Assignment Report

Hadoop MapReduce BM25 Search Engine

20.04.2025

Soldatov Anatoly

a.soldatov@innopolis.university

Contents

1. Methodology

1.1. Pipeline

1. Starting hadoop and cassandra
2. Creating and activating virtual environment
3. Install and pack dependencies
4. Prepare data by sanitazing and putting it on hdfs
5. Indexing it via MapReduce operation on cluster with handwritten mapper and reducer on python
6. Searching result by given text query

1.2. Details

1.2.1. .venv

As hours of debuging showed the only correct way was to use container's python 3.8 to use in venv and no other approaches. Also hours of debuging on how to correctly propogate zipped venv and put it in mapred process (with your assistance on the lab).

The working final version:

```
mapred streaming \
    -files mapreduce/mapper1.py,mapreduce/reducer1.py \
    -archives ".venv.tar.gz#.venv" \
    -mapper ".venv/bin/python3 mapper1.py" \
    -reducer ".venv/bin/python3 reducer1.py" \
    -input "$INPUT_PATH" \
    -output "$OUTPUT_DIR" \
```

1.2.2. Mapper

Here I want to point out on text preprocessing techniques which helped dramatically reduce amount of useless and dirty data.

I normalized text:

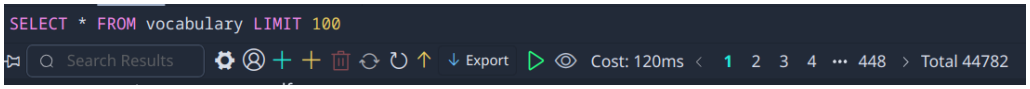
```
def normalize_text(text):
    """
    Normalizes unicode text by decomposing characters (NFKD)
    and removing diacritical marks.
    E.g. 'español' becomes 'espanol'.
    """
    normalized = unicodedata.normalize("NFKD", text)
    # Filter out diacritical marks (combining characters)
    normalized = "".join([c for c in normalized if not
```

```
unicodedata.combining(c)])
    return normalized
```

Lemmatize text separately for Russian and English:

```
def lemmatize_word(word):
    """
    Lemmatises a word using a different backend depending on its script.
    If the word contains any Cyrillic letters, use pymorphy2;
    """
    if re.search(r"[\u0400-\u04FF]", word):
        # Use pymorphy2 for Cyrillic words (e.g., Russian)
        parsed = morph.parse(word)
        if parsed:
            return parsed[0].normal_form
        else:
            return word
    else:
        return lemmatizer.lemmatize(word)
```

Also removed punctuation, digits, stop words for both languages producing relatively small vocabulary of ~45k tokens.



Mapper emits 2 types of strings into stdout:

1. “DOC|{doc_id}t{doc_title}t{doc_length}” - doc related stats
2. “TERM|{term}|{doc_id}t{tf}” - stats of each term’s in each doc’s term frequency

1.2.3. Reducer

Includes interaction with cassandra, so the following statements are executed:

```
CREATE KEYSPACE IF NOT EXISTS search WITH REPLICATION = {'class':
'SimpleStrategy', 'replication_factor': 1};

USE search;

CREATE TABLE IF NOT EXISTS doc_stats (
    doc_id int,
    doc_title text,
    doc_length int,
    PRIMARY KEY (doc_id)
);

CREATE TABLE IF NOT EXISTS inverted_index (
    term text,
```

```
    doc_id int,
    tf int,
    PRIMARY KEY (term, doc_id)
);

CREATE TABLE IF NOT EXISTS vocabulary (
    term text,
    df counter,
    PRIMARY KEY (term)
);
```

Here how insertion and update statements look like:

```
INSERT INTO doc_stats (doc_id, doc_title, doc_length) VALUES (?, ?, ?);
INSERT INTO inverted_index (term, doc_id, tf) VALUES (?, ?, ?);
UPDATE vocabulary SET df = df + 1 WHERE term = ?;
```

When reducer faces “DOC|...” string it performes checks and execute doc_stats insert. Else if “TERM|...” executes inverted_index insert and vocabulary update.

1.2.4. Search

After struggling a lot with usage cassandra format in spark directly in distributed manner I found the only working solution to me - compile **.jar** file directly from official repo. This step is written in build-connector.sh.

Here is main command in search.sh.

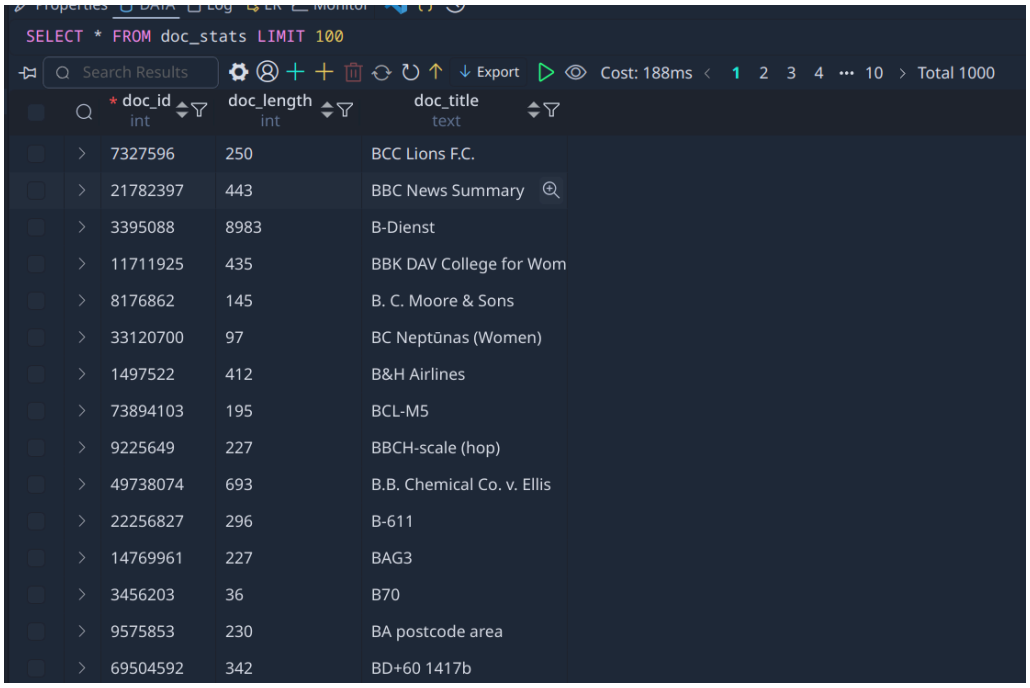
```
spark-submit  \
    --archives /app/.venv.tar.gz#.venv \
    --jars /app/spark-cassandra-connector.jar \
    --packages com.github.jnr:jnr-posix:3.1.15 \
    --master yarn \
    query.py "$1"
```

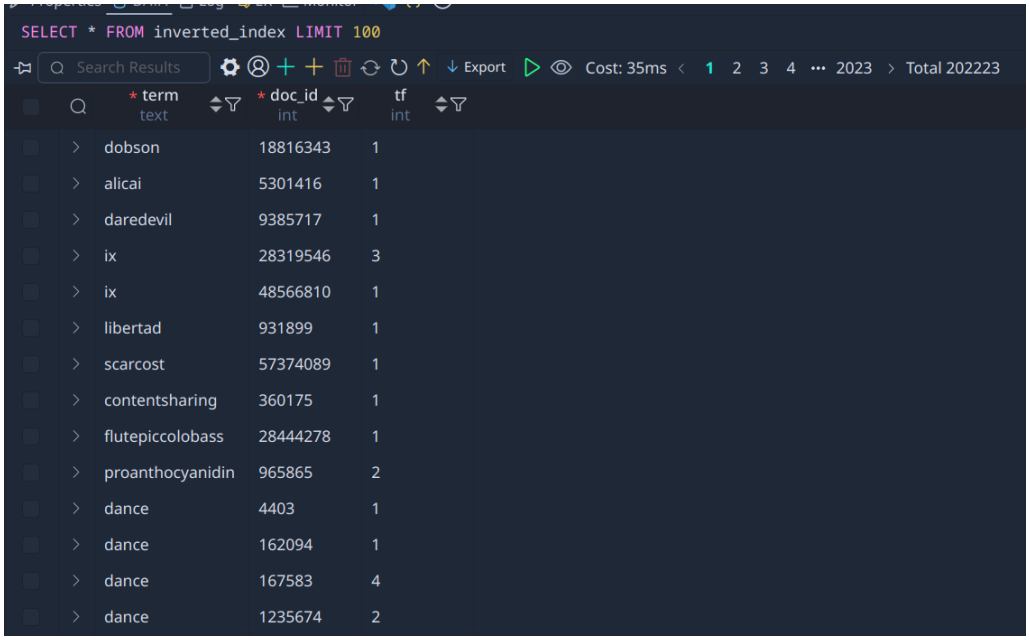
The query.py might be hard to perceive without step by step knowledge, so I prepared ml.ipynb where you can follow each step and observe output of each command used in beam search.

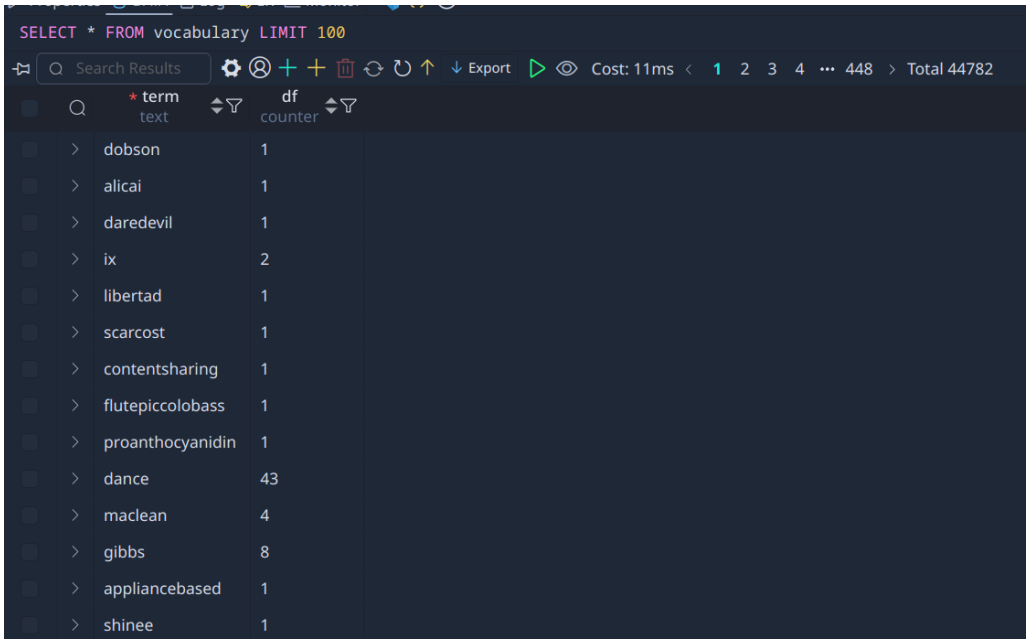
2. Demostration

2.1. MapReduce

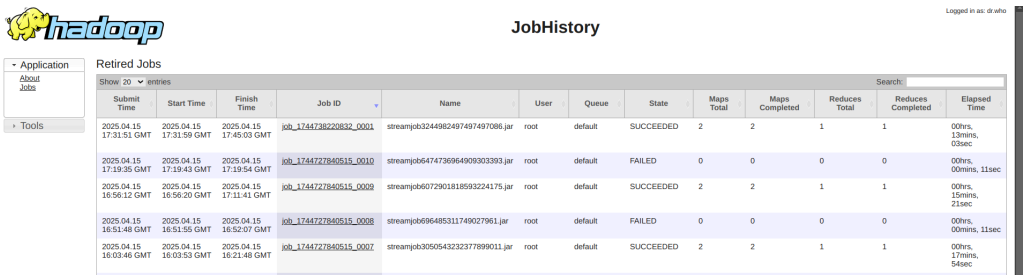
Below you can observe results of indexing document from b.parquet.


SELECT * FROM doc_stats LIMIT 100
Search Results
Export
Cost: 188ms
1 2 3 4 ··· 10
Total 1000
* doc_id int
doc_length int
doc_title text
7327596 250 BCC Lions F.C.
21782397 443 BBC News Summary
3395088 8983 B-Dienst
11711925 435 BBK DAV College for Wom
8176862 145 B. C. Moore & Sons
33120700 97 BC Neptūnas (Women)
1497522 412 B&H Airlines
73894103 195 BCL-M5
9225649 227 BBCH-scale (hop)
49738074 693 B.B. Chemical Co. v. Ellis
22256827 296 B-611
14769961 227 BAG3
3456203 36 B70
9575853 230 BA postcode area
69504592 342 BD+60 1417b


SELECT * FROM inverted_index LIMIT 100
Search Results
Export
Cost: 35ms
1 2 3 4 ··· 2023
Total 202223
* term text
* doc_id int
tf int
dobson 18816343 1
alicai 5301416 1
daredevil 9385717 1
ix 28319546 3
ix 48566810 1
libertad 931899 1
scarcost 57374089 1
contentsharing 360175 1
flutepiccolobass 28444278 1
proanthocyanidin 965865 2
dance 4403 1
dance 162094 1
dance 167583 4
dance 1235674 2

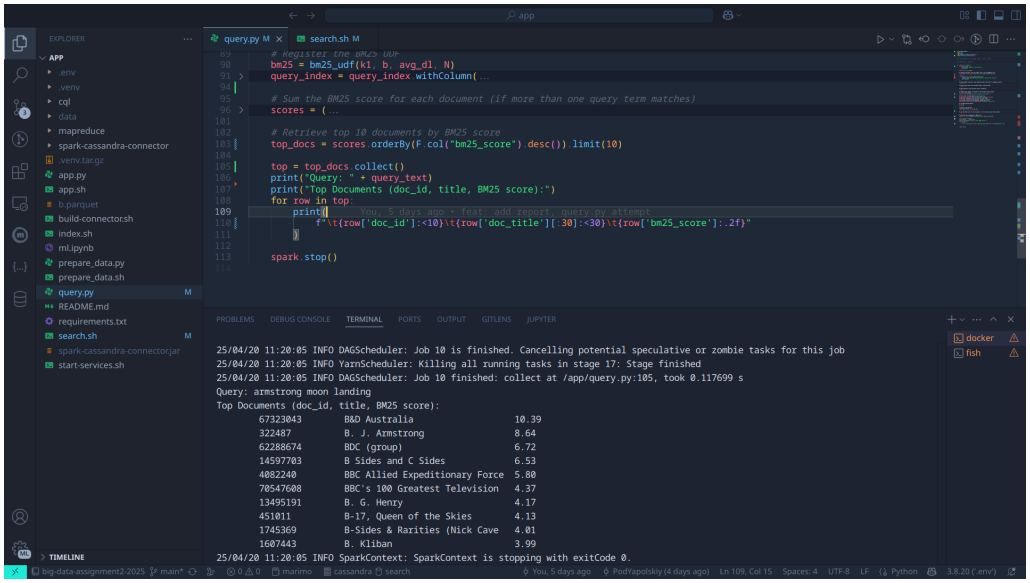


You may definitely see that preprocessing worked and can be improved even more!



Above there are several jobs that are marked as succeded after indexing documents via MapReduce.

2.2. Spark



Results on query “Armstrong moon landing”.

Definitely can conclude that search works, produces relevant result, have appropriate time execution and the most important is horizontally scalable in case of expanding our data.