Advanced Database : Apache Solr

Maazouz Mehdi
Wouter Meire

December 16th, 2018

Summary

1 Introduction

1.1 What is a search engine ?

Basically, a search engine is a a script or a program that is going to search and find documents using keywords.
Nowadays there are a lot of search engines, some of them even have an everyday role in our lives like **Google** or **Yahoo**.
The topic of this article is an important one among them: **Solr**.

2 Solr and Lucene

In order to understand what Solr truly is, we first have to know about Lucene.

2.1 What is Lucene

Lucene is also a search engine, created in 1999 by Doug Cutting, it's an Apache project from the **Apache Software Foundation**. Concretely it's an open source information retrieval software written in java. It has been ported on most popular programming languages.

With Lucene, developers can include features like indexing, searching, analyzing when they develop a website for example.
Altough it's initial release was in 1999 Lucene continues to be updated regularly.

As Lucene is just a library, a person without some programming knowledge won't be able to use it. As a result, several projects have been created in order to extend Lucene's capability as well as their utilisation. Solr is a notable example of those projects.

2.2 What is Solr ?

Solr is an open source NoSQL search platform. It has been written in Java. It has several significant features, including full-text search, database integration, SolrCloud, rich document handling, geospatial search and more.

Solr is oriented for fault tolerance and scalability. The main utilisation of Solr is designed for enterprise search and analytic use cases.
Solr is used by some of the most heavily-trafficked websites in the world like *Disney*, *Netflix* or *Ebay* to name a few.

Solr continues to be updated with regular releases and it has an important community of developers. The latest version (7.5.0) of Solr has been published as recently as 24th September 2018.

2.3 History of Solr

Initially, Solr was created in 2004 by *Yonik Seelay* at CNET Networks in order to provide a better search capability for the company website.

In 2006, Solr's creator gave the source code to the *Apache Software Foundation* and it became a new Apache project. It helped with solving several problems related to organizational and financial issues.

In 2010, Solr and Lucene merged their developer communities. Thus, Solr became a Lucene sub-project. The same teams work on both these projects but each of them has their own releases. Since then, the development of Solr has the purpose of extending Lucene's capabilities.

More specifically, since they merged, Solr has been building around Lucene. A lot of application can use the library Lucene, not just Solr. Solr's purpose is to offer a web application and other features in addition to what Lucene offers. For example, Solr has added XML/HTML and JSON API's, caching, replication and the web administration interface, among others.

From 2010 to 2018, a lot of Solr's versions were released, always adding more features like SolrCloud, a new support for executing Parallel SQL , ...

2.4 Installation

2.4.1 Download

Installation of Solr is fairly easy and generally requires a simple extraction of a download package. Solr works on any of the big operation systems. The downloads are available at `http://www.apache.org/dyn/closer.lua/lucene/solr/7.5.0`.

Among the extracted directories we can find the bin directory, the example directory and the server directory. All contain important or useful stuff for running Solr. There are many README.txt included to help us find our way in the application.

- Bin: contains several scripts and most prominently the Solr control script that starts up the application.
- Example: has some neat examples that are used to demonstrate Solr capabilities.
- Server: Contains the core of Solr.

2.4.2 Starting up

For this the following command is needed to start up the application: bin/solr start

2.4.3 Other notes

Usually the way it works is that you first "play" with the installation to make it suit your needs before you deploy it on a true server environment.

We only showed the Linux commands, but there exist equivalent ones for Windows.

3 Architecture

In the Solr universe, a document is stored by extracting all the important words into a Solr field that's reverse-indexed.

3.1 Analysis phase

First of all, documents which will be indexed go through a serie of transformations. This serie of transformations is called the analysis phase. The purpose of this phase is to extract important words from the documents. There are a lot of transformations that are applied to the documents (case insensitive). Finally, the analysis phase returns a set of tokens which will be indexed.

3.2 Indexing

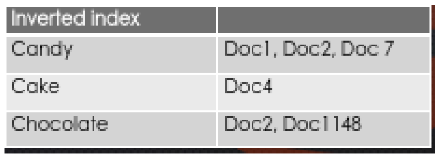
We know that a document is stored into fields. A field can contain a string value, a float value, an integer value, ...

The purpose of indexing is to allow Solr to find information faster. Solr allows you to build an index with many different fields.
It's important to highlight that when you add a document to an index, you make it searchable by Solr. Indeed if a document is not indexed, you cannot execute a query in order to find it.
The document formats which can be indexed by Solr are multiple, hence you can index a JSON, CSV, XML but also PDF, HTML or even MS Word documents if you want.

3.2.1 Inverted index

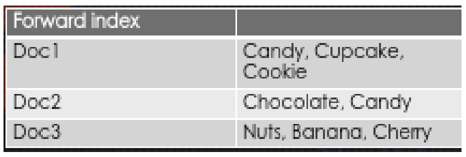
In order to index documents, Solr uses an inverted index. Consequently, documents will be indexed by their words. The inverted index is better

than the forward index because if you perform a query in order to find a word, Solr 's going to immediately return the set of documents containing the word. If you perform a query using a forward index, the query will need to go into each index in order to find the word.
Lets use an example: if you perform a query to find the word "Candy" using the inverted index. With this, it is possible to easily return the set of documents containing the word "candy".



Inverted index	
Candy	Doc1, Doc2, Doc 7
Cake	Doc4
Chocolate	Doc2, Doc1148

Now, Lets contrast this with the forward index. If you try to find the word "Candy", you see that you need to go into each document in order to check if the word is there or not.



Forward index	
Doc1	Candy, Cupcake, Cookie
Doc2	Chocolate, Candy
Doc3	Nuts, Banana, Cherry

3.3 Schema

When you want to index data, Solr is going to create a Schema file, in other words, an XML file which stores the details about the fields and their types Solr is expected to understand.

So, a Schema file 's going to define the field, the field type name, but also any modifications which can happen to a field before it is indexed.
For example, we can add a modification to ensure that a user can enter "abc" or "ABC" and that he still will find the same documents. This is an example of rules that can be implemented in the Schema file.

Remark, most schemas define a field called id, this field can be considered as a unique key or a primary key. Finally, the schema is the place where is described how Solr should build the index from input documents.

3.3.1 Schemaless or Field Guessing

If you want, you can quickly start Solr which then will use "field guessing" which is configured in solrconfig.xml. Hence you can index the documents without manually defining all the fields that Solr requires. So, when Solr 's going to encounter an unknown field it will automatically create it and guess its details in order to be able to index its. This is called "schemaless".

Unfortunately, this concept has its limit. For example we have several documents containing a movie we want to index. Solr's going to index these documents, starting with the first which contains the movie called "300". Solr is going to guess the field type based on the data in this record and it's going to create the field called "Film" which thus will be a numeric field (because 300).
But in the second document, the movie is called "Interstellar", we can see that it's a string value and not a numeric value. As a result, when Solr's going to index this movie and try putting the word "Interstellar" in the "Film" field, an error will occur.

Remark you can use Solr with schema and schemalessly at the same time. You can define the fields you want to control and let Solr create the fields which you are confident will be correctly guessed.

4 SolrCloud

SolrCloud is a significant feature of Solr. When you start Solr you can launch Solr "in simple mode" or in SolrCloud mode. You can see SolrCloud as the answer of "How Solr manages upscalling"

SolrCloud is more complex than the simple mode of Sorl. Indeed in order to start SolrCloud, you have to enter a number of nodes, shards and replicas. So in order to understand much better SolrCloud, you'll see some definitions below.

4.1 Definitions

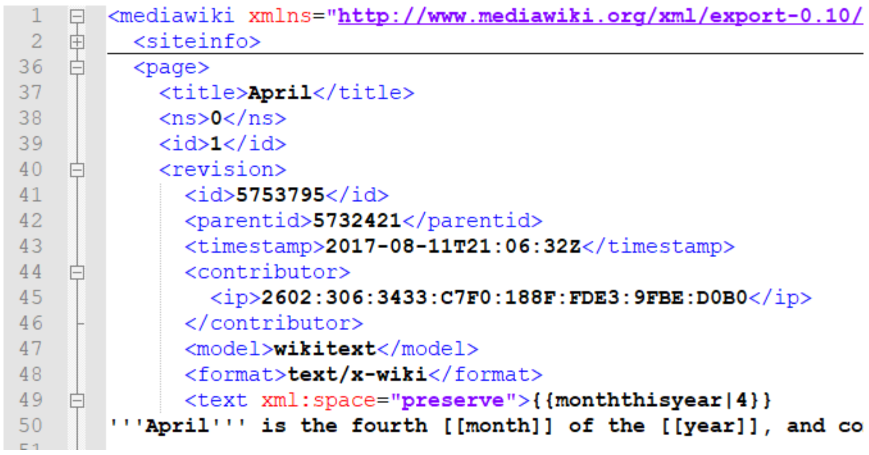
- (Apache) Zookeeper: What SolrCloud uses to keep track of the configuration files and the node names of a collection. It is used as a coordinator for operations that need distributed synchronisation.
- Node: A Java Virtual Machine (JVM) that runs Solr (aka Solr server)
- Shard: When a collection of documents is too large for a single node, then it needs to be split up into shards. Every document in a collection is contained in a single shard. Deciding how documents are distributed among shards depends on the strategy used to split up the documents. A simple way to achieve this would be to distribute the documents on the hash result of a unique key.
- Replica: Every shard in Solr consists of at least one replica where exactly one is the leader. There are different types of replicas, the default being NearRealTime (NRT). Those keep a transaction log and

write indexes locally. The two other types are TLOG, which has the transaction log but does not write the index changes locally -it copies it's indexes from the leader when needed, and PULL, which only replicates the index from the shard leader. This last one cannot be eligible to become leader.

5 MyWiki

In order to experiment sufficiently well with Solr, we needed to find a good document database that we could use for this purpose.
Wikipedia was one of the first large collection of documents that popped into our heads and it was not too difficult to find data dumps of the encyclopaedia that we could use. The file we chose contains a simplified version of Wikipedia in English: "simplewiki-20170820-pages-meta-current.xml"[1]
It's size is about 180 MiB and it contains 4300040 different pages.

All the documents have their information stored between the "page"-tags:

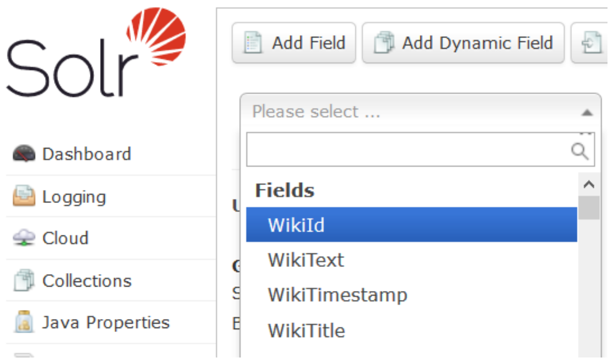


```
<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.10/
  <siteinfo>
  <page>
    <title>April</title>
    <ns>0</ns>
    <id>1</id>
    <revision>
      <id>5753795</id>
      <parentid>5732421</parentid>
      <timestamp>2017-08-11T21:06:32Z</timestamp>
      <contributor>
        <ip>2602:306:3433:C7F0:188F:FDE3:9FBE:D0B0</ip>
      </contributor>
      <model>wikitext</model>
      <format>text/x-wiki</format>
      <text xml:space="preserve">{{monththisyear|4}}
'''April''' is the fourth [[month]] of the [[year]], and co
```

For our project we focussed on the following fields as they seemed the most useful:

- **title** called WikiTitle in Solr's schema
- revision\id called WikiId in Solr's schema
- revision\timestamp called WikiTimestamp in Solr's schema
- revision\text called WikiText in Solr's schema

[1] https://meta.wikimedia.org/wiki/Data_dump_torrents#English_Wikipedia



.

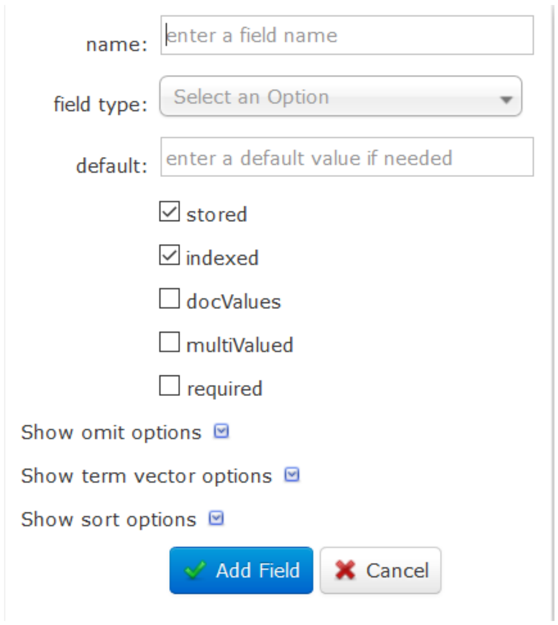
5.1 Import into Solr

To import this document into a useful form for Solr, the following steps needed to be done on a Solr configuration.

- Modify solrconfig.xml: it's the main configuration file, we needed to let it know about the data import handler script that we wrote for our project (see next item).
- Created DIHconfigfile.XML: this document tells solr how to extract the data from a specific xml file. Solr can import many types of documents (csv, json, ...) though this changes the data import handler(DIH).
- Created a new collection based on those configs.
- Added Necessary fields into the new collection's schema (see 5.1.1).
- Executed the dataimport in Admin UI. This does all the work to put the XML file into Solr
- And ready to go! (once the indexing of the dataimport is done).

5.1.1 Solr fields

When adding a field, a few options exist:


name: enter a field name

field type: Select an Option

default: enter a default value if needed

[✓] stored

[✓] indexed

[ ] docValues

[ ] multiValued

[ ] required

Show omit options

Show term vector options

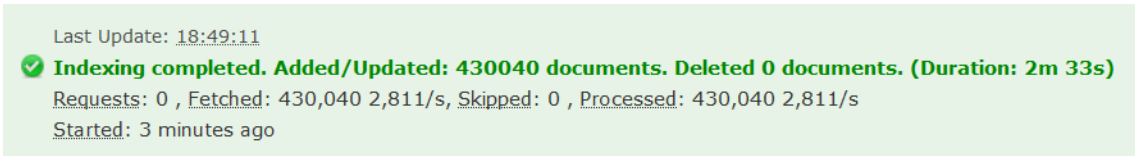
Show sort options

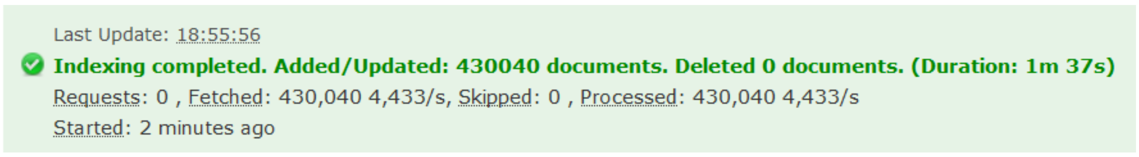
Add Field   Cancel

.

- Storing is important if the results need to be displayed when doing searches. An indexed field with no storing can be searched on, but the query will not be able to show the value of this particular field. This is because when a field is not stored, the index will use tokens which are more efficient and take less space, but they cannot be used to show values.
- Indexing is needed if it's a searchable field
- DocValues enables Solr to make a forward index, instead of the usual inverted one.
- MultiValued if more than one value can be present for the same field.
- Required if it's required (sometimes it's that simple)

Here are two data imports into Solr with the simplified wiki. One with all the fields indexed and stored, the other with all the fields indexed and only the WikiId field stored. We leave it up to the reader to guess which corresponds to which.


```
Last Update: 18:49:11
Indexing completed. Added/Updated: 430040 documents. Deleted 0 documents. (Duration: 2m 33s)
Requests: 0 , Fetched: 430,040 2,811/s, Skipped: 0 , Processed: 430,040 2,811/s
Started: 3 minutes ago
```


Last Update: 18:55:56
Indexing completed. Added/Updated: 430040 documents. Deleted 0 documents. (Duration: 1m 37s)
Requests: 0 , Fetched: 430,040 4,433/s, Skipped: 0 , Processed: 430,040 4,433/s
Started: 2 minutes ago

5.2 Import into SQL

Importing the simple wikipedia into a PostgreSQL database was done with the following command:

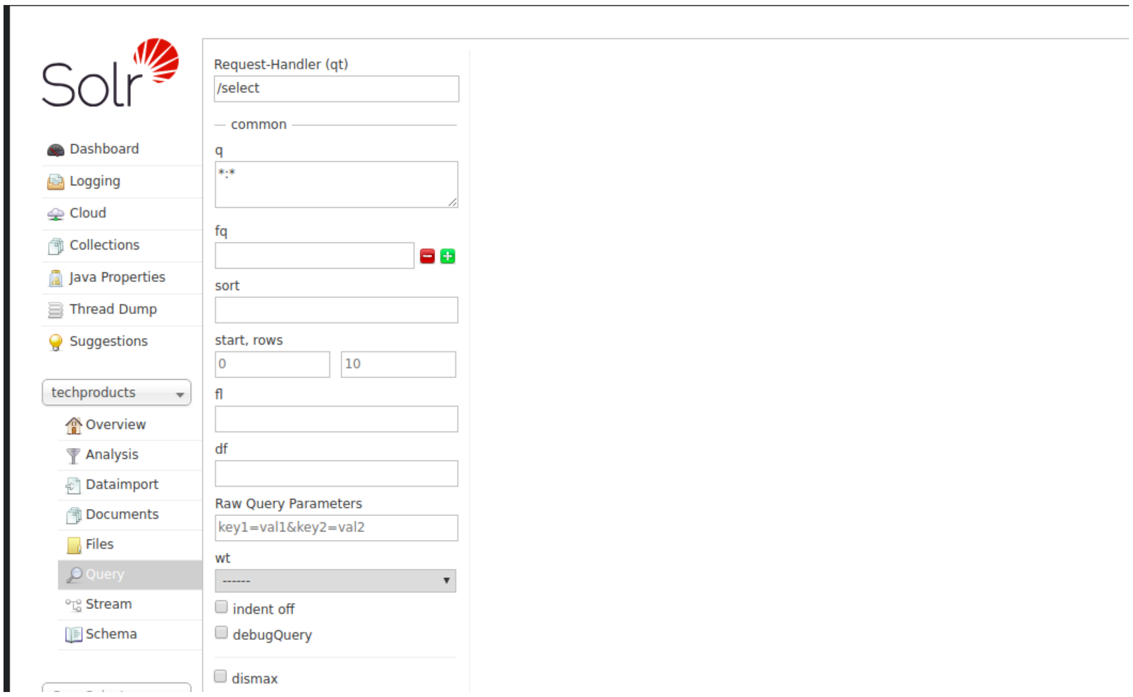
```
DROP TABLE IF EXISTS WikiImport;
CREATE TABLE IF NOT EXISTS WikiImport (
        "id" serial PRIMARY KEY,
        "title" TEXT NOT NULL,
        "timestamp" timestamptz NOT NULL,
        "text" TEXT NULL
);
INSERT INTO WikiImport
SELECT (xpath('//page/id/text()', x))[1]::text::int AS id,
         (xpath('//page/title/text()', x))[1]::text AS title,
         (xpath('//page/revision/timestamp/text()', x))[1]::text::timestamptz AS timestamp,
         (xpath('//page/revision/text/text()', x))[1]::text AS text
FROM unnest(xpath('//page', pg_read_file('simplewiki-20170820-pages-meta-current.xml')::xml)) x;
```

This was much harder than expected as the wikipedia xml contained a parameter that made the SQL query unsuccessful (xmlns="http://www.mediawiki.org/xml/export-0.10/"). Once this part of the XML was removed, the query ran as expected, otherwise all values that returned where empty. We searched for an explanation, but never found a satisfactory one.

6 Query

Solr can be queried by REST clients, curl, wget or via its User Interface.

Below is an overview of the User Interface



Request-Handler (qt): /select

common

q: *:*

fq:

sort:

start, rows: 0 10

fl:

df:

Raw Query Parameters: key1=val1&key2=val2

wt: ------

☐ indent off

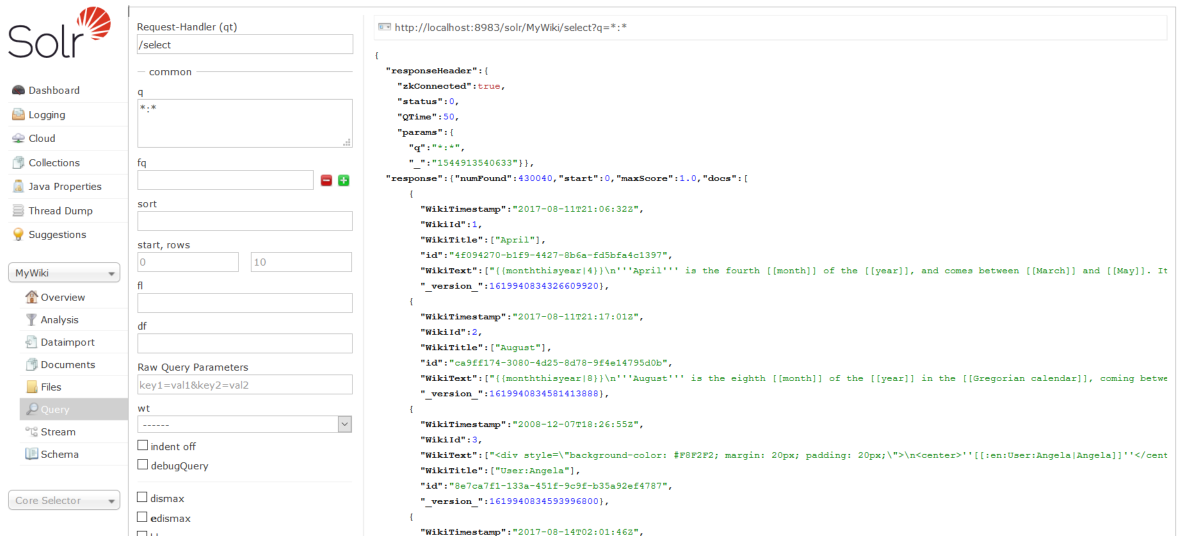
☐ debugQuery

☐ dismax

For the first example, we are going to execute the query with the query parameter (q) equal to *:* (which is a special syntax for quering all indexed documents).

As mentioned before, it is also possible to use curl to make the same request. For a Solr instance running locally it looks something like this: "http://localhost:8983/solr/techproducts/select?indent=on&q=*:*".

We can see the response below. So, what's happening ?



Request-Handler (qt): /select
common
q: *:*
fq
sort
start, rows: 0 10
fl
df
Raw Query Parameters: key1=val1&key2=val2
wt: ------
☐ indent off
☐ debugQuery
☐ dismax
☐ edismax

http://localhost:8983/solr/MyWiki/select?q=*:*

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":50,
    "params":{
      "q":"*:*",
      "_":"1544913540633"}},
  "response":{"numFound":430040,"start":0,"maxScore":1.0,"docs":[
      {
        "WikiTimestamp":"2017-08-11T21:06:32Z",
        "WikiId":1,
        "WikiTitle":["April"],
        "id":"4f094270-b1f9-4427-8b6a-fd5bfa4c1397",
        "WikiText":["{{monththisyear|4}}\n'''April''' is the fourth [[month]] of the [[year]], and comes between [[March]] and [[May]]. It
        "_version_":1619940834326609920},
      {
        "WikiTimestamp":"2017-08-11T21:17:01Z",
        "WikiId":2,
        "WikiTitle":["August"],
        "id":"ca9ff174-3080-4d25-8d78-9f4e14795d0b",
        "WikiText":["{{monththisyear|8}}\n'''August''' is the eighth [[month]] of the [[year]] in the [[Gregorian calendar]], coming betwe
        "_version_":1619940834581413888},
      {
        "WikiTimestamp":"2008-12-07T18:26:55Z",
        "WikiId":3,
        "WikiText":["<div style=\"background-color: #F8F2F2; margin: 20px; padding: 20px;\">\n<center>''[[:en:User:Angela|Angela]]''</cent
        "WikiTitle":["User:Angela"],
        "id":"8e7ca7f1-133a-451f-9c9f-b35a92ef4787",
        "_version_":1619940834593996800},
      {
        "WikiTimestamp":"2017-08-14T02:01:46Z",
```

Let's focus on the response header, as it contains important information on the query that has been executed:

```
"responseHeader":{
  "zkConnected":true,
  "status":0,
  "QTime":50,
  "params":{
    "q":"*:*",
    "_":"1544913540633"}},
"response":{"numFound":430040,"start":0,"maxScore":1.0,"docs":[
```

As you can see with the parameter numFound, 430040 documents are found in 50 milliseconds, shown in Qtime. However, only 10 documents are displayed because by default the number of rows returned is limited to 10. This is normal, as most search engine queries only need to see the "top-candidates" of the result. On a web search, displaying the 1,156,986 results that Google found for a random search term at once would be a great way to crash the browser, as it could exceed memory capacities (and it would be exceedingly slow).
With Solr and mainly with its UI, we can easily use different search options. We will show you different types of queries.

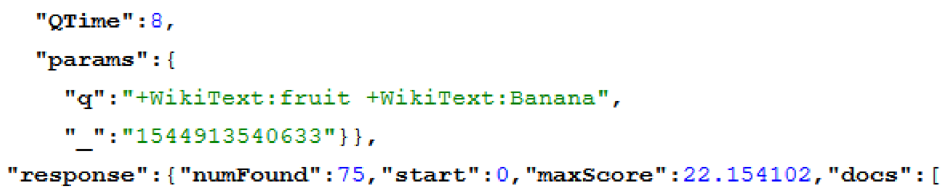
6.1 Field Searches

Solr enables you to return the documents containing the term you want.

For example, if you want to get all the documents containing the word "database" in their text, you just have to enter "TitleWiki:database" in the parameter q and execute the query. The value before the colon represents the field that needs to be searched on (it has to be an indexed field) and the value after the colon is what needs to be found. Wildcards such as "*" or "?" are permitted for this. After that, you'll see the response almost immediately (in normal cases).

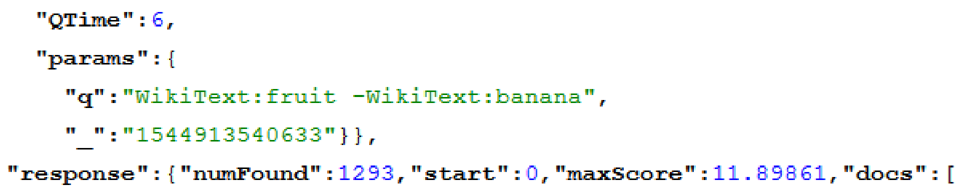
6.2 Combined Search

With Solr, you can easily search documents on several words. You can do this by putting a "+" before the term you search for. It is also possible to put an "AND" between(it yields the same results if all parameters involved have a "+" before them).



```
    "QTime":8,
    "params":{
      "q":"+WikiText:fruit +WikiText:Banana",
      "_":"1544913540633"}},
  "response":{"numFound":75,"start":0,"maxScore":22.154102,"docs":[
```

In addition, you can find documents that don't contain some terms. Instead of +, put a - just before the term.

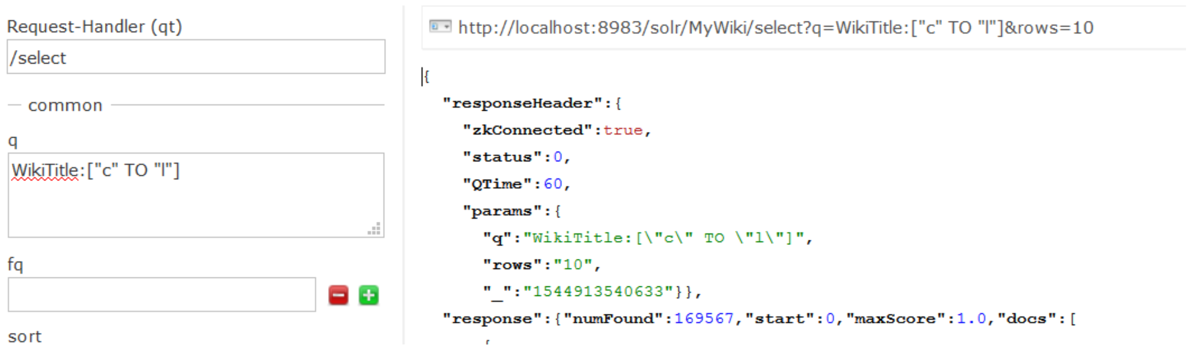


```
    "QTime":6,
    "params":{
      "q":"WikiText:fruit -WikiText:banana",
      "_":"1544913540633"}},
  "response":{"numFound":1293,"start":0,"maxScore":11.89861,"docs":[
```

In comparison, just searching on "fruit" yielded 1368 results.

6.3 Range queries

It is also possible to do searches based on range. This is done with the following syntax: field:[x TO y]. Here is an example where we search for the documents that have a title between "C" and "L", meaning all titles that start with those two letters and everything in-between will be represented:


Request-Handler (qt)
/select
common
q
WikiTitle:["c" TO "l"]
fq
sort

http://localhost:8983/solr/MyWiki/select?q=WikiTitle:["c" TO "l"]&rows=10

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":60,
    "params":{
      "q":"WikiTitle:[\"c\" TO \"l\"]",
      "rows":"10",
      "_":"1544913540633"}},
  "response":{"numFound":169567,"start":0,"maxScore":1.0,"docs":[
```

6.4 Other searches

More advanced searches are available (proximity search, boost, ...). More information available here[2] [3]

7 Solr vs SQL

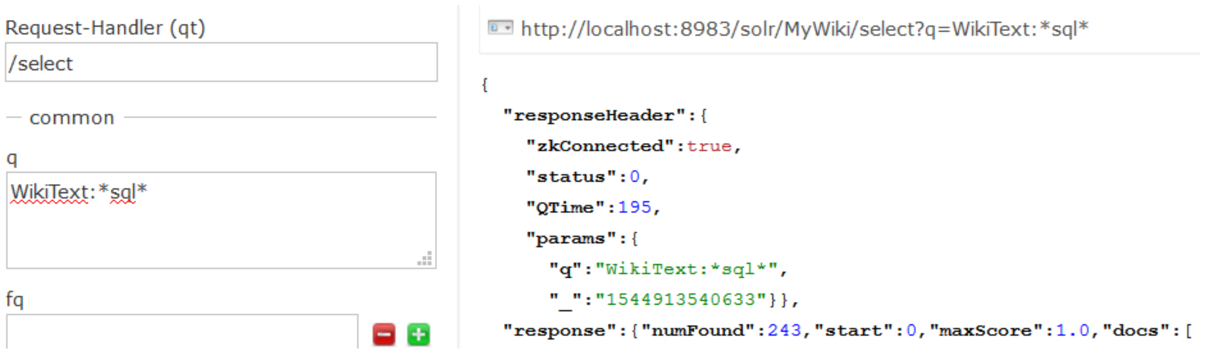
Because we've been bragging about the merits of Solr since the beginning of this document, it's time to put it to the test and to compare it to a relational database. For this we selected PostgreSQL and imported the same data as our Solr database (see 5.2).

7.1 First comparison

Get all the texts that include the word word-combination "sql". In Solr we used "*" to replicate the SQL's "like" behaviour with the percent wildcard. We decided on this because it's the standard way to search for a word on text in SQL. However, to be absolutly fair, we should have made the comparison with PostgreSQL's "Full Text Search" capabilities (see 7.4).

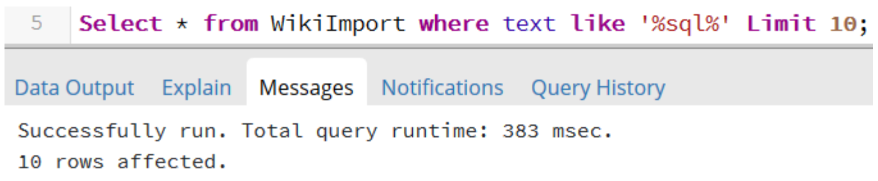
First we limit ourselves to the first ten results as it's the default number of rows Solr uses. We can see with both screenshots listed under this that Solr wins with a nice margin (195msec vs 383msec)

[2] `http://www.solrtutorial.com/solr-query-syntax.html`
[3] `http://lucene.apache.org/solr/guide/7_6/searching.html`


Request-Handler (qt)
/select
common
q
WikiText:*sql*
fq

http://localhost:8983/solr/MyWiki/select?q=WikiText:*sql*

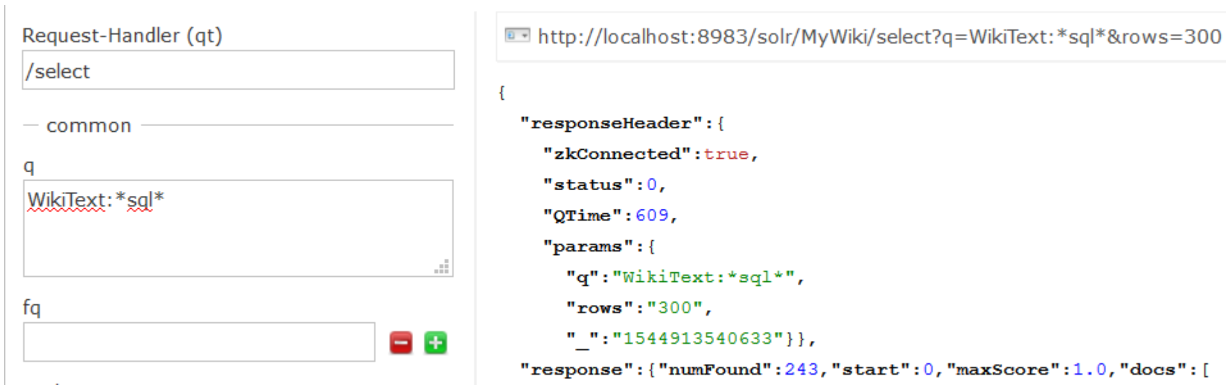
```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":195,
    "params":{
      "q":"WikiText:*sql*",
      "_":"1544913540633"}},
  "response":{"numFound":243,"start":0,"maxScore":1.0,"docs":[
```



```sql
Select * from WikiImport where text like '%sql%' Limit 10;
```

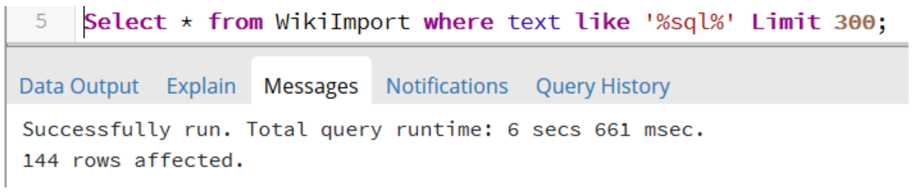
```
Successfully run. Total query runtime: 383 msec.
10 rows affected.
```

Then we went over the total number of matches to see how both systems would stand up to a heavier request. this resulted in an overwhelming victory from Solr (0.61s vs 6.661s)


Request-Handler (qt)
/select
common
q
WikiText:*sql*
fq

http://localhost:8983/solr/MyWiki/select?q=WikiText:*sql*&rows=300

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":609,
    "params":{
      "q":"WikiText:*sql*",
      "rows":"300",
      "_":"1544913540633"}},
  "response":{"numFound":243,"start":0,"maxScore":1.0,"docs":[
```



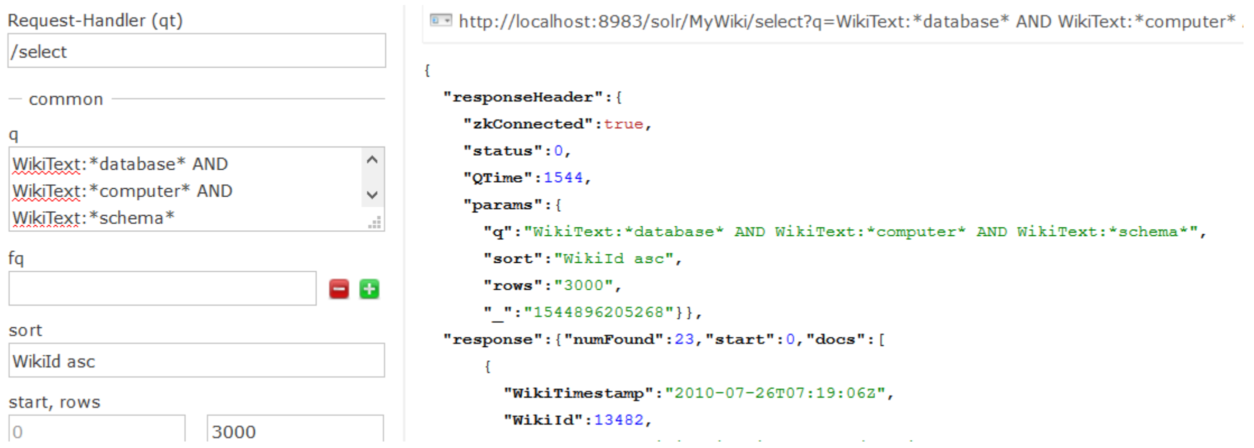
```sql
Select * from WikiImport where text like '%sql%' Limit 300;
```

```
Successfully run. Total query runtime: 6 secs 661 msec.
144 rows affected.
```

Additionally: Solr not being case-sensitive returns more results than SQL's query.

7.2 Second comparison

Get all the texts including the words "database", "computer" and "schema". Again here Solr comes out as the winner(1,55s vs 4,01s)



Request-Handler (qt)
/select

common

q
WikiText:*database* AND WikiText:*computer* AND WikiText:*schema*

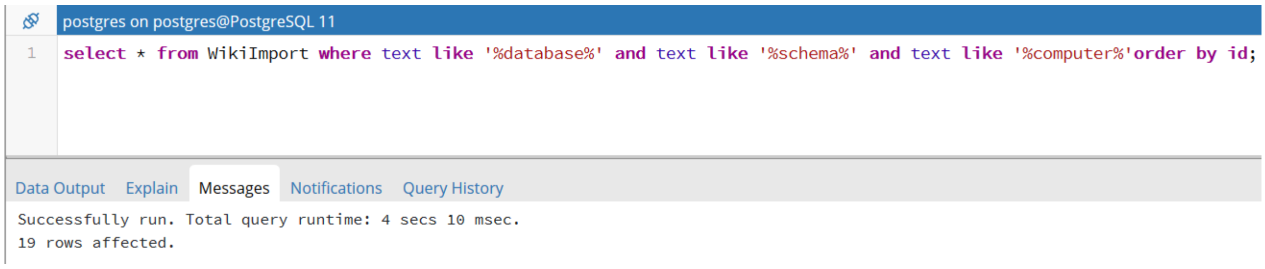
fq

sort
WikiId asc

start, rows
0 3000

http://localhost:8983/solr/MyWiki/select?q=WikiText:*database* AND WikiText:*computer*

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":1544,
    "params":{
      "q":"WikiText:*database* AND WikiText:*computer* AND WikiText:*schema*",
      "sort":"WikiId asc",
      "rows":"3000",
      "_":"1544896205268"}},
  "response":{"numFound":23,"start":0,"docs":[
      {
        "WikiTimestamp":"2010-07-26T07:19:06Z",
        "WikiId":13482,
```



postgres on postgres@PostgreSQL 11

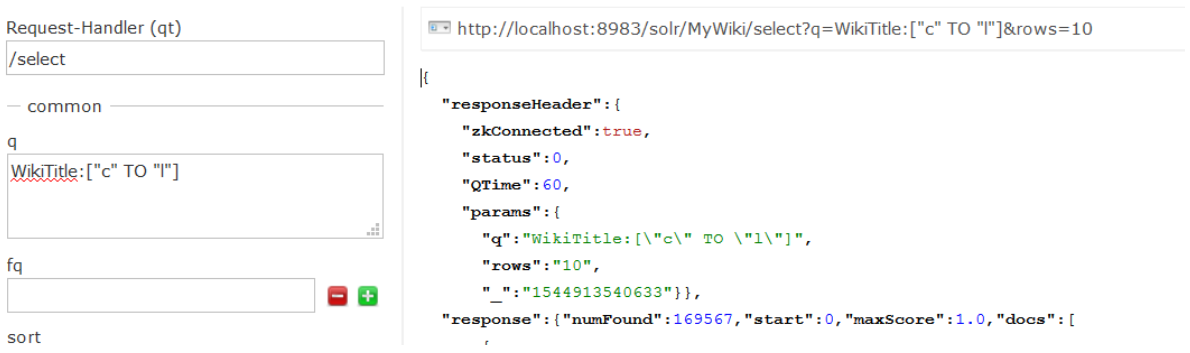
```
select * from WikiImport where text like '%database%' and text like '%schema%' and text like '%computer%'order by id;
```

Data Output Explain Messages Notifications Query History

```
Successfully run. Total query runtime: 4 secs 10 msec.
19 rows affected.
```

7.3 Third comparison

Get all the ten first titles ranging between "c" and "l". 60 against 146 ms, Solr on top again.



Request-Handler (qt)
/select

common

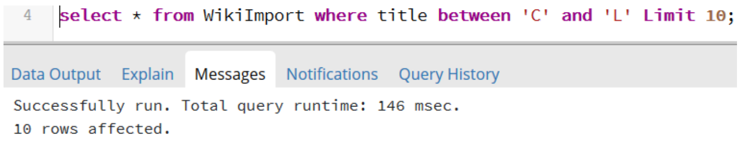
q
WikiTitle:["c" TO "l"]

fq

sort

http://localhost:8983/solr/MyWiki/select?q=WikiTitle:["c" TO "l"]&rows=10

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":60,
    "params":{
      "q":"WikiTitle:[\"c\" TO \"l\"]",
      "rows":"10",
      "_":"1544913540633"}},
  "response":{"numFound":169567,"start":0,"maxScore":1.0,"docs":[
```



```
select * from WikiImport where title between 'C' and 'L' Limit 10;
```

Data Output Explain Messages Notifications Query History

```
Successfully run. Total query runtime: 146 msec.
10 rows affected.
```

Additionally: creating an index on Title in SQL did not make it more competitive.

7.4 Remarks

- The query time for SQL is quite varying for the same request. We chose values here that we intuited were the mean. We never encountered a scenario with our tests where SQL beat Solr.
- The query time for Solr has always been the one obtained on the first try. Repeating a query dropped the Qtime significantly. This is due to automatic caching on Solr's part.
- Pretty late in the writing of this document, we saw that PostgreSQL also had a more specialised text search [4]. However initial trials were difficult and we decided to leave it as it was. It might be useful to look further into this if we ever want to build on this project, the comparison might be more fair this way.

8 Conclusion

Finally, we know that this is just an introduction of Solr. There are a lot of features we didn't mention. Like explained before, Solr continues to be regularly updated. Since its first release, its community has grown up and as of today, the official documentation of Solr is more than 1300 pages long.

We know that we could go into more depth with explaining some features like faceting, geospatial search, stream and so on but we wanted that the content of this article coincides with the content of the presentation.

References

[1] https://smarttechie.org/2014/07/10/apache-solr-the-inverted-index/

[2] http://yonik.com/solr/

[3] https://mikemadisonweb.github.io/2017/01/08/import-xml-into-sql/

[4] https://www.postgresql.org/docs/

[5] http://lucene.apache.org/solr/

[6] http://www.solrtutorial.com/basic-solr-concepts.html

[7] https://meta.wikimedia.org/wiki/Data_dump_torrents#English_Wikipedia

[4] `https://www.postgresql.org/docs/8.3/textsearch.html`

```
<dataConfig>
<dataSource type="FileDataSource" encoding="UTF-8"/>
<document>

<entity name="mywiki"
 url="C:\Users\Wouter\Downloads\simplewiki-20170820-pages-meta-current.xml
 \simplewiki-20170820-pages-meta-current.xml"
 stream="true"
 processor="XPathEntityProcessor"
 forEach="/mediawiki/page/"
 transformer="RegexTransformer,␣DateFormatTransformer">

<field column="WikiId"   xpath="/mediawiki/page/id"/>
<field column="WikiTitle"     xpath="/mediawiki/page/title"/>
<field column="WikiTimestamp"     xpath="/mediawiki/page/revision
/timestamp" dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss'Z'"/>
<field column="WikiText"     xpath="/mediawiki/page/revision/text"/>
</entity>

</document>
</dataConfig>
```