
SearchCop



SearchCop 🛡️

test passing
gem version 1.4.0



SearchCop extends your ActiveRecord models to support fulltext search engine like queries via simple query strings and hash-based queries. Assume you have a `Book` model having various attributes like `title`, `author`, `stock`, `price`, `available`. Using SearchCop you can perform:

```
1 Book.search("Joanne Rowling Harry Potter")
2 Book.search("author: Rowling title:'Harry Potter'")
3 Book.search("price > 10 AND price < 20 -stock:0 (Potter OR Rowling)")
4 # ...
```

Thus, you can hand out a search query string to your models and you, your app's admins and/or users will get powerful query features without the need for integrating additional third party search servers, since SearchCop can use fulltext index capabilities of your RDBMS in a database agnostic way (currently MySQL and PostgreSQL fulltext indices are supported) and optimizes the queries to make optimal use of them. Read more below.

Complex hash-based queries are supported as well:

```
1 Book.search(author: "Rowling", title: "Harry Potter")
2 Book.search(or: [{author: "Rowling"}, {author: "Tolkien"}])
3 Book.search(and: [{price: {gt: 10}}, {not: {stock: 0}}, or: [{title: "
  Potter"}, {author: "Rowling"}]})
4 Book.search(or: [{query: "Rowling -Potter"}, {query: "Tolkien -Rings"}])
```

```
5 Book.search(title: {my_custom_sql_query: "Row1"})
6 # ...
```

Installation

Add this line to your application's Gemfile:

```
1 gem 'search_cop'
```

And then execute:

```
1 $ bundle
```

Or install it yourself as:

```
1 $ gem install search_cop
```

Usage

To enable SearchCop for a model, `include SearchCop` and specify the attributes you want to expose to search queries within a `search_scope`:

```
1 class Book < ActiveRecord::Base
2   include SearchCop
3
4   search_scope :search do
5     attributes :title, :description, :stock, :price, :created_at, :
      available
6     attributes comment: ["comments.title", "comments.message"]
7     attributes author: "author.name"
8     # ...
9   end
10
11   has_many :comments
12   belongs_to :author
13 end
```

You can of course as well specify multiple `search_scope` blocks as you like:

```
1 search_scope :admin_search do
2   attributes :title, :description, :stock, :price, :created_at, :
      available
3
4   # ...
5 end
6
```

```
7 search_scope :user_search do
8   attributes :title, :description
9
10  # ...
11 end
```

How does it work

SearchCop parses the query and maps it to an SQL Query in a database agnostic way. Thus, SearchCop is not bound to a specific RDBMS.

```
1 Book.search("stock > 0")
2 # ... WHERE books.stock > 0
3
4 Book.search("price > 10 stock > 0")
5 # ... WHERE books.price > 10 AND books.stock > 0
6
7 Book.search("Harry Potter")
8 # ... WHERE (books.title LIKE '%Harry%' OR books.description LIKE '%
9   Harry%' OR ...) AND (books.title LIKE '%Potter%' OR books.
10  description LIKE '%Potter%' ...)
11
12 Book.search("available:yes OR created_at:2014")
13 # ... WHERE books.available = 1 OR (books.created_at >= '2014-01-01
14   00:00:00.000000' and books.created_at <= '2014-12-31 23:59:59.99999')
```

SearchCop is using ActiveSupport's `beginning_of_year` and `end_of_year` methods for the values used in building the SQL query for this case.

Of course, these `LIKE '%...%'` queries won't achieve optimal performance, but check out the section below on SearchCop's fulltext capabilities to understand how the resulting queries can be optimized.

As `Book.search(...)` returns an `ActiveRecord::Relation`, you are free to pre- or post-process the search results in every possible way:

```
1 Book.where(available: true).search("Harry Potter").order("books.id desc")
2   .paginate(page: params[:page])
```

Security

When you pass a query string to SearchCop, it gets parsed, analyzed and mapped to finally build up an SQL query. To be more precise, when SearchCop parses the query, it creates objects (nodes), which represent the query expressions (And-, Or-, Not-, String-, Date-, etc Nodes). To build the SQL query,

SearchCop uses the concept of visitors like e.g. used in Arel, such that, for every node there must be a visitor, which transforms the node to SQL. When there is no visitor, an exception is raised when the query builder tries to “visit” the node. The visitors are responsible for sanitizing the user supplied input. This is primarily done via quoting (string-, table-name-, column-quoting, etc). SearchCop is using the methods provided by the ActiveRecord connection adapter for sanitizing/quoting to prevent SQL injection. While we can never be 100% safe from security issues, SearchCop takes security issues seriously. Please report responsibly via security at flakks dot com in case you find any security related issues.

json/jsonb/hstore

SearchCop supports json fields for MySQL, as well as json, jsonb and hstore fields for postgres. Currently, field values are always expected to be strings and no arrays are supported. You can specify json attributes via:

```
1 search_scope :search do
2   attributes user_agent: "context->browser->user_agent"
3
4   # ...
5 end
```

where `context` is a json/jsonb column which e.g. contains:

```
1 {
2   "browser": {
3     "user_agent": "Firefox ..."
4   }
5 }
```

Fulltext index capabilities

By default, i.e. if you don't tell SearchCop about your fulltext indices, SearchCop will use `LIKE '%...%'` queries. Unfortunately, unless you create a trigram index (postgres only), these queries can not use SQL indices, such that every row needs to be scanned by your RDBMS when you search for `Book.search("Harry Potter")` or similar. To avoid the penalty of `LIKE` queries, SearchCop can exploit the fulltext index capabilities of MySQL and PostgreSQL. To use already existing fulltext indices, simply tell SearchCop to use them via:

```
1 class Book < ActiveRecord::Base
2   # ...
3
4   search_scope :search do
```

```

5     attributes :title, :author
6
7     options :title, :type => :fulltext
8     options :author, :type => :fulltext
9 end
10
11 # ...
12 end

```

SearchCop will then transparently change its SQL queries for the attributes having fulltext indices to:

```

1 Book.search("Harry Potter")
2 # MySQL: ... WHERE (MATCH(books.title) AGAINST('+Harry' IN BOOLEAN MODE)
  ) OR MATCH(books.author) AGAINST('+Harry' IN BOOLEAN MODE)) AND (
  MATCH(books.title) AGAINST ('+Potter' IN BOOLEAN MODE) OR MATCH(
  books.author) AGAINST('+Potter' IN BOOLEAN MODE))
3 # PostgreSQL: ... WHERE (to_tsvector('simple', books.title) @@
  to_tsquery('simple', 'Harry') OR to_tsvector('simple', books.author)
  @@ to_tsquery('simple', 'Harry')) AND (to_tsvector('simple', books.
  title) @@ to_tsquery('simple', 'Potter') OR to_tsvector('simple',
  books.author) @@ to_tsquery('simple', 'Potter'))

```

Obviously, these queries won't always return the same results as wildcard [LIKE](#) queries, because we search for words instead of sub-strings. However, fulltext indices will usually of course provide better performance.

Moreover, the query above is not yet perfect. To improve it even more, SearchCop tries to optimize the queries to make optimal use of fulltext indices while still allowing to mix them with non-fulltext attributes. To improve queries even more, you can group attributes and specify a default field to search in, such that SearchCop must no longer search within all fields:

```

1 search_scope :search do
2   attributes all: [:author, :title]
3
4   options :all, :type => :fulltext, default: true
5
6   # Use default: true to explicitly enable fields as default fields (
  whitelist approach)
7   # Use default: false to explicitly disable fields as default fields (
  blacklist approach)
8 end

```

Now SearchCop can optimize the following, not yet optimal query:

```

1 Book.search("Rowling OR Tolkien stock > 1")
2 # MySQL: ... WHERE ((MATCH(books.author) AGAINST('+Rowling' IN BOOLEAN
  MODE) OR MATCH(books.title) AGAINST('+Rowling' IN BOOLEAN MODE)) OR
  (MATCH(books.author) AGAINST('+Tolkien' IN BOOLEAN MODE) OR MATCH(

```

```

books.title) AGAINST('+Tolkien' IN BOOLEAN MODE))) AND books.stock >
1
3 # PostgreSQL: ... WHERE ((to_tsvector('simple', books.author) @@
to_tsquery('simple', 'Rowling') OR to_tsvector('simple', books.title)
) @@ to_tsquery('simple', 'Rowling')) OR (to_tsvector('simple',
books.author) @@ to_tsquery('simple', 'Tolkien') OR to_tsvector('
simple', books.title) @@ to_tsquery('simple', 'Tolkien')) AND books
.stock > 1

```

to the following, more performant query:

```

1 Book.search("Rowling OR Tolkien stock > 1")
2 # MySQL: ... WHERE MATCH(books.author, books.title) AGAINST('Rowling
Tolkien' IN BOOLEAN MODE) AND books.stock > 1
3 # PostgreSQL: ... WHERE to_tsvector('simple', books.author || ' ' ||
books.title) @@ to_tsquery('simple', 'Rowling | Tolkien') and books.
stock > 1

```

What is happening here? Well, we specified `all` as the name of an attribute group that consists of `author` and `title`. As we, in addition, specified `all` to be a fulltext attribute, SearchCop assumes there is a compound fulltext index present on `author` and `title`, such that the query is optimized accordingly. Finally, we specified `all` to be the default attribute to search in, such that SearchCop can ignore other attributes, like e.g. `stock`, as long as they are not specified within queries directly (like for `stock > 0`).

Other queries will be optimized in a similar way, such that SearchCop tries to minimize the fulltext constraints within a query, namely `MATCH() AGAINST()` for MySQL and `to_tsvector() @@ to_tsquery()` for PostgreSQL.

```

1 Book.search("(Rowling -Potter) OR Tolkien")
2 # MySQL: ... WHERE MATCH(books.author, books.title) AGAINST('(Rowling
-Potter) Tolkien' IN BOOLEAN MODE)
3 # PostgreSQL: ... WHERE to_tsvector('simple', books.author || ' ' ||
books.title) @@ to_tsquery('simple', '(Rowling & !Potter) | Tolkien
')

```

To create a fulltext index on `books.title` in MySQL, simply use:

```

1 add_index :books, :title, :type => :fulltext

```

Regarding compound indices, which will e.g. be used for the default field `all` we already specified above, use:

```

1 add_index :books, [:author, :title], :type => :fulltext

```

Please note that MySQL supports fulltext indices for MyISAM and, as of MySQL version 5.6+, for InnoDB as well. For more details about MySQL fulltext indices visit <http://dev.mysql.com/doc/refman/5.6/e>

n/fulltext-search.html

Regarding PostgreSQL there are more ways to create a fulltext index. However, one of the easiest ways is:

```
1 ActiveRecord::Base.connection.execute "CREATE INDEX
    fulltext_index_books_on_title ON books USING GIN(to_tsvector('simple
    ', title))"
```

Moreover, for PostgreSQL you should change the schema format in `config/application.rb`:

```
1 config.active_record.schema_format = :sql
```

Regarding compound indices for PostgreSQL, use:

```
1 ActiveRecord::Base.connection.execute "CREATE INDEX
    fulltext_index_books_on_title ON books USING GIN(to_tsvector('simple
    ', author || ' ' || title))"
```

To handle NULL values with PostgreSQL correctly, use COALESCE both at index creation time and when specifying the `search_scope`:

```
1 ActiveRecord::Base.connection.execute "CREATE INDEX
    fulltext_index_books_on_title ON books USING GIN(to_tsvector('simple
    ', COALESCE(author, '') || ' ' || COALESCE(title, '')))"
```

plus:

```
1 search_scope :search do
2   attributes :title
3
4   options :title, :type => :fulltext, coalesce: true
5 end
```

To use another PostgreSQL dictionary than `simple`, you have to create the index accordingly and you need tell SearchCop about it, e.g.:

```
1 search_scope :search do
2   attributes :title
3
4   options :title, :type => :fulltext, dictionary: "english"
5 end
```

For more details about PostgreSQL fulltext indices visit <http://www.postgresql.org/docs/9.3/static/textsearch.html>

Other indices

In case you expose non-fulltext attributes to search queries (price, stock, etc.), the respective queries, like `Book.search("stock > 0")`, will profit from the usual non-fulltext indices. Thus, you should add a usual index on every column you expose to search queries plus a fulltext index for every fulltext attribute.

In case you can't use fulltext indices, because you're e.g. still on MySQL 5.5 while using InnoDB or another RDBMS without fulltext support, you can make your RDBMS use usual non-fulltext indices for string columns if you don't need the left wildcard within `LIKE` queries. Simply supply the following option:

```
1 class User < ActiveRecord::Base
2   include SearchCop
3
4   search_scope :search do
5     attributes :username
6
7     options :username, left_wildcard: false
8   end
9
10  # ...
```

such that SearchCop will omit the left most wildcard.

```
1 User.search("admin")
2 # ... WHERE users.username LIKE 'admin%'
```

Similarly, you can disable the right wildcard as well:

```
1 search_scope :search do
2   attributes :username
3
4   options :username, right_wildcard: false
5 end
```

Default operator

When you define multiple fields on a search scope, SearchCop will use by default the AND operator to concatenate the conditions, e.g:

```
1 class User < ActiveRecord::Base
2   include SearchCop
3
4   search_scope :search do
5     attributes :username, :fullname
```

```
6   end
7
8   # ...
9   end
```

So a search like `User.search("something")` will generate a query with the following conditions:

```
1 ... WHERE username LIKE '%something%' AND fullname LIKE '%something%'
```

However, there are cases where using AND as the default operator is not desired, so SearchCop allows you to override it and use OR as the default operator instead. A query like `User.search("something", default_operator: :or)` will generate the query using OR to concatenate the conditions

```
1 ... WHERE username LIKE '%something%' OR fullname LIKE '%something%'
```

Finally, please note that you can apply it to fulltext indices/queries as well.

Associations

If you specify searchable attributes from another model, like

```
1 class Book < ActiveRecord::Base
2   # ...
3
4   belongs_to :author
5
6   search_scope :search do
7     attributes author: "author.name"
8   end
9
10  # ...
11 end
```

SearchCop will by default `eager_load` the referenced associations, when you perform `Book.search(...)`. If you don't want the automatic `eager_load` or need to perform special operations, specify a `scope`:

```
1 class Book < ActiveRecord::Base
2   # ...
3
4   search_scope :search do
5     # ...
6
7     scope { joins(:author).eager_load(:comments) } # etc.
8   end
```

```
9
10 # ...
11 end
```

SearchCop will then skip any association auto loading and will use the scope instead. You can as well use `scope` together with `aliases` to perform arbitrarily complex joins and search in the joined models/tables:

```
1 class Book < ActiveRecord::Base
2   # ...
3
4   search_scope :search do
5     attributes similar: ["similar_books.title", "similar_books.
6       description"]
7
8     scope do
9       joins "left outer join books similar_books on ..."
10    end
11
12    aliases similar_books: Book # Tell SearchCop how to map SQL aliases
13      to models
14  end
15 end
```

Associations of associations can as well be referenced and used:

```
1 class Book < ActiveRecord::Base
2   # ...
3
4   has_many :comments
5   has_many :users, :through => :comments
6
7   search_scope :search do
8     attributes user: "users.username"
9   end
10
11   # ...
12 end
```

Custom table names and associations

SearchCop tries to infer a model's class name and SQL alias from the specified attributes to autodetect datatype definitions, etc. This usually works quite fine. In case you're using custom table names via `self.table_name = ...` or if a model is associated multiple times, SearchCop however can't infer the class and SQL alias names, e.g.

```

1 class Book < ActiveRecord::Base
2   # ...
3
4   has_many :users, :through => :comments
5   belongs_to :user
6
7   search_scope :search do
8     attributes user: ["user.username", "users_books.username"]
9   end
10
11   # ...
12 end

```

Here, for queries to work you have to use `users_books.username`, because ActiveRecord assigns a different SQL alias for users within its SQL queries, because the user model is associated multiple times. However, as SearchCop now can't infer the `User` model from `users_books`, you have to add:

```

1 class Book < ActiveRecord::Base
2   # ...
3
4   search_scope :search do
5     # ...
6
7     aliases :users_books => :users
8   end
9
10   # ...
11 end

```

to tell SearchCop about the custom SQL alias and mapping. In addition, you can always do the joins yourself via a `scope {}` block plus `aliases` and use your own custom sql aliases to become independent of names auto-assigned by ActiveRecord.

Supported operators

Query string queries support `AND/and`, `OR/or`, `:`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `NOT/not/-`, `()`, `"..."` and `'...'`. Default operators are `AND` and `matches`, `OR` has precedence over `AND`. `NOT` can only be used as infix operator regarding a single attribute.

Hash based queries support `and: [...]` and `or: [...]`, which take an array of `not: {...}`, `matches: {...}`, `eq: {...}`, `not_eq: {...}`, `lt: {...}`, `lteq: {...}`, `gt: {...}`, `gteq: {...}` and `query: "..."` arguments. Moreover, `query: "..."` makes it possible to create sub-queries. The other rules for query string queries apply to hash based queries as well.

Custom operators (Hash based queries)

SearchCop also provides the ability to define custom operators by defining a [generator](#) in [search_scope](#). They can then be used with the hash based query search. This is useful when you want to use database operators that are not supported by SearchCop.

Please note, when using generators, you are responsible for sanitizing/quoting the values (see example below). Otherwise your generator will allow SQL injection. Thus, please only use generators if you know what you're doing.

For example, if you wanted to perform a [LIKE](#) query where a book title starts with a string, you can define the search scope like so:

```
1 search_scope :search do
2   attributes :title
3
4   generator :starts_with do |column_name, raw_value|
5     pattern = "#{raw_value}%"
6     "#{column_name} LIKE #{quote pattern}"
7   end
8 end
```

When you want to perform the search you use it like this:

```
1 Book.search(title: { starts_with: "The Great" })
```

Security Note: The query returned from the generator will be interpolated directly into the query that goes to your database. This opens up a potential SQL Injection point in your app. If you use this feature you'll want to make sure the query you're returning is safe to execute.

Mapping

When searching in boolean, datetime, timestamp, etc. fields, SearchCop performs some mapping. The following queries are equivalent:

```
1 Book.search("available:true")
2 Book.search("available:1")
3 Book.search("available:yes")
```

as well as

```
1 Book.search("available:false")
2 Book.search("available:0")
3 Book.search("available:no")
```

For datetime and timestamp fields, SearchCop expands certain values to ranges:

```

1 Book.search("created_at:2014")
2 # ... WHERE created_at >= '2014-01-01 00:00:00' AND created_at <=
  '2014-12-31 23:59:59'
3
4 Book.search("created_at:2014-06")
5 # ... WHERE created_at >= '2014-06-01 00:00:00' AND created_at <=
  '2014-06-30 23:59:59'
6
7 Book.search("created_at:2014-06-15")
8 # ... WHERE created_at >= '2014-06-15 00:00:00' AND created_at <=
  '2014-06-15 23:59:59'

```

Chaining

Chaining of searches is possible. However, chaining does currently not allow SearchCop to optimize the individual queries for fulltext indices.

```

1 Book.search("Harry").search("Potter")

```

will generate

```

1 # MySQL: ... WHERE MATCH(...) AGAINST('+Harry' IN BOOLEAN MODE) AND
  MATCH(...) AGAINST('+Potter' IN BOOLEAN MODE)
2 # PostgreSQL: ... WHERE to_tsvector(...) @@ to_tsquery('simple', 'Harry
  ') AND to_tsvector(...) @@ to_tsquery('simple', 'Potter')

```

instead of

```

1 # MySQL: ... WHERE MATCH(...) AGAINST('+Harry +Potter' IN BOOLEAN MODE)
2 # PostgreSQL: ... WHERE to_tsvector(...) @@ to_tsquery('simple', 'Harry
  & Potter')

```

Thus, if you use fulltext indices, you better avoid chaining.

Debugging

When using `Model#search`, SearchCop conveniently prevents certain exceptions from being raised in case the query string passed to it is invalid (parse errors, incompatible datatype errors, etc). Instead, `Model#search` returns an empty relation. However, if you need to debug certain cases, use `Model#unsafe_search`, which will raise them.

```

1 Book.unsafe_search("stock: None") # => raise SearchCop::
  IncompatibleDatatype

```

Reflection

SearchCop provides reflective methods, namely `#attributes`, `#default_attributes`, `#options` and `#aliases`. You can use these methods to e.g. provide an individual search help widget for your models, that lists the attributes to search in as well as the default ones, etc.

```
1 class Product < ActiveRecord::Base
2   include SearchCop
3
4   search_scope :search do
5     attributes :title, :description
6
7     options :title, default: true
8   end
9 end
10
11 Product.search_reflection(:search).attributes
12 # {"title" => ["products.title"], "description" => ["products.
13   description"]}
14 Product.search_reflection(:search).default_attributes
15 # {"title" => ["products.title"]}
16
17 # ...
```

Semantic Versioning

Starting with version 1.0.0, SearchCop uses Semantic Versioning: SemVer

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create new Pull Request