# Full Text Search

16th December, 2020

## Basic Search

### Introduction

- A common task for web applications is to search some data in the database with user input.
- We commonly use pattern searching in SQL to facilitate search, such as:
  - SELECT column_name FROM table_name WHERE column_name LIKE 'pattern';
- In an ORM ( here Django) it may be translated as:
  - In [1]: Product.objects.filter(name__contains=‘Marinela')
  - Out[2]: [<Product: Pinguinos Marinela>, <Product: Gansito Marinela>]
- But these search forms can be very complex to execute. Often returning overwhelming amount of results that may or may not be relevant.

## Basic Search

### Limitations

- Many of the limitations of these search forms include:
  - There is no linguistic support, even for English. Eg: it may miss “satisfy” when we searched “satisfies”. ORing every word is not possible.
  - They provide no ordering (ranking) of search results. What happens when there are thousands of matching documents?
  - They must process all documents for every search because of no indexing.

# Full-text Search

## Introduction

- A more effective way to search is to obtain a "semantic vector” for all the words contained in the document.
- So when you search for a word like "run", it will match all instances of the word and it’s different conjugated forms (ie, "ran" or "running").
- However, it will not search the entire document itself (slow), but the vector (fast).
- Full Text Searching provides the capability to identify natural-language documents that satisfy a query.

# Full-text Search

## Introduction

- We may choose to sort these documents by relevance to the query.
- The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.
- In simple searches,
    - query -> set of words,
    - similarity -> frequency of query words in the document.

## Full-text Search

### Document

- A document is the unit of searching in a full text search system.
- It may be a magazine article, email, etc.
- Normally, a textual field within a row of a database table, or possibly a combination of such fields.
- We may store documents as text-files in system but it requires superuser permissions and is usually less convenient.

## Full-text Search
### Preprocessing

- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching. It includes:
    - **Parsing documents into tokens.** i.e parsing words, numbers, email etc
    - **Converting tokens into lexemes**. i.e. different form of same word, stop words etc.
    - **Storing preprocessed documents optimized for searching.** i.e store as array of normalized lexemes.

# Full-text Search

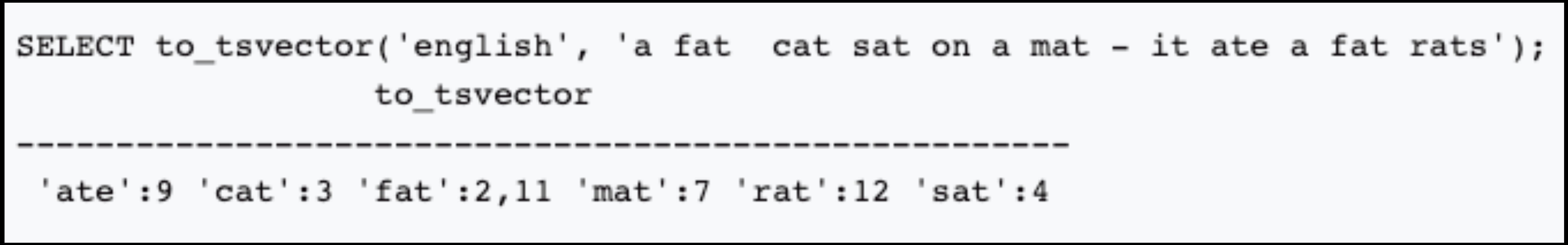
## Preprocessing / Dictionaries

- Dictionaries allow fine-grained control over how tokens are normalized.
- These can include:
  - Define stop words that should not be indexed.
  - Map synonyms to a single word using Ispell.
  - Map phrases to a single word using a thesaurus.
  - Map different variations of a word to a canonical form using an Ispell dictionary.
  - Map different variations of a word to a canonical form using Snowball stemmer rules.

## Full-text Search

### tsvector

- A tsvector value is a sorted list of distinct lexemes
- Each document must be reduced to the preprocessed tsvector format.
  - Searching and ranking are performed entirely on the tsvector representation of a document.
  - The original text need only be retrieved when the document has been selected for display to a user.
- We use to_tsvector for converting a document to the tsvector data type.

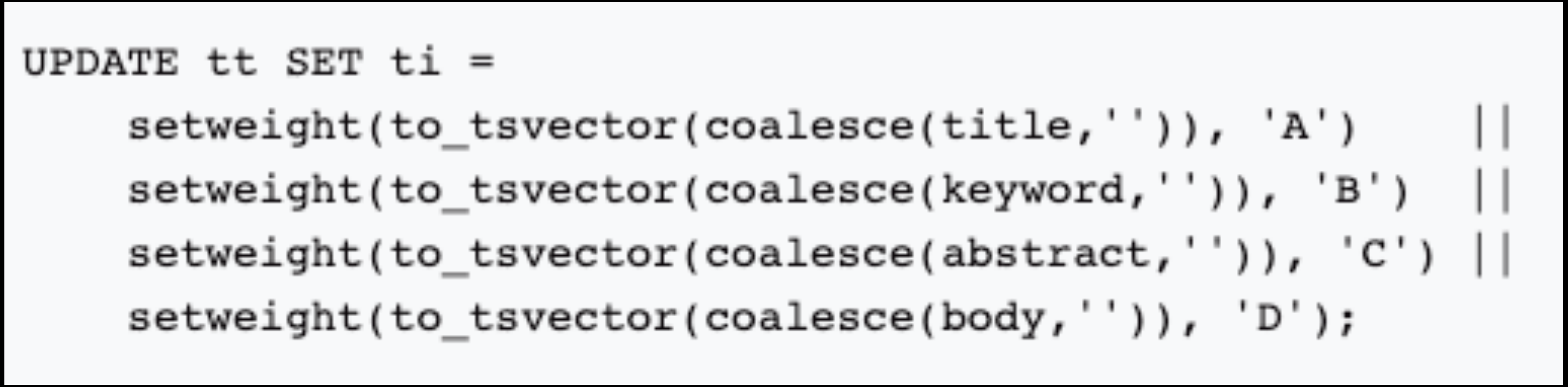


```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

# Full-text Search

## tsvector / weight

- The function setweight can be used to label the entries of a tsvector with a given weight.
- A weight is one of the letters A, B, C, or D mapping to [1.0, 0.4, 0.2, 0.1]
- This is used to give importance to entries coming from different parts of a document, such as title versus body.
- We may use our own set of weights.



```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

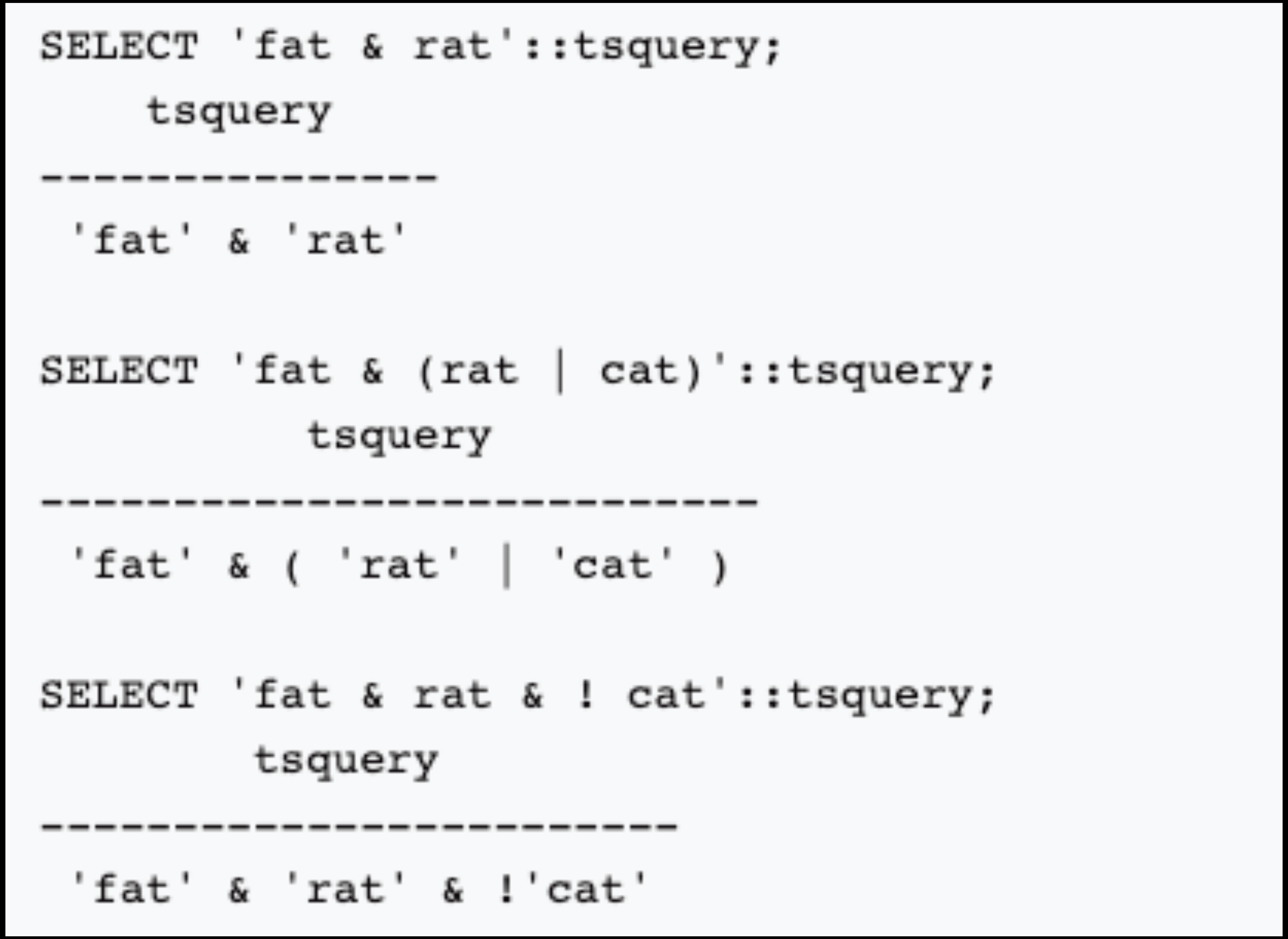
## Full-text Search

### tsquery

- A tsquery value stores lexemes that are to be searched for.
- We can use to_tsquery or phraseto_tsquery to convert to a tsquery
- We can combine them using the Boolean operators & (AND), | (OR), and ! (NOT), as well as the phrase search operator <-> (FOLLOWED BY).
- Also, lexemes in a tsquery can be:
  - labeled with * to specify prefix matching
  - labeled with one or more weight letters, which restricts them to match only tsvector lexemes with one of those weights:

## Full-text Search

### tsquery

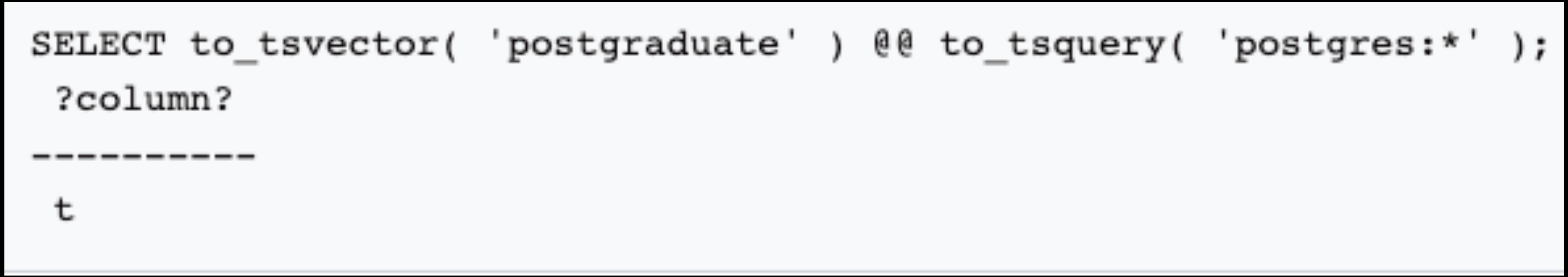


```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

Here, stemming of postgraduate and postgres to postgre

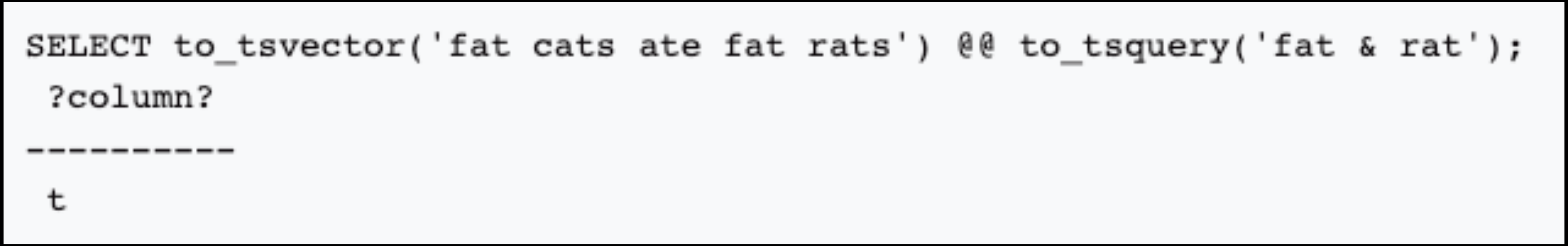


```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

# Full-text Search

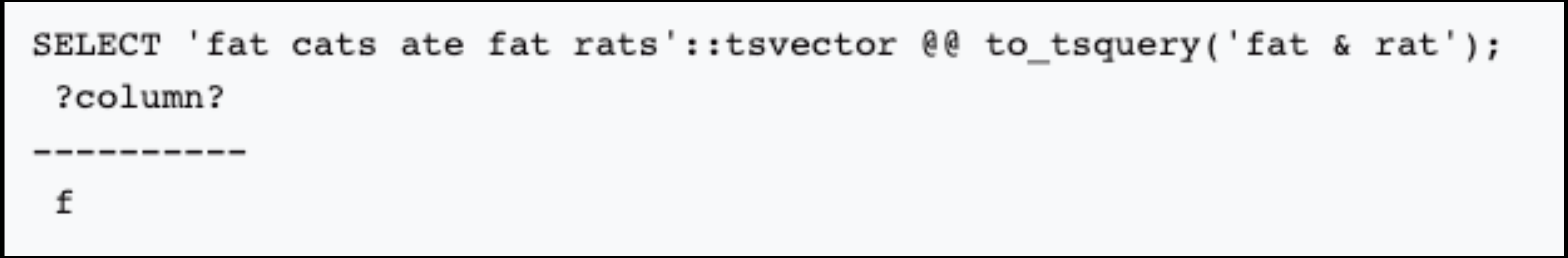
## Basic Text Matching

- Full text searching in PostgreSQL is based on the match operator @@ which returns true if a tsvector (document) matches a tsquery (query).



```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

- This fails because rats is not normalized to rat

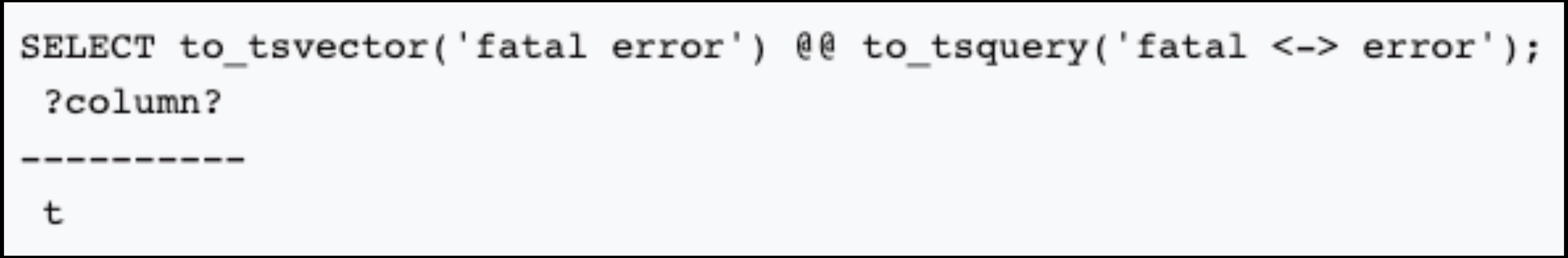


```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

# Full-text Search

## Basic Text Matching

- Searching for phrases is done with the <-> (FOLLOWED BY) tsquery operator
- It matches only if its arguments have matches that are adjacent and in the given order. For example:



```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t
```

# Full-text Search

## Ranking Search Results

- Ranking attempts to measure how relevant documents are to a particular query.
- When there are many matches the most relevant ones can be shown first.
- PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information. That is,
  - they consider how often the query terms appear in the document,
  - how close together the terms are in the document
  - how important is the part of the document where they occur

## Full-text Search

### Ranking Search Results

- Different applications might require additional information for ranking.
- You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.
- The two ranking functions are:
  - ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
  - ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integ er ]) returns float4

The later computes cover density. Cover density is similar to ts_rank ranking except that the proximity of matching lexemes to each other is taken into consideration.

# Full-text Search

## Ranking Search Results / Weights

- For both these functions, the optional weights argument offers the ability to weigh word instances more or less heavily depending on how they are labeled.
- We may provide our own set of weights or use default weights.
- We provide weights in the ranking function but attatch the weight labels to tsvectors.

# Full-text Search

## Ranking Search Results / Normalization

- Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size.
  - e.g., a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances.
- Both ranking functions take an integer normalization option that specifies whether and how a document's length should impact its rank.
- The integer option controls several behaviors : we can specify one or more behaviors using | (eg: 2|4).
- If more than one flag bit is specified, the transformations are applied in the order listed.

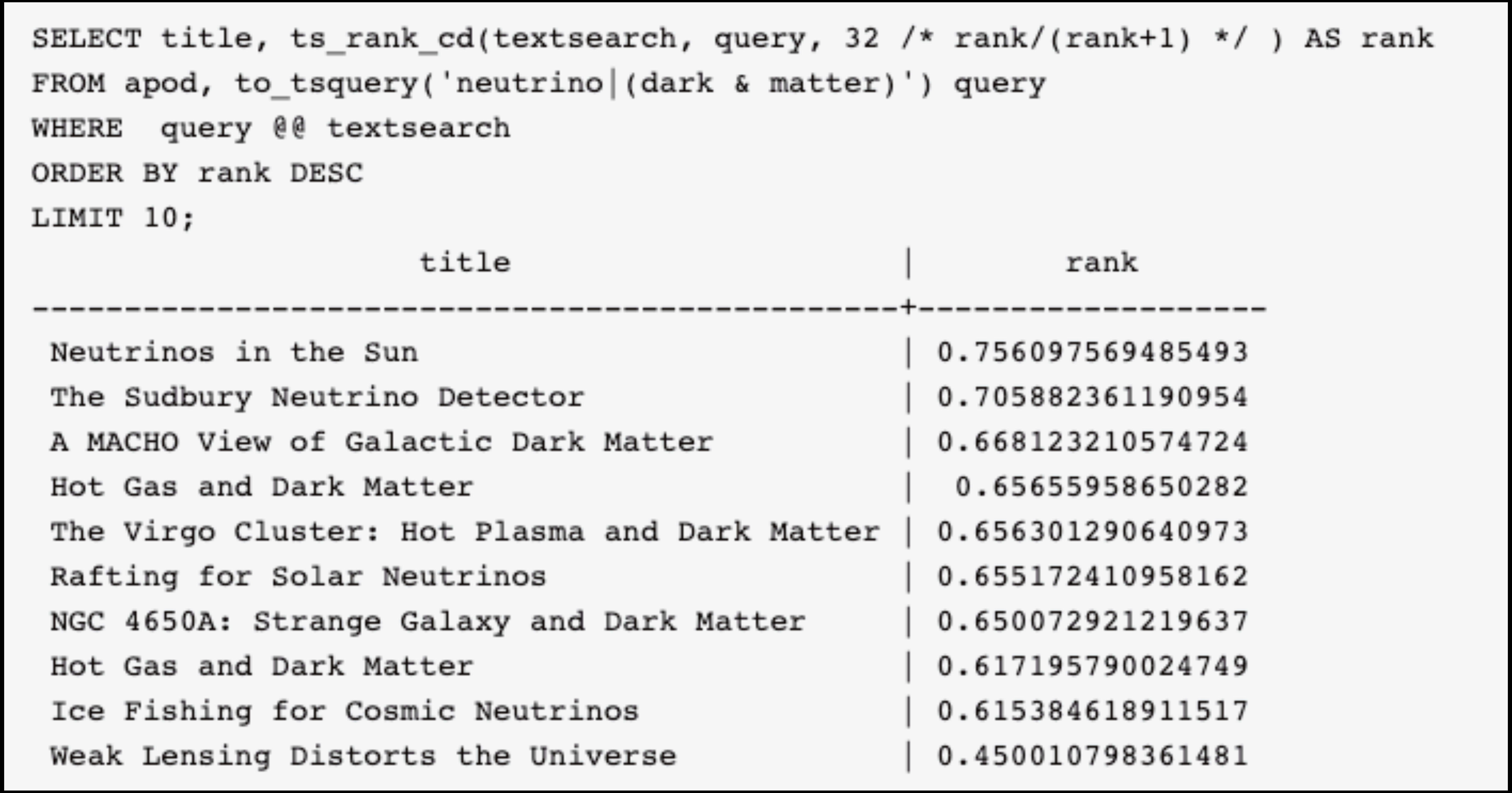
# Full-text Search

## Ranking Search Results / Normalization

- 0 (the default) ignores the document length
- 1 divides the rank by 1 + the logarithm of the document length
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by ts_rank_cd)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by 1 + the logarithm of the number of unique words in document
- 32 divides the rank by itself + 1

# Full-text Search

## Ranking Search Results / Normalization



```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                      |        rank
------------------------------------------------+-------------------
 Neutrinos in the Sun                           | 0.756097569485493
 The Sudbury Neutrino Detector                  | 0.705882361190954
 A MACHO View of Galactic Dark Matter           | 0.668123210574724
 Hot Gas and Dark Matter                        |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter  | 0.656301290640973
 Rafting for Solar Neutrinos                    | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter      | 0.650072921219637
 Hot Gas and Dark Matter                        | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos               | 0.615384618911517
 Weak Lensing Distorts the Universe             | 0.450010798361481
```

# Full-text Search

## Ranking Search Results / Highlighting Results

- To present search results it is ideal to show a part of each document and how it is related to the query.
- PostgreSQL provides a function ts_headline that implements this functionality.
- ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
- ts_headline accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted.
- We can specify various options to configure the result according to our needs.

# Full-text Search

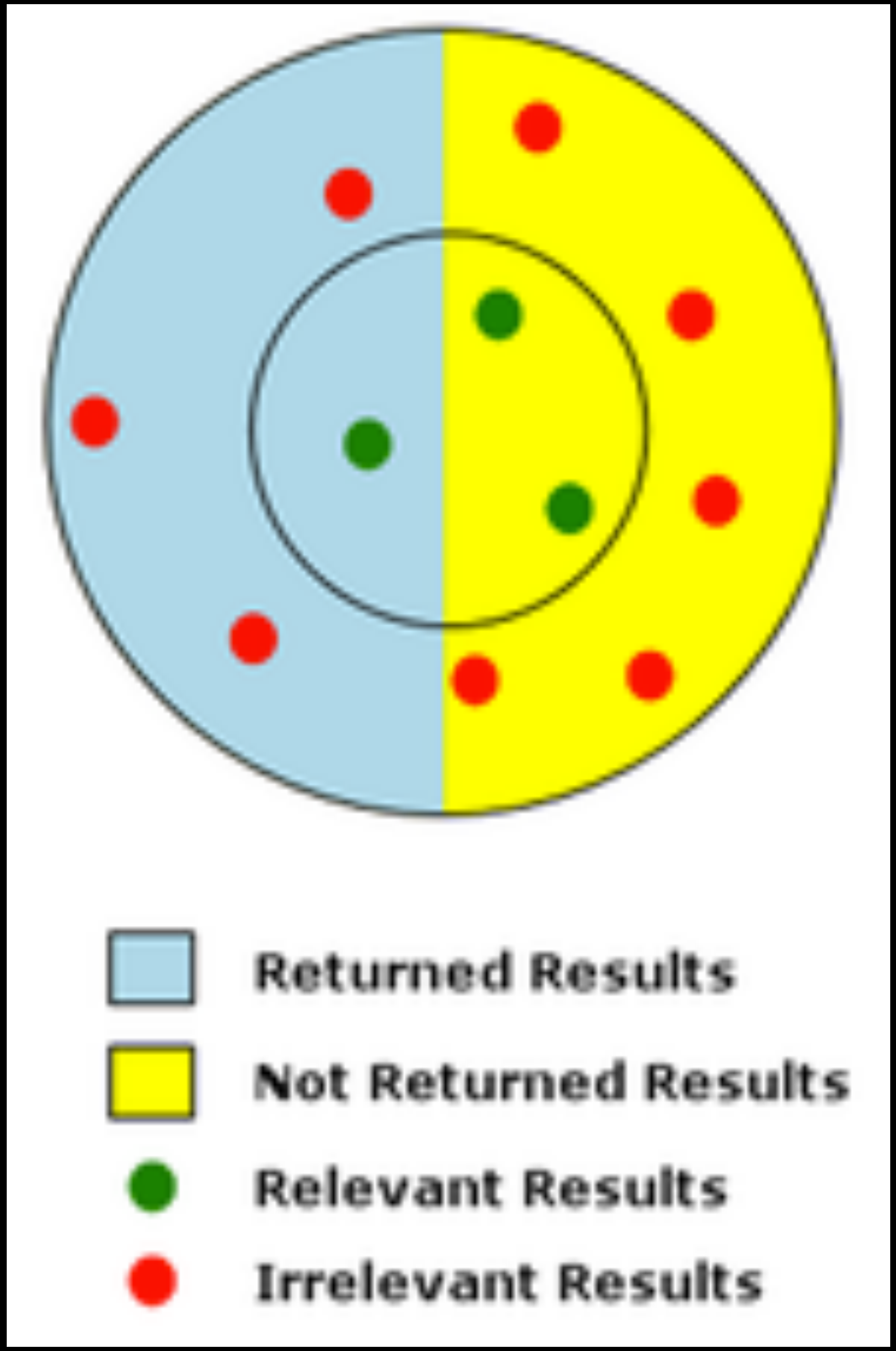
## Precision vs Recall Trade off

Recall measures the quantity of relevant results returned by a search, while precision is the measure of the quality of the results returned.

Recall is the ratio of relevant results returned to all relevant results. Precision is the number of relevant results returned to the total number of results returned.

The trade-off between precision and recall is simple: an increase in precision can lower overall recall, while an increase in recall lowers precision.

Full-text-search systems typically includes options like stop words to increase precision and stemming to increase recall.



Low precision, Low recall

## Full-text Search

### Limitations

- The length of each lexeme must be less than 2 kilobytes
- The length of a tsvector (lexemes + positions) must be less than 1 megabyte
- The number of lexemes must be less than 264
- Position values in tsvector must be greater than 0 and no more than 16,383
- The match distance in a <N> (FOLLOWED BY) tsquery operator cannot be more than 16,384
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operators) in a tsquery must be less than 32,768

# Django Search

## Full-text Search

- The easiest way to use full-text search in django is through the <field_name>__search method to search for a single term in a single column of the database, for example:
    - In [1]: Product.objects.filter(name__search='Shiny')
    - Out[2]: [<Product: Shiny Shoes>, <Product: Very Shiny LED>]
- This is useful for simple searches on single columns.
- For more complex queries Django provides various other functions.

# Django Search

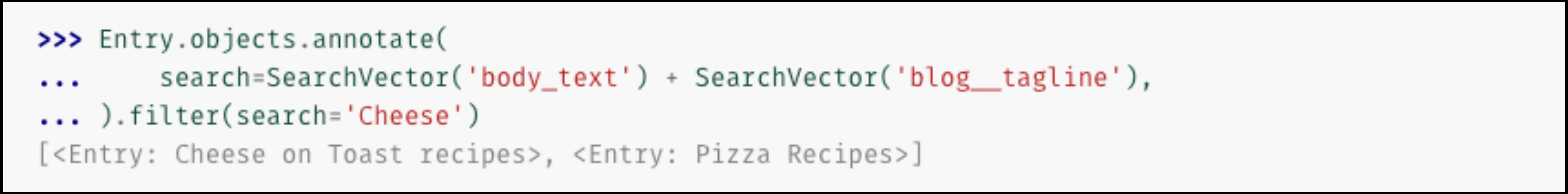
## Full-text Search

- The easiest way to use full-text search in django is through the <field_name>__search method to search for a single term in a single column of the database, for example:
    - In [1]: Product.objects.filter(name__search='Shiny')
    - Out[2]: [<Product: Shiny Shoes>, <Product: Very Shiny LED>]
- This is useful for simple searches on single columns.
- For more complex queries Django provides various other functions.

# Django Search

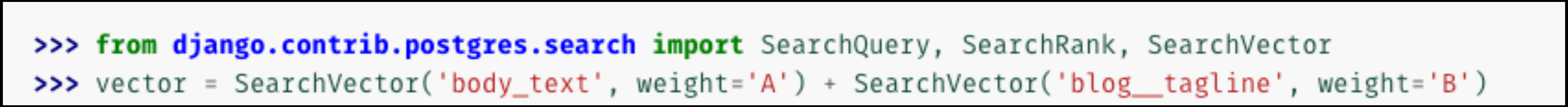
## Full-text Search / SearchVector

- To query multiple fields we use SearchVector.
- SearchVector objects can be combined together allowing for reuse.



```
>>> Entry.objects.annotate(
...     search=SearchVector('body_text') + SearchVector('blog__tagline'),
... ).filter(search='Cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza Recipes>]
```

- Every field may not have the same relevance in a query, so we can set weights of various vectors before you combine them:

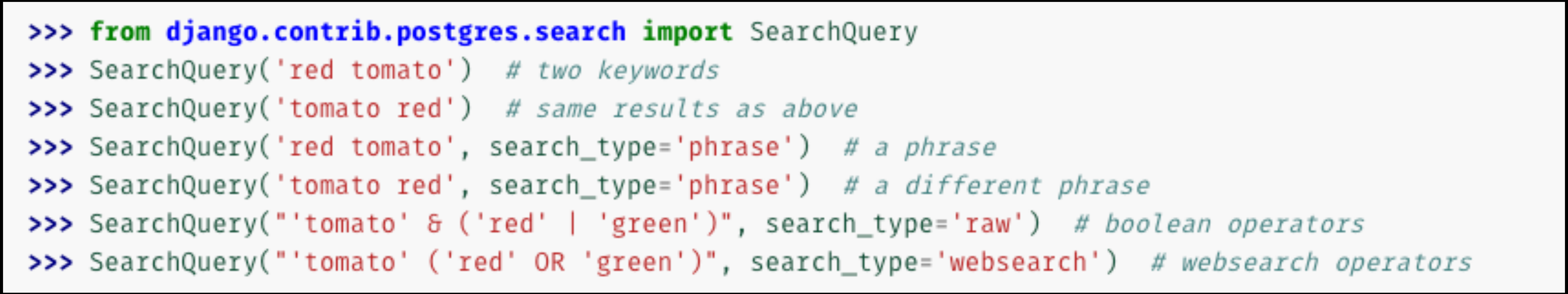


```
>>> from django.contrib.postgres.search import SearchQuery, SearchRank, SearchVector
>>> vector = SearchVector('body_text', weight='A') + SearchVector('blog__tagline', weight='B')
```

# Django Search

## Full-text Search / SearchQuery

- SearchQuery translates the terms the user provides into a search query object that the database compares to a search vector.



```
>>> from django.contrib.postgres.search import SearchQuery
>>> SearchQuery('red tomato')  # two keywords
>>> SearchQuery('tomato red')  # same results as above
>>> SearchQuery('red tomato', search_type='phrase')  # a phrase
>>> SearchQuery('tomato red', search_type='phrase')  # a different phrase
>>> SearchQuery("'tomato' & ('red' | 'green')", search_type='raw')  # boolean operators
>>> SearchQuery("'tomato' ('red' OR 'green')", search_type='websearch')  # websearch operators
```

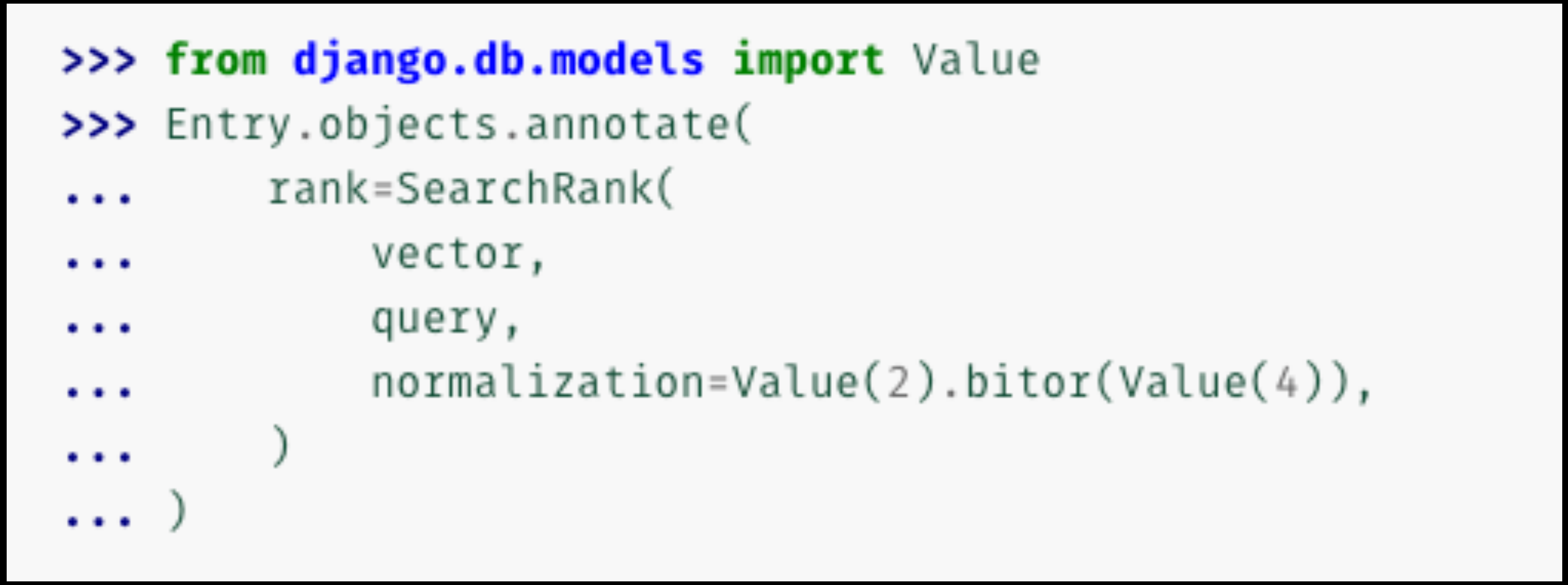
- We can also combine them logically to provide them flexibility.

```
>>> SearchQuery('meat') & SearchQuery('cheese')  # AND
```

# Django Search

## Full-text Search / SearchRank

- We may wish to order the results by some sort of relevancy.
- We use SearchRank to utilize PostgreSQL's ranking function.



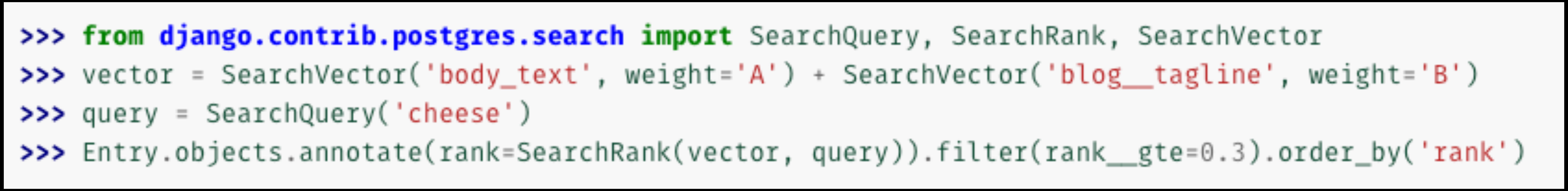
```
>>> from django.db.models import Value
>>> Entry.objects.annotate(
...     rank=SearchRank(
...         vector,
...         query,
...         normalization=Value(2).bitor(Value(4)),
...     )
... )
```

- We can also provide cover_density and normalization parameters

# Django Search

## Full-text Search / Weighting queries

- Every field may not have the same relevance in a query, so we can set weights of various vectors before we combine them.



```
>>> from django.contrib.postgres.search import SearchQuery, SearchRank, SearchVector
>>> vector = SearchVector('body_text', weight='A') + SearchVector('blog__tagline', weight='B')
>>> query = SearchQuery('cheese')
>>> Entry.objects.annotate(rank=SearchRank(vector, query)).filter(rank__gte=0.3).order_by('rank')
```

- The weight should be one of the following letters: D, C, B, A. By default, these weights refer to the numbers 0.1, 0.2, 0.4, and 1.0, respectively.  We may also pass our own weights as:
  - >>> rank = SearchRank(vector, query, weights=[0.2, 0.4, 0.6, 0.8])
  - >>> Entry.objects.annotate(rank=rank).filter(rank__gte=0.3).order_by('-rank')

# Django Search

## Full-text Search / Optimization

- Search may be one of the most resource-intensive operations, especially if it is a search in a few hundred or even thousands of records.
- SearchVectorField:
  - To perform full-text searches efficiently, a database must pre-process the data and summarize it as search vectors.
  - Django 1.10 introduced the SearchVectorField model field to save the search vector in this column, which will be converted to TSVECTOR
- Indexing:
  - GIN Index ( Generalized Inverted Index)
  - GIST Index (Generalized Search Tree)

## References

- https://www.netlandish.com/blog/2020/06/22/full-text-search-django-postgresql/
- https://www.postgresql.org/docs/13/textsearch.html
- https://www.postgresql.org/docs/current/textsearch-intro.html#TEXTSEARCH-DOCUMENT
- https://docs.djangoproject.com/en/3.1/ref/contrib/postgres/search/
- https://en.wikipedia.org/wiki/Full-text_search
- https://findwork.dev/blog/optimizing-postgres-full-text-search-in-django/