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:	ИУ6-33М
Тип задания:	лабораторная работа
Тема:	Spark

Студент _____ Козлов В.М.
подпись, дата Фамилия, И.О.

Преподаватель _____ Григоренко В.М.
подпись, дата Фамилия, И.О.

Москва, 2025

Содержание

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 вабрано начало НИР прошлого семестра по теме "Распределение вершин графа по гетерогенным хранилищам"

Для начала читается текст файла по строкам командой:

```
    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:

```
words = (rdd
    .map(lambda x: x.strip())
    .map(lambda x: x.lower())
    .flatMap(lambda x: x.split()) # ["word1 word2", ...] -> ["word1", "word2"]
    .filter(lambda x: re.match(r'^[A–Za–AzЯая———]{4,}$', x))
)
```

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

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

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

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

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:

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

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

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

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

```
trigrams = (indexed
    .join(shifted)
    .mapValues(lambda words: " ".join(words))
    .join(shifted2)
    .mapValues(lambda words: " ".join(words))
    .map(lambda x: x[1])
)
```

Далее объявляется функция для подсчёта количества и сортировки по возрастанию:

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

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

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

Листинг 1: Полный скрипт ЛР

```
#!/bin/python3
from pyspark import SparkContext
# Initialize SparkContext
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-Za-AzЯая---]{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
       убыванию
31     )
32 # биграммы и триграммы
33 indexed = (words
34     .zipWithIndex()
35     .map(lambda x: (x[1], x[0]))
36 )
37 indexed.take(5)
38 # [(word, index), ...]
39 # [(index, word), ...]
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 )
```

```
bigrams.take(5)
# [(index, ["word1 "word2"]), ...]
# [(i, "word1 word2"), ...]
# убираем индексы
print(counted(bigrams).take(20))
shifted2 = (indexed
    .map(lambda x: (x[0] - 2, x[1]))
)
shifted2.take(5)
# сдвиг индекса [(index - 2, word), ...]
trigrams = (indexed
    .join(shifted)
    .mapValues(lambda words: " ".join(words))
    .join(shifted2)
    .mapValues(lambda words: " ".join(words))
    .map(lambda x: x[1])
)
trigrams.take(5)
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 фундаментальной
    ↪ 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 графами,
    ↪ 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
    ↪ $P=\{S_1,S_2...S_k\}$, где $S_i$ - набор вершин далее( шард) $v\in
    ↪ V$, при этом $S_i\cap S_j=\emptyset$.Требуется

 найти такое распределение $P^*=\{S^*_1,S^*_2...S^*_k\}$,  так, чтобы
    ↪ минимизировать общую мощность разрезов $|\partial
```

```
    ↪ e(P)|=|\cup^n_{i=1}e(S^*_i,V\backslash S^*_i)|$, либо же
    ↪ $\lambda=\frac{|\partial e(P)|}{m}\times100\%$. А также
    ↪ минимизировать нормализованную максимальную нагрузку
    ↪ $\rho=\frac{max^n_{i=1}(|S^*_i|)}{\frac{n}{k}}$ 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:СхлопываниеУпрощениеСвёртка
// англ(. Coarsening) - уменьшение графа схлопыванием вершин в граф $G_1$.
    ↪ \\Исходный
 граф G0 последовательно преобразуется во всё более мелкие графы $G_1, G_2,
    ↪ ..., G_l$. На каждом шаге пары смежных вершин
    ↪ "схлопываютсясворачиваются/" 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 $G_{m-1}$. Так как каждая
    ↪ вершина в $G_{m-1}$ 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, другоие
↪ 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**
распределения вершин (2)	между которые могут (2)
между которые (2)	алгоритм роста графа (2)
распределение графа (2)	распределения вершин графа (2)
общей мощности (2)	edge matching выбор (2)
алгоритм роста (2)	современные графовые базы (1)
роста графа (2)	таких социальные рекомендательные (1)
вершин графа (2)	реальном времени мере (1)
основной операцией (2)	распределения требующих полного (1)
вершин между (2)	предложений комбинированию подходов (1)
graph growing (2)	реальной эксплуатации графовых (1)
edge matching (2)	максимизации эффективности этой (1)
функция улучшения (2)	чтобы минимизировать общую (1)
пока будет (2)	первому также происходит (1)
также стоит (2)	процессе порядок найденное (1)
которые могут (2)	случайное затем случайное (1)
matching выбор (2)	метод прост эффективен (1)
современные графовые (1)	приводит большему общему (1)
сталкиваются фундаментальной (1)	учитывает вершины изначального (1)
распределение становится (1)	если противном выбранная (1)
операцией является (1)	обмена вершин между (1)

Таблица 1: Частотный анализ диграмм и триграмм