SeSQL asynchronous reindexation

Contents

1 Overview

Sometimes, you need to alter SeSQL configuration. You want to change the stopwords list or stemming options. You want to add a new index column. You want to change the way the tables are partitioned.

If your base is not in production, or if it's small and you can tolerate a temporary downtime on search features, you can just drop the SeSQL table, do a `syncdb`, and run `sesqlreindex`. But if your base is big and you can't tolerate a downtime of several minutes, even less of several hours, this is not a solution.

Since version 0.14, SeSQL provides a command to perform those operations in a non-intrusive (well, only slightly intrusive), way. Nonetheless, be careful to read the documentation fully before attempting it, and we strongly advise you to run it in a test server before doing it live.

The idea is that you'll create a new `sesql_config` file elsewhere, using different table names in it. Then you'll run the script, which will reindex, slowly (at a configurable rate) the content into the new tables. And then, you'll switch the production code to the new config, using the new tables.

If you fulfill all the requirements (see limitations below), The script is devised to ease the operation as much as possible :

1. It can be interrupted, and will restart work where it was.

2. It'll handle the content that was changed or added while it was running.

3. It can work slowly enough to not significantly lower the performances of the production. But it's up to you to tune the speed parameters depending of the volume of your base and the available resources.

2 Usage

Here is the typical usage of the script :

1. Make a new `sesql_config` file, with your new indexing rules. Be careful to use different table names in the `TYPE_MAP`.

2. Run the script once, to index the content into the new config. Use a state file so you can restart where it stopped in case of crash of manual interruption. This will be done with a command like :

```
./manage.py sesqlasyncreindex \
--step=1000 --delay=5 \
--state=/path/to/statefile.pickle \
--config=/path/to/new/sesql_config.py \
--datefield=indexed_at
```

3. When the script is finished, stop your `sesql_update` daemon (if any), and your application; switch it to the new configuration file, and restart them.

4. To handle the content that may have been changed/added between the end of the script and the restart of your application, run the script in reverse mode (with the **same** state file, but providing it the old `sesql_config` file) :

```
./manage.py sesqlasyncreindex \
--step=1000 --delay=5 \
--state=/path/to/statefile.pickle \
--config=/path/to/old/sesql_config.py \
--datefield=indexed_at --reverse
```

A more detailed list of options can be found using `--help`.

3 Limitations

In order to work smoothly, this script has several prerequisites :

1. You need enough available disk space.

2. You need a `DateTimeField` index in your SeSQL model, if possible an indexation date defined like :

```
DateTimeField('indexed_at', sql_default = 'NOW()')
```

   If you use another date field, from your business logic, then there is no warranty that related content will be fully indexed. A safe workaround is to disable the *sesql_update*‘ daemon while the script is running, if you can afford that.

3. It’ll not handle content that was not indexed before (mapped to `None` in the configuration, or discarded by the `SKIP_CONDITION`), but should be indexed now. To index them, use the classical `sesqlreindex` command.

4. Since it uses dates to detect content to reindex, if the chunk size is too small, it may loop endlessly reindexing the same contents. The chunk size should be big enough to ensure that there is never more items that have the exactly same indexation date (to the second) than the chunk size. You can check that with a manual SQL command like :

   ```
   SELECT count(*), indexed_at
   FROM sesql_index
   GROUP BY indexed_at
   ORDER BY count(*) DESC LIMIT 5;
   ```

5. If your contents are changing too fast, you may need to increase the indexation speed.

6. It'll not handle suppression. Rows there were deleted during the operation of the script may be still present in the new database after the operation. If this is a problem to you, you'll have to manually fix that (using SQL or Python). In a further version of the SeSQL, this may be fixed.

4 Advanced information

4.1 What is the state file ?

The state file is Python pickle of internal variables used by the script. It'll allow the script to restart near where it was (at the beginning of the current chunk, to be precise) if it is interrupted and restarted. If you don't use a state file, you can try to restart by using a manual start date, but the ETA will not be as precise, and you've to be careful into providing the right start date.

4.2 What is the drift ?

While the script will index rows, your application will be alive. That's the whole purpose of the thing. So it'll change data - some rows will be updated, some will be added. So they'll have to be re-indexed.

Imagine you have 1 000 000 of rows. Reindexing them will take a while, like 10 hours. During those 10 hours, 10 000 rows have been changed/added. So it needs to reindex those. It'll take like 6 minutes. But then, 100 new rows have been changed/added. Which will need to be reindexed. And so on.

The *drift rate* is the rate at which new items are modified/added, compared to the rate at which the indexation is going. In the example, the drift rate is 0.01 (the script reindex rows 100 times faster than they are added).

If the drift rate is higher (or equal) than 1, it'll never manage to reindex the content, new content is added faster than the script is running.

4.3 How do you evaluate the ETA ?

Using a crystal ball, tea leaves and the Large Hardon Collider, why ?

More seriously, the ETA is estimated by looking at the time used to index all previous content, the remaining content, and the estimated drift rate.

It's of course an approximation, but here are the key points :

1. It doesn't count the "down time", when the script was not running (if you interrupt and re-run it, with a state file). That may skew the drift rate, for content added/changed while the script was running is counted in the drift rate.

2. Estimating the drift rate requires an expensive `count(*)` select, so by default it's only performed once every ten chunks.

3. The estimation doesn't tell apart changed and added items, while they are indeed different. The estimation will be accurate if the changes are mostly added content and changing the most recent content, but not much accurate if the changes are evenly spread over all the data.

4. The estimation does compute the limit of the series (well, it's a geometric series, so the limit is easy to compute), and should consider all the "items changed while indexing the items changed while indexing the items changed..." up to infinity.