SQLAlchemy-Searchable Documentation

Release 0.10.4

Konsta Vesterinen

Jul 04, 2017

Contents

SQLAlchemy-Searchable provides full text search capabilities for SQLAlchemy models. Currently it only supports PostgreSQL.

CHAPTER 1

Installation

```
pip install SQLAlchemy-Searchable
```

Supported versions are python 2.7 and 3.3+.

CHAPTER 2

QuickStart

1. Import and call make_searchable function.
2. Define TSVectorType columns in your SQLAlchemy declarative model.

First let's define a simple Article model. This model has three fields: id, name and content. We want the name and content to be fulltext indexed, hence we put them inside the definition of TSVectorType.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_searchable import make_searchable
from sqlalchemy_utils.types import TSVectorType


Base = declarative_base()

make_searchable()


class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)
    search_vector = sa.Column(TSVectorType('name', 'content'))
```

Now lets create the tables and some dummy data. It is very important here that you either access your searchable class or call `configure_mappers` before the creation of tables. SA-Searchable adds DDL listeners on the configuration phase of models.

```
engine = create_engine('postgres://localhost/sqlalchemy_searchable_test')
sa.orm.configure_mappers()  # IMPORTANT!
Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine)
```

```
session = Session()

article1 = Article(name=u'First article', content=u'This is the first article')
article2 = Article(name=u'Second article', content=u'This is the second article')

session.add(article1)
session.add(article2)
session.commit()
```

After we've created the articles, we can search through them.

```
from sqlalchemy_searchable import search


query = session.query(Article)

query = search(query, 'first')

print query.first().name
# First article
```

Optionally specify `sort=True` to get results in order of relevance (ts_rank_cd):

```
query = search(query, 'first', sort=True)
```

CHAPTER 3

Search query parser

As of version 0.3.0 SQLAlchemy-Searchable comes with built in search query parser. The search query parser is capable of parsing human readable search queries into PostgreSQL search query syntax.

AND operator

Example: Search for articles containing 'star' and 'wars'

The default operator is 'and', hence the following queries are essentially the same:

```
query = search(query, 'star wars')
query2 = search(query, 'star and wars')
assert query == query2
```

OR operator

Searching for articles containing 'star' or 'wars'

```
query = search(query, 'star or wars')
```

Negation operator

SQLAlchemy-Searchable search query parser supports negation operator. By default the negation operator is '-'.

Example: Searching for article containing 'star' but not 'wars'

```
query = search(query, 'star or -wars')
```

Parenthesis

1. Searching for articles containing 'star' and 'wars' or 'luke'

```
query = search(query '(star wars) or luke')
```

Special cases

Hyphens between words

SQLAlchemy-Searchable is smart enough to not convert hyphens between words to negation operators. Instead, it simply converts all hyphens between words to spaces.

Hence the following search queries are essentially the same:

```
query = search(query, 'star wars')
query2 = search(query, 'star-wars')
```

Emails as search terms

PostgreSQL tsvectors handle email strings in a way that they don't get split into multiple tsvector terms. As of version 0.7 SQLAlchemy-Searchable doesn't handle emails this way by default, rather it splits the email into separate search terms. If you wish to override this behaviour you can use the *remove_symbols* configuration option.

```
# Tries to match 'john', 'fastmonkeys' and 'com' separately
query = search(query, u'john@fastmonkeys.com')
```

Internals

If you wish to use only the query parser this can be achieved by invoking *parse_search_query* function. This function parses human readable search query into PostgreSQL specific format.

```
parse_search_query('(star wars) or luke')
# (star:* & wars:*) | luke:*
```

CHAPTER 4

Configuration

SQLAlchemy-Searchable provides a number of customization options for the automatically generated search trigger, index and search_vector columns.

Global configuration options

The following configuration options can be defined globally by passing them to make_searchable function.

- search_trigger_name - name of the search database trigger, default: {table}_search_update
- search_trigger_function_name - the name for the database search vector updating function. This configuration option is only used if remove_hyphens is set as *True* otherwise the builtin postgresql *tsvector_update_trigger* is used for updating search vectors.
- regconfig - postgresql regconfig to be used, default: pg_catalog.english
- remove_symbols - String indicating all symbols that should be removed and converted to emptry strings in each search vectorized column. By default this is -@., meaning all -, @ and . will be converted to empty strings.

Before version 0.7.0 this configuration option was known as 'remove_hyphens' and provided only limited conversion of - symbols to empty strings.

Example

```
make_searchable(options={'regconfig': 'pg_catalog.finnish'})
```

Changing catalog for search vector

In the following example we use Finnish regconfig instead of the default English one.

```
class Article(Base):
    __tablename__ = 'article'
```

```
    name = sa.Column(sa.Unicode(255))

    search_vector = TSVectorType('name', regconfig='pg_catalog.finnish')
```

Weighting search results

PostgreSQL supports weighting search terms with weights A through D.

In this example, we give higher priority to terms appearing in the article title than in the content.

```
class Article(Base):
    __tablename__ = 'article'

    title = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)

    search_vector = sa.Column(
        TSVectorType('title', 'content',
                     weights={'title': 'A', 'content': 'B'})
    )
```

Note that in order to see the effect of this weighting, you must search with `sort=True`

```
query = session.query(Article)
query = search(query, 'search text', sort=True)
```

Multiple search vectors per class

```
class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)
    description = sa.Column(sa.UnicodeText)
    simple_search_vector = sa.Column(TSVectorType('name'))

    fat_search_vector = sa.Column(
        TSVectorType('name', 'content', 'description')
    )
```

After that, we can choose which search vector to use.

```
query = session.query(Article)
query = search(query, 'first', vector=fat_search_vector)
```

Combined search vectors

Sometimes you may want to search from multiple tables at the same time. This can be achieved using combined search vectors.

Consider the following model definition. Here each article has one author.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy_utils.types import TSVectorType


Base = declarative_base()


class Category(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    search_vector = sa.Column(TSVectorType('name'))


class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)
    search_vector = sa.Column(TSVectorType('name', 'content'))
    category_id = sa.Column(
        sa.Integer,
        sa.ForeignKey(Category.id)
    )
    category = sa.orm.relationship(Category)
```

Now consider a situation where we want to find all articles, where either article content or name or category name contains the word 'matrix'. This can be achieved as follows:

```
from sqlalchemy_searchable import parse_search_query


search_query = u'matrix'

combined_search_vector = Article.search_vector | Category.search_vector

articles = (
    session.query(Article)
    .join(Category)
    .filter(
        combined_search_vector.match(
            parse_search_query(search_query)
        )
    )
)
```

This query becomes a little more complex when using left joins. Then you have to take into account situations where Category.search_vector is None using coalesce function.

```
combined_search_vector = (
    Article.search_vector
    |
```

```
    sa.func.coalesce(Category.search_vector, u'')
)
```

CHAPTER 5

Vectorizers

Vectorizers provide means for changing the way how different column types and columns are turned into fulltext search vectors.

Type vectorizers

By default PostgreSQL only knows how to vectorize string columns. If your model contains for example HSTORE column which you would like to fulltext index you need to define special vectorization rule for this.

The easiest way to add a vectorization rule is by using the vectorizer decorator. In the following example we vectorize only the values of all HSTORE typed columns are models may have.

```
import sqlalchemy as sa
from sqlalchemy.dialects.postgresql import HSTORE
from sqlalchemy_searchable import vectorizer


@vectorizer(HSTORE)
def hstore_vectorizer(column):
    return sa.cast(sa.func.avals(column), sa.Text)
```

The SQLAlchemy clause construct returned by the vectorizer will be used for all fulltext indexed columns that are of type HSTORE. Consider the following model:

```
class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer)
    name_translations = sa.Column(HSTORE)
    content_translations = sa.Column(HSTORE)
```

Now SQLAlchemy-Searchable would create the following search trigger for this model (with default configuration)

```
CREATE FUNCTION
    textitem_search_vector_update() RETURNS TRIGGER AS $$
BEGIN
    NEW.search_vector = to_tsvector(
        'simple',
        concat(
            regexp_replace(
                coalesce(
                    CAST(avals(NEW.name_translations) AS TEXT),
                    ''
                ),
                '[-@.]', ' ', 'g'
            ),
            ' ',
            regexp_replace(
                coalesce(
                    CAST(avals(NEW.content_translations) AS TEXT),
                    ''
                ),
                '[-@.]', ' ', 'g'),
                ' '
            )
        );
    RETURN NEW;
END
$$ LANGUAGE 'plpgsql';
```

Column vectorizers

Sometimes you may want to set special vectorizer only for specific column. This can be achieved as follows:

```
class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer)
    name_translations = sa.Column(HSTORE)


@vectorizer(Article.name_translations)
def name_vectorizer(column):
    return sa.cast(sa.func.avals(column), sa.Text)
```

Note: Column vectorizers always have precedence over type vectorizers.

API

class sqlalchemy_searchable.vectorizers.**Vectorizer**(*type_vectorizers=None*, *column_vectorizers=None*)

CHAPTER 6

Alembic migrations

When making changes to your database schema you have to make sure the associated search triggers and trigger functions get updated also. SQLAlchemy-Searchable offers a helper function called *sync_trigger* for this.

sqlalchemy_searchable.**sync_trigger**(*conn*, *table_name*, *tsvector_column*, *indexed_columns*, *metadata=None*, *options=None*)

Synchronizes search trigger and trigger function for given table and given search index column. Internally this function executes the following SQL queries:

- Drops search trigger for given table (if it exists)
- Drops search function for given table (if it exists)
- Creates search function for given table
- Creates search trigger for given table
- Updates all rows for given search vector by running a column=column update query for given table.

Example:

```
from sqlalchemy_searchable import sync_trigger


sync_trigger(
    conn,
    'article',
    'search_vector',
    ['name', 'content']
)
```

This function is especially useful when working with alembic migrations. In the following example we add a content column to article table and then sync the trigger to contain this new column.

```
from alembic import op
from sqlalchemy_searchable import sync_trigger
```

```
def upgrade():
    conn = op.get_bind()
    op.add_column('article', sa.Column('content', sa.Text))

    sync_trigger(conn, 'article', 'search_vector', ['name', 'content'])

# ... same for downgrade
```

If you are using vectorizers you need to initialize them in your migration file and pass them to this function.

```
import sqlalchemy as sa
from alembic import op
from sqlalchemy.dialects.postgresql import HSTORE
from sqlalchemy_searchable import sync_trigger, vectorizer


def upgrade():
    vectorizer.clear()

    conn = op.get_bind()
    op.add_column('article', sa.Column('name_translations', HSTORE))

    metadata = sa.MetaData(bind=conn)
    articles = sa.Table('article', metadata, autoload=True)

    @vectorizer(articles.c.name_translations)
    def hstore_vectorizer(column):
        return sa.cast(sa.func.avals(column), sa.Text)

    op.add_column('article', sa.Column('content', sa.Text))
    sync_trigger(
        conn,
        'article',
        'search_vector',
        ['name_translations', 'content'],
        metadata=metadata
    )

# ... same for downgrade
```

Parameters

- **conn** – SQLAlchemy Connection object
- **table_name** – name of the table to apply search trigger syncing
- **tsvector_column** – TSVector typed column which is used as the search index column
- **indexed_columns** – Full text indexed column names as a list
- **metadata** – Optional SQLAlchemy metadata object that is being used for autoloaded Table. If None is given then new MetaData object is initialized within this function.
- **options** – Dictionary of configuration options

CHAPTER 7

Flask-SQLAlchemy integration

SQLAlchemy-Searchable can be neatly integrated into Flask-SQLAlchemy using SearchQueryMixin class.

Example

```
from flask.ext.sqlalchemy import SQLAlchemy, BaseQuery
from sqlalchemy_searchable import SearchQueryMixin
from sqlalchemy_utils.types import TSVectorType
from sqlalchemy_searchable import make_searchable

db = SQLAlchemy()

make_searchable()


class ArticleQuery(BaseQuery, SearchQueryMixin):
    pass


class Article(db.Model):
    query_class = ArticleQuery
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)
    search_vector = db.Column(TSVectorType('name', 'content'))

db.configure_mappers() #very important!
db.create_all()
db.commit()
```

Now this is where the fun begins! SearchQueryMixin provides a search method for ArticleQuery. You can chain calls just like when using query filter calls. Here we search for first 5 articles that contain the word 'Finland'.

```
Article.query.search(u'Finland').limit(5).all()
```

When using Flask-SQLAlchemy, the columns of your models might be set to various DataTypes(i.e db.Datetime, db.String, db.Integer, etc.). As of 30/06/2016, SQLAlchemy-searchable does not support those datatypes and returns a TypeError when said columns are implemented in the search vector. Instead, use Unicode and UnicodeText accordingly.

Python Module Index

S

Index

S

V