Package 'scopusflow'

August 21, 2026

Title A Reproducible Workflow Layer for 'Scopus' Bibliographic Searches

Version 0.4.0

Description A coherent, quota-aware workflow layer over the Elsevier 'Scopus' Search 'API' <https://dev.elsevier.com/sc_apis.html>. It builds reproducible search plans, retrieves records with rate-limit handling, retry with back-off and optional resumable caching, normalises results to a stable tidy schema, extracts and tracks changes in Digital Object Identifiers (DOIs), sizes sets of concepts and their intersections, compares publication trends across topics, writes the search up as a reproducible record for a methods section following the 'PRISMA-S' reporting standard (Rethlefsen and others, 2021) <doi:10.1186/s13643-020-01542-z> and exports to formats compatible with downstream bibliometric tools. Network and 'API' errors are surfaced as typed conditions so that callers can respond to them programmatically. 'Scopus' is a trademark of Elsevier. This package is an independent client and is not affiliated with or endorsed by Elsevier.

License MIT + file LICENSE

URL https://github.com/pablobernabeu/scopusflow, https://pablobernabeu.github.io/scopusflow/

BugReports https://github.com/pablobernabeu/scopusflow/issues

Depends R (>= 4.1.0)

Imports cli, httr2 (>= 1.0.0), jsonlite, rlang (>= 1.1.0), stats, tibble, tools, utils

Suggests bslib, callr, fansi, ggplot2, grid, knitr, rmarkdown, shiny, spelling, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Language en-GB

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Pablo Bernabeu [aut, cre] (ORCID:
<https://orcid.org/0000-0003-1083-2460>)

Maintainer Pablo Bernabeu <pcbernabeu@gmail.com>

Repository CRAN

Date/Publication 2026-08-21 21:00:08 UTC

Contents

as_bibliometrix *Convert records to a bibliometrix-compatible data frame*

Description

Re-maps a scopus_records tibble to the tagged column layout used by the **bibliometrix** package (and the wider ISI/Web of Science convention), so results can flow into downstream science-mapping workflows.

Usage

```
as_bibliometrix(x)
```

Arguments

x	A scopus_records tibble.

Details

This produces the *shape* bibliometrix expects from the core descriptive fields. It reconstructs only what the 'Scopus' Search API returns, so richer fields that some bibliometrix analyses use, such as full author affiliations or cited references, are left out. To obtain those, export a full 'Scopus' CSV or BibTeX file from the web interface and read it with `bibliometrix::convert2df()`.

Value

A data frame (classed `bibliometrixDB`) with the standard tag columns `AU` (authors), `TI` (title), `SO` (source or publication), `DI` (DOI), `PY` (publication year), `TC` (times cited), `UT` (record id) and `DB` (`"SCOPUS"`). Character tag fields are upper-cased to match the bibliometrix convention.

Examples

```
# On the bundled corpus of real articles, which stands in for a retrieval
# of your own because 'Scopus' records may not be redistributed.
m <- as_bibliometrix(example_records)
head(m[, c("AU", "TI", "PY", "SO", "TC", "DB")])
```

as_bibtex *Export records to BibTeX or RIS*

Description

Turns a scopus_records set into a BibTeX or RIS string, the interchange formats that reference managers (Zotero, EndNote, Mendeley) and LaTeX bibliographies import. Each record becomes one entry, with its authors split out and the 'Scopus' identifier kept as a note. Records are treated as journal articles, the dominant 'Scopus' content type. BibTeX citation keys are made unique within the export, and special characters are escaped.

Usage

```
as_bibtex(x, file = NULL)

as_ris(x, file = NULL)
```

Arguments

x	A scopus_records tibble.
file	Optional path to write to. With the default NULL the formatted string is returned; with a path it is written there and returned invisibly. Nothing is written unless a path is given.

Value

A length-one character string of the formatted records (returned invisibly when file is supplied).

See Also

as_bibliometrix(), write_scopus_records(), scopus_extract_dois()

Examples

```
# On the bundled corpus of real articles, which stands in for a retrieval
# of your own because 'Scopus' records may not be redistributed. Only the
# opening of each export is shown; pass `file` to write the whole set.
cat(substr(as_bibtex(example_records), 1, 200))
cat(substr(as_ris(example_records), 1, 200))
```

---

as_tibble.scopus_records

Normalise raw 'Scopus' entries to a stable tidy schema

Description

Converts the nested list returned by the 'Scopus' Search API into a flat, predictable tibble with one row per record. This shape is the common currency of the package. Both scopus_fetch() and scopus_fetch_plan() return it, and the DOI, comparison and export helpers all consume it.

Usage

```
## S3 method for class 'scopus_records'
as_tibble(x, ...)

## S3 method for class 'scopus_records'
as.data.frame(x, ...)

## S3 method for class 'scopus_records'
```

```
autoplot(object, ...)

scopus_records(x, query = NA_character_, view = NULL)

is_scopus_records(x)
```

Arguments

x	An object to test.
...	Ignored, for S3 compatibility.
object	A scopus_records object (for the autoplot() method).
query	Optional character scalar recording the query that produced the entries, kept in the query column for provenance.
view	Optional character scalar naming the Search API view the entries came from. Pass "COMPLETE" to add an authkeywords column (see below); any other value, including the default NULL, reproduces the original columns exactly, so existing callers that never mention view see no change at all. scopus_fetch() and scopus_fetch_plan() pass this through automatically.

Details

The 'Scopus' API signals an empty result set with a single sentinel entry that carries an error field and no identifier. This is detected and turned into a zero-row result, with no spurious record in it, while a genuine record that also carries a per-entry error annotation is kept.

Author keywords are only ever present under view = "COMPLETE"; the STANDARD view (the default throughout the package) never includes them, and authkeywords is not added to the output at all in that case, so existing code that inspects the column names of a STANDARD-view result is unaffected. Even under COMPLETE view, some 'Scopus' API keys do not return populated author keywords (this was observed directly against a live, otherwise fully-entitled key during development, on documents that do carry author keywords in 'Scopus' itself); if your own keywords come back all NA, the field is most likely gated by your account's entitlement rather than genuinely absent, and is worth raising with your 'Scopus'/Elsevier account contact.

Value

The coercion methods return a plain tibble or data frame with the same columns and the scopus_records class removed.

The autoplot() method returns a ggplot2::ggplot of the records per year.

A tibble of class scopus_records with the columns entry_number (integer), scopus_id (character), doi (character), title (character), authors (character, the creator names joined with "; " when several are listed), year (integer, the leading four digits of the cover date), date (character, the ISO cover date), publication (character, the source title), citations (integer) and query (character). A missing field becomes NA, and an empty result set yields a zero-row tibble with the same columns. When view = "COMPLETE", an authkeywords column is added: the author-supplied keywords the 'Scopus' Search API returns under that view, as a single string in 'Scopus' own " | "-delimited form (NA when the document has none, or when the API omits the field for a given key's entitlement; see *Details*).

is_scopus_records() returns a length-one logical.

Examples

```
# An entry in the shape the Search API returns it. The fields are those of
# a real article, taken from the bundled `example_records`, which stands in
# for a harvest because 'Scopus' records may not be redistributed. It
# carries no 'Scopus' identifier, so `dc:identifier` is absent and
# `scopus_id` comes back NA, as it does for any unidentified record.
raw <- list(entry = list(
  list(
    `prism:doi` = "10.1021/am509065d",
    `dc:title` = "Flexible and Stackable Laser-Induced Graphene Supercapacitors",
    `dc:creator` = "Zhiwei Peng",
    `prism:publicationName` = "ACS Applied Materials & Interfaces",
    `prism:coverDate` = "2015-01-13",
    `citedby-count` = "469"
  )
))
scopus_records(raw, query = "TITLE-ABS-KEY(graphene supercapacitor)")

# Under COMPLETE view an entry may also carry author keywords, which the
# Search API returns in its own " | "-delimited form. The bundled corpus
# holds no keywords, so the ones below are illustrative.
raw_complete <- list(entry = list(
  list(
    `prism:doi` = "10.1021/am509065d",
    `dc:title` = "Flexible and Stackable Laser-Induced Graphene Supercapacitors",
    authkeywords = "graphene | supercapacitor | energy storage"
  )
))
scopus_records(raw_complete, view = "COMPLETE")

# An object already in this schema is returned unchanged.
identical(scopus_records(example_records), example_records)
```

`example_records` *A worked example harvest, in the shape 'Scopus' records take*

Description

One hundred and thirty-eight real journal articles on graphene supercapacitors published between 2015 and 2024, carrying their real titles, DOIs, source titles, first authors and citation counts. The dataset lets the package be explored, and every example and vignette be run, without an API key.

Usage

```
example_records
```

Format

A scopus_records tibble with 138 rows and the standard schema:

entry_number Position within the retrieval.

scopus_id Empty throughout. These records did not come from 'Scopus', so they carry no 'Scopus' identifier; de-duplication falls back to the DOI, as it does for any record whose identifier is missing.

doi Digital Object Identifier, missing for eleven records.

title Document title. Publisher markup for subscripts and italics is stripped; nothing else is altered.

authors First author, as named by the source.

year Publication year.

date Publication date in ISO form.

publication Source title, missing for two records.

citations Citation count at the time of retrieval.

query The search phrase that produced the record.

Details

The records are deliberately not a 'Scopus' harvest. The Elsevier API terms do not permit redistributing retrieved records, so no package can ship one. These come instead from OpenAlex, whose metadata is released under CC0 and may therefore be redistributed, and are reshaped into the schema `scopus_fetch()` returns. Running the equivalent query against 'Scopus' yields the same kind of object, with the same columns and the same handling, though not an identical set of records.

Two properties are worth knowing when reading the examples. The harvest is complete, so the number of rows per year is the real number of publications per year for that query, and the trend figures show a real publication curve. The gaps are also genuine: eleven records carry no DOI and two no source title, exactly as they arrive. They are kept because a real harvest has such gaps, and the reference-set examples are the more useful for showing how they are handled.

Source

Retrieved from OpenAlex (https://openalex.org) on 2026-07-22: the complete result set for the phrase "graphene supercapacitor" in title or abstract, restricted to journal articles from 2015 to 2024. OpenAlex data is released under CC0. Retrieval and reshaping are reproducible from `data-raw/example_records.R`.

Examples

```
example_records

# The columns and the behaviour are those of a real retrieval, so the
# analysis helpers work on it directly.
scopus_top(example_records, by = "source", n = 5)
```

`plot_scopus_comparison` *Plot a topic comparison*

Description

Draws a line chart of each comparison topic's share of the reference literature over time, from the output of `scopus_compare_topics()`. The chart uses integer year breaks, a colour-blind-safe palette and, for a handful of topics, labels the lines directly so the reader need not consult a legend. Shaded bands convey how stable each yearly share is.

Usage

```
plot_scopus_comparison(
  x,
  pub_count_in_legend = TRUE,
  highlight = NULL,
  interval = TRUE,
  legend_inside = FALSE,
  ...
)

## S3 method for class 'scopus_comparison'
autoplot(object, ...)
```

Arguments

`x`	A `scopus_comparison` object from `scopus_compare_topics()`.
`pub_count_in_legend`	Logical. When `TRUE` (the default), each topic's label carries its total record count, for example `effect size (n = 1,204)`.
`highlight`	Optional character scalar naming one comparison topic to draw the eye to. The named topic is drawn in an accent colour, and the others in grey, which is useful when one topic is the focus of a figure.
`interval`	Logical. When `TRUE` (the default), a shaded band around each line shows a Wilson interval on the yearly share. See *Details* for how to read it.
`legend_inside`	Logical. When `TRUE`, and a legend is drawn (that is, when `highlight` is not set), the legend is placed inside the plotting panel, in whichever corner has the most free space, on a small semi-transparent background, where the default places it above the panel. Direct line labelling is suppressed so the in-panel legend carries the topic key. Defaults to `FALSE`, which keeps the legend above the panel, or the direct line labels a few topics would otherwise receive.
`...`	Currently unused, present for S3 consistency.
`object`	A `scopus_comparison` object (for the `autoplot()` method).

Details

This needs the suggested package **ggplot2** and raises an informative error when it is absent. The chart shows the comparison topics alone, since the reference is the 100% denominator against which they are measured. A year for which the reference has no records carries no defined share and is omitted, which is noted in the caption.

The shaded band is a Wilson score interval computed from the comparison count and the reference count for each year. 'Scopus' returns exact counts rather than a sample, so the band is not a confidence interval in the inferential sense. It is best read as an illustrative stability range: it is wide where the reference set for a year is small, and so the share would move easily, and narrow where the reference set is large. It says nothing about query wording, indexing lag or coverage, which are the larger real uncertainties.

Value

A ggplot2::ggplot object. Printing it draws the plot.

See Also

`scopus_compare_topics()`

Examples

```
cmp <- tibble::tibble(
  query = "q", query_type = "comparison",
  abridged_query = rep(c("computer vision", "drug discovery"), each = 4),
  year = rep(2017:2020, 2), n = c(220, 280, 360, 430, 30, 55, 90, 150),
  reference_n = rep(1500, 8),
  comparison_percentage = c(14.7, 18.7, 24, 28.7, 2, 3.7, 6, 10),
  average_comparison_percentage = rep(c(21.5, 5.4), each = 4)
)
class(cmp) <- c("scopus_comparison", class(cmp))
plot_scopus_comparison(cmp)
plot_scopus_comparison(cmp, highlight = "drug discovery")
plot_scopus_comparison(cmp, legend_inside = TRUE)
```

`plot_scopus_intersections` *Plot concept and intersection sizes*

Description

Draws the counts from `scopus_intersections()` as a lollipop chart on a log-scale axis, so a niche of a dozen records stays legible beside a parent literature of many thousands. Rows are ordered by size, with the largest at the top, and one or more rows can be shown in an accent colour, typically a study's own niche. The axis range and the gap between each point and its count label are derived from the data, so the chart reads the same whether the counts span one order of magnitude or six.

Usage

```
plot_scopus_intersections(x, highlight = NULL, highlight_label = NULL, ...)

## S3 method for class 'scopus_intersections'
autoplot(object, ...)
```

Arguments

x	A scopus_intersections object from scopus_intersections().
highlight	Optional character vector of row labels to draw in an accent colour, for example the intersection that defines a study's niche.
highlight_label	Legend label for the highlighted rows. The default, NULL, derives the label from what is highlighted: "Focal intersection" when every highlighted row is an intersection, "Focal concept" when every one is a concept, and "Focal set" for a mixture. Supply a string to use that instead.
...	Currently unused, present for S3 consistency.
object	A scopus_intersections object (for the autoplot() method).

Details

A count of zero cannot be placed on a log axis, so rows whose count is zero or NA are dropped with a warning, which the caption also notes. An empty intersection is itself a finding; the printed object keeps the zero even though the chart cannot.

Value

A ggplot2::ggplot object. Needs the suggested package **ggplot2**.

See Also

scopus_intersections()

Examples

```
sets <- tibble::tibble(
  label = c("semantic priming", "mental simulation",
            "semantic priming \u00d7 mental simulation"),
  query = "q",
  n = c(6600, 2100, 15),
  type = c("concept", "concept", "intersection"),
  size = c(1L, 1L, 2L),
  members = c("semantic priming", "mental simulation",
              "semantic priming; mental simulation")
)
class(sets) <- c("scopus_intersections", class(sets))
plot_scopus_intersections(sets)
plot_scopus_intersections(sets, highlight = sets$label[3])
```

plot_scopus_top *Plot the most frequent values in a record set*

Description

Draws a horizontal bar chart from the output of `scopus_top()`.

Usage

```
plot_scopus_top(x, ...)

## S3 method for class 'scopus_top'
autoplot(object, ...)
```

Arguments

`x`	A `scopus_top` object from `scopus_top()`.
`...`	Currently unused, present for S3 consistency.
`object`	A `scopus_top` object (for the `autoplot()` method).

Value

A ggplot2::ggplot object. Needs the suggested package **ggplot2**.

See Also

`scopus_top()`

Examples

```
# On the bundled corpus of real articles, which needs no key.
plot_scopus_top(scopus_top(example_records, by = "source"))
```

plot_scopus_trend *Plot a publication trend*

Description

Draws annual record counts over time from the output of `scopus_trend()`.

Usage

```
plot_scopus_trend(x, ...)

## S3 method for class 'scopus_trend'
autoplot(object, ...)
```

Arguments

x	A scopus_trend object from scopus_trend().
...	Currently unused, present for S3 consistency.
object	A scopus_trend object (for the autoplot() method).

Value

A ggplot2::ggplot object. Needs the suggested package **ggplot2**.

See Also

scopus_trend()

Examples

```
# Drawn from the bundled corpus of real articles, which needs no key. That
# corpus is a complete harvest, so its rows per year are the publications
# per year its query returns.
by_year <- table(example_records$year)
tr <- tibble::tibble(
  query = "TITLE-ABS-KEY(graphene supercapacitor)",
  year = as.integer(names(by_year)),
  n = as.numeric(by_year)
)
class(tr) <- c("scopus_trend", class(tr))
plot_scopus_trend(tr)
```

---

run_app *Launch the scopusflow app*

Description

Starts a local, code-free Shiny app for building a search, retrieving records, comparing topic trends and exporting the results, with a live terminal that streams the retrieval's progress and a panel that mirrors every choice as runnable R code. A *Demo mode* (on by default) draws records from the bundled example_records corpus and synthesises a topic comparison, so the whole workflow can be explored with no key and no network; switch it off and supply a key to query 'Scopus' for real. The app runs on your own machine: your API key never leaves it, and requests originate from your own network, which is what the 'Scopus' API expects. It needs the suggested packages **shiny**, **bslib** and **callr** (and **ggplot2** for the plots, **fansi** for coloured terminal output).

Usage

```
run_app(host = "127.0.0.1", port = NULL, launch.browser = TRUE)
```

Arguments

host	The address to listen on. Defaults to "127.0.0.1", so the app is reachable only from your own machine and the key is never exposed on the network.
port	The port to listen on, or NULL (default) to choose one.
launch.browser	Logical, whether to open a browser. Passed to shiny::runApp().

Value

Called for its side effect of running the app; does not return until the app is closed.

See Also

scopus_plan(), scopus_fetch_plan()

Examples

```
## Not run:
run_app()

## End(Not run)
```

---

scopus_abstract *Retrieve abstracts and richer metadata*

Description

Fetches the abstract text and core metadata for one or more records from the Elsevier 'Scopus' Abstract Retrieval API. This complements the Search API used elsewhere in the package: a search returns many records with a few fields each, whereas this returns the fuller record, including the abstract, for a known identifier. Passing include adds author keywords and/or the document's reference list to the same request.

Usage

```
scopus_abstract(
  ids,
  by = c("doi", "scopus_id"),
  view = NULL,
  include = character(),
  cache_dir = NULL,
  resume = TRUE,
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

ids	Character vector of identifiers to look up, either Digital Object Identifiers or 'Scopus' record identifiers (with or without the "SCOPUS_ID:" prefix), according to by.
by	Either "doi" or "scopus_id", the kind of identifier in ids.
view	Optional character scalar naming the Abstract Retrieval view to request: one of "META", "META_ABS", "REF" or "FULL". NULL (the default) omits the view parameter from the request entirely, exactly as scopusflow has always done, so existing calls that never mention view are unaffected. Retrieving include = "references" requires view = "FULL" or view = "REF"; see *Details* for how the two differ and which to prefer.
include	Optional character vector naming extra fields to retrieve in the same request: "references" and/or "keywords". "references" requires view = "FULL" or view = "REF", and "keywords" requires view = "FULL", since the REF response carries no author keywords. Either way this is an entitlement that is separate from ordinary abstract access and from 'Scopus' Search access, and, per Elsevier's own documentation, some fields (notably author keywords) may need to be requested from your Scopus/Elsevier account contact even when the view itself is otherwise accessible. See *Details*.
cache_dir	Optional directory for per-identifier cache files, as in scopus_fetch_plan(). NULL (the default) performs no caching. Worth setting whenever include is used: Abstract Retrieval draws on its own weekly quota, smaller than and separate from Search's, and every identifier here costs its own request, so re-running an interrupted batch without a cache re-spends quota already spent.
resume	Logical. When TRUE (the default) and cache_dir is set, an identifier whose cache file already exists is loaded from disk, sparing a second request for it. A cache file that cannot be read back, for example one left half-written by an interrupted run, is retrieved again with a warning, and the batch carries on.
api_key, inst_token	Optional credentials (see scopus_has_key()).
verbose	Logical. When TRUE, progress is reported.

Details

Retrieving references needs Abstract Retrieval's FULL or REF view, and keywords need FULL. In development, against a live key with full Abstract Retrieval access, view = "FULL" returned a complete, correctly counted reference list for every document tried. view = "REF" returned the identical, complete list in one case but a truncated (paginated) subset in another, on an otherwise identical request made moments apart, so "FULL" is recommended when your entitlement allows it. "REF" remains available for accounts entitled only to it; when the number of references returned does not match the document's own reported reference count, a warning is issued naming the identifier, since the list may be an incomplete page of the bibliography.

Author keywords were not populated by either 'Scopus' Search's COMPLETE view (see scopus_records()) or Abstract Retrieval's FULL view in this package's own development testing, against a live, otherwise fully-entitled key, on documents that do carry author keywords in 'Scopus' itself. If your own keywords come back all NA, this is most likely an entitlement gap specific to that field, worth raising with your Scopus/Elsevier account contact. The documents do carry keywords.

Value

A tibble of class `scopus_abstracts`, one row per identifier, with columns `id` (the input identifier), `scopus_id`, `doi`, `title`, `abstract`, `publication`, `year` and `citations`. A field the API does not return is `NA`. An identifier that cannot be retrieved (for example one not in 'Scopus') yields a row of `NA`s with a warning, so a batch is not lost to a single failure. The number of Abstract Retrieval requests made and the most recently parsed quota (see `scopus_quota()`) are attached as the `n_requests` and `quota` attributes, since this is a materially more expensive operation than a search call.

When `include` names `"keywords"`, an `authkeywords` column is added: the document's author-supplied keywords, joined the same way as `authors` (`"; "`-separated), or `NA` when the document has none, or when the API omits the field for a given key's entitlement (see *Details*).

When `include` names `"references"`, a `references` list-column is added: one data frame per document, with one row per cited work, where a joined string would have to be parsed apart again. Its columns are `position` (the reference's place in the bibliography), `id` (the 'Scopus' identifier of the cited work, when resolved), `doi`, `title`, `authors`, `source` (the journal or other venue), `year` and `citedbycount` (the cited work's own citation count; populated only under `view = "REF"`, `NA` under `"FULL"`). A document with no resolvable references yields a zero-row data frame, so the column can always be unnested.

API access

This performs one request per identifier and requires a valid API key and internet access; full-text abstract access, and the `FULL`/`REF` views in particular, can also depend on your entitlement. A view or field your key is not entitled to raises a `scopus_error_forbidden` condition with a message naming the view and suggesting who to contact, where a generic HTTP failure would leave you guessing. Because entitlement is a property of the account, retrieval stops at the first such failure, so the same refusal is not repeated for every remaining identifier. See the *API access* section of `scopus_count()` for the other conditions that may be raised.

See Also

`scopus_fetch()`, `scopus_extract_dois()`, `scopus_corpus()` to assemble a minimal keyword/reference corpus across many documents.

Examples

```
# One record of the bundled corpus, looked up for real.
scopus_abstract(scopus_extract_dois(example_records)[1])

# Author keywords and a structured reference list, in the same request.
# Costs one Abstract Retrieval request per identifier, against a smaller,
# separate weekly quota from Search; see the API access section above for
# the entitlement this needs.
rich <- scopus_abstract(
  "10.1038/natrevmats.2016.33",
  view = "FULL", include = c("references", "keywords")
)
rich$references[[1]]
```

```
# The offline companion, which needs no key. The identifiers, titles,
# sources and citation counts are two records of the bundled corpus of real
# articles; the abstract text is what a live call adds, so it is left unset
# left as a placeholder here, as is the 'Scopus' identifier the corpus does
# not carry.
cited <- example_records[order(-example_records$citations), ][1:2, ]
abstracts <- tibble::tibble(
  id = cited$doi,
  scopus_id = NA_character_,
  doi = cited$doi,
  title = cited$title,
  abstract = NA_character_,
  publication = cited$publication,
  year = cited$year,
  citations = cited$citations
)
class(abstracts) <- c("scopus_abstracts", class(abstracts))
abstracts

# A reference list arrives as one data frame per document, in the
# `references` list-column added by include = "references". The corpus
# carries no bibliographies, so its own records fill the columns here,
# standing in for the works the first document cites.
refs <- example_records[1:3, ]
abstracts$references <- list(
  tibble::tibble(
    position = as.character(seq_len(nrow(refs))),
    id = NA_character_,
    doi = refs$doi,
    title = refs$title,
    authors = refs$authors,
    source = refs$publication,
    year = refs$year,
    citedbycount = refs$citations
  ),
  tibble::tibble()
)
abstracts$references[[1]]
```

`scopus_cache_clear` *Clear the scopusflow managed cache*

Description

Deletes the cache files written under `scopus_cache_dir()`. A cache you created in a directory of your own is left untouched.

Usage

```
scopus_cache_clear()
```

Value

Invisibly, TRUE once the managed cache directory is removed or found to be absent.

Examples

```
# Safe to call even when nothing is cached.
scopus_cache_clear()
```

`scopus_cache_dir` *Managed cache directory for scopusflow*

Description

Returns (and creates on request) a per-user cache directory under `tools::R_user_dir()`, suitable for passing to `cache_dir` in `scopus_fetch_plan()`. The cache is entirely optional and can be cleared with `scopus_cache_clear()`.

Usage

```
scopus_cache_dir(create = FALSE)
```

Arguments

`create`	Logical. When TRUE, the directory is created if it is absent.

Value

The cache directory path, invisibly when `create = TRUE`.

Examples

```
scopus_cache_dir(create = FALSE)
```

`scopus_combine` *Combine record sets into one*

Description

Binds several scopus_records objects into a single one, renumbering `entry_number` across the result and, optionally, dropping duplicates. This is the safe way to merge separate fetches: plain `rbind()` would leave duplicate entry numbers, and `c()` would return a list.

Usage

```
scopus_combine(..., dedupe = FALSE)

## S3 method for class 'scopus_records'
c(x, ...)
```

Arguments

...	Two or more scopus_records objects, or a single list of them.
dedupe	Logical. When TRUE, records sharing a 'Scopus' identifier, or failing that a DOI (compared case-insensitively), are kept once.
x	A scopus_records object (for the c() method).

Value

A scopus_records tibble. Per-retrieval attributes such as total_results are not carried over, since they describe a single fetch. The merge itself is recorded in the combined attribute, a list of n_in (records supplied), n_out (records kept), n_removed and deduplicated, which is what lets scopus_search_report() state how many duplicates were removed, so the PRISMA-S item is answered.

See Also

scopus_fetch_plan(), which combines plan cells the same way.

Examples

```
# A baseline retrieval and a later one, merged into a cumulative set. The
# bundled corpus of real articles stands in for both, since 'Scopus'
# records may not be redistributed.
baseline <- example_records[example_records$year <= 2023, ]
later <- example_records
combined <- scopus_combine(baseline, later, dedupe = TRUE)
nrow(combined)

# Those records carry no 'Scopus' identifier, so de-duplication falls back
# to the DOI. The eleven that arrived without one cannot be matched, and so
# survive in both copies, which is why 138 distinct articles come back as
# 149 rows.
sum(is.na(example_records$doi))
```

---

scopus_compare_topics *Compare publication trends across topics*

Description

Compares how often a set of *comparison* topics co-occur with a *reference* topic over time. For each year and each comparison term, the number of records matching the reference combined with that term is expressed as a percentage of the records matching the reference alone. This reveals which sub-topics are growing or shrinking within a literature.

Usage

```
scopus_compare_topics(
  reference_query,
  comparison_terms,
  years,
  field = NULL,
  view = c("STANDARD", "COMPLETE"),
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

`reference_query`	Character scalar. The reference topic that anchors the comparison (for example `"language learning"`).
`comparison_terms`	Character vector of topics to compare against the reference (for example `c("effect size", "Bayesian")`). Each is combined with the reference using a logical AND.
`years`	Integer vector of publication years to span (for example `2015:2020`).
`field`	Optional 'Scopus' field tag applied to every component of every query (see `scopus_plan()`).
`view`	Either `"STANDARD"` or `"COMPLETE"`.
`api_key, inst_token`	Optional credentials (see `scopus_has_key()`).
`verbose`	Logical. When `TRUE`, progress is reported.

Value

A tibble of class `scopus_comparison` with the columns `query` (the full query used), `query_type` (`"reference"` or `"comparison"`), `abridged_query` (the topic label for plotting), `year`, `n` (records that year, as a double so very large counts are exact), `reference_n` (reference records that year, likewise a double), `comparison_percentage` (`100 * n / reference_n`, or `NA` when `reference_n` is 0) and `average_comparison_percentage` (the same ratio computed on period totals, over the years where both counts are available). Comparison rows are sorted by descending average percentage.

API access

This performs one count request per term per year, so it requires a valid API key and internet access. The *API access* section of `scopus_count()` gives the details. A modest number of terms and years keeps the call within quota.

See Also

`plot_scopus_comparison()` to visualise the result.

Examples

```
cmp <- scopus_compare_topics(
  reference_query = "deep learning",
  comparison_terms = c("computer vision", "drug discovery"),
  years = 2018:2022,
  field = "TITLE-ABS-KEY"
)
cmp

# The shape of the return value, built offline so it runs without a key.
years <- 2018:2022
ref_n <- c(4200, 5600, 7100, 8600, 10200)
counts <- list(`computer vision` = c(1500, 2000, 2500, 3000, 3600),
               `drug discovery`  = c(180, 260, 370, 500, 660))
cmp <- tibble::tibble(
  query = "TITLE-ABS-KEY(deep learning)",
  query_type = c(rep("reference", length(years)),
                 rep("comparison", length(counts) * length(years))),
  abridged_query = c(rep("deep learning", length(years)),
                     rep(names(counts), each = length(years))),
  year = rep(years, length(counts) + 1),
  n = c(ref_n, unlist(counts, use.names = FALSE)),
  reference_n = rep(ref_n, length(counts) + 1),
  comparison_percentage = 100 * c(ref_n, unlist(counts, use.names = FALSE)) /
    rep(ref_n, length(counts) + 1),
  average_comparison_percentage = c(rep(100, length(years)),
                                    rep(c(35.3, 5.4), each = length(years)))
)
class(cmp) <- c("scopus_comparison", class(cmp))
cmp
```

---

scopus_corpus *Assemble a minimal, cross-tool corpus with keywords and references*

Description

Takes a `scopus_records()` tibble, such as the output of `scopus_fetch()` or `scopus_fetch_plan()`, and enriches it with author keywords and structured references via Abstract Retrieval, returning a minimal, uniform shape close to what OpenAlex's `works` API already returns: an `id`, `title`, `year`, `keywords` (a list-column of character vectors) and `references` (a list-column of data frames). This is meant for downstream tools that want to consume 'Scopus' output without writing their own parsing layer, for example for keyword co-occurrence or citation-network analysis. It does not replace `as_bibliometrix()`, which keeps its own established field-mapping convention for users who want bibliometrix's tag names instead.

Usage

```
scopus_corpus(
  records,
```

```
  by = c("doi", "scopus_id"),
  view = c("FULL", "REF"),
  cache_dir = NULL,
  resume = TRUE,
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

records	A scopus_records() tibble, or any data frame with doi, title and year columns in the same shape.
by	Either "doi" or "scopus_id", the kind of identifier in records to look records up by (see scopus_abstract()).
view	Either "FULL" (the default) or "REF", passed to scopus_abstract(). "FULL" is recommended: in development, it returned a complete, correctly counted reference list for every document tried, while "REF" returned an inconsistent, sometimes-truncated subset (see scopus_abstract()'s documentation for the details and for the entitlement each view needs). The REF response also carries no author keywords, so under that view only the references are requested and keywords is empty for every record.
cache_dir, resume	As in scopus_abstract(): an optional directory for per-identifier cache files, and whether an existing one is reused. Worth setting for anything beyond a handful of records, since this performs one Abstract Retrieval request per record, against its own, smaller weekly quota.
api_key, inst_token	Optional credentials (see scopus_has_key()).
verbose	Logical. When TRUE, progress is reported.

Value

A tibble with columns id (the identifier records was looked up by), title, year, keywords (a list-column: a character vector of the document's author keywords, split out of scopus_abstract()'s joined authkeywords string, empty when the document has none, the field is unavailable or view = "REF") and references (a list-column: each entry is the references data frame scopus_abstract() returns for that document, with one row per cited work). A record in records whose identifier is NA is dropped, with a warning naming how many.

API access

This performs one Abstract Retrieval request per usable record, on top of whatever retrieved records in the first place; see scopus_abstract()'s *API access* section for the entitlement view = "FULL"/"REF" needs and how a 403 is handled.

See Also

scopus_abstract(), as_bibliometrix()

Examples

```
# Costs one Abstract Retrieval request per record, against a smaller,
# separate weekly quota from Search; see the API access section above.
recs <- scopus_fetch("DOI(10.1038/natrevmats.2016.33)", max_results = 1)
corpus <- scopus_corpus(recs)
corpus$keywords[[1]]
corpus$references[[1]]

# The offline companion, which needs no key: one row per document, with
# keywords and references as list-columns. The identifiers, titles and
# years are records of the bundled corpus of real articles, which stands in
# for a harvest because 'Scopus' records may not be redistributed. It holds
# neither keywords nor bibliographies, so those are illustrative.
docs <- example_records[order(-example_records$citations), ][1:2, ]
corpus <- tibble::tibble(
  id = docs$doi,
  title = docs$title,
  year = docs$year,
  keywords = list(
    c("graphene", "supercapacitor", "energy storage"),
    c("laser-induced graphene", "flexible electronics")
  ),
  references = list(
    tibble::tibble(
      position = c("1", "2"),
      id = NA_character_,
      doi = example_records$doi[3:4],
      title = example_records$title[3:4],
      authors = example_records$authors[3:4],
      source = example_records$publication[3:4],
      year = example_records$year[3:4],
      citedbycount = example_records$citations[3:4]
    ),
    tibble::tibble()
  )
)
corpus
corpus$keywords[[1]]
corpus$references[[1]]
```

---

scopus_count *Count 'Scopus' results for a query*

Description

Retrieves only the total number of records matching a query, without downloading them. This is the inexpensive way to size a retrieval before committing quota. The count can guide how to partition a scopus_plan(), or simply report how large a topic is.

Usage

```
scopus_count(
  query,
  years = NULL,
  field = NULL,
  view = c("STANDARD", "COMPLETE"),
  api_key = NULL,
  inst_token = NULL
)
```

Arguments

query	Character scalar. The base search expression.
years	Optional integer vector of publication years to restrict to.
field	Optional 'Scopus' field tag to wrap the query in (see scopus_plan()).
view	Either "STANDARD" or "COMPLETE". COMPLETE adds an authkeywords column to scopus_fetch()/scopus_fetch_plan() output (see scopus_records()) at no extra cost beyond COMPLETE's own smaller page size, which already means more requests, and so more quota, for the same number of records.
api_key, inst_token	Optional credentials, resolved by default from options or environment variables (see scopus_has_key()).

Value

A single number giving the total number of matching records, or NA when the API reports no total. It is returned as a double so that very large totals are represented exactly, with no risk of overflow, and the parsed quota (see scopus_quota()) attached as the quota attribute so a workflow can pace itself off a count.

API access

This function performs a network request and therefore requires a valid API key and internet access. When no key is configured it raises a scopus_error_no_key condition, and other failures raise typed scopus_error subclasses such as scopus_error_rate_limit. A tryCatch() around the call lets a workflow handle these gracefully.

Examples

```
scopus_count("graphene supercapacitor", years = 2015:2024,
             field = "TITLE-ABS-KEY")

# The offline companion, which needs no key: one number with the parsed
# quota attached. The bundled corpus of real articles is a complete harvest
# of its own query, so its row count is the total that query returned. The
# quota attribute is parsed from real response headers by scopus_quota(),
# so it cannot drift from what a live call attaches.
resp <- httr2::response(
  status_code = 200,
```

```
  headers = list(
    `X-RateLimit-Limit` = "20000",
    `X-RateLimit-Remaining` = "19987",
    `X-RateLimit-Reset` = "1700000000"
  )
)
n <- nrow(example_records)
attr(n, "quota") <- scopus_quota(resp)
n
```

`scopus_diff_dois` *Compare two DOI retrievals*

Description

Identifies which DOIs were added, removed or unchanged between an earlier and a later retrieval. This supports change tracking: re-running a search later and seeing exactly what is new.

Usage

```
scopus_diff_dois(old, new)
```

Arguments

`old`, `new`	scopus_records objects or character vectors of DOIs, representing the earlier (`old`) and later (`new`) retrievals.

Value

A tibble of class `scopus_doi_diff` with columns `doi` and `status`, where `status` is an ordered factor with levels `"added"` (in `new` only), `"removed"` (in `old` only) and `"unchanged"` (in both). Rows are sorted by status then DOI, and printing shows the counts in each category.

See Also

`scopus_extract_dois()`

Examples

```
# A baseline retrieval and the same search re-run a year later, both taken
# from the bundled corpus of real articles: the second pull has gained the
# 2024 records and lost the first one to re-indexing.
baseline <- example_records[example_records$year <= 2023, ]
later <- example_records[-1, ]
scopus_diff_dois(old = baseline, new = later)
```

`scopus_extract_dois` *Extract, clean and optionally export DOIs*

Description

Pulls Digital Object Identifiers from a scopus_records object (or a bare character vector), normalises them and removes missing values. The resulting list can be imported into a reference manager such as Zotero to assemble a bibliography.

Usage

```
scopus_extract_dois(x, dedupe = TRUE, file = NULL)
```

Arguments

`x`	A scopus_records tibble, or a character vector of DOIs.
`dedupe`	Logical, dropping duplicate DOIs by default.
`file`	Optional path at which to write the DOIs as a single-column CSV. A file is written only when this argument is supplied, and only to the exact path given, so the package always leaves the working directory untouched unless asked. Parent directories are assumed to exist already.

Details

Normalisation trims surrounding whitespace and strips common resolver prefixes (`https://doi.org/`, `http://dx.doi.org/`, `doi:`) so that the same article is counted once even when its DOI is formatted differently in two records. Because DOIs are case-insensitive, comparison and deduplication ignore case, while the output keeps the original casing.

Value

A character vector of cleaned DOIs, returned invisibly when `file` is written.

See Also

`scopus_diff_dois()` to compare two retrievals.

Examples

```
# The bundled corpus of real articles stands in for a harvest of your own,
# since 'Scopus' records may not be redistributed.
dois <- scopus_extract_dois(example_records)
length(dois)
head(dois, 3)

# Eleven of its 138 records arrived without a DOI, as records do, and so
# drop out of the list.
sum(is.na(example_records$doi))
```

```
# The same cleaning applies to a bare vector, so a resolver prefix or a
# difference in case does not make one article look like two.
scopus_extract_dois(c("https://doi.org/10.1/A", "doi: 10.1/a", "10.2/B"))

# Write to a temporary file (never the working directory).
path <- tempfile(fileext = ".csv")
scopus_extract_dois(example_records, file = path)
```

`scopus_fetch` *Fetch 'Scopus' records for a query*

Description

Retrieves records page by page, accumulating them and returning a single normalised scopus_records tibble. Pagination, the API's hard `start` < `5000` ceiling, rate-limit handling and retry with back-off are all managed for you.

Usage

```
scopus_fetch(
  query,
  max_results = Inf,
  view = c("STANDARD", "COMPLETE"),
  page_size = NULL,
  field = NULL,
  years = NULL,
  cursor = FALSE,
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

`query`	Character scalar. The base search expression.
`max_results`	Maximum number of records to retrieve. Defaults to `Inf`, meaning all available records up to the API ceiling. With the default offset-based paging the 'Scopus' Search API refuses offsets of 5000 or more, so a single query yields at most 5000 records; set `cursor = TRUE`, or partition the search by year with `scopus_plan()`, to go beyond that.
`view`	Either `"STANDARD"` or `"COMPLETE"`. `COMPLETE` adds an `authkeywords` column to `scopus_fetch()`/`scopus_fetch_plan()` output (see `scopus_records()`) at no extra cost beyond `COMPLETE`'s own smaller page size, which already means more requests, and so more quota, for the same number of records.
`page_size`	Integer records per page, or `NULL` (default) to use the most quota-efficient page the view allows (200 for `STANDARD`, 25 for `COMPLETE`). See `scopus_plan()` for why larger pages cost less quota.

field	Optional 'Scopus' field tag to wrap the query in (see scopus_plan()).
years	Optional integer vector of publication years to restrict to.
cursor	Logical. When TRUE, retrieve the result set with cursor-based pagination, which has no 5000-record ceiling, so an entire large query can be harvested in one call. The records then arrive in the API's deep-paging order, which is no longer relevance order. As a safeguard against a non-conforming server that never signals the end, cursor paging stops after getOption("scopusflow.max_cursor_pages", 1e5) pages with a warning; set that option to Inf to remove the ceiling.
api_key, inst_token	Optional credentials, resolved by default from options or environment variables (see scopus_has_key()).
verbose	Logical. When TRUE, progress is reported as the retrieval proceeds.

Value

A scopus_records tibble. The reported total and the most recent parsed quota are attached as the total_results and quota attributes, the harvest is dated by retrieved_at (a POSIXct) and scopusflow_version, and paging records whether it was retrieved by offset or by cursor. The date and version matter because citations is a snapshot value that keeps moving, so two saved sets are only comparable if each records when it was taken, and scopus_search_report() reads all of them back. They survive a .rds round trip through write_scopus_records() but not a .csv one, which carries columns only.

API access

Requires a valid API key and internet access. The *API access* section of scopus_count() lists the conditions that may be raised.

See Also

scopus_fetch_plan() for cached, resumable, partitioned retrieval.

Examples

```
recs <- scopus_fetch("graphene supercapacitor", field = "TITLE-ABS-KEY",
                     max_results = 50)
recs

# The offline companion, which needs no key. 'Scopus' records may not be
# redistributed, so the package bundles a corpus of real articles in this
# same schema; a live harvest returns exactly this shape.
recs <- example_records
recs
nrow(recs)
is_scopus_records(recs)
```

`scopus_fetch_plan` *Execute a 'Scopus' search plan, with optional caching and resume*

Description

Runs every cell of a `scopus_plan()` in turn, optionally caching each cell's result so that an interrupted or quota-limited retrieval can resume without re-spending quota on the cells already fetched. Results are accumulated and bound once into a single scopus_records tibble.

Usage

```
scopus_fetch_plan(
  plan,
  max_results = Inf,
  cache_dir = NULL,
  resume = TRUE,
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

`plan`	A `scopus_plan` object from `scopus_plan()`.
`max_results`	Maximum records to retrieve per cell (default `Inf`).
`cache_dir`	Optional directory for per-cell cache files. The default of `NULL` performs no caching. Pass an explicit path you control, or `scopus_cache_dir()` to use a managed, clearable cache under `tools::R_user_dir()`. Caching happens only when you opt in through this argument. A cache directory serves one plan: cells are checkpointed by their position in the plan and their year, so give each distinct plan its own directory. As a safeguard, a checkpoint is served only when the query, year, view and page size it was fetched under all match the plan cell, and only when its own `max_results` did not truncate it below what is being asked for now; anything else is a cache miss, refetched and overwritten. A checkpoint holding more records than the current `max_results` asks for is served trimmed to that cap, the fuller set staying on disk. A checkpoint that cannot be read back, for example one left half-written by an interrupted run, is also treated as a miss, and never aborts the harvest.
`resume`	Logical. When `TRUE` and `cache_dir` is set, a cell whose cache file already exists is loaded from disk, sparing a second request.
`api_key, inst_token`	Optional credentials (see `scopus_has_key()`).
`verbose`	Logical. When `TRUE`, per-cell progress is reported.

Value

A scopus_records tibble combining all cells, with the originating `plan` attached as the `plan` attribute and the per-cell accounting as `cell_totals`, a tibble of `cell`, `date`, `n_records` and `reported_total` (the count the API gave for that cell, `NA` where it gave none). `total_results` is their sum, and is `NA` unless every cell reported one, since a partial sum would understate the search while looking like a real figure. A cell that comes back shorter than the API said it should warns, because a truncated or refused download otherwise arrives as a merely small result set; a cell stopped by `max_results` is short by request and does not warn. `scopus_search_report()` reads all of this back. The `retrieved_at` and `scopusflow_version` attributes described in `scopus_fetch()` are carried across from the cells: the time is the earliest of them, since a combined set is only as fresh as its oldest cell, and every version that contributed is listed, since resuming an older cache means more than one did. Both are omitted when any cell cannot supply them, as a checkpoint written before they existed cannot. Dating the whole from the part of it that can be dated would misreport the set.

API access

Any cell not served from cache requires a valid API key and internet access. The *API access* section of `scopus_count()` gives the details.

See Also

`scopus_cache_dir()`, `scopus_cache_clear()`

Examples

```
plan <- scopus_plan("graphene supercapacitor", years = 2015:2024,
                    field = "TITLE-ABS-KEY", partition = "year")
dir <- file.path(tempdir(), "graphene-cache")
# `max_results` caps each yearly cell, so the example stays small and
# quota-light; drop it to harvest every record in the plan.
recs <- scopus_fetch_plan(plan, max_results = 25, cache_dir = dir, resume = TRUE)

# The offline companion, which needs no key: a record set with the plan
# that describes it attached. 'Scopus' records may not be redistributed, so
# the bundled corpus of real articles stands in for the harvest, and the
# plan describes the same search, one cell per year.
plan <- scopus_plan("graphene supercapacitor", years = 2015:2024,
                    field = "TITLE-ABS-KEY", partition = "year")
recs <- example_records
attr(recs, "plan") <- plan
recs
attr(recs, "plan")
```

`scopus_field_tags` *Recognised 'Scopus' field tags*

Description

Lists the field tags that scopus_plan(), scopus_fetch() and scopus_compare_topics() understand, together with a short note on what each one searches. Passing one of these tags as field restricts a query to the corresponding part of a record, so TITLE-ABS-KEY looks in the title, abstract and keywords while AUTH looks only at author names. Other valid 'Scopus' tags are accepted too. This is a guide to the common ones.

Usage

```
scopus_field_tags()
```

Value

A tibble with a tag column and a searches column describing the scope of each tag.

See Also

scopus_plan()

Examples

```
scopus_field_tags()
```

---

scopus_has_key *Locate the 'Scopus' API key and institutional token*

Description

scopus_has_key() reports whether an API key can be found, without revealing it. The key itself is resolved internally and is never printed by the package.

Usage

```
scopus_has_key()
```

Details

The key is looked up first from the api_key argument of whichever function is being called, then from the scopusflow.api_key option, and finally from the SCOPUS_API_KEY environment variable. An optional institutional token, used for off-campus access to subscriber content, is resolved the same way from the inst_token argument, the scopusflow.inst_token option, or the SCOPUS_INST_TOKEN environment variable.

A key is a secret. The safest home for it is ~/.Renviron, as in SCOPUS_API_KEY=xxxx, well away from any script, and it should stay out of version control.

Value

A length-one logical that is safe to print, TRUE when a non-empty key is available and FALSE otherwise.

See Also

scopus_count(), scopus_fetch()

Examples

```
# Does the current session have a key configured?
scopus_has_key()
```

---

scopus_intersections *Count a set of concepts and their intersections*

Description

Counts how many records match each of a named set of concepts, and each requested intersection of those concepts. This gives a size-of-field snapshot that shows where a study or a niche sits within a wider literature: one field may hold thousands of records and another hundreds, while their intersection holds a dozen. Where scopus_compare_topics() tracks topics' shares of a reference over time, this sizes a set of concepts and their overlap at a single point. Like scopus_count(), it retrieves totals only, never records, so a whole landscape costs one request per row of the result.

Usage

```
scopus_intersections(
  concepts,
  intersections = NULL,
  abbrev = NULL,
  sep = " × ",
  years = NULL,
  field = NULL,
  view = c("STANDARD", "COMPLETE"),
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

concepts	Named character vector. The names are display labels and the values are search terms (wrapped in field when one is given) or complete field-tagged query expressions (used as-is). The labels must be unique.

`intersections` Optional list of character vectors, each naming two or more distinct concept labels whose intersection should be counted, for example `list(c("A", "B"), c("A", "B", "C"))`. A single character vector is taken as one intersection.

`abbrev` Optional named character vector of short labels, keyed by concept label and used only when composing intersection labels, so those rows stay readable while the concept rows keep their full names.

`sep` Separator joining the member labels in an intersection label. Defaults to a multiplication sign between spaces.

`years` Optional integer vector of publication years to restrict to.

`field` Optional 'Scopus' field tag wrapped around each concept value that is not already a complete field-tagged expression (see `scopus_field_tags()`).

`view` Either `"STANDARD"` or `"COMPLETE"`. `COMPLETE` adds an `authkeywords` column to `scopus_fetch()`/`scopus_fetch_plan()` output (see `scopus_records()`) at no extra cost beyond `COMPLETE`'s own smaller page size, which already means more requests, and so more quota, for the same number of records.

`api_key`, `inst_token` Optional credentials, resolved by default from options or environment variables (see `scopus_has_key()`).

`verbose` Logical. When `TRUE`, progress is reported.

Details

A concept value that already reads as a complete field-tagged expression, such as `"TITLE(virtual reality)"`, is used exactly as given, so `field` never wraps it a second time, which the API would reject as malformed. Any other value is treated as a bare term and wrapped in `field` when one is supplied. An intersection is counted by joining its members' queries with `AND`, each part in parentheses.

Value

A tibble of class `scopus_intersections` with one row per concept and per intersection: `label` (the display label), `query` (the exact query counted), `n` (the count, as a double so very large totals are exact), `type` (`"concept"` or `"intersection"`), `size` (the number of member concepts) and `members` (the member labels, joined by `"; "`). A row whose response omits a total is recorded as `NA`, with a warning. The `years` restriction, when given, is stored in the `years` attribute.

API access

This performs one count request per concept and per intersection, so it requires a valid API key and internet access; see the *API access* section of `scopus_count()`.

See Also

`plot_scopus_intersections()` to visualise the result, and `scopus_count()` for a single query.

Examples

```
sets <- scopus_intersections(
  concepts = c(
    "semantic priming"  = "semantic priming",
    "mental simulation" = "mental simulation"
  ),
  intersections = list(c("semantic priming", "mental simulation")),
  field = "TITLE-ABS-KEY"
)
sets

# The shape of the return value, built offline so it runs without a key.
sets <- tibble::tibble(
  label = c("semantic priming", "mental simulation",
            "semantic priming \u00d7 mental simulation"),
  query = c("TITLE-ABS-KEY(semantic priming)",
            "TITLE-ABS-KEY(mental simulation)",
            paste("(TITLE-ABS-KEY(semantic priming)) AND",
                  "(TITLE-ABS-KEY(mental simulation))")),
  n = c(6600, 2100, 15),
  type = c("concept", "concept", "intersection"),
  size = c(1L, 1L, 2L),
  members = c("semantic priming", "mental simulation",
              "semantic priming; mental simulation")
)
class(sets) <- c("scopus_intersections", class(sets))
sets
```

`scopus_plan` *Build a reproducible 'Scopus' search plan*

Description

A *plan* is a fully specified, inspectable description of one or more 'Scopus' queries to run. Splitting the act of *describing* a search from *executing* it makes workflows reproducible (the plan can be saved, reviewed and version controlled) and lets large retrievals be partitioned, for example one cell per year, so they can be cached and resumed.

Usage

```
scopus_plan(
  query,
  years = NULL,
  field = NULL,
  view = c("STANDARD", "COMPLETE"),
  page_size = NULL,
  partition = c("none", "year")
)

is_scopus_plan(x)
```

Arguments

query	Character scalar. The base search expression, without field tags or year filters (these are added through field and years).
years	Optional integer vector of publication years to restrict to, for example 2015:2020. When partition = "year", one plan cell is created for each distinct year. Otherwise the minimum and maximum define a single date range.
field	Optional character scalar naming a 'Scopus' field tag to wrap the query in, for example "TITLE-ABS-KEY", "TITLE", "AUTH" or "AFFIL". When NULL, the query is used verbatim. See scopus_field_tags() for the common tags.
view	Either "STANDARD" or "COMPLETE". COMPLETE returns more fields, including an authkeywords column (see scopus_records()), but requires a subscriber entitlement and is limited to a smaller page size, which means more requests, and so more quota, for the same number of records. Even where COMPLETE view itself is accessible, the returned author keywords can still be gated separately by your own account (see scopus_records()).
page_size	Integer number of records to request per page, or NULL (the default) to use the largest page the view allows. The 'Scopus' Search API permits up to 200 records per page for the STANDARD view but only 25 for COMPLETE. Because the weekly quota is charged per request, requesting the maximum page size keeps the number of requests, and so the quota, as low as possible for a given result set. Lower it only where you have a reason to.
partition	Either "none" (a single query cell) or "year" (one cell per year in years). Partitioning by year is the recommended way to stay under the API's hard limit of start < 5000.
x	An object to test or print.

Value

A tibble of class scopus_plan, one row per cell, with columns cell, query (field-wrapped), date (year range string or NA), year (integer or NA), view and page_size. Plan-level settings are stored as attributes.

is_scopus_plan() returns a length-one logical.

See Also

scopus_fetch_plan() to execute a plan, scopus_count() to size it.

Examples

```
scopus_plan("quantum computing", years = 2015:2022, field = "TITLE-ABS-KEY")
scopus_plan("immunotherapy", years = 2010:2020, partition = "year")
```

scopus_query *Build a field-tagged 'Scopus' query*

Description

Combines several terms into one 'Scopus' query string, optionally wrapping each in a field tag and joining them with a boolean operator. It is a tidier alternative to pasting query fragments together by hand, which is where field-tag and bracket mistakes tend to creep in.

Usage

```
scopus_query(..., .op = c("AND", "OR", "AND NOT"), .field = NULL)
```

Arguments

`...`	Character terms to combine, for example `"language learning"` and `"effect size"`.
`.op`	The boolean operator joining the terms, one of `"AND"`, `"OR"` or `"AND NOT"`.
`.field`	Optional field tag applied to every term (see `scopus_field_tags()`).

Value

A length-one character string suitable for `scopus_count()`, `scopus_fetch()` or the `query` of `scopus_plan()`.

See Also

`scopus_field_tags()`, `scopus_plan()`

Examples

```
scopus_query("climate change", "adaptation", .field = "TITLE-ABS-KEY")
scopus_query("graphene", "supercapacitor", .op = "AND")
scopus_query("CRISPR", "Cas9", "Cas12", .op = "OR")
```

scopus_quota *Parse 'Scopus' quota and rate-limit headers*

Description

Elsevier returns the caller's weekly quota and short-term rate-limit status in response headers. `scopus_quota()` extracts them into a tidy list so a workflow can pause, schedule or report on the remaining allowance.

Usage

```
scopus_quota(resp)
```

Arguments

resp	An httr2::response object, typically captured during a request.

Details

The relevant headers are `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `X-RateLimit-Reset` (epoch seconds), `X-ELS-Status` and `Retry-After`. When the API raises a quota or rate-limit error, the parsed quota is also attached to the resulting condition, where it is available as `cnd$quota`.

Value

A list with elements `limit`, `remaining`, `reset` (a `POSIXct` time at which the rate-limit window resets, or `NA`), `status` and `retry_after` (seconds, or `NA`). A missing header yields `NA`.

Examples

```
# Build a fake response to show the shape of the output (no network used).
resp <- httr2::response(
  status_code = 200,
  headers = list(
    `X-RateLimit-Limit` = "20000",
    `X-RateLimit-Remaining` = "19987",
    `X-RateLimit-Reset` = "1700000000"
  )
)
scopus_quota(resp)
```

---

scopus_search_report *Assemble a reproducible record of a 'Scopus' search*

Description

Turns a harvest, or a plan not yet run, into the search-strategy record a systematic review has to report: what was searched, exactly how, when, how much came back, and how much the API said there was. The record prints as a readable report, formats as a methods paragraph fit to paste into a manuscript, and writes as Markdown. The reporting standard it follows is PRISMA-S (Rethlefsen et al., 2021), together with the identification counts of the PRISMA 2020 flow diagram.

Usage

```
scopus_search_report(x, plan = NULL, file = NULL)

## S3 method for class 'scopus_search_report'
format(x, style = c("report", "paragraph", "markdown"), ...)

## S3 method for class 'scopus_search_report'
print(x, ...)
```

Arguments

x	A scopus_records object, which supplies the counts and the retrieval provenance and, through its plan attribute, the plan; or a bare scopus_plan() for a search that has not been run yet. For format() and print(), the report object itself.
plan	Optional scopus_plan() describing x. Supply it when a record set does not carry one, for example one retrieved with scopus_fetch() or read back from a .csv. An explicit plan takes precedence over the one x carries.
file	Optional path at which to write the record as Markdown. A file is written only when this argument is supplied, and only to the exact path given, so the package leaves the working directory untouched unless asked. Parent directories are assumed to exist already.
style	Which rendering to return: "report" for the readable record that print() shows, "paragraph" for the methods paragraph, or "markdown" for the whole record as Markdown, which is what file writes.
...	Ignored, present for compatibility with the generics.

Details

Everything in the record comes from the objects handed to it. The date of the search is the retrieved_at attribute scopus_fetch() attaches, never the current time; the number of records the API reported as matching is what the cells recorded (the cell_totals attribute scopus_fetch_plan() attaches, or total_results for a set retrieved without a plan), never an inference from the number of rows, and it is given overall only when every cell reported one; and the duplicates removed are those scopus_combine() recorded removing. Where an attribute is absent, as it is for a set read back from a .csv, for the bundled corpus, and for a cell resumed from a checkpoint written before these attributes existed, the record says the field is unrecorded and fills nothing in. This matters most for completeness: a harvest whose reported total is unknown is never described as exhaustive.

The PRISMA-S map is decided the same way. Items the package holds evidence for (the database and platform, the full strategy, the limits, the date, the totals, and de-duplication where it was performed) are listed as supplied. The rest, among them peer review of the strategy, grey literature, other databases and citation searching, are listed as the author's to supply, because the package has no way to know them.

Value

A list of class scopus_search_report, returned invisibly when file is written. Its elements are the fields the record is built from: database, platform, query (the base query), field, expression

(the field-wrapped query of each cell), `view`, `page_size`, `paging`, `partition`, `n_cells`, `years`, `cells` (a tibble of `cell`, `limit`, `n_records` and `reported_total`), `searched_at`, `version`, `n_records`, `n_with_doi`, `reported_total`, `records_combined`, `duplicates_removed`, `deduplicated`, `snippet` and `prisma` (a tibble of `item`, `name`, `source` and `note`). A field the objects do not record is `NA`.

`format()` returns a length-one character string.

References

Rethlefsen, M. L., Kirtley, S., Waffenschmidt, S., Ayala, A. P., Moher, D., Page, M. J., & Koffel, J. B. (2021). PRISMA-S: an extension to the PRISMA Statement for Reporting Literature Searches in Systematic Reviews. *Systematic Reviews*, *10*, 39. doi:10.1186/s1364302001542z

See Also

`scopus_plan()`, `scopus_fetch_plan()`, `scopus_combine()`

Examples

```
# A search described but not yet run. The record says so throughout rather
# than implying figures it cannot have.
plan <- scopus_plan("graphene supercapacitor", years = 2015:2024,
                    field = "TITLE-ABS-KEY", partition = "year")
scopus_search_report(plan)

# The same search after a harvest. The bundled corpus of real articles
# stands in for one, since 'Scopus' records may not be redistributed, so the
# attributes a live retrieval records are set here by hand.
recs <- example_records
attr(recs, "plan") <- plan
attr(recs, "retrieved_at") <- as.POSIXct("2026-07-22 09:15:00", tz = "UTC")
attr(recs, "scopusflow_version") <- "0.3.0"
report <- scopus_search_report(recs)
report

# The methods paragraph, and the Markdown record for a supplementary file.
cat(format(report, style = "paragraph"))
path <- tempfile(fileext = ".md")
scopus_search_report(recs, file = path)
```

---

scopus_top *Most frequent values in a record set*

Description

Tallies the most common sources or authors across a scopus_records object. It works on records already in memory, so it makes no network request.

Usage

```
scopus_top(x, by = c("source", "author"), n = 10L)
```

Arguments

x	A scopus_records tibble.
by	What to tally: "source" (the publication titles) or "author". Author strings holding several names separated by "; " are split, so each contributor is counted once per record.
n	The number of rows to return (the top n).

Value

A tibble of class scopus_top with columns value and n, sorted by descending count, with ties broken by value in byte order so the result is reproducible across platforms and locales. Exactly n rows are returned (fewer if there are fewer distinct values), so values tied at the cut-off rank may be dropped. The by choice is stored in the by attribute.

See Also

plot_scopus_top(), summary.scopus_records()

Examples

```
# The bundled corpus of real articles stands in for a harvest of your own,
# since 'Scopus' records may not be redistributed.
scopus_top(example_records, by = "source")

# That corpus names one author per article, so the author tally counts
# first authors; a live harvest lists every author and splits them.
scopus_top(example_records, by = "author", n = 5)
```

---

scopus_trend *Annual publication counts for a query*

Description

Counts how many records match a query in each year, giving the size of a literature over time. It is the single-query companion to scopus_compare_topics(): where the comparison shows topics as a share of a reference, this shows the absolute count.

Usage

```
scopus_trend(
  query,
  years,
  field = NULL,
  view = c("STANDARD", "COMPLETE"),
  api_key = NULL,
  inst_token = NULL,
  verbose = FALSE
)
```

Arguments

query	Character scalar. The base search expression.
years	Integer vector of publication years to count over, for example 2010:2020.
field	Optional 'Scopus' field tag to wrap the query in (see scopus_plan()).
view	Either "STANDARD" or "COMPLETE". COMPLETE adds an authkeywords column to scopus_fetch()/scopus_fetch_plan() output (see scopus_records()) at no extra cost beyond COMPLETE's own smaller page size, which already means more requests, and so more quota, for the same number of records.
api_key, inst_token	Optional credentials, resolved by default from options or environment variables (see scopus_has_key()).
verbose	Logical. When TRUE, progress is reported.

Value

A tibble of class scopus_trend with columns query (the field-wrapped query), year (integer) and n (the count that year, as a double so very large counts are exact). A year whose response omits a total is recorded as NA (with a warning) and contributes nothing to the total shown by print().

API access

This performs one count request per year, so it requires a valid API key and internet access; see the *API access* section of scopus_count().

See Also

plot_scopus_trend(), scopus_compare_topics()

Examples

```
tr <- scopus_trend("graphene supercapacitor", years = 2015:2024,
                   field = "TITLE-ABS-KEY")
tr

# The offline companion, which needs no key. 'Scopus' records may not be
# redistributed, so the package bundles a corpus of real articles instead;
# it is a complete harvest of its own query, so tallying its rows by year
# reproduces the yearly counts that query returns.
by_year <- table(example_records$year)
tr <- tibble::tibble(
  query = "TITLE-ABS-KEY(graphene supercapacitor)",
  year = as.integer(names(by_year)),
  n = as.numeric(by_year)
)
class(tr) <- c("scopus_trend", class(tr))
tr
```

summary.scopus_records *Summarise a set of 'Scopus' records*

Description

Gives a compact overview of a scopus_records object, reporting how many records it holds, the span of publication years they cover, how many distinct sources and Digital Object Identifiers appear among them and how widely they have been cited. It is a convenient way to take stock of a retrieval before any closer analysis.

Usage

```
## S3 method for class 'scopus_records'
summary(object, ...)
```

Arguments

object	A scopus_records tibble.
...	Ignored, present for compatibility with the summary() generic.

Value

A list of class scopus_records_summary, with elements n_records, years (the earliest and latest year present, each NA when no year is known), n_sources, n_with_doi, total_citations, median_citations, top_cited (the title of the most-cited record) and top_source (the most frequent source title). Printing it produces a short readable report.

Examples

```
# The bundled corpus of real articles stands in for a retrieval of your
# own, since 'Scopus' records may not be redistributed.
summary(example_records)
```

write_scopus_records *Read and write 'Scopus' record sets*

Description

Save a scopus_records tibble to disk and read it back, with a stable round-trip. The file extension selects the format. An .rds file preserves the types and class exactly, while a .csv file is portable plain text. The optional authkeywords column a view = "COMPLETE" retrieval adds (see scopus_records()) round-trips through both formats. The attributes a live retrieval carries, including retrieved_at and scopusflow_version (see scopus_fetch()), survive the .rds form only: .csv is a table of columns and cannot hold them, so save as .rds when a set is a baseline to be compared against later.

Usage

```
write_scopus_records(x, path)

read_scopus_records(path)
```

Arguments

x	A scopus_records tibble to write.
path	Explicit file path. The functions read from, or write to, exactly this path and leave the working directory alone. Parent directories are assumed to exist already.

Value

`write_scopus_records()` returns `x` invisibly. `read_scopus_records()` returns a scopus_records tibble.

Examples

```
# A round trip on the bundled corpus of real articles, which stands in for
# a retrieval of your own because 'Scopus' records may not be redistributed.
# The .rds form restores the object exactly.
rds <- tempfile(fileext = ".rds")
write_scopus_records(example_records, rds)
identical(read_scopus_records(rds), example_records)

# The .csv form is portable plain text and reads back to the same schema.
csv <- tempfile(fileext = ".csv")
write_scopus_records(example_records, csv)
head(read_scopus_records(csv))
```

Index