



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Московский государственный технический университет имени  
Н.Э. Баумана (национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»  
КАФЕДРА «Компьютерные системы и сети (ИУ-6)»

**ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ**  
по дисциплине «Большие данные: инструменты и  
технологии»

Студент:	Козлов Владимир Михайлович
Группа:	ИУ6-13М
Тип задания:	лабораторная работа
Тема:	Spark

Студент

\_\_\_\_\_

подпись, дата

Козлов В.М.

Фамилия, И.О.

Преподаватель

\_\_\_\_\_

подпись, дата

Фамилия, И.О.

Москва, 2025

## **Содержание**

<b>Задание .....</b>	<b>3</b>
<b>1   Ход работы.....</b>	<b>3</b>
<b>2   Вывод .....</b>	<b>7</b>

# Задание

## Цель

Целью лабораторной работы является приобретение навыков работы со Spark.

## 1 Ход работы

Для выполнения использовался тот же docker-образ, что и для ЛР, так как там уже предусмотрен spark.

В качестве текста было выбрано начало НИР прошлого семестра по теме "Распределение вершин графа по гетерогенным хранилищам"

Для начала читается текст файла по строкам командой:

---

```
1 rdd = sc.textFile("/workspace/article.txt")
```

---

В качестве предобработки сначала строки очищаются от whitespace-символов по краям, преобразуются в нижний регистр, затем строки разделяются по пробелам на слова; далее отфильтровываются лишь слова из букв (в том числе и через тире) длиной более 3 символов с помощью регулярного выражения:

---

```
1 words = (rdd
2     .map(lambda x: x.strip())
3     .map(lambda x: x.lower())
4     .flatMap(lambda x: x.split()) # ["word1 word2", ...] -> ["word1", "word2"]
5     .filter(lambda x: re.match(r'^[A-Za-zAяА-ЯяА-Яя]{4,}$', x))
6 )
```

---

Далее подготавливается промежуточный RDD indexed, формируемый добавлением каждому элементу индекс; в результате вместо каждого элемента получается пара из индекса и элемента:

---

```
1 indexed = (words
2     .zipWithIndex()
3     .map(lambda x: (x[1], x[0])) # [(word, i), ...] -> [(i, word), ...]
4 )
```

---

Для биграмм формируется другой RDD shifted, состоящий из тех же элементов indexed, у которых индекс смешён на 1:

---

```
1 shifted = (indexed
2     .map(lambda x: (x[0] - 1, x[1]))
3 )
```

---

Далее, чтобы получить биграммы, промежуточный indexed и сдвинутый shifted группируются по ключу, которым выступают индексы, в результате чего и получается объединить все подряд идущие слова по парам; далее результат объединения превращается в одну строку из двух слов и отбрасываются индексы:

---

```
1 bigrams = (indexed
2     .join(shifted) #-> [(i, ["word1", "word2"]), ...]
3     .mapValues(lambda words: " ".join(words)) #-> [(i, "word1 word2"), ...]
4     .map(lambda x: x[1]) #-> ["word1 word2", ...]
5 )
```

---

Аналогичным образом для триграмм создаётся ещё один RDD со сдвигом индекса на 2 от первоначального значения:

---

```
1 shifted2 = (indexed
2     .map(lambda x: (x[0] - 2, x[1])))
3 )
```

---

После чего аналогичным с биграммами образом объединяются сначала первые два RDD, затем с результатом объединяется ещё и третий RDD, благодаря чему получаются триграммы:

---

```
1 trigrams = (indexed
2     .join(shifted)
3     .mapValues(lambda words: " ".join(words))
4     .join(shifted2)
5     .mapValues(lambda words: " ".join(words))
6     .map(lambda x: x[1])
7 )
```

---

Далее объявляется функция для подсчёта количества и сортировки по возрастанию:

---

```
1 def counted(rdd, ascending=False):
2     """Подсчёт количества элементов и сортировка: [(item, num), ...]"""
3     return (rdd
4         .map(lambda x: (x, 1))# добавляем 1 для суммы
5         .reduceByKey(lambda a, b: a + b)# складываем
6         .sortBy(lambda x: x[1], ascending=ascending) # сортировка по
    убыванию
7     )
```

---

И для получения требуемого результата выводятся 20 самых часто встречающихся биграмм и триграмм:

---

```
1 print(counted(bigrams).take(20))
2 print(counted(trigrams).take(20))
```

---

Листинг 1: Полный скрипт ЛР

---

```
1#!/bin/python3
2 from pyspark import SparkContext
3 # Initialize SparkContext
4 sc = SparkContext("local", "TextFileExample")
```

```

5
6 rdd = sc.textFile("lab2_text.txt")
7 rdd.take(5)
8 #предобработка
9 import re
10 words = (rdd
11     .map(lambda x: x.strip())
12     #убираем whitespace по краям
13     .map(lambda x: x.lower())
14     #приводим к нижнему регистру
15     .flatMap(lambda x: x.split()) #разделение на слова
16     #регулярка: только слова 4+ символов:
17     .filter(lambda x: re.match(r'^[A-Zа-яя---]{4,}$', x))
18 )
19
20 words.take(5)
21 def counted(rdd, ascending=False):
22     """Подсчёт количества элементов и сортировка: [(item, num), ...]"""
23     return (rdd
24         .map(lambda x: (x, 1))
25         #добавляем 1 для суммы
26         .reduceByKey(lambda a, b: a + b) #складываем
27         #
28         #.filter(lambda x: x[1] > 1)
29         #фильтруем по более 1 появлению появлений
30         .sortBy(lambda x: x[1], ascending=ascending) #сортировка по
31         #убыванию
32     )
33 #биграммы и триграммы
34 indexed = (words
35     .zipWithIndex()
36     .map(lambda x: (x[1], x[0]))
37 )
38 indexed.take(5)
39 #[(word, index), ...]
40 shifted = (indexed
41     .map(lambda x: (x[0] - 1, x[1]))
42 )
43 shifted.take(5)
44 #сдвиг индекса [(index - 1, word), ...]
45 bigrams = (indexed
46     .join(shifted)
47     .mapValues(lambda words: " ".join(words))
48     .map(lambda x: x[1])
49 )

```

```
50 bigrams.take(5)
51 # [(index, ["word1 "word2"]), ...]
52 # [(i, "word1 word2"), ...]
53 # убираем индексы
54 print(counted(bigrams).take(20))
55 shifted2 = (indexed
56     .map(lambda x: (x[0] - 2, x[1])))
57 )
58 shifted2.take(5)
59 # сдвиг индекса [(index - 2, word), ...]
60 trigrams = (indexed
61     .join(shifted)
62     .mapValues(lambda words: " ".join(words)))
63     .join(shifted2)
64     .mapValues(lambda words: " ".join(words))
65     .map(lambda x: x[1])
66 )
67 trigrams.take(5)
68 print(counted(trigrams).take(20))
```

---

## **2 Вывод**

В ходе лабораторной работы приобретены навыки работы со Spark.

# Приложение 1

## Листинг 2: Текст

---

Современные

распределённые графовые базы данных сталкиваются с фундаментальнойй  
→ проблемой эффективного распределения вершин графа по узлам хранения  
→ шардам(). Оптимальное распределение становится критически важным для  
→ производительности систем, где основной операцией является поиск  
→ путей между вершинами, которые могут находиться в разных шардах.  
→ Неэффективное распределение приводит к значительным задержкам при  
→ выполнении запросов и избыточным сетевым коммуникациям между  
→ узлами. Актуальность

данной работы обусловлена стремительным ростом объёмов графовых данных в

→ таких областях, как социальные сети, рекомендательные системы,  
→ биоинформатика и интернет вещей. Традиционные подходы к распределению  
→ данных демонстрируют ограниченную эффективность при работе с графиками,  
→ требующими учёта структурных особенностей и связности вершин. В

рамках исследования проводится сравнительный анализ двух принципиально

→ различных подходов к распределению графов: потоковых методов (online  
→ partitioning), работающих в реальном времени по мере поступления  
→ данных, и методов оптимизации распределения (offline partitioning),  
→ требующих полного знания структуры графа. Особое внимание уделяется  
→ алгоритмам библиотеки METIS, представляющей собой промышленный  
→ стандарт для задач разбиения графов, и современным потоковым  
→ алгоритмам, таким как Fennel и Streaming Graph Partitioning. Целью

работы является исследование и сравнение эффективности различных методов

→ распределения вершин графа по гомогенным хранилищам, а также  
→ разработка предложений по комбинированию подходов для достижения  
→ оптимального баланса между качеством разбиения и вычислительной  
→ эффективностью в условиях реальной эксплуатации распределённых  
→ графовых баз данных. В

распределённую графовую БД поступают или же поступали вершины графа.

→ Основной операцией в БД будет поиск пути между вершинами, которые  
→ могут оказаться в разных хранилищах БД. Для максимизации  
→ эффективности этой операции требуется решить следующую задачу. Дан

граф  $G=(V,E)$ ,  $|V|=n$ ,  $|E|=m$ . Распределение графа - это такое

→  $P=\{S_1, S_2 \dots S_k\}$ , где  $S_i$  - набор вершин далее( шард)  $v \in S_i$ ,  
→ при этом  $S_i \cap S_j = \emptyset$ . Требуется

найти такое распределение  $P^*=\{S^*_1, S^*_2 \dots S^*_k\}$ , так, чтобы

→ минимизировать общую мощность разрезов  $\sum_{i=1}^k |S^*_i|$ .

↳  $e(P) = \cup_{i=1}^n e(S^*_i, \backslash S^*_i)$ , либо же  
 ↳  $\lambda = \frac{|\partial_e(P)|}{m} \times 100\%$ . А также  
 ↳ минимизировать нормализованную максимальную нагрузку  
 ↳  $\rho = \frac{\max_n \{|S^*_i|\}}{\frac{n}{k}}$  максимизировать (балансировку).

METIS от( MEtis Tournament Inspired Strategy) \cite{MetisOG} - это набор программных библиотек и утилит, разработанных в Университете Миннесоты, для разбиения больших неориентированных графов и разрежения матриц. Ключевая

идея METIS - многоуровневая парадигма (Multilevel Paradigm). Вместо того чтобы работать с огромным исходным графом напрямую, METIS последовательно его упрощает, находит разбиение для маленького графа и затем интерполирует это разбиение обратно на исходный граф. Алгоритм

состоит из трёх этапов: Схлопывание Упрощение Свёртка

// англ(. Coarsening) - уменьшение графа схлопыванием вершин в граф  $G_1$ .  
 ↳ \ Исходный  
 граф  $G_0$  последовательно преобразуется во всё более мелкие графы  $G_1, G_2, \dots, G_l$ . На каждом шаге пары смежных вершин "схлопываются/ сворачиваются/" в одну супервершину. Распределение

англ(. Partitioning) - распределение разбиение() графа  $G_l$ . Поскольку граф  $G_l$  очень мал, для его разбиения можно использовать даже очень дорогие вычислительно( сложные) алгоритмы. Развёртка

англ(. Uncoarsening) - развёртка графа  $G_l$ , то есть действие обратное первому этапу. Также происходит улучшение англ(. refinement) - оптимизация разбиения в процессе развёртки. Порядок следующий: Разбиение

, найденное для  $G_l$ , проецируется на граф  $G_{l-1}$ . Так как каждая вершина в  $G_{l-1}$  соответствовала вершине в  $G_l$ , разбиение определяется автоматически. Улучшение

, то есть переброс вершин между шардами для получения лучшего распределения. Первые

два пункта повторяются пока не будет получено разбиение для  $G_0$ . Свёртка

в Metis сводится к поиску максимального паросочетания, но не наибольшего, так как это слишком затратное действие. и дальнейшего схлопывания рёбер паросочетания. В

базовом методе рассматриваются 4 алгоритма поиска максимального

→ паросочетания:лучайное

С англ(. Random Matching - RM) - берётся случайное ребро, затем случайное

→ ребро, несмежное с ним, после чего ребро не связанное с выбранными

→ ранее, и так далее пока таких рёбер не останется. Метод прост и

→ эффективен для локальной задачи, но для уменьшения общей мощности

→ разрезов в дальнейшем подходит плохо. Паросочетание

тяжёлых рёбер англ(. Heavy Edge Matching - HEM) - выбор рёбер

→ максимального веса по очереди. Экспериментально было установлено, что

→ это приводит к хорошему уменьшению общей мощности разрезов. В Metis

→ этот алгоритм используется как основной. Паросочетание

лёгких рёбер англ(. Light Edge Matching - LEM) - выбор рёбер минимального

→ веса по очереди. Алгоритм приводит к большему общему весу рёбер у

→ свёрнутого графа, что может положительно влиять на качество улучшения

→ некоторыми алгоритмами. Паросочетание

тяжёлых клик англ(. Heavy Clique Matching - HCM) - объединяет вершины,

→ максимизируя плотность ребер создаваемого подграфа. В

отличие от HEM, который выбирает тяжелые ребра, HCM оценивает, насколько

→ две вершины точнее( вершины, которые свернулись в них) близки к

→ формированию клики. Для вершин  $\$u\$$  и  $\$v\$$  вычисляется плотность рёбер

→ англ(. edge density) по формуле:Алгоритм

схож с HEM, но учитывает и вершины изначального графа. Алгоритмы  
распределенияЭтап

распределения является решением поставленной в секции 1 задачи для

→ свёрнутого графа. В Metis рекурсивно используются алгоритмы бисекции

→ графа, поэтому число шардов  $\$k\$$  должно быть степенью двойки, другое

→  $\$k\$$  приведут к плохой балансировке.На

этом этапе также рассматриваются 4 алгоритма:Разбиение

:  $\$P[j]=1\$$  если  $\$y_j \le r\$$ , и  $\$P[j]=2\$$  в противном случае, где  $\$r\$$  -

→ выбранная взвешенная медиана  $\$y_j\$$ .Алгоритм

Кернигана-Лина англ(. Kernighan-Lin - KL) - алгоритм, работающий на идее

→ улучшения разбиения путём обмена вершин между шардами. Для этого

→ определена следующая функция улучшения распределения:при

положительном  $\$g_v\$$  перемещение вершины  $\$v\$$  приведёт к улучшению

→ распределения. За одну итерацию алгоритм проходится по всем вершинам

→ и пытается переместить все, итерации прерываются как только никакое

→ перемещение не приведёт к улучшению, либо когда будет достигнуто

→ предельное число операций. Эксперименты показали, что для достаточно

→ хорошего распределения как правило хватает 5-10

→ итераций.\cite{MetisOG}. Для избегания работы "вхолостую" итерация

→ прерывается при 50 перемещений без улучшения, что хорошо показало  
→ себя при экспериментах. Также стоит отметить, что ещё можно  
→ ограничить алгоритм на минимальное кол-во перемещений за  
→ итераций. Функция

улучшения может быть рассчитана для всех вершин в начале итерации и при

→ перемещении вершины обновляться только для соседей перемещённой  
→ вершины для оптимизации. Алгоритм  
роста графа англ(. Graph Growing - GGP) - выбирается случайная вершина и  
→ от неё как при поиске в ширину обходятся вершины, формируя подграф,  
→ пока не будет набрана ровно половина вершин. Этот подграф будет  
→ первым шардом, остальные вторым. Далее рекомендуется применить KL для  
→ улучшения, что не займёт много времени в силу малого размера  
→ свёрнутого графа. Также стоит отметить, что вполне можно делить не  
→ пополам, а на другие равные части и применять KL к шардам  
→ попарно. Жадный

алгоритм роста графа англ(. Greedy Graph Growing - GGGP) - так как для  
→ предыдущего алгоритма и так рекомендуется применять KL после бисекции  
→ можно сразу при обходе графа идти по вершинам, которые будут давать  
→ наибольшее улучшение.

---

## Приложение 2

Диграммы	Триграммы
<p>распределения вершин (2)</p> <p>между которые (2)</p> <p>распределение графа (2)</p> <p>общей мощности (2)</p> <p>алгоритм роста (2)</p> <p>роста графа (2)</p> <p>вершин графа (2)</p> <p>основной операцией (2)</p> <p>вершин между (2)</p> <p>graph growing (2)</p> <p>edge matching (2)</p> <p>функция улучшения (2)</p> <p>пока будет (2)</p> <p>также стоит (2)</p> <p>которые могут (2)</p> <p>matching выбор (2)</p> <p>современные графовые (1)</p> <p>сталкиваются фундаментальной (1)</p> <p>распределение становится (1)</p> <p>операцией является (1)</p>	<p>между которые могут (2)</p> <p>алгоритм роста графа (2)</p> <p>распределения вершин графа (2)</p> <p>edge matching выбор (2)</p> <p>современные графовые базы (1)</p> <p>таких социальные рекомендательные (1)</p> <p>реальном времени мере (1)</p> <p>распределения требующих полного (1)</p> <p>предложений комбинированию подходов (1)</p> <p>реальной эксплуатации графовых (1)</p> <p>максимизации эффективности этой (1)</p> <p>чтобы минимизировать общую (1)</p> <p>первому также происходит (1)</p> <p>процессе порядок найденное (1)</p> <p>случайное затем случайное (1)</p> <p>метод прост эффективен (1)</p> <p>приводит большему общему (1)</p> <p>учитывает вершины изначального (1)</p> <p>если противном выбранная (1)</p> <p>обмена вершин между (1)</p>

Таблица 1: Частотный анализ диграмм и триграмм