Arvados - Feature #14573

[Spike] [API] Fully functional filename search

12/03/2018 07:13 PM - Tom Clegg

Status:	Resolved	**Start date:**	
Priority:	Normal	**Due date:**	
Assigned To:	Peter Amstutz	**% Done:**	0%
Category:	API	**Estimated time:**	0.00 hour
Target version:	2019-04-24 Sprint		

Description

See #13752, #14560 for previous attempts.

- Indexing on text fields cannot handle medium-size text inputs.
- Indexing on to_tsvector(...) cannot handle certain large text inputs (limit depends on content, not just size). Result: crash when creating the index or when inserting a row, whichever happens last.

Approaches that have been considered:

- Add a tsvector column. Populate it with to_tsvector(...) where possible. Where not possible, either populate with partial content (to_tsvector(substring(...))), or leave it null and adjust the search query to do an unindexed fulltext search on such rows. A function with an exception clause might work.
- Use something other than Postgresql for text search.
- Index of files in collections

Spike goal: validate that the Index of files in collections approach can return the desired results, and performs well on a production-size database.

Suggested implementation:

- Retrieve all collections from a production-size cluster, extract the pdh/dir/file/size info, and insert into a table on a dev database.
- Try various ways of indexing/reformatting the dir/filenames so the example searches run quickly and return useful results.
- Provide table of speeds/results for various approaches.

Related issues:

Related to Arvados - Feature #15106: [API] Index 'like' queries and use for s...	**Resolved**	**06/14/2019**
Has duplicate Arvados - Story #13508: Fix postgres search for filenames	**Duplicate**	
Has duplicate Arvados - Story #14611: [Epic] Site-wide search for text, filen...	**Duplicate**	

History

#1 - 12/03/2018 07:13 PM - Tom Clegg

- Tracker changed from Bug to Feature

#2 - 12/03/2018 07:44 PM - Tom Clegg

- Description updated

#3 - 12/06/2018 04:05 PM - Peter Amstutz

Postgres text search has other problems when it comes to segmenting filenames. However I don't think that means we give up postgres, should create our own filename search table that has the behavior we want.

https://dev.arvados.org/issues/13752#note-7

Proposed solution:

Maintain our own filename index in a new table.

keyword → collection PDH

Perform custom filename tokenizing and support prefix search with "like" (can use B-tree indexes). Split on symbols like "_", "-" and ".", CamelCase (lower&arr;upper transitions). Convert everything to lowercase. For example:

"Sample_RMF1U7F_S27_R1_001.fastq.gz"

Would turn into:

"sample_rmf1u7f_s27_r1_001.fastq.gz"
"rmf1u7f_s27_r1_001.fastq.gz"
"s27_r1_001.fastq.gz"
"r1_001.fastq.gz"
"001.fastq.gz"
"fastq.gz"
"gz"

Searches would be prefix searches, eg a search on "RMF1U7F" would be keyword like 'rmf1u7f%'

#4 - 02/20/2019 03:10 PM - Peter Amstutz

Table of:

(filename, portable data hash of collection, path in collection, file size, content hash)

#5 - 02/20/2019 08:20 PM - Tom Clegg

- Description updated

#6 - 02/20/2019 08:30 PM - Tom Clegg

- Subject changed from [API] Fully functional filename search to [Spike] [API] Fully functional filename search

- Description updated

- Target version changed from To Be Groomed to Arvados Future Sprints

- Story points set to 2.0

#7 - 02/20/2019 08:30 PM - Tom Clegg

- Has duplicate Story #13508: Fix postgres search for filenames added

#8 - 02/20/2019 08:31 PM - Tom Clegg

- Has duplicate Story #14611: [Epic] Site-wide search for text, filenames, data added

#9 - 03/13/2019 03:45 PM - Tom Morris

- Target version changed from Arvados Future Sprints to 2019-03-27 Sprint

#10 - 03/13/2019 03:45 PM - Lucas Di Pentima

- Assigned To set to Lucas Di Pentima

#11 - 03/27/2019 03:14 PM - Lucas Di Pentima

- Target version changed from 2019-03-27 Sprint to 2019-04-10 Sprint

#12 - 03/27/2019 03:16 PM - Peter Amstutz

- Assigned To changed from Lucas Di Pentima to Peter Amstutz

#13 - 03/28/2019 10:45 PM - Peter Amstutz

Strategy 1:

Create two tables:

```
create table filenames (pdh text, filename text);
```

```
create table filetokens (filename text, token text);
```

The "filenames" table has a PDH and each file path contained in the manifest.

The "filetokens" table has a full file path and substring starting from each break point (such as punctuation).

Next, create indexes:

<Pre>
CREATE INDEX filenames_idx ON filenames (filename text_pattern_ops);

CREATE INDEX filetokens_idx ON filetokens (token text_pattern_ops);

Now we can search filenames by doing a prefix search on the filetokens table (this is efficient with the "text_pattern_ops" index) and joining with the "filenames" table to get the associated manifest PDHs.

```
explain analyze select pdh, filenames.filename from filetokens inner join filenames on filenames.filename=file
tokens.filename where token like 'hu589D0B%'
                                                               QUERY PLAN

-----------------------------------------------------------------------------------------------------------
--------------------------
 Nested Loop  (cost=1.12..146749.41 rows=103706 width=101) (actual time=0.254..118.289 rows=16541 loops=1)
   ->  Index Scan using filetokens_idx on filetokens  (cost=0.56..8.58 rows=2290 width=68) (actual time=0.209.
.25.212 rows=5599 loops=1)
         Index Cond: ((token ~>=~ 'hu589D0B'::text) AND (token ~<~ 'hu589D0C'::text))
         Filter: (token ~~ 'hu589D0B%'::text)
   ->  Index Scan using filenames_idx on filenames  (cost=0.56..63.85 rows=23 width=101) (actual time=0.014..0
.016 rows=3 loops=5599)
         Index Cond: (filename = filetokens.filename)
 Planning time: 1.489 ms
 Execution time: 119.139 ms
(8 rows)
```

It took 119ms but it turns out that's because there's a lot of files with 'hu589D0C':

```
arvados_development=> select count(*) from filetokens inner join filenames on filenames.filename=filetokens.fi
lename where token like 'hu589D0B%'
;
 count
-------
 16541
```

We can isolate it to distinct manifest:

```
select count(distinct pdh) from filetokens inner join filenames on filenames.filename=filetokens.filename wher
e token like 'hu589D0B%';
 count
-------
    36
```

Getting unique manifests is a bit quicker compared to getting the whole list of files (note there's no index on the pdh column, I don't know if that would help):

```
explain analyze select count(distinct pdh) from filetokens inner join filenames on filenames.filename=filetoke
ns.filename where token like 'hu589D0B%';

 Aggregate  (cost=147008.67..147008.68 rows=1 width=8) (actual time=155.506..155.507 rows=1 loops=1)
   ->  Nested Loop  (cost=1.12..146749.41 rows=103706 width=40) (actual time=0.027..145.026 rows=16541 loops=1
)
         ->  Index Scan using filetokens_idx on filetokens  (cost=0.56..8.58 rows=2290 width=68) (actual time=
0.012..7.127 rows=5599 loops=1)
               Index Cond: ((token ~>=~ 'hu589D0B'::text) AND (token ~<~ 'hu589D0C'::text))
               Filter: (token ~~ 'hu589D0B%'::text)
         ->  Index Scan using filenames_idx on filenames  (cost=0.56..63.85 rows=23 width=101) (actual time=0.
021..0.023 rows=3 loops=5599)
               Index Cond: (filename = filetokens.filename)
 Planning time: 0.715 ms
 Execution time: 155.543 ms
```

If we limit the number of rows returned, the query is much faster:

```
arvados_development=> explain analyze select pdh, filenames.filename from filetokens inner join filenames on f
ilenames.filename=filetokens.filename where token like 'hu589D0B%' limit 50
;
                                                                 QUERY PLAN

-----------------------------------------------------------------------------------------------------------
------------------------------
 Limit  (cost=1.12..71.87 rows=50 width=101) (actual time=0.039..0.250 rows=50 loops=1)
   ->  Nested Loop  (cost=1.12..146749.41 rows=103706 width=101) (actual time=0.038..0.246 rows=50 loops=1)
         ->  Index Scan using filetokens_idx on filetokens  (cost=0.56..8.58 rows=2290 width=68) (actual time=
0.016..0.033 rows=13 loops=1)
               Index Cond: ((token ~>=~ 'hu589D0B'::text) AND (token ~<~ 'hu589D0C'::text))
               Filter: (token ~~ 'hu589D0B%'::text)
         ->  Index Scan using filenames_idx on filenames  (cost=0.56..63.85 rows=23 width=101) (actual time=0.
012..0.015 rows=4 loops=13)
               Index Cond: (filename = filetokens.filename)
```

```
 Planning time: 1.144 ms
 Execution time: 0.280 ms
```

One drawback of this approach is that the "tokens" table gets pretty large, with about 60% of the collections on qr1hi loaded there's about 29 million tokens:

```
select count(*) from filetokens;
  count
----------
 29426500
```

Also, although our custom tokenizing logic is much better suited to filenames than full text search, it could still happen that someone wants to search for a substring that doesn't start at one of our chosen token break points.

#14 - 03/28/2019 10:45 PM - Peter Amstutz

See https://stackoverflow.com/questions/1566717/postgresql-like-query-performance-variations

#15 - 03/28/2019 10:53 PM - Peter Amstutz

Based on the SO post another thing I'm going to look at is an index using postgres trigram operators:

https://www.postgresql.org/docs/current/pgtrgm.html

#16 - 03/28/2019 11:04 PM - Peter Amstutz

I feel like there might be other tricks with operator indexes that could accomplish what we want as well, but that will require more research.

#17 - 03/29/2019 02:21 PM - Peter Amstutz

install postgresql-contrib

create extension pg_trgm;

CREATE INDEX filenames_trgm_idx ON public.filenames USING gin (filename public.gin_trgm_ops);

```
arvados_development=# explain analyze select * from filenames where filename like '%hu589D0B%';
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------
-------------------
 Bitmap Heap Scan on filenames  (cost=77.82..971.51 rows=235 width=101) (actual time=3.495..6.328 rows=5585 lo
ops=1)
   Recheck Cond: (filename ~~ '%hu589D0B%'::text)
   Heap Blocks: exact=157
   ->  Bitmap Index Scan on filenames_trgm_idx  (cost=0.00..77.76 rows=235 width=0) (actual time=3.443..3.443
rows=5585 loops=1)
         Index Cond: (filename ~~ '%hu589D0B%'::text)
 Planning time: 0.274 ms
 Execution time: 6.731 ms
(7 rows)
```

Searching on filenames with the trigram index:

```
explain analyze select filename, pdh from filenames where filename ilike '%hu589D0B%';
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------
-------------------
 Bitmap Heap Scan on filenames  (cost=77.82..971.51 rows=235 width=101) (actual time=3.485..9.892 rows=5585 lo
ops=1)
   Recheck Cond: (filename ~~* '%hu589D0B%'::text)
   Heap Blocks: exact=157
   ->  Bitmap Index Scan on filenames_trgm_idx  (cost=0.00..77.76 rows=235 width=0) (actual time=3.460..3.460
rows=5585 loops=1)
         Index Cond: (filename ~~* '%hu589D0B%'::text)
 Planning time: 0.869 ms
 Execution time: 10.121 ms
(7 rows)
```

This is faster than the token+join method.

```
explain analyze select filenames.filename, pdh from filetokens inner join filenames on filenames.filename=file
tokens.filename where token like 'hu589D0B%';
                                                                 QUERY PLAN
```

```
------------------------------------------------------------------------------------------------------------------------
--------------------------
 Nested Loop  (cost=1.12..150551.94 rows=104061 width=101) (actual time=0.049..70.267 rows=16541 loops=1)
   ->  Index Scan using filetokens_idx on filetokens  (cost=0.56..8.58 rows=2290 width=68) (actual time=0.020.
.4.339 rows=5599 loops=1)
         Index Cond: ((token ~>=~ 'hu589D0B'::text) AND (token ~<~ 'hu589D0C'::text))
         Filter: (token ~~ 'hu589D0B%'::text)
   ->  Index Scan using filenames_idx on filenames  (cost=0.56..65.50 rows=24 width=101) (actual time=0.010..0
.011 rows=3 loops=5599)
         Index Cond: (filename = filetokens.filename)
 Planning time: 1.462 ms
 Execution time: 70.985 ms
(8 rows)
```

In fact it's worse than that, the token+join method returns duplicate results, so it needs select distinct:

```
select distinct(filenames.filename, pdh) from filetokens inner join filenames on filenames.filename=filetokens
.filename where token like 'hu589D0B%' order by (filenames.filename, pdh);
                                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
--------------------------------------
 Unique  (cost=161715.90..162236.20 rows=104061 width=32) (actual time=242.187..248.489 rows=5585 loops=1)
   ->  Sort  (cost=161715.90..161976.05 rows=104061 width=32) (actual time=242.187..243.733 rows=16541 loops=1
)
         Sort Key: (ROW(filenames.filename, filenames.pdh))
         Sort Method: external merge  Disk: 1976kB
         ->  Nested Loop  (cost=1.12..150551.94 rows=104061 width=32) (actual time=0.083..68.332 rows=16541 lo
ops=1)
               ->  Index Scan using filetokens_idx on filetokens  (cost=0.56..8.58 rows=2290 width=68) (actual
 time=0.031..4.088 rows=5599 loops=1)
                     Index Cond: ((token ~>=~ 'hu589D0B'::text) AND (token ~<~ 'hu589D0C'::text))
                     Filter: (token ~~ 'hu589D0B%'::text)
               ->  Index Scan using filenames_idx on filenames  (cost=0.56..65.50 rows=24 width=101) (actual t
ime=0.009..0.010 rows=3 loops=5599)
                     Index Cond: (filename = filetokens.filename)
 Planning time: 2.323 ms
 Execution time: 249.045 ms
```

So the results lean strongly towards the trigram index

#18 - 03/29/2019 02:39 PM - Peter Amstutz

IT looks like the simplest solution for filename search, would not even require introducing any new tables, is:

1. drop the fulltext index from collections.file_names
2. add a trigram GIN index on collections.file_names
3. update clients to use ilike '%blah%' to do filename searches *OR* adjust the behavior of our '@@' operator to be implemented by either fulltext or ilike searches depending on the record type.

#19 - 03/29/2019 03:13 PM - Peter Amstutz

One benefit of a separate "files" table (as described in the original story description) is that the extra file count and file size columns being added in #14484 become unnecessary, they could be easily implemented using count(*) and sum(filesize) SQL aggregates.

#20 - 04/03/2019 02:37 PM - Peter Amstutz

Whatever query workbench uses, API need to transform into query appropriate for filename search.

#21 - 04/05/2019 05:56 PM - Peter Amstutz

The e51c5 database has 4 million collections. The largest collection has a filename list just under 2 MB in size. With a seq scan, it takes 15 seconds to find a substring. With a trigram index, it takes 11ms.

```
arvados_development=> select count(*) from collections;
  count
---------
 4228696

arvados_development=> select max(length(file_names)) from collections;
   max
---------
 1962256
```

```
arvados_development=> select portable_data_hash from collections where file_names ilike '%AJ5LM4NX7CA%';
Time: 14399.272 ms

arvados_development=> CREATE INDEX collection_filenames_trgm_idx ON collections USING gin (file_names public.g
in_trgm_ops);
CREATE INDEX
Time: 158394.288 ms

arvados_development=> select portable_data_hash from collections where file_names ilike '%AJ5LM4NX7CA%';
Time: 11.429 ms

arvados_development=> select portable_data_hash from collections where file_names ilike '%Sample_AJ5LM4NX7CA%'
;
Time: 29.370 ms

arvados_development=> select count(uuid) from collections where file_names ilike '%Sample_%';
 count
--------
 724071
(1 row)

Time: 9363.349 ms

arvados_development=> select portable_data_hash from collections where file_names ilike '%AJ5LM4NX7CA%' limit
1000;
Time: 4.203 ms
```

I notice the query time seems to be affected by uniqueness of the substring, so searching for "Sample_AJ5LM4NX7CA" (where "Sample_" is an incredibly common substring, appearing in 724071 rows) is slower than searching for just the sample id "AJ5LM4NX7CA". However with a limit on the number of rows to return, the query times are only in the double digits of milliseconds -- either the query is relatively unique and only returns a few rows (and so the index lookup is very quick), or there are many hits, and it hits the limit very quickly (using "offset" to page through the results is a bit slower, but only when there are 10s of thousands of results).

#22 - 04/05/2019 06:17 PM - Tom Morris

- Status changed from New to In Progress

Looks very promising! Thanks for the data. A couple of additional questions:

- What's the size of the index?
- How does that compare to the size of the indexed text (sum(length(file_names)))?
- Does the file_names column include everything or is it capped at some limit? (Already answered in chat - Everything)

It looks to me like this is a good approach, but let's also get the rest of the team to review. We can either do that in the context of grooming an implementation ticket or just do it informally and roll this over into an implementation ticket.

#23 - 04/05/2019 06:35 PM - Peter Amstutz

```
arvados_development=> select pg_size_pretty(pg_total_relation_size('collections'));
 pg_size_pretty
----------------
 16 GB

arvados_development=> select pg_size_pretty(sum(length(manifest_text))) from collections;
 pg_size_pretty
----------------
 8294 MB

arvados_development=> select pg_size_pretty(sum(length(file_names))) from collections;
 pg_size_pretty
----------------
 743 MB

arvados_development=> select pg_size_pretty(pg_relation_size('collection_filenames_trgm_idx'));
 pg_size_pretty
----------------
 368 MB
```

So, the collections table is 16 GB, half of which is manifest_text, the file_names are about 10% the size of manifest_text, and trigram index is about half the size of file_names.

#24 - 04/05/2019 07:48 PM - Peter Amstutz

```
arvados_development=> select portable_data_hash from collections where manifest_text ilike '%AJ5LM4NX7CA%';
Time: 120136.877 ms
```

```
arvados_development=> CREATE INDEX collection_manifest_trgm_idx ON collections USING gin (manifest_text public
.gin_trgm_ops);
CREATE INDEX
Time: 2100488.609 ms

arvados_development=> select portable_data_hash from collections where manifest_text ilike '%AJ5LM4NX7CA%';
Time: 41.068 ms

arvados_development=> select pg_size_pretty(pg_total_relation_size('collections'));
 pg_size_pretty
----------------
 19 GB

arvados_development=> select pg_size_pretty(pg_relation_size('collection_manifest_trgm_idx'));
 pg_size_pretty
----------------
 3191 MB

arvados_development=> select count(uuid) from collections where manifest_text ilike '%Sample_%';
 count
--------
 724071
(1 row)

Time: 85809.407 ms
```

As an experiment, I tried adding a trigram index on the manifest_text column. Building the index takes a very long time. The index itself is pretty large (seems roughly linear on the total amount of data in the column). Queries are a bit slower.

However it looks like having a file_names column + index adds roughly 1 GB compared to 3 GB for an index on manifest_text.

So for filename search, this is strictly worse. Although, it would add the capability to look up which manifests a block hash appears in.

#25 - 04/05/2019 08:17 PM - Tom Clegg

Looks very promising.

It looks like this is available in the postgresql-contrib package in debian 9. Would be good to confirm availability & procedure needed to enable it (if any) on all supported platforms.

I expect we will want something like "search all columns for xyz", not just "search file_names for xyz". I'm guessing we can use trgm to index all searchable columns, including jsonb, so ["any","like","%foo%"] would be fast, and could be used to find collection filenames, container mount filenames, etc...? If this works, could Workbench skip the full-text search entirely, and avoid the trouble of merging results from multiple search queries?

The current (FT) search implementation interprets "foo bar" as "foo & bar". Does this translate to [["any","like","%foo bar%"]] here, or [["any","like","%foo%"],["any","like","%bar%"]] -- and would those queries perform well too?

#26 - 04/08/2019 02:07 PM - Tom Clegg

It would be good to confirm the search is functional (i.e., returns good results for user queries) in addition to being fast. I'm not sure of the best way to do this, or how rigorous we want to get, but it seems like we should check how well the results match for some obvious/typical queries on filenames as well as other kinds of fields that get searched.

#27 - 04/10/2019 03:04 PM - Peter Amstutz

- Target version changed from 2019-04-10 Sprint to 2019-04-24 Sprint

#28 - 04/16/2019 05:03 PM - Peter Amstutz

> It looks like this is available in the postgresql-contrib package in debian 9. Would be good to confirm availability & procedure needed to enable it (if any) on all supported platforms.

Postgres trigram index has existed for a while, however support specifically for accelerating 'LIKE' queries may have only been introduced in 9.1 (that's when it appears in the documentation).

Debian has the postgresql-contrib going back to Jessie (Debian 8) with Postgres 9.4
https://packages.debian.org/search?keywords=postgresql-contrib&searchon=names&suite=all&section=all

Ubuntu has the postgresql-contrib package going back to trusty (14.04) with Postgres 9.3
https://packages.ubuntu.com/search?keywords=postgresql-contrib&searchon=all&suite=all&section=all

Centos7 packages available at postgresql.org include postgresql-contrib https://www.postgresql.org/download/linux/redhat/

#29 - 04/16/2019 05:59 PM - Peter Amstutz

```
select uuid from collections where file_names ilike '%foo%';
Time: 48.968 ms

select uuid from collections where file_names ilike '%foo%' and file_names ilike '%bar%';
Time: 53.635 ms

CREATE INDEX properties_trgm_idx ON public.collections USING gin (properties public.gin_trgm_ops);
ERROR:  operator class "public.gin_trgm_ops" does not accept data type jsonb

CREATE INDEX properties_trgm_idx ON public.collections USING gin ((properties::text) public.gin_trgm_ops);
CREATE INDEX
Time: 66620.780 ms

arvados_development=> select properties from collections where properties::text ilike '%
e51c5-xvhdp-p48azvvjrbhmbf7%';
                              properties
-----------------------------------------------------------------------
 {"type": "log", "container_request": "e51c5-xvhdp-p48azvvjrbhmbf7"}
 {"type": "output", "container_request": "e51c5-xvhdp-p48azvvjrbhmbf7"}
(2 rows)

Time: 67.937 ms

arvados_development=> select properties from collections where properties::text ilike '%foo%';
Time: 6.889 ms
```

#30 - 04/16/2019 06:48 PM - Peter Amstutz

> I expect we will want something like "search all columns for xyz", not just "search file_names for xyz". I'm guessing we can use trgm to index all searchable columns, including jsonb, so ["any","like","%foo%"] would be fast, and could be used to find collection filenames, container mount filenames, etc...? If this works, could Workbench skip the full-text search entirely, and avoid the trouble of merging results from multiple search queries?

If we take the approach of changing the implementation of '@', since queries are only allowed to be in the form [["any", "@", "foo bar"]] then we can construct the search based on each column type and mix both 'ilike' and '@@' in the 'where' clause (so no merging results problem).

In addition to using ilike against individual columns, we can use ilike on a multi-column expression index (same approach used for full text):

```
CREATE INDEX merged_trgm_idx ON public.collections USING gin ((file_names || ' ' || properties::text) public.g
in_trgm_ops);
CREATE INDEX
Time: 175172.242 ms

select file_names, properties from collections where (file_names || ' ' || properties::text) ilike '%176DEMH%'
;
Time: 7.256 ms
```

Using full text indexing for *some* columns would mean continuing to support language-based parsing/stemming/normalization/stop words/etc. The main situation I see where this would be useful would be in searching description fields that contain substantial prose and the user doesn't know exactly what she is looking for (ranking by full text score would be useful in that case, but we don't support that). With 'ilike' you would be able to find exact substring matches in descriptions, but not words that differ by changes in tense, pluralization, etc. For names/titles you might actually want **both** indexing strategies to maximize the chances of the search properly capturing user intent.

This feels like a product feature question. Do we continue to support full text search on certain columns (at the cost of slightly greater implementation complexity), or switch exclusively to substring search?

> The current (FT) search implementation interprets "foo bar" as "foo & bar". Does this translate to ["any","like","%foo bar%"] here, or [["any","like","%foo%"],["any","like","%bar%"]] -- and would those queries perform well too?

The splitting and joining with '&' happens in Ruby. So instead of turning "foo bar" into "foo & bar" it would have turn it into [["any","like","%foo%"],["any","like","%bar%"]] to get the same results.

#31 - 04/16/2019 07:03 PM - Tom Clegg

Peter Amstutz wrote:

> The splitting and joining with '&' happens in Ruby. So instead of turning "foo bar" into "foo & bar" it would have turn it into

> [["any","like","%foo%"],["any","like","%bar%"]] to get the same results.

Currently we implement filters=[["any","@@","foo bar"]] by doing where [...] @@ 'foo & bar'.

The trigram docs have examples like where t like '%foo%bar' and where t ~ '(foo|bar)' ... so I thought maybe one of those could avoid expanding to where ... like '%foo%' and ... like '%bar%'. Either way I thought we might want to confirm performance & results are still promising when the query has more than one word.

#32 - 04/16/2019 07:20 PM - Peter Amstutz

Tom Clegg wrote:

> Currently we implement filters=[["any","@@","foo bar"]] by doing where [...] @@ 'foo & bar'.

Specifically:

```
        # Use to_tsquery since plainto_tsquery does not support prefix
        # search. And, split operand and join the words with ' & '
        cond_out << model_class.full_text_tsvector+" @@ to_tsquery(?)"
        param_out << operand.split.join(' & ')
```

My point is just that splitting and inserting ' & ' happens in the API server not the client, so we don't have to interpret '&' it is just an implementation detail (I think we're in violent agreement).

> The trigram docs have examples like where t like '%foo%bar' and where t ~ '(foo|bar)' ... so I thought maybe one of those could avoid expanding to where ... like '%foo%' and ... like '%bar%'. Either way I thought we might want to confirm performance & results are still promising when the query has more than one word.

```
select uuid from collections where file_names ilike '%foo%bar%' or file_names ilike '%bar%foo%';
35.978 ms

select uuid from collections where file_names ilike '%foo%' and file_names ilike '%bar%';
Time: 56.192 ms

select uuid from collections where file_names ~ '(foo|bar)';
Time: 1208.119 ms
```

'~' is the regular expression operator.

Whoops, that last one is different, it is "contains foo OR bar" which is not the same.

However I don't think regular expressions have any advantage over ilike, we still end up having to do something like this:

```
 select uuid from collections where file_names ~ 'foo.*bar' or file_names ~ 'bar.*foo';
Time: 34.146 ms
```

#33 - 04/16/2019 07:27 PM - Tom Clegg

OK. I was just thinking that if a trigram index works by counting common trigrams, these queries might all return the same results.

```
select uuid from collections where file_names ilike '%foo%bar%';
select uuid from collections where file_names ilike '%bar%foo%';
select uuid from collections where file_names ilike '%foo%bar%' or file_names ilike '%bar%foo%';
```

#34 - 04/16/2019 07:38 PM - Peter Amstutz

Tom Clegg wrote:

> OK. I was just thinking that if a trigram index works by counting common trigrams, these queries might all return the same results.
>
> select uuid from collections where file_names ilike '%foo%bar%';
> select uuid from collections where file_names ilike '%bar%foo%';
> select uuid from collections where file_names ilike '%foo%bar%' or file_names ilike '%bar%foo%';

No? If the 1st and 2nd statements returned the same results that would violate the semantics of 'ilike'.

There are "near match" operators for trigram but we haven't been talking about those.

The following statements return the same results, however the 1st one is slightly more efficient (at least in this case). But searching for all permutations of term ordering could get ugly when there's more than 3 or 4 terms.

```
select uuid from collections where file_names ilike '%foo%bar%' or file_names ilike '%bar%foo%';
35.978 ms

select uuid from collections where file_names ilike '%foo%' and file_names ilike '%bar%';
Time: 56.192 ms
```

I'm speculating but I believe the index works something like:

1. break the search term into the set of trigrams
2. search for each trigram in the index to get a set of potential matches
3. perform the exact match test on each candidate match to confirm or reject it

#35 - 04/17/2019 03:27 PM - Peter Amstutz

- Related to Feature #15106: [API] Index 'like' queries and use for search added

#36 - 04/19/2019 07:45 PM - Peter Amstutz

- Status changed from In Progress to Resolved

#37 - 05/21/2019 10:27 PM - Tom Morris

- Release set to 15