Package: openalexPro (via r-universe)

November 20, 2026

Type Package

Title Providing a more advanced access to OpenAlex for the power user

Version 0.18.1

Description Retrieves large result sets from the 'OpenAlex' API (<https://openalex.org>) without holding them in memory. Queries are built and validated with classic filters and search, or written in 'OpenAlex' query language (OQL) or its JSON form (OQO). Results are downloaded page by page to JSON files, with cursor paging, parallel downloads, retries and resume, and converted per page with 'DuckDB' to a partitioned 'Apache Parquet' dataset that includes reconstructed abstracts and citation strings. Query results can be saved as 'OpenAlex' collections, shareable and reusable sets of IDs that are kept in sync with their queries. Full-text PDFs and TEI XML can be downloaded, and the daily API budget can be inspected.

URL https://github.com/openalexPro/openalexPro, https://openalexpro.github.io/openalexPro/, https://doi.org/10.5281/zenodo.17453180

BugReports https://github.com/openalexPro/openalexPro/issues

License MIT + file LICENSE

Depends R (>= 4.2)

Imports arrow, cli, curl, DBI, dplyr, duckdb, future, future.apply, httr2 (>= 1.0.0), jqr, jsonlite, progressr, rlang

Additional_repositories https://ropensci.r-universe.dev

Suggests keyring, knitr, openalexR (>= 1.4.0), rmarkdown, quarto, testthat (>= 3.0.0), vcr (> 1.7.0), vdiffr, withr

Encoding UTF-8

VignetteBuilder quarto

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake libjq-dev libxml2-dev libssl-dev

Repository https://openalexpro.r-universe.dev

Date/Publication 2026-11-20 08:21:38 UTC

RemoteUrl https://github.com/openalexPro/openalexPro

RemoteRef main

RemoteSha 9aa092c59e3aaf842b46aaf7b801572b2baebb5c

Contents

compatibility_report . . . 3
extract_doi . . . 4
id_block . . . 5
infer_json_schema . . . 6
jq_execute . . . 8
oa_normalize_duckdb_type . . . 9
oa_schema . . . 9
oa_works_abstract_sql . . . 10
oa_works_citation_sql . . . 11
opt_api_key . . . 11
opt_filter_names . . . 12
opt_select_fields . . . 12
pro_api_key . . . 13
pro_collection . . . 14
pro_compact_parquet . . . 17
pro_download_content . . . 18
pro_fetch . . . 20
pro_query . . . 23
pro_rate_limit_status . . . 26
pro_request . . . 27
pro_request_jsonl . . . 29
pro_request_jsonl_parquet . . . 30
pro_request_parquet . . . 31
pro_validate_credentials . . . 34
read_corpus . . . 35
sample_parquet_n . . . 35

Index **38**

`compatibility_report` *Render and open the compatibility report*

Description

Renders the Quarto report at ‘system.file("compatibility.qmd", package = "openalexPro") and opens the resulting HTML in your default browser.

Usage

```
compatibility_report(
  output_dir = "Compatibility Report",
  open = TRUE,
  quiet = FALSE
)
```

Arguments

`output_dir`	Directory to write the rendered HTML and the data into. Defaults to the flder ‘./Compatibility Report‘.
`open`	Logical; if ‘TRUE‘ (default) opens the rendered HTML in the system browser.
`quiet`	Logical; suppress rendering output if ‘TRUE‘. Default: ‘FALSE‘.

Details

This report is designed to help you validate client-API compatibility in real time. During rendering, the report performs live requests against the OpenAlex API and compares the responses to the package’s expected behavior. No cached data are used: every section issues fresh API calls so that the output reflects the current state of the upstream service. The report summarizes differences in fields, types, pagination and response shapes to surface potential regressions from upstream changes or local client updates.

Note: Because it depends on live API calls, rendering may take longer and requires network access. Be mindful of API rate limits when running the report repeatedly.

Value

Invisibly returns the path to the rendered HTML file.

`extract_doi` *Extract DOIs or Components from Character Vectors*

Description

Extracts DOIs or specific DOI components (resolver, prefix, or suffix) from a character vector. Assumes that each element of ‘x‘ contains at most one DOI (with or without resolver).

Usage

```
extract_doi(
  x,
  non_doi_value = "",
  normalize = TRUE,
  lower = TRUE,
  what = c("doi", "resolver", "prefix", "suffix")
)
```

Arguments

`x`	A character vector potentially containing DOIs (e.g., raw DOIs, DOI URLs, or strings with embedded DOIs).
`non_doi_value`	Value to use for elements where no DOI or component is found. If ‘NULL‘, only matched elements are returned.
`normalize`	Logical. If ‘TRUE‘ (default), convert extracted DOIs and suffixes to lowercase and trim surrounding whitespace. Has no effect for ‘what = "prefix"‘ or ‘what = "resolver"‘.
`lower`	Logical. With ‘normalize = TRUE‘, convert to lowercase (default). ‘FALSE‘ keeps the upper and lower case of the input, e.g. to compare the cleaned DOI with the field it came from.
`what`	What to extract from each element. One of: **"doi"** The full DOI name (prefix + "/" + suffix). Example: ‘"10.5281/zenodo.1234567"‘ (default) **"resolver"** The resolver URL (e.g., ‘"https://doi.org/"‘, ‘"http://dx.doi.org/"‘) if present **"prefix"** The DOI prefix only (e.g., ‘"10.5281"‘) **"suffix"** The DOI suffix only (e.g., ‘"zenodo.1234567"‘)

Details

Percent-encoded characters are decoded first, also when encoded twice (‘ encoded line break (‘ not an encoding, such as the unfilled placeholders in ‘10.18461/ijfsd.v2i4. or ‘10.37002/biobrasil.v cut short to something that looks valid. A full stop or semicolon at the end is taken as punctuation and dropped.

Value

A character vector: - If 'non_doi_value' is not 'NULL', a vector of the same length as 'x', with unmatched entries replaced. - If 'non_doi_value' is 'NULL', a vector of only matched entries.

Examples

```
x <- c(
  "https://doi.org/10.5281/zenodo.1234567",
  " 10.1000/XYZ456  ",
  "no doi here",
  NA
)

extract_doi(x)  # Full DOIs (default)
extract_doi(x, what = "resolver")
extract_doi(x, what = "prefix")
extract_doi(x, what = "suffix")
extract_doi(x, non_doi_value = NA_character_)
extract_doi(x, non_doi_value = NULL)
```

---

id_block *Compute ID block from OpenAlex IDs*

Description

This function computes the ID block partition key from OpenAlex IDs. The ID block is calculated as the trailing numeric portion of the ID divided by 10,000.

Usage

```
id_block(ids)
```

Arguments

ids	Character vector of OpenAlex IDs in any format: • Short form: "W2741809807" • Long form: "https://openalex.org/W2741809807" • Path-based: "https://openalex.org/domains/2"

Details

OpenAlex IDs come in several formats:

- Standard: https://openalex.org/{type}{number} (e.g., W1234567890)
- Path-based: https://openalex.org/{entity_type}/{number} (e.g., domains/2, subfields/2208)

The ID block is computed as floor(number / 10000), where number is the trailing numeric portion of the ID. This groups approximately 10,000 IDs into each block, useful for partitioning large datasets.

Value

Integer vector of ID blocks.

Examples

```
# Short form IDs
id_block(c("W2741809807", "W2741809808", "W1234567890"))
# Returns: c(274180, 274180, 123456)

# Long form IDs
id_block("https://openalex.org/W2741809807")
# Returns: 274180

# Path-based IDs
id_block("https://openalex.org/domains/2")
# Returns: 0

# Works with any entity type
id_block(c("A123456789", "I987654321"))
# Returns: c(12345, 98765)
```

---

infer_json_schema *Infer unified JSON schema using DuckDB*

Description

Infers the schema of each JSON/NDJSON file individually via DuckDB's 'read_json_auto()' and merges the results using type-widening rules. Processing files one at a time avoids the out-of-memory errors that occur when opening all files in a single DuckDB query.

Usage

```
infer_json_schema(
  con,
  files,
  sample_size = 20,
  extra_options = "",
  verbose = TRUE,
  schema_cache_dir = NULL
)
```

Arguments

con	An active DuckDB connection ('DBI::dbConnect(duckdb::duckdb())') with the JSON extension loaded ('LOAD json').
files	Character vector of paths to JSON or NDJSON ('.gz') files.

`sample_size` Number of files to sample for schema inference. Higher values give more accurate schemas but take longer. Use ‘0‘ or ‘NULL‘ to use all files. Default is ‘20‘.

`extra_options` Additional options appended to the ‘read_json_auto‘ SQL call, e.g. ‘", maximum_object_size=1000000000"‘ for large JSON objects. Default is ‘""‘.

`verbose` If ‘TRUE‘, print progress messages and a progress bar. Default is ‘TRUE‘.

`schema_cache_dir`
Path to a directory for caching per-file and unified schemas. The directory is created if it does not exist. ‘NULL‘ (default) disables caching.

Details

The returned columns clause can be passed directly to ‘read_json(..., columns = <result>)‘ or ‘read_json_auto(..., columns = <result>)‘ in subsequent DuckDB queries to enforce a consistent schema across all files.

Value

A DuckDB columns clause string (e.g. `"{'col1': 'VARCHAR', 'col2': 'BIGINT', ...}"`) suitable for use as the ‘columns‘ argument to ‘read_json()‘. Returns ‘NULL‘ if schema inference fails for all files.

Caching

When ‘schema_cache_dir‘ is provided, two levels of caching apply: - **Unified schema** (‘unified_schema.csv‘): if present, loaded and returned immediately - no DuckDB queries needed. Delete this file to force re-inference. - **Per-file schemas** (‘<update_date>_<part_name>.csv‘): each file's schema is saved as it is inferred. On restart, already-cached files are skipped, enabling mid-run resume for large file sets.

Type-widening rules

When a column has different types across files, the unified type is chosen by these rules (in order): 1. All identical -> keep as-is. 2. Any ‘STRUCT‘/‘LIST‘/‘MAP‘ vs simpler type -> complex type wins. 3. Multiple ‘STRUCT‘ types -> pick the one with the most fields. 4. All types numeric -> widest type wins (‘TINYINT < SMALLINT < INTEGER < BIGINT < HUGEINT < FLOAT < DOUBLE‘). ‘JSON‘, which DuckDB infers for a column that is null throughout a file, is ignored here. 5. Fallback -> ‘VARCHAR‘.

See Also

[pro_request_jsonl_parquet()] (deprecated), which uses this function internally.

Examples

```
## Not run:
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbExecute(con, "LOAD json")
files <- list.files("path/to/snapshot/works", pattern = "\\.gz$",
```

```
                    recursive = TRUE, full.names = TRUE)
schema <- infer_json_schema(con, files, sample_size = 50,
                            schema_cache_dir = "path/to/cache")
DBI::dbDisconnect(con, shutdown = TRUE)
# schema is now a string like: {'id': 'VARCHAR', 'title': 'VARCHAR', ...}

## End(Not run)
```

`jq_execute` *Execute a jq transformation from an OpenAlex-style JSON to JSONL*

Description

This function runs a jq filter to extract records from the "results" array (or from the root if type = "single"), reconstruct the abstract text, generate a citation string, and optionally add a page field. It writes the result as newline-delimited JSON (.jsonl), suitable for Arrow or DuckDB. For details on the jq filter logic, see the `vignette`("jq", package = "openalexPro").

Usage

```
jq_execute(
  input_json,
  output_jsonl,
  add_columns = list(),
  jq_filter = NULL,
  page = NULL,
  type = c("results", "single", "group_by")
)
```

Arguments

`input_json`	Path to the input JSON file
`output_jsonl`	Path to the output .jsonl file
`add_columns`	List of additional fields to be added to the output. They nave to be provided as a named list, e./g. ‘list(column_1 = "value_1", column_2 = 2)‘. Only Scalar values are supported.
`jq_filter`	Optional custom jq filter string. If NULL, the default filter is used.
`page`	Optional integer to be added as a "page" field in each output record
`type`	Either "results" (default, expects a .results[] array) or "single" (treat input as array of records directly)

Value

Invisibly returns the output path

oa_normalize_duckdb_type *Canonicalise a DuckDB type string.*

Description

Uppercases SQL type keywords (`BIGINT`, `VARCHAR`, `STRUCT`, ...) while preserving the case of struct field identifiers.

Usage

```
oa_normalize_duckdb_type(t)
```

Arguments

`t`	A character scalar: a raw DuckDB type string, e.g. `"struct(author struct(display_name varchar))"`.

Value

A character scalar with normalised type keywords.

oa_schema *Get or refresh an OpenAlex entity schema*

Description

Returns the baseline column schema for an OpenAlex entity (used by `pro_request_parquet(schema = "auto")` to resolve ambiguous DuckDB JSON types), and optionally refreshes the user-level cache from a local Parquet corpus.

Usage

```
oa_schema(
  entity = NULL,
  parquet_dir = NULL,
  entities = "all",
  overwrite = FALSE,
  update = FALSE,
  verbose = TRUE
)
```

Arguments

entity	Character scalar. Entity name, e.g. "works". Required when update = FALSE; ignored when update = TRUE and entities != "all" is a vector.
parquet_dir	Character scalar. Root Parquet directory containing one sub-directory per entity, e.g. "/Volumes/openalex/parquet". Required when update = TRUE; ignored otherwise.
entities	Character vector or "all" (default). Entities to refresh when update = TRUE. "all" auto-discovers sub-directories of parquet_dir, excluding *_aws staging directories.
overwrite	Logical. When update = TRUE, overwrite an existing cached CSV? Default FALSE.
update	Logical. When TRUE, read schema from parquet_dir and write to the user cache. Default FALSE.
verbose	Logical. Print progress messages? Default TRUE.

Value

update = FALSE: a data.frame with columns col_name and col_type, or NULL if no schema is found for entity.
update = TRUE: the path to the schemata cache directory (invisibly).

Resolution order (update = FALSE)

1. User cache (tools::R_user_dir("openalexPro","cache")/schemata/<entity>.csv)
2. Schemas bundled with the package (inst/extdata/schemata/<entity>.csv)

Refreshing the cache (update = TRUE)

Reads the schema of each entity corpus from parquet_dir using DuckDB and writes the result to the user cache. After updating, the new schema is available to subsequent calls with update = FALSE and to pro_request_parquet(schema = "auto"). Run periodically to pick up new fields added by OpenAlex.

See Also

pro_request_parquet for the schema parameter.

---

oa_works_abstract_sql *Return the DuckDB SQL expression that reconstructs a plain-text abstract from the* abstract_inverted_index *column in OpenAlex works data.*

Description

The expression walks the map, collects (position, word) pairs, sorts by position ascending, and joins words with single spaces. Returns NULL when abstract_inverted_index is NULL.

Usage

```
oa_works_abstract_sql()
```

Details

`abstract_inverted_index` is normalised to `MAP(VARCHAR, BIGINT[])` via a double JSON cast (`::JSON::MAP(VARCHAR, BIGINT[])`) before `map_entries()` is called. This makes the expression safe regardless of whether DuckDB inferred the column as `MAP`, `STRUCT`, or `VARCHAR` (raw JSON text): all three round-trip through the JSON representation identically.

Value

A character scalar containing the SQL expression.

`oa_works_citation_sql` *Return the DuckDB SQL expression that builds a short citation string from the* `authorships` *and* `publication_year` *columns in OpenAlex works data.*

Description

Format: `"Author (year)"` / `"A & B (year)"` / `"A et al. (year)"`. Null year renders as `"(n.d.)"`. Null or empty `authorships` yields NULL.

Usage

```
oa_works_citation_sql()
```

Value

A character scalar containing the SQL expression.

`opt_api_key` *Get API key for OpenAlex API*

Description

Get API key for OpenAlex API

Usage

```
opt_api_key(api_key)
```

Arguments

`api_key`	character vector or NULL. If specified, value to assign to the api key option. Default is 'NULL'.

Value

The API key, if ‘api_key‘ is not specified the current one, otherwise the old one.

`opt_filter_names` *Get available filter names from OpenAlex API*

Description

Valid filter names differ between entities (e.g. ‘country_code‘ is a filter for institutions but not for works). The list is taken from the error message OpenAlex returns for an invalid filter, which names exactly the filters it accepts for that entity; error responses cost no credits. Each entity’s list is cached for the R session.

Usage

```
opt_filter_names(update = FALSE, entity = "works")
```

Arguments

`update`	logical. If ‘TRUE‘ update the existing value. Default is ‘FALSE‘.
`entity`	OpenAlex entity, e.g. ‘"works"‘ (default), ‘"authors"‘, ‘"institutions"‘. Use the same value as ‘entity‘ in [pro_query()].

Value

A character vector of available filter names. Empty if OpenAlex does not return a list for ‘entity‘ (e.g. the retired ‘"venues"‘).

`opt_select_fields` *Get available select fields from OpenAlex API*

Description

Valid select fields differ between entities (e.g. ‘orcid‘ for authors). The list is taken from the error message OpenAlex returns for an invalid select field, which names exactly the fields it accepts for that entity; error responses cost no credits. Each entity’s list is cached for the R session.

Usage

```
opt_select_fields(update = FALSE, entity = "works")
```

Arguments

`update`	logical. If ‘TRUE‘ update the existing value. Default is ‘FALSE‘.
`entity`	OpenAlex entity, e.g. ‘"works"‘ (default), ‘"authors"‘, ‘"institutions"‘. Use the same value as ‘entity‘ in [pro_query()].

Value

A character vector of available select fields. Empty if OpenAlex does not return a list for ‘entity‘ (e.g. the retired ‘"venues"‘).

`pro_api_key` *Retrieve the OpenAlex Pro API key*

Description

Retrieves the OpenAlex Pro API key from one of several locations, checked in the following order:

Usage

```
pro_api_key()
```

Details

1. The R option `openalexPro$api_key`
2. The environment variable `openalexPro.api_key`
3. The system keyring via the **keyring** package (only if the package **keyring** is installed)

If no API key is found, `NULL` or an empty string may be returned, depending on the environment variable state.

Value

A character string containing the API key, or `NULL` if no key could be found.

See Also

`options`, `Sys.getenv`

Examples

```
## Not run:
pro_api_key()

options(openalexPro = list(api_key = "my-key"))
pro_api_key()

## End(Not run)
```

`pro_collection` *Create OpenAlex collections from queries*

Description

Turns each query into an OpenAlex collection: a saved set of IDs on OpenAlex's servers that can be reused in later queries (`pro_query(collection = "col_...")`) and combined with other collections. OpenAlex runs the query itself (an *import*); `pro_collection()` waits for it and then downloads what `select` asks for.

Usage

```
pro_collection(
  query_url,
  name = NULL,
  project_folder,
  select = "id",
  refresh = FALSE,
  access = NULL,
  api_key = pro_api_key(),
  verbose = FALSE,
  progress = TRUE,
  ...
)
```

Arguments

`query_url`	A query from `pro_query()`, or a named (nested) list of queries. Without `name`, each element of the list becomes its own collection, named after its list names (nested names joined with `_`). With `name`, all queries in the list go into one collection (see **Several queries in one collection**).
`name`	Name of the collection (and of its folder). Required for a single query. For a list: `NULL` gives one collection per element, a name gives one collection for the whole list.
`project_folder`	Directory for the `<name>.collection` folders.
`select`	What to download: `"id"` (default) the member IDs only; `NULL` nothing; `"all"` the full records; or a character vector of fields (see `opt_select_fields()`) for records with those fields. The member IDs are downloaded whenever `select` is not `NULL`.
`refresh`	If `TRUE`, make the members of the collection match the query now (adding and removing members; the ID stays) and download again. Also takes over a collection of the same name that is not recorded in `project_folder`.
`access`	Who can see the collection: `"private"` (only you, the default for new collections), `"shared_by_link"` (anyone with its link or ID can view it and filter by it; it is not listed) or `"public"` (only OpenAlex can set this for now; OpenAlex refuses it with HTTP 403). `NULL` (default) leaves it as it is.

api_key	A personal OpenAlex API key.
verbose	Print the collection ID(s).
progress	Show the steps and import progress.
...	Passed to pro_fetch() when records are downloaded, and when the queries of a list with name are fetched (e.g. workers, compact_parquet).

Details

Each query gets a folder <project_folder>/<name>.collection/ with

_collection.json the query (URL, OQL, OQO), the collection (collection_id, display_name, entity_type, member_count, ...), each import, the credits used and the select that was downloaded.

_query_oql.txt, _query_oql.html the query in readable OQL.

_parts.parquet for a list of queries in one collection (see **Several queries in one collection**): the name and URL of each query. _collection.json then holds no queries, so it stays small however long the list is.

members/parquet/ the member IDs (id, added_at).

records/ the records when select asks for more than id (the pro_fetch() layout).

Running it again with the same project_folder and name:

- Same query: the collection is kept (same ID). An import still running (e.g. after an interruption) is waited for. Only the downloads are brought in line with select: removed fields are dropped locally (no requests), added fields re-download the records. Credits are charged per request, not per field, so fetching only the new fields would cost the same.
- refresh = TRUE, with the same or a changed query: the collection keeps its ID, and its members are made to match the query now: the query's IDs are fetched with pro_fetch() (select = "id", same credits as an import), new ones are added and members no longer found are removed (free, 10,000 added or 100 removed per request). Everything is then downloaded again.
- Changed query without refresh: the old collection is deleted and a new one is created under the same name (it gets a new ID).

So refresh = TRUE always keeps the ID (links and saved searches keep working). Only when the collection is gone on OpenAlex, or holds another entity, is a new one created.

Local folder gone: a collection with the same name that exists in your account but is not recorded in project_folder is never deleted. Without refresh, pro_collection() stops. With refresh = TRUE it takes that collection over: the ID stays, its members are synced to the query as above, and the folder is written again. Names match ignoring case, and the collection must hold the same entity.

Costs: creating and managing collections and downloading the member IDs are free. An import costs the same credits as paging through the results yourself (one list request per 200 results). Imports run at about 200 results per second (measured November 2026), so a collection of the maximum 1,000,000 members takes over an hour. Interrupting pro_collection() does not stop the import on the server; running it again waits for it.

Several queries in one collection: OpenAlex imports a query of at most 20,000 characters of OQL (about 600 to 800 DOIs); a longer single query stops before anything is created. For long ID or

DOI lists, split them into chunks, e.g. `pro_query(doi = dois, chunk_limit = 200)`, and pass the list with `name`. The queries are then not imported: they are fetched with `pro_fetch()` (`select = "id"`, in parallel with `workers`) and the IDs are added to the collection (`member_ids`, 10,000 per request, free). This costs the same credits as imports and was about five times faster with `workers = 8` (3,000 DOIs: 8.5 s instead of 45 to 55 s, November 2026). Fields in `select` beyond `id` are fetched in the same pass, from the queries themselves, so the records do not depend on the collection filter limit. An interrupted run resumes the fetch and syncs the members again (syncing is idempotent). A changed query re-creates the collection unless `refresh = TRUE`, which keeps it, fetches everything again and syncs the members.

Limits: 1,000,000 members per collection, 500 collections per account, names up to 100 characters and unique per account. A collection with more than 300,000 members (100,000 for authors) cannot be used as a filter, so its records cannot be downloaded with `select`; its member IDs can. Queries with `group_by` or `sample` cannot be imported.

Core and expansion works: a collection holds all its members, and the member IDs in `members/` include expansion works (DataCite, repositories). A *filter* by a works collection returns only its core works unless `corpus = "all"` is set, so the records in `records/` are fetched with `corpus = "all"`, and later queries should set it too: `pro_query(collection = id, options = list(corpus = "all"))`. Whether expansion works get into the collection depends on the query that fills it (`options = list(corpus = "all")` in `pro_query()`).

API key: collections need a *personal* OpenAlex API key; organisation keys and some older keys are refused (HTTP 401) even when they work for queries.

Value

Invisibly, the path of the `.collection` folder, or a named character vector of paths for a list of queries without `name`.

See Also

`pro_fetch()`, `pro_query()`, `vignette("collections")`.

Examples

```
## Not run:
q <- pro_query(search = "coral bleaching", publication_year = 2024)
f <- pro_collection(q, name = "coral bleaching 2024", project_folder = "proj")
id <- jsonlite::read_json(file.path(f, "_collection.json"))$collection_id

# Later: records of the collection that are open access. corpus = "all":
# without it, the filter returns only the core works
pro_fetch(pro_query(collection = id, is_oa = TRUE, options = list(corpus = "all")),
          project_folder = "oa")

# Let anyone with the link or ID use it
pro_collection(q, name = "coral bleaching 2024", project_folder = "proj",
               access = "shared_by_link")

# Add titles to the download (re-downloads the records)
pro_collection(q, name = "coral bleaching 2024", project_folder = "proj",
               select = c("id", "title"))
```

```
# A long DOI list in one collection: the chunks are fetched in parallel
# and their IDs added as members
pro_collection(pro_query(doi = dois, chunk_limit = 200), name = "my dois",
               project_folder = "proj", workers = 8)

## End(Not run)
```

`pro_compact_parquet` *Compact a Parquet dataset into fewer, larger files*

Description

`pro_request_parquet()` writes one Parquet file per API page (up to 200 records). That keeps the conversion parallel and resumable, but a large query becomes thousands of small files, which are slow to read. `pro_compact_parquet()` combines them into files of up to `rows` records. Records, columns and their order do not change.

Usage

```
pro_compact_parquet(parquet_dir, rows, verbose = TRUE)
```

Arguments

`parquet_dir`	Output directory of `pro_request_parquet()`, or the `parquet` folder of a `pro_fetch()` project.
`rows`	Maximum number of records per compacted file, e.g. `250000`.
`verbose`	Logical. Report what was compacted. Default `TRUE`.

Details

Measured for 1.1 million works (5,460 page files): compaction took about 20 seconds; afterwards aggregates and lookups by ID ran 35 to 50 times faster, reading full records with DuckDB 4 times faster, and the dataset was 40% smaller. Files of 50,000 to 1 million records made no difference to read speed; `rows = 250000` is a good choice (about 0.25 GB of full works records; newer works with abstracts and long reference lists are larger).

- Each query's partition (`query=<name>/`) is compacted on its own; its `_query.json` and `_query_oql.*` files stay.
- A compacted file holds whole consecutive pages, in download order, and is named after them: pages 1 to 1250 become `results_pages_1-1250.parquet`. A page with more than `rows` records becomes a file of its own. The `page` column is kept.
- Files are compacted one at a time: each is written to a `.part` file, its record count checked, renamed, and only then are its pages deleted. If interrupted, run it again: it completes what was started.
- Files compacted before are left as they are; only page files are combined.

- If a partition cannot be compacted (for example because its pages have conflicting column types), a warning says so and its page files stay: the dataset is complete either way.
- Indexes built by `openalexSnapshot::build_corpus_index()` or `build_doi_index()` point to files and row numbers, and would be wrong after compaction. If such an index exists next to `parquet_dir`, the function stops: delete the index, compact, and build it again.

Value

`parquet_dir` (invisibly).

See Also

The `compact_parquet` argument of `pro_request_parquet()` and `pro_fetch()`, which compacts right after the conversion.

Examples

```
## Not run:
pro_compact_parquet("my_project/parquet", rows = 250000)

## End(Not run)
```

---

pro_download_content *Download full-text PDFs or TEI XML for OpenAlex works*

Description

Downloads full-text content from the OpenAlex content endpoint (`content.openalex.org`) for a vector of work IDs. One file is written per ID. Downloads can be parallelised via the `workers` argument.

Usage

```
pro_download_content(
  ids,
  format = c("pdf", "grobid-xml"),
  output = ".",
  workers = 1L,
  api_key = pro_api_key(),
  endpoint = "https://content.openalex.org"
)
```

Arguments

ids	Character vector of OpenAlex work IDs (e.g. "W2741809807") or full OpenAlex URLs ("https://openalex.org/W2741809807"). Full URLs are normalised automatically.
format	File format to download. One of "pdf" (default) or "grobid-xml" (TEI XML).
output	Directory to save downloaded files into. Defaults to the current working directory. Created if it does not exist.
workers	Number of parallel download workers. Defaults to 1 (sequential). Set higher for faster batch downloads, subject to the content endpoint's rate limits.
api_key	OpenAlex API key (character string) or 'NULL'. Defaults to pro_api_key() (option or the openalexPro.api_key environment variable). If 'NULL' or '""', requests are sent without an API key.
endpoint	Base URL of the content endpoint. Defaults to "https://content.openalex.org".

Value

A data frame with one row per ID and columns:

id The (normalised) work ID.

file Full path to the saved file, or NA if not downloaded.

status One of "ok", "not_found" (HTTP 404), or "error".

message Error message, or NA on success.

Costs

Content downloads cost **$0.01 per file** - 10x the cost of a metadata search query. Use has_content.pdf:true or has_content.grobid-xml:true as filter arguments to pro_query() to discover which works have downloadable content before downloading.

Formats

"pdf" Full-text PDF (~60 million files available).

"grobid-xml" Machine-readable TEI XML parsed by Grobid (~43 million files). Suitable for structured text extraction.

Licensing

PDFs and XMLs retain their original copyright. OpenAlex does not grant additional rights. Check the best_oa_location.license field of each work for the applicable licence.

Examples

```
## Not run:
# Download a single PDF
result <- pro_download_content(
  ids    = "W2741809807",
  format = "pdf",
```

```
  output = tempdir()
)

# Find works with PDFs available, then download them
urls <- pro_query(
  entity          = "works",
  has_content.pdf = TRUE,
  from_publication_date = "2023-01-01",
  options = list(per_page = 10)
)
works <- pro_request(urls, output = tempdir())
# ... extract IDs from works data, then:
result <- pro_download_content(ids = work_ids, format = "pdf", workers = 4)

# XPAC works: discover via pro_query() with include_xpac = TRUE, then download
# (pro_download_content() works with any valid OpenAlex ID, including XPAC IDs)
urls_xpac <- pro_query(
  entity          = "works",
  has_content.pdf = TRUE,
  from_publication_date = "2023-01-01",
  options = list(include_xpac = TRUE, per_page = 10)
)
works_xpac <- pro_request(urls_xpac, output = tempdir())
# ... extract IDs from works_xpac data, then:
result_xpac <- pro_download_content(ids = xpac_ids, format = "pdf", workers = 4)

## End(Not run)
```

`pro_fetch` *Fetch and convert OpenAlex data to Parquet*

Description

Convenience wrapper that downloads records from OpenAlex via `pro_request()` and converts them directly to an Apache Parquet dataset via `pro_request_parquet()`. No intermediate JSONL files are written.

Usage

```
pro_fetch(
  query_url,
  pages = 10000,
  project_folder = NULL,
  overwrite = FALSE,
  api_key = pro_api_key(),
  delete_input = TRUE,
  workers = 1,
  download_workers = workers,
```

```
  verbose = FALSE,
  progress = TRUE,
  enrich = TRUE,
  count_only,
  error_log = NULL,
  resume = FALSE,
  via = c("oqo", "url"),
  inconsistency_on_resume = c("error", "warning", "ignore", "oldest", "newest"),
  compact_parquet = NULL,
  hive_names = NULL
)
```

Arguments

query_url	The URL of the API query or a list of URLs returned from pro_query().
pages	The maximum number of pages (of 200 records) downloaded per query; for a list, the limit applies to each leaf query. 100000 pages are 20,000,000 records; for more, use the snapshot instead. If NULL, all pages will be downloaded.
project_folder	Directory where intermediate (json) and final (parquet) results are stored. If it does not exist, it is created. If NULL, a temporary directory is created.
overwrite	Logical. If TRUE, the json and parquet subdirectories are deleted from project_folder before the pipeline starts. If FALSE (the default) and any of those subdirectories already exist, the function stops with an error.
api_key	Character string API key or NULL. Defaults to pro_api_key(). If NULL or "", requests are sent without an API key (subject to OpenAlex's unauthenticated limits).
delete_input	Logical. If TRUE (the default), the json subfolder is deleted after successful conversion to Parquet.
workers	Number of parallel workers for the conversion to Parquet (CPU and memory bound), and by default also for the download. Default 1.
download_workers	Number of parallel workers for the download, used when query_url is a list. Defaults to workers. Downloads are limited by the API rather than the machine, so a few (e.g. 2–4) are usually enough, while the conversion can use more.
verbose	Logical indicating whether to show verbose messages.
progress	Logical indicating whether to show a progress bar. Default TRUE.
enrich	Logical. When TRUE (the default) and the inferred schema contains abstract_inverted_index / authorships / publication_year, add abstract and citation computed columns.
count_only	Do not use it here. The function will abort if set to TRUE and give a warning if FALSE.
error_log	location of error log of API calls. (default: NULL (none)).
resume	Logical. When TRUE, keep existing json and parquet subfolders and complete an interrupted run: only leaf queries that did not finish are downloaded again, and only missing Parquet files are converted. A finished run is returned unchanged. overwrite is ignored. Default FALSE.

`via` How each query is sent. `"oqo"` (default): translated once (free) into an OpenAlex Query Object and sent via POST, with the same results as `"url"`, which sends it as a GET request with the query in the URL, as built by `pro_query()`. It has no URL length limit, accepts more than 100 values per filter (so `pro_query(..., chunk_limit = 200)` fetches a DOI or ID list in about one page per chunk), checks the query before any credits are spent, and records the exact query in `_query.json`. Single records (`works/W123`) are always fetched via GET.

`inconsistency_on_resume`
What to do when a query that was resumed in the middle turns out inconsistent, because OpenAlex's data changed between the sessions: records that appear twice (found exactly, by work ID) or missing records (only as a net difference to OpenAlex's current count – a record lost and another added cancel out). `"error"` (default) stops; nothing is lost, and a rerun with `resume = TRUE` and another value applies that choice without downloading again. `"warning"` keeps both copies and warns, `"ignore"` keeps both silently, `"oldest"` / `"newest"` keep the first / last downloaded copy (removing the other from its page file) and warn about missing records.

`compact_parquet`
`NULL` (default) keeps one Parquet file per API page. A number, e.g. `250000`, combines each query's page files into files of up to that many records once every file has converted – much faster to read. See `pro_compact_parquet()`.

`hive_names` Character vector of Hive partition keys, one per nesting level of a list of queries: `c("topic", "year")` writes `topic=<name>/year=<name>/`. `NULL` (default) uses `query` for the first level and `query_l2`, `query_l3`, ... below it; `NA` keeps the default of that level. A query named `key=value` (e.g. `"year=2020"`) supplies its own key, unless `hive_names` names one for that level. Keys must be unique and made of letters, digits and `_`. Use the same value when resuming, or the files are converted again.

Details

The function

- downloads records from OpenAlex via `pro_request()` into a `"json"` subfolder of `project_folder`, and
- converts the JSON files to an Apache Parquet dataset via `pro_request_parquet()` into a `"parquet"` subfolder.

This function assumes `count_only == FALSE`

Value

Invisibly, the normalised path of the `parquet` subfolder inside `project_folder`.

See Also

`pro_request()` for the download step, `pro_request_parquet()` for the conversion step, `pro_compact_parquet()` to compact an existing dataset.

Examples

```
## Not run:
# Combine each query's page files into files of up to 250,000 records,
# which are much faster to read
pro_fetch(
  pro_query(search = "coral reef", from_publication_date = "2020-01-01"),
  project_folder = "coral_reef",
  compact_parquet = 250000
)

## End(Not run)
```

---

pro_query *Build an OpenAlex request (httr2)*

Description

Construct an `httr2` request for the OpenAlex API. All filters must be supplied as named `...` arguments (e.g., `from_publication_date = "2020-01-01"`).

Usage

```
pro_query(
  entity = c("works", "authors", "venues", "institutions", "concepts", "publishers",
    "funders", "sources", "topics", "locations"),
  id = NULL,
  doi = NULL,
  search = NULL,
  search.exact = NULL,
  search.semantic = NULL,
  group_by = NULL,
  select = NULL,
  options = NULL,
  endpoint = "https://api.openalex.org",
  chunk_limit = 50L,
  oql = NULL,
  oqo = NULL,
  ...
)
```

Arguments

`entity`	Character; one of `"works"`, `"authors"`, `"institutions"`, `"publishers"`, `"funders"`, `"sources"`, `"topics"`, `"locations"`. `"venues"` (now `"sources"`) and `"concepts"` (now `"topics"`) are deprecated by OpenAlex and kept only for compatibility.

id	Optional ID or vector of IDs (e.g., "W1775749144"). If a single ID is provided, fetches one entity directly. If multiple IDs are provided, they are automatically moved into the ids.openalex filter.
doi	Optional DOI or vector of DOIs (e.g., "10.1038/s41586-021-03819-2"). Values are moved into the doi filter and automatically chunked into separate requests when the number of DOIs exceeds chunk_limit.
search	Optional full-text search string. Applies stemming and stop-word removal. Supports boolean operators (AND, OR, NOT in uppercase), quoted phrases ("exact phrase") and proximity ("word1 word2"~N). Wildcards are rejected by the API (HTTP 400): use search.exact. Replaces the deprecated filter = field.search:keyword syntax.
search.exact	Optional full-text search without stemming or stop-word removal. Supports the boolean/phrase/proximity syntax of search, plus wildcards and fuzzy matching. Use when you need to match exact word forms (e.g. "surgery" should not match "surgical").
search.semantic	Optional semantic (AI-powered) search string. Uses embeddings to match conceptual meaning rather than exact keywords. Limited to 1 request per second and returns at most 50 results per query.
group_by	Optional field to group by (facets), e.g. "type".
select	Optional character vector of fields to return.
options	Optional named list of additional query parameters (e.g., list(sort = "cited_by_count:desc", sample = 100, seed = 42)). pro_request() does the paging, so page and cursor given here are replaced (with a warning); per_page is kept (1–200, default 200). A sample (up to 10,000) is paged by page number and needs a seed when it spans several pages; pro_request() picks and reports one if none is given.
endpoint	Base API URL. Defaults to "https://api.openalex.org".
chunk_limit	Number of DOIS or ids per chunk if chunked. Default: 50
oql	Optional OQL (OpenAlex Query Language) query string, e.g. "works where year is (2020) and type is (article)". A character vector gives a named list of URLs (names from names(oql), else query_1, query_2, ...), one per query. See section *OQL and OQO*.
oqo	Optional OQO (OpenAlex Query Object): a nested list, or a JSON string. A list of OQOs (lists or JSON strings), or a character vector of JSON strings, gives a named list of URLs as for oql. See section *OQL and OQO*.
...	Filters as named arguments. Values may be scalars or vectors (vectors are collapsed with "|" to express OR). Arguments of the functions that send the query, such as via or workers, belong to [pro_fetch()], [pro_request()] or [pro_count()], not here.

Details

Filter names are validated against opt_filter_names(entity = entity) and select fields against opt_select_fields(entity = entity), i.e. against the names OpenAlex accepts for the queried entity.

If multiple more then 50 ‘doi‘ or openalex ‘id‘s are provided, the request is automatically split into chunks of 50 and a named list of URLs is returned.

Value

An individual URL or a list of URLs.

Search syntax

All three search parameters (`search`, `search.exact`, `search.semantic`) accept a query string. For `search` and `search.exact`, the following syntax is supported:

- Boolean: `biodiversity AND finance`, `climate OR weather`, `ocean NOT pollution` (operators must be uppercase); group with parentheses. Words without an operator between them are combined with `AND`.
- Exact phrase: `"biodiversity finance"` (double quotes).
- Proximity: `"biodiversity finance"~5` (words within 5 positions), or two phrases near each other: `"machine learning"~5~"neural network"`.

Only with `search.exact` (the stemmed `search` rejects wildcards with HTTP 400, and stemming distorts fuzzy terms):

- Wildcard: `microbio*` (zero or more characters), `organi?ation` (exactly one). At least 3 characters before the wildcard; no leading wildcard.
- Fuzzy: `biodiversty~1` (up to N character edits, N = 0–2; at least 3 characters before the `~`).

See https://help.openalex.org/api/searching/. `search.semantic` does not use keyword syntax; pass a natural-language phrase or even a full abstract. It returns at most 50 results per call.

OQL and OQO

OpenAlex's query language (`oql`) and its JSON form (`oqo`) express queries the classic filter syntax cannot, such as nested `and` / `or` conditions. The entity, all conditions, grouping (`group by`), `sample` and `corpus` belong inside the query, so `entity`, `id`, `doi`, the search arguments, `group_by` and filters in `...` cannot be combined with it – OpenAlex would silently ignore them. `select` and `options` (e.g. `sort`) are passed alongside. The URL targets the API root (`https://api.openalex.org/?oql=...`) and works with `pro_request()`, `pro_count()` and `pro_fetch()` as usual. An invalid query fails at request time with OpenAlex's parse error. See https://help.openalex.org/api/oql/.

Deprecated search filters

Filter arguments with a `.search` suffix (e.g. `title_and_abstract.search = "biodiversity"`) are deprecated by the OpenAlex API. They still work but emit a warning. Use the `search`, `search.exact`, or `search.semantic` parameters instead. See https://help.openalex.org/api/searching/ for details.

Examples

```
## Not run:

url <- pro_query(
  entity = "works",
  search = "biodiversity",
  from_publication_date = "2020-01-01",
```

```
  language = c("en","de"),
  select = c("id","title","publication_year"),
  options = list(per_page = 5)
)

# OQL: nested conditions the filter syntax cannot express
url <- pro_query(
  oql = "works where year >= (2020) and (title has (coral) or title has (reef))"
)

# OQO: the same kind of query as a nested list
url <- pro_query(oqo = list(
  get_rows = "works",
  filter_rows = list(list(column_id = "publication_year", value = 2020))
))

## End(Not run)
```

`pro_rate_limit_status` *Check OpenAlex rate limit status*

Description

Queries the OpenAlex rate-limit endpoint and returns current API usage and remaining budget as a parsed list.

Usage

```
pro_rate_limit_status(api_key = pro_api_key(), verbose = TRUE)
```

Arguments

`api_key`	API key (character string) or 'NULL'. Defaults to `pro_api_key()`. If 'NULL' or '""', this function returns `FALSE` with an informational message.
`verbose`	Logical. If `TRUE` (default), prints rate limit info via `message()`.

Value

Invisibly, the parsed JSON list with all rate limit fields; `FALSE` if the API key is missing or invalid; or `NULL` if the request failed due to a network error.

pro_request *Fetch works from OpenAlex*

Description

All returned values from OpenAlex will be saved as json files in the `output` directory and the return value is the directory of the json files.

Usage

```
pro_request(
  query_url,
  pages = 1e+05,
  output = NULL,
  overwrite = FALSE,
  api_key = pro_api_key(),
  workers = 1,
  verbose = FALSE,
  progress = TRUE,
  count_only = FALSE,
  error_log = NULL,
  resume = FALSE,
  via = c("oqo", "url"),
  inconsistency_on_resume = c("error", "warning", "ignore", "oldest", "newest")
)
```

Arguments

`query_url`	The URL of the API query or a list of URLs returned from `pro_query()`.
`pages`	The maximum number of pages (of 200 records) downloaded per query; for a list, the limit applies to each leaf query. 100000 pages are 20,000,000 records; for more, use the snapshot instead. If `NULL`, all pages will be downloaded.
`output`	directory where the JSON files are saved. Required (the default `NULL` stops with an error) unless `count_only = TRUE`.
`overwrite`	Logical. If `TRUE`, `output` will be deleted before downloading. For a list `query_url`, the entire top-level `output` directory is removed upfront. If `FALSE` (the default) and `output` already exists, the function stops with an error.
`api_key`	Character string API key or `NULL`. Defaults to `pro_api_key()`. If `NULL` or `""`, requests are sent without an API key (subject to OpenAlex's unauthenticated limits).
`workers`	Number of parallel workers to use if `query_url` is a list. Defaults to 1.
`verbose`	Logical indicating whether to show verbose messages.
`progress`	Logical indicating whether to show a progress bar. Default `TRUE`.
`count_only`	return count only as a data.frame.

`error_log` location of error log of API calls. (default: `NULL` (none)).

`resume` Logical. When `TRUE`, keep an existing `output` and download only what is missing: finished queries are skipped (detected by the `00_in.progress` sentinel each query directory carries while it is being written), and an interrupted query continues after its last saved page, using the cursor stored in its `_paging.json` (the message says how old it is). If that state cannot be trusted, or for samples and semantic search, the query is fetched again from page 1. A single URL is treated as one query. Default `FALSE`.

`via` How each query is sent. `"oqo"` (default): translated once (free) into an OpenAlex Query Object and sent via POST, with the same results as `"url"`, which sends it as a GET request with the query in the URL, as built by `pro_query()`. It has no URL length limit, accepts more than 100 values per filter (so `pro_query(..., chunk_limit = 200)` fetches a DOI or ID list in about one page per chunk), checks the query before any credits are spent, and records the exact query in `_query.json`. Single records (`works/W123`) are always fetched via GET.

`inconsistency_on_resume`
What to do when a query that was resumed in the middle turns out inconsistent, because OpenAlex's data changed between the sessions: records that appear twice (found exactly, by work ID) or missing records (only as a net difference to OpenAlex's current count – a record lost and another added cancel out). `"error"` (default) stops; nothing is lost, and a rerun with `resume = TRUE` and another value applies that choice without downloading again. `"warning"` keeps both copies and warns, `"ignore"` keeps both silently, `"oldest"` / `"newest"` keep the first / last downloaded copy (removing the other from its page file) and warn about missing records.

Details

If query_url is a list, the function is called for each element of the list in parallel using a maximum of `workers` parallel R sessions. The results from the individual URLs in the list are returned in a folder named after the names of the list elements in the `output` folder.

When starting the download, a file `00_in.progress` which is deleted upon completion.

If one query of a list fails (e.g. HTTP 404 for an ID that does not exist), the other queries still complete. At the end, one error names each failed query with OpenAlex's reason. The failed queries keep their `00_in.progress`, so `resume = TRUE` fetches only them again.

Pages are fetched with cursor paging. A `per_page` in the query is kept; `page` and `cursor` are replaced, with a warning. A `sample` is fetched by page number, as OpenAlex ends cursor paging of a sample after the first page; if it spans several pages and has no `seed`, one is chosen and reported. Semantic search (`search.semantic`) is fetched as its single page of at most 50 results.

Each query directory also gets a record of the query: `_query.json` (the URL, how it was sent, the query as OpenAlex ran it as OQL and OQO, records, requests, credits, cost, time), and for papers `_query_oql.txt` and a formatted, self-contained `_query_oql.html` with the query in OpenAlex's readable query language (OQL). The OQL is translated back and compared with the query that ran; the files say whether it reproduces it exactly. Single records get only `_query.json`. For a list of queries, `_queries_oql.txt` and `_queries_oql.html` at the top of `output` summarise all queries (records, exactness, OQL), e.g. for a paper's supplement.

Value

If `count_only` is `FALSE` (the default) the complete path to the expanded and normalized `output`. If `count_only` is `TRUE`, a data.frame with metadata about the query (count, db_response_time_ms, page, per_page, error). When `query_url` is a list, an additional `query` column identifies each query.

`pro_request_jsonl` *Convert JSON files to jsonl files*

Description

The function takes a directory of JSON files as written from a call to `pro_request(...)` and is preparing the json files to be processed further using DuckDB by converting them to `jsonl` files. The subfolders in `input_json` are preserved in `output`, i.e. results of a list of initial queries passed to `pro_request()` are maintained.

Usage

```
pro_request_jsonl(
  input_json = NULL,
  output = NULL,
  add_columns = list(),
  overwrite = FALSE,
  verbose = TRUE,
  progress = TRUE,
  delete_input = FALSE,
  workers = 1
)
```

Arguments

`input_json`	The directory of JSON files returned from `pro_request(..., json_dir = "FOLDER")`.
`output`	output directory for the jsonl files as created by calls to 'jq_execute().
`add_columns`	List of additional fields to be added to the output. They nave to be provided as a named list, e./g. `list(column_1 = "value_1", column_2 = 2)`. Only Scalar values are supported.
`overwrite`	Logical indicating whether to overwrite `output`.
`verbose`	Logical indicating whether to show a verbose information. Defaults to `TRUE`
`progress`	Logical indicating whether to show a progress bar. Default `TRUE`.
`delete_input`	Determines if the `input_json` should be deleted afterwards. Defaults to `FALSE`.
`workers`	Number of parallel workers to use. Defaults to 1.

Details

See jq_execute or the vignette("jq", package = "openalexPro") for more information on the conversion of the JSON files. The folder/filename is converted to a value named page As an example:

1. the subfolder in the output folder is called Chunk_1
2. the page othe json file represents is 2
3. The resulting cvalus for page will be Chunk_1_2

When starting the conversion, a file 00_in.progress which is deleted upon completion.

The function uses DuckDB to read the JSON files and to create the Apache Parquet files. The function creates a DuckDB connection in memory and readsds the JSON files into DuckDB when needed. Then it creates a SQL query to convert the JSON files to Apache Parquet files and to copy the result to the specified directory.

Value

The function does returns the output invisibly.

Examples

```
## Not run:
  source_to_parquet(
  input_json = "json",
  source_type = "snapshot",
  output = "parquet"
)
## End(Not run)
```

---

pro_request_jsonl_parquet

Convert JSON files to Apache Parquet files

Description

The function takes a directory of JSONL files as written from a call to pro_request_jsonl(...) and converts each file individually to a Parquet file. The subfolder structure from the input is preserved in the output, so files in Chunk_1/ will be written to Chunk_1/ in the output directory.

Usage

```
pro_request_jsonl_parquet(
  input_jsonl = NULL,
  output = NULL,
  overwrite = FALSE,
  verbose = TRUE,
  delete_input = FALSE,
```

```
  sample_size = 1000,
  workers = NULL
)
```

Arguments

input_jsonl	The directory of JSON files returned from pro_request(..., json_dir = "FOLDER").
output	output directory for the parquet dataset; default: temporary directory.
overwrite	Logical indicating whether to overwrite output.
verbose	Logical indicating whether to show verbose information. Defaults to TRUE
delete_input	Determines if the input_jsonl should be deleted afterwards. Defaults to FALSE.
sample_size	Number of records to sample from each file when inferring the unified schema. Higher values give more accurate schema inference but use more memory. Default is 1000. Set to -1 to read all records (may be slow for large files).
workers	Number of parallel workers for file conversion via future.apply::future_lapply(). Default is NULL (sequential processing).

Details

The page column (added by pro_request_jsonl()) is preserved as a regular column in the Parquet data.

When starting the conversion, a file 00_in.progress is created which is deleted upon completion.

The function uses DuckDB to read the JSON files and to create the Apache Parquet files. Each JSON file is converted individually using its own DuckDB connection, which enables parallel processing via future.apply::future_lapply().

To ensure consistent schemas across all Parquet files, the function first infers a unified schema by sampling records from all JSONL files. This prevents type mismatches (e.g., a column being struct in one file but string in another) that would cause errors when reading the combined Parquet dataset.

Value

The function returns the output path invisibly.

---

pro_request_parquet *Convert JSON files from pro_request() directly to Apache Parquet*

Description

Single-step replacement for the two-step pro_request_jsonl() + pro_request_jsonl_parquet() pipeline. Reads the JSON files written by pro_request() and converts each one to a Parquet file using DuckDB, with no intermediate JSONL on disk.

Usage

```
pro_request_parquet(
  input_json = NULL,
  output = NULL,
  add_columns = list(),
  overwrite = FALSE,
  verbose = TRUE,
  progress = TRUE,
  delete_input = FALSE,
  sample_size = 1000,
  workers = NULL,
  enrich = TRUE,
  schema = "auto",
  resume = FALSE,
  on_error = c("error", "warn", "ignore"),
  memory_limit = NULL,
  retry_memory_limit = NULL,
  compact_parquet = NULL,
  hive_names = NULL
)
```

Arguments

input_json	Directory of JSON files returned by pro_request().
output	Output directory for the Parquet dataset.
add_columns	Named list of scalar constant columns to embed in every output record (e.g. list(query = "my_filter")). Values are embedded as SQL string literals; only character scalars are supported.
overwrite	Logical. Overwrite output if it already exists. Default FALSE.
verbose	Logical. Show progress messages. Default TRUE.
progress	Logical. Show a progress bar. Default TRUE.
delete_input	Logical. Delete input_json after a successful conversion. If any file failed to convert, input_json is kept so the run can be completed with resume = TRUE. Default FALSE.
sample_size	Integer. Number of records per file passed to DuckDB's sample_size option during schema inference. Use -1 to read all records (accurate but slow for large files). Default 1000.
workers	Integer. Number of parallel workers. NULL or 1 runs sequentially. Default NULL.
enrich	Logical. When TRUE (the default) and the inferred schema contains abstract_inverted_index / authorships / publication_year, add abstract and citation computed columns.
schema	Controls use of a pre-built baseline schema for type resolution. Possible values: "auto" **(default)** Auto-detect the OpenAlex entity type from the inferred columns, then load the matching schema from the user cache (populated by oa_schema(update = TRUE)) or the schemas bundled with the package. For each column where

DuckDB runtime inference produced the ambiguous `JSON` fallback type, the baseline type is used instead. Falls back silently to runtime-only inference when the entity cannot be detected or no schema is found.

`"none"` **or** `NULL` Skip the baseline entirely; behaviour is identical to package versions before this feature was added.

A file path Path to a CSV with columns `col_name` / `col_type`. Used directly as the baseline.

A directory path Auto-detect entity, then look for `<entity>.csv` inside that directory.

`resume` Logical. When `TRUE`, keep an existing `output` and convert only the files that are not already there. Safe because each file is written to a `.part` sidecar and renamed only on success, so a present file is always complete. Default `FALSE`.

`on_error` How to handle files that still fail after the sequential retry: `"error"` (the default) stops and names them, `"warn"` warns and returns, `"ignore"` is silent.

Before 0.12.0 failures were only `message()`d when `verbose`, and the run reported success regardless – so a page that failed to convert went silently missing from the corpus. `"warn"` is the closest to that old behaviour, but visible.

`memory_limit` DuckDB `memory_limit` for each conversion worker. `NULL` (default) derives one from physical RAM and `workers`, rather than letting every worker claim DuckDB's default of 80% of the machine.

`retry_memory_limit`

`memory_limit` for the sequential retry pass, where no workers compete. `NULL` (default) uses the whole budget.

`compact_parquet`

`NULL` (default) keeps one Parquet file per API page. A number, e.g. `250000`, combines each query's page files into files of up to that many records once every file has converted – much faster to read. See `pro_compact_parquet()`.

`hive_names` Character vector of Hive partition keys, one per nesting level of a list of queries: `c("topic", "year")` writes `topic=<name>/year=<name>/`. `NULL` (default) uses `query` for the first level and `query_l2`, `query_l3`, ... below it; `NA` keeps the default of that level. A query named `key=value` (e.g. `"year=2020"`) supplies its own key, unless `hive_names` names one for that level. Keys must be unique and made of letters, digits and `_`. Use the same value when resuming, or the files are converted again.

Details

For works entities the function detects the presence of `abstract_inverted_index`, `authorships`, and `publication_year` in the inferred schema and, when `enrich = TRUE` (the default), adds two computed columns:

- `abstract` - plain text reconstructed from `abstract_inverted_index`.
- `citation` - `"Author (year)"` / `"A & B (year)"` / `"A et al. (year)"`.

Value

Output directory path (invisibly).

File format

pro_request() writes one JSON file per API page. For paginated queries each file has the structure {"results": [...], "meta": {...}}. For group-by queries the array field is "group_by". For single-record lookups the file is a bare JSON object. All three formats are handled automatically.

Output layout

The subdirectory structure of input_json is preserved, with hive-partition naming (query=<name>/, query_l2=<name>/, ...) so that Arrow/DuckDB can read the result as a partitioned dataset. A page column is added to each record with a value derived from the source filename (or subdirectory for multi-query inputs).

With compact_parquet, the page files of each query are combined into files of up to that many records after the conversion (see pro_compact_parquet()).

See Also

pro_request() to download the JSON files, pro_compact_parquet() to compact an existing dataset, pro_request_jsonl() and pro_request_jsonl_parquet() for the older two-step pipeline (now deprecated).

Examples

```
## Not run:
pro_request_parquet("my_project/json", "my_project/parquet",
                    compact_parquet = 250000)

## End(Not run)
```

---

pro_validate_credentials

Validate OpenAlex credentials

Description

Makes a minimal API request to verify that the api_key is valid.

Usage

```
pro_validate_credentials(api_key = pro_api_key(), show_credentials = FALSE)
```

Arguments

api_key	API key to validate (character string) or ‘NULL‘.
show_credentials	shows the api_key using ‘message()‘. USE WITH CAUTION!

Value

TRUE if credentials work, FALSE otherwise

---

read_corpus *Read corpus from Parquet Dataset*

Description

This function reads a corpus in Apache Parquet format and returns an `ArrowObject` representing the corpus which can be fed into a `dplyr` pipeline or a `tibble` which contains all the data.

Usage

```
read_corpus(corpus, return_data = FALSE)
```

Arguments

`corpus`	The directory of the Parquet files.
`return_data`	Logical indicating whether to return an `ArrowObject` representing the corpus (default) or a `tibble` containing the whole corpus shou,d be returned.

Value

An `ArrowObject` representing the corpus or a `tibble`.

---

sample_parquet_n *Sample rows from Parquet files using DuckDB reservoir sampling*

Description

Draw a uniform random sample of `n` rows from one or more Parquet files using DuckDB's SQL `USING SAMPLE reservoir(n ROWS)` clause. The sampling is performed entirely inside DuckDB, so the full dataset is never loaded into R.

This is well-suited for large Parquet corpora (e.g. OpenAlex works) where you want a random subset of rows without materialising the whole table.

Usage

```
sample_parquet_n(path, n, seed = NULL, con = NULL, select = NULL)
```

Arguments

path	Character scalar. Path or glob pointing to one or more Parquet files, as understood by DuckDB's parquet_scan() table function. For example, "spc_corpus/output/chapter_3/co
n	Integer scalar. Number of rows to sample. If n is larger than the total number of rows in the dataset, DuckDB returns all rows.
seed	Optional integer scalar. If supplied, a REPEATABLE(seed) clause is added to the DuckDB query so that repeated calls with the same input data and seed return the same sample. If NULL (default), the sample is not forced to be reproducible at the DuckDB level.
con	Optional DBIConnection to an existing DuckDB database. If NULL (the default), the function creates a temporary in-memory DuckDB instance, uses it for the query, and shuts it down before returning. If a connection is supplied, it is left open and not modified beyond running the sampling query.
select	Optional character vector of column names to return. If NULL (default), all columns are returned (equivalent to SELECT *). Column names are passed through DBI::dbQuoteIdentifier() to safely handle special characters. If any requested column does not exist in the Parquet schema, DuckDB will raise an error.

Details

The function delegates to the following SQL pattern (simplified):

```
SELECT [columns]
FROM parquet_scan('path/to/files/*.parquet')
USING SAMPLE reservoir(n ROWS)
[REPEATABLE (seed)]
```

Using reservoir(n ROWS) gives an exact uniform sample of size n from all rows in the dataset (unless n exceeds the total row count, in which case all rows are returned).

Note that the path argument is passed directly to DuckDB's parquet_scan() function, so you can use:

- A single Parquet file:
  - "works.parquet"
- A glob for many files:
  - "works/*.parquet"
- A directory, depending on your DuckDB version/configuration.

When con is NULL, the function creates an in-memory DuckDB database. If you want to reuse the same DuckDB instance for multiple queries (for performance reasons or to control pragmas), you can create a DuckDB connection yourself and pass it via con.

Value

A data.frame with up to n rows, containing a uniform random sample from the union of all Parquet files matched by path, restricted to the columns specified in select (or all columns if select is NULL).

Examples

```
## Not run:
# Sample 1,000 rows from a directory of Parquet files
sample_df <- sample_parquet_duckdb(
  path = "spc_corpus/output/chapter_3/corpus/*.parquet",
  n = 10000L,
  seed = 1234
)

# Sample only a subset of columns
sample_df_small <- sample_parquet_duckdb(
  path = "spc_corpus/output/chapter_3/corpus/*.parquet",
  n = 10000L,
  seed = 1234,
  select = c("id", "doi", "citation", "author_abbr", "display_name", "ab")
)

# Reuse a DuckDB connection for multiple samples
con <- DBI::dbConnect(duckdb::duckdb())
on.exit(DBI::dbDisconnect(con, shutdown = TRUE), add = TRUE)

s1 <- sample_parquet_duckdb(
  path = "openalex_works/*.parquet",
  n = 500L,
  seed = 42,
  con = con
)

s2 <- sample_parquet_duckdb(
  path = "openalex_works/*.parquet",
  n = 500L,
  seed = 777,
  con = con
)

## End(Not run)
```

Index

compatibility_report, 3

DBI::dbQuoteIdentifier(), *36*
DBIConnection, *36*

extract_doi, 4

future.apply::future_lapply(), *31*

id_block, 5
infer_json_schema, 6

jq_execute, 8, *30*

oa_normalize_duckdb_type, 9
oa_schema, 9, *32*
oa_works_abstract_sql, 10
oa_works_citation_sql, 11
opt_api_key, 11
opt_filter_names, 12
opt_select_fields, 12
opt_select_fields(), *14*
options, *13*

pro_api_key, 13
pro_collection, 14
pro_compact_parquet, 17
pro_compact_parquet(), *22*, *33*, *34*
pro_download_content, 18
pro_fetch, 20
pro_fetch(), *15–18*
pro_query, 23
pro_query(), *14*, *16*, *22*, *28*
pro_rate_limit_status, 26
pro_request, *20*, 27
pro_request(), *22*, *31*, *32*, *34*
pro_request_jsonl, 29
pro_request_jsonl(), *31*, *34*
pro_request_jsonl_parquet, 30
pro_request_jsonl_parquet(), *34*
pro_request_parquet, *9*, *10*, *20*, 31
pro_request_parquet(), *17*, *18*, *22*
pro_validate_credentials, 34

read_corpus, 35

sample_parquet_n, 35
Sys.getenv, *13*

vignette, *8*, *30*