[sil]

Reference

a search index with facets and suggestions

version 3.4.1 · swindl.seecr.nl

seecr

Contents

An index 2
Index fields 2
Adding and deleting 3
Commits 3
Searching 3
The query language 4
CQL 4
Sorting 5
Drilldown 5
    Hierarchical drilldown 5
Suggestions 6
Changing the profile 6
Threads 6
Troubleshooting 6

sil (Seecr Indexing Lib) is a search index for Python applications, on tantivy (a Rust search engine, like Lucene). It adds what web applications need on top: an index profile in Python, a forgiving query language, sorting on several fields, drilldown (counts per value), suggestions while typing, and commits on a timer. One `Index` is one directory; no server, no JVM.

Install with `pip install seecr-sil` (the package is `sil`); it needs Python 3.11 or newer and tantivy (installed with it).

An index

```
from sil.index import Index
from sil.indexfield import IndexField
from sil.indexprofile import IndexProfile

index = Index(
    path,                                       # a pathlib.Path; made when missing
    IndexProfile([
        IndexField.unqualified("all"),
        IndexField.text("title", stored=True),
        IndexField.untokenized("genre", drilldown=True),
        IndexField.int("year", sortable=True),
    ]),
    immediate_commit=True,                      # or a timer, see Commits
)
index.add_record("book-1", {"all": "Dune Herbert", "title": "Dune",
                            "genre": "sf", "year": 1965})
result = index.search_query("dune", sort=[{"field": "year", "reverse": True}])
result["count"], result["docs"][0]            # 1, {"__id__": "book-1", "title": "Dune"}
index.shutdown()                                # commits and closes
```

`Index(path, profile, commit_timeout=10, immediate_commit=False)`:

`path`	the directory: `index/` (tantivy) and, for suggest fields, `suggest/<field>/`
`profile`	the fields, an `IndexProfile`
`commit_timeout`	seconds between commits of the timer; `0`: no timer
`immediate_commit`	commit after every change

Index fields

`IndexField.untokenized(name)`	the value as one term: codes, ids, statuses, tags. Exact match only; case matters
`IndexField.text(name)`	the value split into lowercased words: titles, descriptions
`IndexField.int(name)`	a 64 bit integer; ranges (`year:[1960 TO 1970]`) are fast
`IndexField.unqualified(name)`	a text field that a query without `field:` searches; at most one per profile

Options (keyword arguments):

`stored=True`	the value comes back in the hits (`hit["title"]`); otherwise only `__id__` does
`sortable=True`	can be sorted on (the first value of a document)
`drilldown=True`	values are counted in a search (untokenized and int fields)
`suggest=True`	values are offered by `suggest()`
`ascii_folding=True`	text and unqualified fields: accents and other non-ASCII letters are folded to ASCII, in the index and in queries, so `cafe` finds `Café` and `strasse` `Straße` (as Lucene's

	ASCIIFoldingFilter; meresco-lucene did this). Changing it needs a reindex
`hierarchical=True`	the values are paths (`vehicle>bicycle`), counted as a tree; with `drilldown=True` on an untokenized field (see Hierarchical drilldown)

A value can be a list: every value is indexed (searching finds any of them; drilldown counts all of them; sorting uses the first). `None` and `[]` mean no value. Every index has `__id__`, the record id, which is always stored.

Adding and deleting

`add_record(record_id, values)`	adds, or replaces the record with this id (an update is a delete followed by an add)
`delete_record(record_id, immediate_commit=False)`	removes it
`delete_all()`	removes everything, also the suggestions

`values` is a dict of field name to value; fields not in the profile are ignored. Nothing is searchable before a commit.

Commits

A search sees the index as it was at the last commit. A commit takes milliseconds (it writes and syncs files), so sil offers three ways:

`immediate_commit=True`	after every change: searchable at once; for tests and scripts
`commit_timeout=10` (default)	a timer (an asyncio task: make the index in a running event loop) commits every 10 seconds when there are changes
`commit_timeout=0`	only when asked: `commit(immediate_commit=True)`, and at `shutdown()`

`with index.deferred_commit():` puts off the commits of an immediate-commit index until the end of the block: one commit for many changes. Nested blocks commit at the end of the outermost one, also when it raises.

The writer (tantivy runs a few threads for it) is opened at the first change, not before: an index that is only searched costs little memory.

Searching

```
index.search(query, page=1, limit=20, sort=None, drilldown=None)
index.search_query("title:dune", ...)          # parses the query language
index.search_all(...)                           # every document
index.search_by_term("genre", "sf", ...)        # one exact term
index.search_by_id("book-1")
```

All give a dict:

`count`	the number of hits
`docs`	the hits of this page: `Hit`s, dicts of stored field to its first value (`__id__` included); `hit.get_values(name)` gives all values
`drilldown`	per drilldown field asked for, a list of `(value, count)`, most first

Build queries for `search()` with `parse_query(text)`, `term_query(field, value)` (exact, not analysed) and `Index.all_of(q1, q2, ...)` (all must match):

```
q = Index.all_of(index.parse_query("dune"), index.term_query("genre", "sf"))
index.search(q, page=2, limit=10)
```

Without sort, the best matches come first and equal scores in the order they were added; `*` and term queries list in the order of adding.

The query language

tantivy's query language, made forgiving for search boxes:

`dune herbert`	either word (the best matches, with both, first)
`+dune +herbert`, `dune AND herbert`	both
`rock -roll`	rock, not roll
`"and roll"`	the phrase
`title:dune`	in a field (words without a field search the `unqualified` field)
`roc*`	a prefix (made into a regex: tantivy has no prefix query)
`year:[1960 TO 1970]`, `year:{1960 TO *]`	ranges, inclusive `[]`, exclusive `{}`
`(a OR b) AND c`	grouping
`*`	everything

A query that is not valid syntax (a stray quote, an unknown field) is searched for literally, with the special characters as spaces, so a search box never gives an error.

CQL

For applications that speak CQL (SRU, or a portal that came from Meresco), sil understands CQL as well. It gives the same hits as Meresco (cqlparser and meresco-lucene) gave:

```
index.search_cql('fiets AND untokenized.type exact "vehicle>bicycle"', limit=10)
query = index.cql_query("year >= 2010 AND title = fiets")      # a Query for search()
```

`fiets`, `"fiets band"`	a word or a phrase in the `unqualified` field
`fiets*`	a prefix (two or more letters, digits, `_` or `-` before the `*`; `f*` is the word `f`, as in Meresco)
`*`	everything
`title = fiets`	in a text field: analysed as the field is (lowercase, folded with `ascii_folding`; several words are a phrase)
`title exact Fiets`, `title == Fiets`	the term as it is, not analysed; on an untokenized field `=` does the same
`type exact "a>b"`	a path in a hierarchical field
`year >= 2010`, `<`, `<=`, `>`	ranges, on int fields only
`a AND b`, `a OR b`, `a NOT b`	booleans; `OR` binds weakest, `AND` and `NOT` are read from left to right (as Meresco's cqlparser did; the CQL standard has no precedence)
`title =/boost=2 fiets`	a boost

Unlike the query language, CQL is not forgiving: CQL that can not be parsed, or that sil does not support (`any`, `all`, `adj`, `<>`, `prox`, modifiers other than `boost`, prefix assignments), raises `sil.cql.CqlError` (a `ValueError`). An index that is not in the profile matches nothing; a term without words (`title = "*"`) is left out of the query, as Meresco did.

The tree of a query, to look at or change before searching (for instance to rename old index names):

```
from dataclasses import replace
from sil.cql import parse, rewrite, to_cql

tree = parse("dc.title = fiets OR boten")
tree = rewrite(tree, lambda c: replace(c, index="title") if c.index == "dc.title" else c)
to_cql(tree)                        # 'title = fiets OR boten'
index.search_cql(tree)
```

A tree is made of `SearchClause(term, index, relation, boost)` and `Boolean(operator, left, right)` (operator `"and"`, `"or"` or `"not"`).

Sorting

`sort` is a field name, a dict `{"field": name, "reverse": True}`, or a list of those: sorted on the first, then the second for equal values, and so on; then in the order of adding. Only `sortable=True` fields (else `RuntimeError: Unsortable field`). A document without a value sorts as `""` or `0`.

Paging is stable: page 2 continues exactly where page 1 stopped, also when many documents have the same value.

Drilldown

```
result = index.search_query("dune", drilldown=[{"name": "genre", "count": 10}])
result["drilldown"]                    # {"genre": [("sf", 12), ("fantasy", 3)]}
```

Counts the values of `drilldown=True` fields in all hits of the query (not only the page), most first; equal counts alphabetically. `count` (default 10) is how many values; `0` gives all of them. To filter on a value someone clicked, add a `term_query` with `all_of`.

Hierarchical drilldown

Some values have levels: a category `vehicle` with below it `bicycle` and `car`, or an organisation with its departments. A hierarchical field counts them as a tree:

```
IndexField.untokenized("type", drilldown=True, hierarchical=True)

index.add_record("1", {"type": ("vehicle", "bicycle")})   # a path: a tuple
index.add_record("2", {"type": "vehicle>bicycle"})        # or levels with ">"
index.add_record("3", {"type": ("vehicle", "car")})
index.add_record("4", {"type": "boat"})                   # one level

result = index.search_all(drilldown=[{"name": "type"}])
result["drilldown"]
# {"type": [
#     ("vehicle", 3, [
#         ("bicycle", 2, []),
#         ("car", 1, []),
#     ]),
#     ("boat", 1, []),
# ]}
```

Each value comes with its count and the values one level below it, in the same form (`[]` for the last level), so `value, count = item[:2]` works for flat and hierarchical fields alike. A document counts once for every level of its paths, also when two of its paths share a level. In the request:

`count`	values per level (default 10; `0`: all)
`path`	start below this value: `["vehicle"]` or `"vehicle"` gives `[("bicycle", 2, []), ("car", 1, [])]`
`depth`	how many levels (default all); `1` gives the top level only, each with `[]`

A term query on a level finds everything below it: `term_query("type", "vehicle")` finds documents 1, 2 and 3; `term_query("type", "vehicle>car")` (or the tuple `("vehicle", "car")`) finds document 3.

A list of values is several values, as for any field, so a path goes in a tuple or as one string with `>`: `["vehicle", "bicycle"]` would be two values of one level each. A level can therefore not contain `>`. A hierarchical field can not be sortable, stored or suggest.

Suggestions

```
index.suggest("place", "new yo")       # ["New York", "York"]
```

Values of a `suggest=True` field, for typing ahead: the last word may be incomplete, the words before it must be whole; words anywhere in the value count. At most 10, in the order they were first added, without duplicates. Kept in `suggest/<field>/`; removed by `delete_all()`, not by `delete_record()`.

Changing the profile

tantivy has a schema: when the fields (or their options) of the profile change, the existing index cannot be used. sil then starts that index empty, logs a warning, and `index.needs_reindex` is `True`. The application rebuilds it from its data; sil never does that by itself.

Threads

Adding, deleting and committing are safe from several threads (one writer, behind a lock). A search works on a snapshot of the index: a commit in another thread does not disturb it. One process per index: tantivy allows one writer, and a second process fails to get it.

Troubleshooting

A search doesn't find what was just added	no commit yet: `immediate_commit=True`, or wait for the timer
`RuntimeError: no running event loop`	the timer needs one: make the index in the application's startup, or use `immediate_commit=True` / `commit_timeout=0`
An exact search on a text field finds nothing	text is split into lowercased words: `term_query("title", "dune")`
An untokenized search finds nothing	exact, case and all: `"Dune"` is not `"dune"`
`RuntimeError: Unsortable field`	add `sortable=True`, then rebuild the index
The index is empty after an upgrade	the profile changed; rebuild it (see the log)
`ValueError: Failed to acquire Lockfile: LockBusy`	another process (or another `Index` on the same directory) is writing to it