades, mientras que `12A` y `13B` se tratan como códigos. Si hay más de una unidad observada, `unidades_mixtas` informa sus frecuencias y no convierte ni compara sus magnitudes. Una única unidad no genera ese hallazgo. `formatos_fecha_mixtos` compara representaciones dentro de una columna cuyo dominio es la fecha o la fecha-hora. Una fecha agregada adentro de una columna de horas del día —`12/02/2011 6:55 a.m.` entre valores `7:10 a.m.`— no produce ese hallazgo, porque la columna no es de fechas: aparece como `patron_raro`, que es lo que de verdad cambió. Decidir que esa columna sólo debe contener horas es una regla de dominio y el paquete no la supone. Del mismo modo, un número sin sufijo no cuenta como una segunda unidad: una columna que mezcla `5` con `5 %` observa una sola unidad y se informa como `numero_como_texto`, no como `unidades_mixtas`. `celdas_multivaluadas` es deliberadamente conservador: usa los patrones de `descubrir_patrones()` y exige partes numéricas, alfanuméricas

o identificadoras puntuadas homogéneas, compatibles con el patrón del resto de la columna. No interpreta comas en nombres o direcciones como listas; el delimitador, la cantidad de celdas y la distribución de valores por celda quedan en la evidencia del hallazgo.

See Also

descubrir_patrones(), detectar_dependencias(), proponer_modelo(), planificar_limpieza()

Examples

```
perfil <- perfilar(datos_administrativos)
perfil
summary(perfil)
```

---

perfilar_coleccion *Perfilar una colección declarada*

Description

Recorre las tablas declaradas en coleccion() y devuelve **una fila por tabla** con su resumen exacto, más la cobertura de lo que no se pudo medir o no se solicitó.

Usage

```
perfilar_coleccion(
  coleccion,
  muestra = Inf,
  conservar_perfiles = FALSE,
  cobertura_metricas = c("no_medidas", "completa", "ninguna"),
  tope_cobertura_metricas = 20000L,
  bloque_muestra = c("con_muestra", "solo_agregados"),
  ...
)
```

Arguments

coleccion	Objeto creado por coleccion(). Una colección **guardada por una versión anterior** se perfila igual: las columnas que a un objeto viejo le faltan —catalogo, tipo, declaracion— se completan con el valor que significa «esta colección no lo declaró», mientras identificador y referencia, que son derivados, se recalculan con las mismas funciones que los arman. Lo único que no se puede inventar es tabla, que es el nombre mismo: su ausencia se rechaza nombrando la columna. Así una columna faltante no corta la corrida con un error que no nombra ni la columna ni la tabla, ni deja **cero** tablas perfiladas sin abortar: ante un fallo parcial se devuelve lo medido con su alcance declarado.

`muestra` Filas solicitadas por tabla para el bloque en memoria. Por omision `Inf`: cada tabla se trae entera. El resumen por tabla se calcula en el motor y no depende de esto; lo que depende son los diagnosticos que miran los valores en R. Acotarlo con `muestra = n` es mas rapido a cambio de mirar menos filas, y sobre una coleccion grande conviene mirar antes `plan_perfilado_dbi()` tabla por tabla.

`conservar_perfiles`
Si se retienen los objetos `perfil_dbi` completos. Por omisión `FALSE`. Retenidos y con la protección activa, cada uno se protege además con los valores protegidos de las **demás** tablas de la colección: una observación de `movimientos` que cita la cédula y el nombre de un cliente se tapa aunque `movimientos` no tenga ninguna columna personal, porque la colección es un solo objeto. Esos valores se leen de la base, una consulta por columna protegida, con la regla de `perfilar()` —también la de los rellenos, con sus frecuencias—; si alguna lectura falla —también si una tabla entera no se pudo perfilar, por ejemplo porque el rol no tiene permiso sobre ella, o no se perfiló porque se declaró de otro tipo —`tipo = "vista"`—: sus valores son desconocidos—, o una tabla no se pudo clasificar porque su muestra no se leyó, en los perfiles de las otras tablas se tapan los valores y la evidencia de las columnas no personales, y `resumen_tabla$meta$proteccion_personal$otras_tablas` dice por qué. Y con esa regla, una columna que copia SIN su verificador los documentos protegidos de OTRA tabla —`tramites.ref` con la cédula de `personas` sin el dígito— se protege entera, como la que los copia en su propia tabla, y su fundamento dice de qué tabla salen: sus valores distintos se leen de la base, una consulta por columna no protegida que publica un número de seis cifras o más, que la muestra clasificó como documento o cuyos valores la muestra describe como cifras con separadores —`"4.685.467,00"`, `"4 685 467.0"`, `"4.685467e6"`, aunque los acompañe un `"S/D"`—, con la misma expresión con que la leyó `perfilar_dbi()` —`money` y `real` convertidos—; la prueba es sobre la tabla entera y en cualquier orden de las tablas, y la tabla con una copia se vuelve a perfilar con esa columna como personal; si la lectura o el nuevo perfil fallan, el perfil de esa tabla se cierra igual. Limitación conocida: ese nuevo perfil reemplaza al conservado, pero `resumen_coleccion`, `cobertura_coleccion`, `cobertura_metricas` y `meta$lecturas` quedan con lo del primero —los estados de las métricas de la copia dicen `calculado` donde el perfil dice que se ocultaron—; no publican sus valores. Con esos valores se barren también `cobertura_coleccion`, `cobertura_metricas` y `meta$lecturas`, que juntan los motivos de todas las tablas. Con o sin retenerlos, esos tres campos, los perfiles retenidos y el mensaje del motor de cada tabla pasan además por el piso con los valores que las tablas leyeron y que un mensaje del motor puede llevar —todo valor con letras, porque cualquier palabra puede ser el nombre de una columna de otra tabla, y el número que el mensaje no omite entero, como `"099;123;456"`, escrito con un signo que no une; ver `perfilar_dbi()`—, sin consultas de más: así se tapa el mensaje de una vista que cita `"Campos Cadena"`, la celda de su tabla base; y con perfiles retenidos y una tabla base incompleta —una columna protegida `ntext` de SQL Server, que no admite el `GROUP BY` que lee sus valores— el piso no se limita a las tablas completas, y el motivo de la vista no cita el valor entero. De cada uno se guarda una huella —ocho bytes de su forma plegada y sin separadores— y no el texto: la memoria no crece con el largo de los nombres que se leen, y un tramo del mensaje con la

huella de uno se tapa como esté escrito; dos formas con la misma huella, una en 2^52, sólo pueden tapar de más. Y cada mensaje del motor se guarda como llegó mientras dura la colección —en memoria; nunca entra al objeto— y al final se vuelve a leer con todo lo que la colección leyó: con una vista perfilada antes que su tabla base, el piso todavía no conoce el nombre cuando llega el mensaje, y releer el texto ya tapado dejaría su resto —`"De Los Campos"` de `"María De Los Campos"` en SQL Server en sueco, `"Campos Cadena"` después de un apodo entre comillas—. El orden de las tablas no cambia lo que se publica; lo vuelto a leer reemplaza a lo publicado sólo si publica menos. La relectura corre también después del cruce entre tablas, sobre lo que el cruce escribe: el motivo de `otras_tablas` lleva el mensaje del motor de la tabla que no se pudo leer, y con la vista antes que su base citaría ese resto. En ese cierre, un valor igual al nombre de una tabla o de un esquema de la colección no se tapa: es estructura, como en `perfilar_dbi()`. Se reemplaza el texto tal como se publicó: si el piso de la propia tabla ya lo cambió —porque el mensaje cita entero, además, un valor protegido de esa misma tabla—, la relectura no lo encuentra y queda sólo el piso de las huellas, que tapa los valores enteros y no el resto. Y los avisos de consola de cada tabla se emiten al final, después del mismo piso que esos tres campos, con o sin perfiles: emitido en el momento, el aviso de una vista perfilada antes que su tabla base citaría la celda. También si la colección se corta —una interrupción, un error— después de leer alguna tabla: los avisos juntados salen por ese mismo piso, con los valores conocidos hasta el corte. `perfilar_dbi()` sobre una tabla sola protege sólo con los de esa tabla. Sin protección (`proteger_datos_personales = FALSE`) no se cruza nada: cada perfil retenido sale como lo dio su tabla, también si otra no se pudo perfilar.

`cobertura_metricas`
: Qué se sube a `cobertura_metricas`: `"no_medidas"` (por omisión) sólo las métricas que no se calcularon, `"completa"` todas con su estado, `"ninguna"` para omitirla. Los conteos por tabla de `resumen_coleccion` no dependen de esta elección.

`tope_cobertura_metricas`
: Máximo de filas de `cobertura_metricas`. El total real queda siempre en `meta$n_metricas_no_medidas` aunque la tabla se haya recortado.

`bloque_muestra`
: Qué bloques se solicitan en cada tabla: `"con_muestra"` (por omisión) calcula también el perfil de la muestra, o `"solo_agregados"` evita leerla y devuelve sólo los agregados SQL. Se pasa al mismo argumento de `perfilar_dbi()` para que todas las tablas de la colección respeten la misma decisión.

`...`
: Argumentos enviados a `perfilar_dbi()`. `proteger_datos_personales` vale para toda la colección: sin protección, el mensaje de un error del motor se publica entero —el literal que hizo fallar la lectura y el SQL— también en `cobertura_coleccion`, `cobertura_metricas`, `meta$lecturas` y los avisos; ver la sección «Datos personales» de `perfilar_dbi()`. Lo que va a `perfilar()` se valida al entrar, una vez. Los argumentos propios de `perfilar_dbi()` —`estrategia_distintos`, `max_consultas`, `umbral_cardinalidad` y los demás— los valida `perfilar_dbi()` en cada tabla: uno mal escrito deja cada tabla en `cobertura_coleccion` con ese motivo y sin aviso. Es una limitación conocida: si ninguna tabla se perfiló, mirar el motivo antes de sospechar de los permisos.

Details

Conserva el resumen calculado en SQL sobre toda la tabla, que es acotado, y **no** retiene el perfil de la muestra de cada tabla, que es el objeto pesado: con cientos de tablas eso no entra en memoria. `conservar_perfiles` permite retenerlos cuando la colección es chica y se los necesita.

Cada fila de `cobertura_coleccion` nombra la tabla con las tres partes de su identidad —`catalogo`, `esquema` y `tabla`—, las mismas que `coleccion()` declaró. Es lo que permite que el motivo siga pegado a la tabla cuando el número sube de nivel: sin el catálogo, una tabla declarada como `catalogo.esquema.tabla` se publicaría río abajo como `esquema.tabla`.

Lo que no se pudo medir se declara, y nunca a cero. Una tabla que la credencial no puede leer, un objeto que no es una tabla base y una tabla vacía van a `cobertura_coleccion` con su motivo y su `como_resolverlo`. Un motor que rechaza un agregado —o una columna cuyo tipo no lo admite— va a `cobertura_metricas`, con una fila por columna y métrica, y deja además una línea de resumen en `cobertura_coleccion`. Esa cobertura por métrica **se calcula antes de descartar el perfil**, así que existe también con `conservar_perfiles = FALSE`. En bases institucionales los permisos parciales son el caso normal, no el borde.

Donde no hubo medición va `NA`, no `0`. Una tabla vacía deja `prop_faltantes` en `NA` en todas sus columnas, y entonces `prop_faltantes_maxima` y `n_columnas_sin_faltantes` también quedan en `NA`: un máximo de `-Inf` saldría del `[0, 1]` que el paquete promete y ordenaría la tabla como la de mejor calidad de la base, y un conteo afirmaría algo sobre una tabla de la que no se sabe nada. `n_columnas_medidas` declara sobre cuántas columnas se conoce la proporción de ausentes, para que los dos números anteriores se lean con su alcance.

No hay lectura instantánea. Perfilar una colección son muchas consultas y la base puede cambiar entre ellas: una tabla puede truncarse después de que se contaron sus filas. Por eso cada fila declara el `momento` en que se midió y `meta$snapshot` declara que no lo hubo.

Cada fila de `resumen_coleccion` trae `universo`, que nombra el universo de las metricas SQL que alimentan sus agregados. En particular, con `universo = "muestra_motor"`, `n_filas` sigue siendo el conteo de la tabla completa pero las proporciones observadas declaran que provienen de la muestra, en vez de presentarse como cifras de la tabla entera.

`meta$lecturas` conserva el SQL emitido por las tablas perfiladas, con su identidad, aunque `conservar_perfiles = FALSE`. `sql_perfil()` lo devuelve como una tabla unica; los perfiles completos siguen disponibles por tabla cuando se pidio conservarlos.

Limitación conocida: las declaraciones por columna valen para todas las tablas. `columnas_codigo`, `columnas_opcionales`, `columnas_personales` y las demás llegan por `...` iguales a cada `perfilar_dbi()`, y `coleccion()` no tiene declaraciones por tabla. Una tabla que no tiene una columna declarada en `columnas_codigo` no se perfila —sale de `resumen_coleccion` y `cobertura_coleccion` nombra la causa, «`columnas_codigo` nombra columnas inexistentes», aunque `como_resolverlo` hable de permisos—; con `columnas_opcionales` o `columnas_personales` la tabla conserva su resumen y pierde la muestra. `columnas_sin_ceros` y `columnas_no_negativas` no la afectan. Para declarar columnas que no están en todas las tablas, perfilar esas tablas por separado o partir la colección.

Value

Objeto de clase `perfil_coleccion` con `resumen_coleccion`, `cobertura_coleccion`, `cobertura_metricas`, `meta` y, si se pidió, `perfiles`.

See Also

coleccion(), perfilar_dbi()

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "personas", data.frame(id = 1:3, nombre = letters[1:3]))
  perfilar_coleccion(coleccion(con, "personas", nombre = "padron"))
  DBI::dbDisconnect(con)
}
```

---

perfilar_dbi *Perfilar una muestra leída mediante DBI*

Description

Calcula en SQL un resumen sobre la tabla completa o sobre una relación muestreada por el motor, según `universo`. Con `universo = "muestra_motor"` y `bloque_muestra = "con_muestra"`, ejecuta una sola selección y la materializa en un spool externo de la sesión cliente; el resumen y perfilar() leen esa misma materialización. Con `bloque_muestra = "solo_agregados"`, el resumen usa la relación muestreada sin seleccionar filas para el cliente. `bloque_muestra = "solo_agregados"` sigue omitiendo el objeto `perfil_muestra`, pero no vuelve a seleccionar filas.

Usage

```
perfilar_dbi(
  conexion,
  tabla,
  universo = c("tabla_completa", "muestra_motor"),
  muestra_motor = NULL,
  muestra = Inf,
  orden_muestra = NULL,
  metricas = .METRICAS_DBI,
  estrategia_distintos = "exacta",
  estrategia_mediana = c("exacta", "aproximada_motor"),
  politica_costo = c("todas", "por_cardinalidad"),
  bloque_muestra = c("con_muestra", "solo_agregados"),
  max_consultas = Inf,
  dialecto = "auto",
  incluir_valores = TRUE,
  tamano_lote = NULL,
  tamano_lote_planos = .TAMANO_LOTE_PLANOS_DBI,
  tamano_lote_distintos = .TAMANO_LOTE_DISTINTOS_DBI,
  instrumentar = TRUE,
  umbral_cardinalidad = .UMBRAL_CARDINALIDAD_COSTO_DBI,
```

```
  avisar_costo_distintos = TRUE,
  umbral_segundos_aviso_distintos = .UMBRAL_SEGUNDOS_AVISO_DISTINTOS_DBI,
  avisar_costo_moda = TRUE,
  umbral_segundos_aviso_moda = .UMBRAL_SEGUNDOS_AVISO_MODA_DBI,
  avisar_costo_mediana = TRUE,
  umbral_segundos_aviso_mediana = .UMBRAL_SEGUNDOS_AVISO_MEDIANA_DBI,
  avisar_derrame_estimado = TRUE,
 umbral_bytes_aviso_derrame_estimado = .UMBRAL_BYTES_AVISO_DERRAME_ESTIMADO_DBI,
  max_celdas_muestra = .MAX_CELDAS_MUESTRA,
  max_bytes_muestra = .MAX_BYTES_MUESTRA,
  bloque_filas = NULL,
  max_bytes_procesamiento = .MAX_BYTES_MUESTRA,
  max_bytes_materializacion = .MAX_BYTES_MUESTRA,
  columnas_opcionales = character(),
  aplicabilidad = NULL,
  sentinelas_numericos = c(-9, -99, -999, -9999, 999),
  ...
)
```

Arguments

conexion	Conexión abierta compatible con DBI.
tabla	Nombre de tabla o un objeto aceptado por DBI::dbQuoteIdentifier().
universo	Universo sobre el que se calculan los agregados SQL: "tabla_completa" (por omisión) o "muestra_motor". En el segundo caso, todas las métricas SQL usan la relación muestreada por el motor y no se reemplazan silenciosamente por resultados de la tabla completa.
muestra_motor	Cantidad positiva y finita de filas que el motor debe tomar cuando universo = "muestra_motor". Es obligatorio en ese universo y debe quedar NULL para "tabla_completa"; la función rechaza temprano valores no enteros, no positivos o Inf. Si una vista falla en una fila, el resumen conserva el alcance de la materialización que sí se pudo leer: las columnas legibles se perfilan y la columna rota queda no_disponible; la cobertura no presenta esa lectura parcial como una lectura completa de la vista.
muestra	Cantidad positiva de filas solicitadas para el perfil de muestra que se trae a R, o Inf para traer la tabla entera. Este límite es independiente de universo: en muestra_motor, muestra_motor decide las filas del resumen SQL y muestra decide las filas del bloque perfil_muestra. En tabla_completa, muestra no cambia los agregados. Cuando universo = "muestra_motor", el resumen usa toda la materialización del motor y este argumento solo recorta el diagnóstico en R; por eso meta$materializacion$n_filas puede ser mayor que perfil_muestra$meta$filas. Sin orden_muestra, las filas del bloque en R no son una muestra aleatoria garantizada sino las primeras que devuelva el motor. El límite también alcanza la muestra común con que se buscan dependencias. Use un entero finito para acotar ese trabajo cuando el tiempo no sea la restricción.
orden_muestra	Columnas para ORDER BY. La salida solo declara orden reproducible cuando la combinación es única en toda la tabla. Sin este argumento, DBI no garantiza el

orden ni la pertenencia de una muestra limitada, y meta lo declara expresamente. No se usa cuando bloque_muestra = "solo_agregados". Con la lectura por bloques, la identidad de la fuente por bloques gobierna el recorrido y este pedido queda declarado en meta$orden_muestra con el motivo estable de que no gobierna esa via. Si el motor falla al evaluar una columna del ORDER BY, la vía prueba cada componente, recupera las columnas legibles sin ese orden cuando puede y publica alcance_distinto con el motivo de que el orden pedido no se pudo aplicar; no lo presenta como si ninguna columna fallara.

metricas
: Selección explícita de grupos de métricas: "validos", "distintos", "moda", "basicos", "mediana" y "desvio". El valor por omisión solicita las seis. c("validos", "basicos", "desvio") omite distintos, moda y mediana, y "validos" sólo cuenta los no nulos.

estrategia_distintos
: Procedencia explícita para n_distintos: "exacta" (por omisión) emite COUNT(DISTINCT); "aproximada_motor" usa una función nativa aceptada por el motor y deja la métrica en no_disponible si no existe; "catalogo" lee pg_stats.n_distinct en PostgreSQL y publica el resultado como estimado_catalogo, nunca como medición, cuando universo = "tabla_completa". En muestra_motor queda no_disponible, porque el catálogo describe la relación entera y la corrida mide un subconjunto; y "omitida" no emite ninguna consulta. No hay repliegue automático entre estrategias. El resultado publica estrategia_solicitada, estrategia_resuelta y estado en meta$estrategia_distintos, y las dos primeras también en resumen_tabla$sql. En pg_stats, un valor positivo es el conteo estimado y uno negativo es una fracción de las filas. Cuando la relación tiene descendientes se elige inherited = TRUE, porque esa fila describe lo que lee una consulta sin ONLY; una relación sin hijas usa su única fila propia. Las fracciones se convierten con la suma de pg_class.reltuples de la jerarquía. Si no hay una fila utilizable —por ejemplo, antes de ANALYZE— o hay ambigüedad, la métrica queda no_disponible, no en cero.

estrategia_mediana
: Preferencia para resolver mediana: "exacta" (por omisión) o "aproximada_motor". La sonda prueba siempre primero una forma nativa exacta consolidada, luego una forma exacta por columna y deja las funciones nativas aproximadas para el final. Por eso esta opción describe la estrategia habilitada, no garantiza el método ejecutado. El método que efectivamente corrió está en resumen_tablasqlmetodo, en la fila de metrica = "mediana" de cada columna, y es la única fuente por columna. meta$estrategia_mediana publica la resolución de la tabla: lo pedido (estrategia_solicitada), la forma que el motor aceptó para la tabla (estrategia_resuelta), su estado y su motivo. No es el método de cada columna: una columna que no puede usar esa forma —un float8 de PostgreSQL, que la deja por la lectura exacta, o una columna con extremos— corre por otra (dos_consultas en la tabla completa, la CTE de ventanas en la muestra) y lo declara en su fila de resumen_tabla$sql, aunque meta$estrategia_mediana siga nombrando la consolidada. El estado de meta$estrategia_mediana describe la forma, no la fila: en universo = "muestra_motor" la fila de cada columna queda estimado. Las columnas estrategia_solicitada, estrategia_resuelta y estado_estrategia de resumen_tabla$sql son las de n_distintos (meta$estrategia_distintos) repetidas en todas las filas: en la fila de la mediana no describen la mediana.

Una mediana resuelta por una forma exacta queda `estado = "calculado"` y `error_esperado = "no_aplica"`, aunque se haya pedido `"aproximada_motor"`; sólo una aproximación ejecutada queda `estado = "estimado"`. `universo = "muestra_motor"` combinado con `"aproximada_motor"` se rechaza temprano.

`politica_costo` Política optativa para las métricas caras. El valor por omisión, `"todas"`, conserva moda y mediana para todas las columnas solicitadas. `"por_cardinalidad"` resuelve primero las fuentes estructurales y mide valores válidos y distintos sólo cuando hace falta y la estrategia lo permite. Luego omite, por columna, sólo la moda cuando la proporción de distintos alcanza `umbral_cardinalidad`; la mediana se conserva porque las mediciones disponibles muestran que su costo depende de las filas y no de la cardinalidad. Los únicos valores aceptados son `"todas"` y `"por_cardinalidad"`; no hay alias históricos.

`bloque_muestra` Qué bloques se solicitan: `"con_muestra"` (por omisión) calcula también `perfil_muestra`, o `"solo_agregados"` omite su lectura y devuelve sólo los agregados SQL. La decisión vale para los dos universos: con `muestra_motor`, los agregados siguen usando la relación muestreada, pero no se abre el spool ni se leen filas; la corrida publica el estado `no_solicitado` y `meta$bloques$filas_vistas = 0`. La segunda opción no cambia el alcance de esos agregados: eso lo decide `universo`.

`max_consultas` Presupuesto declarado de consultas. Al agotarse, las métricas restantes quedan en `no_disponible` con ese motivo. Cuenta toda sentencia que el perfilado manda al motor —la preparación, las lecturas de catálogo, el recorrido por bloques y, por el spool de la muestra del motor, la preparación que la corrida repite para publicar su plan—, la misma cuenta de `plan_perfilado_dbi()` y de `resumen_tabla$meta$consultas$emitidas`. No cuentan las sentencias de la guarda de transacción (`SAVEPOINT` y, en SQL Server, la pregunta por la transacción abierta antes de cada consulta), ni las lecturas de `pg_stat_statements` de `instrumentar`, ni las que algunos controladores mandan para contestar `DBI::dbGetInfo()` o `DBI::dbIsValid()`: duckdb 1.1 pregunta su versión y manda `SELECT 1`, una o dos veces por corrida. La preparación y el recorrido por bloques se cuentan pero no se omiten: sin ellos no hay perfil. Con `max_consultas = attr(plan, "total_maximo")` ninguna métrica queda afuera por el presupuesto.

`dialecto` Capacidad de acotar filas: `"auto"` la sondea, y `"limit"`, `"top"`, `"fetch_first"`, `"rownum"` o `"portable"` la declaran sin sondeo.

`incluir_valores`

Si publica valores de celda. Con `FALSE` no se emiten las consultas de moda, extremos y mediana, y se limpian esos campos en el resumen, la muestra, los bloques, la cobertura y los motivos; también se retiran ejemplos de patrones, familias de acumuladores y cualquier muestra cruda. El mensaje de un error del motor, que cita la celda que lo provocó, se publica sin sus literales, como con `proteger_datos_personales = TRUE`. Los conteos, proporciones y agregados que no son valores de celda pueden seguir publicándose. La elección queda en `resumen_tabla$meta$incluir_valores` y `perfil_muestra$meta`.

`tamano_lote` Cantidad máxima de columnas por consulta consolidada. Se conserva por compatibilidad y, si se informa, fija el tamaño de las dos familias. Para control separado, usar `tamano_lote_planos` y `tamano_lote_distintos`.

`tamano_lote_planos`

Cantidad máxima de columnas por consulta de agregados planos. En SQL

Server y en columnas `float8` de PostgreSQL, el desvío conserva su cálculo centrado en una segunda pasada del mismo tamaño. El valor por omisión es 20.

`tamano_lote_distintos`
: Cantidad máxima de columnas por consulta de cardinalidades exactas. El valor por omisión es 2, un supuesto del código medido sobre un PostgreSQL de referencia: el motor hace una sola pasada por la tabla para todos los conteos de una consulta, así que un lote de uno cuesta por columna bastante más que uno de dos; uno de dos cuesta por columna casi lo mismo que uno de cuatro y derrama menos, y los lotes mayores derraman más porque los conteos compiten por `work_mem`. Una sola cardinalidad todavía puede forzar un agregado pesado y derramar mucho más que un lote plano.

`instrumentar`
: Si se cronometra cada consulta y las etapas grandes de R y, en PostgreSQL, se intenta atribuir el uso de bloques temporales de los `COUNT(DISTINCT)` exactos mediante `pg_stat_statements`. Por omisión es `TRUE`; agrega `duracion_ms`, `cpu_ms`, `n_filas_resultado`, `bytes_resultado_r`, `consulta_id`, `etapa` y `nivel` a `resumen_tabla$sql`, y el resumen `resumen_tabla$tiempos`. Con `FALSE` se conserva el mismo plan, la misma cantidad y el mismo orden de consultas, pero los campos medibles quedan en `NA`. `id_consulta` **no** depende de esta opcion: no es una medicion sino un hecho estructural sobre que consulta produjo cada metrica, y se publica igual con `FALSE`. Las duraciones usan `Sys.time()` y el CPU del cliente usa la suma de `proc.time()[c("user.self", "sys.self")]`. `cpu_ms` es cero cuando el proceso no consumió CPU; `NA` significa que no se pudo medir. Los intervalos que el reloj no puede resolver no se publican como cero. En PostgreSQL, la atribución del derrame lee `pg_stat_statements` dos veces —antes y después—, limitado al usuario y la base de la conexión, y atribuye solo las consultas cuyas llamadas crecieron en esa ventana; una entrada que el catálogo repite —la misma consulta de otro usuario o anidada— no se atribuye. Medido con unas 4.900 filas en el catálogo: sobre una tabla de 23 filas, 0,3 s con `TRUE` contra 0,15 s con `FALSE`.

`umbral_cardinalidad`
: Proporción entre valores distintos y válidos que activa la omisión de la moda con `politica_costo = "por_cardinalidad"`. El valor por omisión es `0.5` sólo cuando esa política se pide explícitamente; se puede mover en cada llamada. Este argumento no gobierna la mediana: `meta$decisiones_costo` explica la decisión de cada métrica por separado. Para pedir todas las métricas use `politica_costo = "todas"`.

`avisar_costo_distintos`
: Si es `TRUE`, avisa, después de medir el primer lote y antes del segundo, cuando la proyección del costo de `COUNT(DISTINCT)` alcanza `umbral_segundos_aviso_distintos`. Por omisión es `TRUE`. Este aviso no depende de `interactive()`: también llega en guiones porque el costo se paga en el servidor y puede durar decenas de segundos.

`umbral_segundos_aviso_distintos`
: Segundos estimados a partir de los cuales se emite el aviso del costo de `COUNT(DISTINCT)`, después del primer lote. Por omisión es `30`, el umbral histórico; `Inf` lo desactiva explícitamente. Con un solo lote no se publica una proyección porque el costo ya se pagó. El valor no cambia la proyección ni la medición que se publica.

avisar_costo_moda
: Si es TRUE, avisa antes de ejecutar las modas pendientes cuando su proyección alcanza umbral_segundos_aviso_moda. Por omisión es TRUE. La tasa se mide con modas anteriores de esta corrida; si falta cardinalidad, se declara en la proyección y no se supone.

umbral_segundos_aviso_moda
: Segundos estimados a partir de los cuales se emite el aviso de la moda. Por omisión es 30; Inf lo desactiva explícitamente. El valor no cambia la medición ni la proyección publicada.

avisar_costo_mediana
: Si es TRUE, avisa antes de ejecutar las medianas pendientes cuando su proyección alcanza umbral_segundos_aviso_mediana. Por omisión es TRUE. La proyección sigue las filas, no la cardinalidad.

umbral_segundos_aviso_mediana
: Segundos estimados a partir de los cuales se emite el aviso de la mediana. Por omisión es 30; Inf lo desactiva explícitamente. Cuando no existe una primera medición local, una sola mediana total usa la referencia de banco declarada de 68 ms por millón de filas de otra corrida. Si la consulta inicial que obtuvo las filas fue medida y da una cota mayor, se publica como cota de lectura, no como medición de mediana.

avisar_derrame_estimado
: Si es TRUE, avisa cuando un lote de COUNT(DISTINCT) supera la memoria efectiva y su tamaño estimado alcanza umbral_bytes_aviso_derrame_estimado. Por omisión es TRUE. Este aviso sólo puede aparecer cuando PostgreSQL permite estimar el hash.

umbral_bytes_aviso_derrame_estimado
: Tamaño estimado del hash, en bytes, a partir del cual se avisa un derrame potencial entre los lotes que ya superan la memoria efectiva. Por omisión es 0, que conserva el aviso para cualquier lote que la supere; Inf lo desactiva explícitamente. El valor no cambia la estimación ni la medición posterior del derrame.

max_celdas_muestra
: Máximo de celdas que puede contener el bloque perfil_muestra. Por defecto es 1000000; se calcula antes de leer como filas por columnas del esquema. Si reduce la muestra, perfil_muestra$cobertura_diagnosticos usa la misma declaración que perfilar(), con las celdas solicitadas, el umbral y el tope que mandó. Inf desactiva este tope. No modifica los agregados SQL.

max_bytes_muestra
: Máximo de bytes de la muestra materializada que alimenta perfil_muestra. Por defecto es 512 MiB. En muestra_motor, el límite se cruza con el presupuesto de materialización y se respeta antes de cada dbFetch(n): cuando la cota de tipos es demasiado conservadora se lee una fila inicial, se mide lo que ocupa en la materialización y los bloques siguientes se piden con el doble de margen sobre la fila más grande medida; el spool nunca pasa el tope. Si reduce la muestra, meta$muestreo, meta$materializacion y cobertura_diagnosticos informan las filas solicitadas, efectivas y cuál tope mandó. meta$muestreo$filas_por_bytes publica la cantidad obtenida de la medicion y meta$muestreo$filas_por_bytes_fuente lo declara. Inf desactiva este tope. No modifica los agregados SQL.

bloque_filas
: Entero positivo que activa el recorrido de tabla_completa por bloques. Se abre un unico result set con dbSendQuery() y cada bloque se lee con dbFetch(n = bloque_filas) antes de liberar la entrada; NULL conserva la via existente. El resultado publica meta$alcance$orden, meta$bloques, meta$bytes y los eventos del vigilante. Los drivers sin retencion incremental demostrada (incluido DuckDB en esta matriz) quedan no_disponible con su motivo y un aviso lupa_aviso_muestra_motor, sin caer en dbGetQuery(). Si se pide moda o mediana, la lectura por bloques inicia tambien el mapa central de distintos aunque n_distintos no se haya pedido: la mediana depende de ese mapa y la fila n_distintos conserva no_solicitado. Las columnas binarias compatibles se solicitan en la misma forma que la muestra, largo:md5, para que R no materialice los bytes crudos; los largos que se publican se reconstruyen desde el prefijo y son los largos originales, no el largo de esa cadena. Bajo bloque_filas, perfil_muestra sigue siendo la muestra diagnóstica acotada por muestra, max_celdas_muestra y max_bytes_muestra; su general$filas no es la cobertura, que se publica en meta$bloques$filas_vistas. Si el recorrido por bloques no puede leer una columna de una vista, se vuelve a leer la muestra diagnóstica por columnas: las columnas sanas conservan su perfil y sus magnitudes, y la columna rota queda en no_disponible con el motivo del motor. No se publica como privacidad ni como no_solicitado.

max_bytes_procesamiento
: Límite de bytes del estado retenido por el procesamiento por bloques y por la lectura del spool en R. Se comprueba antes de publicar un bloque o una pasada; si la materialización cabe pero el objeto que R necesita para perfilarla supera el límite, el perfil se descarta y se emite el mismo aviso de alcance degradado, con tope_que_mando = "procesamiento". Inf desactiva este límite.

max_bytes_materializacion
: Presupuesto de bytes del spool externo de muestra_motor. Se mide por chunk antes de cada escritura, dejando espacio para el trailer; si no alcanza, se publica spool_presupuesto_excedido y se conserva la parte que entró como muestra recortada, con muestra_inestable:presupuesto_materializacion. En muestra_motor, el primer bloque se acota antes de pedirlo y los siguientes se adaptan a los bytes medidos; el resultado conserva meta$bytes_muestra y meta$tiempos. Inf desactiva este límite.

columnas_opcionales
: Nombres de columnas que sólo son aplicables a parte de las filas. Esta política se aplica al perfil_muestra; el resumen SQL no la traduce y lo declara en resumen_tabla$cobertura. Los dos bloques cuentan n_faltantes sobre universos distintos, así que la corroboración no lo compara: ver «Lo que el motor guarda distinto que R».

aplicabilidad
: Lista nombrada de fórmulas de un solo lado, con una regla por columna. Se reenvía al perfil_muestra; el resumen SQL no la traduce y lo declara en resumen_tabla$cobertura, y la corroboración no compara lo que la regla cambia.

sentinelas_numericos
: Vector completo de valores numéricos que se declaran como faltantes en el perfil_muestra. Se normaliza como un conjunto numérico, por lo que el orden, el tipo entero o doble y los duplicados no cambian el resultado. El resumen

SQL no los excluye de sus agregados y lo declara en `resumen_tabla$cobertura`; la corroboración no compara las métricas de magnitud de una columna de la que la muestra excluyó alguno. En una columna `real`, una representacion float32 del centinela fraccionario se alinea antes de perfilar la muestra; la corroboración reconoce esa representacion como el mismo centinela.

Rige la misma reserva que en `perfilar()`: un valor de la lista **por omisión** que aparece una sola vez y dentro del rango de los demás no se señala. Declarar el valor lo vuelve a contar.

`...` Argumentos enviados a `perfilar()` para analizar la muestra. `columnas_codigo` vale además en `resumen_tabla`: `media`, `mediana` y `desvio` de una columna declarada quedan en `NA` y `resumen_tabla$cobertura` lo declara con estado `no_aplica`. Sus nombres se validan contra las columnas de la tabla aunque no haya muestra. Un nombre abreviado vale lo mismo que el entero, como lo completa R al llamar a `perfilar()`: `proteger = FALSE` es `proteger_datos_personales = FALSE` también para `resumen_tabla`. La bandera, `fecha` y los nombres se validan al entrar, también con `bloque_muestra = "solo_agregados"`: un nombre que `perfilar()` no tiene es un error, no un fallo parcial. Los demás valores los valida `perfilar()` al perfilar la muestra —sin muestra no se validan—, y uno inválido deja `perfil_muestra = NULL` con el error de `perfilar()` como motivo, legible también con la protección: un error que se repite sobre la muestra sin filas no cita ninguna celda.

Details

La función no escribe en la conexión ni crea objetos temporales del motor; `muestra_motor` puede crear un spool externo temporal de sesión, con bytes, checksum, presupuesto y estado. `DBI` es una dependencia opcional. Cada agregado no disponible queda en `NA` y su consulta, estado y motivo se conservan en `resumen_tabla$sql`. Las expresiones se ejecutan como capacidades a comprobar, no como un dialecto SQL universal.

En `muestra_motor`, `max_celdas_muestra` y `max_bytes_muestra` se aplican antes de armar la selección y se publican como filas pedidas y filas efectivas. El spool estima bytes por fila desde el catálogo cuando el motor lo permite; si la estimación no existe, comienza con un bloque pequeño y lo adapta. La materialización publica la fuente, la estimación, el recorte y el presupuesto efectivo en `meta$materializacion`. Un recorte, un muestreo corto de `TABLESAMPLE` o una falla de una consulta emiten la clase `lupa_aviso_muestra_motor`, que se puede capturar o silenciar por clase. Las columnas binarias compatibles se leen como largo y resumen MD5, por lo que se conservan faltantes, duplicados y largos sin traer el contenido a R. En SQL Server anterior a 13, o si la versión no se pudo determinar, el MD5 usa un prefijo de 8000 bytes y conserva el largo completo; esta decisión queda en la metadata de `perfil_muestra` y también convierte `image` a `varbinary(max)` antes de resumirla. Si `TABLESAMPLE` entrega menos de la mitad de las filas efectivas, se hace un solo reintento con una fracción mayor. PostgreSQL prefiere `TABLESAMPLE BERNOULLI`; en los demás motores se conserva el método por bloques y, solo si no existe, se considera el orden pseudoaleatorio sujeto a la misma guarda de costo. La metadata publica ambos métodos, la cantidad de reintentos, la guarda y cuál spool quedó elegido: una falla o una muestra más corta en el segundo intento conserva la primera muestra válida.

Las fallas de consultas solicitadas se conservan en `resumen_tabla$meta$consultas$fallas`, con `etapa`, `consulta_id`, `sqlstate`, `tipo` y `motivo`. Un `SQLSTATE 57014` se publica como `cancelada`; los rechazos tienen `tipo = "rechazada"`. Las etapas internas `clave_primaria`, `proteccion_coleccion`,

sonda_collation_dbi, sonda_fuente_bloques, sonda_desvio, estimacion_derrame, explain_moda, sonda_muestreo, sonda_mediana_escalar, sonda_mediana_cte_ventana, sonda_mediana_cte_ventana_control_neg sonda_moda_guardian, sonda_mediana_consolidada, sonda_aproximacion, sonda_dialecto, consulta, cardinalidad_catalogo, tipos_guardados, sonda_infinitos, sonda_magnitud, esquema, sonda_lectura_muestra, sonda_bytes_muestra, proteccion_fuera_de_muestra, porton_tabla y catalogo_bytes_muestra se conservan aparte en resumen_tabla$meta$consultas$sondas_fallidas y no disparan el aviso de degradacion. El aviso que informa que la muestra no esta disponible conserva ademas la clase historica lupa_muestra_dbi_no_disponible.

Cuando el catálogo estima al menos 1e8 filas y se pidió universo = "tabla_completa", la función avisa antes de los agregados con la clase lupa_aviso_costo_dbi y sugiere usar universo = "muestra_motor". El umbral se llama .UMBRAL_FILAS_AVISO_AGREGADOS_DBI y no afecta tablas menores ni la vía de muestra. La consulta de ese aviso es interna y no consume el presupuesto de max_consultas, para que el aviso no cambie qué métricas entran en la corrida.

Value

Objeto de clase perfil_dbi con dos componentes: resumen_tabla, de alcance completo o muestreado según universo, y perfil_muestra, un objeto perfil cuyo meta$origen_dbi declara tabla, conexión, SQL y alcance. perfil_muestra es NULL si la muestra no se pudo obtener o si se pidió bloque_muestra = "solo_agregados"; resumen_tabla$cobertura distingue esos casos con no_disponible y no_solicitado, respectivamente. Cuando universo = "muestra_motor" y no se solicitó el bloque, la metadata además declara meta$materializacion$estado = "no_solicitado" y meta$bloques$filas_vistas = 0: las filas no se materializaron y luego se descartaron. resumen_tabla$meta$clave conserva siempre la respuesta del catálogo de la clave primaria: columnas, fuente, motivo, garantia y estado. garantia puede ser garantizada, declarada_no_garantizada, desconocida o no_declarada; estado conserva, cuando el motor los expone, visible, restriccion_diferible, universo_incluye_descendientes e indice_no_unico. Una consulta fallida queda diferenciada de una clave no declarada. En muestra_motor, resumen_tabla$tiempos incluye materialización, lectura del spool y perfilado local; perfil_muestra$meta$bytes_muestra conserva los bytes del spool. meta$muestreo$reintentos y meta$muestreo$metodo_inicial describen la recuperación de una muestra corta. En muestra_motor o bajo bloque_filas, las columnas binarias de motores compatibles se representan como largo:md5 durante la lectura y el perfil publica sus largos originales; el contenido no se conserva.

Dos tablas se llaman cobertura

El resultado trae dos, y cubren cosas distintas. resumen_tabla$cobertura habla de **métricas SQL**: qué pidió esta función al motor y qué pasó, con bloque, elemento, estado —no_disponible, no_solicitado, degradado, presupuesto_agotado, alcance_distinto— y la consulta en sql. alcance_distinto declara que dos valores exactos incoherentes salieron de grupos de consistencia distintos: es evidencia de que la tabla cambio durante la corrida, no una acusacion contra el motor. perfil_muestra$cobertura_diagnosticos habla de **diagnósticos**: qué comprobación no se corrió sobre la muestra y por qué, con diagnostico, columna, motivo y como_resolverlo, el mismo esquema que devuelve perfilar(). Un motor que rechaza una columna aparece en la primera; una prueba estadística que no corresponde a esa columna, en la segunda. Comparten la palabra y no el vocabulario, así que conviene mirar cuál se está leyendo. Si una columna declarada como entero de 64 bits o mas ancho, o como decimal -numeric, decimal, con o sin precision-, llega como doble y contiene un valor con valor absoluto de al menos 2^53, resumen_tabla deja en no_disponible las métricas de magnitud que dependen de un valor que el doble no representa: minimo, maximo, media, n_ceros, n_negativos y desvio, por la columna. La mediana depende

sólo de sus centrales y se decide por su valor: con una cantidad impar de valores es la central, y se publica si queda por debajo de 2^53 —una central de al menos 2^53 daría un doble de al menos 2^53—; con una cantidad par promedia dos centrales y su valor no dice si alguna pasó de 2^53: se publica sólo si las dos quedan entre -2^52 y 2^52, lo que en la tabla completa cuenta una consulta más (cuántos valores llegan a 2^52 y cuántos bajan de -2^52) y en la muestra del motor no se puede saber. La misma regla corre cuando la consulta de básicos desborda (un `HUGEINT` cerca de su tope en DuckDB): la consulta se repite sin el promedio, la media queda `no_disponible` con su motivo, y el mínimo, el máximo y los conteos se juzgan como en cualquier otra columna. En una columna que no pasa de 2^53, la mediana de una cantidad par tampoco se publica en PostgreSQL si sus extremos distan más de 2^53: `PERCENTILE_CONT` interpola `bajo + (alto - bajo) * 0.5` en doble y esa resta redondea. La de un `numeric` no pasa por ahí: es la suma de sus centrales. Lo que el motor cuenta con exactitud -`n_distintos`, la moda- se sigue publicando. La moda de un entero ancho que llego como `numeric` se pide al motor como texto exacto cuando la magnitud cruza 2^53; un `BIGINT` con valores chicos conserva la moda numerica de siempre. Lo decide el valor: un decimal desde 2^53 ya no cabe en un doble -90071992547409001 y 90071992547409002 son el mismo-, y el tipo solo dice que viene de un decimal: RPostgres lo publica como `numeric` sin precision y odbc con su codigo ODBC (2 y 3), asi que una regla que esperara `numeric(20,0)` no se ejerceria. Para la muestra, la seleccion SQL trae esas columnas como texto y, si `bit64` esta instalado, las convierte a `integer64` antes de perfilar: `n_distintos`, la moda y los diagnosticos por identidad conservan los valores exactos. Los estadisticos de magnitud por encima de 2^53 siguen usando `estado_resumen_cuantitativo = "omitidos_precision"`, como en memoria, y la mediana se publica cuando sus una o dos centrales caben en un doble, como en `perfilar()`. Un decimal entero que la muestra convirtió a `integer64` se resume como tal. En SQL Server el texto de un decimal de 38 cifras se pide como `VARCHAR(48)` —`VARCHAR` a secas es `VARCHAR(30)`, y la lectura de la muestra entera abortaba— y el de `money` con sus cuatro decimales. Si falta `bit64`, la columna queda como texto y `cobertura_diagnosticos` explica la limitacion y recomienda instalarlo o pedir `integer64` al conectar. Las columnas declaradas como `double`, `real` o `float` no entran en esta guarda. Con `bloque_filas` rige la misma regla, sobre los valores que el controlador entrega como `integer64` —un `BIGINT` o `bigserial` de PostgreSQL, un `INTEGER` de SQLite que no entra en 32 bits—: por debajo de 2^53 se opera con el valor exacto, y desde 2^53 las seis que dependen de toda la columna quedan `no_disponible` con su motivo y la mediana se decide por su valor, como en el resumen SQL. Un bloque que el controlador entrega como `integer` y otro como `integer64` —RSQLite elige la clase por los valores de cada bloque— se unen como `integer64` antes de contar distintos, moda y mediana. Un decimal o un entero ancho que llega como doble desde 2^53 deja ademas `n_distintos` y `tasa_distintos` en `no_disponible`: con bloques se contarian en R sobre esos dobles, que no distinguen los valores. La moda de un decimal, y la de un entero ancho entregado como `numeric`, se le pide al motor como texto exacto en las tres vias, en PostgreSQL, SQL Server, MySQL, MariaDB y DuckDB; si la via o el adaptador no puede construir esa forma, queda `no_disponible` con su motivo. Las cifras de `perfil_muestra` de esa columna salen de los mismos dobles; sin bloques, la corroboracion avisa la diferencia en `n_distintos` y la moda contra las del motor. Un decimal por debajo de 2^53 con mas cifras de las que guarda un doble -`numeric(30,20)`- no entra en esta guarda. Ninguna columna cambia la clase de un campo de la tabla: `minimo`, `maximo`, `media`, `mediana` y `desvio` de `resumen_tabla$columnas` son siempre dobles. Cuando `incluir_valores = FALSE`, la promesa de privacidad alcanza las tres vias de salida: resumen SQL, `perfil_muestra` y `bloque_filas`. Se omiten los valores de celda -incluidos moda, extremos, medianas, centinelas y ejemplos de patrones-, se limpian la cobertura y los motivos de las metricas que publicarian un valor de celda (los de un estadistico como la media o el desvio se conservan: no contienen valores), y no se conservan familias de acumuladores ni una muestra cruda en el resul-

tado. La limpieza recorre también los atributos de cada hoja del resultado, incluidos los resúmenes de patrones. Los conteos, proporciones y agregados que no son valores de celda pueden seguir publicandose. La marca queda registrada como `meta$incluir_valores = FALSE` en ambos resultados. Si se omite el bloque con `bloque_muestra = "solo_agregados"`, la cobertura usa el estado `no_solicitado`: no es un fallo ni se cuenta como una métrica no disponible. En `muestra_motor`, la selección materializada se hace una sola vez antes de las pasadas; el spool se relee y se verifica mediante su trailer. Un trailer ausente publica `spool_incompleto:trailer_ausente`; un trailer o checksum que no coincide publica `spool_checksum_invalido`. Si la lectura materializada falla, se cierran el resultado DBI y el archivo del spool aun cuando el error deshaga una transacción; no queda una conexión de archivo abierta para un finalizador posterior.

Fallo parcial

Ningún bloque descarta al otro. Si se pide la muestra pero el motor rechaza su consulta, o si la muestra no se puede perfilar, el resultado sale igual con `perfil_muestra = NULL` y una fila en `resumen_tabla$cobertura` que declara el estado `no_disponible`, el motivo y cómo resolverlo. Si se pide sólo agregados, `perfil_muestra = NULL` se acompaña de una fila `no_solicitado`: no se intentó leer la muestra. Si el motor rechaza una columna, esa columna queda con sus métricas en `no_disponible` y las demás se perfilan enteras. Lo mismo con una columna cuyo nombre la sesión no puede escribir: odbc pasa la consulta por la codificación nativa, y bajo el locale `C` —el de un `Rscript` de cron en Linux— un nombre con tilde no se puede mandar; mandarlo dejaría la conexión rota y la consulta siguiente haría caer a R. La consulta no se envía: la columna queda en `no_disponible` con ese motivo, y una tabla con un nombre así da `lupa_error_codificacion_dbi`. Para leerlas, correr la sesión con un locale UTF-8 (`LC_ALL=C.UTF-8`). RPostgres, RSQLite y duckdb reciben la consulta en UTF-8 y leen esos nombres en cualquier locale. Una columna que la credencial no puede leer —un `GRANT SELECT` por columnas o un `DENY SELECT` sobre una columna— se sabe antes de leer, por el catálogo del motor: en PostgreSQL `pg_attribute` con `has_column_privilege()`, legibles por cualquier rol (`dbListFields()` de RPostgres no lista esas columnas), y en SQL Server `sys.columns` con `HAS_PERMS_BY_NAME()`. La columna aparece en `resumen_tabla$columnas` con sus métricas en `no_disponible` y una fila de cobertura con el motivo y el remedio, no se nombra en ninguna consulta, y las demás se perfilan enteras. SQL Server rechaza además cualquier consulta sobre la tabla, también `COUNT(*)`, cuando hay una columna sin permiso: la lectura va sobre una tabla derivada con las columnas legibles, y el SQL publicado lo muestra. Con `universo = "muestra_motor"` la columna se declara igual, en `resumen_tabla$columnas` y en la cobertura; `TABLESAMPLE` no se puede escribir sobre la tabla derivada, así que la muestra del motor usa otra forma que el motor acepte o se degrada con su motivo (limitación conocida). Con odbc las consultas se ejecutan sin preparar (`immediate = TRUE`, como el `dbGetQuery()` de odbc): preparada, una consulta rechazada por un permiso de columna dejaría la conexión rota y la siguiente haría caer a R. Lo que el catálogo no ve —un sinónimo, una tabla de otra base, una tabla temporal— se intenta leer: el rechazo del motor se declara y la conexión sigue. Con enmascaramiento dinámico de SQL Server (`MASKED WITH`, 2016+) y una credencial sin `UNMASK` sobre la columna o la base, el motor devuelve la máscara y no el dato. La completitud de esa columna se publica —los no nulos se cuentan con una expresión que el motor no enmascara—; distintos, moda, extremos, media, mediana y desvío quedan `no_disponible` con el motivo, la columna no entra en `perfil_muestra` (una fila `alcance_distinto` lo dice) y `resumen_tabla$cobertura` la declara. Con `UNMASK` se perfila entera. SQL Server tampoco admite `COUNT(DISTINCT)`, `GROUP BY`, `ORDER BY` ni `MIN`/`MAX` sobre `text`, `ntext`, `image`, `xml` y `geography`. El catálogo reconoce esos tipos antes de armar las consultas: `validos` se conserva con la expresión de nulos, mientras `n_distintos` y `moda` quedan `no_disponible` con su motivo y no se manda una sentencia que el motor rechazaría. Si

orden_muestra pide una de esas columnas, la llamada se rechaza antes de enviar el orden. En una vista SQL Server, universo = "muestra_motor" usa el límite pseudoaleatorio compatible en vez de TABLESAMPLE, que sólo acepta tablas locales. Limitación: el catálogo no resuelve el objeto al que apunta un sinónimo. Si apunta a una vista, muestra_motor prueba TABLESAMPLE, el motor lo rechaza y la vía pasa al límite pseudoaleatorio; si apunta a una tabla con text, ntext, image, xml o geography, la vía pide los distintos y la moda de esas columnas, el motor los rechaza y quedan sin medir. Con XACT_ABORT ON y una transacción abierta, en los dos casos el motor deshace la transacción, y la vía lo informa con lupa_error_transaccion_dbi. Si la lectura de la muestra falla, la columna que la hace fallar se busca por descarte leyendo las MISMAS filas que la muestra con menos columnas —un fallo que está en una fila que no es la primera, como el CAST de un texto sucio en una vista, también se encuentra—; la muestra sale con las demás y una fila alcance_distinto nombra la culpable. Si una vista falla en una fila durante universo = "muestra_motor" o bloque_filas, la vía intenta el mismo resumen SQL sobre la tabla completa: así conserva las métricas de las columnas sanas y declara la muestra o el recorrido de bloques como no_disponible, en vez de convertir todas las columnas en NA. Si el resumen completo tampoco puede ejecutarse sobre la vista, sólo se conservan las métricas que el motor alcanzó a contestar y las demás quedan no_disponible; la cobertura identifica la columna o la consulta que impidió completar el resumen. El nombre de la tabla se normaliza antes de la primera llamada a la conexión, también la que comprueba que existe: un nombre marcado latin1 —el que da read.csv(encoding = "latin1")— se convierte a UTF-8 y se lee, y uno con bytes que no son UTF-8 válido y no declaran su codificación —read.csv() sin fileEncoding sobre un archivo latin1— no se adivina: con odbc da lupa_error_codificacion_dbi antes de tocar la conexión, que así no queda rota. Una columna de orden_muestra que la sesión no puede escribir da también lupa_error_codificacion_dbi, con ese remedio: no la rechazó el motor, no se le mandó. Los **valores** de texto tienen el mismo origen. Bajo un locale que no es UTF-8, FreeTDS entrega las columnas de texto no Unicode (varchar, char, text) en latin1 y odbc las marca UTF-8 sin convertirlas; las nvarchar llegan en UTF-8. Un valor así se lee como latin1 —lo que es—, y no como texto UTF-8 roto: no se publica el hallazgo de bytes que no forman UTF-8 válido, que sería del controlador y no del dato, y los valores entran en los análisis de texto y en las comparaciones de relaciones_coleccion(). Lo que no se puede recuperar es un carácter que no está en latin1 —el signo del euro—: el controlador lo entrega como ?. Por eso cada columna de texto no Unicode leída bajo una sesión así queda en resumen_tabla$cobertura con estado degradado, el motivo y el remedio —correr con un locale UTF-8—, salvo que el controlador haya entregado en ella texto UTF-8 no ASCII (está configurado en UTF-8). Bajo una sesión UTF-8, la columna que el controlador entregue en latin1 —un ClientCharset configurado así— se lee igual y se declara igual. Los stop() de esta vía llevan clase de condición propia —todas heredan de lupa_error_dbi— para que se puedan rescatar con tryCatch(): lupa_error_conexion_dbi, lupa_error_tabla_dbi, lupa_error_campos_dbi, lupa_error_conteo_dbi, lupa_error_esquema_dbi, lupa_error_codificacion_dbi, lupa_error_argumento_dbi, lupa_error_transaccion_dbi y lupa_error_integer64. La última sale cuando la lectura por bloques recibe una columna integer64 y bit64 no está instalado: sin él R iguala entre sí todos los valores negativos. Sin bloques, los agregados que calcula el motor se devuelven y la muestra no, con un aviso que dice por qué. Alcanza bit64 4.0.5 o posterior.

Dentro de una transacción

La vía se puede correr dentro de una transacción abierta por el usuario —es lo que recomienda el motivo de consistencia para una lectura consistente—. En PostgreSQL un error cualquiera aborta la transacción, y la vía manda consultas que pueden fallar a propósito —las sondas de dialecto— o por el dato —una métrica que el motor no calcula sobre un tipo, una vista rota—: con

una transacción abierta, cada sentencia de la vía va entre un SAVEPOINT propio y su RELEASE, y si falla se vuelve con ROLLBACK TO SAVEPOINT. La transacción se reconoce por RPostgres (dbBegin(), dbWithTransaction()) o por el motor, para un BEGIN mandado a mano. El paquete nunca manda BEGIN, COMMIT ni ROLLBACK; fuera de una transacción no agrega ninguna sentencia de ese tipo. Lo mismo vale para plan_perfilado_dbi(), perfilar_coleccion() y relaciones_coleccion(). SQL Server puede deshacer la transacción completa ante un error de ejecución, incluso con XACT_ABORT apagado: antes y después de cada envío y lectura la vía mide @TRANCOUNT y XACT_STATE(). Si cambian, levanta lupa_error_transaccion_dbi y no continúa leyendo en autocommit como si la instantánea siguiera viva. Si la sesión tiene IMPLICIT_TRANSACTIONS ON, el motivo lo declara como una transacción implícita de la sesión, no como una transacción abierta por el usuario. SAVE TRANSACTION no evita los errores del motor que deshacen la transacción completa. La etapa publicada en el motivo es la que detectó el cambio: dbSendQuery, dbFetch o dbClearResult. Las sondas de sintaxis de otros motores no se ejecutan cuando el controlador ya fue reconocido; así se evita que XACT_ABORT ON aborte la transacción por una sentencia que no corresponde a ese motor. Un desbordamiento de datos que el motor no puede aislar puede abortarla igualmente y queda documentado en lupa_error_transaccion_dbi. MySQL, MariaDB y SQLite no abortan la transacción por el error de una consulta. duckdb sí la aborta por un error al preparar, como el de una sonda de dialecto— y no tiene SAVEPOINT: dentro de una transacción de duckdb, un objeto que falla al leerse —una vista que convierte un texto sucio— la deja abortada, y hay que cerrarla con ROLLBACK.

Muestra y aproximaciones

universo = "muestra_motor" sondea las formas declaradas por el adaptador y usa TABLESAMPLE cuando el motor lo acepta, o una función pseudoaleatoria con el límite del dialecto. Si ninguna forma es compatible, las métricas SQL quedan en no_disponible: no se sustituyen por resultados de la tabla completa. La estimación de bytes de SQL Server usa sys.partitions y sys.allocation_units, que no requieren VIEW DATABASE STATE para una tabla accesible al usuario. Si el catálogo no se puede leer, esa consulta interna queda en la metadata sin convertirse en una falla de consulta solicitada. Cada registro publica alcance, universo, tamano_muestra, fraccion, metodo y error_esperado. En resumen_tabla$meta$muestreo, tamano_muestra conserva el nombre historico y declara el tamano efectivo solicitado a la consulta; filas_solicitadas y filas_pedidas declaran el pedido original y filas_obtenidas las filas que devolvio el spool. La materializacion publica muestra_id, snapshot_id, orden_id, n_filas, bytes, checksum, backend, version y presupuesto en meta$materializacion; las pasadas publican el mismo muestra_id. También publica la estimación de bytes por fila del catálogo, los bytes medidos, la fracción real materializada y tope_que_mando cuando un tope recortó la copia. La estimación sólo fija el primer bloque: el tamaño medido gobierna los siguientes y la parte que entra se conserva cuando el presupuesto se agota. El contrato de perfil_muestra es campo por campo: meta$filas_analizadas es la cantidad efectivamente entregada a los diagnósticos; hallazgos solo contiene familias disponibles; y cobertura_diagnosticos tiene una fila por familia no evaluada. meta$origen_dbi$muestreo conserva filas solicitadas, entregadas, reproducibilidad, muestra_id, snapshot_id y checksum. El método de impresión remite a esa cobertura cuando el campo no está disponible; no reemplaza la ausencia con NULL silencioso. Cuando el orden de la muestra del motor no se demostró estable, cada traza de hallazgos se publica no_disponible con alcance = "orden_muestra_no_estable", sin índices ni claves y con su total; los patrones raros conservan su cantidad y retiran sus nombres. Bajo bloque_filas, perfil_muestra sigue siendo la muestra diagnóstica acotada por muestra, max_celdas_muestra y max_bytes_muestra; no es la cobertura completa. Por eso perfil_muestra$general$filas es el tamaño de esa muestra y la cobertura se lee en meta$bloques$filas_vistas. En muestra_motor,

todas las metricas SQL salvo n describen la relacion muestreada; n es el total de la tabla completa y esta marcado con alcance = "tabla_completa", metodo = "conteo_universo" y error_esperado = "no_aplica". Los conteos n_validos, n_faltantes, prop_faltantes, n_distintos, tasa_distintos, n_ceros, n_negativos y frecuencia_moda son observaciones de esa muestra, no extrapolaciones al universo. Sus motivos declaran la escala local y el denominador n_total_consulta cuando corresponde. TABLESAMPLE SYSTEM es la fuente preferida cuando el adaptador la declara y la sonda la acepta; publica metodo_muestreo = "tablesample_system", sesgo_muestreo = "por_bloques" y el motivo estable sesgo_muestreo:tablesample_system_por_bloques. Si se usa el fallback random_limit con NEWID(), la metadata publica metodo_muestreo = "random_limit", funcion_muestreo = "newid", sesgo_muestreo = "por_fila" y sesgo_muestreo:random_limit_newid_por_fila. Ese fallback se acepta solo si la politica de costo conserva n_total, la proyeccion, la tasa y sus tres umbrales; si no, la mediana queda no_disponible con capacidad_no_aceptada:newid_costo_excede_presupuesto aunque se silencien los avisos. Si TABLESAMPLE devuelve cero o menos de la mitad de las filas pedidas, se intenta una vez con una forma de muestreo más amplia. Si el reintento logra una muestra suficiente, se conservan en meta$muestreo ambos métodos y no se emite un aviso por la escasez transitoria. Si la muestra final sigue corta, se publica el aviso y las métricas de alcance quedan con el estado correspondiente. Un random_limit que devuelve cero conserva el motivo muestra_vacia:random_limit_sin_filas (o el método efectivo equivalente). Esto no permite concluir que la columna esté vacía, por lo que no se publica cero ni se dispara la cascada sin_valores. Una muestra no vacía sin valores válidos usa muestra_inestable:sin_valores_validos; una capacidad no aceptada usa capacidad_no_aceptada:.... n conserva el conteo de la tabla completa.

Con bloque_muestra = "solo_agregados" cada sentencia sobre la muestra del motor sortea la suya. Lo que decide los agregados de una columna -la cuenta de sus valores, su magnitud, la media y el desvío- sale del lote de agregados, una sola sentencia: si la magnitud pide la media o el desvío escalados (valores de más de 1e150 o de menos de 1e-150 en valor absoluto), no se recalculan en otra consulta, que sortearía otra muestra, y quedan no_disponible con su motivo. La moda, la mediana y los distintos se calculan cada uno en su sentencia, sobre su propia muestra; las cotas de la moda y de los distintos se comprueban dentro de esa sentencia (ver el consulta_id). Con "con_muestra" todo sale de la única muestra materializada en el spool.

Un estado que afirma un valor (calculado, estimado, observado_muestra) nunca se publica con el valor ausente. Si el valor falta, el registro dice por qué: sin_valores cuando la columna no tiene valores no nulos en la tabla completa (en la muestra, no_disponible con muestra_inestable:sin_valores_validos); no_aplica para una métrica numérica sobre una columna que no lo es, para el desvío con menos de dos valores y para la moda cuando ningún valor de la tabla se repite; no_disponible en cualquier otro caso, también la moda de una muestra sin repetidos, porque la tabla puede tenerla.

En una muestra, error_esperado vale no_estimado para metricas cuyo error podria calcularse bajo un plan probabilistico pero no se calculo, no_estimable para la moda, la mediana y la cardinalidad observada, que no tienen una cota simple sin supuestos o un estimador declarado, y no_aplica cuando no hubo muestreo efectivo. El motivo de cada registro explica la distincion. metodo, tamano_muestra y fraccion conservan las condiciones de la corrida; no se publica una cota numerica sin una formula justificada. Si la materialización completa falla y sólo se puede construir un perfil aislado de las columnas legibles, ese perfil queda como diagnóstico. Las métricas SQL del resumen pasan a no_disponible, con tamano_muestra y fraccion en NA, y la cobertura nombra que la muestra aislada no representa la materialización solicitada. Los distintos de una muestra se publican como cardinalidad de la muestra, no como cardinalidad del universo. estrategia_distintos es explicita y vale "exacta" por omision: calcula COUNT(DISTINCT) sobre las filas de la corrida. "aproximada_motor" sondea una funcion nativa y, si no hay una ca-

pacidad aceptada, deja la metrica no_disponible; nunca ejecuta el conteo exacto como repliegue. "catalogo" lee pg_stats.n_distinct y publica estimado_catalogo sólo cuando universo = "tabla_completa". Con universo = "muestra_motor" queda no_disponible, porque el catálogo describe la relación entera y la corrida mide un subconjunto; no se inventa una equivalencia entre universos. "omitida" no emite la consulta. Cada resultado y el atributo meta$estrategia_distintos separan estrategia_solicitada, estrategia_resuelta y estado.

Las comparaciones que tienen una cota dura usan solo valores del mismo grupo de consistencia. El grupo queda probado por el consulta_id que ya se registra en resumen_tabla$sql: dos metricas con el mismo identificador salieron de la misma sentencia. La consulta exacta de distintos trae el conteo de no nulos AS n_validos_guard junto a COUNT(DISTINCT columna). Si una capacidad aproximada sólo construye una consulta completa y no puede traer ese guardian, la cota no se comprueba y el motivo lo declara; no se atribuye un valor imposible al motor. La consulta de la moda intenta traer SUM(COUNT(*)) OVER () AS n_validos_guard junto a su frecuencia. La forma se sondea antes de usarla; si el motor la rechaza, se conserva la consulta anterior y meta$moda_guardian publica el repliegue. Cuando la sonda pasa, la cota frecuencia_moda <= n_validos se comprueba dentro de la misma sentencia y el motivo de la métrica lo declara. meta$snapshot queda en FALSE, siguiendo la declaracion de colecciones: no hubo lectura instantanea del motor. snapshot_id identifica positivamente la materializacion cliente, pero solo se publica evidencia positiva de snapshot transaccional cuando el adaptador la demuestra. La cobertura agrega una entrada concreta solo si n_validos y n_distintos son exactos, incoherentes y provienen de grupos distintos; su motivo conserva ambas sentencias. La mediana muestral que no tiene una forma consolidada o escalar, y la de cada columna que no puede usar la consolidada (un decimal, un float8 de PostgreSQL o una columna con extremos), usa una sola CTE con COUNT(*) OVER (), ROW_NUMBER() OVER y el promedio de las posiciones centrales, elegidas sin dividir; con extremos, el promedio no desborda y sigue en la misma sentencia. La sonda barata repite esa construccion sobre SELECT 1.0 UNION ALL ... SELECT 4.0, exige 2.5 y un control negativo que falle; no se sondea la tabla de produccion. Si esa capacidad falla, la mediana se publica como no_disponible con capacidad_no_aceptada:sonda_mediana_cte_ventana, nunca como dos_consultas bajo muestra_motor. Si la preparación no la probó —porque hay mediana consolidada— y una columna decimal o un float8 de PostgreSQL la van a necesitar, plan_perfilado_dbi() cuenta esa prueba y su control negativo entre los portones; la que deja la consolidada por sus extremos depende de los datos y no se anticipa.

Las cotas de error no documentadas de una aproximacion nativa quedan como "desconocido". Una aproximacion solo se consolida cuando entrega una expresion que se puede incrustar en el SELECT; si solo construye una consulta completa, se emite por separado. Una consulta no emitida o sin un valor utilizable queda no_disponible.

Motores medidos

El camino principal es PostgreSQL por RPostgres, en la 9.3 y en la 16, y SQL Server 2022 por odbc con el controlador FreeTDS, en sesiones UTF-8 y bajo el locale C de un Rscript de cron. Los siete motores —PostgreSQL 9.3 y 16, SQL Server 2022, MySQL y MariaDB (RMariaDB), SQLite (RSQLite) y duckdb— se verifican contra el motor real con el guion benchmark/verificar_motor.R, y SQLite y duckdb se ejercitan además en las pruebas del paquete. Con otro motor, otro controlador u otra versión la función corre por la vía portable (ver «Dialecto») y declara en la cobertura lo que no pudo medir, pero sus cifras no están comprobadas contra ese motor. Las configuraciones avanzadas —una transacción abierta por el usuario, permisos por columna, enmascaramiento dinámico, universo = "muestra_motor", bloque_filas— tienen sus limitaciones conocidas en «Fallo par-

cial» y «Dentro de una transacción».

Dialecto

Acotar filas no es SQL estándar. En vez de suponer `LIMIT`, la función sondea el motor con una consulta de cero filas y elige la primera capacidad que acepte: `LIMIT n [OFFSET k]`, `TOP (n)` y `OFFSET ... FETCH NEXT`, `FETCH FIRST ... ROWS ONLY`, o `ROWNUM`. Si ninguna sirve queda la vía portable, que acota en el cliente con `dbSendQuery()` y `dbFetch(n)`; en ese caso la mediana exacta se declara no disponible en vez de traer media tabla a memoria. El alias de subconsulta se escribe con `AS` o sin él según el motor, y los alias de columna van comillados y se comparan sin distinguir caja, porque hay motores que los pliegan a mayúsculas. Para los motores del dialecto `limit`, la mediana exacta usa una sola sentencia: el `COUNT` queda como subconsulta escalar de la consulta que ordena y recorta. La forma se sondea antes de usarla; en SQLite y PostgreSQL se usan `%` y `/` con division entera. Bajo `muestra_motor`, los dialectos sin una forma escalar ni consolidada usan la CTE de ventanas descrita arriba y no degradan a `dos_consultas`. En `tabla_completa` se conserva el camino previo, incluido `dos_consultas` si es el que resuelve ese dialecto; `PERCENTILE_CONT` no cambia. `dos_consultas` necesita el conteo de valores no nulos para el salto: si `metricas` no incluye `"validos"`, la mediana lo cuenta en una consulta propia y sale igual que con todas las métricas. La mediana consolidada tambien se sondea antes de usarla; en SQL Server requiere nivel de compatibilidad >= 110. Se sondea; estos son los motivos conocidos de que no se active: la funcion no esta disponible, el motor rechaza la sonda o devuelve el percentil sin todas sus cifras. La prueba lee una fila con un valor de más de 1e8 y otro de menos de 1e-10 y exige los dos idénticos: MariaDB tipa `PERCENTILE_CONT` como `double(18,10)`, que satura en 1e8 y redondea a diez decimales, y por eso usa la mediana por columna como MySQL. En ese caso el mensaje del motor queda en `meta$mediana_consolidada$motivo` —sin sus literales con la protección activa, entero sin ella, como todo mensaje del motor: ver «Datos personales»—, y se conserva la mediana por columna. La mediana de un decimal no entra en la consolidada: en MySQL, MariaDB, SQL Server, PostgreSQL y DuckDB se piden, en el tipo de la columna, la suma de sus una o dos filas centrales y cuántas son, y R divide: es el doble más cercano de la mediana exacta, por la tabla completa y por la muestra del motor. Cuesta una consulta por columna decimal, que el plan cuenta. Con `bloque_filas` y en el resumen de la muestra del motor con `bloque_muestra = "con_muestra"` (el spool), la mediana de un decimal que el controlador entrega como doble también es la del decimal: por bloques, con una cantidad impar la central es un valor y su doble es el más cercano, y con dos centrales que caen en el mismo doble la mediana cae en él; con dos centrales distintas se le piden al motor como texto, con el mismo salto que la vía de dos consultas —una consulta ordenada por columna—, y R las suma. `money` de SQL Server se pide siempre, porque odbc no lo lee al doble más cercano. El spool ya trae cada decimal como texto y la calcula de ahí. `PERCENTILE_CONT` interpola en doble, y la de un decimal quedaba a un ulp del doble más cercano (la de 0,10 y 0,20 daba 0,15000000000000002, el promedio de los dobles); en DuckDB, además, interpola un `DECIMAL` en su propia escala y trunca, y convierte a doble un decimal de más de 18 cifras sin dar el más cercano: ahí la suma se lee por su texto. `AVG(x * 1.0)` necesita una cifra de escala más que la columna y en la escala máxima del motor —30 en MySQL, 38 en MariaDB— se recortaba (la mediana de 1e-30 y 2e-30 salía 2e-30). La mediana consolidada de las demás columnas numéricas interpola en doble y puede quedar a una unidad de la última cifra del doble más cercano.

Lo que el motor guarda distinto que R

El resumen SQL describe lo que el motor tiene delante, y en estos casos eso no es lo mismo que la tabla en R:

- **Fechas en SQLite.** SQLite no tiene tipo fecha: RSQLite guarda un `Date` como número de días desde 1970 y un `POSIXct` como segundos, y sin más información el motor los describe como números —`minimo = 17956`—. Con `DBI::dbConnect(RSQLite::SQLite(), ..., extended_types = TRUE)` la columna se declara temporal y `perfilar_dbi()` no le aplica agregados cuantitativos, y lo dice.
- `NaN` **en duckdb.** duckdb conserva `NaN` como un valor: lo cuenta como presente y distinto, y responde `MAX` y `AVG` con `NaN`. En R, `NaN` cuenta como ausente. Los conteos pueden diferir por eso, y una métrica que el motor respondió con `NaN` no se publica: queda `no_disponible` con su motivo.
- **Infinitos.** SQLite, PostgreSQL y duckdb admiten `Inf` y `-Inf` en una columna de punto flotante, y el motor los cuenta en `MIN`, `MAX`, `AVG` y el orden de la mediana. Los estadísticos de magnitud se calculan sólo con valores finitos, como en `perfilar()` y con `bloque_filas`: cuando el motor responde un infinito, `minimo`, `maximo`, `media`, `mediana`, `desvio`, `n_ceros` y `n_negativos` se vuelven a pedir sobre una subconsulta que deja afuera los infinitos, y su `sql` lo muestra. Donde la consulta no sabe escribir el infinito, o sobre una muestra del motor, esas métricas quedan `no_disponible` con ese motivo. Un infinito es un número: no se informa como un valor que no se pudo leer como número.
- `money` **y** `real` **en PostgreSQL.** RPostgres lee `0` en toda celda `money`, sin avisar, y un `real` por el texto con que lo escribe el servidor, mientras el motor opera sobre el binario guardado —21.299999237060547—. Las dos columnas se leen convertidas en el motor: `money` como `CAST(x AS NUMERIC)` y `real` por la forma decimal más corta que vuelve al valor guardado —`21.3`, la cifra que lee R desde PostgreSQL 12—, que se calcula en el motor redondeando a 6, 7, 8 y 9 cifras. No se toma del texto de la sesión: antes de PostgreSQL 12, o con `extra_float_digits` de 0 o menos en cualquier versión, ese texto tiene seis cifras y 1234501 se escribe `1.2345e+06` —200 valores distintos se leerían como 21, con un mínimo que no está en la columna—. Así la lectura da el valor guardado en cualquier versión y con cualquier `extra_float_digits` desde -6, sin tocar el estado de la sesión; en un empate de la última cifra puede elegir la otra forma corta, que vuelve al mismo `real`. Por debajo de -6 el servidor escribe todo doble con menos de nueve cifras, y ninguna lectura —tampoco la de un `double precision`— llega entera a R. La conversión vale para toda lectura —la muestra, `bloque_filas`, `muestra_motor`— y el resumen SQL calcula sus agregados sobre esa misma lectura, así que las vías publican la misma cifra. Las columnas convertidas y su expresión quedan en `resumen_tabla$meta$lectura_convertida`. Se reconocen por el tipo que publica `DBI::dbColumnInfo()` de RPostgres; con otro controlador no se convierte nada. Para `double precision`, PostgreSQL 12 o posterior con `extra_float_digits >= 1` conserva la lectura nativa: el servidor ya escribe una representación redonda de 17 cifras y no se agrega una reconstrucción que pueda cambiar el valor ni encarecer cada consulta. En PostgreSQL 9.3, en versiones anteriores a la 12 o con `extra_float_digits <= 0`, la lectura usa `float8send()` para decodificar los ocho bytes IEEE-754 y formar el valor exacto como `NUMERIC`; esto conserva también los dobles vecinos bajo una potencia de dos, los valores desde `2^53` y los extremos finitos. `NaN` e infinitos se representan por sentinelas numéricos que RPostgres vuelve a leer como sus valores originales. La expresión de lectura se mantiene sin alias para poder entrar en una consulta derivada; sólo la lista de proyección le agrega el alias original. Esa misma distinción se usa en el resumen SQL, la muestra, los bloques, las expresiones

anidadas y las relaciones de una colección. Los agregados de esa misma lectura excluyen `NaN` e infinitos con sentinelas de `NUMERIC`; la media se escala en `NUMERIC` para no desbordar ni perder subnormales por underflow. La lectura nativa de PostgreSQL 12+ con 17 cifras no pasa por esa reconstrucción. En PostgreSQL 9.3 y en sesiones con pocas cifras la decodificación exacta es más costosa que la lectura nativa; se conserva ese costo para no publicar cifras falsas y se aplica sólo cuando la sesión no garantiza los 17 dígitos. Cuando la media o el desvío de esa lectura vuelven por una sesión de PostgreSQL 9.3 o con `extra_float_digits <= 0`, el resultado se decodifica por sus ocho bytes antes de publicarse. Así no queda recortado a 15 cifras por el transporte; la aritmética del motor todavía puede diferir de R en unos ulps, y esa diferencia se conserva como tal en la corroboración.

`datetimeoffset` **en SQL Server.** FreeTDS puede entregar a `odbc` un `datetimeoffset` con desplazamiento distinto de `+00:00` ya convertido a UTC. Una vez perdido el desplazamiento, Lupa no puede reconstruirlo desde el valor recibido: la corroboración lo deja como divergencia de una fecha, no como diferencia de cotejamiento. Para conservar el desplazamiento hay que leer la columna como texto o normalizarla en la consulta de origen.

- **Extremos y equivalencia entre motores.** SQL Server puede desbordar al agregar un `float` que contiene `-DBL_MAX` o `+DBL_MAX`; en ese caso las métricas que desbordan quedan `no_disponible` con el mensaje del motor, aunque la muestra conserve el valor, y el desvío se recalcula escalado (ver «Magnitudes extremas»). El mismo `real` puede mostrar una última cifra distinta entre PostgreSQL y SQL Server porque RPostgres lee la forma decimal corta y FreeTDS/odbc entrega el `float32`; cada resumen es coherente con los valores que su controlador devolvió, pero la comparación entre motores puede declararlo `materialmente_distinto`. En un `real`, un centinela fraccionario se alinea a `float32`; si el valor declarado tiene más precisión que la representación binaria almacenada, no se considera el mismo centinela y no se cuentan faltantes disfrazados. La lista explícita de centinelas conserva además la reserva documentada de los valores por omisión: declarar un valor lo vuelve a contar incluso si aparece una sola vez. En PostgreSQL anterior a la 12, o con `extra_float_digits <= 0`, una combinación de `+DBL_MAX` o `-DBL_MAX` con infinitos se calcula sobre los finitos que sobreviven la limpieza. En PostgreSQL 12 o posterior con la sesión por omisión, esa misma combinación hace desbordar el motor y el mínimo, el máximo, la media y el desvío quedan `no_disponible`, con el error del motor como motivo: se declara, no se calcula. Si una columna que conserva ambos extremos finitos hace desbordar la suma de cuadrados, el desvío se recalcula escalado.
- **Magnitudes extremas.** Cuando el mínimo y el máximo ya medidos dejan el mayor valor absoluto fuera de `[1e-150, 1e150]` —donde los cuadrados desbordan o caen en subnormales y pierden cifras—, el desvío se le pide al motor sobre `x / 2^k`, centrado en la media de esos mismos valores, y se multiplica por `2^k`: escalar por una potencia de dos es exacto, y no cuesta una consulta para decidirlo. También se escala cuando el rango —el máximo menos el mínimo— es tan chico que sus cuadrados caerían en subnormales, y la magnitud se conoce aunque no se pidan los básicos ni se publiquen los valores. En MySQL, MariaDB y SQL Server la media de esas columnas sale igual, `AVG(x / 2^k) * 2^k`, porque la suma de `AVG()` desborda; la mediana de dos centrales enormes se promedia como `a / 2 + b / 2`; y por encima de `1e65` el centro va como doble, porque un `DECIMAL` tiene 65 cifras. Una columna constante da `0`. Si el motor no puede, la métrica queda `no_disponible` con su falla en `meta$consultas$fallas` y el aviso: no se publica `Inf` ni un `0` sobre valores distintos. `perfilar()` y `perfilar_dbi(..., bloque_filas = n)` suman los cuadrados sin error en una unidad que es potencia de dos —la que deja el mayor valor en 2^400— y que sube cuando llega un bloque que no cubre: el desvío es el doble más cercano también en esas magnitudes. Las columnas de magnitud común no cambian de consulta ni de cifra.

- **La media en coma flotante.** Cada lado suma a su modo; sobre una columna que mezcla magnitudes muy distintas —`1e20` junto a `1`— las medias pueden diferir entre motores y con la de R. La media y el desvío de R —`perfilar()`, `perfil_muestra` y `bloque_filas`— son los dobles más cercanos de la media y el desvío exactos de los dobles que leyó. La media y el desvío de un decimal que el controlador entrega como doble son los del decimal, los dobles más cercanos de los exactos, por la tabla completa, por `bloque_filas` y por la muestra del motor, con `bloque_muestra = "con_muestra"` y con `"solo_agregados"`. El motor suma en el tipo de la columna, y devuelve como texto, la columna y los cuadrados de sus diferencias con la media escrita en su escala —o de los valores, donde no hay media o las diferencias pasan de 38 cifras—, con expresiones elegidas para que no desborden ni redondeen con las cifras del tipo, las de los extremos y las filas: en SQL Server y duckdb la parte entera y la fracción por separado, porque la suma de un `decimal(38,30)` desborda; en duckdb el cuadrado con un factor de 19 cifras, que lo lleva a 128 bits —el de dos `DECIMAL(15,0)` desbordaba en 64—; y donde nada de eso entra, en trozos de nueve cifras. R divide y saca la raíz con aritmética entera, sin cancelación. La muestra leída en R hace lo mismo con el texto de cada valor. Por la tabla completa y bajo la muestra del motor las sumas van en la misma consulta que los demás agregados, sin consultas de más y, bajo la muestra, sobre las mismas filas; por bloques, en lotes de hasta `tamano_lote_planos` columnas, con las expresiones que acepta el motor (1.664 en PostgreSQL). Un lote que el motor rechaza se parte hasta aislar la columna, y las demás siguen. Entran con cualquier `metricas`, con `incluir_valores = FALSE` —la media es un literal que sale de los datos, y el SQL publicado la lleva tapada— y con valores de `2^53` o más; en PostgreSQL, `NaN` e `Infinity` quedan fuera, como en la media de la tabla completa. En duckdb la media y el desvío del motor de esas columnas se piden sobre el doble: `x * 1.0` de un `DECIMAL(38,30)` desborda. Las sumas reemplazan la media y el desvío del motor, que los calculaba en doble: en MySQL, MariaDB y SQL Server promediaba el doble de cada valor, y con valores que se cancelan la media salía lejos (`999999999999999.99`, `-999999999999999.98`, `0.01`, `0.02`, `0.03` y `-0.04` daban 0,00333 por 0,005); el desvío, y la media con escala 30, quedaban a un ulp. Si las sumas no llegan —el presupuesto, o el motor rechaza el lote aun con la columna sola—, por la tabla completa la del motor queda sólo si su error cabe en una parte en mil millones (el desvío, con la guarda de los dobles; la media, si `n` redondeos del mayor valor no pasan de esa parte de la media), y si no queda `no_disponible` con el motivo; por bloques, la media y el desvío quedan `no_disponible`, como si la suma del motor no es la de lo que leyeron los bloques. Sin escala declarada —un decimal que el catálogo no describe— el desvío por bloques sale de los dobles, con su guarda; la de una tabla temporal la leen el catálogo de `tempdb` en SQL Server y `SHOW COLUMNS` en MySQL y MariaDB. La media y el desvío de una columna de coma flotante o entera que calcula el motor salen de su aritmética y no son, en general, el doble más cercano del exacto: medido sobre columnas de cuatro a nueve valores en los siete motores, por la tabla completa, la media quedó a uno o dos ulps en hasta 19 de 112 columnas (SQL Server) y el desvío a un ulp en 6 a 11 de cada 21 a 112; con más filas la diferencia puede crecer, y con valores que se cancelan, más: `1e20`, `1`, `-1e20` y `2` dan 0,5 por 0,75 en MySQL, MariaDB, SQL Server y duckdb. La corroboración cruzada la tolera hasta `1e-10` relativa y la declara cuando pasa de ahí. La media y el desvío se le piden al motor como doble donde su aritmética decimal los recortaría: en SQL Server `AVG(CAST(x AS FLOAT))`, porque `AVG` de un `numeric` devuelve seis decimales, y en PostgreSQL `STDDEV_SAMP(CAST(x AS DOUBLE PRECISION))`, porque sobre magnitudes grandes el desvío de un `numeric` sale con cuatro; en MySQL y MariaDB `AVG(x * 1e0)`, porque `AVG` de un `DECIMAL` devuelve cinco decimales. En los demás motores, `x * 1.0`.

- **El desvío de una sola pasada.** `STDEV` de SQL Server —y, con menos pérdida, el desvío nativo de SQLite, duckdb, MariaDB y MySQL— acumula en una sola pasada, y cuando la media es muy grande frente al ancho de la columna pierde las cifras del desvío: sobre marcas de tiempo en milisegundos de una ventana de un segundo, SQL Server responde `0`. El desvío no cambia si a cada valor se le resta una constante, así que se le pide al motor sobre los valores centrados en la media ya calculada: siempre en la consulta por columna, y en la consulta compartida —donde la media sale de la misma sentencia— se vuelve a pedir centrado cuando la media supera diez veces el desvío, o cuando respondió `0` sobre una columna con valores distintos, con una consulta más que el plan no anticipa porque depende de los datos. Su fila de `sql` lo dice. La resta se hace en un tipo que no pierde cifras: en SQL Server y duckdb una columna `real` (`FLOAT` en duckdb) restaría en 24 bits —en T-SQL `real - decimal` es `real`— y el desvío centrado se apartaría del de los valores guardados, así que se convierte a doble antes de restar; en SQL Server el centro de una columna de coma flotante va como literal `float`, que no tiene el tope de 38 cifras de un decimal. Una columna decimal o entera resta en su tipo, que es exacto; en SQL Server, si la resta pasa de 38 cifras el motor recorta la escala y redondea, y un empate se redondearía hacia lados distintos según el signo, así que el centro de un decimal nunca termina en 5. Con `bloque_filas` los momentos se acumulan centrados, sobre los dobles que entrega el controlador. El desvío de un decimal sale de las sumas exactas del motor (ver «La media en coma flotante»): la resta con la media escrita en la escala de la columna es exacta (medido en SQL Server, con compatibilidad 160 y 100, y en MySQL hasta `decimal(38,30)` y `DECIMAL(65,30)`). Sin escala declarada en SQL Server, MySQL y MariaDB, el de los dobles: si es tan grande frente a su desvío que el redondeo de cada valor a doble puede moverlo más de una parte en mil millones (`max |x| / desvio * 2^-53`) queda `no_disponible`, salvo que el tipo declare escala 0 —`numeric(18,0)`, `decimal(38,0)`— y los valores queden por debajo de 2^53: ahí los dobles son los valores. La escala la publican PostgreSQL, MySQL, MariaDB, DuckDB y el catálogo de SQL Server, también la de una tabla temporal. Bajo `muestra_motor`, la media y el desvío se piden en la misma consulta, sobre las mismas filas de una sola evaluación de la muestra: una relación derivada calcula la media de esas filas con una ventana (`AVG(x) OVER ()`) y el desvío se pide centrado en ella; esa misma evaluación da las filas obtenidas y decide el reintento. En SQLite, que suma los cuadrados en la consulta, se resta además `n * (media - centro)^2` (las dos pasadas corregidas), porque la media de la ventana no es la exacta. En SQL Server, MySQL, MariaDB y duckdb esa media y la resta son dobles; la media y el desvío de un decimal salen de sus sumas exactas, pedidas en la misma consulta (ver «La media en coma flotante»). Con `bloque_muestra = "con_muestra"` el resumen de un decimal sale del texto que la muestra trae de cada valor: el desvío es el doble más cercano del exacto, de las sumas sin redondeo de las diferencias con una de sus centrales y de sus cuadrados. El de `perfil_muestra` lo calcula `perfilar()` sobre lo leído, que para un decimal es el doble de su texto, y rige la regla de los dobles en todas las vías, también por la tabla completa, donde el de `resumen_tabla` es el exacto: no se publica (queda `NA`) y la columna queda en `perfil_muestra$meta$desvio_decimal_doble`. No se fija una semilla ni se toca el generador de números aleatorios de R. Como red en todas las vías, un desvío `0` sobre una columna con dos valores distintos —un mínimo menor que el máximo— es imposible y queda `no_disponible` con su motivo.

- **El cotejamiento.** El resumen SQL agrupa y ordena con el cotejamiento de la columna: sobre una `COLLATE NOCASE` de SQLite o una colación ICU no determinista de PostgreSQL, "Montevideo" y "montevideo" son un valor, y `n_distintos`, la moda y su frecuencia salen de esa cuenta. Con `bloque_filas` las filas se leen y se comparan en R, byte a byte, como el perfil de muestra. Cada `resumen_tabla` lo dice en `meta$criterio_comparacion` y `meta$criterio_moda`.

La corroboración no llama divergencia a la diferencia que ese criterio explica —menos distintos en el resumen SQL, con la moda y la frecuencia que de ahí se siguen, o una moda empatada que cada lado desempata con su orden—: la deja en una fila `no_corroborada` con ese motivo. En SQL Server, el resumen SQL aplica la igualdad de texto que ignora los espacios finales; la vía por bloques compara en R los valores que recibió y, por eso, puede contar `"A1"` y `"A1 "` como distintos en `n_distintos`. Esa diferencia de alcance se publica como criterio de comparación y no se corrige inventando una equivalencia en R.

- `datetimeoffset` **en SQL Server.** FreeTDS/odbc puede entregar un `datetimeoffset` con desplazamiento distinto de `+00:00` como `POSIXct` en UTC y perder el desplazamiento original. La lectura no lo puede recuperar después; la corroboración no lo excusa como un cotejamiento y deja la divergencia publicada. Para conservar el instante y el desplazamiento, leer la columna como texto con el desplazamiento o normalizarla en el origen antes de perfilar.
- **Una columna que mezcla tipos de almacenamiento.** SQLite admite texto en una columna de afinidad numérica, y RSQLite la lee como número: convierte ese texto en `0` y lo avisa ("mixed type"). El resumen SQL ve los valores guardados —la moda puede ser el texto, y las métricas de magnitud quedan `no_disponible` porque el motor devolvió algo que no se lee como número—, y `perfil_muestra` ve lo que entregó el controlador: `minimo = 0`, una moda `0` que nadie escribió. `perfilar_dbi()` no relee la columna como texto: cuenta con `typeof()` los valores guardados como texto o blob y declara cada divergencia con esa causa medida en `resumen_tabla$cobertura`, y dice que las cifras de magnitud de `perfil_muestra` para esa columna no describen lo guardado. Para perfilar los valores guardados, leer la columna como texto (`CAST(columna AS TEXT)` en una vista) o limpiarla en el origen. **Limitación conocida:** un texto que R sí lee como número —`'0x10'`, hexadecimal, que `as.numeric()` lee 16— pasa la sonda de magnitud como un número, y el resumen SQL publica `maximo = 16` sobre una columna `NUMERIC` cuyo máximo numérico es otro; las divergencias de la corroboración son la señal.
- **Lo que la corroboración compara.** Sólo las métricas que los dos bloques calculan con la misma semántica. No compara lo que una declaración o una guarda hacen distinto a propósito, porque esa diferencia no es del motor, del controlador ni de un snapshot: la `media`, la `mediana` y el `desvio` de una columna de `columnas_codigo`, que no se calculan; `n_faltantes` de una columna de `columnas_opcionales` o de `aplicabilidad`, que la muestra cuenta sobre su universo —y todas las métricas de la columna con `aplicabilidad` si la muestra dejó afuera valores presentes—; las métricas de magnitud de una columna de la que la muestra excluyó los `sentinelas_numericos` declarados; las de un entero ancho desde 2^53, que el resumen no publica; y el `desvio` de un decimal que la muestra leyó como doble y no publicó (causa `decimal_doble`, ver `perfil_muestra$meta$desvio_decimal_doble`). Tampoco lo que un bloque no calculó: una métrica que `resumen_tabla$sql` declara `no_disponible` —el motor rechazó la consulta sobre el tipo (`json`, `xml` o `point` de PostgreSQL; `text`, `ntext`, `image` o `geography` de SQL Server), una guarda la retuvo—, la moda que la muestra no pudo contar —una columna de blobs— y las métricas de una columna compuesta, que la muestra no analiza: su motivo ya está publicado, y una relectura daría lo mismo. Un `NaN` o un infinito sí se compara, y un valor guardado como texto en una columna numérica también: es la diferencia del motor que la corroboración existe para declarar. Lo que no se comparó queda en `resumen_tabla$meta$corroboracion_bloques$no_comparadas`, una fila por columna y métrica con su causa. La tolerancia está en `meta$corroboracion_bloques$tolerancia`: `1e-10` absoluta más `64` épsilon de máquina relativos para los valores leídos —`minimo`, `maximo`, `mediana`, la moda—, y `1e-10` relativa para `media` y `desvio`, que cada lado acumula con su algoritmo y en su orden. Medido en SQL Server, la diferencia de la acumulación llega a 2e-

13; una divergencia real es de otro orden y sigue saliendo. Limitación conocida: la tolerancia absoluta se suma siempre, tambien a `desvio`; por eso puede ocultar diferencias en columnas cuyos valores no son pequeños pero cuyo desvio es menor que `1e-4`. En valores menores que `1e-10` —p-valores, concentraciones en unidades del sistema internacional— la corroboración tampoco distingue dos lecturas.

Costo

Los agregados de una tabla ancha se emiten por lotes; `muestra` acota lo que se trae a R, no el trabajo del motor. `bloque_muestra` decide si se trae esa muestra; `universo`, `metricas`, `tamano_lote_planos`, `tamano_lote_distintos` y `max_consultas` acotan el trabajo SQL. `plan_perfilado_dbi()` dice cuántas consultas se van a emitir antes de emitirlas. El orden de degradación es agregados planos, total del universo cuando hace falta, distintos, moda y mediana. Los agregados planos sobre la misma tabla y filtro —el conteo de no nulos, mínimos, máximos, medias, ceros, negativos y desvío— comparten una consulta por lote y cada consulta que trae `n_validos` lleva además `COUNT(*) AS n_total_consulta` en la misma sentencia. La completitud usa ese denominador local, no el total de otro lote. Los no nulos se cuentan en PostgreSQL con `COUNT(col)`, para no contar como faltante un tipo compuesto cuyos campos sean todos nulos. En SQL Server y en los motores que rechazan `COUNT()` sobre `text`, `ntext` e `image` se usa una expresión CASE; la pregunta por el nulo se contesta sin publicar el valor enmascarado. La fusión conserva la medición, no una identidad bit a bit entre agrupamientos: la media y el desvío pueden diferir en el último bit según cómo se agrupen las sumas, porque la suma en punto flotante no es asociativa.

El total del universo se conserva por separado cuando el perfil se calcula sobre una muestra. Si el lote completo es rechazado, sus mitades se sondean por bisección: los grupos aceptados se reutilizan como mediciones y las columnas culpables se reintentan por métrica, con su denominador local. Las fuentes `TABLESAMPLE` que necesitan el total del universo para escribir un porcentaje lo cuentan antes. `COUNT(DISTINCT ...)` queda en una clase separada y usa su propio tamaño de lote, conservador por omisión porque una cardinalidad puede derramar mucho más que veinte agregados planos; la consulta exacta trae su `n_validos_guard` compañero. Cuando se pide `mediana` o `desvio` sin pedir `basicos`, cada columna que el esquema expone como numérica paga además una sonda propia de magnitud. Usa `MIN` para comprobar que el motor no está coaccionando texto a cero; la fila `sonda_magnitud` queda declarada en `resumen_tabla$sql`, con su estado y la etapa `sonda_magnitud`. Si la sonda no entra en `max_consultas`, las métricas dependientes quedan `no_disponible` con el motivo del presupuesto. Cuando `incluir_valores = FALSE`, el motivo no publica el valor que devolvió el motor. Bajo `universo = "muestra_motor"` con `bloque_muestra = "solo_agregados"`, si se pide también `validos` o `desvio`, la sonda no es una consulta aparte: va en el lote de agregados, la misma sentencia y la misma muestra que el desvío, y su fila `sonda_magnitud` comparte el `consulta_id` del lote. La proyección temporal no usa esos agregados planos: si hay más de un lote y `instrumentar = TRUE`, se mide el primer lote de distintos y, después de ejecutarlo, se multiplica su mediana por la cantidad total de lotes. El aviso llega antes del segundo lote, en la unidad que se va a evitar; con un solo lote no hay nada que proyectar. Si la duración no se pudo medir, el resultado declara la proyección como no disponible. La moda tiene otro canal: después de cada moda medida se obtiene una tasa en ms por distinto y se usa para proyectar las modas pendientes. La cardinalidad se toma del agregado de la corrida, de una clave garantizada o de la estimación de catálogo que esté disponible; si falta, `meta$costo_moda` lo declara y no inventa un número. El aviso llega antes de la siguiente moda. La mediana se proyecta en ms por fila. La primera mediana medida en esta corrida sirve para proyectar las restantes y el aviso precede a ese trabajo. Si no existe una primera medición local para una mediana total,

usa la referencia declarada de otra corrida de 68 ms por millón de filas. Si la consulta inicial que obtuvo las filas fue medida y resulta una cota mayor, se publica también esa cota de lectura —no como medición de mediana— para no subestimar una tabla grande recién cargada. Las dos proyecciones quedan separadas en `meta$costo_moda` y `meta$costo_mediana`; apagar el aviso no apaga su medición ni su metadata. Antes de la primera consulta exacta se estima, cuando PostgreSQL expone `pg_stats`, el tamaño de los hashes con `n_distinct`, `avg_width` y `pg_class.reltuples`; `SHOW work_mem` y, desde PostgreSQL 13, `SHOW hash_mem_multiplier` dan el límite efectivo. `meta$estimacion_derrame` y `attr(meta$plan, "estimacion_derrame")` conservan el diagnóstico, siempre rotulado como estimación y nunca como derrame medido. Si supera el límite se avisa antes de pagar `COUNT(DISTINCT)` y se recomienda subir `work_mem` en la sesión; el paquete no lo modifica. Una estadística ausente, un permiso insuficiente o una versión sin el parámetro dejan la parte correspondiente como no disponible. `n_distinct` es una estimación de muestra y puede quedar corta; si luego `pg_stat_statements` mide un derrame, esa medición manda. La misma corrida publica `meta$estimacion_derrame_moda` y `meta$estimacion_derrame_mediana`. La moda deriva `metodo` del plan exacto mediante `EXPLAIN (FORMAT JSON, COSTS OFF)` sin `ANALYZE`; la mediana usa siempre un sort contra `work_mem`, con pisos de 32 bytes para tipos fijos y 42 para `numeric`. El plan usa `n_validos` de catálogo. En la meta y en el aviso, cuando se pidió la familia `validos`, ese denominador se reemplaza por el medido en los agregados planos y se conserva también el valor de catálogo; si no se midió, el motivo declara que se retuvo el catálogo. Ese reemplazo alcanza solo al denominador: si la estimación publicada marca `supera_memoria = TRUE`, el aviso se emite también con catálogo y declara su fuente. La consolidada decide por el máximo de columna y publica la suma de tapes sólo como `estado_io_total_bytes` informativo.

Los avisos de esta vía son deliberadamente distintos de los de `perfilar()`: `perfilar_dbi()` los emite también en guiones no interactivos porque el costo relevante ocurre en el servidor y puede consumir decenas de segundos antes de que el llamador pueda hacer algo. En cambio, el aviso de tabla ancha de `perfilar()` estima trabajo local sobre una tabla ya en R y queda limitado a sesiones interactivas para no convertir la salida de un guion en ruido. Cada aviso DBI tiene su propio interruptor y umbral porque sus unidades no son comparables —segundos frente a bytes— y silenciar uno no debe ocultar el otro. Apagar un aviso no apaga ninguna medición: `meta$costo_distintos`, `meta$costo_moda`, `meta$costo_mediana`, `meta$derrame` y `meta$estimacion_derrame` se publican igual. Lo que no entra en el presupuesto queda en `no_disponible` con su motivo, nunca en cero. `meta$tamano_lote_funciono` conserva el mayor lote aceptado durante esa corrida; no se guarda estado global asociado a la conexión.

Instrumentación

`resumen_tabla$sql` conserva una fila por métrica y agrega la duración de la consulta que la respalda, las filas devueltas y los bytes que ese resultado ocupa en R. `consulta_id` identifica el intento dentro de la corrida y `duracion_ms` es la duración de la consulta, repetida en cada fila que esa consulta produjo; no es una duración por métrica. La columna `nivel` marca qué filas se pueden sumar: la primera fila de cada `consulta_id` queda en `nivel = 1` y sus repeticiones en `nivel = 2`. Por eso una suma segura usa sólo `duracion_ms[nivel == 1]` (con `na.rm = TRUE` si corresponde), no la columna completa. `id_consulta` identifica la consulta de datos que produjo la medición: dos métricas con el mismo identificador vieron exactamente las mismas filas y se pueden comparar directamente. `NA` declara que esa garantía no se puede hacer; en particular, las métricas por columna —moda, frecuencia de la moda y mediana— no comparten filas con otras métricas. `etapa` permite agruparlo (`conteos`, `moda`, `basicos`, `mediana`, `desvio`, `lectura_muestra` y las sondas). Las métricas no solicitadas o que no emitieron consulta conservan esos campos y los dejan en `NA`; en

particular, NA no significa cero.

resumen_tablasqlmemoria_trabajo agrega el eje del estado de trabajo del motor: se deriva en orden como NA para una fila sin medición, acotado para una muestra con fraccion < 1 y, en los demás alcances efectivos, según el método resuelto del registro; una muestra saturada (fraccion = 1) cae a este último caso porque su tope es vacuo y se clasifica como tabla completa. Sus únicos valores son "creciente", "acotado" y NA.

En PostgreSQL, con instrumentar = TRUE, se toma una foto de pg_stat_statements antes y después de la ventana que cubre los agregados exactos instrumentables de distintos, moda y mediana. Las firmas se filtran por texto y se normalizan en el paquete: cada literal numérico o de cadena se reemplaza por su parámetro posicional y los identificadores entre comillas dobles se conservan. Sólo se publica un derrame cuando una consulta coincide y su contador aumentó en al menos una llamada dentro de la ventana. En ese caso, resumen_tabla$sql agrega derrame, bloques_temporales_leidos, bloques_temporales_escritos y fuente_derrame, además de llamadas_en_ventana; resumen_tabla$meta$derrame conserva el resumen y la fuente. Cuando el contador aumentó más de una vez, el motivo declara que los bloques son un agregado de ese texto exacto y pueden incluir otra sesión o llamada concurrente. Si la extensión no está disponible o la instrumentación está apagada, el estado queda no_disponible o no_medido: el paquete no deduce un derrame del tiempo ni modifica work_mem.

resumen_tabla$meta$estimacion_derrame es un diagnóstico distinto: conserva la estimación de memoria y siempre la rotula como no medida. Puede quedar parcial o no disponible por permisos, falta de ANALYZE, particiones sin estadísticas o un motor que no sea PostgreSQL. Si ambos diagnósticos existen, meta$derrame es la evidencia posterior y prevalece sobre la estimación; una estimación que no superó el límite no contradice un derrame medido. meta$estimacion_derrame_moda y meta$estimacion_derrame_mediana siguen la misma regla, con el método del plan de la moda y el sort de la mediana; n_validos_medido sólo aparece con valor cuando la familia validos fue solicitada y sus agregados planos lo pudieron medir.

resumen_tabla$tiempos reúne las etapas grandes del cliente en las mismas unidades (duracion_ms): lectura_muestra, perfilado_muestra, perfilado_columnas, y los análisis opcionales cuando se solicitan. Una etapa apagada queda con estado no_solicitado; si la instrumentación se apaga queda con estado no_medido y duración NA. La columna nivel dice cuáles se pueden sumar: las de nivel = 1 son disjuntas entre sí, y las de nivel mayor están contenidas en alguna de ellas. perfilado_muestra es inclusivo -contiene el perfilado por columna, las dependencias y los casi-duplicados-, así que sumar la columna entera da más que la corrida.

Datos personales

proteger_datos_personales viaja en ... hacia perfilar() y vale FALSE por omisión: la protección es un complemento que se pide. Con TRUE alcanza a los dos bloques: en resumen_tabla se reemplaza la moda de las columnas clasificadas como personales, se ocultan sus estadísticos de orden y sus momentos, se sanean los datos de conexión y el SQL guardado no contiene ningún valor derivado de los datos. La clasificación se toma del perfil de la muestra; si la muestra no se pudo leer, la protección se aplica a todas las columnas y meta lo declara. Lo que se oculta no queda calculado: su fila de resumen_tabla$sql dice omitido_por_privacidad, con un motivo que distingue la columna clasificada como personal de la que se cerró porque, sin muestra, no se pudo clasificar. Una columna que no se protege publica sus estadísticos aunque comparta el tipo personal con una protegida, como en perfilar(): un importe entero de siete cifras o un número de factura clasificado documento_identidad débil. Una copia del documento la tapa el piso por valor, que

corre con la tabla entera, y la que lo copia sin su verificador se clasifica como columna personal con la regla de `perfilar()`, sobre la muestra. Por la vía por bloques eso incluye `meta$familias`, que guarda cada valor distinto de toda columna no protegida: una columna `debil` —las cédulas sin su verificador de OTRAS personas, como las de un garante— publica ahí cada uno de sus valores, donde `perfilar()` publica sus dos extremos. Limitación conocida: como la clasificación es sobre la muestra, con una muestra que ve menos de tres valores distintos de la copia —chica y ordenada sobre una tabla agrupada por persona— la copia no se clasifica, y el resumen de la tabla entera publica sus extremos, que pueden ser documentos de personas que la muestra no vio; por bloques, cada valor distinto en `meta$familias`. Una muestra mayor o aleatoria lo evita, y también declararla en `columnas_personales`. Lo mismo vale para una columna cuya palabra el resto del nombre califica y que decide por sus valores —`celular_estado`, `contacto_alta`, `usuario_baja`, `nombre_departamento`—: rala, con una muestra que no trae ninguno de sus valores o unos pocos sin su forma, no se protege, y por bloques publica cada valor en `meta$familias`. No se cierra protegiendo por la palabra a la columna sin valores, porque en `perfilar()` taparía la evidencia de toda columna vacía con un nombre así. Un documento por el nombre, en cambio, se protege aunque la muestra no traiga valores —una columna rala de documentos, `cedula_conyuge`, `ci_tutor`—: se publica sólo si la muestra trae veinte valores o más, de dos distintos, que lo desmientan. El desvío se oculta como la media, como en `perfilar()`; también en `resumen_tabla$meta$familias`, donde la vía por bloques (`bloque_filas`) guarda lo que devolvió cada acumulador: ahí se tapan los extremos, la media y el desvío de las columnas protegidas, y cada valor distinto. Y por la vía por bloques corre el mismo piso por valor que sin ella —con los valores de la muestra y, si la muestra no cubre la tabla, con los de la tabla entera—, sobre **todas** las columnas: una cédula copiada en una columna de códigos o un nombre citado en un texto libre se tapan en `resumen_tabla$columnas`, en la corroboración y en `meta$familias` igual que sin `bloque_filas`; la misma tabla publica lo mismo por las dos vías. Lo que el piso tapa en la tabla de columnas —un extremo que es un documento, y la media y el desvío que lo reconstruyen— se tapa también en `meta$familias`. Si la muestra se recorta, la clasificación personal sólo cubre esas filas: las cifras de valor del resumen SQL completo quedan marcadas en `resumen_tabla$cobertura` y, con la protección activa, se ocultan hasta que la clasificación cubra el mismo alcance. La corroboración no repite esas cifras dudosas en su anotación. Y cuando la muestra no cubre la tabla, se traen los valores distintos de las columnas protegidas de la tabla entera —de texto o numéricas, una consulta por columna— y se protege con ellos lo que se publica, con la misma regla que `perfilar()` en memoria: tildes, caja, separadores, el documento parcial, los campos numéricos y la evidencia de los hallazgos. Un valor de una persona que sólo está fuera de la muestra se tapa igual. El **relleno** que sale del piso en memoria —el dígito repetido cinco veces o más en una columna que sin él es casi una clave— se decide sobre la tabla entera, no sobre la muestra: en la muestra de una tabla de movimientos los clientes que se repiten casi no se repiten, y la cédula válida de un cliente con seiscientos movimientos saldría como moda de otra columna. Con la tabla leída completa decide la muestra, que es la tabla; si no, la confirman los conteos exactos de la tabla —`n_distintos` y `n_validos` calculados—, y sin ellos el relleno sigue en el piso. Con `universo = "muestra_motor"` no hay tabla entera que lo confirme —los conteos son los de la muestra del motor, y con `muestra = Inf` la muestra en R es esa misma relación—: el relleno sigue en el piso. Y sale del piso sólo en su columna, como en memoria: si otra columna protegida tiene el mismo valor —el `1.111.111-1` de relleno de `cedula` es la cédula de un cónyuge—, ese valor se sigue tapando. Y la protección es **por tabla**: un valor protegido de otra tabla escrito en ésta no se conoce; `perfilar_coleccion()` los cruza. Se traen hasta dos millones de valores; si hay más, o si la consulta no se puede hacer, se tapan todos los valores de celda de las columnas no personales —moda, extremos, ejemplos y, por la vía por bloques, cada valor distinto y los extremos de `meta$familias`— y la evidencia de sus hallazgos, y

`resumen_tabla$meta$proteccion_personal$fuera_de_muestra` dice por qué. Salvo un valor igual al nombre de la tabla o de su esquema, que es estructura: se publica en `meta`, en cada SQL guardada y en `print()`, y un valor de celda igual a él no se tapa —taparlo taparía el nombre en todo el objeto—; lo mismo cuando sin muestra se cierran todas las columnas. Limitación conocida: un valor igual al nombre de una **columna** sí se tapa, porque un nombre de columna puede ser el de una persona —una planilla con una columna por docente—, y tapa también ese nombre en la descripción de las sondas de dialecto (`meta$dialecto$sondas`) y, con `bloque_filas`, en la de la fuente por bloques (`meta$fuente_bloques`, `meta$plan`); en `columnas$columna` y en la SQL guardada queda. En SQLite los valores se leen como texto, porque una columna de afinidad numérica guarda texto y el controlador lo convertiría en número; también sin muestreo, cuando la muestra es la tabla entera: ahí se releen como texto, con una consulta, las columnas protegidas que el controlador no leyó como texto. Y el motivo de una cifra que no se pudo leer como número no cita el valor de una columna protegida. Tampoco el **mensaje de un error del motor**, que es texto ajeno y cita la celda que hizo fallar la consulta —`invalid input syntax for type bigint: "4.685.467-9"`, en una vista que castea una cédula escrita con puntos—: ese valor nunca se leyó, así que el piso no lo conoce y puede ser de cualquier columna. Con la protección activa, todo mensaje de error que se publica —en `sql`, `cobertura`, `meta`, en la colección y en los avisos— lleva sus literales entre comillas —simples, dobles, tipográficas, bajas como las del polaco, angulares— cambiados por `[literal omitido]`; también un identificador entre comillas, salvo el nombre de la tabla o de una de sus columnas. Y fuera de las comillas el mensaje publica **sólo lo permitido**: el vocabulario de los mensajes de los motores, en inglés y en castellano, las palabras de SQL, los nombres de tipo, los nombres de la tabla y de sus columnas, los códigos SQLSTATE, los signos y los números de menos de seis cifras —el piso de la protección: la línea, la posición, un código—; todo lo demás sale `[literal omitido]`. Un número se cuenta con sus separadores: `4.685.467-9` es uno de ocho cifras, y se lee como lo lee el piso de la protección, con los mismos separadores —un espacio duro o uno fino, el punto medio, una raya, `" - "`, dos espacios, un tabulador o un salto de línea entre los grupos: `"099 - 123 - 456"` es uno de nueve—. El mensaje se lee como lo dijo el motor: odbc lo arma con cli, que pone una viñeta después de cada salto de línea y corta las líneas al ancho de la consola, y un número partido en dos líneas —`"2901\n1234"`— quedaría partido por la viñeta; se toman las líneas que guarda la condición, unidas por el salto que puso el motor, y así se publican, sin las viñetas. Tampoco llevan los colores ni los enlaces que cli pone en una consola a color —la de RStudio—: las secuencias ANSI quedarían guardadas en el motivo y, con protección, pegadas a los marcadores. Lo que cli agrega dentro de una línea no se distingue del texto del motor y queda: sin color, cli cita entre comillas el archivo del controlador (`'nanodbc/nanodbc.cpp:1802'`) y con protección esa cita sale `[literal omitido]`; con color se publica. Así se tapa también el valor que el motor cita sin comillas —duckdb: `Type INT64 with value 59899123456 can't be cast`, y la cadena que `strptime` repite en otra línea, que se tapa entera; SQL Server con la sesión en sueco o neerlandés— y la cola de un valor que lleva una comilla adentro. Y el mensaje pasa también por el piso con los valores protegidos que la llamada ya leyó —en `perfilar_coleccion()`, los de todas sus tablas—, antes de las dos pasadas, con el valor entero a la vista, y otra vez al final: un valor escrito sólo con lo permitido —los apellidos `"Campos Cadena"`, dos palabras del vocabulario, o un nombre cuyas palabras son columnas de otra tabla— se tapa entero, y el nombre de una columna que es también uno de esos valores —la docente `"Florencia"` de una planilla con una columna por docente— deja de exceptuarse, con la comparación del piso. En `perfilar_coleccion()` cada mensaje se vuelve a leer al final, desde como llegó, con todo lo que la colección leyó: con una vista perfilada antes que su tabla base, el piso todavía no conoce el nombre cuando llega el mensaje, y releer el texto ya tapado dejaría su resto —`"De Los Campos"` de `"María De Los Campos"`—. Lo que queda abierto: un valor que la llamada **no** leyó —la celda que hizo fallar la lectura de la muestra, de una columna

que la vista castea— escrito sólo con lo permitido, o igual al nombre de una columna, se publica, porque el mensaje no tiene cómo distinguirlo del vocabulario ni de un nombre. Un mensaje en otro idioma conserva su SQL, sus tipos y sus nombres y se lee peor, que es el lado cerrado. Quedan una palabra de SQL entre comillas —`Incorrect syntax near 'LIMIT'` dice que el motor no acepta ese dialecto— y un signo suelto. El texto que PostgreSQL cita sin comillas al no poder leer un JSON (`CONTEXT: JSON data, line 1: ...`) se tapa hasta el fin de la línea, en PostgreSQL, en duckdb y en SQL Server. **Sin protección** (`proteger_datos_personales = FALSE`) el mensaje del motor se publica entero, en todas esas salidas: el literal que hizo fallar la lectura y el SQL de esa lectura, sólo sin las viñetas, los cortes ni los colores que agrega cli. El diagnóstico es el cometido del paquete —`Conversion failed when converting the nvarchar value 'N/D' to data type int` dice que la columna trae un faltante disfrazado—, y sin protección ya se publican valores de todas las columnas —modas, ejemplos, patrones—: omitirlo ahí no protegería nada y quitaría el diagnóstico. Vale igual en `perfilar_coleccion()`, que pasa la bandera a cada tabla y la aplica a sus propios motivos. `plan_perfilado_dbi()`, `coleccion()`, `estimar_costo_coleccion()` y `relaciones_coleccion()` no tienen la bandera: llamadas por su cuenta, tapan el mensaje como con la protección activa. `incluir_valores = FALSE` va más lejos: no emite las consultas de moda ni de mediana y no informa mínimo ni máximo, útil cuando la tabla es un padrón y la moda de un identificador único es un documento real. Si se pidió `desvio`, la sonda de magnitud que ese cálculo necesita puede emitirse, pero su motivo nunca publica el valor devuelto por el motor.

Valores de texto extensos

En `muestra_motor`, las columnas de texto de PostgreSQL, SQL Server, MySQL y MariaDB que superan `max_largo_valor_vocabulario` se solicitan como largo en caracteres, largo en bytes y resumen MD5. El contenido completo se conserva solo para los valores que no superan ese umbral, por lo que el tamaño máximo de una fila se conoce antes de `dbFetch(n)`. La cobertura agrega `texto_largo_no_evaluado` para declarar los diagnósticos textuales que no se pudieron calcular sobre esos valores; sus faltantes, duplicados y longitudes siguen entrando en las métricas disponibles.

Progreso

Traer la tabla entera puede tardar minutos sobre una tabla grande, y una corrida callada no se distingue de una colgada. En una sesión interactiva se muestra una barra que avanza contra las consultas que el plan dice que se van a emitir —un total conocido, no una estimación—, y no aparece cuando la corrida es de menos de una docena de consultas, porque termina antes de que sirva.

`options(lupa.progreso = TRUE)` la fuerza y `FALSE` la apaga; fuera de una sesión interactiva está apagada, para que la salida de un guion no traiga ruido que despues haya que filtrar. No cambia ningún valor de lo que se mide: hay una prueba que compara el perfil con la barra y sin ella.

See Also

`plan_perfilado_dbi()`, `perfilar()`

Examples

```
if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
```

```
  DBI::dbWriteTable(con, "ejemplo", data.frame(id = 1:10, valor = 11:20))
  resultado <- perfilar_dbi(con, "ejemplo", muestra = 5, orden_muestra = "id")
  DBI::dbDisconnect(con)
}
```

`perfilar_por` *Perfilar una tabla por grupos de filas*

Description

Aplica `perfilar()` a cada grupo de filas por separado y devuelve los hallazgos de todos los grupos en una sola tabla. Es la respuesta al formato largo: una tabla donde cada fila describe un atributo distinto no es una tabla, son muchas apiladas, y perfilarla como si fuera una sola mezcla dominios que no tienen nada que ver entre sí.

Usage

```
perfilar_por(datos, por, clave = NULL, min_filas = 30L, ...)
```

Arguments

`datos`	Data frame a perfilar. Sus nombres de columna tienen que ser distintos: cada grupo se arma por nombre, y con dos columnas iguales se perfilaría una en lugar de la otra. Si los hay, la función se detiene; renómbrelas, por ejemplo con `names(datos) <- make.unique(names(datos))`.
`por`	Nombre de una columna atómica cuyos valores definen los grupos. Los ausentes forman un grupo propio, con la etiqueta `"(ausente)"`. Si la columna trae ese mismo texto como valor real **y además hay ausentes**, los dos caen en un solo grupo —no se pierde ninguna fila— y la colisión se declara en `cobertura_grupos`, con cuántas filas aporta cada una. Sin ausentes no hay colisión y no se declara ninguna. La etiqueta de cada grupo **vuelve a su valor**. En una columna de dobles, `as.character()` usa 15 cifras significativas y dos valores distintos pueden escribirse igual —`1e17` y `1e17 + 32` dan los dos `"1e+17"`—; en ese caso la etiqueta lleva las cifras que hacen falta para distinguirlos (`"1.0000000000000003e+17"`), de modo que cada grupo corresponde a un valor y `as.numeric()` de la etiqueta recupera ese valor exacto. Para la enorme mayoría de los números la etiqueta es la de siempre. Lo mismo vale para las otras clases: un `integer64` se etiqueta con sus cifras exactas, aun por encima de 2^53; un complejo, con sus dos partes; una fecha, con su escritura `AAAA-MM-DD`, y una fecha-hora, con su instante en UTC y el rótulo `UTC` —al segundo, con la fracción exacta si la tiene—, la misma escritura en cualquier sesión; la zona declarada de la columna la publica `perfilar()` en `zona_horaria_origen`. No con su número de días o de segundos. Si la columna es un **factor**, un nivel declarado sin ninguna fila es un grupo de cero filas: no se perfila y se declara en `cobertura_grupos`, igual que cualquier grupo por debajo de `min_filas`. Eso incluye un nivel llamado `"(ausente)"`

sin filas cuando la columna también tiene NA: los NA se perfilan como su propio grupo, y el nivel vacío se declara aparte.

`clave` Nombres de columnas de identidad que se conservan en cada grupo aunque estén enteramente ausentes. Importa: sin la clave de entidad, el diagnóstico de filas duplicadas informa como duplicada cada repetición del valor del atributo.

La declaración viaja al perfilado de cada grupo, igual que si se pasara `clave` a `perfilar()`: la ley de Benford no corre sobre esas columnas y la cobertura publica que la clave fue declarada, en vez de deducir que «parece un identificador». Si la clave nombra la columna de agrupación, ese nombre se recorta antes de reenviarlo, porque en la rebanada esa columna ya no está.

`min_filas` Grupos con menos filas que este número no se perfilan y se declaran en la cobertura. Tiene que ser un entero positivo: un valor fraccionario se rechaza, en vez de truncarlo y perfilar un grupo que estaba por debajo del umbral. El valor por omisión evita conclusiones sobre grupos donde ningún diagnóstico tiene soporte.

`...` Argumentos enviados a `perfilar()` para cada grupo. Los que acotan el trabajo —`muestra` entre ellos— se aplican **dentro de cada grupo**, no sobre la tabla entera: con `muestra = 100` y grupos de 500 filas, los diagnósticos que muestrean miran 100 de cada grupo. El alcance viaja en la evidencia de cada hallazgo, que declara sobre cuántos valores se midió; `n_filas_grupo` sigue siendo el tamaño del grupo, no el de la muestra. Un nombre abreviado vale lo mismo que el entero, como lo completa R al llamar a `perfilar()`: `proteger = FALSE` es `proteger_datos_personales = FALSE` también para las etiquetas de grupo. La bandera, `fecha` y los nombres se validan al entrar, aunque ningún grupo llegue a `min_filas`. Los demás valores los valida `perfilar()` al perfilar el primer grupo: si ninguno llega a `min_filas`, uno inválido no da error.

Details

Dentro de cada grupo se descartan las columnas enteramente ausentes antes de perfilar. En un modelo entidad-atributo-valor bien formado eso deja viva exactamente la columna de valor que corresponde al atributo del grupo, y es lo que evita informar como falta lo que es la forma del dato. Las declaraciones que nombran columnas retiradas se recortan para ese grupo y el nombre de cada recorte, con su motivo, se declara en `cobertura_grupos$columnas_descartadas`. En `aplicabilidad` también se recorta una regla si `all.vars()` de su fórmula menciona una columna original que no está en la rebanada; una variable que no es columna de la tabla se conserva porque puede venir legítimamente del entorno de la fórmula.

La función no adivina cuál es la columna de agrupación: la declara quien conoce el dato, igual que `perfilar()` no adivina claves ni jerarquías.

Value

Data frame de clase `hallazgos_por_grupo` con las columnas de `hallazgos` de `perfilar()` precedidas por `grupo` y `n_filas_grupo`. El atributo `cobertura_grupos` declara los grupos no perfilados, las columnas enteramente ausentes y las declaraciones recortadas por grupo. Su columna lógica `grupo_perfilado` dice de qué clase de grupo habla cada fila: hay filas sobre grupos que **sí** se perfilaron —la colisión de `"(ausente)"`, las columnas descartadas— y filas sobre grupos que

no. Sin esa distinción n_filas_grupo no se podría sumar, porque un grupo perfilado aparece en los hallazgos y otra vez en la cobertura.

Cómo reconciliar las filas. Hay una fila por grupo y motivo, así que un grupo puede aparecer más de una vez; la cuenta se hace sobre grupos distintos. La suma de n_filas_grupo sobre los grupos distintos —los perfilados, que están en los hallazgos y en las filas con grupo_perfilado = TRUE, más los no perfilados— da las filas de la tabla. Para que la cuenta cierre, un grupo perfilado que no produjo ningún hallazgo también tiene su fila en cobertura_grupos, con grupo_perfilado = TRUE y ese motivo: si no, no estaría en ninguna de las dos tablas.

El atributo etiquetas_personales declara si la columna de agrupación lleva datos personales. Las etiquetas de grupo **son** valores de esa columna, así que la salida los publica —en los hallazgos y en las dos tablas de cobertura— aunque perfilar() enmascare esa misma columna. No se enmascaran aquí porque la etiqueta es el eje del resultado y sin ella los grupos no se distinguen; pero tampoco ocurre en silencio: se avisa al ejecutar y queda declarado en el objeto. Para que no se publiquen, agrupe por una columna seudonimizada. Lo mismo, con tipo contiene_valor_protegido, cuando la columna de agrupación no es personal pero alguna etiqueta contiene un valor protegido de otra columna —un proveedor juanperezsrl junto a un titular protegido «Juan Perez»—: la etiqueta se publica igual, y n_grupos cuenta cuántas lo contienen. Cada grupo se perfila sobre sus filas, pero la salida entera se barre además con los valores protegidos de toda la tabla: el titular que en la columna protegida sólo aparece en un grupo no se publica en la evidencia de otro. El atributo queda vacío cuando ninguna etiqueta lleva datos personales, y también cuando proteger_datos_personales es FALSE, porque entonces ya está declarado que se quieren los valores. Si el léxico no reconoce el nombre de la columna, decláresela con columnas_personales: ese argumento —como columnas_opcionales, clave, columnas_sin_ceros, columnas_no_negativas, columnas_codigo y aplicabilidad— puede nombrar la columna de agrupación, y se recorta de lo que se envía a perfilar() para cada grupo, donde esa columna ya no está. El recorte se publica por grupo en cobertura_grupos$columnas_descartadas, junto con el nombre de la declaración y el motivo. Para aplicabilidad, también se recorta una regla cuando alguna variable de su fórmula —según all.vars()— es una columna original ausente de la rebanada; las referencias que no son columnas originales se conservan como referencias posibles al entorno de la fórmula.

El atributo cobertura_diagnosticos declara, **por grupo**, los diagnósticos que no se evaluaron y por qué. Cada grupo se perfila por separado, así que cada uno declina los suyos: una columna puede tener bastantes filas en un grupo y muy pocas en otro. Sin esa tabla, un grupo sin hallazgos se lee como un grupo sano, cuando puede ser un grupo sobre el que no se miró.

Las senales de numeracion densa y de identificacion se miden tambien sobre columnas integer64 con comparaciones exactas. Si un grupo vuelve a conjeturar un centinela por la particion, el hallazgo se mueve a esta cobertura y no se publica como defecto del grupo. "La columna entera" es la que perfilar() mide con la misma declaración: las filas que aplicabilidad deja fuera no cuentan para el centinela, la numeración ni la guarda de Benford. Y lo que la columna entera sí conjetura —una numeración cuya moda sobresale, o un candidato fuera de su rango— no es de la partición: el grupo lo publica.

See Also

perfilar(), cobertura_analisis()

Examples

```
set.seed(1)
```

```
largo <- data.frame(
  entidad = rep(1:40, each = 2),
  atributo = rep(c("pais", "edad"), 40),
  valor = c(rbind(sample(c("UY", "AR"), 40, TRUE), as.character(20:59)))
)
hallazgos <- perfilar_por(largo, "atributo", clave = "entidad", min_filas = 10)
head(hallazgos[, c("grupo", "columna", "tipo_hallazgo")])
```

`persistir_analisis` *Guardar y recuperar un análisis*

Description

Persiste un objeto `analizar()` en RDS con número de esquema. Los datos de entrada no se guardan por omisión. Las funciones de reglas se sustituyen por declaraciones pequeñas: una dependencia funcional se reconstruye al leer sólo si los datos fueron incluidos; una función arbitraria queda desactivada. Así el archivo no serializa entornos de ejecución completos.

Usage

```
guardar_analisis(
  x,
  archivo,
  incluir_datos = FALSE,
  proteger_datos_personales = FALSE,
  sobrescribir = FALSE,
  comprimir = "xz"
)

leer_analisis(archivo)
```

Arguments

`x`	Objeto creado por `analizar()`.
`archivo`	Ruta del archivo RDS. Una ruta que ya es un directorio se rechaza: el archivo va dentro, con su nombre.
`incluir_datos`	Si se persiste la copia de los datos conservada por `analizar(conservar_datos = TRUE)`.
`proteger_datos_personales`	Si se protege toda evidencia derivada. `FALSE` por omisión.
`sobrescribir`	Si se permite reemplazar un archivo existente.
`comprimir`	Compresión admitida por `saveRDS()`: un lógico, `"gzip"`, `"bzip2"` o `"xz"`.

Details

La protección de datos personales con evidencia suficiente se vuelve a aplicar antes de escribir. Incluir datos que contienen columnas protegidas exige desactivar expresamente esa protección; una clasificación débil se conserva como información pero no activa esa restricción.

El archivo se escribe con el **formato RDS 2**, no con el 3 que `saveRDS()` usa por omisión. El 3 anota la codificación nativa de quien escribe y al leer *traduce* desde ella el texto que no declara la suya; en Windows esa traducción no falla y cambia los bytes en silencio, así que un análisis guardado bajo un locale y leído bajo otro podía volver con el texto alterado.

Ese formato tiene un costo y conviene saberlo: **no conserva la representación compacta** de los vectores que R guarda así. Medido, una columna `1:1e6` ocupa 103 bytes con el formato 3 y 2.127.903 con el 2; con `incluir_datos = TRUE` sobre una tabla con una columna `id = seq_len(n)`, eso son unos 2 MB por cada millón de filas, y a escala la escritura puede exigir materializar en memoria lo que el formato 3 no materializa. Se eligió así porque un archivo más grande es visible y recuperable, y un texto corrompido en silencio no lo es. Quien prefiera la otra relación puede serializar el objeto por su cuenta.

Value

`guardar_analisis()` devuelve la ruta de forma invisible; `leer_analisis()` devuelve un objeto `analisis`.

El objeto guardado gana `meta$persistencia`, que declara qué se guardó y qué no: `version_esquema`, si los datos viajan (`datos_incluidos`), si la evidencia salió protegida (`evidencia_protegida`) y cuántas funciones se sustituyeron por su descripción (`funciones_sustituidas`). Es el único campo que un objeto leído tiene y el original no: lo que se guarda no es idéntico a lo que se analizó, y esa diferencia se declara en vez de suponerse. Al leer, si hubo funciones sustituidas, `advertencias` agrega una fila que declara que la propuesta se volvio a derivar sin ellas y que puede no coincidir con la decision original, que se conserva.

See Also

`analizar()`, `reportar()`, `guardar_historico()`

Examples

```
a <- analizar(datos_administrativos[, c("sexo", "ingreso")],
              analizar_dependencias = FALSE)
ruta <- tempfile(fileext = ".rds")
guardar_analisis(a, ruta)
b <- leer_analisis(ruta)
unlink(ruta)
```

`planificar_limpieza` *Construir y aplicar un plan de limpieza auditable*

Description

planificar_limpieza() transforma los hallazgos de un objeto perfil en un objeto de datos editable, sin modificar los datos examinados. Cada fila representa una acción propuesta. Sólo se marcan como recomendadas las estrategias correctas con independencia del dominio; algunas, como marcar valores extremos, permanecen inactivas hasta que se decida actuar. Las decisiones contextuales quedan desactivadas y los formatos de fecha ambiguos quedan bloqueados.

Usage

```
planificar_limpieza(perfil, datos = NULL, soporte_minimo_dependencia = 2L)

aplicar(plan, datos, permitir_eliminacion = FALSE, conservar_eliminados = TRUE)
```

Arguments

perfil	Objeto de clase perfil creado por perfilar().
datos	data.frame, tibble o data.table sobre el que se ejecuta el plan. El objeto recibido no se modifica.
soporte_minimo_dependencia	Cantidad mínima de observaciones concordantes por valor determinante para proponer una imputación.
plan	Objeto de clase plan_limpieza o data frame con el mismo contrato. Puede filtrarse y editarse antes de aplicarlo; una acción cuyos parámetros no son la lista con nombres que propone el plan —un vector, NULL— queda fallida con ese motivo.
permitir_eliminacion	Segundo consentimiento obligatorio para ejecutar acciones que eliminan filas o columnas. Sin él, una acción eliminatoria seleccionada hace que aplicar() **se niegue antes de tocar nada**: no se ejecuta ninguna acción del plan —tampoco las que no eliminan— y no hay registro, porque no hubo ejecución. El error nombra las acciones y los dos caminos: dar el consentimiento, o desactivarlas y aplicar el resto.
conservar_eliminados	Si se conservan en el resultado las filas y columnas retiradas. Es TRUE de forma predeterminada.

Details

aplicar() ejecuta exclusivamente las filas con aplicar == TRUE, sobre una copia de datos. Verifica que cada columna siga siendo identificable y que las conversiones sean completas antes de sustituirla. Devuelve los datos nuevos junto con un registro de las acciones y sus parámetros. El mismo registro queda en el atributo registro_limpieza de los datos resultantes.

Las alternativas para un mismo hallazgo comparten grupo; las acciones independientes usan NA. Como máximo una alternativa de cada grupo puede tener aplicar == TRUE, invariante que aplicar() vuelve a validar. No se agrega una fila ficticia para "no hacer nada": decision_grupo distingue pendiente, recomendada, desactivada, elegida y omitida, mientras recomendacion_grupo = "no_hacer_nada" representa una recomendación explícita de conservar los datos. Esto permite separar un grupo aún no revisado de una omisión deliberada.

`estado` distingue acciones `lista`, `bloqueada` e `informativa`; `orden` fija la secuencia reproducible. La acción `reparar_codificacion`, cuando está seleccionada, tiene precedencia sobre las demás acciones de su misma columna: corre primero en esa columna para que las restantes reciban texto ya decodificado, aunque su `orden` editado sea mayor. No reordena acciones de otras columnas. Entre las acciones restantes, y entre empates, el desempate lo resuelve `id_accion`, no la posición de la fila: reordenar el plan no cambia el resultado. Y un grupo marcado como `elegida` sin ninguna acción activa se rechaza, porque no es ni una elección ni una omisión. `n_afectadas` es la estimación del perfil sobre **lo que esta acción tocaría**, que puede ser menos que el conteo del hallazgo que la originó cuando la acción sólo cubre parte del caso: una columna con tres valores de codificación rota, de los cuales uno es reparable, produce un hallazgo con `n_afectados = 3` y una acción `reparar_codificacion` con `n_afectadas = 1`. Las dos cifras son ciertas y cuentan cosas distintas. `unidad_conteo` dice si cuenta filas, columnas o valores distintos —lo declara la acción cuando cuenta en una unidad propia, y sólo si no lo hace se hereda del hallazgo—. El registro informa `n_cambiadas` sobre los datos recibidos.

`n_afectadas` **y** `n_cambiadas` **pueden no coincidir, y las dos son ciertas.** La estimación se calcula sobre los datos que se perfilaron; el registro cuenta lo que pasó al aplicar. Si una acción anterior del mismo plan ya tocó esa columna, la posterior encuentra menos —o más— de lo estimado. La única excepción de precedencia es `reparar_codificacion`, que se adelanta a las demás acciones de su columna para que ninguna opere sobre mojibake. Por ejemplo, con `convertir_sentinelas_numericos` (orden 110) convirtiendo tres `-999` en ausentes, `winsorizar_outliers` (orden 520) recorta dos valores donde el plan estimaba siete. No es un desvío que ocultar: `orden`, `n_afectadas` y `n_cambiadas` se publican los tres, y compararlos es la forma de ver el efecto de la composición. Sólo el caso extremo —la acción no produce **ningún** efecto —también cuando el plan no trae una estimación válida— se registra como `fallida` con su motivo.

Esa comparación sólo lee composición cuando las dos cifras cuentan en la misma unidad. `n_afectadas` cuenta en la `unidad_conteo` que el plan declara y `n_cambiadas` cuenta lo que la acción tocó al aplicarse, que es su unidad natural. Cuando la acción declara `unidad_conteo = "valor_distinto"` —las conversiones de mayúsculas y minúsculas—, las dos cifras miden poblaciones distintas y su diferencia no dice nada sobre la composición: medido sobre `c("Ana", "ana", "ANA", "Beto", " Ana ")`, `convertir_minusculas` estima `n_afectadas = 3` valores distintos y el registro publica `n_cambiadas = 4` celdas. Las dos son ciertas. La unidad del plan se recupera uniendo el registro con el plan por `id_accion`, que los dos publican. `reversible` indica si la conversión conserva la identidad de cada valor. Las conversiones se comprueban sobre todos los valores de `datos`: las numéricas bloquean ceros iniciales, colisiones no inyectivas y valores con más cifras significativas de las que guarda un número de doble precisión —un decimal se compara con la precisión con que vino escrito—, mientras que fechas, fechas-hora y lógicos sólo bloquean conversiones no ejecutables o no inyectivas. Las fechas pueden cambiar a la representación canónica del tipo sin que eso sea una pérdida. Sin `datos` no se puede hacer la comprobación y la acción queda bloqueada. Cuando **una conversión de tipo** se ejecuta y no es reversible se marca `destructiva` —también si la columna no era segura y se activa a mano—, no se activa por defecto y el registro conserva `n_no_reversibles` y la justificación de la decisión.

`destructiva` no es sinónimo de "pierde algo": marca las acciones que el usuario tiene que activar a mano —las que retiran filas o columnas y las conversiones que pierden representación, como `winsorizar_outliers` sobre una columna **guardada como entera**, donde los límites de Tukey son cuartiles y la columna queda en doble precisión: la justificación lo dice y `parametros` publica `tipo_original` y `tipo_resultante`. Sobre cualquier otra clase —texto con números, factor, fecha— la acción conserva la clase y reescribe sólo las celdas recortadas; `Inf`, `NaN` y `NA` quedan como estaban— y por eso ninguna acción `destructiva` puede estar `recomendada`. Hay acciones

recomendadas que sí pierden el valor de una celda: `convertir_ausencias_textuales` cambia un marcador por `NA` y `eliminar_controles_invisibles` quita un carácter. Esas lo dicen en su justificación y el registro las cuantifica en `n_no_reversibles`; leer `destructiva = FALSE` no significa que no se haya perdido nada, sino que el paquete pudo recomendar la acción sin conocer el dominio. Sobre una columna **factor** las acciones por celda devuelven texto: el resultado no puede ser un factor incompleto, así que el orden declarado y los niveles sin observaciones no se conservan. La justificación de la acción lo dice, y si el factor es ordenado la acción queda recomendada pero **sin activar**, porque conservar el orden es una decisión del dominio.

Tres atributos del plan declaran lo que el plan no cubre. `cobertura_diagnosticos` trae los diagnósticos que el perfil no pudo evaluar, para que leer tres acciones no se confunda con "lo demás está bien". `hallazgos_sin_accion_por_columna_ambigua` trae los hallazgos medidos que no produjeron acción porque su columna no se puede identificar: comparte nombre con otra y no hay forma de saber sobre cuál actuaría la limpieza —el remedio es normalizar los nombres y volver a perfilar—, o no está entre las columnas del perfil, o no tiene nombre —su nombre es `NA`; una columna llamada con el texto `"NA"` sí lo tiene, y el plan actúa sobre ella—; el `motivo` dice cuál. Un hallazgo de varias columnas —una relación aritmética en `neto,iva,total`, una de orden— no es una columna ambigua: lo es sólo si alguna de sus columnas lo es. Sus columnas se leen de la trazabilidad del hallazgo y no del texto de `columna`: una columna llamada `a,b` no se confunde con la relación entre `a` y `b`. `hallazgos_sin_accion` trae el resto: los hallazgos que el perfil midió, de una columna identificada, de varias o de la tabla, y que ninguna acción del plan atiende —porque el plan no tiene una estrategia para ese tipo, o porque la que tiene necesitaba una condición que no se cumplió—. Su `motivo` dice si la misma columna (o esas columnas) recibió otras acciones, sin afirmar que alguna cubra el hallazgo: eso es una decisión del dominio. Un hallazgo `ok` no espera acción y no entra a ninguno de los dos. Los tres se imprimen con el plan.

`convertir_numero_regional` sólo se recomienda si todos los valores presentes comparten convención decimal, unidad y moneda. Un valor con `%` se divide por 100 y queda como proporción en `[0, 1]`, la escala con que el paquete publica toda proporción; una unidad o un símbolo de moneda dejan de formar parte del valor y quedan en `parametros`. La justificación lo dice cuando ocurre. Una columna que mezcla `5` con `5 %` no se convierte: habría que decidir si el número sin `%` está en la misma escala. La acción de codificación prueba las tablas congeladas de varias codificaciones y deja en `estado_reparacion` uno de `reparado`, `reparado_parcialmente` o `no_se_pudo`. Una reparación parcial —una celda que el motor repara sólo en parte— no se activa automáticamente: la acción queda `lista` con `aplicar = FALSE`, y se selecciona de forma explícita marcando `aplicar = TRUE`; entonces se aplican también las parciales. Una celda sin reparación local —con el carácter de reemplazo U+FFFD, o que el motor no pudo reparar— no frena la de las demás: queda como está y la justificación de la acción la cuenta. La estrategia se llama `reparar_codificacion` y no limita el motor a latin-1. Un texto legítimo que se escribe con los mismos bytes que un mojibake —por ejemplo, uno que cita literalmente la Ã seguida de ©— no se distingue de uno roto: se acusa, la evidencia del plan lo muestra antes de aplicar, y si se aplica la acción se reescribe como la letra que esos bytes codifican. Si la columna contiene texto así, no active la acción. Unos mismos bytes pueden parecer un control: el segundo byte de una «Á» mal leída es un C1, el de una «í» un guion blando, y `c2 80` es la huella de un euro de Windows-1252 leído como latin-1. Lo que la reparación reemplaza no cuenta como carácter invisible y ninguna otra acción del plan lo toca: aunque se desmarque `reparar_codificacion`, la secuencia queda entera y el perfil siguiente la sigue encontrando. Lo mismo si el motor no puede reparar la celda: sus letras rotas quedan como están. Los controles que la reparación deja como están, como un U+0081 suelto o un guion blando fuera de una letra rota, se diagnostican y se quitan, también en una celda con mojibake. El perfil cuenta un invisible sólo donde la acción puede quitarlo. Una celda cuyo texto no se puede leer —declarada `bytes` con

bytes que no son UTF-8, o sin marca en una sesión UTF-8, que es lo que deja `read.csv()` sin `fileEncoding` sobre un archivo latin1— no se transforma: el perfil no la midió y la informa como `codificacion_invalida`, cuyo remedio es volver a leer la fuente declarando su codificación. Las acciones de texto trabajan sobre las demás celdas de la columna y cuentan sólo lo que cambiaron. Una celda sin marca cuyos bytes son UTF-8 válido se trata como UTF-8 en cualquier locale, igual que en el perfil: con `LC_CTYPE=C` las acciones hacen lo mismo que en una sesión UTF-8.

Una celda `latin1` **sí** se transforma —ahí R conoce la codificación y la convierte sin pérdida—, y el resultado queda marcado `UTF-8`. Como la acción toca sólo las celdas que cambia, una columna `latin1` puede quedar con marcas **mixtas**: las celdas transformadas en `UTF-8` y las intactas en `latin1`. El valor es el mismo, pero los bytes no, así que el registro lo cuenta en `n_codificacion_normalizada` en vez de dejarlo implícito. Ese conteo es de recodificaciones, no de marcas: una celda sin marca cuyos bytes son UTF-8 válido ya es UTF-8 y no cambia de codificación aunque la acción la devuelva marcada, y un texto ASCII se escribe igual en toda codificación —decodificar `José` cambia el valor, no la codificación—. No se convierte la columna entera a propósito: eso tocaría celdas que el plan no declaró como cambiadas y haría mentir a `n_cambiadas` por el otro lado. Si se marca una acción que no está `lista`, `aplicar()` aborta antes de modificar la copia y enumera las filas problemáticas. Una acción que sí está lista pero falla se registra con su error y no impide aplicar las siguientes: cada una conserva atomicidad sobre su propia columna o tabla. Una acción de nombres preparada para otros nombres —un perfil de otra versión de la tabla, con dos `x` donde los datos tienen una— es una de esas: queda `fallida` con el motivo, y en un plan armado con `datos` ya sale `bloqueada`. Las acciones que efectivamente eliminan filas o columnas requieren además `permitir_eliminacion = TRUE`; una conversión `destructiva` requiere selección explícita y deja la pérdida cuantificada. Por defecto, el resultado conserva lo retirado en `eliminados`; use `conservar_eliminados = FALSE` para evitar ese costo de memoria.

Los hallazgos `controles_invisibles`, `entidades_html` y `separadores_en_campo` tienen acciones separadas. La detección de invisibles informa tanto los caracteres que se pueden normalizar como los ZWJ/ZWNJ significativos; la normalización actúa sobre un conjunto más pequeño que la detección. La acción `eliminar_controles_invisibles` quita controles C0/C1 que no son separadores y los invisibles Unicode de transporte, y se recomienda por defecto; conserva ZWJ/ZWNJ. `normalizar_espacios_invisibles` colapsa espacios Unicode (incluido NBSP) a un espacio ASCII y, como su hermana, se recomienda y se activa por defecto: queda un espacio, así que el valor sigue ahí; si al colapsarlo dos valores que eran distintos quedan iguales, el registro lo cuenta en `n_no_reversibles`. `decodificar_entidades_html` cubre las entidades con nombre comunes en español y referencias numéricas válidas, pero no se activa sola porque un ampersand puede ser contenido legítimo. Decodifica **una capa**: un texto codificado dos veces —`&`— queda en `&`, y el perfil del resultado lo vuelve a señalar. No se itera hasta el fondo porque no se puede distinguir una doble codificación de un texto que quería decir `&`: la segunda capa se decide viéndola. `reemplazar_separadores` convierte tabulaciones, saltos de línea, avances de página y tabulaciones verticales (`\\t`, `\\n`, `\\r`, `\\r\\n`, `\\f` y `\\v`) en un espacio y también requiere una decisión explícita. Son exactamente los códigos 9 a 13 que cuenta el hallazgo: los separadores Unicode U+2028 y U+2029 son de `normalizar_espacios_invisibles` y el control U+0085 de `eliminar_controles_invisibles`, y sobre celdas `bytes` la acción no toca ningún byte de un carácter multibyte. Las tres acciones registran el número de valores cambiados. Una comparación aproximada con `normalizar = TRUE` usa estas mismas clases: colapsa espacios y omite basura de transporte, pero conserva ZWJ/ZWNJ.

Las imputaciones por dependencia funcional se ofrecen desactivadas. Aunque una dependencia exacta permite deducir un valor sin usar media, moda o un modelo externo, sigue siendo una regularidad aprendida de una sola entrega y puede reflejar un error sistemático en vez de una regla de

negocio. El plan conserva el mapa y su soporte para que el usuario la confirme; sólo entonces se aplica y se vuelve a validar contra los datos recibidos. Si la protección enmascaró alguna celda del mapa —una clave o un valor, también cuando ninguna de las dos columnas es personal pero una celda contiene un valor protegido, como la sucursal que se llama igual que un titular—, queda marcado `mapa_enmascarado = TRUE` y se valida igual que sin protección: las filas que el plan publica en claro son el mapa tal cual, y una fila tapada se resuelve con el par de los datos recibidos que tiene su forma y que, tapado con los valores protegidos de esos mismos datos, da lo que el plan publicó; la relación de los datos se toma con el soporte declarado, sin publicar sus valores. Si los datos contradicen una fila —la publicada en claro o la tapada—, la acción queda `fallida`, y una clave que no está en el plan no se imputa: reconstruir el mapa sobre los datos recibidos escribiría, en otra entrega, valores que el plan nunca mostró. Una fila tapada no se puede validar si los datos recibidos no traen el valor protegido que la tapó —un subconjunto sin ese titular—: también queda `fallida`, con el motivo, y se aplica planificando sobre esos datos. Y si los datos piden imputar una clave cuya fila el plan publicó tapada pero no traen ningún par de esa clave que la resuelva —la entrega que trae sólo las filas a completar—, la acción tampoco escribe a medias: queda `fallida`, con el motivo, porque sin la protección el plan escribiría esas celdas y con la fila tapada no se sabe con qué. Con una regla de `aplicabilidad` sobre la columna dependiente, la imputación respeta su universo como las demás acciones por celda: no llena las filas declaradas fuera, que son vacío por diseño, y `n_afectadas` las excluye.

Cuando el determinante o el dependiente es una columna protegida, se tapa por construcción todo ese lado del mapa: el texto usa `[valor protegido]` y las columnas numéricas, `Date` y `POSIXct` usan `NA`. El lado no protegido conserva el resultado de los barridos generales, incluido el piso por contenido. Los demás parámetros de una acción sobre una columna protegida tampoco publican objetos de clase `Date` o `POSIXct`.

Con un lado entero tapado, todas sus filas quedan tapadas y `aplicar()` sólo puede resolverlas con datos que traigan la relación y el soporte declarado. Una entrega parcial —las filas a completar y un par de cada clave— queda `fallida`, con el motivo de que los datos no traen un par de esa clave que la resuelva. Una contradicción en una clave que no hay que completar —una fila con el dependiente presente que cambió de valor— no se detecta: el plan no publica esa clave para compararla, la acción sale `ejecutada` y lo que imputa es correcto sobre los datos recibidos. Es una limitación de esta representación tapada.

Sobre una columna protegida, `convertir_sentinelas_numericos` publica el catálogo de centinelas del paquete tal cual —es vocabulario, no un dato— y tapa el resto. Lo tapado se marca con `valores_enmascarados = TRUE` y `aplicar()` lo resuelve sobre los datos que recibe: son los rellenos de la columna, el dígito repetido que ocupa el lugar de un documento. El plan ofrece la acción así sólo si esa resolución convierte, sobre los datos del plan, exactamente las celdas que estima; si no, la acción queda `bloqueada` y la justificación dice cómo ejecutarla. Lo mismo en una columna que no es personal cuando el centinela declarado aparece en una protegida —el `99999999` que se repite en la cédula de una tabla de movimientos—: no se publica, se declara y se resuelve con los rellenos de esa columna, o la acción queda `bloqueada`. Un centinela declarado que no aparece en ninguna protegida se publica: es la declaración del usuario.

`marcar_filas_duplicadas` añade dos columnas. `.fila_duplicada` reproduce la semántica de `duplicated()` y marca sólo las apariciones posteriores; `.grupo_duplicado` identifica a **todas** las filas que participan en cada grupo de contenido idéntico. Las acciones de duplicados comparan las filas con la misma igualdad que el perfil, también con columnas `integer64`, matrices, `data.frame` anidados o `POSIXlt`; una tabla con columnas de lista deja las acciones bloqueadas. Las de columnas duplicadas comparan el contenido como el perfil: un factor con nivel `NA` y el mismo factor sin ese

nivel, un POSIXlt y su POSIXct, o dos factores sin orden con los mismos niveles en otro orden, son la misma columna también al aplicar. Si alguna columna del grupo se llama como otra de la tabla, no se propone acción: el hallazgo queda en hallazgos_sin_accion_por_columna_ambigua, con el remedio. Hay una decisión por grupo de columnas iguales, no por par: marcarlo, o eliminar las repetidas y conservar la primera del grupo. columna_1 nombra la que se conserva, y columna_2 y eliminar, todas las repetidas; el atributo columnas_duplicadas_marcadas lleva una fila por repetida, junto a la primera. Marcar no elimina filas. Con las dos columnas incluidas, las marcas distinguen la primera aparición de las repetidas, pero no a las repetidas entre sí: en un grupo de tres o más filas un perfil posterior sigue contando duplicadas. Para saber si los duplicados siguen en la tabla hay que quitar las columnas de marca antes de perfilarla.

Al volver a planificar sobre lo aplicado, el plan no propone acciones sobre las marcas que agrega —.fila_duplicada, .grupo_duplicado, .ausente_<columna> y .outlier_<columna>—: sus hallazgos quedan en hallazgos_sin_accion con ese motivo, porque el NA de .grupo_duplicado significa que la fila no participa de ningún grupo y no una ausencia. Una acción cuya columna de marca ya existe en los datos queda bloqueada: aplicar() no puede volver a crearla.

El orden operativo se aparta deliberadamente de la secuencia dimensional frescura–completitud–exactitud–consistencia–unici sugerida por el marco. Primero marca duplicados sin borrar, luego normaliza ausencias y texto, y deja los cambios de esquema para el final. Esto evita perder la evidencia original, permite imputar antes de convertir tipos y mantiene identificables las columnas durante todo el plan. Las eliminaciones nunca se activan por defecto, por lo que deduplicar temprano no puede hacer desaparecer registros. destructiva también marca una conversión que pierde representación, aunque no elimine filas o columnas. El consentimiento permitir_eliminacion sólo se exige para las estrategias que efectivamente retiran filas o columnas; una conversión destructiva requiere que el usuario la active explícitamente y deja su pérdida cuantificada en el registro.

Marcar o eliminar ausentes va después de normalizarlos, no antes. Una columna con "N/A" y "sin dato" tiene ausentes disfrazados que convertir_ausencias_textuales convierte en NA reales; si la marca corriera primero, quedaría una columna .ausente_x que dice FALSE en filas que terminan en NA —una afirmación falsa sobre los datos, publicada con las dos acciones en ejecutada y sin error—. El plan ordena la marca después de la conversión, así que el idioma documentado plan$aplicar <- plan$recomendada no produce esa marca falsa. Por la misma razón, eliminar_filas_ausentes también va después: eliminar antes deja sin eliminar las filas cuyo ausente todavía estaba disfrazado.

convertir_ausencias_textuales convierte exactamente los textos de parametros$valores, comparados sin mayúsculas ni espacios al borde. El plan arma la lista con los marcadores que el perfil detectó y los que se declararon en cadenas_ausencia; editarla es una decisión del usuario: quitar un texto lo conserva y agregar uno lo convierte, como si se hubiera declarado. Con protección, la lista y la declaración guardan su valor también sobre una columna protegida: un marcador del catálogo —sin dato— o uno declarado —SIN REGISTRO— no es el dato de nadie, y la declaración ya se publica en meta$cadenas_ausencia. Tapados, saldrían [valor protegido] y aplicar() registraría fallida la acción recomendada.

convertir_sentinelas_numericos convierte los números de parametros$valores. El plan arma la lista con los centinelas que el perfil contó en esa columna y con los que se declararon en sentinelas_numericos: un 999 dentro de una numeración de 1 a 2000 no es un centinela para perfilar(), y la acción no lo convierte. Así, aplicada a los datos del perfil, cambia las celdas que n_afectadas anuncia. La lista se lee de la evidencia del hallazgo, que muestra las marcas más frecuentes: si la evidencia no alcanza a nombrar todo lo contado —una columna de texto con muchas marcas de ausencia distintas—, la lista es el catálogo entero y conviene revisarla antes de marcar la

acción. También es el catálogo entero cuando el perfil se hizo con `proteger_datos_personales = TRUE`. Editarla es una decisión del usuario, como en la acción de las ausencias textuales.

Value

`planificar_limpieza()` devuelve un data frame de clase `plan_limpieza`. `aplicar()` devuelve una lista de clase `resultado_limpieza` con `datos`, `registro`, `plan_aplicado`, el `plan` sincronizado y `eliminados`. El `registro` conserva `estado` (`ejecutada` o `fallida`), `error`, `n_no_reversibles` y la `justificacion` de cada acción seleccionada, incluso cuando una falla y las siguientes continúan. `n_codificacion_normalizada` cuenta, por acción, las celdas cuya **marca de codificación** cambió: es el caso de una acción de texto sobre celdas `latin1`, cuyo valor se conserva y cuyos bytes no. No cuenta una celda sin marca cuyos bytes ya eran UTF-8 válido ni un texto ASCII, así que no depende del locale. `n_no_reversibles` cuenta las celdas cuyo VALOR se perdió y no se puede recuperar desde el resultado: un centinela `-999` que pasa a ausencia, un extremo recortado a su límite, un marcador de ausencia convertido, o un número que se redondeó al convertirlo. Las acciones que **normalizan la escritura** —recortar espacios, quitar un invisible de transporte, reemplazar separadores o cambiar mayúsculas— dejan el valor en su lugar y no cuentan, **salvo cuando la normalización fusiona valores que eran distintos**: ahí lo que los separaba no queda en ningún lado y esas celdas sí se cuentan. El criterio se mide sobre el resultado, no por el nombre de la acción: quitar un guion suave de `PRO<U+00AD>DUCTO-A` no pierde nada, y quitar un espacio de ancho cero que distinguía dos claves fusiona las dos. Si una acción seleccionada no produce ningún efecto, se registra como `fallida` con el motivo y su copia no se incorpora al resultado. La comprobación del efecto observa `n_cambiadas` aunque `n_afectadas` sea `NA` o haya sido editado: una acción seleccionada que no cambia nada queda `fallida`, porque la ausencia de efecto es observable sin depender de la estimación. Si una columna de entrada es un factor, las acciones que transforman su texto devuelven una columna `character`: no se reconstruyen los niveles originales, porque una limpieza puede introducir valores nuevos. Los atributos de cada columna —la etiqueta de variable que deja `haven`— sobreviven a las acciones que la reescriben, a las conversiones de tipo y a las que quitan filas; quitar filas de un `data.table` conserva su clave.

See Also

`perfilar()`, `guiar_limpieza()`, `detectar_dependencias()`

Examples

```
datos <- data.frame(categoria = c(" A", "S/D", "B"))
perfil <- perfilar(datos)
plan <- planificar_limpieza(perfil, datos)
plan[, c("grupo", "estrategia", "recomendada", "aplicar")]
resultado <- aplicar(plan, datos)
resultado$datos
```

---

plan_perfilado_dbi *Planificar el costo de* `perfilar_dbi()` *antes de pagarlo*

Description

Emite sólo consultas de preparación —leer el esquema y sondear capacidades— y devuelve cuántas consultas emitiría el perfilado completo, de qué clase y con qué alcance sobre la tabla. No escanea datos para decidir el costo. Cuando `politica_costo = "por_cardinalidad"`, una clave estructural exacta puede cerrar la decisión; si no hay una fuente de catálogo utilizable, el plan publica el rango entre omitir y ejecutar la moda; la mediana no entra en ese criterio proporcional. Nunca lanza `COUNT(DISTINCT ...)` para despejar esa incertidumbre. Las fuentes estructurales se resuelven cuando la política necesita la cardinalidad, aunque `estrategia_distintos` no permita medirla. La disponibilidad de la estrategia gobierna la medición, no el conocimiento que ya da el catálogo. Para el `COUNT(DISTINCT)` exacto, la preparación puede consultar además las estadísticas de PostgreSQL (`pg_stats`, `pg_class.reltuples`, `SHOW work_mem` y, desde PostgreSQL 13, `SHOW hash_mem_multiplier`) para estimar el tamaño del hash y avisar un posible derrame. Esa consulta de metadatos no publica cardinalidad medida ni reemplaza la medición posterior. Para la moda y la mediana se publican, además, los atributos `estimacion_derrame_moda` y `estimacion_derrame_mediana`. La moda deriva su `metodo` (`"hash"` o `"sort"`) del primer `Aggregate` de un `EXPLAIN (FORMAT JSON, COSTS OFF)` de la consulta exacta, sin `ANALYZE` ni lectura de datos. La mediana siempre modela un sort y distingue su huella de decisión de la magnitud del tape. Ambas usan `n_validos` de catálogo en el plan, pisos de 32 bytes para tipos fijos y 42 para `numeric`, y dejan el objeto como `no_disponible` cuando el catálogo o el motor no permiten una estimación. Cuando esa lectura de catálogo trae `pg_class.reltuples` positivo, el plan lo reutiliza como una estimación declarada del número de filas. La magnitud y las proyecciones de trabajo de moda y mediana quedan entonces disponibles y dicen explícitamente `estimado_catalogo`; no son duraciones ni mediciones. Un valor cero o negativo —típico de una relación sin `ANALYZE`— conserva el estado `sin dato filas`. Los demás motores conservan ese estado si no tienen una lectura de catálogo ya disponible.

Usage

```
plan_perfilado_dbi(
  conexion,
  tabla,
  universo = c("tabla_completa", "muestra_motor"),
  muestra_motor = NULL,
  muestra = Inf,
  orden_muestra = NULL,
  metricas = .METRICAS_DBI,
  estrategia_distintos = "exacta",
  estrategia_mediana = c("exacta", "aproximada_motor"),
  politica_costo = c("todas", "por_cardinalidad"),
  bloque_muestra = c("con_muestra", "solo_agregados"),
  max_consultas = Inf,
  dialecto = "auto",
  incluir_valores = TRUE,
  tamano_lote = NULL,
  tamano_lote_planos = .TAMANO_LOTE_PLANOS_DBI,
  tamano_lote_distintos = .TAMANO_LOTE_DISTINTOS_DBI,
  instrumentar = FALSE,
  umbral_cardinalidad = .UMBRAL_CARDINALIDAD_COSTO_DBI,
```

```
  max_celdas_muestra = .MAX_CELDAS_MUESTRA,
  max_bytes_muestra = .MAX_BYTES_MUESTRA,
  bloque_filas = NULL,
  max_bytes_procesamiento = .MAX_BYTES_MUESTRA,
  max_bytes_materializacion = .MAX_BYTES_MUESTRA
)
```

Arguments

conexion	Conexión abierta compatible con DBI.
tabla	Nombre de tabla o un objeto aceptado por DBI::dbQuoteIdentifier().
universo	Universo sobre el que se calculan los agregados SQL: "tabla_completa" (por omisión) o "muestra_motor". En el segundo caso, todas las métricas SQL usan la relación muestreada por el motor y no se reemplazan silenciosamente por resultados de la tabla completa.
muestra_motor	Cantidad positiva y finita de filas que el motor debe tomar cuando universo = "muestra_motor". Es obligatorio en ese universo y debe quedar NULL para "tabla_completa"; la función rechaza temprano valores no enteros, no positivos o Inf. Si una vista falla en una fila, el resumen conserva el alcance de la materialización que sí se pudo leer: las columnas legibles se perfilan y la columna rota queda no_disponible; la cobertura no presenta esa lectura parcial como una lectura completa de la vista.
muestra	Cantidad positiva de filas solicitadas para el perfil de muestra que se trae a R, o Inf para traer la tabla entera. Este límite es independiente de universo: en muestra_motor, muestra_motor decide las filas del resumen SQL y muestra decide las filas del bloque perfil_muestra. En tabla_completa, muestra no cambia los agregados. Cuando universo = "muestra_motor", el resumen usa toda la materialización del motor y este argumento solo recorta el diagnóstico en R; por eso meta$materializacion$n_filas puede ser mayor que perfil_muestra$meta$filas. Sin orden_muestra, las filas del bloque en R no son una muestra aleatoria garantizada sino las primeras que devuelva el motor. El límite también alcanza la muestra común con que se buscan dependencias. Use un entero finito para acotar ese trabajo cuando el tiempo no sea la restricción.
orden_muestra	Columnas para ORDER BY. La salida solo declara orden reproducible cuando la combinación es única en toda la tabla. Sin este argumento, DBI no garantiza el orden ni la pertenencia de una muestra limitada, y meta lo declara expresamente. No se usa cuando bloque_muestra = "solo_agregados". Con la lectura por bloques, la identidad de la fuente por bloques gobierna el recorrido y este pedido queda declarado en meta$orden_muestra con el motivo estable de que no gobierna esa via. Si el motor falla al evaluar una columna del ORDER BY, la vía prueba cada componente, recupera las columnas legibles sin ese orden cuando puede y publica alcance_distinto con el motivo de que el orden pedido no se pudo aplicar; no lo presenta como si ninguna columna fallara.
metricas	Selección explícita de grupos de métricas: "validos", "distintos", "moda", "basicos", "mediana" y "desvio". El valor por omisión solicita las seis.

c("validos", "basicos", "desvio") omite distintos, moda y mediana, y "validos" sólo cuenta los no nulos.

estrategia_distintos

Procedencia explícita para n_distintos: "exacta" (por omisión) emite COUNT(DISTINCT); "aproximada_motor" usa una función nativa aceptada por el motor y deja la métrica en no_disponible si no existe; "catalogo" lee pg_stats.n_distinct en PostgreSQL y publica el resultado como estimado_catalogo, nunca como medición, cuando universo = "tabla_completa". En muestra_motor queda no_disponible, porque el catálogo describe la relación entera y la corrida mide un subconjunto; y "omitida" no emite ninguna consulta. No hay repliegue automático entre estrategias. El resultado publica estrategia_solicitada, estrategia_resuelta y estado en meta$estrategia_distintos, y las dos primeras también en resumen_tabla$sql. En pg_stats, un valor positivo es el conteo estimado y uno negativo es una fracción de las filas. Cuando la relación tiene descendientes se elige inherited = TRUE, porque esa fila describe lo que lee una consulta sin ONLY; una relación sin hijas usa su única fila propia. Las fracciones se convierten con la suma de pg_class.reltuples de la jerarquía. Si no hay una fila utilizable —por ejemplo, antes de ANALYZE— o hay ambigüedad, la métrica queda no_disponible, no en cero.

estrategia_mediana

Preferencia para resolver mediana: "exacta" (por omisión) o "aproximada_motor". La sonda prueba siempre primero una forma nativa exacta consolidada, luego una forma exacta por columna y deja las funciones nativas aproximadas para el final. Por eso esta opción describe la estrategia habilitada, no garantiza el método ejecutado. El método que efectivamente corrió está en resumen_tablasqlmetodo, en la fila de metrica = "mediana" de cada columna, y es la única fuente por columna. meta$estrategia_mediana publica la resolución de la tabla: lo pedido (estrategia_solicitada), la forma que el motor aceptó para la tabla (estrategia_resuelta), su estado y su motivo. No es el método de cada columna: una columna que no puede usar esa forma —un float8 de PostgreSQL, que la deja por la lectura exacta, o una columna con extremos— corre por otra (dos_consultas en la tabla completa, la CTE de ventanas en la muestra) y lo declara en su fila de resumen_tabla$sql, aunque meta$estrategia_mediana siga nombrando la consolidada. El estado de meta$estrategia_mediana describe la forma, no la fila: en universo = "muestra_motor" la fila de cada columna queda estimado. Las columnas estrategia_solicitada, estrategia_resuelta y estado_estrategia de resumen_tabla$sql son las de n_distintos (meta$estrategia_distintos) repetidas en todas las filas: en la fila de la mediana no describen la mediana. Una mediana resuelta por una forma exacta queda estado = "calculado" y error_esperado = "no_aplica", aunque se haya pedido "aproximada_motor"; sólo una aproximación ejecutada queda estado = "estimado". universo = "muestra_motor" combinado con "aproximada_motor" se rechaza temprano.

politica_costo Política optativa para las métricas caras. El valor por omisión, "todas", conserva moda y mediana para todas las columnas solicitadas. "por_cardinalidad" resuelve primero las fuentes estructurales y mide valores válidos y distintos sólo cuando hace falta y la estrategia lo permite. Luego omite, por columna, sólo la moda cuando la proporción de distintos alcanza umbral_cardinalidad; la mediana se conserva porque las mediciones disponibles muestran que su costo

depende de las filas y no de la cardinalidad. Los únicos valores aceptados son `"todas"` y `"por_cardinalidad"`; no hay alias históricos.

`bloque_muestra` Qué bloques se solicitan: `"con_muestra"` (por omisión) calcula también `perfil_muestra`, o `"solo_agregados"` omite su lectura y devuelve sólo los agregados SQL. La decisión vale para los dos universos: con `muestra_motor`, los agregados siguen usando la relación muestreada, pero no se abre el spool ni se leen filas; la corrida publica el estado `no_solicitado` y `meta$bloques$filas_vistas = 0`. La segunda opción no cambia el alcance de esos agregados: eso lo decide `universo`.

`max_consultas` Presupuesto declarado de consultas. Al agotarse, las métricas restantes quedan en `no_disponible` con ese motivo. Cuenta toda sentencia que el perfilado manda al motor —la preparación, las lecturas de catálogo, el recorrido por bloques y, por el spool de la muestra del motor, la preparación que la corrida repite para publicar su plan—, la misma cuenta de `plan_perfilado_dbi()` y de `resumen_tabla$meta$consultas$emitidas`. No cuentan las sentencias de la guarda de transacción (`SAVEPOINT` y, en SQL Server, la pregunta por la transacción abierta antes de cada consulta), ni las lecturas de `pg_stat_statements` de `instrumentar`, ni las que algunos controladores mandan para contestar `DBI::dbGetInfo()` o `DBI::dbIsValid()`: duckdb 1.1 pregunta su versión y manda `SELECT 1`, una o dos veces por corrida. La preparación y el recorrido por bloques se cuentan pero no se omiten: sin ellos no hay perfil. Con `max_consultas = attr(plan, "total_maximo")` ninguna métrica queda afuera por el presupuesto.

`dialecto` Capacidad de acotar filas: `"auto"` la sondea, y `"limit"`, `"top"`, `"fetch_first"`, `"rownum"` o `"portable"` la declaran sin sondeo.

`incluir_valores` Si publica valores de celda. Con `FALSE` no se emiten las consultas de moda, extremos y mediana, y se limpian esos campos en el resumen, la muestra, los bloques, la cobertura y los motivos; también se retiran ejemplos de patrones, familias de acumuladores y cualquier muestra cruda. El mensaje de un error del motor, que cita la celda que lo provocó, se publica sin sus literales, como con `proteger_datos_personales = TRUE`. Los conteos, proporciones y agregados que no son valores de celda pueden seguir publicándose. La elección queda en `resumen_tabla$meta$incluir_valores` y `perfil_muestra$meta`.

`tamano_lote` Cantidad máxima de columnas por consulta consolidada. Se conserva por compatibilidad y, si se informa, fija el tamaño de las dos familias. Para control separado, usar `tamano_lote_planos` y `tamano_lote_distintos`.

`tamano_lote_planos` Cantidad máxima de columnas por consulta de agregados planos. En SQL Server y en columnas `float8` de PostgreSQL, el desvío conserva su cálculo centrado en una segunda pasada del mismo tamaño. El valor por omisión es 20.

`tamano_lote_distintos` Cantidad máxima de columnas por consulta de cardinalidades exactas. El valor por omisión es 2, un supuesto del código medido sobre un PostgreSQL de referencia: el motor hace una sola pasada por la tabla para todos los conteos de una consulta, así que un lote de uno cuesta por columna bastante más que uno de dos; uno de dos cuesta por columna casi lo mismo que uno de cuatro y derrama menos, y los lotes mayores derraman más porque los conteos compiten por `work_mem`. Una sola cardinalidad todavía puede forzar un agregado pesado y derramar mucho más que un lote plano.

`instrumentar` En el plan, si es `TRUE`, cronometra las consultas de preparación. No habilita consultas de datos ni agrega mediciones al objeto devuelto: sus costos siguen siendo predicciones. Por omisión es `FALSE`.

`umbral_cardinalidad`
Proporción entre valores distintos y válidos que activa la omisión de la moda con `politica_costo = "por_cardinalidad"`. El valor por omisión es `0.5` sólo cuando esa política se pide explícitamente; se puede mover en cada llamada. Este argumento no gobierna la mediana: `meta$decisiones_costo` explica la decisión de cada métrica por separado. Para pedir todas las métricas use `politica_costo = "todas"`.

`max_celdas_muestra`
Máximo de celdas que puede contener el bloque `perfil_muestra`. Por defecto es `1000000`; se calcula antes de leer como filas por columnas del esquema. Si reduce la muestra, `perfil_muestra$cobertura_diagnosticos` usa la misma declaración que `perfilar()`, con las celdas solicitadas, el umbral y el tope que mandó. `Inf` desactiva este tope. No modifica los agregados SQL.

`max_bytes_muestra`
Máximo de bytes de la muestra materializada que alimenta `perfil_muestra`. Por defecto es `512 MiB`. En `muestra_motor`, el límite se cruza con el presupuesto de materialización y se respeta antes de cada `dbFetch(n)`: cuando la cota de tipos es demasiado conservadora se lee una fila inicial, se mide lo que ocupa en la materialización y los bloques siguientes se piden con el doble de margen sobre la fila más grande medida; el spool nunca pasa el tope. Si reduce la muestra, `meta$muestreo`, `meta$materializacion` y `cobertura_diagnosticos` informan las filas solicitadas, efectivas y cuál tope mandó. `meta$muestreo$filas_por_bytes` publica la cantidad obtenida de la medicion y `meta$muestreo$filas_por_bytes_fuente` lo declara. `Inf` desactiva este tope. No modifica los agregados SQL.

`bloque_filas` Entero positivo que activa la fuente por bloques de `tabla_completa`. Cada bloque se obtiene con un unico result set (`dbSendQuery()` + `dbFetch(n = bloque_filas)`) y se absorbe antes de liberar la entrada. `NULL` conserva el plan historico. El plan publica la proyeccion, el orden, los bloques previstos y el costo declarado sin leer las filas. Cuenta la preparación, las pruebas de la fuente, el recorrido, la segunda pasada del localizador, la moda exacta de binarios y decimales anchos, la media de los decimales en el decimal y la lectura de `perfil_muestra`; la mediana de un decimal cuyas centrales son dobles distintos, su desvío centrado y la moda de un entero leído como doble los piden los datos y van al extremo superior.

`max_bytes_procesamiento`
Límite de bytes del estado retenido por el procesamiento por bloques y por la lectura del spool en R. Se comprueba antes de publicar un bloque o una pasada; si la materialización cabe pero el objeto que R necesita para perfilarla supera el límite, el perfil se descarta y se emite el mismo aviso de alcance degradado, con `tope_que_mando = "procesamiento"`. `Inf` desactiva este límite.

`max_bytes_materializacion`
Presupuesto de bytes del spool externo de `muestra_motor`. Se mide por chunk antes de cada escritura, dejando espacio para el trailer; si no alcanza, se publica `spool_presupuesto_excedido` y se conserva la parte que entró como muestra recortada, con `muestra_inestable:presupuesto_materializacion`. En

muestra_motor, el primer bloque se acota antes de pedirlo y los siguientes se adaptan a los bytes medidos; el resultado conserva meta$bytes_muestra y meta$tiempos. Inf desactiva este límite.

Details

estrategia_distintos declara la procedencia de n_distintos antes de la corrida y conserva por separado lo pedido, lo resuelto y el estado. No hay auto: "exacta" es el valor por omisión, "aproximada_motor" queda no_disponible si el motor no ofrece una función aceptada, "catalogo" lee pg_stats.n_distinct en PostgreSQL y publica una estimación con estado estimado_catalogo cuando universo = "tabla_completa", y "omitida" no emite el agregado. Con universo = "muestra_motor", catalogo queda no_disponible: sus estadísticas describen la relación entera y no el subconjunto de la corrida. Un n_distinct positivo es un conteo y uno negativo una fracción de las filas. Si la relación tiene descendientes se usa la fila inherited = TRUE, que describe la consulta sin ONLY; si no tiene hijas se usa la única fila propia. La fracción se convierte con la suma de pg_class.reltuples de la jerarquía. Si falta una estimación utilizable —por ejemplo, antes de ANALYZE— o hay filas ambiguas, se conserva no_disponible, nunca cero. fuente_cardinalidad_costo sigue siendo independiente y sólo describe el número usado por la política de costo cuando esa política se pide.

El plan previo no proyecta segundos para COUNT(DISTINCT): no lee los datos y, por tanto, no tiene una referencia medida. La única referencia honesta es el primer lote de distintos de la corrida, pero obtenerla cuesta una consulta que este planificador no emite. Durante perfilar_dbi(), cuando hay más de un lote y la instrumentación está activa, esa primera medición se multiplica por la cantidad de lotes y se publica en resumen_tabla$meta$costo_distintos. El aviso temporal llega después del primer lote y antes del segundo; con un solo lote se declara que no hay nada que proyectar.

La memoria del procesamiento no se estima: no escala de forma predecible con las filas ni con las celdas, y eso se midió. El atributo memoria_procesamiento conserva esa declaración, la magnitud conocida del trabajo (filas, celdas y pares_texto) y referencias medidas de otras corridas. Esas referencias no son una predicción para la tabla del plan: traer costó aproximadamente 0,13 GB por millón de filas y procesar en R aproximadamente 1,0-1,5 MB por cada mil filas, pero esta segunda cifra varió 1,62x entre tablas de la misma magnitud.

Ver todas las filas y tener todas las filas en memoria no son lo mismo. En corridas de referencia, 4,5 millones de filas entraron en 0,6 GB y tardaron 25 segundos, mientras que procesar 4,5 millones ocupó aproximadamente 7 GB y procesar 12,8 millones aproximadamente 19 GB. El problema observado está en el procesamiento en R, no en la red ni en el motor.

Para universo = "muestra_motor", el plan declara una única selección que se pagará para cerrar el spool externo de sesión cliente. attr(plan, "materializacion") publica pagado = FALSE, backend, versión, presupuesto, result set, fetches esperados, filas y bytes; attr(plan, "pasadas") declara que valor, índice y LSH leerán el mismo spool. La referencia medida para justificar este costo es PostgreSQL 16, 2 millones de filas: 10.000, 100.000 y 500.000 filas dieron 0,448/1,684/5,598 s con spool frente a 0,814/2,178/3,265 s reordenando cada pasada; el cruce está entre 100.000 y 500.000, y la elección prioriza identidad. Con el bloque de filas, esa vía no corre los agregados SQL de la muestra: el plan cuenta la preparación, la estimación de bytes por fila del catálogo, el conteo del universo cuando la preparación no lo hace, la selección de la muestra —dos en el extremo superior, si un TABLESAMPLE corto se reintenta— y la preparación que la corrida repite para publicar su plan en meta$plan.

Value

Data frame de clase plan_perfilado_dbi con clase_consulta, n_consultas, n_consultas_max y alcance, y los atributos total, total_minimo, total_maximo, total_lotes_rechazados, columnas, columnas_numericas, dialecto, consultas_emitidas, metricas, metricas_ejecucion, politica_costo, estrategia_distintos, fuente_cardinalidad_costo, moda_guardian, mediana_consolidada, filas, filas_fuente, estimacion_filas, proyecciones, mediana_escalar, tamano_lote_planos, tamano_lote_distintos, estimacion_derrame, estimacion_derrame_moda, estimacion_derrame_mediana, celdas, memoria_procesamiento, max_celdas_muestra, max_bytes_muestra, tope_muestra y muestreo, y, cuando se pide distintos, supuesto_costo_distintos. memoria_procesamiento siempre tiene estado = "no_estimada": no es una estimación de consumo, sino la declaración de su ausencia, el motivo, la magnitud del trabajo y referencias medidas de otras corridas. El atributo estimacion_derrame es independiente: sólo describe la estimación del hash en el motor para COUNT(DISTINCT) y no la memoria del procesamiento en R. estimacion_derrame_moda y estimacion_derrame_mediana describen, respectivamente, el nodo real de la moda y el sort de la mediana. Sus lotes deciden por el máximo de columna; en una mediana consolidada, estado_io_total_bytes es sólo la suma informativa de los tapes. muestreo declara si la forma muestreada se pudo construir sin emitir una consulta de datos. En universo = "muestra_motor", cuando su estado es "no_disponible", el plan excluye las métricas SQL de esa muestra y supuesto conserva el motivo. Cuando se pide bloque_muestra = "solo_agregados", también conserva ese valor en el atributo bloque_muestra y no incluye la fila de la lectura de muestra.

Los atributos max_celdas_muestra, max_bytes_muestra y tope_muestra declaran la cota que se aplicará al bloque perfil_muestra. El plan no lee datos: resuelve la cota de celdas con el ancho del esquema y declara que la cota de bytes requiere una sonda de hasta cien filas durante la corrida. Si la muestra pedida supera la cota de celdas, tope_muestra y print() lo dicen antes de correr. La fila de consulta de muestra conserva su conteo separado de los agregados; los topes no cambian ninguna consulta SQL de resumen.

El costo no se declara como un número sino como un rango: total es el extremo inferior, que supone que la política omite la moda cuya cardinalidad no se conoce, y total_maximo el superior, que supone que la ejecuta. La mediana queda fuera de ese supuesto proporcional y se cuenta según las columnas numéricas solicitadas. Ambos incluyen la preparación y el perfilado; el rechazo de lotes puede agregar las sondas de bisección declaradas por total_lotes_rechazados. El costo real cae entre los extremos cuando universo es "tabla_completa" o attr(plan, "muestreo")$estado es "disponible"; en "no_sondeado" la forma sólo se construyó localmente y el intervalo queda condicionado a que la sonda de la corrida la acepte. Si la forma muestreada no se puede construir, el plan declara ese caso, excluye sus métricas del rango y attr(plan, "supuesto") dice por qué. El extremo superior incluye además, con universo = "muestra_motor" y una forma TABLESAMPLE, el reintento del resumen cuando la muestra vuelve con menos de la mitad de lo pedido, que depende de las páginas que sortee el motor. La cuenta es la de max_consultas y la de resumen_tabla$meta$consultas$emitidas: toda sentencia que el perfilado manda al motor, salvo las de la guarda de transacción, las lecturas de pg_stat_statements de instrumentar y las que algunos controladores mandan para contestar DBI::dbGetInfo() o DBI::dbIsValid() (duckdb 1.1, una o dos por corrida). Con max_consultas igual a total_maximo ninguna métrica queda afuera por el presupuesto; con total, sólo las consultas que piden los datos pueden quedar afuera.

Cuántas consultas se emiten no dice cuánto cuestan: catorce consultas sobre dos millones de filas son mucho más trabajo que doscientas sobre mil. Por eso el plan estima además la magnitud, y la estima en sus dos mitades, porque el reloj de una corrida no lo decide siempre el motor.

La del motor va en `filas_leidas` (cuántas filas habría que leer) y `ordenaciones_completas` (cuántas veces habría que ordenar la tabla entera), y se resume en `magnitud_motor`. La del cliente va en `columnas_texto` y `pares_texto` —cuántos pares de formas podría comparar en R el detector de vocabulario sobre la muestra— y se resume en `magnitud_texto`. `magnitud` es la mayor de las dos: `"baja"`, `"media"`, `"alta"`, o `"desconocida"` si no se conoce el número de filas. `supuesto_costo` dice de dónde sale cada cuenta.

El plan previo no publica duraciones, CPU, ni filas o bytes medidos, ni agrega una proyección temporal de `COUNT(DISTINCT)`. Puede publicar filas estimadas por `pg_class.reltuples` y proyecciones de trabajo de moda/mediana, siempre rotuladas como estimación de catálogo y no como medición. Aunque el plan de una corrida conserve el atributo `supuesto_costo_distintos`, la medición y la proyección sólo aparecen en `resumen_tabla$meta$costo_distintos`, después de ejecutar el primer lote. El atributo sólo declara por qué esa proyección no existe antes de correr; no es una duración ni una estimación temporal.

Si se pide `politica_costo = "por_cardinalidad"`, el plan busca primero una garantía estructural o una fuente de catálogo. Si la fuente queda desconocida, no emite un agregado para aclararla: `n_consultas` omite la moda y `n_consultas_max` deja abierto el camino que la ejecuta. La mediana se conserva porque su costo medido es plano frente a la cardinalidad y está gobernado por las filas. La corrida mide `distintos` sólo si la política lo necesita. La política por omisión es `"todas"`: el paquete no elige por el usuario. Una fuente estructural se resuelve aunque la estrategia de distintos este omitida o no disponible; esta ultima solo gobierna si se puede medir. El catalogo de la clave primaria se consulta siempre para conservar esa respuesta en `resumen_tabla$meta$clave`; es una lectura de metadatos y no un recorrido de la tabla.

Contar sólo el motor daría juicios falsos con números ciertos: una tabla chica con una columna de geometría en texto pide pocas lecturas —magnitud `"baja"` del lado del motor— y puede tardar mucho más de lo que eso sugiere, porque el trabajo está en la comparación de formas, que no es una lectura de fila. El método de impresión muestra las dos mitades, avisa cuando la magnitud es alta y nombra las palancas para acotarla, que no son las mismas de un lado que del otro.

See Also

`perfilar_dbi()`

Examples

```
if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "ejemplo", data.frame(id = 1:10, valor = 11:20))
  plan_perfilado_dbi(con, "ejemplo")
  DBI::dbDisconnect(con)
}
```

---

propiedades_regla *Propiedades declaradas de una regla de evaluación*

Description

Devuelve, en una tabla, lo que una regla declara: a qué métricas se engancha, en qué nivel evalúa, qué desenlace produce y **qué umbrales usa**. Los umbrales viajan aparte de la condición justamente para esto: encerrados en el *closure* quedarían invisibles y obligarían a escribir otra regla para mover un número.

Usage

```
propiedades_regla(regla)
```

Arguments

`regla`	Objeto creado por `regla_evaluacion()`.

Details

Es la contraparte de `propiedades_metrica()`, que describe métricas. Un umbral pertenece a una regla, no a una métrica, así que no cabe allí.

Value

Data frame con una fila por propiedad: `propiedad`, `valor`.

See Also

`regla_evaluacion()`, `propiedades_metrica()`, `evaluar()`

Examples

```
regla <- regla_evaluacion(
  "cobertura minima",
  function(x, minimo) x >= minimo,
  umbrales = list(minimo = 0.9)
)
propiedades_regla(regla)
```

---

proponer_modelo *Proponer un modelo de calidad desde el profiling*

Description

Construye el puente entre el examen de datos y la medición. El resultado no ejecuta métricas: es una tabla editable donde cada fila conserva el hallazgo o diagnóstico que la originó, la justificación, la configuración y una marca `incluir`. `modelo_desde_propuesta()` materializa únicamente las filas que el usuario deja activas.

Usage

```
proponer_modelo(
  perfil,
  datos = NULL,
  relaciones = NULL,
  entidades_relacion = character(),
  max_valores_dominio = 20L,
  max_sugerencias = 100L
)
```

Arguments

perfil	Objeto creado por perfilar().
datos	Datos originales opcionales. Son necesarios para proponer dominios observados, materializar reglas de dependencia funcional e identificar los faltantes disfrazados que cuenta NoNulo.
relaciones	Resultado opcional de detectar_relaciones().
entidades_relacion	Dos nombres de entidad correspondientes a las tablas usadas en relaciones.
max_valores_dominio	Máximo de valores para proponer un dominio por extensión.
max_sugerencias	Máximo de filas devueltas.

Details

Las reglas observadas que pueden sobreajustarse a una entrega —dominios y patrones dominantes— se proponen inactivas. Los controles estructurales directos, como NoNulo, duplicación exacta y dependencias funcionales exactas, se activan. max_sugerencias recorta después de ordenar por prioridad y el objeto declara el total en sus atributos. Dentro de cada prioridad las filas se ordenan por métrica, entidad y atributos según sus bytes —no según la intercalación del locale—: el mismo orden en cualquier sesión.

id_sugerencia se arma con el **contenido** de la sugerencia, nunca con su posición: la métrica y sus atributos en su orden —NoNulo:edad, ReglaIntegridadIntraEntidad:depto,codigo— y, cuando la sugerencia liga dos tablas, cada atributo con su entidad delante (ReglaIntegridadInterEntidad:personas.id,v
Una sugerencia sin atributos lleva sólo la métrica (EntidadDuplicada). Así una columna que se agrega o se quita no cambia el id de las demás, y una regla de regla_evaluacion() que lo nombra sigue midiendo lo mismo en la corrida siguiente: modelo_desde_propuesta() lo usa como nombre de la instancia, que es la metrica_instanciada de medir(), del tablero y del histórico. La entidad de una sugerencia de una sola tabla no entra, porque es el nombre del perfil y cambia con la expresión perfilada; la configuración tampoco: un cambio de patrón o de dominio lo declara el histórico en configuracion_modelo. Dentro de cada nombre se escribe %25, %2C y %23 en lugar de %, , y # —y %2E en lugar de . en una entidad—, y un nombre NA se escribe %NA, de modo que dos contenidos distintos no dan el mismo id. Un nombre que no es UTF-8 válido escribe cada byte alto como %XX —los bytes latin1 de «notaé» sin declarar dan nota%E9—, así que el id es siempre texto UTF-8 o ASCII, el mismo en cualquier locale. Si dos sugerencias tienen el mismo contenido

—una tabla que repite el nombre de una columna—, la segunda lleva `#2` y la tercera `#3`, en el orden de las columnas de la tabla.

Si una columna contiene faltantes disfrazados, `NoNulo` se propone activa, con estado `lista`, y **se mide contándolos**: los valores que el perfil contó como disfrazados —con la misma regla y la misma declaración de `sentinelas_numericos` y `cadenas_ausencia`— se declaran en `valores_nulos` de su `configuracion`, y la justificación lo dice. Sobre cuarenta valores con un `NA` y tres `-999` mide 0,90, no 0,975. Si no se pueden identificar —sin `datos`, la propuesta no tiene de dónde sacarlos—, se mide igual con los valores crudos, y la justificación dice que cuenta sólo los `NA` y que sobrestima la completitud. `analizar()` la mide y `modelo_desde_propuesta()` la materializa como a cualquier otra sugerencia activa. Si la columna se declaró en `columnas_opcionales` de `perfilar()`, la `NoNulo` lleva además `opcional = TRUE` y se mide sobre las celdas presentes, el mismo universo que el perfil: con un `NA` y tres `-999` en cuarenta celdas mide 36/39 = 0,923, que es `1 - prop_faltantes_totales`. Una sugerencia sobre una columna cuyo nombre la tabla repite sale inactiva y con estado `nombre_repetido`: una métrica se liga por nombre, y ese nombre no dice cuál de las columnas mide. `modelo_desde_propuesta()` no la materializa y `analizar()` no la mide hasta que las columnas se renombren.

Value

Data frame S3 de clase `propuesta_modelo`; las columnas de listas contienen la configuración y los vínculos sin convertirlos en texto. Su método `print()` no falla por codificación en ningún locale: la copia que se exhibe convierte lo que trae codificación declarada, declara como UTF-8 los bytes válidos sin declarar —R los escapa como `<U+00F1>` donde el locale no los represente— y deja el resto tal cual mientras R pueda imprimirlo. Bajo un locale UTF-8 la salida es exactamente la de R; bajo uno que no puede representar un carácter difiere a propósito, publicando el punto de código donde R escaparía los bytes. Sólo cuando R no puede imprimir se reintenta con su propio octal, `\377`. Eso no convierte a `print()` en infalible: si una clase suya tiene un método `format()` que da error, ese error le llega sin alterar. Y un `format()` que consulte `Encoding()` verá las marcas de la copia, no las del objeto original.

See Also

`perfilar()`, `detectar_dependencias()`, `modelo_desde_propuesta()`, `modelo()`

Examples

```
datos <- data.frame(
  codigo = rep(1:3, each = 4),
  categoria = rep(c("A", "B", "C"), each = 4),
  valor = c(1:11, NA)
)
propuesta <- proponer_modelo(perfilar(datos), datos)
propuesta[, c("metrica", "origen", "incluir")]
```

`referencial` *Declarar un conjunto de datos referencial*

Description

Un referencial representa conocimiento externo mediante una clave y, de forma opcional, valores asociados a ella. `normalizar` controla la representación usada para emparejar referenciales; no modifica los datos guardados. La declaración de completitud es explícita: `RatioCobertura` exige `completo = TRUE` y un `alcance` explícito; un referencial parcial sólo puede usarse para correctitud.

Usage

```
referencial(
  datos,
  clave,
  valor = character(),
  completo = FALSE,
  alcance = NULL,
  nombre = NULL,
  normalizar = TRUE
)
```

Arguments

`datos`	Tabla de referencia. Se conserva una copia ordinaria de R.
`clave`	Columnas que identifican unívocamente cada fila. Puede contener varias columnas.
`valor`	Columnas asociadas que pueden contrastarse en correctitud débil. Es opcional.
`completo`	Si la tabla cubre todo el universo declarado. Es `FALSE` por omisión.
`alcance`	Descripción obligatoria cuando `completo = TRUE`.
`nombre`	Nombre legible del referencial. Si se omite, usa el nombre del objeto de entrada o `"referencial"`.
`normalizar`	`TRUE`, `FALSE`, `"amplio"`, un perfil de `normalizacion()` o una lista nombrada por columna. `TRUE` es el valor predeterminado.

Details

La clave no admite ausentes y debe identificar cada fila de forma única. `valor` no puede repetir columnas de `clave` y representa atributos que se contrastan en correctitud semántica débil. El constructor copia la tabla y no consulta fuentes externas.

Value

Objeto de clase `referencial`: una lista con la tabla de referencia en `datos`, los nombres de columna `clave` y `valor`, el indicador `completo` -si la referencia agota el dominio-, el `alcance` declarado o `NA`, el `nombre`, y el perfil de normalización resuelto en `normalizar`. Declara la referencia; no compara nada todavía.

See Also

metricas_referencial(), instanciar(), detectar_relaciones()

Examples

```
padron <- referencial(
  data.frame(codigo = c("01", "02"), departamento = c("Artigas", "Canelones")),
  clave = "codigo", valor = "departamento",
  completo = TRUE, alcance = "departamentos del Uruguay"
)
padron
```

---

reglas_evaluacion *Reglas y perfiles de evaluación*

Description

Una regla aplica una condición a los resultados de una o más métricas instanciadas. Un perfil reúne reglas y su evaluación es la media aritmética simple de las evaluaciones de esas reglas; no es un índice de dimensión ni un índice global de calidad.

Usage

```
regla_evaluacion(
  nombre,
  condicion,
  metricas = NULL,
  proporcion_minima = NULL,
  desenlace = NULL,
  umbrales = list()
)

perfil_evaluacion(nombre, ...)

perfiles_madurez(metricas = NULL, umbrales = NULL)
```

Arguments

nombre	Nombre de la regla o del perfil.
condicion	Función que recibe el vector resultado de las medidas seleccionadas, en el orden de la tabla, y debe devolver un vector lógico sin ausentes de la misma longitud. Puede declarar un segundo argumento orientacion para recibir el metadato homónimo de cada medida; las funciones existentes de un argumento siguen siendo válidas. No modifica las medidas.
metricas	Nombres de métricas instanciadas a las que se aplica la regla, es decir, valores de la columna metrica_instanciada. NULL, el valor predeterminado, aplica la condición a todas.

`proporcion_minima`
: `NULL`, para conservar una regla por medida, o un número entre `0` y `1` que declara la proporción mínima de medidas que deben cumplir `condicion`. En este segundo caso la regla es agregada: el umbral queda guardado en el objeto y `evaluar()` publica la proporción, el veredicto y el universo de medidas que la produjo.

`desenlace`
: `NULL`, para limitar la regla a evaluar, o `"suprimir"` para declarar que las medidas que no cumplen `condicion` no deben publicarse. No existe un desenlace predeterminado. Suprimir tapa el **valor** de esas medidas en cada salida; no es supresión complementaria: un agregado suprimido se puede recalcular de sus medidas crudas si se publican al lado —por ejemplo, en un histórico que acumula también la medición—.

`umbrales`
: Vector numérico con nombres, estrictamente creciente y en `[0, 1]`. `NULL` conserva los tres perfiles incluidos de fábrica.

`...`
: Reglas creadas por `regla_evaluacion()` o una única lista que las contenga.

Details

`perfiles_madurez()` crea por omisión los perfiles `Básico`, `Intermedio` y `Avanzado` de AGESIC, con condiciones estrictas `> 0.5`, `> 0.7` y `> 0.9`. El argumento `umbrales` permite construir otra familia con nombres y cortes crecientes propios sobre las mismas métricas instanciadas. Los cortes son umbrales sobre el **resultado agregado** de cada métrica —una proporción en `[0, 1]`, lo que deja `agregar()`—: sobre las medidas por fila de `medir()`, que valen `0` o `1`, los tres cortes son la misma condición y los tres perfiles dan la misma cifra, sin aviso. Es lo que hace `analizar()`, que evalúa la medición agregada.

`regla_evaluacion()` almacena la función sin ejecutarla. `evaluar()` selecciona las medidas mediante `metricas`, llama una vez a `condicion` y rechaza resultados que no sean lógicos, que tengan otra longitud o que contengan `NA`. Si `proporcion_minima` no es `NULL`, calcula sobre esos mismos lógicos la proporción que cumple y la compara mediante `>=` con el umbral declarado; no pondera medidas ni construye un puntaje global. Las evaluaciones cuyas reglas no declaran `desenlace` conservan su estructura anterior. Cuando una regla declara `desenlace = "suprimir"`, `evaluar()` añade un plan trazable con una fila por medida incumplida y por regla; no modifica la medición ni los datos que la originaron. La función expresa un criterio de evaluación; no es un método de medición ni recibe el data frame original. Si ningún nombre de `metricas` coincide, el error enumera tanto los nombres solicitados como las métricas instanciadas disponibles, que normalmente tienen la forma `MetricaEspecifica@entidad.atributo`.

Value

`regla_evaluacion()` devuelve una `regla_evaluacion`; `perfil_evaluacion()` devuelve un `perfil_evaluacion`; y `perfiles_madurez()` devuelve una lista de perfiles.

See Also

`medir()`, `evaluar()`, `perfiles_madurez()`

`regla_evaluacion()`, `comparar_evaluaciones()`, `historico_calidad()`

`evaluar()`, `detectar_deriva_calidad()`

Examples

```
regla <- regla_evaluacion("Completitud suficiente", function(x) x > 0.9)
regla_70 <- regla_evaluacion(
  "Al menos 70 %", function(x) x > 0.9, proporcion_minima = 0.7
)
regla_publicacion <- regla_evaluacion(
  "Medida publicable", function(x) x > 0.9, desenlace = "suprimir"
)
perfil <- perfil_evaluacion("Operativo", regla)
madurez <- perfiles_madurez("NoNulo")
propios <- perfiles_madurez(
  "NoNulo", c(Exploratorio = 0.3, Operativo = 0.65, Consolidado = 0.85)
)
names(madurez)
names(propios)
perfil$nombre
```

---

relaciones_coleccion *Buscar claves foráneas candidatas entre pares declarados*

Description

Corre `detectar_relaciones()` sobre los pares de tablas que se declaren, y devuelve las relaciones candidatas junto con la cobertura de los pares que no se pudieron comparar.

Usage

```
relaciones_coleccion(
  coleccion,
  pares,
  muestra = 10000,
  umbral_cobertura = 0.9,
  orden = NULL,
  tope_cache_mb = 512,
  columnas_candidatas = NULL,
  podar = FALSE,
  tope_memoria_mb = 512
)
```

Arguments

`coleccion`	Objeto creado por `coleccion()`.
`pares`	Data frame con columnas `tabla_1` y `tabla_2`. Cero filas es una declaración válida de «ningún par».
`muestra`	Filas traídas por tabla para comparar, y **el mismo tope con que se calcula cada cobertura**: se reenvía a `detectar_relaciones()`, así que la evidencia publicada y la cobertura salen de las mismas filas, también con `muestra > 1e5`. Sin

reenviarlo regiría el tope por omisión de esa función, y la fila se contradiría a sí misma: `n_valores_comunes` contado sobre la lectura entera junto a una cobertura calculada sobre un submuestreo que ningún campo declara.

`umbral_cobertura`
: Cobertura mínima para informar una relación.

`orden`
: Columnas para `ORDER BY`: un vector de texto que se aplica a todas las tablas, o una lista nombrada por identificador de tabla. Sin él la lectura no es repetible y el objeto lo declara. Ordena la lectura y no la recorta: una tabla sin `columnas_candidatas` se lee con todas sus columnas, y a una con candidatas se le suman las de orden sólo para ordenar.

`tope_cache_mb`
: Presupuesto de memoria para la caché de lecturas, en megabytes. Al agotarse se sigue leyendo sin cachear y `meta` lo declara.

`columnas_candidatas`
: Lista nombrada por identificador de tabla. Cada vector declara las columnas que pueden participar; una tabla no nombrada conserva todas sus columnas. Un nombre se busca con la grafía exacta y, si no está, sin distinguir mayúsculas —como `orden` y como `orden_muestra` de `perfilar_dbi()`—; uno que no existe da `lupa_error_argumento_dbi`, la clase de todo argumento mal formado de esta función.

`podar`
: Si se aplican las podas que cambiarían lo informado —tipos incompatibles y cardinalidades imposibles—, tal como en `detectar_relaciones()`. `FALSE` por omisión; la poda cierta por rangos disjuntos del universo se aplica siempre porque no cambia ninguna fila.

`tope_memoria_mb`
: Presupuesto de memoria para resultados de relaciones, en megabytes. Al agotarse, las combinaciones pendientes se declaran en `cobertura_podas` y los pares pendientes en `cobertura_pares`.

Details

Los pares se declaran, igual que la frontera. Con mil tablas hay casi un millón de pares dirigidos, así que explorar todos no es una opción cara sino una opción imposible. `estimar_costo_coleccion()` permite verlo antes.

Cada tabla se lee una sola vez. Los pares repetidos se descartan, los pares de una tabla consigo misma se rechazan, y las lecturas se guardan en una caché con presupuesto de memoria declarado: dos pares que comparten una tabla no la leen dos veces. `meta$lecturas` deja el SQL exacto, la vía usada, las filas leídas y el momento de cada lectura.

El orden no se supone. Sin `orden`, una lectura acotada devuelve un subconjunto arbitrario y el resultado declara `meta$estable = FALSE`. Declarar columnas de orden hace la lectura repetible.

Una columna que la sesión no puede escribir no se lleva la tabla. Con odbc bajo el locale `C` -un `Rscript` de cron- un nombre no ASCII no se puede mandar, y un nombre con bytes que no son UTF-8 válido y no declaran su codificación -un CSV latin1 leído sin `fileEncoding`- no se manda con ningún controlador. Una candidata así sale de la lectura y sus combinaciones quedan en `cobertura_podas` con motivo `columna_no_escribible` y el remedio en `detalle`; las demás se comparan. Una columna de `orden` así deja la lectura de esa tabla sin `ORDER BY` —no se puede ordenar como se declaró—, y `meta$lecturas` y `meta$estable` lo dicen. Los nombres publicados

son los de `DBI::dbListFields()`, los mismos que publica `perfilar_coleccion()`, aunque el controlador devuelva los de la lectura en otra codificación.

Una columna que la credencial no puede leer, o que llega enmascarada, no se compara y no se lleva la tabla. Las columnas de cada tabla y lo que la credencial puede leer de ellas salen del catálogo del motor, de la misma función que usa `perfilar_dbi()`: en PostgreSQL `pg_attribute` con `has_column_privilege()` —`dbListFields()` de RPostgres no lista las columnas sin privilegio—, y en SQL Server `sys.columns` con `HAS_PERMS_BY_NAME()` y `sys.masked_columns` (enmascaramiento dinámico, salvo que la credencial tenga `UNMASK`). Sin candidatas declaradas la lectura nombra las demás columnas en vez de `SELECT *`, que con un permiso por columna se rechaza entero, y la columna no se nombra en ninguna consulta. Sus combinaciones quedan en `cobertura_podas` con motivo `columna_sin_permiso` o `columna_enmascarada`: una columna enmascarada lee la máscara —`xxxx`—, y dos así parecerían relacionadas sin compartir un solo valor. Una columna de `orden` así deja la lectura sin `ORDER BY`, como la que no se puede escribir.

Los valores se leen como los lee `perfilar_dbi()`. RPostgres entrega `0` en toda celda `money`, y `MIN`/`MAX` de una columna `money` también: la lectura de cada tabla y la consulta de rangos de la poda la leen convertida en el motor —`CAST(x AS NUMERIC)`—, con la misma expresión que `perfilar_dbi()` publica en `meta$lectura_convertida`. Saber el tipo cuesta una consulta de cero filas por tabla, sólo en las conexiones donde la lectura convierte algo. Y el texto que el controlador entrega en latin1 con la marca UTF-8 —odbc con FreeTDS bajo un locale que no es UTF-8, sobre una columna `varchar`— se lee como latin1, así que se compara con el de una `nvarchar`; la lectura lo declara en `meta$lecturas$motivo`, con el remedio, porque un carácter fuera de latin1 llega como `?` (ver «Fallo parcial» en `perfilar_dbi()`). Con `muestra = Inf` no se escribe `LIMIT`: la tabla se lee entera por la vía sin tope. Dentro de una transacción del usuario, la lectura sigue la regla de `perfilar_dbi()`: en PostgreSQL cada sentencia va dentro de un punto de guardado y la transacción no queda abortada.

Cada par se compara sobre una muestra de filas de cada tabla, y el resultado declara ese alcance: una relación candidata sobre una muestra **no es una clave foránea comprobada**, es un indicio que hay que confirmar contra el diccionario de datos.

Las columnas de ancho fijo (`char(n)` y equivalentes del controlador) se leen sin el relleno final que forma parte de su representación en el motor. Así, una clave `char` y su contraparte `varchar` se comparan por el valor lógico que el motor iguala, no por los espacios que entrega DBI. En SQL Server la misma regla alcanza a los pares de texto `varchar` contra `varchar`, porque su cotejamiento ignora los espacios finales. En PostgreSQL no se recortan todas las columnas de texto: se normaliza el lado `bpchar` y la contraparte `varchar` que el motor compara con él, pero un `text` conserva su espacio final porque PostgreSQL lo distingue. Así la relación publicada sigue la igualdad del motor, no una igualdad de cadenas recortadas.

Si una lectura conjunta falla en una fila, la colección aísla por descarte las columnas culpables y deja sus nombres en `cobertura_podas` con motivo `columna_lectura_fallida`; las columnas sanas conservan sus relaciones. Si se declaró `orden`, el aislamiento repite las sondas sin `ORDER BY` sólo cuando la propia columna de orden es la culpable, y `meta$estable` se calcula desde las lecturas que realmente llevaron ese orden.

Con la tabla de referencia truncada la cobertura es una cota inferior. Un valor de `tabla_2` que no está entre las filas leídas de `tabla_1` puede estar en las demás, así que la cobertura sólo puede subir al leer más. Una combinación de columnas que con la referencia truncada no alcanza `umbral_cobertura` no se descarta en silencio, **aunque otra del mismo par lo alcance**: un indicador de dos valores llega a cobertura 1 trivialmente, y no puede borrar la clave foránea que quedó en 0,5 por la lectura. El par queda en `cobertura_pares` como «sin conclusión» —una fila por par, no

una por combinación—, con cuántas combinaciones no alcanzaron el umbral, las cinco de mayor cobertura medida con su cota, cuántas lo alcanzaron —que están en `relaciones`— y cómo resolverlo. `meta$pares_sin_conclusion` los cuenta —también están en `pares_comparados`, porque se compararon—.

Los motivos de `cobertura_pares` citan en claro lo que el usuario declaró y lo que el resto de la salida ya publica: los nombres de las tablas, de sus esquemas y de sus columnas, y las de `orden` y `columnas_candidatas`; el resto del mensaje del motor sigue la regla de `perfilar_dbi()`.

El alcance se declara por par y en la fila. Cada relación publica `filas_leidas_1`/`filas_leidas_2` —cuántas se compararon—, y además `filas_totales_1`/`filas_totales_2` y `muestreado_1`/`muestreado_2`: cuando la lectura fue completa, el total de la tabla es exacto; cuando quedó truncada, el total va `NA` y `muestreado` vale `TRUE`, porque lo único que se sabe es que hay más filas que el tope. Publicar el tope como si fuera el total sería decir que se leyó todo. Para saberlo se pide **una fila más** que el tope, que cuesta mucho menos que un `COUNT(*)` por tabla y contesta exacto; esa fila de sobra no viaja al resultado.

Los atributos de `detectar_relaciones()` no viajan en `relaciones`: sobrevivirían sólo los del **primer** par publicado —`rbind()` conserva los del primer argumento—, con etiquetas `tabla1`/`tabla2` que no nombran a ningún par, y con dos pares dirían `muestreado = FALSE` aunque el segundo hubiera leído 10.000 filas de 20.000. Lo que dirían vive en las columnas, una por par, y el conteo de pares en `meta`, que cuenta todos.

Las columnas candidatas se podan antes de materializar cada comparación. Las podas quedan declaradas en `cobertura_podas`, con su motivo y conteo. La poda cierta por rangos usa `MIN` y `MAX` sobre el universo completo de cada tabla, no sobre las filas de la muestra. Si el motor no puede entregar ese rango, la comparación sigue sin aplicar esa poda.

Un entero ancho que llega como doble no se compara. Por encima de 2^53 un doble no distingue dos enteros consecutivos, y `dbConnect(duckdb::duckdb())` entrega BIGINT como doble por omisión: dos identificadores distintos se fundirían y el objeto afirmaría una relación que el motor no tiene. Una columna leída como doble con algún valor de al menos 2^53 en valor absoluto deja sus pares en `sin_comparar`, con motivo `entero_como_doble` en `cobertura_podas` y el remedio en el detalle: conectar pidiendo `integer64` —`duckdb(bigint = "integer64")`, o `bigint = "integer64"` en RSQLite—.

El campo `detalle` de una poda por rangos disjuntos **no publica extremos que identifiquen**: si el mínimo o el máximo de un lado llega al piso de la protección —seis caracteres o más—, ese rango sale como `[valor protegido]`. El mínimo de una columna de documentos es el documento de una persona real, y el par mínimo–máximo acota además a todos los valores intermedios.

Value

Objeto `relaciones_coleccion` con `relaciones`, `cobertura_pares`, `cobertura_podas` y `meta`.

See Also

`coleccion()`, `estimar_costo_coleccion()`, `detectar_relaciones()`

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE) &&
    requireNamespace("DBI", quietly = TRUE)) {
```

```
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "personas", data.frame(id = 1:20, nombre = letters[1:20]))
  DBI::dbWriteTable(con, "visitas", data.frame(persona_id = c(1:15, 1:5)))
  col <- coleccion(con, c("personas", "visitas"), nombre = "padron")
  relaciones_coleccion(
    col, pares = data.frame(tabla_1 = "personas", tabla_2 = "visitas"),
    columnas_candidatas = list(personas = "id", visitas = "persona_id"),
    orden = list(personas = "id", visitas = "persona_id")
  )
  DBI::dbDisconnect(con)
}
```

`reportar` *Crear un reporte HTML autocontenido*

Description

Genera un unico archivo HTML en espanol usando solo funciones de R base. El estilo se incluye dentro del documento: no requiere red, navegador especial, conversor externo ni archivos auxiliares. Cada valor dinamico se escapa antes de incorporarlo al documento.

Usage

```
reportar(
  x,
  ...,
  archivo = tempfile("reporte-lupa-", fileext = ".html"),
  sobrescribir = FALSE,
  titulo = "Reporte de calidad de datos",
  fecha = Sys.time(),
  max_filas = 100L,
  max_patrones = 20L,
  proteger_datos_personales = FALSE
)
```

Arguments

`x`	Un objeto compatible o una lista de objetos compatibles.
`...`	Objetos adicionales de clase `analisis`, `perfil`, `medicion`, `evaluacion_calidad`, `historico_calidad`, `deriva_perfil`, `deriva_calidad`, `plan_limpieza` o `duplicados_aproximados`.
`archivo`	Ruta de salida. De forma predeterminada crea un archivo en `tempdir()`. Una ruta que ya es un directorio se rechaza.
`sobrescribir`	Si se permite reemplazar un archivo existente.
`titulo`	Titulo visible del reporte.
`fecha`	Fecha y hora de generacion, inyectable para obtener resultados reproducibles. Se normaliza a UTC.

`max_filas` Maximo de filas por tabla y de columnas del perfil cuyos patrones se detallan. Las omisiones se informan dentro del reporte.

`max_patrones` Maximo de patrones mostrados por columna. Las omisiones se informan dentro del reporte.

`proteger_datos_personales`
Si se enmascaran modas, ejemplos, evidencia, estadisticos de orden, cuantiles y rangos temporales de columnas cuya clasificacion activa proteccion automatica. Es `FALSE` por omision: la proteccion es un complemento que se pide, y publicar o no los valores de una base es decision de quien la usa. Con `TRUE`, las coincidencias debiles se informan sin suprimir, y lo que el perfil ya trae enmascarado no se puede volver a mostrar.

El enmascarado es el mismo que el de `perfilar()` -ver su `proteger_datos_personales`-: la forma exacta, la que solo difiere en separadores, tildes, caja o escritura, y en los digitos, el documento entero con cualquier agrupacion, sin su verificador, o sin su primer digito detras de un comodin. La comparacion parcial se limita a digitos a proposito: aplicada al texto taparia una palabra corriente por compartir un tramo con un apellido, y eso silencia contenido del informe en vez de proteger un dato.

Un plan de limpieza, una medicion, un historico que incluye una medicion o una comparacion de deriva armados con la proteccion desactivada no se publican en un informe protegido: su seccion se reemplaza por una que dice por que se omitio y como pedirla. Lo sabe por una marca que el objeto lleva, y que `[`, `subset()` y `rbind()` conservan. En el historico, una medida suprimida por una regla se enmascara como en la evaluacion. Un perfil -o un analisis que no conserva la tabla- armado con la proteccion desactivada se vuelve a proteger con lo que conserva, y su seccion lo declara: sin la tabla, una variante de un valor protegido escrita en otra columna puede quedar a la vista. Para la proteccion completa, el perfil se arma con la proteccion activa.

Details

Se pueden combinar objetos producidos por el profiling, la medicion, la evaluacion, el historico, las comparaciones de deriva y la planificacion de limpieza. Cada tipo anade su seccion; el reporte no modifica datos ni aplica planes. Si una evaluacion contiene desenlaces de supresion declarados por reglas, el reporte enmascara su `valor_medido` y el `resultado` de las mismas medidas incluidas en el documento, también en las otras evaluaciones del informe: la supresión de una vale para todas, en cualquier orden, y la de una medida que un histórico del informe ya trae suprimida, también. El enmascarado se hace sobre copias y no modifica los objetos recibidos.

Una seccion que no se puede armar **se declara dentro del informe** y no lo interrumpe: si a un `historico_calidad` le faltan campos que su seccion necesita -`fecha`, `id_medicion`, `nivel`, `perfil` o `resultado`-, la seccion dice cuales faltan y el resto del documento se escribe igual. Lo mismo vale para cualquier otra seccion que falle al armarse -un perfil al que le falta un componente-: queda una seccion que lo dice. Un texto con bytes que no son UTF-8 valido se muestra con esos bytes en hexadecimal, como los muestra la consola, y no interrumpe el informe. Un carácter de control —que un navegador no dibuja— se escribe con su código, `<U+0001>`, como en la evidencia de los hallazgos. Las cifras se escriben con todos sus digitos significativos, sin el redondeo de la consola: con las cifras que vuelven al valor del objeto —quince, o dieciséis o diecisiete cuando hacen falta: `0.1 + 0.2` se escribe `0.30000000000000004`— y sin notación científica. Una cifra no se

abrevia como un texto: la que sin exponente pasaría de 240 caracteres —`8.98846567431158e+307`, el ausente de Stata leído como número— se escribe con exponente y las mismas cifras, y una celda con varias se abrevia por cifras enteras.

Value

La ruta normalizada del archivo, de forma invisible.

See Also

`perfilar()`, `medir()`, `evaluar()`, `historico_calidad()`, `comparar_perfiles()`, `planificar_limpieza()`

Examples

```
perfil <- perfilar(datos_administrativos)
archivo <- reportar(perfil)
unlink(archivo)
```

---

requisitos_motor *Consultar requisitos para conectarse a motores de bases*

Description

Devuelve el catálogo de paquetes R, bibliotecas del sistema, dialectos y estado de prueba que `lupa` conoce. La presencia del paquete R se comprueba en la máquina actual. La biblioteca del sistema no se declara instalada sin una comprobación específica: cuando corresponde, la columna lo deja como `no_comprobada` y conserva las rutas de resolución posibles.

Usage

```
requisitos_motor(motor = NULL)
```

Arguments

`motor`	`NULL` para devolver el catálogo completo, o el nombre de un motor, como `"oracle"`, `"postgresql"` o `"sql server"`.

Details

El estado `probado` se toma de la verificación que corre `benchmark/verificar_motor.R` en el repositorio del paquete; el artículo "Motores y escala" del sitio publica, por motor, la versión del servidor, el controlador, la fecha y el resultado de cada comprobación. `esperado` significa que el dialecto está implementado pero no se comprobó contra ese motor real en esa verificación. Los motores que no cubre no se presentan como probados.

Ante una conexión abierta, `coleccion()` y `perfilar_coleccion()` piden el paquete del **controlador con que se abrió**, que reconocen por la clase de la conexión y el paquete que la define, no por el motor que informa el servidor: un MySQL abierto con `RMariaDB::MariaDB()` usa RMariaDB y no pide ni carga RMySQL, cuya clase se llama igual y rompería la conexión.

Value

Un `data.frame` de clase `requisitos_motor`, con una fila por variante documentada. Incluye el estado comprobado del paquete R y un estado explícito para la biblioteca del sistema: `no_comprobada` no significa instalada.

Examples

```
# El catálogo completo: una fila por variante documentada.
todos <- requisitos_motor()
nrow(todos)

# Lo que hace falta para un motor en particular, antes de intentar conectarse.
requisitos_motor("postgresql")[, c("paquete_r", "estado_paquete_r")]

# `no_comprobada` en la biblioteca del sistema no significa instalada: el
# paquete no puede saberlo sin intentarlo, y lo dice en vez de suponerlo.
unique(requisitos_motor()$estado_biblioteca_sistema)
```

`senal_redundante` *Declarar una señal redundante entre columnas*

Description

Declara que varias columnas de la misma tabla codifican **el mismo hecho**, de modo que `detectar_discordancias()` pueda informar las filas donde no concuerdan. El grupo lo declara quien conoce los datos: `lupa` no lo adivina, porque dos columnas de año pueden ser el año de nacimiento y el año de ingreso y no tienen por qué coincidir.

Usage

```
senal_redundante(columnas, ventana = 0, transformacion = NULL, nombre = NULL)
```

Arguments

`columnas`	Nombres de al menos dos columnas que codifican el mismo hecho.
`ventana`	Tolerancia máxima admitida entre los valores, en las unidades del valor comparado. Por omisión `0`: coincidencia exacta. Sólo se aplica si todas las columnas comparadas son numéricas; con fechas o texto la comparación es de igualdad exacta y se avisa.
`transformacion`	Lista opcional con nombres de columna y funciones que las llevan a una escala comparable.
`nombre`	Etiqueta de la señal. Por omisión, las columnas unidas.

Details

`transformacion` permite comparar columnas que guardan el hecho de formas distintas —por ejemplo extraer el año de una fecha para compararlo con una columna de año—. Cada función recibe la columna entera y devuelve un vector de la misma longitud.

`ventana` es la tolerancia, **en las unidades del valor transformado**: con años, `ventana = 1` acepta un año de diferencia. La unidad no se adivina; es la del resultado de la transformación.

La ventana **sólo gobierna una comparación numérica**. Si alguna de las columnas comparadas no es numérica —fechas, texto, factores—, la comparación es de igualdad exacta sobre el texto y la ventana no se aplica: eso queda dicho en la evidencia (`comparacion textual exacta`) y, si se declaró una ventana mayor que cero, se avisa al medir. Para comparar fechas con tolerancia hay que llevarlas con `transformacion` a un número en la unidad de la ventana.

Esa igualdad exacta es **de valores, no de su impresión**. Una fecha-hora se compara por su instante: el mismo instante guardado en dos zonas horarias concuerda. Se escribe en UTC, al segundo y con su fracción exacta si la tiene —la misma escritura que usa `detectar_relaciones()`—, así que comparada con una columna de texto, el texto tiene que estar en UTC; para otra zona, se la lleva con `transformacion`. Un número se compara con todas sus cifras —`1e5` y el texto `"100000"` concuerdan; `0.1 + 0.2` y `0.3` no—. Si todas las columnas son duraciones (`difftime`), se comparan en segundos: una hora y sesenta minutos concuerdan. Un texto se compara por sus bytes, así que una palabra con la tilde compuesta y la misma con la tilde descompuesta son dos textos distintos.

Value

Objeto de clase `senal_redundante`: una lista con `nombre` -el declarado, o las columnas unidas por =-, el vector `columnas` que la señal relaciona, la `ventana` numérica y la `transformacion` validada. Describe la relación esperada; no la evalúa.

See Also

`detectar_discordancias()`, `detectar_dependencias()`

Examples

```
senal_redundante(c("anio_fiscal", "anio_archivo"))
senal_redundante(
  c("fecha", "anio_fiscal"),
  transformacion = list(fecha = function(x) as.integer(format(x, "%Y")))
)
```

Description

Vector para solicitar explícitamente la lista numérica completa publicada por naniar. Sus autores incluyen a Nicholas Tierney, Di Cook, Miles McBain y Colin Fay. Incluye `66`, `77`, `88` y `9999`, que pueden ser edades, códigos o años legítimos y por eso no se aplican de forma predeterminada. Se usa como `perfilar(datos, sentinelas_numericos = sentinelas_naniar)`. Tanto este vector como las cadenas de ausencia incorporadas en el paquete están congelados con referencia a naniar 1.1.0; no cambian según la versión instalada.

Usage

```
sentinelas_naniar
```

Format

Vector numérico de ocho elementos.

Source

`naniar::common_na_numbers`, versión 1.1.0. Véase el repositorio de naniar.

See Also

`perfilar()`, `planificar_limpieza()`

Examples

```
sentinelas_naniar
datos <- data.frame(codigo = c(1, 66, 9999))
perfil <- perfilar(
  datos, sentinelas_numericos = sentinelas_naniar,
  analizar_dependencias = FALSE
)
perfil$columnas[, c("columna", "n_faltantes_disfrazados")]
```

---

sugerir_clave *Sugerir qué columnas podrían ser la clave*

Description

Ordena las columnas que **podrían** identificar una fila, para que quien conoce la tabla elija. No decide: una columna única puede ser una clave o puede ser una magnitud que no repite, y esa diferencia no está en los datos.

Usage

```
sugerir_clave(datos, maximo = 5L, umbral_casi = 0.95)
```

Arguments

`datos`	Tabla a examinar.
`maximo`	Cuántas sugerencias devolver como máximo: un entero positivo.
`umbral_casi`	Proporción de valores distintos a partir de la cual una columna que repite se informa igual, como candidata con duplicados.

Details

El orden combina tres señales, y las tres se publican para que se pueda discutir el orden en vez de aceptarlo: si la columna **identifica** cada fila sin repetir, si **no tiene ausentes** —una clave con ausentes no identifica—, y cuánto se **parece su nombre** al de una clave (`id_persona`, `documento`, `expediente`). El nombre se compara sin acentos ni separadores, así que `ID_Persona` e `idpersona` pesan igual.

Las columnas que repiten valores no se ofrecen como clave de una sola columna, pero sí se informan cuando estuvieron cerca: una columna que identifica al 99 % suele ser una clave con duplicados de carga, que es exactamente lo que conviene mirar.

Una columna `double` con valores de parte fraccionaria —un importe, una coordenada— puede identificar cada fila y aun así no ser una clave. No se oculta que identifica, porque es un hecho medido: se dice en el motivo y la columna queda al final del orden. Es el mismo criterio que aplica `detectar_claves()`, consultado a la misma función. Las columnas compuestas —matrices o arreglos de más de una dimensión— no se ofrecen como claves ni reciben una tasa de valores distintos.

Una columna cuyo nombre se repite en la tabla no se mide: la clave se pasa a `perfilar()` por nombre, y ese nombre no dice cuál de las dos es. Aparece al final, fuera de `maximo`, con `identifica`, `sin_faltantes` y `tasa_distintos` en `NA` y el motivo; `elegir_clave()` no la ofrece.

Value

Un `data.frame` con una fila por columna candidata, ordenado de más a menos probable, con las tres señales por separado y el motivo en texto. Cero filas si ninguna columna se acerca.

See Also

`detectar_claves()` para las combinaciones de varias columnas, y el argumento `clave` de `perfilar()` para declarar la que se elija.

Examples

```
personas <- data.frame(
  id_persona = 1:4,
  documento = c(11111111, 22222222, 33333333, 33333333),
  edad = c(30, 41, 25, 25)
)
sugerir_clave(personas)
```

tablero_calidad *Construir un tablero de calidad*

Description

Resume una corrida en una fila por métrica y objeto. Las métricas booleanas usan `ratio` por omisión y las reales usan `promedio`; la elección queda siempre en la columna `agregacion`. `ratio_umbral` sólo se aplica cuando se declara también el umbral correspondiente.

Usage

```
tablero_calidad(
  medidas,
  agregaciones = NULL,
  umbrales = NULL,
  marco = NULL,
  cobertura = NULL
)
```

Arguments

`medidas`	Objeto creado por `medir()` o `agregar()`.
`agregaciones`	`NULL`, una agregación para todas las métricas, un vector con nombres de métrica instanciada o un data frame con `metrica_instanciada`, `agregacion` y, opcionalmente, `umbral`.
`umbrales`	Vector numérico opcional con nombres de métrica instanciada. Se exige para cada `ratio_umbral` que no lo declare dentro de `agregaciones`.
`marco`	Marco conceptual. Si se omite, usa el asociado a la medición, el marco AGESIC cuando corresponde o el conjunto de factores medidos —los tres caminos contienen lo que se publica—. Un marco declarado puede no contener los pares medidos: no se rechaza, porque informar contra la taxonomía propia es un uso previsto, pero lo que queda afuera se cuenta en `medidos_fuera_del_marco` y se nombra en `pares_fuera_del_marco`.
`cobertura`	Cobertura opcional creada por `cobertura_analisis()`. **La correspondencia entre esa cobertura y `medidas` es suya, no del paquete**, igual que en `cobertura_analisis()`: el tablero cruza las dos por dimensión y factor y reescribe el estado de un factor cuando la medición lo cubre, sin verificar que las dos describan la misma tabla. Con una cobertura de otro análisis, el factor va a figurar como medido por esta corrida.

Details

La identidad de cada celda son tres columnas: `metrica_instanciada`, `entidad` y `objeto`. No alcanza con `metrica` y `objeto`, porque `metrica` es el nombre genérico: dos tablas con una columna del mismo nombre, o dos especializaciones de la misma genérica sobre la misma columna, publicarían dos filas con la **misma** identidad y valores distintos, y rehacer la celda con las claves

publicadas mezclaría las medidas de las dos sin reproducir ninguna. `metrica_instanciada` sola tampoco alcanza: sobre un agregado usual vale `agregada:<funcion>:<metrica>` para todas las filas, y ahí la entidad es lo único que separa. En una agregación de instancias de entidad a entidad, conserva además la instancia completa para que dos claves de una tabla no se fundan. Cuando hay varias entidades, `objeto` además nombra la tabla —`cod (tabla: t1)`—, en el mismo idioma con el que la granularidad de tabla publica `(tabla: t1)`.

Ese trío es la clave con la que se **agrupan** las medidas, y no sólo la que se publica: agrupar por menos de lo que se declara como identidad funde dos cosas que el objeto dice distintas. Vale también en las granularidades sin rama propia —`conjuntoAtributos` entre ellas—: agrupando sólo por el objeto, dos entidades con el mismo objeto saldrían como **una** celda con la entidad de la primera y el promedio de las dos.

Y la unicidad **se verifica**: si dos celdas quedaran con el mismo trío, la función falla nombrando las dos y sus valores, en lugar de publicar dos filas con la misma identidad. Se rechaza en vez de quedarse con una porque dos valores distintos para la misma identidad es una contradicción de la entrada, y elegir uno sería elegir por quien llama.

El objeto conserva la cobertura completa del marco: factores medidos, sin métrica declarada, no aplicables y fuera de alcance. `print()` muestra ambos elementos para que unas pocas filas nunca se lean como cobertura total.

Esa cobertura se cuenta sobre los factores **del marco**, así que una medida cuyo par dimensión-factor el marco no declara no entra en ninguna de sus casillas. Eso es legítimo —declarar el marco propio y medir con las métricas de `lupa` es un flujo previsto, y la cobertura de ese marco puede satisfacerse por `perfil_mide`—, pero leer «se midieron 0 factores de este marco» al lado de una fila con valor parece una contradicción del objeto. Por eso el alcance publica además `medidos_fuera_del_marco`, que **no** suma con las otras cuatro casillas, y el atributo `pares_fuera_del_marco` los nombra.

Value

Data frame S3 `tablero_calidad`. Los atributos `alcance` y `cobertura` conservan los conteos y el detalle del marco, y `pares_fuera_del_marco` nombra los pares medidos que ese marco no declara.

See Also

`medir()`, `agregar()`, `indice_calidad()`

Examples

```
nucleo <- metricas_nucleo()
instancia <- instanciar(especializar(nucleo$NoNulo), "personas", "edad")
medidas <- medir(modelo(instancia), data.frame(edad = c(20, NA, 35)))
tablero_calidad(medidas)
```

validadores_formato *Validadores internacionales de sintaxis y dígitos de control*

Description

Estas funciones son puras y vectorizadas: no consultan la red ni prueban la existencia de un país, moneda, buzón o entidad emisora. Los códigos se contrastan con copias locales de las listas vigentes al preparar esta versión del paquete. Los valores ausentes devuelven NA; todo valor presente que no cumple el contrato devuelve FALSE. Los espacios **al borde** se recortan antes de validar —son ruido de captura, no parte del dato—, salvo en validar_url(), donde un espacio literal en cualquier posición rompe la sintaxis y el valor es inválido. Se recorta todo espacio: el ASCII, el tabulador y los separadores de Unicode, como el espacio duro (U+00A0) que dejan las planillas y las páginas web. Un espacio en el medio no se recorta en ningún validador.

Usage

```
validar_url(x, esquema_obligatorio = TRUE)

validar_iso3166(x, tipo = c("alpha2", "alpha3", "numerico"))

validar_iso4217(x)

validar_correo(x)

validar_luhn(x)

validar_mod97(x)
```

Arguments

x	Vector que se desea validar.
esquema_obligatorio	Si es TRUE, exige http:// o https://. TRUE es el valor estricto por omisión; FALSE permite una URL con host sin esquema, como ejemplo.uy.
tipo	Forma de código ISO 3166: dos letras ("alpha2"), tres ("alpha3") o tres dígitos ("numerico"). Los valores numéricos de uno o dos dígitos se completan con ceros a la izquierda.

Details

La respuesta es sobre el dato, no sobre cómo lo escribe R. Un número se valida por sus dígitos: el doble 12000000 se lee como "12000000" aunque scipen lo imprima en notación científica. Un doble desde 2^53 no conserva sus dígitos —es el más cercano de varios enteros— y devuelve FALSE; el mismo número como texto o como integer64 sí se valida. Una magnitud con clase —Date, POSIXct, difftime, hms, units— no es la escritura de un documento y devuelve FALSE (o NA donde falta), sin validar los días o segundos que guarda. NaN es ausente y devuelve NA, igual que en el resto del paquete. Las letras se pasan a mayúsculas sólo dentro de ASCII: toupper()

convierte la s larga (U+017F) en `S` y la i sin punto (U+0131) en `I`, y con eso una s larga seguida de `e` pasaría por un código ISO; con la conversión limitada a ASCII es inválida. Y la marca de codificación del texto —`latin1` o UTF-8— no cambia la respuesta.

`validar_correo()` comprueba un subconjunto práctico y deliberadamente conservador de la sintaxis `addr-spec`: parte local de puntos y caracteres ASCII habituales, seguida por un dominio DNS con al menos un punto. No admite comentarios, cadenas entre comillas ni literales de dominio válidos en la gramática completa de RFC 5322, y no prueba entrega ni existencia.

`validar_url()` comprueba URLs jerárquicas con host y acepta sólo los esquemas `http` y `https`: son los esquemas de red que el paquete puede reconocer sin atribuir semántica a protocolos arbitrarios. `javascript:` y `data:` se rechazan deliberadamente, incluso si se parecen a una cadena con dos puntos. Por omisión el esquema es obligatorio; con `esquema_obligatorio = FALSE` también se acepta un host como `ejemplo.uy`. Se aceptan nombres de dominio Unicode (IDN) y nombres ASCII en punycode, puertos entre 1 y 65535, rutas, consultas y fragmentos con codificación porcentual válida. No se consultan DNS ni se afirma que el recurso exista. Los espacios, separadores Unicode y caracteres de control literales hacen que el valor sea inválido; un espacio porcentualmente codificado sigue la sintaxis de una URL y no es un espacio literal. También lo hacen los invisibles de formato —U+200B, U+FEFF, U+200D, las marcas de dirección—, que hacen que dos URL distintas se vean iguales, y la contrabarra, que no está en la gramática de RFC 3986 y que un navegador lee como barra: delante de una arroba, el host al que va el navegador no es el que se validaría.

`validar_luhn()` acepta únicamente dígitos y aplica el algoritmo de Luhn. `validar_mod97()` acepta letras ASCII y dígitos, transforma las letras a `A = 10, ..., Z = 35` y exige resto 1 conforme a ISO 7064 MOD 97-10. No reordena caracteres: protocolos como IBAN deben hacer antes su transformación propia.

Value

Vector lógico de la misma longitud que `x`.

References

International Organization for Standardization. *ISO 3166 Country Codes*. https://www.iso.org/iso-3166-country-codes.html

SIX Group. *ISO 4217 Currency Codes*. https://www.six-group.com/en/products-services/financial-information/market-reference-data/data-standards.html

Resnick P (2008). *Internet Message Format*, RFC 5322. https://www.rfc-editor.org/info/rfc5322/

Luhn HP (1960). *Computer for Verifying Numbers*, US Patent 2,950,048. https://patents.google.com/patent/US2950048A/en

International Organization for Standardization. *ISO/IEC 7064:2003*. https://www.iso.org/standard/31531.html

See Also

`pack_validadores()`, `validar_ci_uy()`, `especializar()`

Examples

```
validar_iso3166(c("UY", "CL", "ZZ"))
validar_iso4217(c("UYU", "CLP", "ZZZ"))
validar_correo(c("persona@example.org", "sin-arroba"))
validar_url(c("https://ejemplo.uy", "ejemplo.uy"))
validar_url("ejemplo.uy", esquema_obligatorio = FALSE)
validar_luhn(c("79927398713", "79927398714"))
validar_mod97(c("99991234567890121414 90", "99991234567890121414 91"))
```

`validadores_uy` *Validadores estructurales de Uruguay*

Description

`validar_ci_uy()` comprueba la cédula de identidad mediante su dígito verificador módulo 10. Acepta siete u ocho dígitos y separadores de puntos, espacios o guion; una cédula de siete dígitos se completa con cero inicial. Cualquier espacio separa, también el fino o el duro de Unicode.

Usage

```
validar_ci_uy(x)

validar_rut_uy(x)
```

Arguments

`x`	Vector que se desea validar.

Details

`validar_rut_uy()` comprueba la estructura de doce dígitos del RUT y su dígito verificador módulo 11. Un RUT de once dígitos se completa con cero inicial, igual que la cédula de siete: es el que pierde su cero al guardarse como número (`read.csv()` lee `073602060015` como `73602060015`). El algoritmo operativo no está publicado por DGI como especificación normativa abierta; esta implementación se contrastó con la implementación pública de `python-stdnum` y con ejemplos públicos. Por eso valida estructura y dígito, no vigencia ni existencia registral.

Los separadores son de la escritura, no del número. Punto, espacio y guion se admiten porque una persona escribe `1.234.567-2`; en un valor numérico el punto es el separador decimal y el guion es el signo, así que un número fraccionario o negativo devuelve `FALSE` —no se le quita la fracción ni el signo para hacerlo pasar—. Un número entero no negativo sí se valida, como cualquier otra escritura del mismo documento, y por sus dígitos: el doble `12000000` es la cédula `"12000000"` aunque `scipen` lo imprima en notación científica. Un número no guarda ceros iniciales, así que en `validar_ci_uy()` sus dígitos se completan con ceros hasta ocho: el número `1236` es la cédula `"00001236"`. El texto, que sí los guarda, sigue exigiendo siete u ocho dígitos: `"1236"` devuelve `FALSE`. El cuerpo `0000000` no es una cédula —su dígito cuadra, pero la numeración empieza en 1—, así que `0` y `"00000000"` devuelven `FALSE`, como el RUT de secuencia `000000`. `NaN` es ausente y devuelve `NA`.

Value

Vector lógico de la misma longitud que `x`.

References

Poder Ejecutivo de Uruguay (1978). Decreto 501/978, artículo 2. https://www.impo.com.uy/bases/decretos/501-1978/2

Unidad de Acceso a la Información Pública (2024). Resolución 145/024. https://www.gub.uy/unidad-acceso-informacion-publica/institucional/normativa/resolucion-n-145024-sobre-reserva-informa

de Jong A. *python-stdnum: Uruguay RUT*. https://github.com/arthurdejong/python-stdnum/blob/master/stdnum/uy/rut.py

See Also

`pack_validadores()`, `validar_iso3166()`

Examples

```
validar_ci_uy(c("1.234.567-2", "1.234.567-3"))
validar_rut_uy(c("21 100 342 0017", "21 030 367 0014"))
```

Index

∗ **datasets**
  datos_administrativos, 27
  datos_operativos, 28

accesores_perfil, 4
acumular_historico *(*historico_calidad*)*, 64
acumular_historico(), *65*
agregar, 5
agregar(), *11*, *14*, *75, 76*, *79*, *81*, *86*, *180*, *192, 193*
analizar, 8
analizar(), *4*, *13*, *16*, *29*, *32*, *52, 53*, *91*, *93, 94*, *101*, *158, 159*, *177*, *180*
analizar_tiempo, 12
analizar_tiempo(), *11*
aplicar *(*planificar_limpieza*)*, 159
aplicar(), *64*, *92*, *94, 95*
as.data.frame(), *89*

catalogo_agesic, 13
catalogo_agesic(), *17*, *69, 70*, *79*
clasificar_variables, 15
clasificar_variables(), *9*, *13*, *53*, *101*, *111*
cobertura *(*accesores_perfil*)*, 4
cobertura_analisis, 16
cobertura_analisis(), *11*, *18*, *27*, *69, 70*, *157*, *192*
coleccion, 18
coleccion(), *6*, *59, 60*, *86*, *124*, *127, 128*, *154*, *181*, *184*
columnas *(*accesores_perfil*)*, 4
comparar_equivalencia, 20
comparar_equivalencia(), *24–26*
comparar_evaluaciones, 22
comparar_evaluaciones(), *7*, *63*, *180*
comparar_perfiles, 23
comparar_perfiles(), *9*, *21*, *91*, *112*, *187*
contratos_medicion, 26
datos_administrativos, 27, *29*
datos_operativos, 28
DBI::dbQuoteIdentifier(), *129*, *168*
descubrir_patrones, 29
descubrir_patrones(), *68*, *123, 124*
detectar_asociaciones, 31
detectar_asociaciones(), *10*
detectar_claves, 33
detectar_claves(), *37*, *50*, *113*, *117*, *191*
detectar_dependencias, 34
detectar_dependencias(), *32*, *34*, *40*, *124*, *166*, *177*, *189*
detectar_deriva_calidad, 37
detectar_deriva_calidad(), *22*, *26*, *63*, *66*, *180*
detectar_discordancias, 38
detectar_discordancias(), *101*, *188, 189*
detectar_duplicados_aproximados, 40
detectar_duplicados_aproximados(), *57–59*, *79*, *102*, *117, 118*
detectar_formatos_fecha, 47
detectar_formatos_fecha(), *13*, *31*, *68*
detectar_relaciones, 48
detectar_relaciones(), *34*, *40*, *176*, *179*, *181, 182*, *184*, *189*
distribucion_valores, 51
distribucion_valores(), *16*, *101*, *113*
duplicated(), *164*

elegir_clave, 53
elegir_clave(), *191*
escala *(*contratos_medicion*)*, 26
escala(), *14*, *18*, *79*
especializar *(*modelo_calidad*)*, 76
especializar(), *27*, *78*, *81*, *87*, *195*
estadisticos_estimacion, 54
estadisticos_estimacion(), *71, 72*
estimar_costo, 54
estimar_costo_coleccion, 59
estimar_costo_coleccion(), *154*, *182*, *184*

evaluar, 60
evaluar(), *7*, *9*, *11*, *54*, *62*, *64*, *66*, *71–73*, *175*, *180*, *187*

granularidades, 61
granularidades(), *19*, *86*
guardar_analisis *(*persistir_analisis*)*, 158
guardar_analisis(), *11*, *24*
guardar_historico, 62
guardar_historico(), *24*, *63*, *159*
guiar_limpieza, 63
guiar_limpieza(), *166*

hallazgos *(*accesores_perfil*)*, 4
historico_calidad, 64
historico_calidad(), *7*, *22*, *37*, *61*, *63*, *66*, *72, 73*, *86*, *180*, *187*

indice_calidad, 66
indice_calidad(), *6*, *18*, *193*
inferir_tipo, 68
inferir_tipo(), *31*, *48*, *113*
instanciar *(*modelo_calidad*)*, 76
instanciar(), *81*, *179*

leer_analisis *(*persistir_analisis*)*, 158
leer_historico *(*guardar_historico*)*, 62
leer_historico(), *66*

marco_agesic *(*marco_calidad*)*, 69
marco_agesic(), *9*, *17*
marco_calidad, 69
marco_calidad(), *9*, *17, 18*, *73*, *78*, *92*
marco_cepal *(*marco_calidad*)*, 69
marco_cepal(), *72*
marco_iso25012 *(*marco_calidad*)*, 69
medicion_desde_estimaciones, 71
medicion_desde_estimaciones(), *54*
medir, 73
medir(), *7*, *9*, *14*, *17, 18*, *54*, *62*, *64*, *66*, *71, 72*, *78–81*, *83*, *92*, *176*, *180*, *187*, *192, 193*
metrica *(*modelo_calidad*)*, 76
metricas_nucleo *(*modelo_calidad*)*, 76
metricas_nucleo(), *14*, *27*, *76*
metricas_referencial, 75
metricas_referencial(), *14*, *78–80*, *179*
modelo *(*modelo_calidad*)*, 76
modelo(), *9*, *17*, *62*, *69, 70*, *83*, *177*
modelo_calidad, 76
modelo_desde_propuesta, 83
modelo_desde_propuesta(), *175–177*

n_filas *(*accesores_perfil*)*, 4
normalizacion, 84
normalizacion(), *42, 43*, *56, 57*, *68*, *93*, *117*, *178*

organizacion, 85
organizacion(), *6*

pack_validadores, 87
pack_validadores(), *195*, *197*
perfil_evaluacion *(*reglas_evaluacion*)*, 179
perfilar, 88
perfilar(), *4, 5*, *9–11*, *13*, *15–18*, *20, 21*, *23*, *25, 26*, *28, 29*, *31*, *33*, *35–37*, *39, 40*, *45*, *47, 48*, *51–53*, *61*, *68*, *73, 74*, *78, 79*, *87*, *116*, *125, 126*, *128*, *135–137*, *144*, *146, 147*, *150–152*, *154–157*, *160*, *165, 166*, *176, 177*, *186, 187*, *190, 191*
perfilar_coleccion, 124
perfilar_coleccion(), *4–6*, *18, 19*, *86*, *91*, *140*, *152–154*, *183*
perfilar_dbi, 128
perfilar_dbi(), *4, 5*, *17*, *19–21*, *91*, *95*, *100*, *108*, *116*, *120*, *125–128*, *174*, *182–184*
perfilar_por, 155
perfilar_por(), *91*, *96*
perfiles_madurez *(*reglas_evaluacion*)*, 179
perfiles_madurez(), *11*, *180*
persistir_analisis, 158
plan_perfilado_dbi, 166
plan_perfilado_dbi(), *125*, *131*, *140*, *142*, *149*, *154*, *170*
planificar_limpieza, 159
planificar_limpieza(), *28*, *47*, *64*, *91, 92*, *94*, *109*, *124*, *187*, *190*
print(), *193*
propiedades_metrica *(*modelo_calidad*)*, 76
propiedades_metrica(), *78*, *81*, *175*
propiedades_regla, 174
proponer_modelo, 175

proponer_modelo(), *11*, *16*, *37*, *50*, *79*, *81*, *83*, *91*, *124*

read.csv(), *163*
referencial, 178
referencial(), *50*, *75, 76*, *78*, *80, 81*
regla_evaluacion *(*reglas_evaluacion*)*, 179
regla_evaluacion(), *175, 176*, *180*
reglas_evaluacion, 179
relaciones_coleccion, 181
relaciones_coleccion(), *49*, *60*, *139, 140*, *154*
reportar, 185
reportar(), *11*, *18*, *26*, *47*, *66*, *94*, *159*
requisitos_motor, 187

saveRDS(), *23*, *158*
senal_redundante, 188
senal_redundante(), *39, 40*
sentinelas_naniar, *93*, 189
sql_perfil *(*accesores_perfil*)*, 4
sugerir_clave, 190
sugerir_clave(), *53*, *113*

tablero_calidad, 192
tablero_calidad(), *11*, *18*, *66, 67*
transiciones_granularidad *(*granularidades*)*, 61

validadores_formato, 194
validadores_internacionales *(*pack_validadores*)*, 87
validadores_uruguay *(*pack_validadores*)*, 87
validadores_uy, 196
validar_ci_uy *(*validadores_uy*)*, 196
validar_ci_uy(), *87*, *195*
validar_correo *(*validadores_formato*)*, 194
validar_iso3166 *(*validadores_formato*)*, 194
validar_iso3166(), *87*, *197*
validar_iso4217 *(*validadores_formato*)*, 194
validar_luhn *(*validadores_formato*)*, 194
validar_mod97 *(*validadores_formato*)*, 194
validar_rut_uy *(*validadores_uy*)*, 196
validar_url *(*validadores_formato*)*, 194

vigencia *(*contratos_medicion*)*, 26
vigencia(), *14*, *18*, *79*