

Package: scopusflow (via r-universe)

October 16, 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](https://doi.org/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 (>= 1.0.4), jsonlite, rlang (>= 1.1.0), stats,
tibble, tools, utils

Suggests bslib, callr, fansi, ggplot2, grid, knitr, rmarkdown, shiny
(>= 1.6.0), spelling, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Language en-GB
Roxygen list(markdown = TRUE)
Config/roxygen2/version 8.0.0
Config/pak/sysreqs libssl-dev
Repository <https://pabloernabeu.r-universe.dev>
Date/Publication 2026-10-16 21:57:55 UTC
RemoteUrl <https://github.com/pabloernabeu/scopusflow>
RemoteRef HEAD
RemoteSha db01af82ec7b811458449b615b14af24beeb8dce

Contents

as_bibliometrix	3
as_bibtex	4
as_theoryforge_corpus	5
as_tibble.scopus_records	7
example_records	11
plot_scopus_comparison	13
plot_scopus_intersections	15
plot_scopus_top	16
plot_scopus_trend	17
run_app	18
scopus_abstract	19
scopus_cache_clear	22
scopus_cache_dir	23
scopus_combine	24
scopus_compare_topics	25
scopus_corpus	27
scopus_count	30
scopus_diff_dois	32
scopus_diff_records	33
scopus_extract_dois	34
scopus_fetch	36
scopus_fetch_plan	38
scopus_field_tags	40
scopus_has_key	41
scopus_intersections	42
scopus_plan	44
scopus_query	46
scopus_quota	47
scopus_search_report	48
scopus_top	50
scopus_trend	51
summary.scopus_records	53
write_scopus_records	54

as_bibliometrix	<i>Convert records to a bibliometrix-compatible data frame</i>
-----------------	--

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. The fields follow the conventions `bibliometrix::convert2df()` applies to a 'Scopus' CSV export, so the two frames describe a record in the same terms.

Usage

```
as_bibliometrix(x)
```

Arguments

x A [scopus_records](#) tibble.

Details

Each author in AU is written as bibliometrix writes a 'Scopus' author, the surname followed by the initials without full stops. The authors are joined by "; ". A name in the "Surname, Given" form the COMPLETE view gives becomes the surname followed by the initial of each given name, and a hyphenated given name keeps its hyphen. "Tour, James M." thus becomes "TOUR JM" and "Lee, Chul-Hee" becomes "LEE C-H". A name without a comma, such as the STANDARD view's "Lee C. -H. ", loses its full stops and becomes "LEE C-H". Each given name yields one initial, so a name that 'Scopus' abbreviates to two letters, such as "Yu." for Yuri, keeps both under STANDARD and gives one under COMPLETE.

UT holds the EID, "2-S2.0-" followed by the 'Scopus' identifier, as a 'Scopus' import gives it, and is NA for a record without an identifier. `sub("^2-S2.0-", "", UT)` gives back the bare identifier. DE holds the author keywords joined by "; ", as in a 'Scopus' CSV export. Keywords are split at the " | " the Search API joins them with and at the "; " [scopus_abstract\(\)](#) joins its own with, so either source can fill the authkeywords column.

This produces the *shape* bibliometrix expects from the 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()`.

AU holds every author only for a retrieval made with `view = "COMPLETE"`. Under STANDARD the API sends the first author alone, so each document would look single-authored to the collaboration measures and co-authorship networks bibliometrix builds from AU. Converting a set retrieved under STANDARD therefore raises a warning of class `scopus_warning_first_author_only`, once per session (see [scopus_records\(\)](#)).

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 (the record's EID) and DB ("SCOPUS"). DE (author keywords) follows when the set carries the `authkeywords` column of a `view = "COMPLETE"` retrieval. DT (document type), VL (volume) and IS (issue) follow when it carries the columns of a `fields = "extended"` retrieval (see [scopus_records\(\)](#)). Character tag fields other than DI are upper-cased, as `bibliometrix` upper-cases them.

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. 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 <code>NULL</code> the formatted string is returned; with a path it is written there and returned invisibly. Nothing is written unless a path is given.

Details

A set retrieved with `fields = "extended"` (see [scopus_records\(\)](#)) carries each record's 'Scopus' document type, and the entry is typed by it. A conference paper (cp) becomes an RIS CPAPER and a BibTeX @inproceedings, and a conference review (cr) an RIS CONF and a BibTeX @inproceedings. A book chapter (ch) becomes an RIS CHAP and a BibTeX @incollection, and a book (bk) an RIS BOOK and a BibTeX @book. Every other record, and every record of a set retrieved with the core fields, is a journal article (JOUR, @article). In BibTeX, the source title of a conference paper or chapter goes under `booktitle`, braced as the title is, and a journal article's

under journal. A book in a book series gives the series under series, and a book outside a series, whose source title 'Scopus' gives as the book's own, names no source. RIS gives the source under T2 except for such a book, since Zotero reads a book's T2 as its series. The entries also carry the volume, issue, pages, ISSN and document type (RIS M3). Under view = "COMPLETE" they carry the abstract and the author keywords, one KW line each in RIS and joined by commas in BibTeX. Keywords are split at the " | " the Search API joins them with and at the "; " `scopus_abstract()` joins its own with. A value the record lacks is left out, never filled in.

Names are written family name first, with a comma. Under the STANDARD view, the authors column holds 'Scopus' names in the form Zhang F., which BibTeX would read as given name "Zhang" and family name "F.", so a name ending in initials is exported as Zhang, F.. BibTeX entries also space run-together initials (Kitchin, J. R.), which BibTeX otherwise takes for a single given name. Names that already carry a comma, as under COMPLETE, and names given first are written as they stand. The authors column itself and the citation keys are unchanged. In a BibTeX title, every word with a capital after its first letter is braced ({DNA}, {CRISPR-Cas9}), so bibliography styles cannot lower-case it.

An RIS entry gives the source title under T2, the tag 'Scopus' itself uses and reference managers read as the publication title, records the database as DB - Scopus and, when the record has an identifier, gives the EID as the accession number (AN - 2-s2.0-<scopus_id>).

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_theoryforge_corpus *Write a corpus in theoryforge's corpus schema*

Description

Turns a `scopus_corpus()` result into the literature corpus that theoryforge's `tf_read_corpus()`, `tf_litmap()` and `tf_landscape()` read, following the corpus schema theoryforge publishes (`schema/corpus.schema.json` in its repository). `as_theoryforge_corpus()` returns the document as a list, and `write_scopus_corpus()` writes it to a JSON file. Neither needs theoryforge installed, and neither makes a request.

Usage

```
as_theoryforge_corpus(corpus, id)
```

```
write_scopus_corpus(corpus, path, id)
```

Arguments

corpus	A scopus_corpus() result, or a data frame with the same id, title, year, keywords and references columns, where each references entry is a data frame with id and doi columns. A scopus_id column is used when present.
id	A non-empty string naming the corpus, written as its top-level id. It is required, since a scopus_corpus() result carries no name of its own.
path	The file to write, exactly as given. Parent directories are assumed to exist already.

Details

theoryforge compares references as exact strings, so the writer gives each cited work one key across the whole corpus, its 'Scopus' identifier, as `scopus:<id>`. 'Scopus' gives a cited work's DOI in some bibliographies only, and in different cases. A key built from the DOI wherever there is one would therefore split one work into several nodes and lose the co-citation edges between them. A reference that carries only a DOI takes the 'Scopus' identifier that a document or reference elsewhere in the corpus pairs with the same DOI, the documents' own identifiers counting first. A DOI that nothing pairs with is keyed as `doi:<doi>`, without its resolver prefix and with its ASCII letters lower-cased, since DOIs are case-insensitive over ASCII (DOI Handbook, section 2.4). A reference with neither is dropped, and a `scopus_warning_dropped_references` warning says how many. Each document is keyed the same way, from its `scopus_id` or else from an `id` that is a DOI or a 'Scopus' identifier, so a citation of a document in the corpus and the document itself share one key.

Keywords are trimmed, lower-cased with [tolower\(\)](#) and made unique, keeping the first of each, and empty ones are dropped, so "Graphene" and "graphene " count as one keyword. The Python twin lower-cases with `str.lower()`. The two agree on the Latin, Greek and Cyrillic letters in common use and differ on a few special cases, such as the Turkish capital I with a dot and a capital sigma ending a word, which Python writes as a final sigma. A missing year is written as `null`, and a missing title is left out, since the schema admits the one and not the other.

The document also carries a provenance object, which the schema admits beside its own fields: `source` ("Scopus via `scopusflow` <version>"), `query`, `retrieved_at` (in UTC, as YYYY-MM-DDTHH:MM:SSZ), `keywords`, `n_records`, `n_references_dropped` and `n_records_without_keywords`. `query`, `retrieved_at` and `keywords` come from the attributes [scopus_corpus\(\)](#) attaches, and are `null` when the corpus lacks them. `query` is a string, or an array of strings when the records were searched with several expressions. Author keywords can come back empty under some entitlements (see [scopus_abstract\(\)](#)), which leaves theoryforge's keyword themes empty whatever the writer does, and `n_records_without_keywords` shows when that happened.

The file is UTF-8, with no escaping beyond what JSON requires, sorted keys, no spaces between tokens and one line feed at the end, so the same corpus gives the same bytes from either twin.

Value

as_theoryforge_corpus() returns the document as a list with the elements id, provenance, records and schema_version ("1.0"). write_scopus_corpus() returns the same list invisibly.

See Also

[scopus_corpus\(\)](#), which builds the corpus.

Examples

```
# Two records of the bundled corpus of real articles, each citing the same
# two others. The keywords and the citations are illustrative, since the
# bundled records carry neither.
docs <- example_records[!is.na(example_records$doi), ][1:4, ]
cited <- tibble::tibble(id = NA_character_, doi = docs$doi[3:4])
corpus <- tibble::tibble(
  id = docs$doi[1:2],
  scopus_id = NA_character_,
  title = docs$title[1:2],
  year = docs$year[1:2],
  keywords = list(c("Graphene", "supercapacitor"), "graphene"),
  references = list(cited, cited)
)
doc <- as_theoryforge_corpus(corpus, id = "graphene-supercapacitors")
doc$records[[1]]$keywords
doc$records[[1]]$references

# Written to a temporary file (never the working directory).
path <- tempfile(fileext = ".json")
write_scopus_corpus(corpus, path, id = "graphene-supercapacitors")
substr(readlines(path, encoding = "UTF-8"), 1, 160)
```

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,
  fields = c("core", "extended")
)

is_scopus_records(x)

```

Arguments

x	An object to test.
...	Ignored, for S3 compatibility.
object	A scopus_records object (for the <code>autoplot()</code> 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 read authors from the author list that view carries and to add an authkeywords column (see below). Any other value, including the default NULL, reads the first author and 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.
fields	"core" (the default) builds the columns listed under <i>Value</i> , exactly as before. "extended" also keeps the document type, source type, volume, issue, pages, ISSN, PubMed identifier and open-access status the Search API sends with every record, and under view = "COMPLETE" the abstract, at no extra quota (see <i>Extended fields</i>). scopus_fetch() and scopus_fetch_plan() pass this through.

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.

What authors holds depends on the view. The Search API's `dc:creator` field is the first author alone under either view, in the indexed form 'Scopus' uses ("Zhang F."). The author list arrives only under view = "COMPLETE", as the author objects Elsevier's views table lists for that view alone. A STANDARD record therefore names its first author, and a tally of STANDARD authors counts first authorships. A harvest records its view in the view attribute (see [scopus_fetch\(\)](#)), and [scopus_top\(\)](#) and [as_bibliometrix\(\)](#) say once per session that a set retrieved under STANDARD names first authors only. Under COMPLETE, each author is written as "Surname, Given", as the surname alone when the given name is missing or empty, or as the indexed name when the surname is missing. Exact duplicates are dropped. The API lists at most 100 authors per record, and its

author-count field gives the total. A record whose list names nobody falls back to dc:creator. The Python twin builds the same string from pybliometrics' author_names in the common case, and parts company at two edges. Where an author object lacks a surname or given-name key, pybliometrics builds no list, so that twin keeps the first author while this one lists every author. Where a surname is null or empty, pybliometrics writes the given name after a bare comma, and this package uses the indexed name.

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. The bars cover every year the set's plan searched, a year without records drawn as a zero bar, and no year the plan did not search. A plan over 2015, 2016, 2019 and 2020 therefore draws no bar for 2017 or 2018. A harvest stopped before its last cell covers only the cells it ran. Without a plan, or with one that sent no year limit, the bars run from the first year in the records to the last. When a max_results cap stopped any cell of the retrieval, a caption says that the bars count records retrieved, not publications per year. When the years plotted reach the year of the recorded retrieved_at, the caption also says that year was incomplete when the records were retrieved, as [plot_scopus_trend\(\)](#) does.

A tibble of class scopus_records with the columns entry_number (integer), scopus_id (character), doi (character), title (character), authors (character, the author list under view = "COMPLETE" and the first author otherwise, as *Details* explains), 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*). With fields = "extended", the character columns doc_type, doc_type_description, source_type, volume, issue, pages, issn and pubmed_id follow, then the integer open_access and, under view = "COMPLETE", the character abstract. doc_type and doc_type_description hold the document-type code and name 'Scopus' gives the record, such as "ar" and "Article", and source_type holds the source type, such as "Journal". issn is the print ISSN as 'Scopus' writes it, without the hyphen, kept as text so its leading zeros survive. open_access is the open-access status code (see *Extended fields*).

is_scopus_records() returns a length-one logical.

Extended fields

Elsevier's [Search views table](#) lists every field fields = "extended" reads under both views, except the abstract (dc:description), which only the COMPLETE view carries. They arrive in the responses

a retrieval already pays for, so they cost no extra quota. A COMPLETE harvest returns up to 25 abstracts with each request, while `scopus_abstract()` spends one request per record from the separate Abstract Retrieval quota.

`doc_type` holds the code the DOCTYPE() search field takes. Elsevier's [search tips](#) list the codes with their names: ar for an article, re a review, cp a conference paper, er an erratum and others. A retracted item carries tb, which that list leaves out. The 'Scopus' Content Coverage Guide (section 3.1) defines an erratum as a report of an error, correction or retraction of a previously published paper, so er covers retraction notices as well as corrections. The guide gives an article the type Retracted once a retraction notice for it is published. These codes are the labels 'Scopus' assigns, and they do not screen a set for retractions, since 'Scopus' does not label every retracted item (Ortega and Delgado-Quirós, 2024). Check a set against the [Retraction Watch database](#), which Crossref makes openly available, before relying on it. `summary()` reports how many records 'Scopus' labels er and tb. Errata can also be left out of the search itself with AND NOT DOCTYPE(er), joined by `scopus_query()`, which brackets the rest of the query first, since 'Scopus' applies AND NOT last (see *Examples*).

`open_access` holds the code 'Scopus' sends in its openaccess field. Elsevier's Scopus APIs Getting Started Guide (section 17) illustrates its values with 0 for a record that is not open access, 1 for gold and 2 for green. The column therefore holds a status code, which can take values other than 0 and 1.

References

- Elsevier. (2023). *Scopus APIs getting started guide* (Version 1). https://dev.elsevier.com/guides/Scopus%20API%20Guide_V1_20230907.pdf
- Elsevier. (2023). *Scopus content coverage guide*. https://supportcontent.elsevier.com/Support%20Hub/Scopus/Scopus_ContentCoverage_Guide_WEB.pdf
- Ortega, J. L., & Delgado-Quirós, L. (2024). The indexation of retracted literature in seven principal scholarly databases: A coverage comparison of Dimensions, OpenAlex, PubMed, Scilit, Scopus, The Lens and Web of Science. *Scientometrics*, 129(7), 3769–3785. doi:10.1007/s1119202405034y

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")

# fields = "extended" keeps the fields both views send beyond the core ones.
# The volume, issue, pages and print ISSN below are the article's own. The
# document type and source type are illustrative, since the bundled corpus
# came from OpenAlex and records no 'Scopus' labels.
raw_extended <- list(entry = list(
  list(
    `prism:doi` = "10.1021/am509065d",
    `dc:title` = "Flexible and Stackable Laser-Induced Graphene Supercapacitors",
    subtype = "ar",
    subtypeDescription = "Article",
    `prism:aggregationType` = "Journal",
    `prism:volume` = "7",
    `prism:issueIdentifier` = "5",
    `prism:pageRange` = "3414-3419",
    `prism:issn` = "19448244"
  )
))
extended <- scopus_records(raw_extended, fields = "extended")
as.data.frame(extended)[, c("doc_type", "volume", "issue", "pages", "issn")]

# Errata can be left out of the search itself. scopus_query() brackets the
# rest of the query, since 'Scopus' applies AND NOT last.
scopus_query(
  scopus_query("graphene", "graphite", .op = "OR", .field = "TITLE-ABS-KEY"),
  "DOCTYPE(er)", .op = "AND NOT"
)

# 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 phrase of the OpenAlex search that produced the record.

Details

The records are deliberately not a 'Scopus' harvest. The Elsevier API terms do not permit re-distributing 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. They are the result of OpenAlex's exact-phrase search for "graphene supercapacitor" in titles and abstracts, limited to journal articles, run on 22 July 2026. The worked examples send the phrase to 'Scopus' as `TITLE-ABS-KEY("graphene supercapacitor")`, which also searches keywords and every document type. That search yields the same kind of object, with the same columns and the same handling, but a different set of records. Without the quotation marks the set is far larger, since 'Scopus' joins the words of an unquoted phrase with AND. In OpenAlex, the unquoted words match about a hundred times as many 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 search, 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

<code>x</code>	A <code>scopus_comparison</code> object from <code>scopus_compare_topics()</code> .
<code>pub_count_in_legend</code>	Logical. When TRUE (the default), each topic's label carries its total record count, for example effect size ($n = 1,204$).
<code>highlight</code>	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.
<code>interval</code>	Logical. When TRUE (the default), a shaded band around each line shows a Wilson interval on the yearly share. See <i>Details</i> 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

<code>x</code>	A <code>scopus_intersections</code> object from <code>scopus_intersections()</code> .
<code>highlight</code>	Optional character vector of row labels to draw in an accent colour, for example the intersection that defines a study's niche.
<code>highlight_label</code>	Legend label for the highlighted rows. The default, <code>NULL</code> , 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.
<code>...</code>	Currently unused, present for S3 consistency.
<code>object</code>	A <code>scopus_intersections</code> object (for the <code>autoplot()</code> 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

<code>x</code>	A <code>scopus_top</code> object from <code>scopus_top()</code> .
<code>...</code>	Currently unused, present for S3 consistency.
<code>object</code>	A <code>scopus_top</code> object (for the <code>autoplot()</code> 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	<i>Plot a publication trend</i>
-------------------	---------------------------------

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).

Details

When the years plotted reach the year recorded in the retrieved_at attribute that [scopus_trend\(\)](#) attaches, the caption adds a line such as "2026 was incomplete when retrieved on 2026-09-29." 'Scopus' goes on indexing a year's publications after the year has begun, so the last point of such a figure is a count so far, easily read as a decline. The line is read from the recorded time alone, never from the clock of the session drawing the figure, so a trend that records none, such as one tallied by hand, is drawn without it.

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 of OpenAlex's exact-phrase search for the
# phrase below, so its rows per year are the publications per year of that
# search.
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 the author keywords, the index keywords and the document's reference list, or any of them, to the same request.

Usage

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

Arguments

<code>ids</code>	Character vector of identifiers to look up, either Digital Object Identifiers or 'Scopus' record identifiers (with or without the "SCOPUS_ID:" prefix), according to <code>by</code> .
<code>by</code>	Either "doi" or "scopus_id", the kind of identifier in <code>ids</code> .
<code>view</code>	The Abstract Retrieval view to request: "META_ABS" (the default), "META", "REF" or "FULL", or NULL. "META_ABS" is the view Elsevier documents as carrying the abstract with the core metadata. NULL sends no view parameter, as scopusflow 0.4.0 and earlier did, and leaves the choice to the API. Elsevier documents its default as "META", a view without the abstract, but a live request in October 2026 returned the abstract without a view. Retrieving <code>include = "references"</code> requires <code>view = "FULL"</code> or <code>view = "REF"</code> ; see <i>Details</i> for how the two differ and which to prefer.
<code>include</code>	Optional character vector naming extra fields to retrieve in the same request: any of "references", "keywords" (the author keywords) and "index_terms" (the index keywords 'Scopus' assigns). "references" requires <code>view = "FULL"</code> or <code>view = "REF"</code> , and "keywords" and "index_terms" require <code>view = "FULL"</code> , since the REF response carries neither kind of keyword. 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 <i>Details</i> .
cache_dir	Optional directory for per-identifier cache files, as in <code>scopus_fetch_plan()</code> . 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 at least one request of its own, so re-running an interrupted batch without a cache re-spends quota already spent. A cache_dir that cannot be written is refused before any request, and a checkpoint that cannot be written all the same warns and keeps the row, as in <code>scopus_fetch_plan()</code> .
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 <code>scopus_has_key()</code>).
verbose	Logical. When TRUE, progress is reported.

Details

Retrieving references needs Abstract Retrieval's FULL or REF view, and keywords need FULL. FULL returns the whole bibliography in one request. REF serves it a page at a time, about 40 references per request, so `scopus_abstract()` follows the view's `startref` parameter until it holds as many references as the document reports, as pybliometrics does. A REF document with N references therefore costs $\text{ceiling}(N / 40)$ requests, each counted in `n_requests`, which makes "FULL" the cheaper view when your entitlement allows it. Elsevier's guide numbers `startref` from 0 and pybliometrics from 1, and which of the two the API follows has not been confirmed. Paging stops on an empty page, on a page that adds no new reference, or after as many pages as the first page's length implies, so neither numbering can make it loop or repeat references. When the number of references held still differs from the document's reported count, a `scopus_warning_incomplete_references` warning says so. An identifier whose later page fails is a failed retrieval: it gets a row of NAs with a warning and is not checkpointed.

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.

Index keywords are the other kind 'Scopus' records. Authors choose their own keywords, whereas 'Scopus' chooses the index keywords and standardises them to vocabularies derived from thesauri Elsevier owns or licenses, with synonyms, spellings and plurals brought together. The two kinds therefore give different keyword co-occurrence results, and a study should say which it used. Elsevier's guide lists index keywords in the FULL view as not available by default, so whether they arrive depends on your key's entitlement as well. In this package's development testing, they did arrive on the same key whose author keywords came back empty.

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 "index_terms", an `idxterms` column is added after any `authkeywords`: the index keywords of the document, joined by " ; ", or `NA` when the response carries none. An entry without text is dropped.

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, or one per page of references under `view = "REF"` with `include = "references"`, 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.

A spent weekly quota (HTTP 429 marked `QUOTA_EXCEEDED`) also stops the requests, but returns what the batch holds, since it usually arrives part-way through: the rows already retrieved are kept, each identifier not retrieved gets a row of `NA`s that is not checkpointed, and one `scopus_warning_quota_exceeded` warning names the identifier reached, how many were not retrieved and when the quota resets. A checkpoint in `cache_dir` is still read after the stop. 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 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]]

```

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

```
# The managed cache is pointed at a temporary directory first, so that
# running this example cannot delete the checkpoints of a real harvest.
# Nothing is cached there, and clearing an absent cache is safe.
old <- Sys.getenv("R_USER_CACHE_DIR", unset = NA)
Sys.setenv(R_USER_CACHE_DIR = tempfile())
scopus_cache_clear()
if (is.na(old)) Sys.unsetenv("R_USER_CACHE_DIR") else
  Sys.setenv(R_USER_CACHE_DIR = old)
```

scopus_cache_dir	<i>Managed cache directory for scopusflow</i>
------------------	---

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

<code>...</code>	Two or more scopus_records objects, or a single list of them.
<code>dedupe</code>	Whether, and by which rule, to keep once the records that are one record. <code>FALSE</code> , the default, keeps every row. <code>TRUE</code> , also spelled " <code>id_or_shared_doi</code> ", treats records as one when they share a 'Scopus' identifier, or when they share a DOI and one of them lacks an identifier. " <code>id_or_doi</code> " also merges records with different identifiers that share a DOI, and " <code>id_then_doi</code> " applies the rule of <code>scopusflow</code> 0.4.0 and earlier. See Details.
<code>x</code>	A scopus_records object (for the <code>c()</code> method).

Details

De-duplication compares DOIs as [scopus_extract_dois\(\)](#) does, without resolver prefixes such as `https://doi.org/` or `doi:` and ignoring case. It keeps the first record of each group, in the order the sets were given, and a record with neither an identifier nor a DOI cannot be matched and is kept.

Under the default rule, records with the same 'Scopus' identifier are one record, and a record without an identifier joins the identified record that carries its DOI. Records with different identifiers are never merged, even when they share a DOI. 'Scopus' holds duplicate records of one paper (Valderrama-Zurián et al., 2015), but it also gives one DOI to distinct papers (Franceschini et al., 2015), and a review loses a wrongly removed record for good where a surviving duplicate is caught at screening. Such DOIs are counted in `doi_conflicts`, which the search record reports, so that they can be checked by hand. A record without an identifier whose DOI two identified records carry is kept once, apart from both. Records without an identifier that share a DOI no identified record carries are one record.

"`id_or_doi`" merges every record linked to another by a shared identifier or a shared DOI, directly or through a chain of such links. It suits a set whose shared DOIs have been checked and found to be duplicates. "`id_then_doi`" keys each record on its identifier, failing that on its DOI lower-cased but not otherwise cleaned, and so reproduces a set de-duplicated by `scopusflow` 0.4.0 or earlier.

Value

A `scopus_records` tibble. Per-retrieval attributes such as `total_results` and `plan` are not carried over, since they describe a single fetch. The provenance a merge leaves true is kept, as `scopus_fetch_plan()` keeps it for plan cells. `retrieved_at` is the earliest of the inputs' times and `retrieved_until` the latest, so a baseline merged with its update is dated from one to the other. Every contributing `scopusflow_version` is listed. The merge itself is recorded in the combined attribute, a list of `n_in` (records supplied), `n_out` (records kept), `n_removed`, `deduplicated`, `rule` (the rule applied, NA when none was) and `doi_conflicts` (how many DOIs records with different identifiers share, NA when no rule was applied). It is what lets `scopus_search_report()` state how many duplicates were removed and how, so the PRISMA-S item is answered.

References

Franceschini, F., Maisano, D., & Mastrogiacomio, L. (2015). Errors in DOI indexing by bibliometric databases. *Scientometrics*, 102(3), 2181–2186. doi:10.1007/s1119201415034

Valderrama-Zurián, J.-C., Aguilar-Moya, R., Melero-Fuentes, D., & Aleixandre-Benavent, R. (2015). A systematic analysis of duplicate records in Scopus. *Journal of Informetrics*, 9(3), 570–576. doi:10.1016/j.joi.2015.05.002

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 joined to the reference as (reference) AND (term), after field has wrapped each side. A reference containing OR or AND NOT therefore keeps its meaning, and every comparison set is a subset of the reference.
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` ($100 * n / \text{reference_n}$, 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 year for the reference topic and one per term per year, $(\text{length}(\text{comparison_terms}) + 1) * \text{length}(\text{years})$ requests in all. It therefore 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 keywords (the author keywords by default, or the index keywords 'Scopus' assigns) 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,
  keywords = c("author", "index", "both")
)

```

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" returns a document's whole bibliography in one request. "REF" serves it about 40 references per request, which scopus_abstract() follows page by page, so a document with N references costs $\text{ceiling}(N / 40)$ requests (see its documentation for the details and for the entitlement each view needs). The REF response also carries no keywords of either kind, 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. |
| keywords | Which keywords fill the keywords column: "author" (the default) for the author keywords, "index" for the index keywords 'Scopus' assigns, or "both" for the author keywords followed by each index keyword not already among them. The match is on the exact text, so a term that differs only in case appears twice. "index" and "both" need view = "FULL", since the REF response carries no keywords of either kind, and raise a <code>scopus_error_bad_input</code> error under "REF" before any request. Index keywords and author keywords give different co-occurrence results, and the <i>Details</i> of scopus_abstract() describe how each is chosen and the entitlement each depends on. Either choice costs the same request per record. |

Value

A tibble with columns `id` (the identifier records was looked up by), `scopus_id` (the document's own 'Scopus' identifier, as Abstract Retrieval gives it, or the `scopus_id` of records where the lookup failed or the REF response omitted it, and NA when neither has one), `title`, `year`, `keywords` (a list-column: a character vector of the document's keywords of the kind `keywords` names, split out of `scopus_abstract()`'s joined `authkeywords` and `idxterms` strings, 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.

The number of Abstract Retrieval requests made and the most recently parsed quota are attached as the `n_requests` and `quota` attributes, as `scopus_abstract()` attaches them. Three more attributes carry what `write_scopus_corpus()` records of the corpus's provenance. `query` holds the field-wrapped search expression of each cell of the records' `plan`, or, for records without a `plan`, the distinct values of their own `query` column. `retrieved_at` is when the search that found the records ran, and `keywords` is the kind of keyword chosen. `query` and `retrieved_at` are absent when records carries neither.

API access

This performs one Abstract Retrieval request per usable record, or one per page of references under `view = "REF"`, 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()`, and `write_scopus_corpus()` to write the corpus for theoryforge.

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
# no keywords, bibliographies or 'Scopus' identifiers, so the keywords and
# references are illustrative and the identifiers are left NA.
docs <- example_records[order(-example_records$citations), ][1:2, ]
corpus <- tibble::tibble(
  id = docs$doi,
  scopus_id = NA_character_,
  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	<i>Count 'Scopus' results for a query</i>
--------------	---

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, checked offline before any request (see the <i>Query checks</i> section of scopus_plan()).
years	Optional integer vector of publication years. scopus_count() , scopus_fetch() and scopus_intersections() send one inclusive range from the earliest year to the latest, as a scopus_plan() with <code>partition = "none"</code> does, so <code>c(2015,</code>

	2020) covers every year from 2015 to 2020. <code>scopus_trend()</code> , <code>scopus_compare_topics()</code> and a plan with <code>partition = "year"</code> take each listed year. An empty vector places no year restriction, as <code>NULL</code> does, where the years form a range, and is refused where each year is counted.
<code>field</code>	Optional 'Scopus' field tag to wrap the query in (see <code>scopus_plan()</code>).
<code>view</code>	Either "STANDARD" or "COMPLETE". COMPLETE adds an <code>authkeywords</code> column to <code>scopus_fetch()/scopus_fetch_plan()</code> output (see <code>scopus_records()</code>) 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.
<code>api_key</code> , <code>inst_token</code>	Optional credentials, resolved by default from options or environment variables (see <code>scopus_has_key()</code>).

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. Throttling (HTTP 429) and server errors are retried with back-off first, and so are transport failures such as a timeout or a reset connection. A spent weekly quota, which Elsevier also answers with HTTP 429, is not retried: it raises `scopus_error_quota_exceeded`, a subclass of `scopus_error_rate_limit` whose message gives the reset time. When the last attempt at a request times out after `getOption("scopusflow.timeout", 60)` seconds, the error is `scopus_error_timeout`, a subclass of `scopus_error_offline`.

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 OpenAlex's exact-phrase search for the same phrase, so its row count is
# the total of that search. A 'Scopus' count of the phrase would differ. 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	<i>Compare two DOI retrievals</i>
------------------	-----------------------------------

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.

Details

A record without a DOI cannot be compared this way, and 'Scopus' gives no DOI to a share of what it indexes, more of it in the arts and humanities and among books and proceedings (Gorraiz et al., 2016). Such records are left out of the rows and counted in the attributes `n_old_without_doi` and `n_new_without_doi` and on the printed line. [scopus_diff_records\(\)](#) compares the records themselves, matching on the 'Scopus' identifier.

Value

A tibble of class `scopus_doi_diff` with columns `doi` and `status`, where `status` is a factor with levels "added" (in new only), "removed" (in old only) and "unchanged" (in both). Rows are sorted by status then DOI, in byte order, so the order is the same under every locale and in the Python twin. The integer attributes `n_old_without_doi` and `n_new_without_doi` count the records (or vector elements) on each side that had no DOI and so were not compared. Printing shows the counts in each category, and those records where there are any.

References

Gorraiz, J., Melero-Fuentes, D., Gumpenberger, C., & Valderrama-Zurián, J.-C. (2016). Availability of digital object identifiers (DOIs) in Web of Science and Scopus. *Journal of Informetrics*, 10(1), 98–109. doi:10.1016/j.joi.2015.11.008

See Also

[scopus_extract_dois\(\)](#), [scopus_diff_records\(\)](#)

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. Eleven records in each
# pull carry no DOI, and the printed line says so.
baseline <- example_records[example_records$year <= 2023, ]
later <- example_records[-1, ]
scopus_diff_dois(old = baseline, new = later)
```

scopus_diff_records *Compare two retrievals record by record*

Description

Identifies which records were added, removed or unchanged between an earlier and a later retrieval, and gives each record's citation count in both. `scopus_diff_dois()` compares DOIs alone, and this function compares the records themselves, matched as `scopus_combine(dedupe = TRUE)` matches them. A record without a DOI is therefore still tracked by its 'Scopus' identifier, and two records under one DOI are not taken for one.

Usage

```
scopus_diff_records(old, new)
```

Arguments

`old, new` `scopus_records` objects, or data frames with a `scopus_id` or a `doi` column, representing the earlier (old) and later (new) retrievals. The title, year and citations columns are read where present.

Details

Two records are one record when they share a 'Scopus' identifier, or when they share a DOI and one of them lacks an identifier. DOIs are compared as `scopus_extract_dois()` compares them, without resolver prefixes and ignoring case. Records with different identifiers are never matched, even under one DOI, since 'Scopus' gives one DOI to distinct papers as well as holding duplicate records of one paper (`scopus_combine()` gives the sources).

A record found in one retrieval only is a "conflict" where the other retrieval gives its DOI to a different record. It may have been re-indexed under a new identifier, or it may be a distinct paper sharing the DOI. Only the two records themselves can tell, so it is counted neither as added nor as removed. A record without an identifier whose DOI two identified records carry is in the same position. A record with neither an identifier nor a DOI cannot be matched at all, and is left out of the rows and counted.

`scopus_id` is the identifier the record was matched on, missing for a record matched on its DOI alone. `doi` is the DOI without its resolver prefix, in the case the record gives it. `doi`, `title` and `year` come from the later retrieval where it holds the record and gives a value, and from the earlier one otherwise, while `citations_old` and `citations_new` come from each retrieval in turn. Where one retrieval holds a record more than once, the first value it gives is used.

Value

A tibble of class `scopus_record_diff`, one row per record, with the columns `status`, `scopus_id`, `doi`, `title`, `year`, `citations_old` and `citations_new`. `status` is a factor with levels "added" (in new only), "removed" (in old only), "conflict" (see Details) and "unchanged" (in both). Rows are sorted by status, then DOI, identifier and title, in byte order, so the order is the same under every locale and in the Python twin. The integer attributes `n_old_without_id_or_doi` and `n_new_without_id_or_doi` count the records on each side that carried neither key and so were not compared. Printing shows the counts in each category, and those records where there are any.

See Also

`scopus_diff_dois()` to compare DOIs alone, `scopus_combine()` to merge the retrievals.

Examples

```
# The bundled corpus carries no 'Scopus' identifiers, so its records are
# matched on their DOIs, and the eleven in each pull without one cannot be
# compared at all.
baseline <- example_records[example_records$year <= 2023, ]
later <- example_records[-1, ]
scopus_diff_records(old = baseline, new = later)

# Two pulls of three records each, with identifiers as a live harvest
# carries them. 10.5555 is the DOI prefix reserved for tests, so none of
# these resolve. The later pull holds 10.5555/c under a new identifier,
# which makes both records under it a conflict to check, and a record from
# another source, without an identifier, that is new.
old <- data.frame(
  scopus_id = c("850000000001", "850000000002", "850000000003"),
  doi = c("10.5555/a", "10.5555/b", "10.5555/c"),
  citations = c(5L, 3L, 8L)
)
new <- data.frame(
  scopus_id = c("850000000001", "850000000033", NA),
  doi = c("10.5555/a", "10.5555/c", "https://doi.org/10.5555/D"),
  citations = c(9L, 10L, NA)
)
scopus_diff_records(old, new)
```

`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,
  fields = c("core", "extended")
)
```

Arguments

query	Character scalar. The base search expression, checked offline before any request (see the <i>Query checks</i> section of scopus_plan()).
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. scopus_count() , scopus_fetch() and scopus_intersections() send one inclusive range from the earliest year to the latest, as a scopus_plan() with partition = "none" does, so c(2015,

	2020) covers every year from 2015 to 2020. scopus_trend() , scopus_compare_topics() and a plan with <code>partition = "year"</code> take each listed year. An empty vector places no year restriction, as <code>NULL</code> does, where the years form a range, and is refused where each year is counted.
cursor	Logical. When <code>TRUE</code> , 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 <code>getOption("scopusflow.max_cursor_pages", 1e5)</code> pages with a warning; set that option to <code>Inf</code> 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 <code>TRUE</code> , progress is reported as the retrieval proceeds.
fields	"core" (the default) or "extended". "extended" also keeps the document type, source type, volume, issue, pages, ISSN, PubMed identifier and open-access status the Search API sends with every record, and the abstract under <code>view = "COMPLETE"</code> . They arrive in the same responses, so they cost no extra quota. See the <i>Extended fields</i> section of scopus_records() .

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. `retrieved_until` gives the same time as `retrieved_at` for a single search. A set combined from several keeps the earliest of their times as `retrieved_at` and the latest as `retrieved_until`. 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. `view` records the view, which decides what authors holds: the first author under `STANDARD` and the author list under `COMPLETE` (see [scopus_records\(\)](#)). `max_results` records the cap the call was given, when it was given one, so the search record can say whether the retrieval stopped at it. `plan` is the one-cell plan the call ran, `scopus_plan(query, years, field, view, page_size, partition = "none")`, so the search record states the field tag and the year limit the request carried. These attributes survive a `.rds` round trip through [write_scopus_records\(\)](#) but not a `.csv` one, which carries columns only. A retrieval that comes back with fewer records than the API reported warns with a `scopus_warning_shortfall`, as each cell of [scopus_fetch_plan\(\)](#) does, unless `max_results` or the offset ceiling stopped it.

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	<i>Execute a 'Scopus' search plan, with optional caching and resume</i>
-------------------	---

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,
  cursor = FALSE,
  fields = c("core", "extended")
)
```

Arguments

plan	A scopus_plan object from scopus_plan() .
max_results	Maximum records to retrieve per cell (default Inf). A finite cap is recorded on the result, and scopus_search_report() says which cells stopped at it.
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, page size and paging mode 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. A COMPLETE checkpoint written by a release that read authors from the first author alone (0.4.0 and earlier) is a miss as well, and is refetched once with a warning. STANDARD checkpoints from those releases are served as before, since their authors were read correctly. Checkpoints also record their fields. One fetched with `fields = "extended"` serves a core request, with the extra columns dropped, while a core one is a miss for an extended request and is refetched with a warning. A checkpoint written before the fields were recorded counts as core. A `cache_dir` that is not a directory and cannot be created as one, or in which no file can be created, is refused with a `scopus_error_bad_input` error before any request. A checkpoint that cannot be written all the same, for example on a full disk, gives a `scopus_warning_cache_unwritable` warning, and the cell's records are kept in the result.

<code>resume</code>	Logical. When TRUE and <code>cache_dir</code> is set, a cell whose cache file already exists is loaded from disk, sparing a second request.
<code>api_key, inst_token</code>	Optional credentials (see scopus_has_key()).
<code>verbose</code>	Logical. When TRUE, per-cell progress is reported.
<code>cursor</code>	Logical. When TRUE, each cell is retrieved with cursor-based pagination, which has no 5000-record ceiling, so a cell matching more records than that is retrieved whole and still checkpointed. The records then arrive in the API's deep-paging order (see scopus_fetch()). The default, FALSE, pages by offset, as plans always have, so a cell stops at its first 5000 records with a warning. Checkpoints record the paging mode and serve only requests made in the same mode. A checkpoint written before the mode was recorded counts as offset-paged. The Python twin's <code>fetch_plan()</code> pages by cursor unless told otherwise, the mode it has always used.
<code>fields</code>	"core" (the default) or "extended". "extended" also keeps the document type, source type, volume, issue, pages, ISSN, PubMed identifier and open-access status the Search API sends with every record, and the abstract under a COMPLETE plan. They arrive in the responses each cell already pays for, so they cost no extra quota. Like <code>cursor</code> , it is an argument of the <code>fetch</code> , since it changes how the records are read while the search returns the same records. See the <i>Extended fields</i> section of scopus_records() .

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. A finite `max_results` is attached as the attribute of that name, the cap of this call: a checkpoint written under a wider cap is served trimmed to it, so no cell holds more. `scopus_search_report()` reads all of this back. The `retrieved_at`, `retrieved_until` and `scopusflow_version` attributes described in `scopus_fetch()` are carried across from the cells. `retrieved_at` is the earliest of their times, since a combined set is only as fresh as its oldest cell. `retrieved_until` is the latest, the last search, which the record of a harvest resumed over several days has to give. Every version that contributed is listed, since resuming an older cache means more than one did. Each is omitted when any cell cannot supply it, as a checkpoint written before these attributes existed cannot. Dating the whole from the part of it that can be dated would misreport the set. A cell written before `retrieved_until` existed counts by its `retrieved_at`. paging and view are kept where every cell records the same one.

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 a plan
# attached. 'Scopus' records may not be redistributed, so the bundled corpus
# of real articles stands in for the harvest. It came from OpenAlex's
# exact-pharse search for the same phrase, so the plan, one cell per year,
# describes a 'Scopus' search that was not run and would return other
# records.
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")
```

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	<i>Locate the 'Scopus' API key and institutional token</i>
----------------	--

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 `~/.Renviro`n, 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 = " x ",
  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 <code>list(c("A", "B"), c("A", "B", "C"))</code> . 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. <code>scopus_count()</code> , <code>scopus_fetch()</code> and <code>scopus_intersections()</code> send one inclusive range from the earliest year to the latest, as a <code>scopus_plan()</code> with <code>partition = "none"</code> does, so <code>c(2015, 2020)</code> covers every year from 2015 to 2020. <code>scopus_trend()</code> , <code>scopus_compare_topics()</code> and a plan with <code>partition = "year"</code> take each listed year. An empty vector places no year restriction, as <code>NULL</code> does, where the years form a range, and is refused where each year is counted.
field	Optional 'Scopus' field tag wrapped around each concept value that is not already a complete field-tagged expression (see <code>scopus_field_tags()</code>).
view	Either "STANDARD" or "COMPLETE". COMPLETE adds an <code>authkeywords</code> column to <code>scopus_fetch()/scopus_fetch_plan()</code> output (see <code>scopus_records()</code>) 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 <code>scopus_has_key()</code>).
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, for example 2015:2020. When partition = "year", one plan cell is created for each distinct year, so c(2015, 2020) searches those two years alone, as scopus_trend() and scopus_compare_topics() do. Otherwise, the plan sends one inclusive range from the earliest year to the latest, so c(2015, 2020) covers every year from 2015 to 2020, as in scopus_count() , scopus_fetch() and scopus_intersections() . An empty vector places no year restriction, as NULL does.
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.

Query checks

`scopus_plan()`, [scopus_count\(\)](#), [scopus_fetch\(\)](#), [scopus_trend\(\)](#), [scopus_compare_topics\(\)](#) and [scopus_intersections\(\)](#) check each query offline, before any request, and refuse three mis-

takes with a `scopus_error_bad_input` condition. `LIMIT-TO()` and `EXCLUDE()` belong to the web interface's refinement panel, and the Search API ignores them, so the search would run unfiltered. Write such a limit as a search field, for example `DOCTYPE(ar)`, `LANGUAGE(english)`, `SUBJAREA(PSYC)`, `SRCTYPE(j)` or `PUBYEAR > 2015`. A field given for a query that already opens with a field tag would nest one tag inside the other, which the API rejects, so leave field as `NULL` for such a query. Brackets, quotation marks and braces must balance, counting brackets inside a phrase as text. `INDEXTERMS()`, which pybliometrics documents as not working through the Search API, raises a `scopus_warning_query_syntax` warning. `KEY()` searches the index terms together with author keywords, trade names and chemical names. The messages are those of the Python twin, word for word.

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	<i>Build a field-tagged 'Scopus' query</i>
--------------	--

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

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

Details

Each term is trimmed and, given `.field`, wrapped in that tag, whose brackets hold the term together. Without a field tag, a term of several words is put in brackets before two or more terms are joined, because 'Scopus' joins the words of an unquoted term with AND. Joined bare, machine learning OR deep learning would read as machine AND (learning OR deep) AND learning. A term that is already one operand is left as written: a quoted "... " phrase, a braced {...} phrase,

a bracketed group or a single field-tagged group such as `TITLE(deep learning)`. Single words are never bracketed, so `scopus_query("CRISPR", "Cas9", .op = "OR")` stays `CRISPR OR Cas9`, and a query composed by an earlier call keeps its meaning when it is joined again. A term that already opens with a field tag is refused when `.field` is given, because the API rejects one tag nested inside another.

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")
# Without a field tag, multi-word terms are bracketed and phrases kept as written.
scopus_query("machine learning", "deep learning", .op = "OR")
scopus_query('"heart attack"', '"myocardial infarction"', .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`, `status` and `retry_after` (seconds, or NA). `reset` is the POSIXct time, in UTC, at which the key's quota resets, read from X-RateLimit-Reset (epoch seconds, per Elsevier's API key settings page), 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

<code>x</code>	A scopus_records object, which supplies the counts and the retrieval provenance and, through its <code>plan</code> attribute, the plan; or a bare scopus_plan() for a search that has not been run yet. For <code>format()</code> and <code>print()</code> , the report object itself.
<code>plan</code>	Optional scopus_plan() describing <code>x</code> . Supply it when a record set does not carry one, for example one read back from a .csv, a merged set, or one saved by scopus_fetch() in <code>scopusflow</code> 0.4.0 or earlier, which attached none. An explicit plan takes precedence over the one <code>x</code> carries. A one-cell plan supplied for a set without per-cell counts takes the set's own row count and <code>total_results</code> .

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 dates of the search are the `retrieved_at` and `retrieved_until` attributes `scopus_fetch()` attaches, never the current time. A harvest resumed over several days, or a baseline merged with its update, is dated from its first retrieval to its last. PRISMA-S item 13 then gives the date of the last search, as the checklist asks. Stamps that fall on one UTC day give that single date. The years are those the cells' limits sent, and years with gaps between them are listed, since a span would claim years no cell searched. The number of records the API reported as matching is what the cells recorded (the `cell_totals` attribute `scopus_fetch_plan()` attaches, or, for a set from a single search, its `total_results`), 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, matched by the rule it recorded. 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.

A harvest capped by `max_results` says so. A cell that holds as many records as the cap allowed, where the API reported more or reported nothing, reads "stopped at the requested cap". The completeness line puts the shortfall down to the cap when every cell that fell short stopped at it, and otherwise counts the cells that did. Item 9 names the cap, or says that it did not bind, and the reproduction snippet passes it. The Python twin's `fetch_plan()` can also be stopped between cells, as its app's Cancel stops it. Its record then says how many of the planned cells ran, the years they covered and that the harvest is incomplete, and it marks the rest "not run". This package's harvests cannot stop part-way, but a set carrying those attributes is described the same way, so that the two records stay one rendering.

The PRISMA-S map is decided the same way. Items the package holds evidence for (the database and platform, that Scopus was searched on its own, the full strategy and the limits where a plan records them, the date, the totals, and de-duplication where it was performed) are listed as supplied. Without a plan, the year limit is unrecorded, not absent, and the record says so. The rest, among them peer review of the strategy, grey literature 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`, `max_results` (the cap on records per cell), `partition`, `n_cells`, `years`, `years_run` (the years of the cells a stopped harvest ran), `cells` (a tibble of `cell`, `limit`, `n_records` and `reported_total`), `searched_at`

and searched_until (the earliest and latest retrieval times), version, n_records, n_with_doi, reported_total, cells_reported (how many cells reported a total), stopped_early, cells_planned, cells_run, not_run (the cells a stopped harvest never reached), records_combined, duplicates_removed, deduplicated, dedupe_rule and doi_conflicts (the rule `scopus_combine()` matched records by, and the DOIs it left shared across identifiers), 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. Their values are
# illustrative, and the records came from OpenAlex, so the record below
# describes no search that was run.
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 are split at the semicolon, with or without a space after it, so each contributor is counted once per record. Only a view = "COMPLETE" retrieval holds several names, since STANDARD records carry the first author alone (see [scopus_records\(\)](#)). A tally by author of a set retrieved under STANDARD therefore counts first authorships, and says so with a warning of class `scopus_warning_first_author_only`, once per session.
- 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 harvest retrieved with view = "COMPLETE" lists every
# author, and the tally splits them.
scopus_top(example_records, by = "author", n = 5)
```

scopus_trend	<i>Annual publication counts for a query</i>
--------------	--

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, checked offline before any request (see the <i>Query checks</i> section of scopus_plan()).
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()`. The time the counts were taken is attached as the `retrieved_at` attribute (a POSIXct), as a harvest records its own. 'Scopus' goes on indexing a year's publications after the year has begun, so a count for the year of retrieval is a count so far, and [plot_scopus_trend\(\)](#) says so from this time.

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 OpenAlex's exact-phrase search for the same
# phrase, so tallying its rows by year gives the yearly counts of that
# search. A 'Scopus' count of the phrase would differ.
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), `top_source` (the most frequent source title) and `doc_types`. `doc_types` counts the records of each document-type code, most frequent first, for a set retrieved with `fields = "extended"` (see [scopus_records\(\)](#)), and is NULL for a set without the `doc_type` column. Printing it produces a short readable report, which

for such a set also says how many records 'Scopus' labels errata (er) or retracted (tb). The counts report the labels 'Scopus' assigns, which do not screen a set for retractions. The *Extended fields* section of [scopus_records\(\)](#) explains why, and what to check a set against.

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, and so do the columns `fields = "extended"` adds. A `.csv` file reads them back as text, apart from the integer `open_access`, so an ISSN keeps its leading zeros. 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

<code>x</code>	A scopus_records tibble to write.
<code>path</code>	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.

Details

A `.csv` file is written and read as UTF-8 in every locale. One written by another tool reads back as well, and a blank field in it reads as missing, since pandas, the Python twin and spreadsheet software write a missing value that way, where R writes NA.

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

- * **datasets**
 - example_records, 11
- as.data.frame.scopus_records
 - (as_tibble.scopus_records), 7
- as_bibliometrix, 3
- as_bibliometrix(), 5, 8, 27, 29
- as_bibtex, 4
- as_ris(as_bibtex), 4
- as_theoryforge_corpus, 5
- as_tibble.scopus_records, 7
- autoplot.scopus_comparison
 - (plot_scopus_comparison), 13
- autoplot.scopus_intersections
 - (plot_scopus_intersections), 15
- autoplot.scopus_records
 - (as_tibble.scopus_records), 7
- autoplot.scopus_top(plot_scopus_top), 16
- autoplot.scopus_trend
 - (plot_scopus_trend), 17
- c.scopus_records(scopus_combine), 24
- example_records, 11, 18
- format.scopus_search_report
 - (scopus_search_report), 48
- ggplot2::ggplot, 9, 14–17
- httr2::response, 47
- is_scopus_plan(scopus_plan), 44
- is_scopus_records
 - (as_tibble.scopus_records), 7
- plot_scopus_comparison, 13
- plot_scopus_comparison(), 27
- plot_scopus_intersections, 15
- plot_scopus_intersections(), 44
- plot_scopus_top, 16
- plot_scopus_top(), 51
- plot_scopus_trend, 17
- plot_scopus_trend(), 9, 52
- print.scopus_search_report
 - (scopus_search_report), 48
- read_scopus_records
 - (write_scopus_records), 54
- run_app, 18
- scopus_abstract, 19
- scopus_abstract(), 3, 5, 6, 10, 28, 29
- scopus_cache_clear, 22
- scopus_cache_clear(), 23, 40
- scopus_cache_dir, 23
- scopus_cache_dir(), 23, 40
- scopus_combine, 24
- scopus_combine(), 33, 34, 49, 50
- scopus_compare_topics, 25
- scopus_compare_topics(), 13, 14, 31, 37, 41–43, 45, 51, 52
- scopus_corpus, 27
- scopus_corpus(), 5–7, 21
- scopus_count, 30
- scopus_count(), 21, 26, 30, 36, 37, 40, 42–47, 52
- scopus_diff_dois, 32
- scopus_diff_dois(), 33–35
- scopus_diff_records, 33
- scopus_diff_records(), 32
- scopus_extract_dois, 34
- scopus_extract_dois(), 5, 21, 24, 32, 33
- scopus_fetch, 36
- scopus_fetch(), 7, 8, 12, 21, 27, 30, 31, 36, 39–43, 45, 47–49, 52, 54
- scopus_fetch_plan, 38
- scopus_fetch_plan(), 7, 8, 18, 20, 23, 25, 27, 31, 36, 37, 43, 46, 49, 50, 52
- scopus_field_tags, 40

`scopus_field_tags()`, 43, 45–47
`scopus_has_key`, 41
`scopus_has_key()`, 20, 26, 28, 31, 37, 39, 43, 52
`scopus_intersections`, 42
`scopus_intersections()`, 15, 30, 36, 43, 45
`scopus_plan`, 44
`scopus_plan()`, 18, 26, 30, 31, 36, 38, 41, 43, 47, 48, 50, 52
`scopus_query`, 46
`scopus_query()`, 10
`scopus_quota`, 47
`scopus_quota()`, 21, 31
`scopus_records`, 3, 4, 8, 12, 24, 25, 32–39, 48, 50, 51, 53, 54
`scopus_records`
 (`as_tibble.scopus_records`), 7
`scopus_records()`, 3, 4, 20, 27, 28, 31, 36, 37, 39, 43, 45, 51–54
`scopus_search_report`, 48
`scopus_search_report()`, 25, 37, 38, 40
`scopus_top`, 50
`scopus_top()`, 8, 16
`scopus_trend`, 51
`scopus_trend()`, 17, 31, 37, 43, 45
`shiny::runApp()`, 18
`summary()`, 10, 53
`summary.scopus_records`, 53
`summary.scopus_records()`, 51

`tibble`, 7, 9, 41
`tolower()`, 6
`tools::R_user_dir()`, 23, 38
`tryCatch()`, 31

`write_scopus_corpus`
 (`as_theoryforge_corpus`), 5
`write_scopus_corpus()`, 29
`write_scopus_records`, 54
`write_scopus_records()`, 5, 37