

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и кибербезопасности

Работа допущена к защите  
Руководитель ОП  
\_\_\_\_\_ А.В. Щукин  
«\_\_\_\_\_» \_\_\_\_\_ 2024 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**  
**РАБОТА БАКАЛАВРА**  
**ИССЛЕДОВАНИЕ И ЭКСПЕРИМЕНТАЛЬНЫЙ АНАЛИЗ СРЕДСТВ**  
**РАБОТЫ С БОЛЬШИМИ ДАННЫМИ В ВЕБ-ПРИЛОЖЕНИЯХ**  
по направлению подготовки 09.03.03 Прикладная информатика  
Направленность (профиль) 09.03.03\_03 Интеллектуальные инфокоммуникацион-  
ные технологии

Выполнил  
студент гр. 5130903/00301

Г.О. Фролов

Руководитель  
ст. преподаватель ВШПИ

В.А. Пархоменко

Консультант  
по нормоконтролю

Е.Е. Андрианова

**САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
ПЕТРА ВЕЛИКОГО**

**Институт компьютерных наук и кибербезопасности**

УТВЕРЖДАЮ

Руководитель ОП

\_\_\_\_\_ А.В. Щукин

« \_\_\_\_\_ » \_\_\_\_\_ 2024г.

**ЗАДАНИЕ**

**на выполнение выпускной квалификационной работы**

студенту Фролову Георгию Оскаровичу гр. 5130903/00301

1. Тема работы: Исследование и экспериментальный анализ средств работы с большими данными в веб-приложениях.
2. Срок сдачи студентом законченной работы: 17.05.2024.
3. Экспериментальные данные: данные о сообщениях из чата на стриминговой платформе Twitch [3.1].
  - 3.1. Стриминговая платформа Twitch. — URL: <https://www.twitch.tv/> (дата обращения: 11.02.2024).
4. Содержание работы (перечень подлежащих разработке вопросов):
  1. Выбор и последующий анализ документации и исследований по устройству выбранных средств работы с большими данными
  2. Сравнение выбранных средств работы с большими данными в рамках обозначенных категорий(пакетная и потоковая обработка)
  3. Проектирование и реализация системы, направленной на проведение экспериментального сравнительного анализа средств как для потоковой, так и для пакетной обработки данных
  4. Проведение экспериментов, направленных на оценку временной эффективности выбранных средств работы с большими данными, для пакетной обработки - время выполнения, для потоковой - задержка
  5. Формулирование выводов о целесообразности использования и конкурентоспособности тех или иных средств работы с большими данными в рамках

обозначенных категорий на основании результатов экспериментов и анализа их архитектуры

5. Перечень графического материала (с указанием обязательных чертежей): нет

6. Консультанты по работе:

6.1. Ст. преподаватель ВШ ПИ, Е.Е. Андрианова (нормоконтроль).

7. Дата выдачи задания: 12.02.2024.

Руководитель ВКР \_\_\_\_\_ В.А. Пархоменко

Задание принял к исполнению 12.02.2024

Студент \_\_\_\_\_ Г.О. Фролов

## **РЕФЕРАТ**

На 106 с., 107 рисунков, 13 таблиц, 18 приложений

**КЛЮЧЕВЫЕ СЛОВА:** БОЛЬШИЕ ДАННЫЕ, APACHE SPARK, APACHE FLINK, HADOOP, MAP REDUCE, HDFS, APACHE KAFKA, APACHE HIVE, ПАКЕТНЫЕ ВЫЧИСЛЕНИЯ, ПОТОКОВЫЕ ВЫЧИСЛЕНИЯ

Тема научно-исследовательской работы: «ИССЛЕДОВАНИЕ И ЭКСПЕРИМЕНТАЛЬНЫЙ АНАЛИЗ СРЕДСТВ РАБОТЫ С БОЛЬШИМИ ДАННЫМИ В ВЕБ-ПРИЛОЖЕНИЯХ».

Цель работы заключается в выборе, и последующем анализе архитектурных особенностей средств работы с большими данными, а также оценка временной эффективности средств работы с большими данными. В рамках работы были выбраны наиболее популярные средства пакетной и потоковой обработки данных. Были сравнены их архитектурные особенности и подход к формированию и исполнению задач. После чего было проведено экспериментальное сравнение временной эффективности выполнения ими потоковой и пакетной обработки данных. В рамках потоковой обработки данных сравнивались Apache Flink и Spark Streaming. Оба продемонстрировали достойные результаты, и оба могут считаться актуальными инструментами для потоковой обработки данных. Тем не менее Apache Flink продемонстрировал более стабильную и значительно меньшую абсолютную задержку, нежели Spark Streaming. В рамках пакетной обработки данных сравнивались фреймворки Spark, Hive on Tez и Hadoop Map Reduce. Был сделан вывод, что Spark на сегодняшний день превосходит другие сравниваемые инструменты в отношении временной эффективности. Hive on Tez демонстрирует небольшое отставание по эффективности выполнения задач пакетной обработки данных по сравнению со Spark, и ввиду удобства использования HiveQL синтаксиса так признается эффективным и актуальным инструментом. Hadoop Map Reduce, напротив, демонстрирует достаточно слабые результаты и не рекомендуется к использованию ввиду существования более быстрых и удобных альтернатив, перечисленных выше.

## **ABSTRACT**

106 pages, 107 figures, 13 tables, 18 appendices

**KEYWORDS:** BIG DATA, APACHE SPARK, APACHE FLINK, HADOOP, MAP REDUCE, HDFS, APACHE KAFKA, APACHE HIVE, BATCH COMPUTING, STREAM COMPUTING

Theme of the research work: “RESEARCH AND EXPERIMENTAL ANALYSIS OF LARGE DATA WORKING SOLUTIONS IN WEB APPLICATIONS”. The purpose of the work is to select and then analyze the architectural features of big data tools, as well as to evaluate the temporal efficiency of big data tools. Within the framework of the work the most popular means of batch and stream data processing were selected. Their architectural features and approach to task generation and execution were compared. After that, an experimental comparison of the temporal efficiency of their execution of streaming and batch data processing was conducted. Apache Flink and Spark Streaming were compared in terms of streaming data processing. Both demonstrated decent results, and both can be considered relevant tools for stream processing. However, Apache Flink demonstrated a more stable and significantly lower absolute latency than Spark Streaming. The Spark, Hive on Tez, and Hadoop Map Reduce frameworks were compared within the batch processing framework. It was concluded that Spark by far outperforms the other compared tools in terms of temporal efficiency. Hive on Tez shows a slight lag in the efficiency of batch processing tasks compared to Spark, and due to the ease of use of the HiveQL syntax is so recognized as an effective and relevant tool. Hadoop Map Reduce, on the contrary, demonstrates rather poor results and is not recommended for use due to the existence of faster and more convenient alternatives listed above.

## СОДЕРЖАНИЕ

Введение .....	9
Глава 1. Анализ архитектурных особенностей средств работы с большими данными .....	11
1.1. Инструменты для пакетных вычислений.....	11
1.1.1. MapReduce.....	12
1.1.2. Spark .....	15
1.1.3. Apache Hive .....	19
1.1.4. Apache Tez .....	22
1.1.5. Сравнение перечисленных инструментов .....	23
1.2. Инструменты для потоковых вычислений.....	25
1.2.1. Spark Streaming .....	26
1.2.2. Apache Flink .....	28
1.2.3. Сравнение Spark Streaming и Apache Flink .....	29
Глава 2. Проектирование, развертывание и разработка системы для проведения экспериментального сравнения выбранных средств работы с большими данными .....	32
2.1. Выбор источников данных и их формат .....	32
2.2. Формулирование требований к системе .....	35
2.3. Выбор вспомогательных средств обработки и хранения данных .....	37
2.3.1. Система хранения данных .....	37
2.3.2. Брокер сообщений .....	38
2.4. Архитектура системы .....	39
2.5. Установка и конфигурация компонентов системы .....	41
2.5.1. Среда выполнения.....	41
2.5.2. Apache Hadoop .....	43
2.5.3. Apache Spark.....	46
2.5.4. Apache Hive .....	48
2.5.5. Apache Flink .....	50
2.5.6. Kafka.....	53
2.6. Разработка сервера-слушателя IRC сокетов .....	54
2.7. Автоматизация проведения экспериментального сравнения средств потоковой обработки данных .....	59
Глава 3. Экспериментальное сравнение фреймворков для потоковой обработки данных: Apache Flink и Spark Streaming .....	69
3.1. Задача чтения данных из Kafka Produсera, обработки и записи в HDFS .....	69

3.1.1. Реализация на Apache Flink .....	69
3.1.2. Реализация на Spark Streaming .....	72
3.1.3. Низкая нагрузка .....	75
3.1.4. Средняя нагрузка .....	77
3.1.5. Высокая нагрузка .....	79
3.2. Задача цензурирования ненормативной лексики .....	82
3.2.1. Реализация на Apache Flink .....	82
3.2.2. Реализация на Spark Streaming .....	85
3.2.3. Низкая нагрузка .....	86
3.2.4. Средняя нагрузка .....	88
3.2.5. Высокая нагрузка .....	90
3.3. Выводы по итогам экспериментов .....	93
Глава 4. Экспериментальное сравнение фреймворков для пакетной обра- ботки данных: Hadoop MapReduce, Apache Hive и Spark .....	94
4.1. Задача подсчёта количества записей в датасетах .....	94
4.1.1. Реализация на Apache Hive .....	94
4.1.2. Реализация на Map Reduce .....	95
4.1.3. Реализация на PySpark .....	96
4.1.4. Результаты эксперимента .....	97
4.2. Задача фильтрации брани в хранимых записях .....	98
4.3. Задача формирования топа стримеров по количеству сообщений .....	100
4.4. Задача формирования топа пользователей по количеству сообщений .	102
4.5. Выводы .....	104
Заключение .....	106
Список использованных источников .....	110
Приложение 1. Отличные от переменных по умолчанию системные пере- менные в WSL .....	114
Приложение 2. IRC Socket server .....	115
Приложение 3. Скрипт для выполнения экспериментов в рамках потоковой обработки данных .....	118
Приложение 4. Класс для анализа задержки во процессе потоковой обра- ботки данных .....	121
Приложение 5. Класс для анализа нагрузки во время потоковой обработки данных .....	124
Приложение 6. Функции, запускающие Spark Steaming и Apache Flink работы извне WSL .....	127
Приложение 7. Модуль utils для скрипта, автоматизирующего проведение экспериментов .....	129

Приложение 8. Модуль-логгер .....	131
Приложение 9. Apache Flink Job, выполняющая чтение сообщений из Kafka Producer, инициированного сервером, слушающим IRC сокет и сохраняющая записи в HDFS в формате csv .....	132
Приложение 10. Файл зависимостей(pom.xml) для Apache Flink Job .....	138
Приложение 11. Spark Streamig Job, выполняющая чтение сообщений из Kafka Producer, инициированного сервером, слушающим IRC сокет и сохраняющая записи в HDFS в формате csv .....	142
Приложение 12. Конфигурация первого эксперимента по потоковой обработке данных .....	144
Приложение 13. Flink Job для фильтрации нецензурной лексики из сообщений .....	145
Приложение 14. Spark Job для фильтрации нецензурной лексики из сообщений .....	151
Приложение 15. Конфигурация второго эксперимента по потоковой обработке данных .....	154
Приложение 16. Работы для второго эксперимента по пакетной обработке данных .....	155
Приложение 17. Работы для третьего эксперимента по пакетной обработке данных .....	160
Приложение 18. Работы для четвертого эксперимента по пакетной обработке данных .....	166



## ВВЕДЕНИЕ

Разработчики уже достаточно давно столкнулись с проблемой обилия данных, которые нужно где-то и каким-то стандартизированным образом хранить, а также обрабатывать. Один из путей борьбы с этой проблемой - уменьшение потоков данных, но это, в действительности, невозможно, так как с развитием систем искусственного интеллекта данные стали напрямую определять точность и скорость работы многих компонентов ПО. Значит единственный способ решить данную проблему - улучшать и создавать новые методики и инструменты для хранения и обработки данных, которые принято называть "большими данными". В связи с этим интересной и актуальной задачей является сравнение существующих на рынке средств для анализа больших данных. При этом сравнение должно быть реализовано как на архитектурном, так и на практическом, экспериментальном уровне.

**Актуальность** проведения исследовательской работы обусловлена стремительным развитием сферы больших данных и обилием инструментов для их анализа, в связи с чем оценка и сравнение различных характеристик и практических показателей вышеуказанных инструментов станет ценным руководством по выбору определенного из инструментов для конкретной задачи.

**Цель** работы заключается в выборе, и последующем анализе архитектурных особенностей средств работы с большими данными, а также создании инфраструктуры для проведения экспериментов, направленных на оценку временной эффективности выбранных средств работы с большими данными.

Для достижения этой цели были поставлены следующие задачи:

1. Выбор и последующий анализ документации и исследований по устройству выбранных средств работы с большими данными
2. Сравнение выбранных средств работы с большими данными в рамках обозначенных категорий(пакетная и потоковая обработка)
3. Проектирование и реализация системы, направленной на проведение экспериментального сравнительного анализа средств как для потоковой, так и для пакетной обработки данных
4. Проведение экспериментов, направленных на оценку временной эффективности выбранных средств работы с большими данными, для пакетной обработки - время выполнения, для потоковой - задержка

5. Формулирование выводов о целесообразности использования и конкурентоспособности тех или иных средств работы с большими данными в рамках обозначенных категорий на основании результатов экспериментов и анализа их архитектуры

## ГЛАВА 1. АНАЛИЗ АРХИТЕКТУРНЫХ ОСОБЕННОСТЕЙ СРЕДСТВ РАБОТЫ С БОЛЬШИМИ ДАННЫМИ

В данной главе будут выбраны средства, которые будут сравниваться между собой. Средства будут подразделены на две категории: средства для пакетной обработки данных(раздел 1.1) и средства для потоковой обработки данных(раздел 1.2). Все выбранные средства относятся к свободно распространяемому ПО с открытым исходным кодом и имеют различные архитектурные особенности, именно за счет этого имеет смысл их сравнение.

### 1.1. Инструменты для пакетных вычислений

При планировании работы было решено разделить сравниваемые инструменты на две категории, инструменты для потоковых вычислений и инструменты для пакетных вычислений. В этой главе мы рассмотрим наиболее популярные инструменты из второй категории. И те, и те инструменты предназначены для работы с большими данными, но при этом решают разные задачи. Задачи до того разные, что они, почти всегда, выполняются на разных этапах работы с данными. Сейчас мы опишем задачу пакетной обработки данных, и приведем примеры задач, которые можно было бы отнести к задачам пакетной обработки данных, в начале следующей главы мы сделаем то же самое с задачей потоковой обработки данных. Это необходимо, чтобы сформулировать различия между этими задачами, даже самые неочевидные, а также обосновать легитимность подобной категоризации сравниваемых фреймворков.

Итак, пакетные вычисления - это метод обработки данных, при котором данные обрабатываются в "пакетах"или "партиях"в отличие от обработки в реальном времени. В задачах пакетных вычислений данные обычно анализируются и обрабатываются в оффлайн режиме, без необходимости реагировать на изменения данных в реальном времени.

Вот несколько групп задач, которые являются частью множества задач пакетных вычислений:

1. **Анализ больших объёмов данных.** Обработка и анализ больших объёмов данных, например, журналов серверов или записей транзакций.

2. **Обработка данных в партиях.** Выполнение операций агрегации, фильтрации, сортировки и других манипуляций с данными на больших наборах данных.
3. **Подготовка данных для аналитики.** Подготовка данных для последующего анализа, например, визуализации, отчётности или машинного обучения.
4. **Выполнение вычислений над большими данными.** Выполнение сложных вычислений, таких как статистические анализы, машинное обучение или графовые алгоритмы.

В качестве примеров конкретных задач пакетной обработки данных, поставленных в контексте деятельности каких-либо компаний или организаций можно указать:

1. Процессинг ежемесячных банковских выписок
2. Обработка данных датчиков в умных устройствах
3. Анализ логов серверов для обнаружения аномалий и мониторинга производительности
4. Обработка данных из социальных медиа для анализа трендов

Таким образом, пакетные вычисления это работа с большим количеством собранных заранее данных, а именно выполнение операций агрегации, сортировки и других достаточно требовательных по ресурсам статистических вычислений.

Далее мы рассмотрим наиболее широко использующиеся для пакетных вычислений средства работы с большими данными, изучим их внутреннее устройство и проведем их внеэкспериментальное предварительное сравнение.

### ***1.1.1. MapReduce***

MapReduce – это модель распределённых вычислений от компании Google, используемая в технологиях Big Data для параллельных вычислений над очень большими (до нескольких петабайт) наборами данных в компьютерных кластерах, и фреймворк для вычисления распределённых задач на узлах кластера.

Важно отметить, что MapReduce это именно модель, то есть конкретная реализация лежит на руках разработчика, и эта реализация будет относиться к MapReduce, только при условии соблюдения определенных стандартов модели разработчиком.

Также важно понимать, что MapReduce задуман как альтернатива SQL для вычислений на узлах кластера. Соответственно любые операции, которые можно реализовать с помощью SQL также могут быть реализованы и с помощью MapReduce. При этом SQL, как мы уже выяснили, неприменим в случае работы с большими данными, так как размер наших данных может составлять несколько Петабайт, соответственно такие данные не поместятся на жесткий диск одной машины и не могут стать доступными с помощью реляционной некластерной БД. HDFS же запишет данные на разные узлы, создав реплики каждого блока данных и никаких проблем с хранением и получением доступа к данным не возникнет.

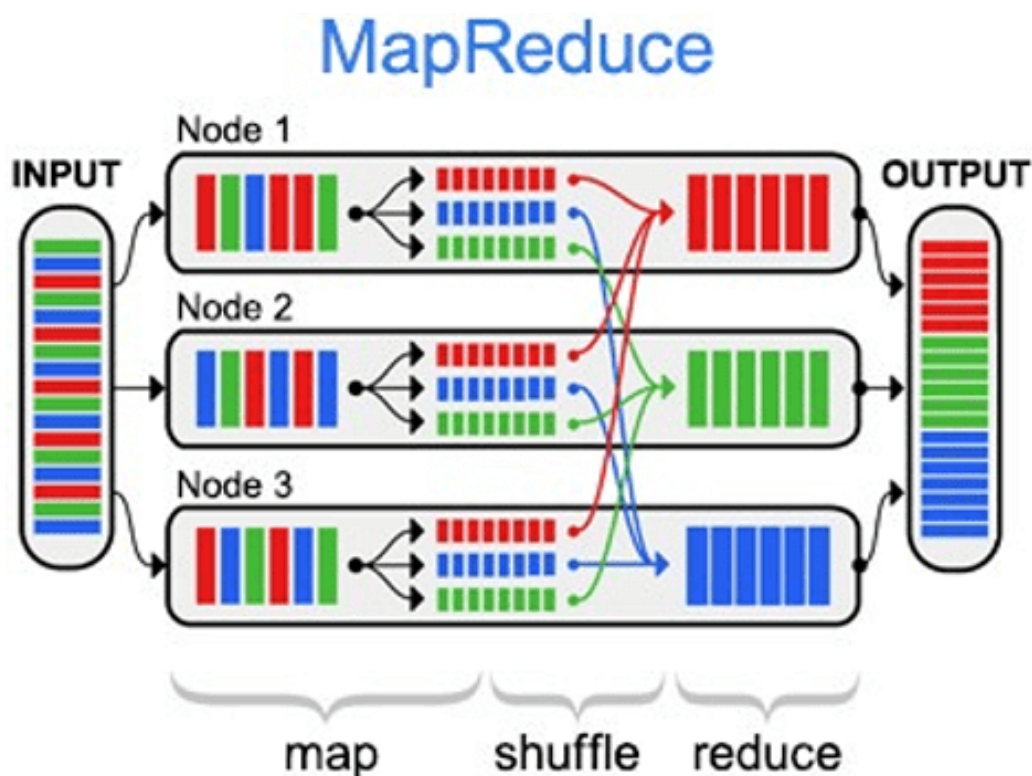


Рис.1.1. Схема манипуляций с данными, производимых Map Reduce

Теперь обсудим алгоритмическую структуру MapReduce. Основных функции, как можно понять из названия, в модели две, это Map и Reduce. В функциональном программировании существуют одноименные функции, которые, кстати, очень похожи на функции Map и Reduce внутри методики.

Действия, которые функция Map будет производить с данными зависят от ее конкретной имплементации. В общем случае, map функция будет применена к каждой строчке нашей таблицы данных(если представить данные, как таблицу), и каким-то образом трансформирует ее, вернув новую версию.

В функциональном программировании функция map возвращает новый массив, с данными того же типа, что она и принимала, но каждая элементарная

единица данных будет изменена. В парадигме MapReduce происходит то же самое, функция `map` добавляет видоизмененные строки в новый массив, чтобы была возможность применить к строкам определенные операции агрегирования, например, отсортировать строки по одному из показателей. В реализации Map Reduce, которая используется в Hadoop к данным обязательно применяется сортировка по "ключу". Ключом считается столбец, который определен таковым разработчиком. На следующем этапе Map Reduce это упорядочивание сыграет значительную роль в оптимизации работы алгоритма. Таким образом формируется промежуточная таблица данных, с которой будут работать функции на следующих этапах.

После завершения этапа `map`, происходит этап, который играет ключевую роль в выведении производительности MapReduce на новый уровень. Этот этап называется шаффл и заключается в распределении строк таблицы данных по узлам кластера. Одно из полей таблицы определяется как ключ, после чего строки с одним и тем же значением ключа перемещаются на один узел кластера. Ключ определяется разработчиком, а шаффл строк временной таблицы, полученной в результате работы `map` функции выполняется самим Hadoop. Естественно, ключ в каждой задаче будет разный.

Итак, по истечении этапа "шаффл" все строки условной промежуточной таблицы с одинаковыми значениями поля-ключа находятся на одном и том же узле кластера. Мы можем удобно применить функцию `reduce`, которая принимает фрагмент таблицы, а возвращает одну строчку с определенным значением. Что именно будет делать функция `reduce`, также зависит от конкретной имплементации, которую пишет программист.

Между этапами происходит запись промежуточных данных на диск, при этом перемещение информации между узлами кластера происходит только на этапе шаффла, что позволяет значительно сократить время выполнения задачи Map Reduce.

К сожалению, в данный момент технология Map Reduce считается устаревшей и на практике используется очень редко, особенно при разработке новых систем. Устаревание Map Reduce на самом деле не удивительно, так как сам концепт появился еще в 2005 году, даже до первой версии релиза Hadoop. Несмотря на это, если у технологии нет конкурентов, она не устаревает, и устаревание Map Reduce в процессе развития сферы хранения и анализа больших данных как раз связано с появлением множества конкурирующих инструментов, которые мы рассмотрим в следующих разделах главы.

### 1.1.2. Spark

Spark имеет более широкую специализацию, нежели Map Reduce. Тем не менее, именно компонент Spark SQL, который решает задачи пакетной обработки больших данных принес Spark популярность и именно этот компонент используется чаще всех компонентов Spark. В данном разделе мы проанализируем алгоритмическую структуру Spark и попробуем ответить на вопрос, почему за последние несколько лет Spark SQL почти полностью вытеснил Map Reduce, как инструмент для пакетной обработки больших данных.

Spark можно описать, как вычислительный движок, реализующий функциональный подход для обработки больших данных на серверных кластерах.

Spark содержит 5 основных компонентов:

- A. **Spark SQL** - механизм SQL запросов к различным типам хранилищ данных, реализованный на декларативном языке SQL.
- B. **Spark Core** - пакет утилит для управления памятью и восстановлении в кластере.
- C. **Spark Streaming** - библиотека для обработки потоковых данных в режиме реального времени.
- D. **MLlib** - компонент для машинного обучения.
- E. **GraphX** - компонент для работы для построение и анализа графов данных.

Наиболее подходящим для сравнения в рамках работы с пакетными вычислениями является компонент экосистемы Spark SQL, но помимо него мы рассмотрим устройство экосистемы Spark в целом.



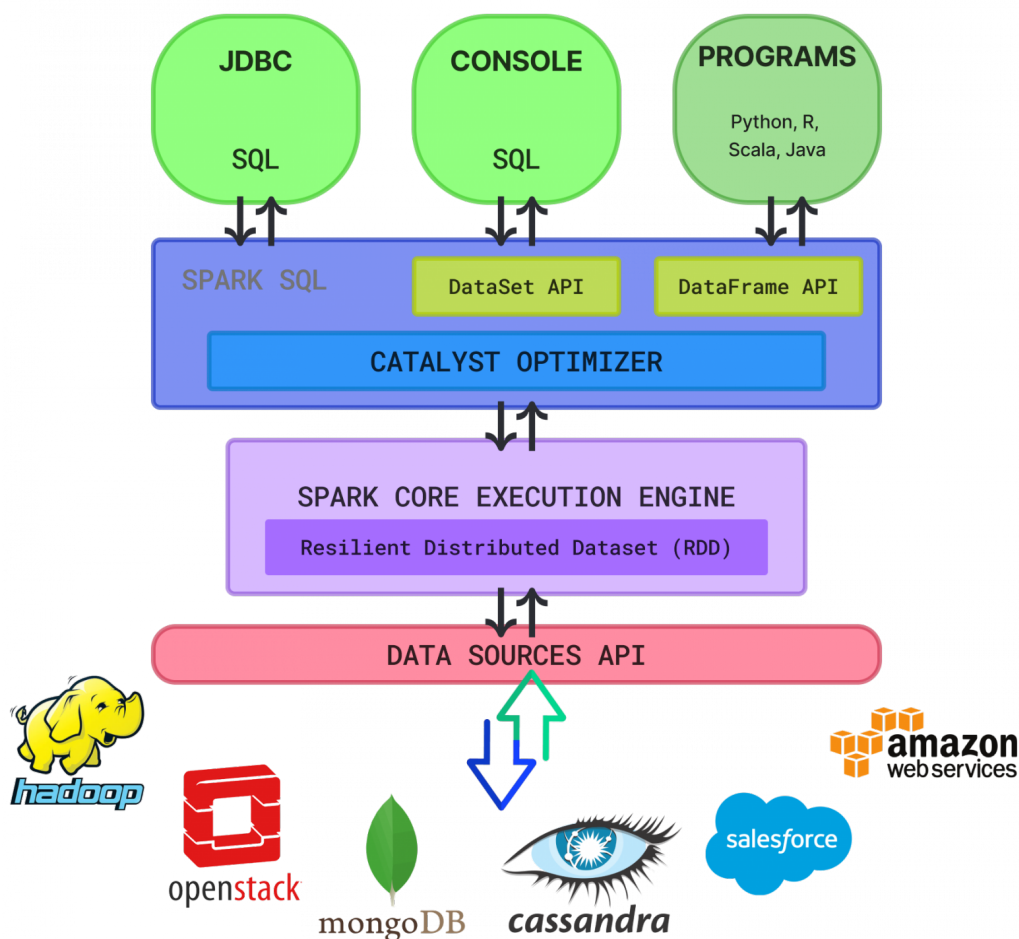


Рис.1.2. Экосистема Apache Spark

Экосистема Apache Spark состоит из следующих основных компонентов:

1. **Spark Core Execution Engine** - движок исполнения Apache Spark. Он отвечает за планирование задач и мониторинг ошибок при работе с пакетными вычислениями. На уровне SCEE данные представлены структурой данных RDD. Это абстракция, которая взаимодействует с неструктурированными наборами данных, хранящимися на разных узлах кластера. RDD состоит из набора данных, которые могут быть разбиты на блоки. Блоком, или партицией, можно считать цельную логическую неизменяемую часть данных, создать которую можно в том числе посредством преобразования уже существующих блоков. Именно RDD является универсальной программной абстракцией, над которой производятся параллельные вычисления. RDD является распределенным набором данных, что означает, что он физически разделен на части (партиции), которые распределены по различным узлам в кластере. Каждый узел кластера обрабатывает только ту часть RDD, которая хранится на этом узле.



2. **Data Sources API** - интерфейс для взаимодействия с хранилищами данных. Имея широкий набор методов для взаимодействия с разными распределенными хранилищами (Hadoop, OpenStack, MongoDB, Cassandra, Salesforce, Amazon S3), Spark сохраняет универсальность, так как вне зависимости от хранилища данные на уровне SCHE хранятся в экземпляре структуры данных RDD.
3. **Spark SQL** - модуль для работы с структурированными данными. Он позволяет выполнять SQL запросы к данным, представленными в виде DataFrames и Datasets, с возможностью оптимизации запросов благодаря встроенному оптимизатору SQL запросов Catalyst Optimizer.
4. **DataSet API и DataFrame API** - библиотеки высокого уровня, предоставляющие удобные абстракции для обработки данных. DataFrame - таблица, блоки которой обрабатываются на разных узлах кластера. DataFrame может быть представлен только в виде структурированных данных. Данные представлены именованным набором столбцов, напоминая таблицы в реляционных БД. DataSet - набор строк, разбитых на блоки. При этом строки могут уже быть не структурированными. DataFrame поддерживает работу с такими форматами данных, как AVRO, JSON, HDFS, HIVE таблицы и MySQL). И DataSet и DataFrame при работе с распределенным хранилищем формируются из RDD, о которой мы говорили в первом пункте.
5. **Программы** - приложения или скрипты, написанные на поддерживаемых Spark языках, таких как Python, Scala, R и Java. Приложения как раз и выполняют задачи, относящиеся, в том числе, к пакетной обработке данных, при этом их поведение и структура абсолютно кастомизируемы, так как они работают со структурами данных такими как DataSet и DataFrame, а не с блоками данных внутри распределенной файловой системы в рамках парадигмы, как это происходит в Map Reduce.
6. **JDBC, Console** - интерфейсы для декларативных запросах на языке SQL. В SparkSQL полностью соблюдены условия стандарта SQL-92. По сути это альтернатива программным запросам на декларативном языке, знакомом каждому разработчику.

Apache Spark использует master-worker архитектуру для выполнения вычислений в распределенной среде. В этой архитектуре есть два ключевых компонента: Driver и Executors.

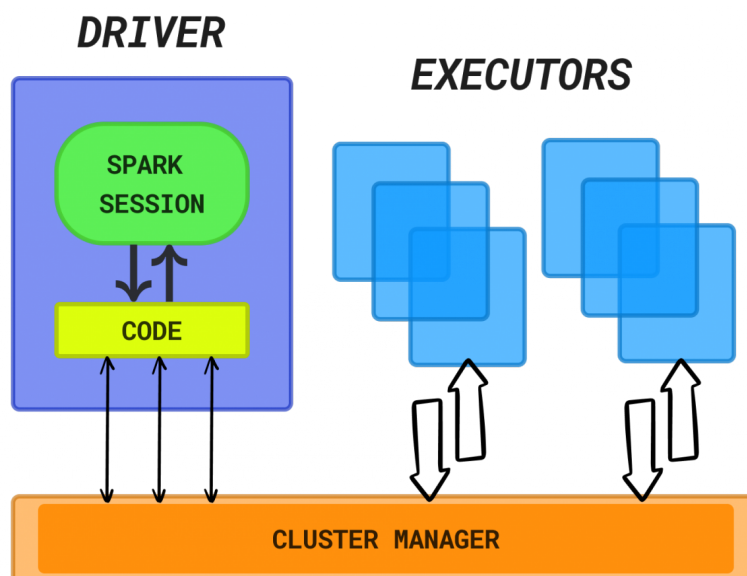


Рис.1.3. Архитектура вычислений Apache Spark

Driver — это процесс, который выполняет главную функцию приложения и создает SparkContext. SparkContext это динамическая структура данных, которая контролирует Spark сессию с кластерным менеджером и координирует выполнение задач.

Также Driver порождает программный код, написанный на SQL, Scala, Java, Python или R в логический план выполнения. Это возможно, благодаря единообразию API Spark для всех этих инструментов.

Далее Driver разбивает логический план по выполнению операций пакетной обработки данных на стадии, которые разбиваются на задачи. Задачи формулируются исходя из написанного кода и, соответственно, построенного плана.

Сформулировав задачи, Driver запускает такие процессы, как Executors, которые исполняют поставленные задачи. Executors с точки зрения архитектуры являются workerами и отчитываются перед Driver после выполнения определенной задачи. Также именно Executors выполняют чтение и запись из хранилища данных.

Существует несколько подходов к распределению Executors по узлам кластера:

1. Один Executor на Узел - такой подход обеспечивает максимальное использование ресурсов каждого узла, минимизируя потери на переключение контекста и обмен данными между различными процессами на одном и том же узле.

2. Множество Executors на Узел - такой подход может быть полезен, если узлы имеют большое количество ядер и объем памяти, и вы хотите более детально контролировать распределение ресурсов между задачами
3. Один Executor на каждую задачу - применяется либо когда задачи выполняемые Executorом очень простые, либо когда кластер состоит из очень мощных серверов, так как это наиболее ресурсоемкий подход.

Количество Executors настраивается параметрами конфигурации Spark при запуске приложения. Их зависит от количества ядер, объема памяти на каждом узле, размера данных, требований к задачам и характеристик самого кластера.

Преимущество Spark в скорости выполнения пакетной обработки достигается за счет того, что Spark был разработан с возможностью обработки данных в памяти, в отличие от Hadoop MapReduce, который для каждой операции записывает промежуточные результаты на диск. Это снижает задержку, связанную с операциями чтения/записи на диск, и значительно ускоряет обработку данных, особенно в задачах с множественными промежуточными этапами.

Также, уже упомянутый оптимизатор Catalyst Optimizer генерирует эффективный план выполнения для операций с данными. Это позволяет Spark выбирать самый оптимальный способ выполнения задачи на основе текущего состояния кластера и данных.

Кроме того, вместо жестко фиксированной последовательности операций Map и Reduce, как в Hadoop, Spark строит для каждой задачи Directed Acyclic Graph, что позволяет оптимизировать цепочку операций и уменьшить количество шагов обработки.

Таким образом, Apache Spark не только опережает Hadoop MapReduce по скорости пакетных вычислений, но и предоставляет гораздо больше интерфейсов для кастомизации операций с данными: если программа для Map Reduce пишется только на Java, то Spark поддерживает Python, Scala, R, Java и SQL, при этом разработчику не обязательно соблюдать паттерн Map Reduce, так как оптимизатор самостоятельно разделит запрос разработчика на элементарные задачи агрегации.

### *1.1.3. Apache Hive*

Мы рассмотрели два инструмента для пакетных вычислений: один из них является прародителем пакетных параллельных вычислений - Map Reduce, вто-

рой является современным стандартом для решения всех видов задач пакетных вычислений - Spark.

Рассмотрим третий инструмент, который выступает в качестве некоего усреднителя, между двумя предыдущими - Apache Hive.

Apache Hive можно описать, как API для доступа к данным внутри Apache Hadoop. Hive позволяет проводить пакетные вычисления с данными используя синтаксис SQL.

На самом деле синтаксис SQL, это не совсем точная характеристика языка, на котором делаются запросы в Hive, так как в Hive не полностью реализована поддержка стандарта SQL-92. Язык Hive носит название HiveQL и с уверенностью можно сказать, что он основан на SQL.

Hive может работать с двумя движками для проведения пакетных вычислений это уже рассмотренный нами Map Reduce и Apache Tez, который мы рассмотрим в следующем разделе. Стоит отметить, что использование Map Reduce, как движка для Hive более не считается стандартом и на момент написания работы считается устаревшей практикой. Именно Apache Tez сегодня почти всегда используется в качестве движка для Hive запросов.

Hive транслирует HiveQL код, написанный разработчиком в задачи Map Reduce, Tez или Spark. В разделе 1 этой главы мы выяснили, что с помощью Map Reduce можно сделать все, что можно сделать с помощью SQL, а значит используя Apache Hive можно решить любую задачу, относящуюся к пакетной обработке данных.

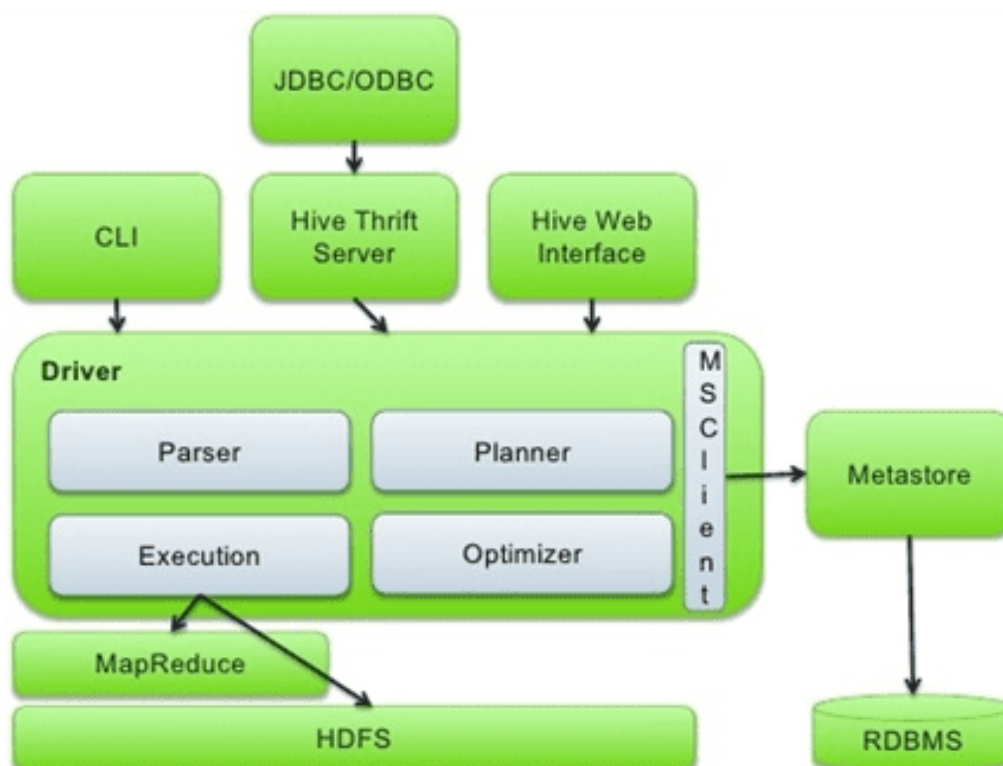


Рис.1.4. Архитектура Apache Hive

Hive обладает следующими интерфейсами для взаимодействия пользователя со средой анализа данных:

- A. **Java Database Connectivity (JDBC) и Open Database Connectivity (ODBC)** - стандартизированные API для подключения клиентских приложений к Hive для выполнения запросов SQL.
- B. **CLI** - командный интерфейс для взаимодействия с Hive напрямую из командной строки.
- C. **Hive Thrift Server** - сервер, который по-умолчанию hostится на 10000 порет и позволяет клиентам удаленно подключаться к Hive через Thrift протокол. Это обеспечивает совместимость с языками, которые поддерживают Thrift.
- D. **Hive Web Interface** - фронтенд для доступа к Hive через браузер.

Объект Driver в рамках архитектуры Hive состоит из следующих компонен-

тов:

- A. **Parser** - анализирует и проверяет синтаксис HiveQL-запросов, преобразуя их в абстрактное синтаксическое дерево.
- B. **Planner** - преобразует абстрактное синтаксическое дерево в логический план выполнения запроса, состоящие из последовательностей Map Reduce.

- C. **Optimizer** - оптимизирует логический план, используя различные оптимизации, например, меняет порядок выполнения операций для достижения лучшей производительности.
- D. **Execution** - выполняет оптимизированный логический план и возвращает пользователю результат запроса.

Также компонент Driver связан с компонентом Metastore. Он хранит метаданные о структуре и хранении в Hive, например, схемы таблиц, типы данных и другие детали. Для долгосрочного хранения метаданных в рамках одного проекта чаще всего используется реляционная база данных.

Итак, Hive это совокупность программных абстракций для создания SQL запросов к распределенной файловой системе HDFS, которая преобразует синтаксис SQL в набор задач Apache Tez, и существует как прослойка между пользователем и вычислительным движком. Соответственно при сравнении и проведении экспериментов мы будем использовать так называемый Hive on Tez. Hive в качестве интерфейса для запросов, а Tez в качестве движка для их выполнения. Map Reduce и Spark так же могут выступать движками для Apache Hive, но мы будем рассматривать именно Hive on Tez, так как и Map Reduce и Spark обладают собственными интерфейсами для написания кода. Также, именно Hive on Tez на практике используется как альтернатива Spark, так как Map Reduce не позволяет обеспечивать конкурентной со Spark производительности, а Spark обладает собственными разнообразными, развитыми и поддерживающимися интерфейсами для написания задач пакетной обработки данных.

#### ***1.1.4. Apache Tez***

Apache Tez - платформа для создания высокопроизводительных приложений пакетной обработки данных. Архитектурный подход к формированию задачи в Tez - формирование DAG графа заданий, что делает его более эффективным и производительным, нежели Map Reduce.

Начиная с версии 0.13 Apache Hive использует Tez, а не Map Reduce в качестве движка по умолчанию для исполнения запросов.



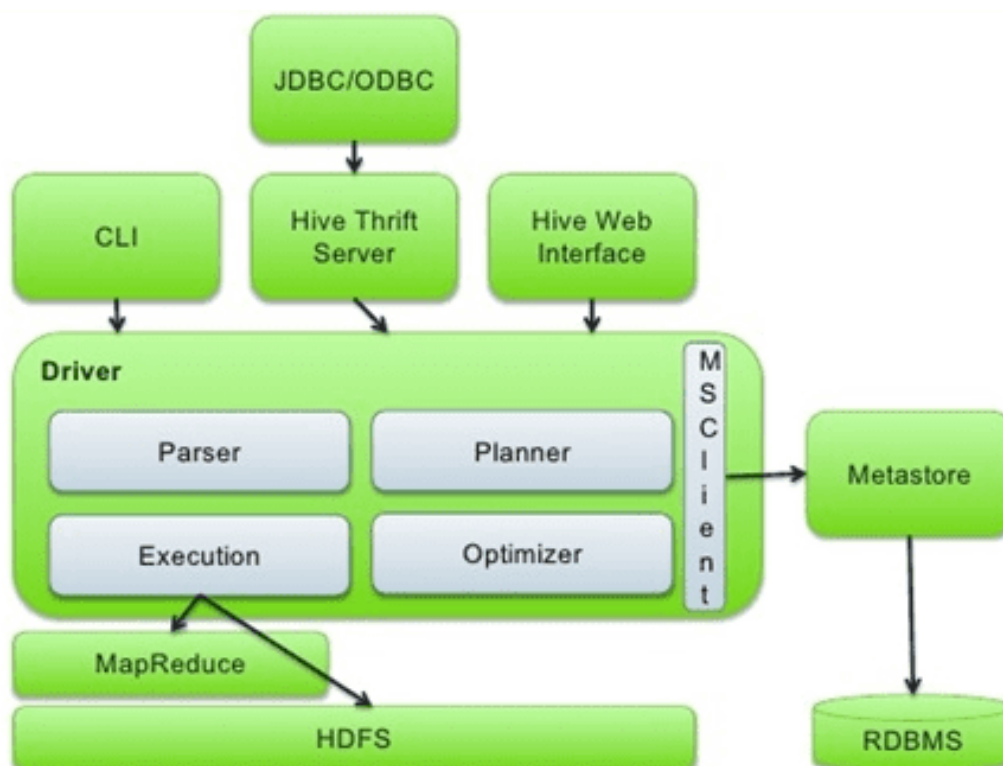


Рис.1.5. Сравнение этапов выполнения Tez и MR задач

Tez обеспечивает прирост производительности по сравнению с традиционным Hive на Map Reduce за счет исключения дополнительных операций записи на диск и чтения с диска между задачами в DAG выполнения запросов. В традиционном механизме выполнения Map Reduce вывод каждой задачи записывается в хранилище, а затем считывается из него последующими задачами. Tez позволяет избежать этого за счет облегчения доступа к общей памяти между задачами, включая RDMA.

Возможность совместного использования памяти задачами позволяет повысить пропускную способность на несколько порядков для задач, расположенных на одном узле. Тем не менее, задачи Tez DAG будут распределены по многим узлам кластера, поэтому прирост пропускной способности данных будет умеренным, поскольку не все задачи, требующие общей памяти, будут выполняться на одних и тех же узлах.

### ***1.1.5. Сравнение перечисленных инструментов***

Мы рассмотрели внутреннюю структуру инструментов, используемых для пакетных вычислений в рамках работы с большими данными.

Для иллюстративного сравнения инструментов предлагается составить следующую сравнительную таблицу:

Таблица 1.1

## Сравнение MapReduce, Spark и Hive

Критерий	MapReduce	Spark	Hive on Tez
Модель программирования	Низкоуровневая	Высокоуровневая + низкоуровневая	SQL запросы
Производительность	Средняя	Высокая	Средняя-Высокая
Удобство использования	Низкое	Высокое	Высокое
Устойчивость к ошибкам	Высокая	Высокая	Высокая
Экосистема	Широкая	Очень широкая	Широкая
Язык разработки	Java	Scala	Java

Итак, еще раз поясним выводы, сделанные в таблице. Модель программирования в Hadoop Map Reduce весьма ограничена, ведь мы должны работать в рамках одной парадигмы и на одном языке программирования. Spark Позволяет работать со множеством языков программирования и обращаться с большими данным так, как хочешь.

Высокая производительность Spark достигается за счет множества факторов, но основным является поузловая обработка данных в памяти, чего не делает MapReduce и соответственно Hive, который основан на нем. Map Reduce сохраняет промежуточные данные на диск, во время выполнения пакетных вычислений. Стоит также учитывать, что производительность Hive зависит от движка, на котором он выполняет вычисления. Существуют неофициальные версии Hive, позволяющие ему работать на движке Tez или даже на основе Spark.

Удобство использования Map Reduce оказывается значительно ниже конкурентов, так как ты можешь писать Map Reduce запросы исключительно на Java, при это в Spark ты можешь работать как с декларативным, так и с императивным подходом, причем на нескольких языках. Hive также дает возможность работать по сути с SQL, который знаком почти каждому разработчику.

Устойчивость к ошибкам у всех инструментов достаточно высокая, но поддерживается по-разному. Map Reduce работает напрямую с HDFS, архитектура которой предусматривает отказоустойчивость, а Spark имеет дополнительную



программную абстракцию RDD, в назначение которой, помимо прочего, входит мониторинг ошибок при вычислениях на узлах кластера.

Экосистема Spark невероятно Широка, в ней присутствуют инструменты для работы с машинным обучением и графовыми структурами данных, а также инструмент для потоковой обработки данных Spark Streaming. Тем не менее, Map Reduce, как составная часть Hadoop, тесно интегрирован с его экосистемой, в которой присутствуют альтернативы для инструментов Spark, такие как Apache Flink, Apache Storm и т.д. То же самое можно сказать и о Hive.

## 1.2. Инструменты для потоковых вычислений

Потоковая или пошаговая обработка данных - технология, предназначенная для обработки данных в реальном времени по мере их поступления. В отличие от пакетной обработки, где данные обрабатываются в больших объемах после их накопления в распределенной базе данных или файловой системе за определенный период времени, потоковая обработка позволяет немедленно реагировать на события и выполнять преобразования/анализировать поступающие данные.

Основное отличие потоковой обработки от пакетной заключается в необходимости низкой задержки между операциями, так как данные поступают очень быстро и, фактически, без остановки, в то время как для пакетных вычислений задержка при выполнении операций неизбежна.

Потоковая обработка идеально подходит для мониторинга, алертинга, аналитики в реальном времени, преобразования данных перед записью и других операций, требующих быстрого отклика. Пакетная обработка лучше подходит для сложных вычислений и анализа больших объемов данных, где время отклика не является критичным.

В качестве примеров задач потоковой обработки данных можно перечислить:

1. Мониторинг финансовых транзакций в реальном времени
2. Рекомендательные системы(именно в реальном времени, например, рекомендация видео, которое посмотреть следующим на основе просмотром и других действий пользователя за последнее время)
3. Обработка логов и поиск ошибок в них с целью последующего уведомления администратора
4. Обработка изображений с камер для увеличения скорости реакции компетентных лиц

Далее мы рассмотрим несколько фреймворков с открытым исходным кодом, созданных для анализа потоков данных.

### 1.2.1. Spark Streaming

Spark Streaming – это библиотека фреймворка Apache Spark для обработки непрерывных потоковых данных, о которой мы упоминали в предыдущей главе.

Для управления потоком данных Spark Streaming использует программные абстракции, представляющие из себя дискретизированные потоки, они называются DStreams. DStreams функционируют путем создания нескольких RDD, каждый из которых представляет данные, собранные за определенный интервал времени, который задается программистом и называется batch interval. Таким образом, каждый новый RDD внутри DStream содержит данные, полученные в течение этого интервала. Как только интервал времени истекает, процесс приема данных для текущего пакета прекращается и начинается обработка данных из этого пакета.

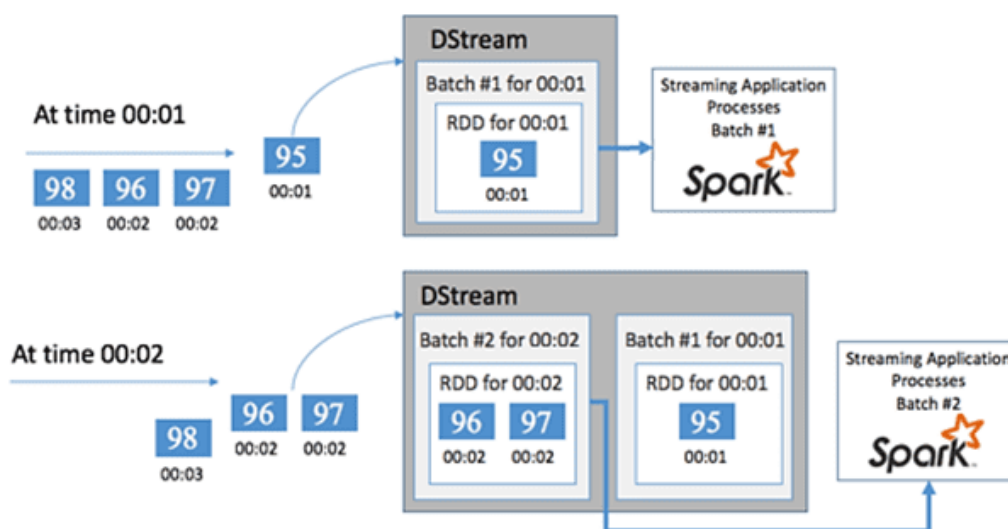


Рис.1.6. Архитектура создания пакетов в DStream

Spark Streaming позволяет выполнять различные операции с каждым RDD, что обеспечивает удобный механизм для интеграции с разнообразными внешними системами и хранилищами данных. Данные обрабатываются методом `foreachRDD()`: он последовательно применяет операции к RDD из каждого микропакета. В дополнение к этому, если требуется выполнить несколько операций над одним и тем же RDD или распределить данные по разным приемникам, данные можно кэшировать для повышения эффективности процесса.

Spark Streaming применяет механизмы параллельной обработки для повышения отказоустойчивости, что делает его более эффективным по сравнению с

традиционными методами репликации и резервного копирования данных. В случае необходимости обеспечения дополнительной отказоустойчивости используются контрольные точки, которые позволяют системе восстанавливаться после сбоев, возвращаясь к последнему сохраненному состоянию.

Кроме того, Spark Streaming поддерживает оконные операции, которые позволяют выполнять вычисления на пакетах данных, поступивших за определенный период времени. Это расширяет возможности аналитики, позволяя реализовывать функции, такие как агрегация и суммирование данных, в рамках скользящего окна. Эти окна могут пересекаться, что позволяет текущему окну использовать промежуточные вычисления из предыдущего, ускоряя тем самым процесс оконных вычислений.

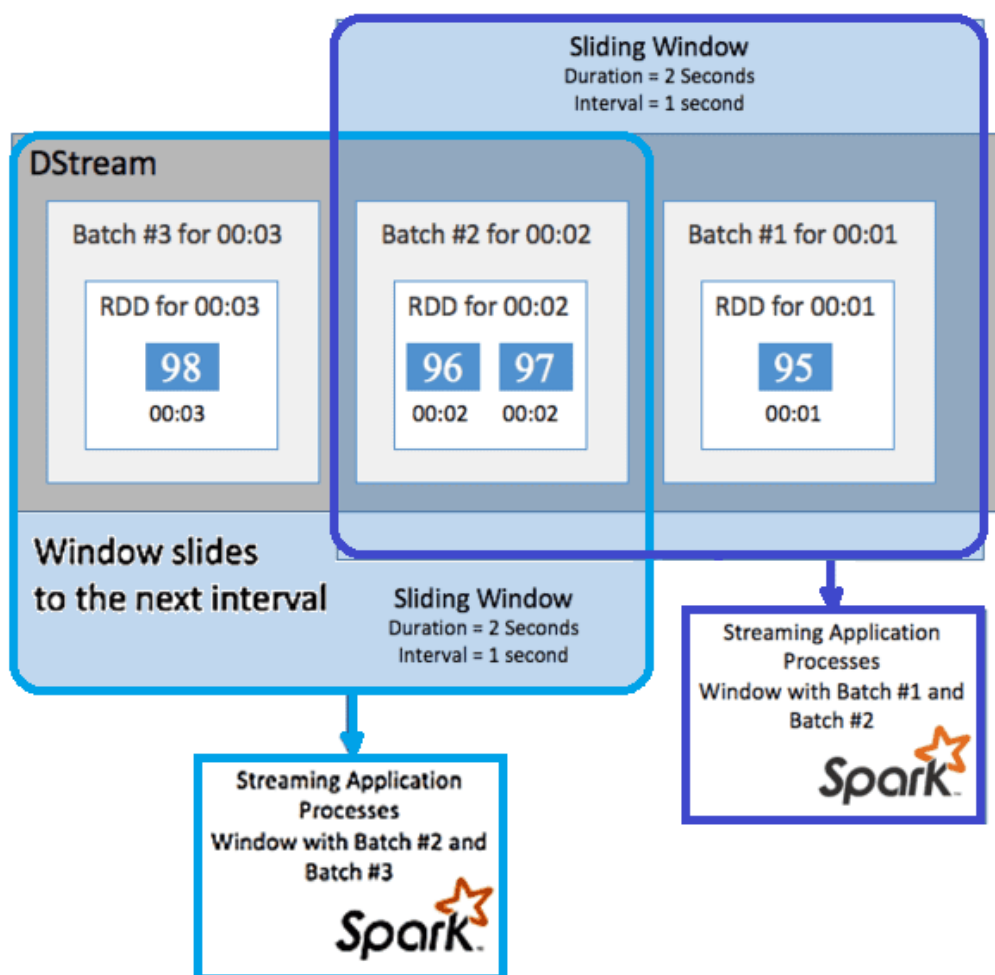


Рис.1.7. Схема выполнения оконных операций в Spark Streaming

Таким образом, Spark Streaming имеет достаточно сложную архитектуру, безусловно, имеющую возможности для работы с потоковыми вычислениями. Тем не менее, нельзя назвать архитектуру Spark Streaming полностью соответствующей всем реалиям потоковой обработки данных, так как в библиотеке используется

микро-пакетный подход, данные все равно накапливаются в пакеты, хоть и очень быстро и размер пакетов очень небольшой.

В следующем разделе мы рассмотрим инструмент, который обладает более совершенной для задач потоковых вычислений архитектурой.

### *1.2.2. Apache Flink*

Apache Flink был разработан в 2010 году в Берлинском университете. На самом деле, он очень похож на Apache Spark, так как обладает практически аналогичным набором компонентов: пакетные вычисления, потоковые вычисления, машинное обучение, графовые структуры данных - инструменты для работы со всем этим есть в обоих фреймворках.

Но нас в рамках текущей главы больше интересует те компоненты Flink, которые отвечают именно за потоковую обработку данных. За потоковую обработку данных в Flink отвечает программная абстракция DataStream API, который позволяет разработчикам определять трансформации на потоках данных. DataStream представляет собой последовательность данных, которые можно преобразовывать, агрегировать или перенаправлять. Работа с данными в DataStream происходит в виде функциональных операций: map, filter, reduce и window.

Источников данных для DataStream API могут быть распределенные файловые системы, базы данных и брокеры сообщений, например, HDFS, Apache Cassandra или Apache Kafka. Интеграция с этими источниками данных осуществляется через специальные коннекторы, которые предоставляются в Flink или разрабатываются сообществом. Потоки данных из этих источников поступают в Flink в виде непрерывных потоков событий.

После получения исходных потоков данных Flink обрабатывает их с использованием заданных программистом функциональных операций. Обработка данных организована в виде ориентированного ациклического графа (DAG), где вершины графа представляют операции над данными, а ребра — потоки данных между этими операциями. Это позволяет выполнить параллельную и распределенную обработку на узлах кластера.

В Flink реализован механизм exactly-once, что означает, что каждая элементарная единица потока данных будет программно обработана ровно один раз, причем в случае падения системы не возникнет ни данных, которые не будут обработаны, ни данных, которые будут обработаны несколько раз. С помощью

такого механизма целостности обработки данных достигается как отсутствие лишнего влияния одних и тех же данных, так и отсутствие влияния от потерянных данных на результат потоковых вычислений.

Обработанные данные могут быть отправлены в различные приемники или хранилища данных, такие как базы данных, распределенные файловые системы или другие системы для дальнейшей обработки или хранения. Как и источники данных, для внешних хранилищ данных в Flink предусмотрены различные API

Flink поддерживает сложные операции управления состоянием в приложениях потоковой обработки. Состояние можно сохранять и восстанавливать, что важно для обеспечения отказоустойчивости и восстановления после сбоев. Для этого Flink использует механизм контрольных точек, который регулярно сохраняет состояние всей обработки в надежное хранилище. В случае сбоя Flink может восстановиться из последней контрольной точки, обеспечивая точное восстановление без потери данных.

Flink DataStream API также включает возможности для оптимизации выполнения потоковых приложений. Это включает оптимизацию потока данных, где система автоматически настраивает план выполнения задач в зависимости от нагрузки и ресурсов кластера. Оптимизация во время выполнения позволяет Flink адаптироваться к изменяющимся условиям работы, например, при изменении объема входящих данных или при сбоях в узлах кластера.

Как и Spark Streaming, Flink поддерживает сложные потоковые операции, такие как управление временными окнами, соединения потоков и агрегации.

В целом, Apache Flink представляет собой мощную и гибкую систему для реализации потоковых и пакетных приложений обработки данных, обеспечивая эффективные и масштабируемые решения для современных задач анализа больших данных.

### ***1.2.3. Сравнение Spark Streaming и Apache Flink***

Мы перечислили архитектурные и функциональные особенности Spark Streaming и Apache Flink. Было выяснено, что оба фреймворка могут на практике эффективно применяться для решения задач потоковой обработки данных. В ходе анализа были выявлены следующие сходства между рассмотренными фреймворками:

1. Lambda-функции: оба фреймворка используют архитектуру lambda-функций - универсальных функций, которые применяются к каждой элементарной единице потоковых данных.
2. Инструменты для машинного обучения и аналитики: оба фреймворка имеют встроенный функционал для потоковой обработки данных не только стандартными агрегационными функциями, но и методами машинного обучения.
3. Отказоустойчивость и масштабируемость: оба фреймворка при работе с большими данными могут использовать, как в качестве источников данных, так и в качестве хранилища данных распределенные файловые системы, которые функционируют на различных узлах кластера хранения.

При этом между ними существуют значимые архитектурные различия.

Таблица 1.2

Сравнение Spark Streaming и Apache Flink

Критерий	Spark Streaming	Apache Flink
Модель обработки	Микропакетная обработка	Потоковая обработка
Производительность	Высокая с задержками	Очень высокая с минимальными задержками
Удобство использования	Высокое	Высокое
Поддержка реального времени	Нет (микро-пакеты)	Да
Экосистема и интеграция	Широкая	Широкая, но меньше чем у Spark
Управление состоянием	Ограниченное	Расширенные возможности

Spark Streaming использует микропакетный подход для обработки потоков данных. Это означает, что данные собираются и обрабатываются в небольших пакетах на регулярной основе. Apache Flink реализует истинную потоковую обработку, обеспечивая непрерывную и мгновенную обработку данных по мере их поступления. Это подход позволяет Flink достигать значительно более низкой



задержки и подходит для приложений, требующих обработку данных в режиме реального времени.

Spark Streaming предоставляет хорошую производительность для большинства задач, но из-за своего микропакетного подхода может страдать от высокой задержки в сценариях, требующих очень низкого времени реакции. Apache Flink предлагает высочайшую производительность с минимальной задержкой, что делает его идеальным выбором для задач, требующих непрерывной и мгновенной обработки потоков данных.

Обе системы имеют высокий уровень удобства использования с подробной документацией и большим сообществом. Однако, начать работать с Spark Streaming может быть проще для тех, кто уже знаком с экосистемой Apache Spark, которая является более популярной, нежели экосистема Apache Flink.

Spark Streaming обеспечивает близкую к реальному времени обработку данных, но все же зависит от интервала микропакетов. Flink предлагает поддержку обработки данных в реальном времени благодаря своей потоковой архитектуре, что делает его предпочтительным выбором для приложений, требующих мгновенной реакции.

Apache Flink предоставляет более развитые и гибкие инструменты для управления состоянием. Это включает поддержку асинхронных и инкрементных контрольных точек (checkpoints), которые минимизируют влияние на производительность и позволяют точно восстановить состояние после сбоев.

Выбор между Spark Streaming и Apache Flink зависит от специфических требований проекта и предпочтений команды разработчиков. Spark Streaming интегрируется с многими другими компонентами Spark и предлагает широкую экосистему для анализа данных, что делает его идеальным для проектов, где необходимы комплексные аналитические возможности на больших данных, как потоковые, так и пакетные. Flink же предлагает более высокую производительность и лучшие возможности для реальной потоковой обработки, что делает его лучшим выбором для систем, требующих максимально низкой задержки и сложного управления состоянием в реальном времени.

## **ГЛАВА 2. ПРОЕКТИРОВАНИЕ, РАЗВЕРТЫВАНИЕ И РАЗРАБОТКА СИСТЕМЫ ДЛЯ ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТАЛЬНОГО СРАВНЕНИЯ ВЫБРАННЫХ СРЕДСТВ РАБОТЫ С БОЛЬШИМИ ДАННЫМИ**

В этой главе мы выберем источник данных, который мы будем использовать для экспериментов в рамках экспериментального сравнения фреймворков как для пакетной, так и для потоковой обработки данных. Определившись с источником данных и конкретизировав способ получения данных мы выдвинем требования к системе, способствующие проведению достоверных и разнообразных экспериментов, спроектируем ее, после чего развернем все необходимые для тестирования компоненты в изолированной и ограниченной по ресурсам среде исполнения, а также создадим некоторые необходимые инструменты с нуля.

### **2.1. Выбор источников данных и их формат**

Во введении настоящей работы мы определили следующие свойства больших данных:

1. Данные разнообразны и не имеют общего формата.
2. Данные поступают с большой скоростью.
3. Данные поступают в больших объемах.

На практике существует очень много данных, которые подпадают под эти критерии, наиболее очевидными примерами могут выступать твиты, данные о посещаемости веб-страниц, данные метеорологических данных и так далее.

Обратимся к кандидатским диссертациям, которые исследуют различные аспекты работы с большими данными, для того, чтобы определить, какие данные с точки зрения исследования можно считать "большими".

В работе "Семантические технологии больших данных для многомасштабного моделирования в распределенных вычислительных средах" исследователь А. А. Вишератин разрабатывает программные средства для оптимизации исполнения многомасштабных вычислительных приложений в распределенных средах. Для тестирования разработанного программного обеспечения исследователь использует записи пользователей из социальной сети Instagram [16].

В этой работе исследователь занимается оптимизацией задач пакетной обработки данных. В нашей работе также будут сравниваться фреймворки для



потоковой обработки данных, поэтому в требования к источнику данных в рамках нашей работы также нужно включить их поступление в реальном времени.

В диссертации "Разработка и исследование моделей для оптимизации информационного потока при интерактивном анализе больших данных в геоинформационных системах данные анализируются по следующим показателям: объем (volume), скорость (velocity) и разнообразие (variety) [14].

В качестве источника данных мы предлагаем чаты стриминговой платформы Twitch[30]: стример ведет прямую трансляцию видео, а зрители в чате могут общаться с ним, при этом и зрители и стример видят содержимое чата. Проанализируем выбранный источник данных по этим критериям.

Объем и скорость потока данных в рамках выбранного источника, в сущности, зависят от количества зрителей на стриме, а также "режима чата"[10]. Тем не менее, существуют системы для оценки общей статистики сообщений в чатах на Twitch. Платформа StreamElements[9], которая собирает глобальную статистику по всем стримам на Twitch сообщает, что с 9 января 2016 года всего было отправлено 166 084 111 740 сообщений. При средней длине сообщения в 30 символов, объем таких данных без учета метаданных о сообщениях(отправитель, стрим в ходе которого оно было отправлено, временная метка отправки) составил бы 4.53157 петабайт. Средняя скорость потока сообщений со всех стримов на платформе составляет от 1100 до 26000 сообщений в секунду(в зависимости от времени дня, так как все таки на платформе доминируют англоязычные стримеры) [11]. К примеру, рекордной нагрузкой в Twitter считается 750 твитов в секунду [13], и твиты часто используются в научных работах в качестве примера источника больших данных [17, 23, 15]. Таким образом рассматриваемый источник данных соответствует критериям Volume и Velocity.

На счет разнообразности данных можно отметить, что данные, которые поступают в чат - это текстовые данные. Текстовые данные сами по себе являются достаточно разнообразными, помимо этого в сообщениях из чатов также могут присутствовать Twitch Emotes(аналог смайликов) и иные различные способы выражения эмоций с помощью символов, которые также должны быть подвержены обработке или фильтрации. По сути формат наших данных аналогичен твитам или постам пользователей в Instagram, таким образом, является достаточно разнообразным для восприятия в качестве одного из источников больших данных.

Таким образом, данные из выбранного источника соответствуют всем рассматриваемым критериям больших данных.

Что же касается практической реализации доступа к данным из выбранного источника, Twitch предоставляет возможности для подключения к чату двумя протоколами: Websocket и IRC. Обе технологии позволяют получать и отправлять текстовые сообщения, формат сообщения внутренней структурой переданной по сокету строки. Так как в нашем случае отсутствует необходимость соблюдать требования для интеграции передатчика данных в среду Web приложения, мы делаем выбор в пользу IRC сокета. Приведем пример структуры сообщения, которое приходит по IRC сокету:

```
:gateban_228!gateban_228@gateban_228.tmi.twitch.tv PRIVMSG #simple
:@simple кс ночью будет?
```

- gateban\_228!gateban\_228@gateban\_228.tmi.twitch.tv - идентификатор пользователя, отправившего сообщение
- PRIVMSG - тип сообщения
- #simple - id канала, на котором проходит данный стрим
- @simple кс ночью будет? - сообщение, где @simple обращение в чате к конкретному пользователю, в данном случае к самому стримеру

Итак, такое сообщение - элементарная единица данных в рамках нашей системы. Twitch предоставляет возможность любому пользователю просматривать чат, даже пользователю, у которого нет аккаунта Twitch. Подключение к Websocket соединению в таком случае происходит с указанием случайных параметров в сообщениях инициализации:

```
Data
↑ CAP REQ :twitch.tv/tags twitch.tv/commands
↑ PASS SCHMOOPIIE
↑ NICK justinfan38959
↑ USER justinfan38959 8 * :justinfan38959
↓ :tmi.twitch.tv CAP * ACK :twitch.tv/tags twitch.tv/commands
↓ :tmi.twitch.tv 001 justinfan38959 :Welcome, GLHF! :tmi.twitch.tv 002 justinfan38959 :Your host is tmi.twitch.tv :...
↑ JOIN #dota2_paragon_ru
↓ :justinfan38959!justinfan38959@justinfan38959.tmi.twitch.tv JOIN #dota2_paragon_ru @emote-only=0;follow...
↓ :justinfan38959.tmi.twitch.tv 353 justinfan38959 = #dota2_paragon_ru :justinfan38959 :justinfan38959.tmi.twit...
```

Рис.2.1. Инициализационные сообщения, отправляемые клиентом при подключении к вебсокету чата

В нашем случае, так как мы сделали выбор в пользу IRC сокета, мы не можем указать случайные данные при подключении, хотя инициализация подключения и происходит похожим образом:

```

def irc_connection(self, channel):
    # server = 'irc.chat.twitch.tv'
    # port = 6667
    # nickname = 'FrolovGeorgiy'
    # token = 'oauth:ryh3hgq656pi5c34ki2jjkqfjakh4f'
    irc_sock = socket.socket()
    irc_sock.connect((self.server, self.port))
    irc_sock.send(f"PASS {self.token}\n".encode('utf-8'))
    irc_sock.send(f"NICK {self.nickname}\n".encode('utf-8'))
    irc_sock.send(f"JOIN {channel}\n".encode('utf-8'))
    print(f"Connected to IRC channel {channel}")

```

Рис.2.2. Метод, выполняющий подключение к IRC сокету

При подключении указывается url, порт(6667, т.к. не используется SSL[27]), никнейм на твиче и аутентификационный токен, который можно найти в настройках аккаунта на Twitch [28]. В качестве программного объекта для инициализации подключения используется объект модуля socket, который, в числе многих протоколов поддерживает, поддерживает и протокол IRC [22].

Такие IRC подключения и станут элементарными семантическими идентичными источниками данных в рамках нашей системы.

## 2.2. Формулирование требований к системе

В данной главе мы сформулируем требования к системе для проведения экспериментального сравнения средств для работы с данными в веб приложениях. Итак, для разрабатываемой системы были выдвинуты следующие требования:

1. *Возможность модификации нагрузки* - скорость потока сообщений, поступающих в систему, а также размер выборки, на которой производятся эксперименты, связанные с пакетными вычислениями должны быть настраиваемыми.
2. *Доступ к источнику данных для нескольких процессов* - источники данных как для потоковых, так и для пакетных экспериментов должны быть доступны одновременно для нескольких программ для того, чтобы эксперимент проводился на одних и тех же данных.
3. *Возможность сбора разнообразных данных о результатах эксперимента*, как в момент его проведения, так и после, так как без данных о

результатах эксперимента невозможно будет формализовать и представить его результаты.

4. *Независимость системы от условий конкретного эксперимента* или, если угодно, возможность выполнять любые задачи в рамках потоковой и пакетной обработки большими данными с безусловным соблюдением предыдущих трех требований.

Перечисленные условия позволят подтверждать/опровергать выдвинутые гипотезы, подкрепляя выводы, сделанные о верности гипотез убедительными данными. Далее подробнее обсудим выдвинутые требования.

Для соблюдения объявленных требований компоненты системы должны соответствовать некоторым положениям. Очевидно, что соблюдение требований может быть достигнуто по-разному, далее будет описан один из вариантов архитектуры системы, которая будет отвечать выдвинутым требованиям.

Возможность модификации нагрузки будет достигаться наличием возможности удалять/добавлять семантически идентичные элементарные источники данных (сами данные при этом различны, идентичен тип источника): больше источников - больше данных, меньше источников - меньше данных. При такой реализации возможности модификации объема потока данных требуется объединить элементарные источники в один основной, с которым и будут взаимодействовать средства для работы с большими данными.

Наличие основного источника данных, объединяющего элементарные однотипные источники данных подразумевает возможность доступа нескольких процессов именно к этому источнику. Это значительно удобнее, чем если бы средства для работы с большими данными подключались к каждому из элементарных источников. Тем более, что существует ПО, эффективно выполняющее роль "бутылочного горлышка" в таких случаях. Для пакетной обработки данных таким источником данных должно стать распределенное хранилище.

Возможность сбора данных о результатах работы средств с большими данными чаще всего реализована на уровне самих средств работы с большими данными. Существуют Web интерфейсы, которые хранят информацию о как уже выполненных, так и выполняемых в текущий момент задачах. Для средств работы с большими данными, которые планируется сравнивать, а именно Apache Spark, Apache Flink, Apache Hive, Hadoop MapReduce существование таких интерфейсов было подтверждено в ходе предварительного анализа документации [3, 6, 20].

Независимость эксперимента достигается стабильным функционированием компонентов системы вместе и корректной работой хранилища. При корректной работе хранилища, в условиях пакетной обработки данных проведение одного и того же эксперимента будет возможно на идентичных данных несколько раз, что позволит подкрепить или опровергнуть результаты проведения первого эксперимента. При работе с экспериментами в рамках потоковой обработки идентичность данных при сравнении двух фреймворков обеспечивается возможностью подключения нескольких процессов к основному источнику данных. Однако, при повторении эксперимента данные будут отличаться, тем не менее, учитывая настраиваемость скорости потока сообщений в основном источнике компенсирует их внутренние отличия - данных будет столько же.

Разработав такую систему, у нас появится возможность проводить любые эксперименты в рамках работы с большими данными, при этом быть уверенными в адекватности условий проведения эксперимента, а также иметь возможность детально проанализировать их результаты с помощью Web интерфейсов фреймворков.

В данной главе мы рассмотрим различные этапы разработки системы для экспериментального анализа средств работы с большими данными, а также выделим нюансы и сложности, с которыми мы столкнулись в процессе конфигурации компонентов системы.

## **2.3. Выбор вспомогательных средств обработки и хранения данных**

### ***2.3.1. Система хранения данных***

Для проведения экспериментов потоковой обработки данных нам требуется технология, которая будет где-то хранить данные, которые мы будем анализировать.

Хранилище, используемой в нашей системе должно выполнять следующие требования:

- А. Высокая скорость записи - в ходе потоковой обработки данных информация в систему будет поступать с высокой скоростью и хранилище должно иметь возможность быстро сохранять эту информацию.
- В. Актуальность в сфере хранения больших данных - при проведении экспериментов важно, чтобы мы использовали хранилище, которое на практике используется для этих целей, так мы можем быть уверены, что не

создаем "бутылочного горлышка" для нашей системы и измеряем именно показатели обработки данных, а не I/O bound нагрузку на хранилище.

- C. Интеграция со всеми выбранными средствами - хранилище должно обладать интерфейсами для взаимодействия со всеми выбранными фреймворками, как для пакетной, так и для потоковой обработки данных.
- D. Инструменты мониторинга накопления информации в хранилище - так как в ходе пакетных экспериментов нам нужно будет сформировать датасеты определенного размера, мы должны будем где-то их хранить, для этого само хранилище должно обладать функционалом для отслеживания информации такого рода.
- E. Низкое потребление вычислительных ресурсов - важно, чтобы СХД не сильно нагружала машину, так как это будет ставить под сомнение достоверность всего нашего экспериментального анализа.

Кроме того, нам необходимо будет установить Apache Hadoop, так как его составной частью является Hadoop Map Reduce - один из методов пакетной обработки данных, производительность которого мы будем анализировать.

Учитывая все вышеназванные требования, а также то, что мы обязаны устанавливать Hadoop, выбор хранилища падает на HDFS.

HDFS (Hadoop Distributed File System) — это распределенная файловая система, разработанная для надежного хранения и управления большими объемами данных. Она легко масштабируется горизонтально, добавляя новые узлы для увеличения емкости и производительности.

HDFS является одним из основных компонентов Apache Hadoop, поэтому все средства, выбранные нами для экспериментального сравнения обладают API для интеграции с HDFS. Также HDFS была разработана для работы на низкопроизводительных серверных машинах, так что не будет сильно нагружать систему в ходе экспериментов.

Таким образом, HDFS отвечает всем выдвинутым требованиям и может быть использована в качестве хранилища данных в нашей системе.

### ***2.3.2. Брокер сообщений***

Одно из требования к системе - наличие основного источника данных, к которому могут иметь доступ несколько задач потоковой обработки данных в одно и то же время. При этом нам нужно быть уверенными, что все данные,



которые мы высылаем в источник будут получены потребителем(работой потоковой обработки).

Такое требование часто возникает при разработке, при этом не только при разработке приложений, связанных с большими данными и на рынке существуют решения с открытым исходным кодом, предназначенные для этих задач.

Такие решения называются брокерами сообщений. Брокер сообщений — это промежуточное программное обеспечение, которое позволяет различным системам и приложениям обмениваться сообщениями. Он обеспечивает надежную доставку сообщений, управление очередями и маршрутизацию, а также позволяет асинхронную связь между различными компонентами системы. В качестве наиболее распространенных решений можно перечислить Apache Kafka, RabbitMQ и ActiveMQ.

По аналогии с СХД, к брокеру сообщений мы также выдвигаем требование по интеграции с анализируемыми в ходе экспериментов средствами потоковой обработки данных.

В том числе потому, что Kafka так же, как и другие анализируемые фреймворки является проектом верхнего уровня репозитория Apache, он тесно интегрирован и с Apache Spark, а соответственно и Spark Streaming, и с Apache Flink.

Также он обладает развитыми инструментами для отслеживания и очистки источников данных, что будет актуально для нас при проведении экспериментов.

Таким образом, в качестве основного источника данных при проведении потоковых экспериментов мы будем использовать Producer и Consumer брокера сообщений Apache Kafka.

## **2.4. Архитектура системы**

Мы выбрали внешний источник данных, обосновали его релевантность к теме нашего исследования, а также специфицировали тип подключения, а также указали все необходимые для этого подключения параметры. Теперь можно приступить к созданию внутреннего устройства системы.

Во втором разделе предыдущей главы мы переложили требования к системе на задачи программного обеспечения. Согласно этим задачам, мы можем сформировать понимание структуры системы в виде следующей диаграммы:

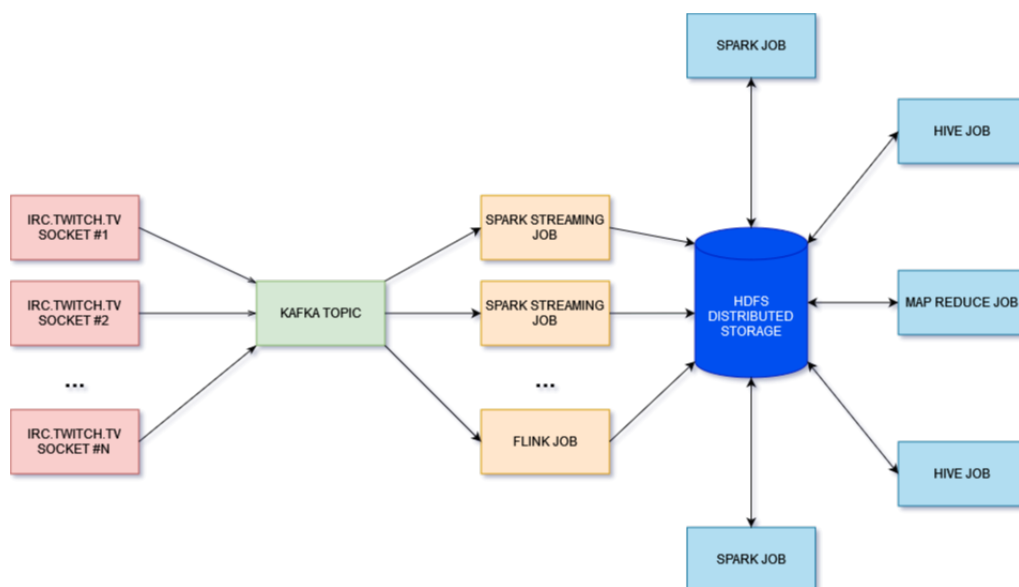


Рис.2.3. Диаграмма манипуляций большими данными в рамках системы

Итак, на диаграмме видно, что в рамках системы мы можем иметь несколько элементарных источников данных, количеством источников данных будет определяться нагрузка на часть системы, отвечающую за потоковые вычисления. Элементарные источники данных обозначены розоватыми прямоугольниками на диаграмме.

Элементарные источники данных перенаправляют данные на заранее инициализированный Kafka producer. Kafka Producer — это компонент в системе Apache Kafka, который отправляет (публикует) данные в один или несколько Kafka-брокеров. Kafka брокер — это сервер в системе Apache Kafka, который получает, хранит и раздает сообщения. Однотипные сообщения(в нашем случае сообщения из чата Twitch) в рамках Apache Kafka часто объединяются в Topic, на который могут "подписаться"потребители данных. К Topicу с сообщениями из чата может подключаться неограниченное число процессов, при этом все они будут получать информацию с низкой задержкой, и, что важно, одну и ту же информацию. Этот Kafka Topic, как раз станет основным источником данных в рамках приложения и объединит данные из несольких IRC сокетов в единый стабильный поток [22].

Процессами, которые получают данные от основного источника, т.е. являются потребителями(consumers) Topica Kafka с сообщениями из IRC сокетов, являются обработчики потоковых данных двух типов - Spark Streaming и Apache Flink. Именно эти фреймворки сравниваются нами в рамках работы. Мы можем запустить неограниченное количество процессов потоковой обработки, так как это предусмотрено внутренним устройством основного источника данных. Именно



эти процессы потоковой обработки данных преобразовывают строковые данные, которые мы получаем из IRC Socketов. До этого момента данные остаются в строковом формате. Единственное преобразование, которое с ними происходит - на этапе получения их, еще до перенаправления в Kafka Topic, мы присваиваем каждому сообщению `arrival_timestamp` - временную метку прибытия данных, с целью оценить задержку при потоковой обработке данных.

Если система функционирует и в данный момент не проводится никаких экспериментов в рамках потоковой обработки данных, то все еще могут быть запущены две работы, одна на основе Flink, другая на основе Spark Streaming. Эти работы накапливают данные для экспериментов в рамках пакетной обработки. Эти работы преобразуют данные из текстовых сообщений, которые они потребляют из `topic kafka`. Обе работы производят преобразование из формата сообщения по умолчанию, в формат для хранения в распределенном хранилище HDFS.

Фреймворки, с помощью которых мы будем обрабатывать пакетные данные, а именно, заявленные в рамках работы Hadoop MapReduce, Apache Hive и SparkSQL также могут одновременно взаимодействовать с распределенной файловой системой HDFS. Таким образом, в рамках системы обеспечивается возможность проведения экспериментов пакетной обработки данных с соблюдением указанных в предыдущей главе требований. По результатам проведения экспериментов планируется сравнить данные фреймворки по различным критериям.

## **2.5. Установка и конфигурация компонентов системы**

Мы привели диаграмму манипуляций с данными в рамках системы, и описали логику этих манипуляций. Почти все манипуляции с данными осуществляются в рамках описанных выше средств работы с большими данными. Соответственно следующий шаг в разработке нашей системы - установка компонентов системы и приведение их к состоянию, в котором возможны описанные выше манипуляции с данными, т.е. корректная конфигурация.

### **2.5.1. Среда выполнения**

Сразу заметим важную деталь развертывания системы. В ходе предварительного анализа документации используемых средств работы с большими данными было выявлено, что Apache Flink не имеет версии для ОС Windows. Со стороны

разработчиков такой шаг выглядит обоснованным, так как на поприще серверных ОС с момента его появления доминирует Linux, кроме того даже в продакшене в последнее время набирает популярность контейнеризация, так что отсутствие версии Apache Flink для Linux перестает быть проблемой для многих разработчиков. Таким образом, в связи с недоступностью Apache Flink на Windows, при создании разворачивании системы мы могли пойти двумя путями: использование Docker контейнера, использование WSL(Windows Subsystem for Linux). Использование Docker контейнера и использование WSL в нашем случае почти одно и то же, так как на Windows для контейнеризации используется именно среда WSL, тем не менее, найти подходящий, заранее сконфигурированный контейнер было бы невозможно, так как целью нашей системы является экспериментальное сравнение синонимичных инструментов, которые в среде реальной разработки вряд ли будут существовать в одной системе, так как выполняют, по сути, одни и те же задачи. Так что было принято решение использовать WSL и настроить среду, в которой будет выполняться система "под себя". В качестве дистрибутива была выбрана Ubuntu 22.04 LTS, как наиболее широко используемая с долговременной поддержкой конфигурация Linux.

Эксперименты будут проводиться, и, соответственно, система будет функционировать на устройстве с 8ГБ оперативной памяти(такое ограничение выставлено в конфигурации WSL) и с процессором Intel Core i7 с частотой 1.8ГГц и 4-мя ядрами. В целом такая аппаратная конфигурация воспроизводит конфигурацию не очень мощного и при этом недорогого в аренде сервера.

Базовым компонентом системы, безусловно, является Apache Hadoop. Это объясняется тем, что именно Hadoop включает в себя HDFS, распределенную систему хранения данных, без которой невозможно будет провести, как минимум эксперименты пакетной обработки данных, а результат экспериментов потоковой обработки данных потеряет возможность быть записанным куда-либо.

Стоит отметить, что при установке мы старались выбирать последние версии всех используемых программных средств, так как это основное условие для того, чтобы иметь возможность претендовать на актуальность результатов проведенных экспериментов.

### 2.5.2. *Apache Hadoop*

Последняя версия Apache Hadoop на момент написания работы - 3.4.0, она и была установлена на наш WSL. Для успешного взаимодействия с HDFS, которая входит в дистрибутив Hadoop, требовалось сконфигурировать систему особым образом. Прежде всего, было создан отдельный пользователь `hadoop`, с правами суперпользователя для удобства администрирования системы.

Перед установкой непосредственно Hadoop, с помощью утилиты `sudo` на WSL был установлен JDK версии 11:

```
| sudo apt install openjdk-11-jdk
```

Рис.2.4. Команда для установки OpenJDK 11 версии

Версия JDK выбрана не случайно последние версии Apache Spark, Apache Kafka и Apache Flink, которые мы и намереваемся использовать в работе требуют именно этой версии JDK установленной на устройстве. Кроме того JDK до сих пор является LTS версией, поэтому можно выдвигать предположения об исправлении большинства неисправностей, с которыми мы могли бы столкнуться при установке и конфигурации вышеперечисленных инструментов.

Установка Hadoop была произведена путем скачивания из репозитория Apache Software архива с бинарниками и последующей его распаковки, как в прочем и установка всех компонентов системы. Корневая директория Hadoop имеет путь `/home/hadoop/hadoop`, это же значение имеет переменная среды `$HADOOP_HOME`.

В рамках системы мы не используем все поставляемые в рамках Hadoop инструменты. В частности мы не используем менеджер ресурсов YARN, так как все сервисы выполняются как standalone кластеры и коммуницируют между собой по собственным протоколам (`spark`, `hdfs`, `flink`) и так далее, использование такого метода исполнения программных компонентов позволяет изолировать каждый компонент системы и выделить сравнимым компонентам одинаковое количество аппаратных ресурсов, чтобы сравнение их производительности было более объективным.

Запуск и остановка HDFS выполняются скриптами `start-dfs.sh` и `stop-dfs.sh`, находящиеся в папке `sbin` корневого каталога `hadoop`.

```

hadoop@LAPTOP-AM1MP0U8:~/hadoop/sbin$ ls
distribute-exclude.sh  mr-jobhistory-daemon.sh  start-dfs.sh  stop-balancer.sh  workers.sh
FederationStateStore  refresh-namenodes.sh    start-secure-dns.sh  stop-dfs.cmd  yarn-daemon.sh
hadoop-daemon.sh      start-all.cmd           start-yarn.cmd  stop-dfs.sh  yarn-daemons.sh
hadoop-daemons.sh    start-all.sh           start-yarn.sh   stop-secure-dns.sh
httpfs.sh             start-balancer.sh       stop-all.cmd   stop-yarn.cmd
kms.sh               start-dfs.cmd           stop-all.sh    stop-yarn.sh

```

Рис.2.5. Службы Apache Hadoop 3.4.0

Наиболее интересные среди конфигурационных параметров Hadoop представлены ниже:

```

<configuration>
<property>
  <name>fs.default.name</name>
5  <value>hdfs://0.0.0.0:9000</value>
  <description>The default file system URI</description>
</property>
<property>
10  <name>hadoop.proxyuser.hadoop.hosts</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hadoop.groups</name>
  <value>*</value>
15 </property>
</configuration>

```

Рис.2.6. Файл конфигурации Apache Hadoop

В приведенной конфигурации Hadoop свойство `<name>fs.default.name</name>` с значением `<value>hdfs://0.0.0.0:9000</value>` указывает URI для файловой системы по умолчанию, которой будет пользоваться Hadoop. 0.0.0.0 обозначает, что Hadoop будет слушать все сетевые интерфейсы на порту 9000.

Свойство `<name>hadoop.proxyuser.hadoop.hosts</name>` с значением `<value>*</value>` разрешает пользователю `hadoop` использовать прокси на всех хостах. Это свойство управляет разрешениями для проксирования, позволяя пользователю `hadoop` передавать запросы от имени других пользователей на любом хосте, что необходимо для обеспечения гибкости и управления доступом в распределенной среде.

Аналогично, свойство `<name>hadoop.proxyuser.hadoop.groups</name>` с значением `<value>*</value>` разрешает пользователю `hadoop` использовать прокси для всех групп. Это свойство расширяет контроль над разрешениями для проксирования, позволяя пользователю `hadoop` передавать запросы от имени пользователей,

принадлежащих к любой группе, что обеспечивает еще больший уровень гибкости и управления в системе безопасности Hadoop.

Еще одним важным для нашей системы компонентом hadoop является Hadoop Namenode Web UI, который является по сути веб интерфейсом к WebHDFS REST API. По умолчанию он hostится на порте 9870 и предоставляет следующую информацию о состоянии Data Nodeов:

In operation

DataNode State: All

Show: 25 entries

Search:

Node	Http Address	Last contact	Last Block Report	Used	Non DFS Used	Capacity	Blocks	Block pool used	Block pool usage StdDev	Version
✓/default-rack/LAPTOP-AM1MPOU8-9866 (127.0.0.1:9866)	<a href="http://LAPTOP-AM1MPOU8-9866">http://LAPTOP-AM1MPOU8-9866</a>	2s	46m	51 MB	10.84 GB	1006.85 GB	6052	51 MB (0%)	0%	3.4.0

Showing 1 to 1 of 1 entries

Previous 1 Next

Рис.2.7. Информация о Data Node, доступная в Hadoop Namenode Web UI

Еще одной важной функцией Hadoop Namenode Web UI именно для нашей системы является вкладка, в которой можно просматривать содержимое самой файловой системы, так как мы храним данные, с которыми будут взаимодействовать Spark, Hive и MapReduce процессы именно в HDFS.

Show: 25 entries

Search:

<input type="checkbox"/>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	<input type="checkbox"/>
<input type="checkbox"/>	drwxr-xr-x	hadoop	supergroup	0 B	May 20 15:43	0	0 B	_spark_metadata	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	61 B	May 20 14:52	1	128 MB	part-00000-00007fc0-8c17-4f1b-a655-857839528e0e-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	523 B	May 20 15:34	1	128 MB	part-00000-0048ed6f-4415-4c99-87ce-b7258d80330a-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	226 B	May 20 14:59	1	128 MB	part-00000-0131f26b-7305-4f29-8010-4302c7afc63e-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	143 B	May 20 14:48	1	128 MB	part-00000-015ae037-1480-4f4b-b393-26ac7150d93f-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	81 B	May 20 14:57	1	128 MB	part-00000-02055dd0-5b0c-4939-bf25-f29a504ccf14-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	61 B	May 20 14:53	1	128 MB	part-00000-024e80cb-f80b-4e95-b725-d8b117c83b1d-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	107 B	May 20 15:29	1	128 MB	part-00000-02af98d0-97ad-4ae9-b278-64db09b4315a-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	173 B	May 20 15:32	1	128 MB	part-00000-02d7654a-2e81-42b5-a39f-b534293b5c6f-c000.csv	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	hadoop	supergroup	64 B	May 20 15:30	1	128 MB	part-00000-02de9b0a-fb20-4711-832d-feddd20840cc5-c000.csv	<input type="checkbox"/>

Рис.2.8. Содержимое директории /user/hadoop/data внутри HDFS

Конечно, того же результата можно добиться с помощью командной утилиты dfs, но просматривать содержимое HDFS так же, как мы будем просматривать данные об исполнении задач пакетной и потоковой обработки данных значительно ускоряет процесс верификации корректности проведения эксперимента.

Итак, установив hadoop и запустив HDFS у нас появился смысл устанавливать остальные компоненты системы.

### 2.5.3. Apache Spark

Следующим компонентом, установку которого мы рассмотрим будет Apache Spark. Apache Spark включает в себя два инструмента, которые мы будем использовать в ходе экспериментального анализа - это Spark и Spark Structured Streaming. Их внутренне устройство и назначение подробно описаны в первой главе работы.

Для установки Spark мы с помощью утилиты wget скачиваем запакованный архив с официального сайта Apache Spark. Важно при скачивании указать ссылку на Spark No Hadoop, так как в нашем случае Hadoop уже установлен и его будут использовать наравне со Spark и другие компоненты системы. Переменная среды \$SPARK\_HOME имеет значение /home/hadoop/spark-3.5.1-bin-without-hadoop. В файловой системе фреймворк находится на одном уровне с Hadoop.

Так как в качестве языка разработки "работ" Spark было принято решение использовать Python на этом этапе мы устанавливаем на нашу WSL машину Python и добавляем следующие переменные среды, чтобы связать установленный и добавленный в PATH дистрибутив Python версии 3.12 с установленным Spark версии 3.5.1.

Среди продуктов по умолчанию входящих в Spark нам так же понадобятся не все. Все доступные Bash скрипты Spark версии 3.5.1 представлены ниже:

```
hadoop@LAPTOP-AM1MP0U8:~/spark-3.5.1-bin-without-hadoop/sbin$ ls
decommission-slave.sh      start-history-server.sh    start-workers.sh           stop-slaves.sh
decommission-worker.sh     start-master.sh            stop-all.sh               stop-thriftserver.sh
slaves.sh                 start-mesos-dispatcher.sh  stop-connect-server.sh    stop-worker.sh
spark-config.sh           start-mesos-shuffle-service.sh stop-history-server.sh     stop-workers.sh
spark-daemon.sh           start-slave.sh             stop-master.sh            workers.sh
spark-daemons.sh         start-slaves.sh            stop-mesos-dispatcher.sh
start-all.sh             start-thriftserver.sh      stop-mesos-shuffle-service.sh
start-connect-server.sh   start-worker.sh            stop-slave.sh
```

Рис.2.9. Bash скрипты Spark

Из них нам понадобится три скрипта: start-master.sh, start-worker.sh и start-history-server.sh.

Spark Master - центральный управляющий узел, который координирует распределение задач между рабочими узлами (Spark Worker). Он управляет ресурсами кластера и отслеживает состояние выполнения задач. После его инициализации с помощью скрипта start-master.sh нам становится доступен Spark WEB UI, который hostится на порте 8080:

Spark 3.5.1 Spark Master at spark://LAPTOP-AM1MP0U8:7077

URL: spark://LAPTOP-AM1MP0U8:7077

Alive Workers: 0

Cores in use: 0/0 B Total, 0/0 B Used

Memory in use: 0/0 B Total, 0/0 B Used

Resources in use:

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (0)

Worker Id	Address	State	Cores	Memory	Resources
-----------	---------	-------	-------	--------	-----------

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Рис.2.10. Spark master Web UI без активных Worker'ов и приложений

Spark Worker - рабочие узлы, которые выполняют задачи, распределённые Spark Master. Они могут выполнять несколько задач параллельно в зависимости от доступных ресурсов.

При инициализации задания, которое выполняется через командную строку с помощью команды `spark-submit`, аргументами для которой являются в нашей системе адреса файлов с кодом заданий(jobs) Spark, задания явно(в коде задания) привязываются к мастеру.

```

5 | spark = SparkSession.builder \
    |     .master("spark://LAPTOP-AM1MP0U8:7077") \
    |     .appName("SocketStreamToHDFS") \
    |     .config("spark.hadoop.fs.defaultFS", "hdfs://localhost:9000") \
    |     .getOrCreate()

```

Рис.2.11. Инициализация Spark сессии

Выше представлен фрагмент кода, в котором специфицируется адрес Spark Mastera и файловой системы по умолчанию, в нашем случае адрес `hdfs`. Именно такая конфигурация Spark сессии будет актуальной для каждого проводимого в рамках системы эксперимента.

Последний необходимый нам компонент Spark это `history-server` [20]. Spark History Server — это инструмент, предоставляющий веб-интерфейс для просмотра и анализа прошлых и текущих Spark приложений. Он позволяет детально изучать выполнение заданий и этапов, включая планы выполнения, информацию об окружении и детализированные логи. Этот сервер помогает в отладке, настройке производительности и понимании поведения Spark приложений, предоставляя полный обзор их выполнения. В нашем случае максимально полезным будут данные о нагрузке на память и информация о времени выполнения каждого Spark приложения.



Spark приложения направляются на исполнение через терминал WSL добавленной в PATH командой spark-submit:

```
hadoop@LAPTOP-AM1MP0U8:~/spark-3.5.1-bin-without-hadoop/jobs/
batch$ spark-submit test.py
```

Рис.2.12. Запуск Spark работы

Ниже представлен пример информации, которую мы можем получить из Spark WEB UI о выполнении тестовой задачи, которая просто выбирает данные из HDFS и выводит их в консоль.

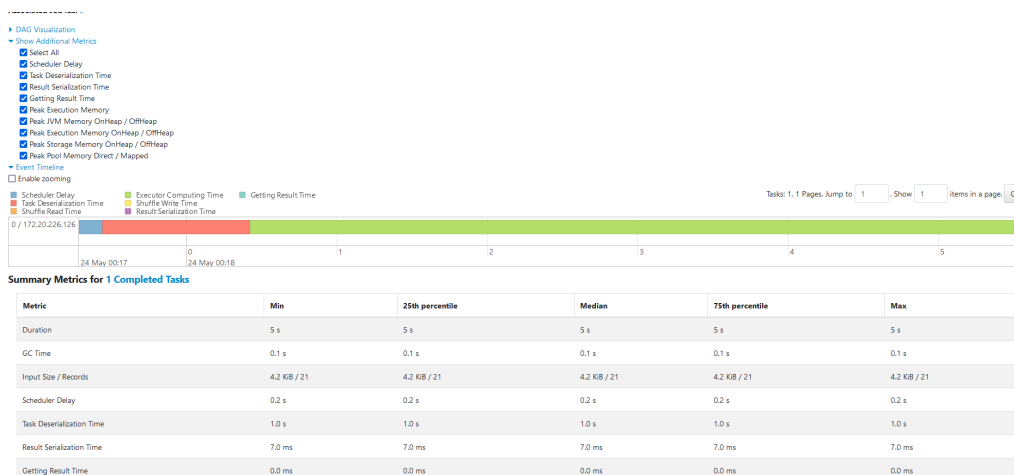


Рис.2.13. Пример метрик выполненной Spark Job

## 2.5.4. Apache Hive

Следующим компонентом, необходимым для функционирования системы является Apache Hive. Установка Apache Hive была произведена аналогично предыдущим инструментам. В системные переменные была добавлена переменная \$HIVE\_HOME, этот же путь был добавлен в PATH:

```
export HIVE_HOME=/home/hadoop/hadoop/apache-hive-4.0.0-bin
export PATH=$PATH:/home/hadoop/hadoop/apache-hive-4.0.0-bin/
bin
```

Рис.2.14. Занесение исполняемых файлов Apache Hive в системную переменную PATH

Работа HIVE в рамках нашей системы завязана на компоненте hiveserver2. Это сервис в Apache Hive, который обеспечивает многопользовательский доступ к данным, хранящимся в HDFS. Он предоставляет интерфейс JDBC и ODBC для подключения к Hive и выполнения SQL-запросов. Основные функции HS2

закljučаются в управлении сессиями и ресурсами, а также хранением логов и информации.

Чтобы выполнять HiveQL запросы к данным из HDFS нужно установить подключение к запущенному hiveserver2. В качестве интерфейса для выполнения запросов мы будем использовать встроенную в дистрибутив Hive командную оболочку beeline. Beeline использует JDBC для подключения к HiveServer2, поддерживает аутентификацию, и позволяет управлять сессиями пользователей. Основное назначение Beeline — обеспечить простой и удобный способ взаимодействия с Hive через командную строку, что упрощает выполнение и отладку запросов. Инициализация beeline с подключением к hiveserver2 выполняется следующей командой:

```
hadoop@LAPTOP - AM1MPOU8: $HIVE_HOME/bin/beeline -u "jdbc:
hive2://localhost:10000"
```

Рис.2.15. Подключение командного интерпретатора beeline к Hiveserver2 по протоколу JDBC

Ниже приведен пример результата запроса к собранным данным, выбирающий последние 10 сообщений. В диалекте HiveQL он имеет структуру:

```
SELECT * FROM messages
LIMIT 10;
```

Рис.2.16. Пример синтаксиса элементарного HiveQL запроса

messages.arrival_timestamp	messages.user	messages.streamer	messages.msg	messages.id	messages.processing_timestamp	messages.latency
1716065107306	sfjsfjs	t2x2	))	126df8cbe9704942758c91f230eb3df	1716065107307	1
1716065107893	asuzhdayuladno	t2x2	ТОХА ДАВАЙ ЗАНИМАТЕЛЬНЫЕ ИСТОРИИ С АЛИСОЙ	54da13f1b6ec6ab5c76c3f684b33ae1c	1716065107895	2
1716065108875	amlr7487	t2x2	дайте денег ТГ @DdAmiron	3c30a8ec7afe64058916c760690f77d3	1716065108877	2
1716065109257	0_01234668	t2x2	тебе монитор не хуже светит	aea9ea51e0ceed5f842664422b8c7aaa	1716065109307	50
1716065109882	i_black_1	t2x2	hehe	680dbbf5ca216fc1a500883261ce5d8c	1716065109884	2
1716065110033	positiv5555	t2x2	Для тебя это копейки	6804e6eb47321416c626e808ec84faa	1716065110044	11
1716065110186	yourpikota	t2x2	До сих пор стримит	ed1e94a863c43b6d8c55deb0fe887cfb	1716065110653	467
1716065110497	laaakeee	t2x2	ох	58dc734f8d1c3b218f3b2ebd0c438fe1	1716065110741	244
1716065111758	rari_exe	t2x2	суфлер	4c005237f2533ed192d7196c85711ae9	1716065111759	1
1716065111933	sobeukwon	t2x2	hehe	26f82dbf85bc8f484d419c1c4058632a	1716065111934	1

Рис.2.17. Результат примера HiveQL запроса

Для получения данных о выполнении конкретного запроса используется Hiveserver2 WebUI, который по-умолчанию hostится на порте 10002, в то время, как сам Hiveserver2 hostится на порте 10000.

Ниже представлен пример информации о ходе выполнения этого же запроса в Hiveserver2 WebUI:

## Query Information: SELECT \* FROM messages LIMIT 10;

<a href="#">Base Profile</a> <a href="#">Stages</a> <a href="#">Query Plan</a> <a href="#">Performance Logging</a> <a href="#">Operation Log</a>	
Compile-time metadata operations	
Call Name	Time (ms)
getTableColumnStatistics_(String, String, List, String)	35
isCompatibleWith_(Configuration)	0
getAllTableConstraints_(AllTableConstraintsRequest)	11
getTable_(GetTableRequest)	21
flushCache_()	0

Рис.2.18. Временные затраты на выполнение каждой из стадий запроса

### 2.5.5. Apache Flink

Последним компонентом, который будет участвовать в сравнении средств работы с большими данными станет Apache Flink. Apache Flink в системе будет исключительно в рамках выполнения задач потоковой обработки данных. Его установка производилась идентично предыдущим фреймворкам. После установки была создана переменная `$FLINK_HOME`, этот же путь был добавлен в `PATH`:

```
export FLINK_HOME=~/.hadoop/flink
export PATH=$PATH:$FLINK_HOME/bin
```

Рис.2.19. Занесение исполняемых файлов Apache Flink в системную переменную `PATH`

Версия Apache Flink, установленного на WSL - 1.15.0. Примечательно, что в отличие от предыдущих фреймворков формат конфигурационного файла `flink-conf.yaml`, у всех остальных же - `xml`. В ходе конфигурации для `flink` в файле `conf/flink-conf.yaml` были установлены следующие отличные от параметров по-умолчанию свойства:

```
taskmanager.numberOfTaskSlots: 4
rest.port: 8085
rest.bind-address: 0.0.0.0
```

Рис.2.20. Конфигурация Apache Flink

`taskmanager.numberOfTaskSlots: 4` - Устанавливает количество слотов задач для каждого `TaskManager`. В данном случае каждый `TaskManager` имеет 4 слота для выполнения задач, что позволяет параллельно выполнять 4 задачи на одном

TaskManager. rest.port: 8085 - Указывает порт, на котором будет запущен REST API сервер Flink. Здесь это порт 8085. rest.bind-address: 0.0.0.0 - Определяет адрес, к которому привязывается REST API сервер. Адрес 0.0.0.0 означает, что сервер будет доступен по всем IP-адресам на машине, где он запущен, а в нашем случае и на "родительской" windows машине.

Чтобы запускать Flink Jobs, для начала нужно запустить Flink Cluster на нашей WSL машине. Это делается с помощью команды

```
hadoop@LAPTOP-AM1MP0U8:~/hadoop/flink/bin$ start-cluster.sh
```

Рис.2.21. Запуск кластера Apache Flink

На этом этапе нам становится доступен Flink Web UI, для которого в конфигурации был задан порт 8085.

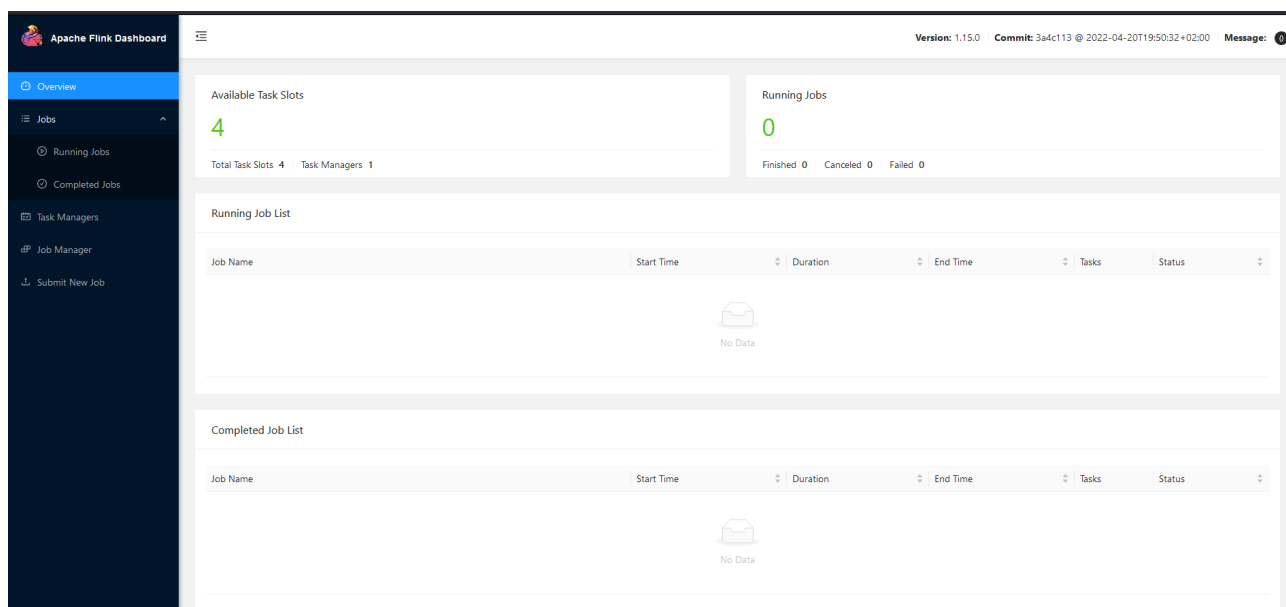


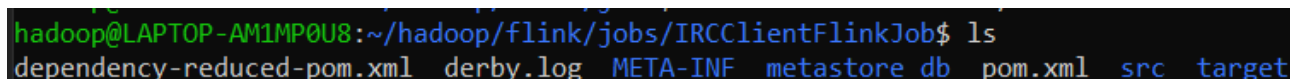
Рис.2.22. Вид Flink Web UI без выполняемых задач

После чего нужно где-либо в файловой системе инициализировать проект Maven. Maven — это инструмент управления проектами и система сборки для Java-проектов. Он упрощает процесс сборки, управления зависимостями и проектами, предоставляет стандартизированную структуру проектов и автоматизирует задачи, такие как компиляция, тестирование и создание пакетов. Maven использует файл конфигурации `pom.xml`, в котором описаны зависимости проекта, плагины и другие настройки. Основные преимущества Maven включают в себя управление зависимостями, стандартизацию процесса сборки и возможность легкой интегра-

ции с системами контроля версий и другими инструментами разработки. Он был установлен вместе с Flink.

Установка зависимостей для проекта Flink, несмотря на кажущуюся тривиальность, оказалась одним из наиболее трудозатратных шагов при конфигурации системы. Так же это стало причиной для установки Flink версии 1.15, так как для версии Apache Flink 1.19 в репозитории Maven отсутствуют необходимые для интеграции с Kafka и многие другие зависимости. Полная структура файла конфигурации `pom.xml` для любой Flink Job (благо она одинаковая для любой Flink Job) будет указана в приложении к отчёту.

Итак, после инициализации структура проекта Maven выглядит следующим образом:

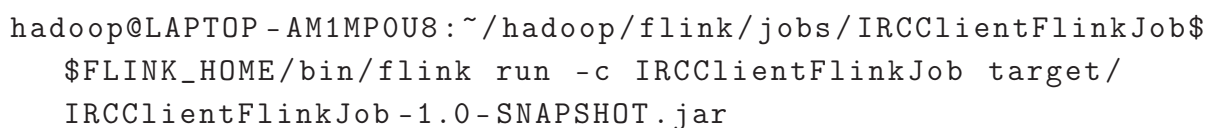


```
hadoop@LAPTOP-AM1MP0U8:~/hadoop/flink/jobs/IRCClietnFlinkJob$ ls
dependency-reduced-pom.xml  derby.log  META-INF  metastore  db  pom.xml  src  target
```

Рис.2.23. Структура проекта Maven

Код для Flink Job помещается в файл с названием проекта в папку `src/main/java/com/example`. После этого проект компилируется в JAR файл. JAR файл — это пакет, содержащий скомпилированный код Java (в виде файлов `.class`), ресурсы (например, изображения и свойства), и метаданные (включая манифест). Этот файл позволяет легко распространять и использовать Java-приложения и библиотеки. В Maven JAR файл включает все необходимые зависимости и сборочные артефакты для запуска или использования проекта в других приложениях.

Скомпилированный JAR файл можно отправлять на исполнение в `flink cluster`:



```
hadoop@LAPTOP-AM1MP0U8:~/hadoop/flink/jobs/IRCClietnFlinkJob$
$FLINK_HOME/bin/flink run -c IRCClietnFlinkJob target/
IRCClietnFlinkJob-1.0-SNAPSHOT.jar
```

Рис.2.24. Запуск Apache Flink работы из командной строки

`IRCClietnFlinkJob` - название класса-точки входа в программу, также специфицируется путь до скомпилированного JAR файла.

В ходе проведения экспериментов в Flink Web UI мы будем пользоваться вкладкой Task Managers, выбирать там Task Manager, управляющий текущей задачей и просматривать данные об исполнении этой задачи.

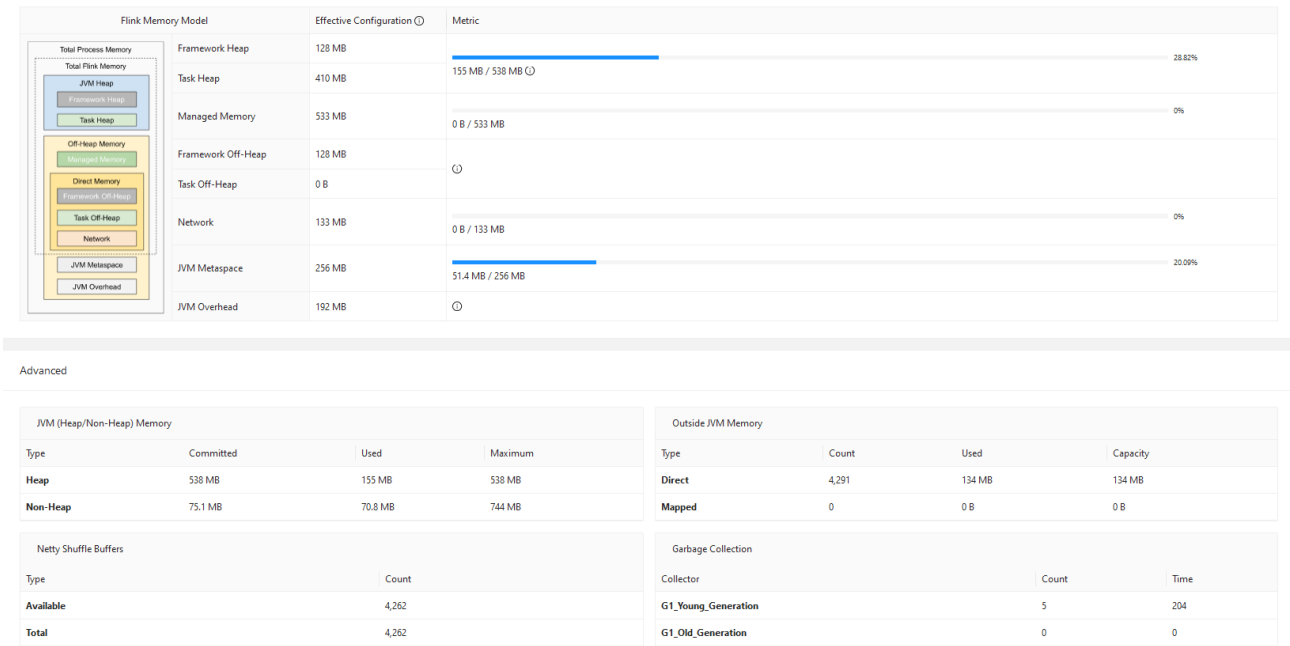


Рис.2.25. Данные о ходе исполнения задачи в Flink Web UI

Последним компонентом системы, необходимым для ее функционирования является Apache Kafka. Он необходим для создания основного источника данных, который будет поддерживать целостность передаваемых на Spark Streaming или Flink приложения. Архив с исполняемыми файлами Kafka был скачан с помощью утилиты wget, версия - 3.7.0.

2.5.6. Kafka

Чтобы запустить кластер kafka, нужно сгенерировать CLUSTER\_ID:

```
| KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

Рис.2.26. Генерация KAFKA\_CLUSTER\_ID

Далее запустить Kafka Storage для хранения логов используя сгенерированный CLUSTER\_ID:

```
| bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
```

Рис.2.27. Сохранение сгенерированного Kafka Cluster ID в настройки сервера

После чего нужно запустить сервер Kafka с заданными параметрами:

```
bin/kafka-server-start.sh config/kraft/server.properties
```

Рис.2.28. Запуск сервера с хранимыми настройками

На сервере, который потребляет сообщения IRC сокетов с чатов Twitch программно создаются Kafka Producer и Kafka Topic. Для тестирования и демонстрации корректности установки и конфигурации Kafka был создан консольный Kafka Consumer используя поставляемый в рамках Kafka shell скрипт:

```
bin/kafka-console-consumer.sh --topic irc_messages --from-  
beginning --bootstrap-server localhost:9092
```

Рис.2.29. Запуск Console Consumera Kafka, прослушивающего Topic с сообщениями из IRC сокетов

После флага `--topic` задается название Kafka Topic с сообщениями(его название определяется создаваемым на сервере Producerом), а также адрес kafka bootstrap сервера, который мы запустили ранее.

Результат работы Kafka Console Consumer можно увидеть на скриншоте ниже:

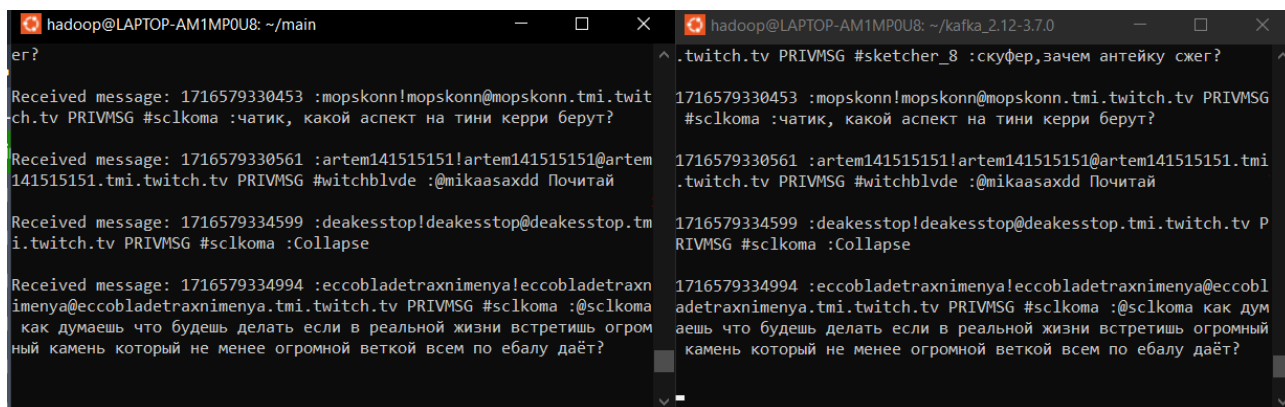


Рис.2.30. Логи полученных с IRC сообщений(слева) и Kafka console consumer

## 2.6. Разработка сервера-слушателя IRC сокетов

Итак, все компоненты системы, являющиеся средствами работами с большими данными, которые мы собираемся сравнивать установлены. Теперь перед нами стоит задача создания источника данных, нагрузка в котором, то есть количество сообщений, может изменяться. В рамках экспериментального анализа это позволит нам тестировать одни и те же задачи при разной нагрузке. Добавив к этому действительность природы наших данных мы получим твердую основу



для проведения экспериментов, так как почти в любой системе данные играют определяющую роль.

Ядром сервера является класс `IRCCClient`. Он имеет следующий конструктор:

```

def __init__(self, server, port, nickname, token, kafka_server
    , kafka_topic):
    self.server = server
    self.port = port
    self.nickname = nickname
    self.token = token
    self.channels = []
    self.kafka_server = kafka_server
    self.kafka_topic = kafka_topic
    self.command_queue = Queue()
    self.producer = KafkaProducer(bootstrap_servers=[
        kafka_server])
    self.stop_event = threading.Event()
    self.threads = []

    threading.Thread(target=self.process_commands, daemon=
        True).start()

```

Рис.2.31. Конструктор класса `IRCSocketServer`

В нем создаются свойства класса для подключения к IRC сокетам, а также инициализируется `Kafka Producer`. Кроме того, создается событие `stop_event`, которое сыграет роль при необходимости очистки списка каналов, с которых будут считываться сообщения из сокета.

Также уже в конструкторе в отдельном потоке запускается метод `process_commands`, который имеет следующую структуру.

```

def process_commands(self):
    while True:
        command, args = self.command_queue.get()
        if command == "add_channel":
            self._add_channel(args)
        elif command == "clear_channels":
            self._clear_channels()
        self.command_queue.task_done()

```

Рис.2.32. Функция, асинхронно выполняющая команды, пришедшие по запросу к API

По структуре этого метода можно понять, что в рамках нашего класса доступны 2 команды - `add_channel` и `clear_channels`. На самом деле этого вполне достаточно для проведения эксперимента. С помощью метода `add_channel` будут добавляться каналы, для которых открываются IRC слушатели, а с помощью метода `clear_channels` эти слушатели будут удаляться, таким образом будет регулироваться нагрузка на систему во время эксперимента.

IRC слушатели создаются в методе `irc_connection`, который реализован следующим образом:

```

def irc_connection(self, channel):
    irc_sock = socket.socket()
    irc_sock.connect((self.server, self.port))
    5   irc_sock.send(f"PASS {self.token}\n".encode('utf-8'))
    irc_sock.send(f"NICK {self.nickname}\n".encode('utf-8'))
    irc_sock.send(f"JOIN {channel}\n".encode('utf-8'))
    print(f"Connected to IRC channel {channel}")

    10   while not self.stop_event.is_set():
        try:
            resp = irc_sock.recv(2048).decode('utf-8')
            if resp.startswith('PING'):
                irc_sock.send("PONG\n".encode('utf-8'))
            15   elif "PRIVMSG" in resp:
                arrival_timestamp = str(int(float(time.time()) * 1000))
                formatted_resp = f"{arrival_timestamp} {resp}"
                print(f"Received message: {formatted_resp}")
                self.producer.send(self.kafka_topic,
                                   formatted_resp.encode('utf-8'))
            20   except socket.error:
                break

    irc_sock.close()
    print(f"Disconnected from IRC channel {channel}")

```

Рис.2.33. Метод, слушающий IRC сокет

Итак, создается объект `socket`, отправляются все необходимые для подключения сообщения. Кодировка сообщений `utf-8`. После чего пока `self.stop_event` не

считываются приходящие не сокет сообщения. На сокет могут приходить сообщения двух типов "PING" и "PRIVMSG". "PING" сообщения присылаются сервером, чтобы определить, "жив" ли клиент, и имеет ли смысл поддерживать с ним подключение. Поэтому, чтобы сервер не разрывал созданное подключение в ответ на PING, согласно реальному поведению клиента во время подключения через браузер мы отправляем PONG. Если же сообщение имеет подстроку "PRIVMSG" то мы создаем `arrival_timestamp` с точностью до миллисекунд и перенаправляем сообщение в Kafka Producer.

Важно понимать, как именно создается подключение, то есть как именно вызывается функция `irc_connection`.

```

5 | def _add_channel(self, channel):
    |     self.channels.append(channel)
    |     thread = threading.Thread(target=self.irc_connection,
    |                             args=(channel,))
    |     thread.daemon = True
    |     thread.start()
    |     self.threads.append(thread)
    |     print(f"Added channel {channel}")

```

Рис.2.34. Метод, запускающий функцию прослушивания заданного IRC сокета в отдельном потоке

Она вызывается отдельным потоком в скрытом методе `_add_channel`. Аргументом для нее становится переданный через REST API endpoint идентификатор канала.

Вторым методом в REST API является метод, очищающий каналы, так как именно добавленными каналами будет определяться нагрузка на систему во время эксперимента поткоовой обработки данных.

```

5 | def _clear_channels(self):
    |     self.stop_event.set()
    |     for thread in self.threads:
    |         thread.join()
    |     self.threads = []
    |     self.channels = []
    |     self.stop_event.clear()
    |     print("Cleared all channels and stopped threads")

```

Рис.2.35. Функция, разрывающая все активные в данный момент сокеты

При вызове этого метода `stop_event`, инициализированный в конструкторе выставляется на `true`, в связи с чем все запущенные в других потоках экземпляры метода `irc_connection` обрываются, после чего потоки объединяются и исполнение возвращается в основной поток.

REST API является всего лишь инструментом для управления системой, его код максимально тривиален, поэтому рассматривать его смысла здесь нет, упомянем лишь, что структура метода состоит из вызова метода класса и формирования ответа на запрос. API реализован на фреймворке Flask.

Мы обозрели принцип работы сервера, сейчас покажем действия, которые будут осуществляться с нашим сервером тестирующим скриптом при проведении эксперимента.

Запустим сервер, учитывая, что для запуска сервера должен быть запущен Kafka cluster, иначе возникнет ошибка при инициализации экземпляра класса `IRCCClient` при попытке подключиться к bootstrap серверу Kafka на порте 9092:

```
|hadoop@LAPTOP-AM1MP0U8:~/main$ python3 socket_manager.py
```

Рис.2.36. Запуск сервера-слушателя IRC сокетов

После чего зайдём на Twitch.tv и выберем несколько каналов, которые в данный момент ведут прямые трансляции. Добавим их для отслеживания чата, с помощью POST запросов к серверу с указанием id канала в body:

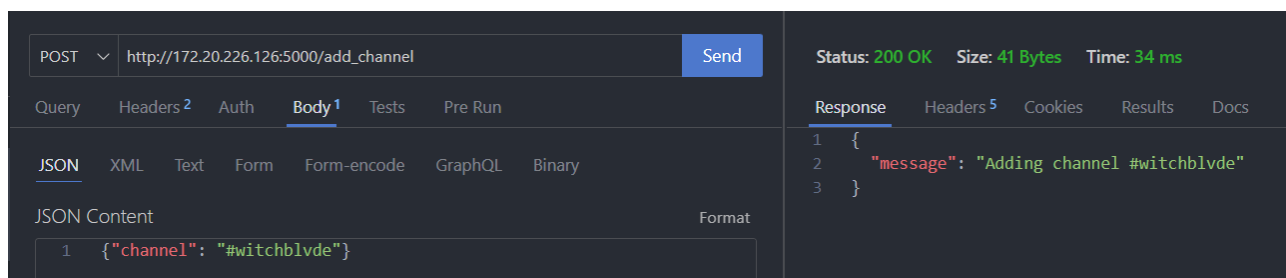


Рис.2.37. Запрос, осуществляющий добавление канала для отслеживания чата

После этого в логах сервера начинают появляться сообщения, который в данный момент отправляются в чат-комнату прямой трансляции.

Добавим еще один канал, после чего сравним содержимое логов сервера и чат-комнат отслеживаемых трансляций:

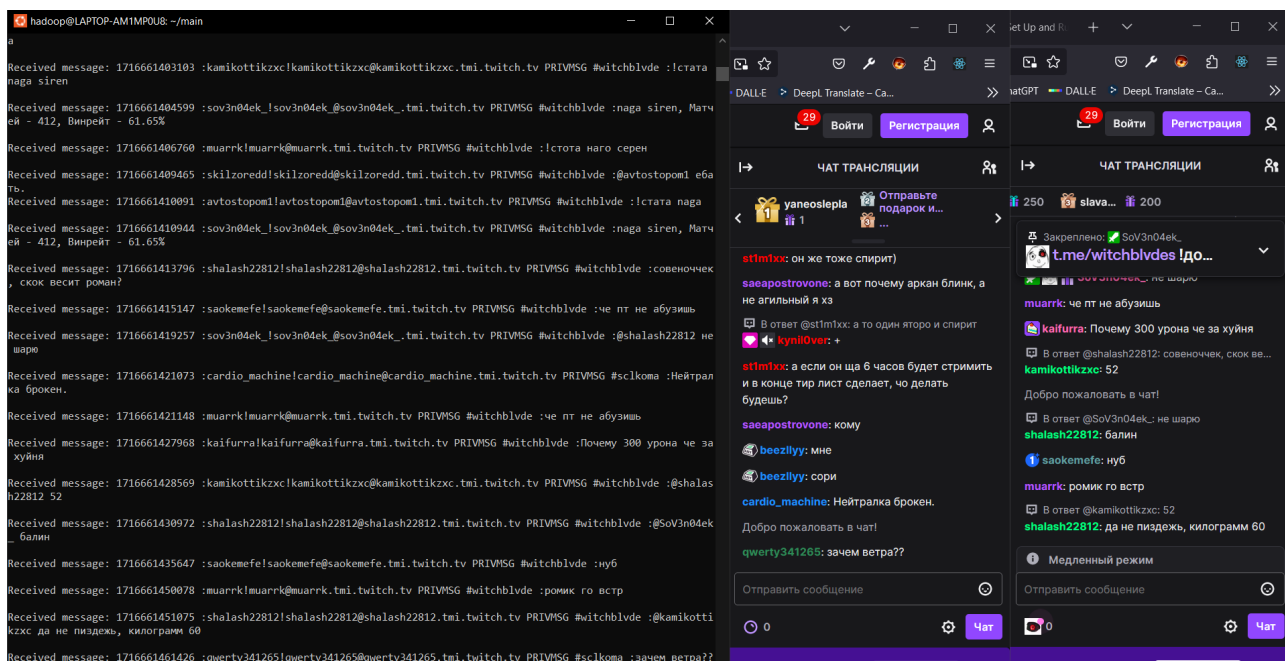


Рис.2.38. Логи сервера и чат комнаты

Как можно видеть, последние логи сервера и последние сообщения в чатах совпадают, причем в логах присутствуют сообщения из обеих чат-комнат. Очистим список каналов соответствующим запросом:

```
172.20.224.1 - - [25/May/2024 21:34:18] "POST /clear_channels HTTP/1.1" 200 -
Received message: 1716662058848 :everyonewantshelpme!everyonewantshelpme@everyonewantshelpme.tmi.twitch.tv PRIVMSG #witchblvde :Уже взял

Disconnected from IRC channel #witchblvde
Received message: 1716662069147 :raydjer9!raydjer9@raydjer9.tmi.twitch.tv PRIVMSG #sclcoma :Дорого прикольное аниме

Disconnected from IRC channel #sclcoma
Cleared all channels and stopped threads
```

Рис.2.39. Логи сервера после очистки списка отслеживаемых каналов

## 2.7. Автоматизация проведения экспериментального сравнения средств потоковой обработки данных

Задумавшись о проведении экспериментов, я понял, что имеет смысл попытаться унифицировать этот процесс. Под экспериментом далее понимается выполнение прикладных задач с описанными выше данными чатов Twitch с параллельным сбором информации о выполнении задач.

*Перечислим порядок проведения эксперимента в рамках потоковой обработки данных:*

1. Проверка работы необходимых для выполнения задачи сервисов(Flink Cluster, Flink Web UI, Spark WEB UI, Spark Master...)

2. Запуск задач на параллельное исполнение в течении времени, заданном в конфигурации эксперимента
3. Запуск средств для отслеживания показателей выполнения задач(задержка, память, входящая нагрузка)
4. По завершении времени проведения эксперимента остановка задач и проверка корректности их выполнения
5. Ознакомление с результатами эксперимента и занесение их в зачет

Все эти шаги звучат так, что их можно нетрудно автоматизировать. Это мы и попытаемся сделать далее в работе.

Так как у всех экспериментов в рамках потоковой обработки данных много общего было решено создать один скрипт, и передавать нюансы эксперимента в файле config.json.

Скрипт будет запускаться через cmd и в программу будет передаваться аргумент - путь к директории эксперимента. Помимо файла конфигурации эксперимента в этой директории самим скриптом будут создаваться папка для каждого запуска эксперимента, в них же будут сохраняться логи и результаты выполнения эксперимента.

Парсинг пути к директории эксперимента их командной строки происходит при помощи библиотеки argparse:

```

5 | parser = argparse.ArgumentParser(
    |     description='Experiment directory')
    | parser.add_argument('experiment_directory_path', type=str,
    |                     help='Path to the experiment directory')
    | path = parser.parse_args().experiment_directory_path
    |
    | config = load_config(f"{path}/config.json")

```

Рис.2.40. Парсинг аргументов из командной строки при запуске скрипта, автоматизирующего проведения потокового эксперимента

С помощью переданного в аргументах запуска пути будет парситься конфигурация эксперимента из файла config.json.

Так осуществляется запуск эксперимента из командной строки:

```
(bigdata) C:\Users\79270\Desktop\ВУЗ\Четвертый курс\Диплом\
testing_app>python stream_experiment.py "experiment_configs
/stream/Data formatting and writing to HDFS"
```

Рис.2.41. Команда для запуска потокового эксперимента с заданной по указанному пути конфигурацией

Сразу после запуска и получения конфигурации мы инициализируем папку для текущей попытки проведения эксперимента, ее название будет соответствовать текущим времени и дате. Также в эту папку будет скопирована текущая конфигурация эксперимента, и в этой папке будет создан файл, куда будут дублироваться все логи, которые выводятся в консоль в течение эксперимента. В эту же папку будут сохранены результаты эксперимента:

```
current_time = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')

new_directory_path = os.path.join(os.path.dirname(path),
    current_time)
5 os.makedirs(new_directory_path, exist_ok=True)

old_config_path = os.path.join(new_directory_path, 'config_old
    .json')

save_config(config, old_config_path)
10 config_logger(new_directory_path)
```

Рис.2.42. Инициализация логгера и создание папки, в которой будет храниться информация о проведенном эксперименте

Вид папки для текущей попытки эксперимента сразу после создания:

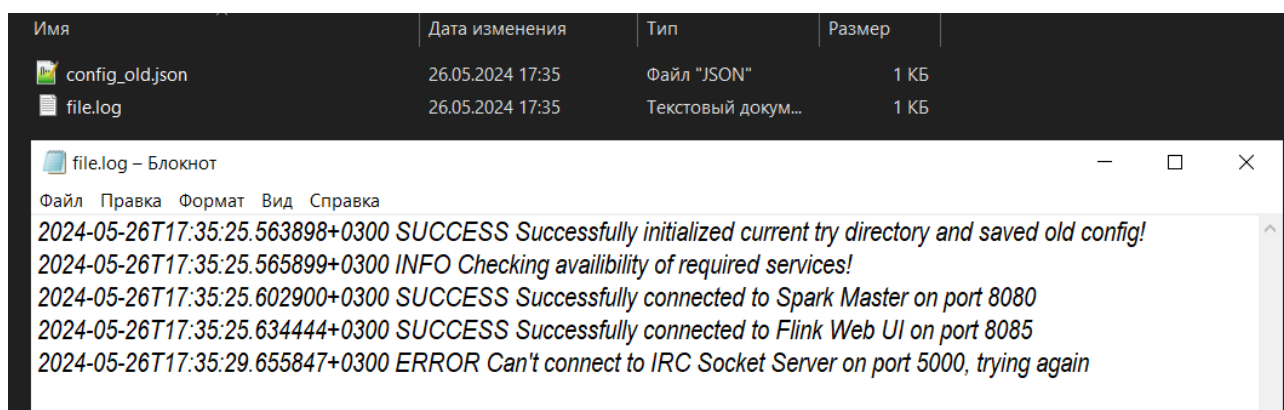


Рис.2.43. Папка для текущей попытки эксперимента сразу после создания



Сразу после завершения создания папки и копирования файла конфигурации проверяется доступность необходимых сервисов. Так как для любого эксперимента относящегося к потоковой обработке данных список необходимых сервисов одинаковый, его можно инициализировать прямо в программе.

```
REQUIRED_SERVICES = [
    {"title": "Spark Master", "port": 8080},
    {"title": "Flink Web UI", "port": 8085},
5    {"title": "IRC Socket Server", "port": 5000},
]
```

Рис.2.44. Необходимые для потокового эксперимента сервисы, работоспособность которых будет проверяться

Далее выполняется последовательное подключение к указанным портам. В зависимости от ответа на запрос, делается вывод о доступности инструмента в данный момент. Производится 10 попыток подключения к каждому сервису:

```
check_connections(REQUIRED_SERVICES)

def check_connections(services):
5    logger.info("Checking availability of required services!")
    for service in services:
        try_connect(10, 10, is_http_service_running, service)
```

В случае, если один из сервисов недоступен, логи выглядят следующим образом:

```
(bigdata) C:\Users\79270\Desktop\ВУЗ\Четвертый курс\Диплом\testing_app>python stream_experiment.py "experiment_configs/stream/Data formatting and writing to HDFS"
2024-05-26 17:51:08.868 | SUCCESS | __main__:<module>:17 - Successfully got configuration. Initializing current experiment directory!
2024-05-26 17:51:08.876 | SUCCESS | __main__:<module>:34 - Successfully initialized current try directory and saved old config!
2024-05-26 17:51:08.879 | INFO | connections_checker:check_connections:6 - Checking availability of required services!
2024-05-26 17:51:08.930 | SUCCESS | utils:submit_func:9 - Successfully connected to Spark Master on port 8080
2024-05-26 17:51:08.970 | SUCCESS | utils:submit_func:9 - Successfully connected to Flink Web UI on port 8085
2024-05-26 17:51:13.002 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:17.026 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:22.049 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:28.066 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:35.093 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:43.118 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:51:52.140 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:52:02.156 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:52:13.171 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:52:25.198 | ERROR | utils:unsuccess_func:19 - Can't connect to IRC Socket Server on port 5000, trying again
2024-05-26 17:52:34.200 | CRITICAL | utils:reject_func:14 - Unable to connect to IRC Socket Server on port 5000, make sure the service is running
```

Рис.2.45. Логи скрипта в случае недоступности одного из сервисов

После проверки доступности сервисов, скрипт запускает Flink или Spark Streaming работу. Это весьма нетривиальная задача, так как задачу нужно запустить в wsl, а скрипт работает внутри windows. Причина, по которой нельзя поместить скрипт также в среду wsl - мы должны будем получить результаты эксперимента,

такие как логи, изображения построенных графиков и т.д., было бы очень неудобно после выполнения каждого эксперимента передавать файлы с wsl на windows, конечно, существует множество способов для этого, тем не менее, выбранная конфигурация кажется нам наиболее удобной.

Flink или Spark работа запускается с помощью утилиты wsl, которая поддерживает выполнение bash команд. В рамках bash команд мы можем использовать переменные среды и так как в ходе разработки системы мы задали переменную \$FLINK\_HOME, мы используем ее:

```

def run_flink_job():
    try:
        result = subprocess.run(
5           ['wsl', '-u', 'hadoop', 'bash', '-c', 'source ~/.
              profile && \${FLINK_HOME}/bin/flink run -c
              IRCCClientFlinkJob \${FLINK_HOME}/jobs/
              IRCCClientFlinkJob/target/IRCCClientFlinkJob-1.0-
              SNAPSHOT.jar'], capture_output=True, text=True)
        if result.returncode == 0:
            print(result.stdout)
            return result.stdout
        else:
10         raise Exception(result.stderr)
    except Exception as e:
        print(f"Error running command:\nException: {e}")
        return None

```

Рис.2.46. Функция, осуществляющая запуск Flink работы

Важно отметить, что задача запускается в отдельном потоке, так как она бы заблокировала дальнейшее исполнение, если бы запускалась синхронно, ввиду того, что процесс терминала бы не закрылся сам по себе даже после отправки работы в исполнение.

После запуска задачи мы запускаем методы, которые обращаются к REST API сервера, взаимодействующего с IRC сокетами, разработку которого мы рассмотрели в предыдущей главе в последнем разделе. Сначала мы очищаем список стримов, на случай, если они остались после предыдущего эксперимента:

```
| channel_management.clear_channels()
```

Рис.2.47. Вызов метода класса ChannelManager, отвечающего за очистку каналов после эксперимента

После чего добавляем всех указанных в конфигурации эксперимента стримеров:

```
for streamer in config['streamers']:
    channel_management.add_channel(streamer)
```

Рис.2.48. Вызов метода класса ChannelManager, отвечающего за добавление каналов, выбранных для использования в эксперименте

Дальше начинается процесс получения экспериментальных данных от Flink или Spark Streaming задачи.

В Kafka Topic, Producer для которого создается в задаче помещаются данные о задержке между сообщениями. Consumer для этого Topic'a открывается с скрипте, а именно в экземпляре класса LatencyAnalyzer:

```
latency_analyzer = LatencyAnalyzer(
    kafka_bootstrap_servers=kafka_bootstrap_servers,
    kafka_topic=latency_topic, output_path=
    new_directory_path)
```

Рис.2.49. Инициализация класса, собирающего данные о задержке в ходе выполнения задач потоковой обработки данных

Этот класс имеет два метода start и stop. При вызове метода start, класс подключается к Kafka Topicу и начинает коллекционировать значения из него. После вызова метода stop накопленные за время эксперимента значения анализируются, вычисляются минимальная, максимальная, средняя задержки. Также вычисляется среднеквадратичное отклонение для выборки задержек и строится график задержек для каждого 100-го сообщения(невозможно построить график для каждого из сообщений, их слишком много).

Похожим образом устроен второй класс - PayloadDebugger. Он подключается к тому же Kafka Topicу, к которому подключен Flink или Spark Streaming Job и замеряет количество входящих сообщений. Это нужно, чтобы формализовать входящую нагрузку при подведении итогов эксперимента, так как в конфигурации эксперимента задаются лишь ники стримеров, а уже по итогам эксперимента мы получим график прихода сообщений.

После завершения времени эксперимента, заданного в конфигурации в секундах потоки завершаются, и данные о выполнении эксперимента заносятся в папку текущей попытки.

Итак, давайте протестируем наш скрипт, т.е. проведем эксперимент со следующей конфигурацией:

```

5 | {
   |   "experiment_duration": 100,
   |   "streamers": [
   |     "kaicenat"
   |   ],
   |   "flink_job_name": "IRCClientFlinkJob"
   | }

```

Рис.2.50. Пример конфигурации файла потоковой обработки данных

Длительность эксперимента - 100 секунд, стример - "kaicenat". Этот ник не обязан ни о чем говорить, упомянем лишь, что на момент проведения эксперимента прямую трансляцию смотрело 57 тыс. человек, так что в теории нагрузка на систему в ходе эксперимента должна быть довольно сильной.

Запустим эксперимент из командной строки, указав путь к папке, в которой лежит config.json.

В результате полученные следующие данные о работе задачи Apache Flink, которая принимает сообщения в сыром виде, переводит их в записи формата csv, с полями содержащими временную метку прибытия, никнейм пользователя, канал, содержание сообщения и временную метку завершения обработки:

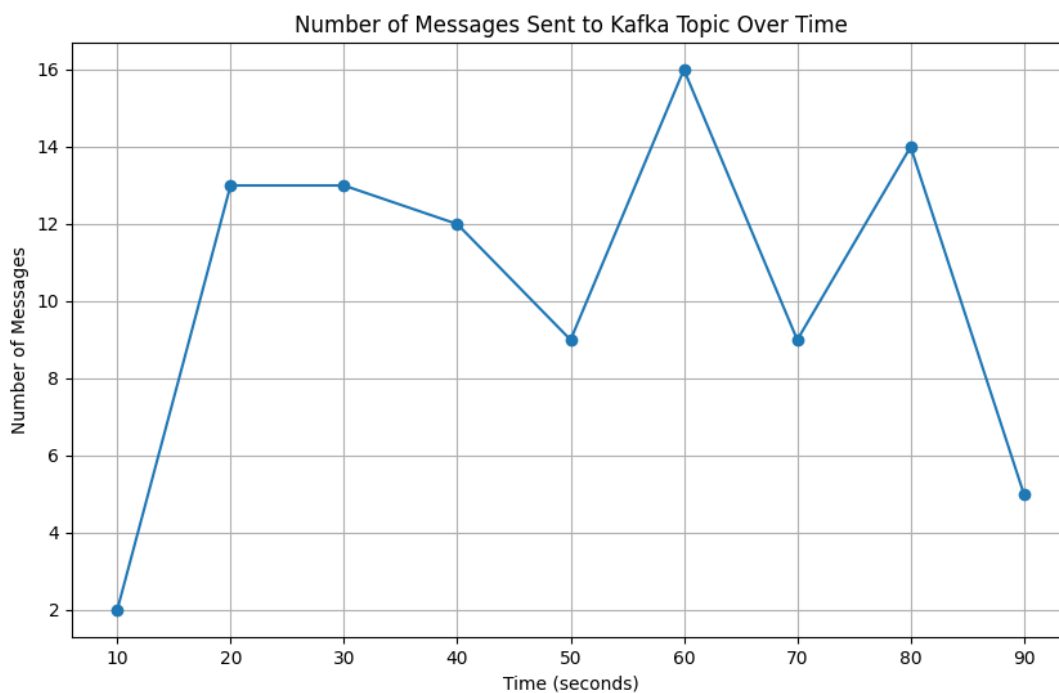


Рис.2.51. Количество сообщений приходящих из IRC сокета чата(данные измерялись каждые 10 секунд)

```

Max messages: 16
Min messages: 2
Mean messages: 10.333333333333334
5 Standard Deviation: 4.268749491621899

```

Рис.2.52. Результат работы сборщика данных о нагрузке в ходе тестового эксперимента

Получается в систему на протяжении 100 секунд в среднем приходило примерно 10 сообщений. Посмотрим, какова была задержка между прибытием сообщения и его сохранением в распределенное хранилище данных:

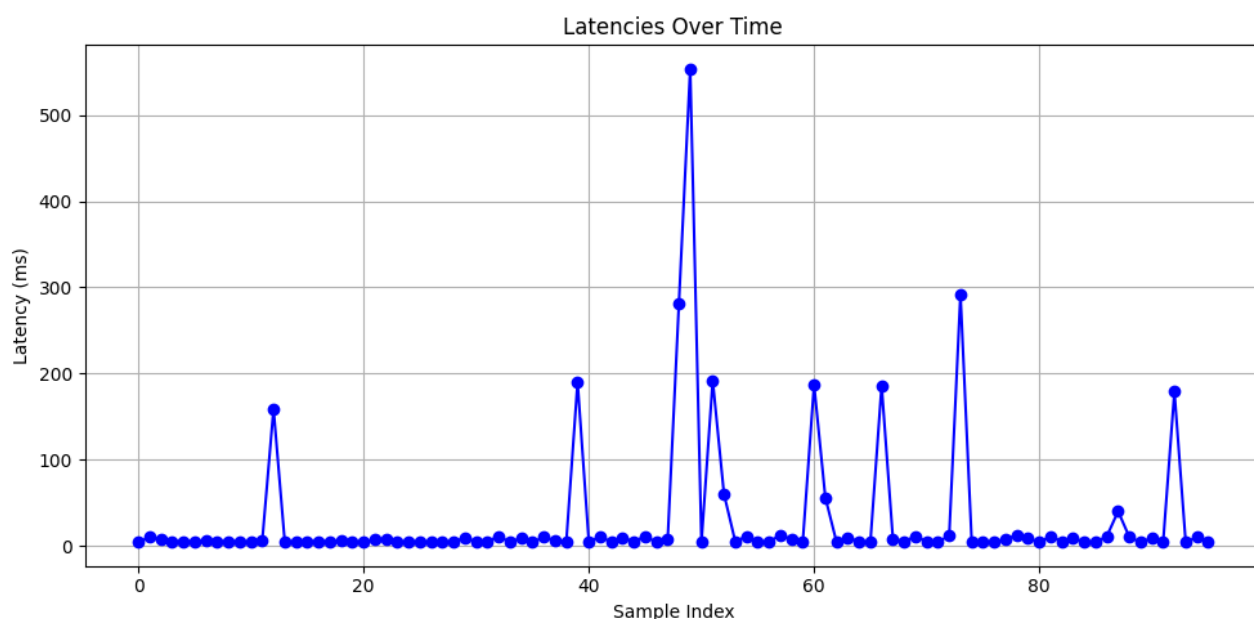


Рис.2.53. Средняя за каждую секунду эксперимента задержка между прибытием сообщения из сокета и его записью в HDFS

```

Min Latency: 4
Max Latency: 554
Mean Latency: 30.239583333333332
5 Standard Deviation: 78.79003543062868

```

Рис.2.54. Результат работы сборщика информации о задержке в ходе тестового эксперимента

Средняя задержка между получением сообщения из сокета и его записью в HDFS составила 30 миллисекунд.

По графикам можно заметить, что момент наибольшей задержки, в действительности, совпадает с моментом наибольшей нагрузки на систему.

Итак, нами была разработана система, которая полностью соответствует выдвинутым требованиям. Мы можем модифицировать входящую нагрузку с помощью разработанного IRC Socket сервера и его API, у нас создается источник данных, к которому могут получать доступ несколько процессов(Kafka Topic с названием irc\_messages). Также мы можем собирать данные об исполнении различных задач потоковой и пакетной обработки данных с помощью Web клиентов фреймворков для работы с большими данными. И наконец, мы можем выполнять задачи пакетной и потоковой обработки данных на выбранных для сравнения фреймворках: для потоковой обработки - Spark Streaming и Apache Flink, а для пакетной - Apache Spark, Apache Hive и Hadoop Map Reduce. Все рассмотренные фреймворки, их Web интерфейсы, а также брокер сообщений Kafka и сервер для

отслеживания IRC советов станут необходимыми инструментами для проведения экспериментального сравнения в следующей главе.



## ГЛАВА 3. ЭКСПЕРИМЕНТАЛЬНОЕ СРАВНЕНИЕ ФРЕЙМВОРКОВ ДЛЯ ПОТОКОВОЙ ОБРАБОТКИ ДАННЫХ: APACHE FLINK И SPARK STREAMING

В задачах потоковой обработки данных наиболее важная характеристика системы - задержка. То есть количество времени, которое проходит между появлением сообщения в системе и завершением его обработки. В рамках потоковой обработки больших данных, количество таких сообщений может достигать сотен в секунду и именно задержка будет определять, как скоро другие компоненты кластера или приложения, работающие с вашим ПО получают доступ к информации, которую вы получаете извне и в измененном виде предоставляете им.

В предыдущей главе нами был разработан скрипт для проведения автоматизированного сравнения выполнения задач потоковой обработки данных двумя фреймворками - Apache Flink и Spark Streaming.

Под экспериментом здесь и далее понимается совокупность запусков скрипта и получением данных его выполнения с различной входной нагрузкой.

До начала экспериментов была определена аппаратная конфигурация фреймворков, согласно которой обоим фреймворкам выделялось одинаковое количество оперативной памяти(4 Гб) и одинаковое количество ядер процессора(2 ядра).

### 3.1. Задача чтения данных из Kafka Producera, обработки и записи в HDFS

Сначала рассмотрим способ реализации задачи на двух использованных технологиях.

#### 3.1.1. Реализация на Apache Flink

Полный код реализации доступен в приложении 9.

В начале файла инициализируется среда выполнения Flink:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.  
getExecutionEnvironment();
```

Рис.3.1. Инициализация среды выполнения Flink

После чего определяются настройки для Kafka, адрес сервера и название Темы для приходящих сообщений и отправки задержки:

```
String kafkaBootstrapServers = "localhost:9092";
String kafkaTopic = "irc_messages";
String latencyTopic = "flink_latency";
```

Рис.3.2. Данные для подключения к Kafka Topicам

Далее создается тема для отправки данных о задержке, если она еще не создана:

```
createKafkaTopic(kafkaBootstrapServers, latencyTopic);
```

Рис.3.3. Вызов функции, инициализирующей Kafka Producer, отправляющий данные о задержке во время выполнения Flink работы

Основной функционал, задержку в работе которого мы и будем тестировать, лежит в следующем участке кода:

```

DataStream<String> text = env.addSource(kafkaConsumer);

DataStream<Message> messages = text.map(new MapFunction<String
    , Message>() {
5      @Override
      public Message map(String value) {
          try {
              String[] parts = value.split(" ");
              if (parts.length < 5) {
10                  throw new IllegalArgumentException("Unexpected
                      message format: " + value);
              }

              long arrivalTimestamp = Long.parseLong(parts[0]);
              String user = parts[1].split("!")[0].substring(1);
15              String streamer = parts[3].substring(1);
              String msg = value.split(":", 3)[2];

              long processingTimestamp = System.
                  currentTimeMillis();

20              Message message = new Message(arrivalTimestamp,
                  user, streamer, msg, processingTimestamp);
              System.out.println("Processed message: " + message
                  );
              return message;
          } catch (Exception e) {
25              e.printStackTrace();
              return null;
          }
      }
    }).filter(message -> message != null);

```

Рис.3.4. Фрагмент кода, реализующий десериализацию сообщений, приходящих из IRCsocket сервера

Приходящее текстовое сообщение разбивается объектом DataStream на части и приводится к csv формату следующего содержания: <временная метка прибытия>, <ник отправителя сообщения>, <название канала>, <текст сообщения>, <временная метка завершения обработки сообщения>. Временная метка завершения обработки создается сразу после завершения разбиения сообщения единицы информации.

Задержка в данном случае является разностью между временной меткой прибытия и временной меткой завершения обработки сообщения.

Структурированные данные сохраняются в HDFS:

```
messages.addSink(new HDFSSink());
```

Рис.3.5. Создание HDFSSink для записи десериализованных данных в HDFS

А задержка отправляется в Kafka Topic, инициализированный в начале программы:

```
messages.map(new MapFunction<Message, String>() {
    @Override
    public String map(Message message) {
5        long latency = message.getProcessingTimestamp() -
            message.getArrivalTimestamp();
        return Long.toString(latency);
    }
}).addSink(kafkaProducer);
```

Рис.3.6. Последовательное применения функции, подсчитывающей задержку для каждого фрагмента данных

### 3.1.2. Реализация на Spark Streaming

Теперь рассмотрим реализацию той же задачи на Spark Streaming.

Полный код реализации доступен в приложении 11.

В начале программы инициализируется Spark сессия

```
spark = SparkSession.builder \
    .appName("KafkaToHDFS") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql
        -kafka-0-10_2.12:3.5.1") \
5    .master("spark://LAPTOP-AM1MPOU8:7077") \
    .config("spark.hadoop.fs.defaultFS", "hdfs://localhost
        :9000") \
    .getOrCreate()
```

Рис.3.7. Инициализация Spark сессии

Конфигурируются внешние пакеты для использования Kafka, добавляется Master процесс, в рамках которого будет контролироваться исполнения задачи, а также конфигурируется файловая система по умолчанию, в нашем случае HDFS.

Далее создается объект для чтения данных с Kafka topic, такого же как в Flink:

```

lines = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", kafka_bootstrap_servers
5      ) \
    .option("subscribe", kafka_topic) \
    .option("startingOffsets", "latest") \
    .load()

```

Рис.3.8. Инициализация объекта, выполняющего чтение сообщений из Kafka Topic

Настройка `startingOffsets=latest` дает объекту, осуществляющему чтение данных понять, что стоит начать обрабатывать данные, прибывающие в Kafka топик с момента запуска Spark приложения, а данные прибывающие раньше - отбрасывать, так как для них будет невозможно измерить задержку, и хотя они и хранятся в Kafka Topic, мы их не обрабатываем.

Далее производится разбиение строки на данные и преобразование их в нужный нам формат с последующим добавлением временной метки окончания обработки:

```

messages = lines.selectExpr("CAST(value AS STRING) as value")
\
  .select(split(col("value"), " ", 5).alias("parts")) \
  .filter(expr("size(parts) > 3")) \
5   .selectExpr(
      "parts[0] as arrival_timestamp",
      "split(parts[1], '!')[0] as raw_username",
      "parts[3] as channel",
      "parts[4] as message"
10  ) \
    .selectExpr(
      "CAST(arrival_timestamp AS LONG) as arrival_timestamp"
      ,
      "substring(raw_username, 2, length(raw_username) - 1)
      as user",
      "substring(channel, 2, length(channel) - 1) as
      streamer",
15  "substring(message, 2, length(message) - 1) as msg"
    ) \
    .withColumn("processing_timestamp", (unix_millis(
      current_timestamp()))).cast("long"))

```

Рис.3.9. Фрагмент кода, выполняющий преобразование приходящих из Kafka Topic сообщений в структурированный формат с помощью Spark DataFrame API

Преобразованные данные, хранящиеся в формате DataFrame(модифицированном для Spark Streaming) передаются в записывающий поток, который осуществляет перенаправление данных о задержке в соответствующий Kafka Topic, а также сохраняет данные в HDFS.

```

query = messages.writeStream \
  .outputMode("append") \
  .format("csv") \
5  .option("path", "/user/hadoop/data") \
  .option("checkpointLocation", "/user/hadoop/checkpoints")
  \
  .foreachBatch(send_to_kafka) \
  .start()

```

Рис.3.10. Инициализация объекта, выполняющего запись в HDFS, а также отправляющего данные о задержке в Kafka Topic

Итак, мы рассмотрели реализацию задач, убедились, что они делают одно и то же, работают с одними и теми же источниками данных и помещают данные в

одно и то же место, соответственно задержка, которую мы будем получать может быть сравнена и разница в этой задержке будет отображать именно разницу в работе фреймворков.

Эксперимент будет проходить в три стадии, на каждой стадии будет отличаться входная нагрузка на систему.

### ***3.1.3. Низкая нагрузка***

Все фазы эксперимента будут длиться 100 секунд. С конфигурацией фаз экспериментов потоковой обработки данных можно ознакомиться в приложении 12.

В этой фазе эксперимента сообщения будут поступать с одного стрима, который, на момент проведения эксперимента смотрело 53 тысячи человек.

Запускаем эксперимент из командной строки:

```
python stream_experiment.py "experiment_configs\stream\Data
    formatting and writing to hdfs"
```

Рис.3.11. Запуск потокового эксперимента

По прошествии 100 секунд мы получаем следующие результаты. Ниже можно увидеть график нагрузки, который отражает количество сообщений пришедших за время проведения эксперимента. Точно оценить нагрузку мы имеем возможность только по факту исполнения эксперимента. Зависимость поступающих в секунду сообщений от времени эксперимента приведена ниже.



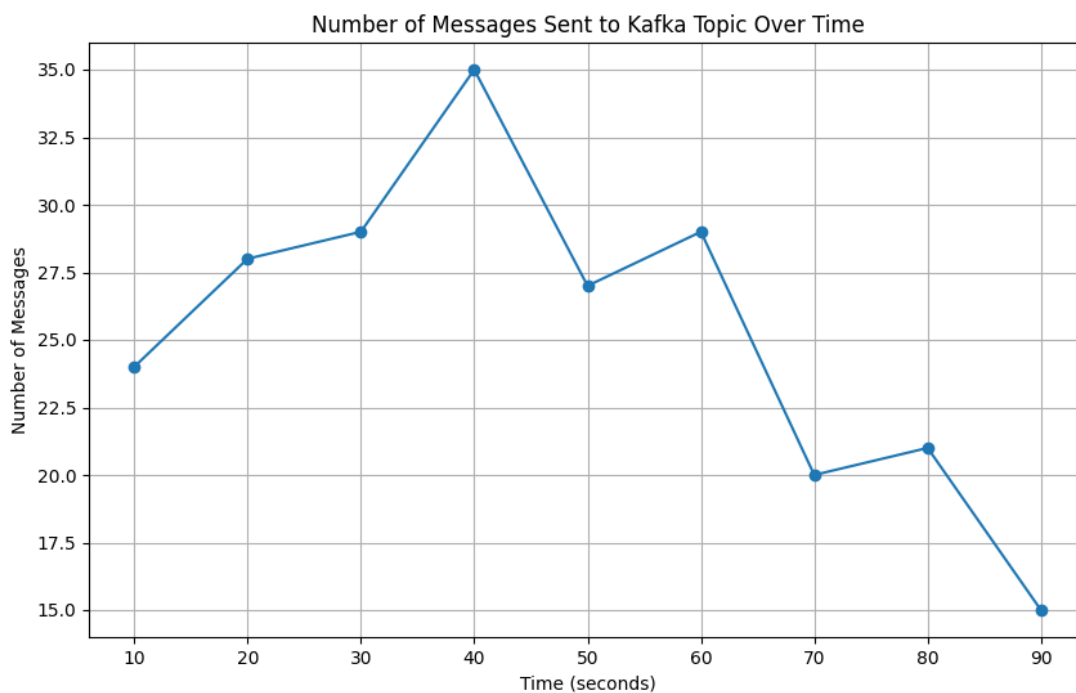


Рис.3.12. График нагрузки

Также мы получили графики задержки.

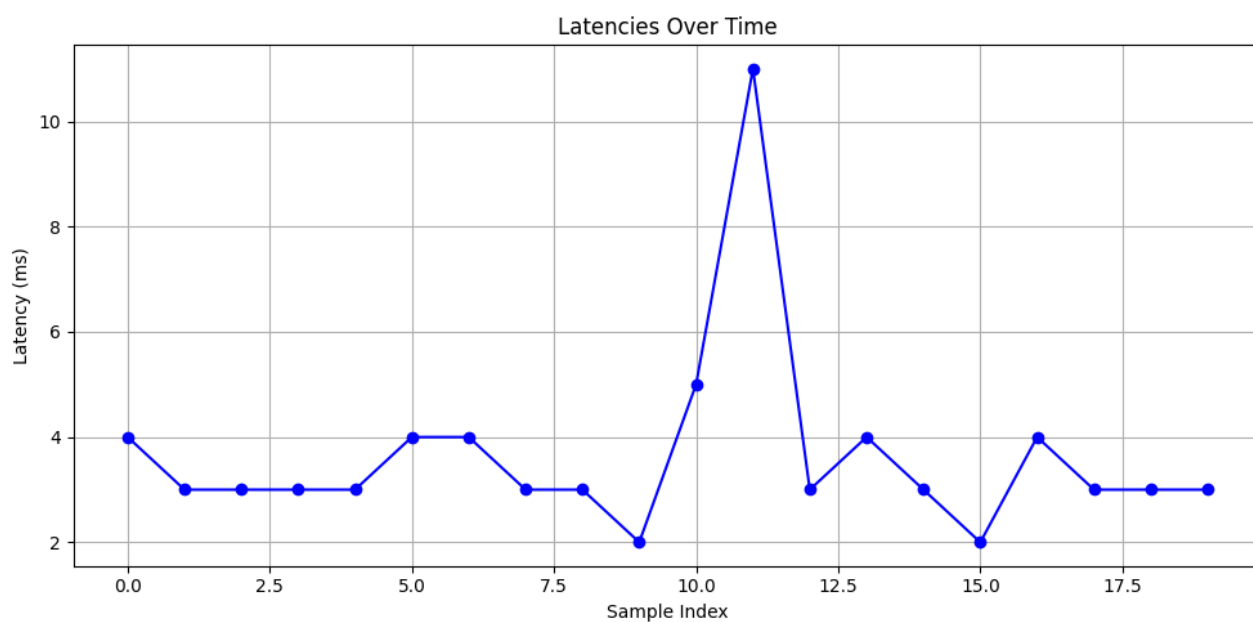


Рис.3.13. Задержка при выполнении задачи Apache Flink

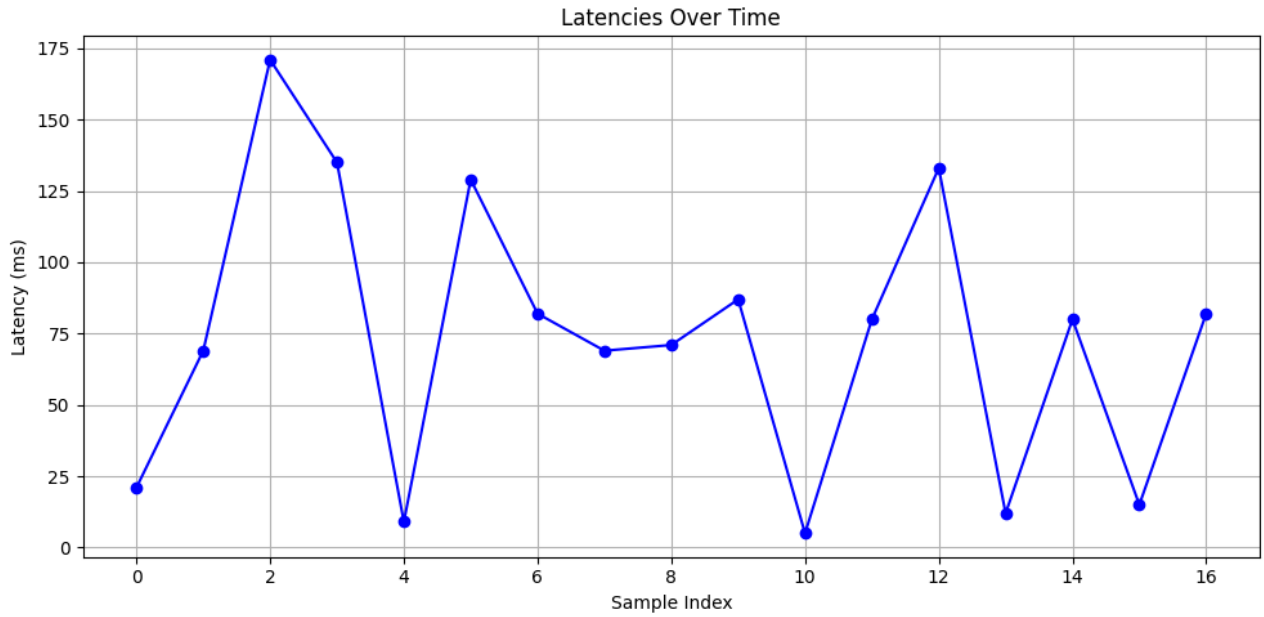


Рис.3.14. Задержка при выполнении задачи Spark Streaming

Таблица 3.1

Сравнение задержки Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	11	3.62±1.62	3
Spark Streaming	600	93.83±75.97	3

3.1.4. Средняя нагрузка

Для увеличения нагрузки к стриму, используемому в предыдущем эксперименте был добавлен стрим, который смотрело 30 тыс. человек, и в ходе этого этапа эксперимента обрабатывались сообщения с этим двух стримов.

График нагрузки:

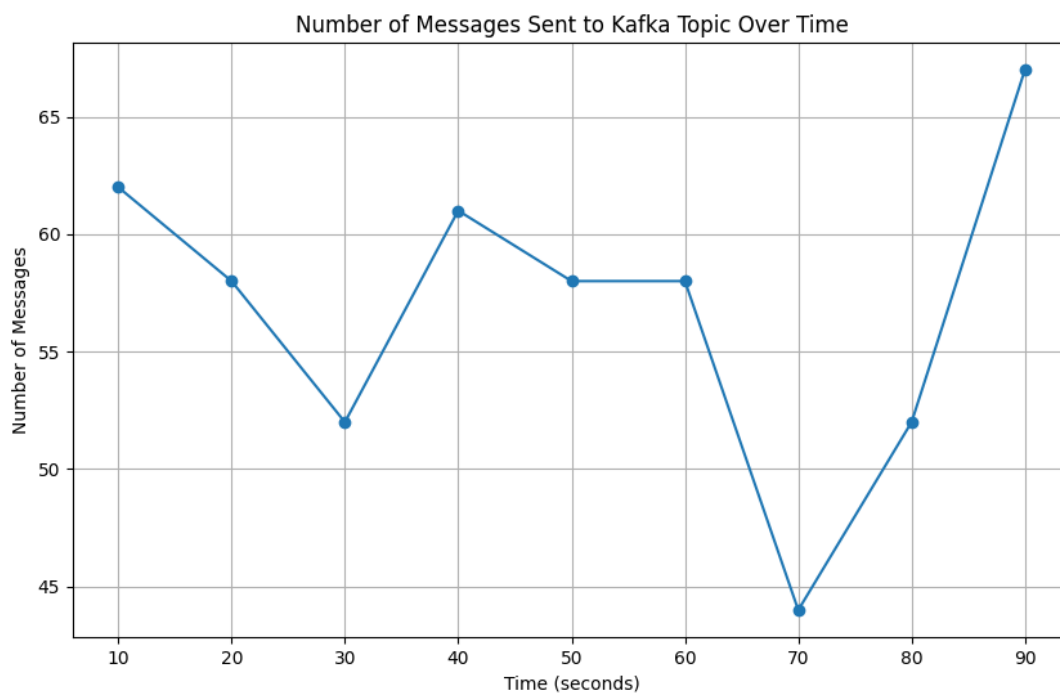


Рис.3.15. График нагрузки

График задержки:

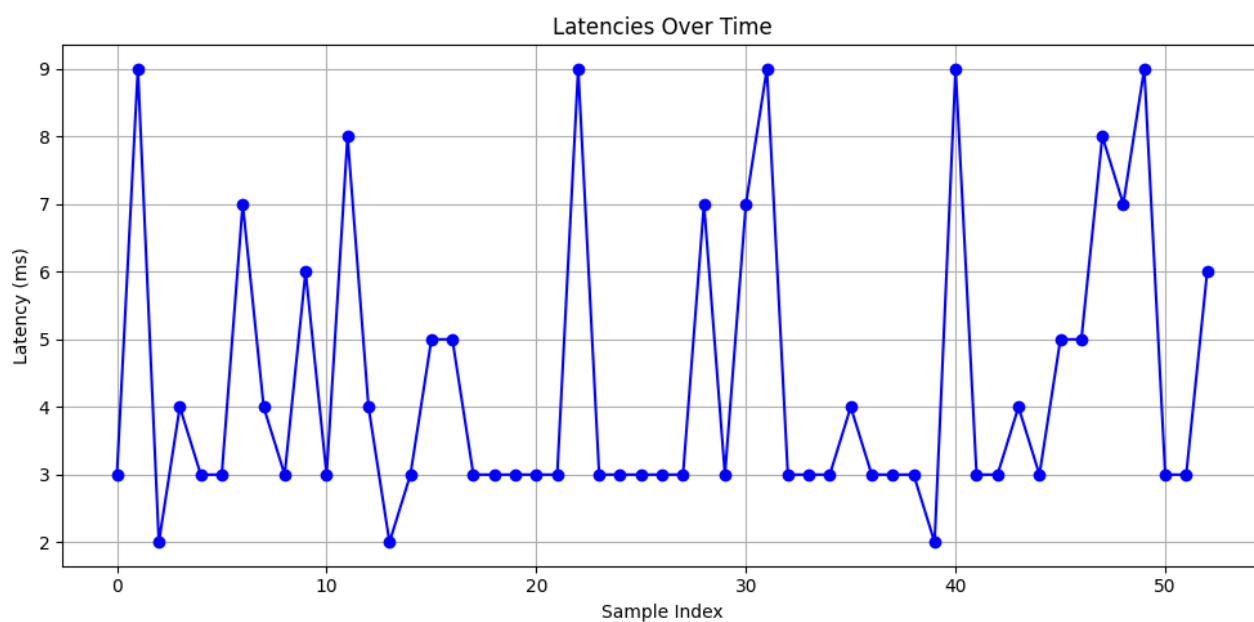


Рис.3.16. Задержка при выполнении задачи Apache Flink

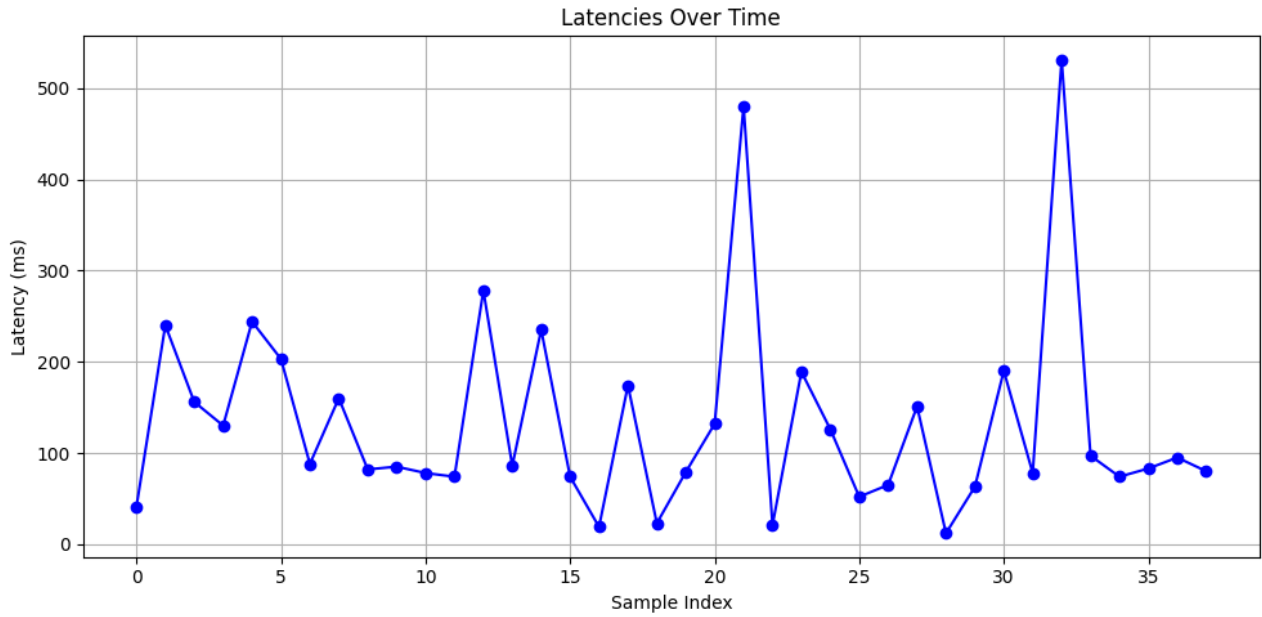


Рис.3.17. Задержка при выполнении задачи Spark Streaming

Таблица 3.2

Сравнение Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	18	4.40±2.24	2
Spark Streaming	626	137.10±99.41	2

3.1.5. Высокая нагрузка

Для формирования высокой входной нагрузки на систему помимо стримов, используемых в предыдущей стадии эксперимента использовались также сообщения с 7 дополнительных стримов, суммарная аудитория которых на момент проведения эксперимента составляла 90 тыс. человек.

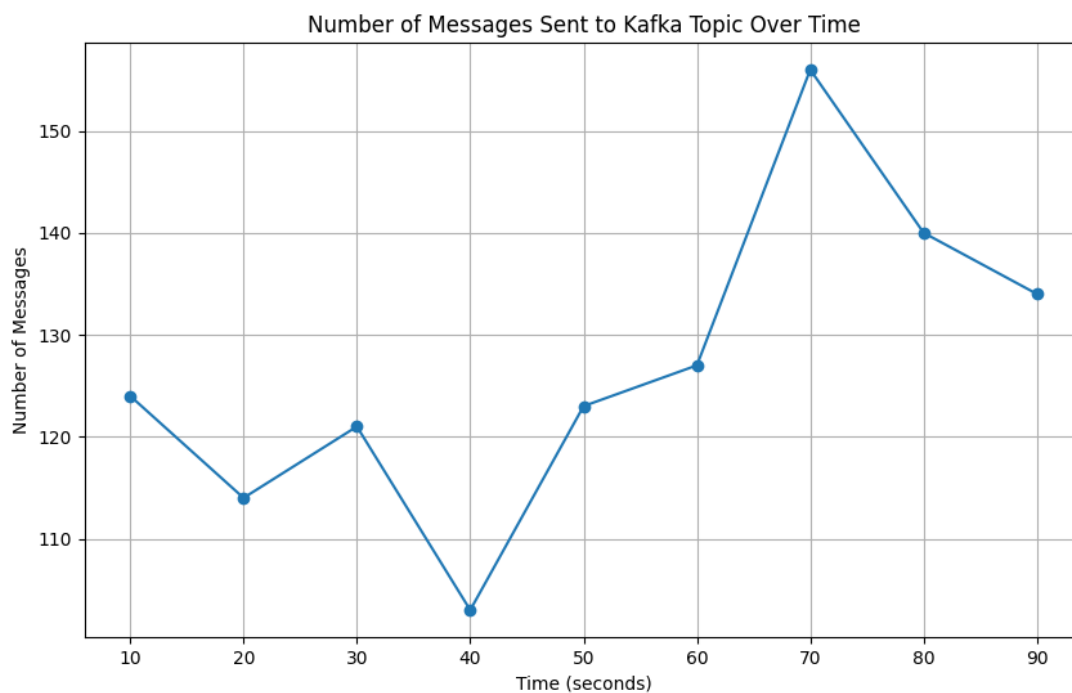


Рис.3.18. График нагрузки

График задержки:

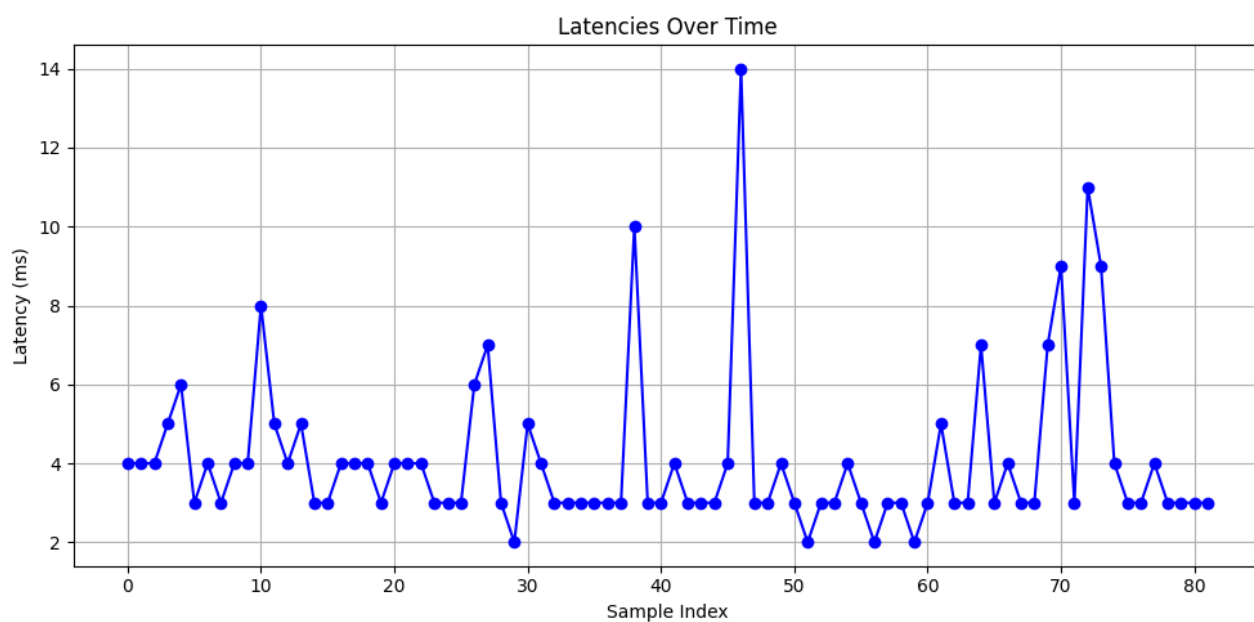


Рис.3.19. Задержка при выполнении задачи Apache Flink

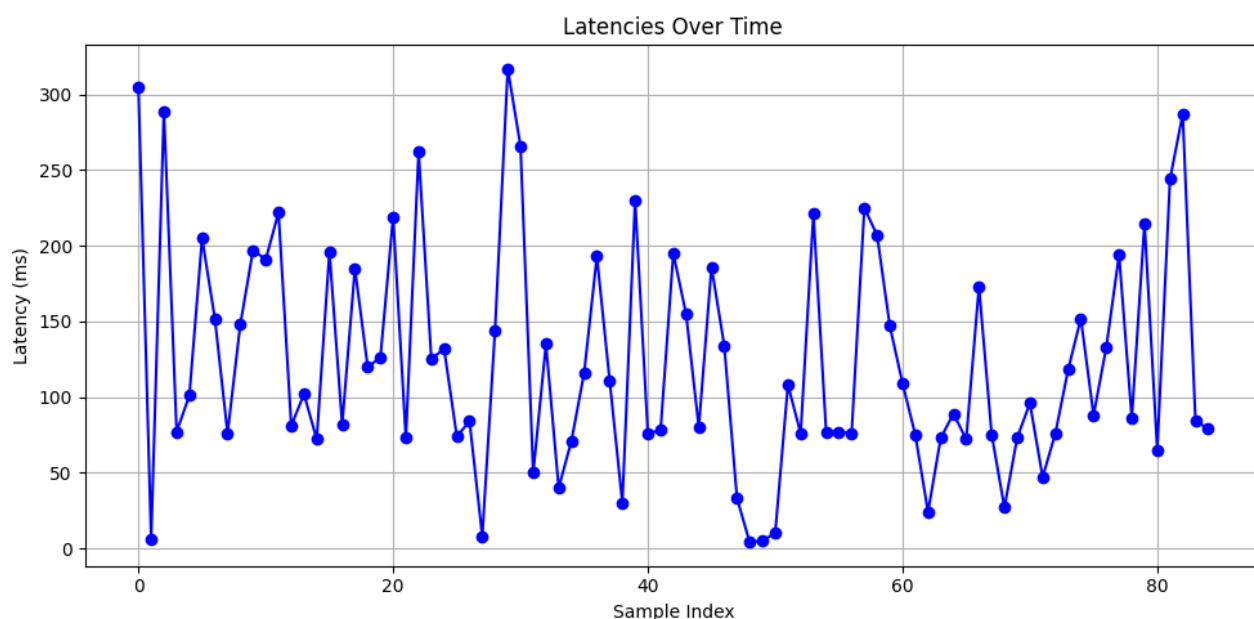


Рис.3.20. Задержка при выполнении задачи Spark Streaming

Таблица 3.3

Сравнение Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	30	$4.11 \pm 2.29$	2
Spark Streaming	457	$134.21 \pm 81.62$	2

По итогам эксперимента Apache Flink при любой конфигурации эксперимента показал меньшую задержку, чем Spark Streaming.

Оба фреймворка демонстрируют ухудшение производительности при увеличении потока данных с низкой нагрузки на среднюю. При этом, при высокой нагрузке значительных улучшений не наблюдается, хотя количество приходящих в систему сообщений возрастает в 2 раза. Это позволяет нам сделать вывод, что нагрузка в среднем в 126 сообщений в секунду (третья стадия эксперимента) не является для обоих фреймворков запредельной.

Также, стоит отметить, что Flink демонстрирует гораздо более стабильную задержку, нежели Spark Streaming, о чем можно судить по приведенным графикам задержки. Эта тенденция сохраняется на протяжении всех стадий эксперимента.

Максимальная задержка для Spark Streaming составила 626 мс, а для Apache Flink - 30 мс (при средней задержке 4.11 мс).

В ходе эксперимента Apache Flink продемонстрировал значительно меньшую среднюю(> 33 раза) задержку, а также значительно меньшую максимальную(> 15 раз) задержку, что позволяет делать предположение о его превосходстве в производительности в рамках решения задач потоковой обработки больших данных.

### 3.2. Задача цензурирования ненормативной лексики

Задача цензурирования ненормативной лексики является достаточно интересной в рамках специфики наших данных. Это одно из немногих действий, которой выгодно производить с данными чатов именно в рамках потоковой обработки данных, а не сохранять данные и позже подводить статистику по ним.

На вход обоим работам также будут подаваться данные из Kafka Produсера, но на этот раз мы не будем записывать данные в HDFS, а будем отсылать их обратно в Kafka Topic.

Результатом работы обоих работ станет список сообщений с зацензурированной ненормативной лексикой:

```
hadoop@LAPTOP-AM1MP0U8: ~/kafka_2.12-3.7.0
uviqwert::ну там не очень *****
jumle::при****
menyaent::@sasavot Д*** жалуются на меня (( Не дай вам Бог попасть туда где я отбываю наказание.
kalyvan_:@impossiblelove666 пошел н***й:qwizuz!qwizuz@qwizuz.tmi.twitch.tv PRIVMSG #sasavot :ДА, НО ИМБА
brokzn1337:@impossiblelove666 пошел ты н***й долбаеб
nuxezzy:Та ***уй
seksparenn:@lobkovaya_vsha у тебя друзей нет? н***й ты мне нужен
ekstrakprox:какого н***й деда
justcosman:@danger_sherlock ***** из тебя шерлок
unshine3:***** он 1 на 1 играл?
linaxonme:***
holdmytears:*** тут онлайн , хорош старый
entrydog:@ne\еще как щелкает, бывает ***леще. Только им всё прощается
rtfclearance:иди н***й усач
oxazepam44434:дед *****ал тебя за 15 секунд
deadslemens:9к часов псих *****
lobkovaya_vsha:***** смайлики
hayen11:@stariy_bog тебя проверяют на читы *****
kumihoo123:@screamofsoul5 ПЕРЕКЛИЧКА ДЕДОВ СТРЕЯ: **** БОЧАРОВ, ХИРОХИТО БОЧАРОШИ, БЕНИТО БОЧАРИНИ, ЙОН БОЧАРЕСКУ, БУБНО
ЛЬФ БОЧЛЕР, ХРЮЧЕЛИНИ БОЧАРЕЛЛИ, ЖИРНОЛЬД БОЧАРЕГГЕР, ЖИРНАРДО БОЧАПРИО, АШОТ БОЧАРЯН, ЛУЛУМБА БОЧАРУМБА, БУБНЕСТО БО
ЧЕВАРА, СИСИЛЕОН БОЧАПАРД, ЖИРНЕЛЬ БОЧАСТРО, ДЖО БОЧАЙДЕН.
```

Рис.3.21. Пример результата работы Flink Job, осуществляющего цензурирование данных чата Twitch

#### 3.2.1. Реализация на Apache Flink

Полный код реализации доступен в приложении 13.

Помимо шагов, совпадающих с реализацией задачи для предыдущего эксперимента, таких как инициализация среды исполнения, создания топиков для



входных данных и задержки, мы также создаем дополнительный Kafka Topic, как раз для отфильтрованных сообщений:

```
String filteredTopic = "filtered_messages";  
createKafkaTopic(kafkaBootstrapServers, filteredTopic);
```

Рис.3.22. Инициализация Kafka Топика для очищенных от нецензурной лексики сообщений

Основной функционал, задержку в работе которого мы и будем тестировать, лежит в следующем участке кода:

```

DataStream<String> text = env.addSource(kafkaConsumer);
List<String> curseWords = loadCurseWords("curse_words.txt");
DataStream<Message> messages = text.map(new MapFunction<String
    , Message>() {
5         @Override
        public Message map(String value) {
            try {
                String[] parts = value.split(" ");
                if (parts.length < 5) {
10                 throw new IllegalArgumentException("
                    Unexpected message format: " +
                        value);
                }

                long arrivalTimestamp = Long.parseLong(
                    parts[0]);
                String user = parts[1].split("!")[0].
                    substring(1);
15                String msg = value.split(":", 3)[2].
                    replace("\n", "").replace("\r", "").
                    trim();

                msg = censorText(msg, curseWords);

                long processingTimestamp = System.
                    currentTimeMillis();
20

                return new Message(arrivalTimestamp, user,
                    msg, processingTimestamp);
            } catch (Exception e) {
                e.printStackTrace();
                return null;
25            }
        }
    }).filter(message -> message != null);

```

Рис.3.23. Фрагмент кода, выполняющий фильтрацию нецензурной лексики из входящих сообщений

Kafka Topic, содержащий сообщения из чатов становится источником для объекта Data Stream(часть библиотеки Flink), и в методе Map описываются манипуляции, которые мы будем производить с поступающими сообщениями. На этот раз мы не извлекаем информации о стримере, нам нужен только идентификатор

пользователя, и само сообщение. После чего мы применяем к сообщению функцию `sensorText`, и преобразованное сообщение отправляем в инициализированный в предыдущем приведенном участке кода Kafka Topic.

### 3.2.2. Реализация на Spark Streaming

Теперь рассмотрим реализацию той же задачи на Spark Streaming.

Полный код реализации доступен в приложении 14.

Мы также инициализируем дополнительный Kafka Topic:

```
2 kafka_bootstrap_servers = "localhost:9092"
3 kafka_topic = "irc_messages2"
4 filtered_topic = "filtered_messages"
5 latency_topic = "spark_latency"
```

Рис.3.24. Данные для инициализации Kafka Topicов

Далее считываем слова, которые относятся к ненормативной лексике из файла и используя Spark Data Frame Api преобразовываем приходящие данные в нужный нам вид:

```

2 messages = lines.selectExpr("CAST(value AS STRING) as value")
  \
3   .select(split(col("value"), " ", 5).alias("parts")) \
4   .selectExpr(
5       "parts[0] as arrival_timestamp",
6       "split(parts[1], '!')[0] as raw_username",
7       "parts[4] as message"
8   ) \
9   .selectExpr(
10      "CAST(arrival_timestamp AS LONG) as arrival_timestamp"
11      ,
12      "substring(raw_username, 2, length(raw_username) - 1)
13      as user",
14      "substring(message, 2, length(message) - 1) as msg"
15  ) \
16  .withColumn("msg", regexp_replace(col("msg"), "\n|\r", "")) \
17  .withColumn("msg", censor_udf(col("msg"))) \
18  .withColumn("msg", trim(col("msg"))) \
19  .withColumn("filtered_msg", censor_udf(col("msg"))) \
20  .withColumn("processing_timestamp", (unix_millis(
21      current_timestamp()).cast(LongType()))

```

Рис.3.25. Фрагмент кода, выполняющий фильтрацию сообщений с использованием Spark Data Frame API

Пренебрегая записью в HDFS отправляем преобразованные данные в Kafka Topic:

```

2 query = messages.writeStream \
3   .outputMode("append") \
4   .foreachBatch(send_to_kafka) \
5   .start()
6
7 query.awaitTermination()

```

Рис.3.26. Отправка отфильтрованных сообщений в Kafka Topic

### 3.2.3. Низкая нагрузка

Все фазы эксперимента будут длиться 100 секунд. С конфигурацией фаз эксперимента, реализовывающего задачу фильтрации ненормативной лексики из сообщений можно ознакомиться в приложении 14.

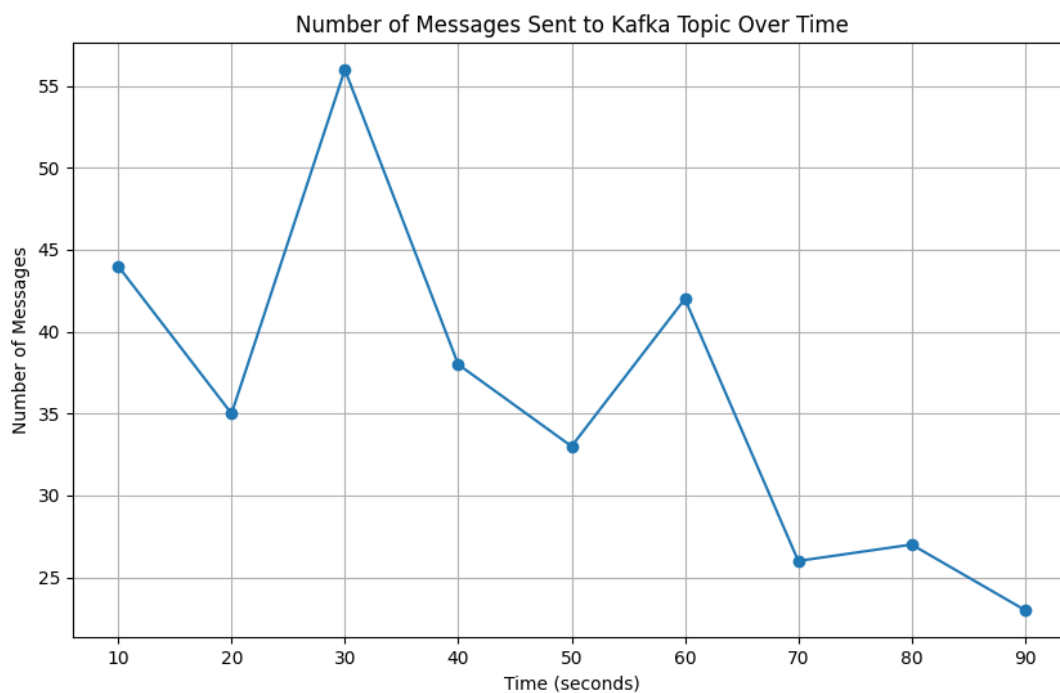


Рис.3.27. График нагрузки

Также мы получили графики задержки.

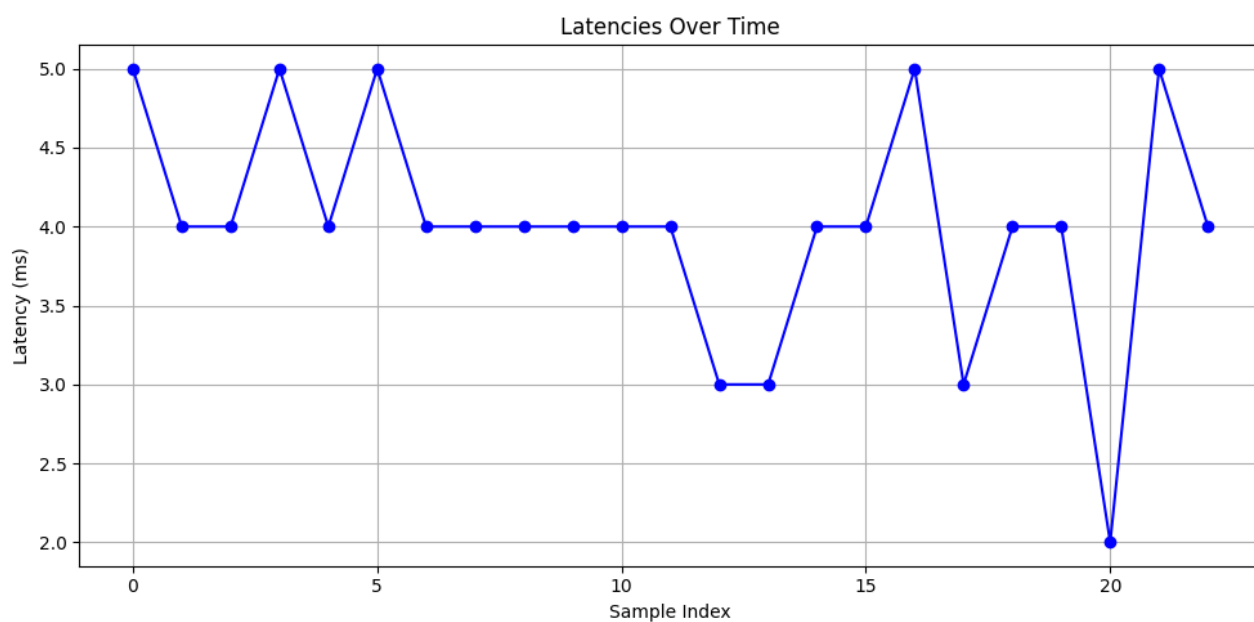


Рис.3.28. Задержка при выполнении задачи Apache Flink

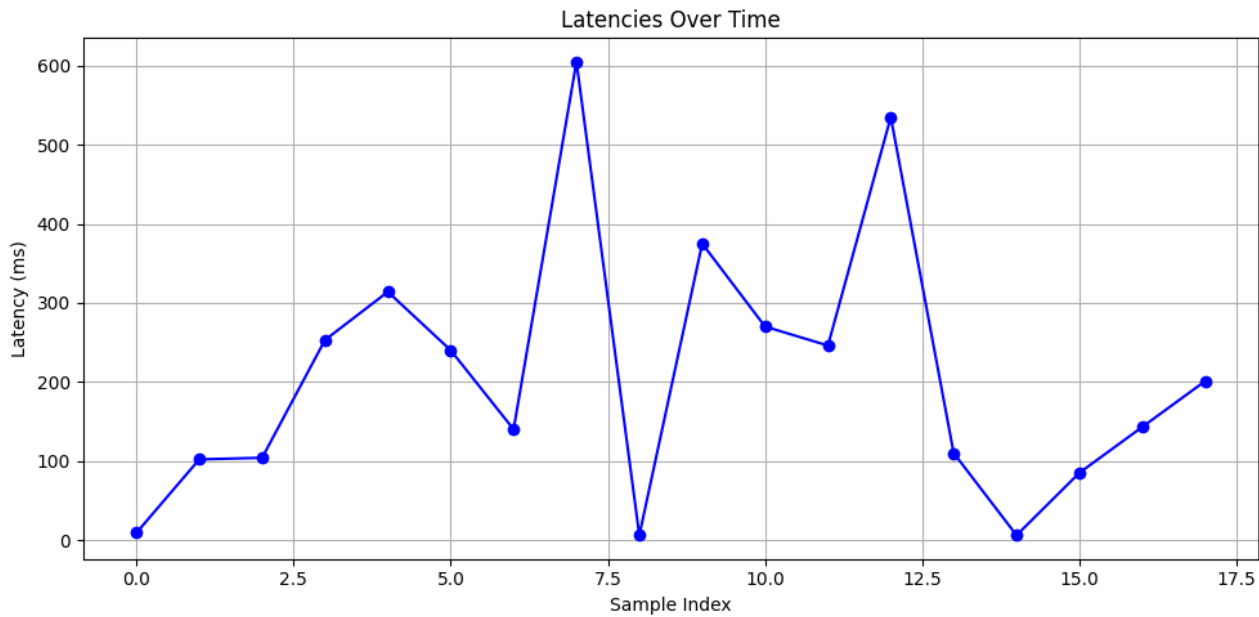


Рис.3.29. Задержка при выполнении задачи Spark Streaming

Таблица 3.4

Сравнение задержки Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	14	4.24±1.55	2
Spark Streaming	670	209.45±156.96	2

3.2.4. Средняя нагрузка

Для увеличения нагрузки к стриму, используемому в предыдущем эксперименте был добавлен стрим, который смотрело 30 тыс. человек, и в ходе этого этапа эксперимента обрабатывались сообщения с этим двух стримов.

График нагрузки:

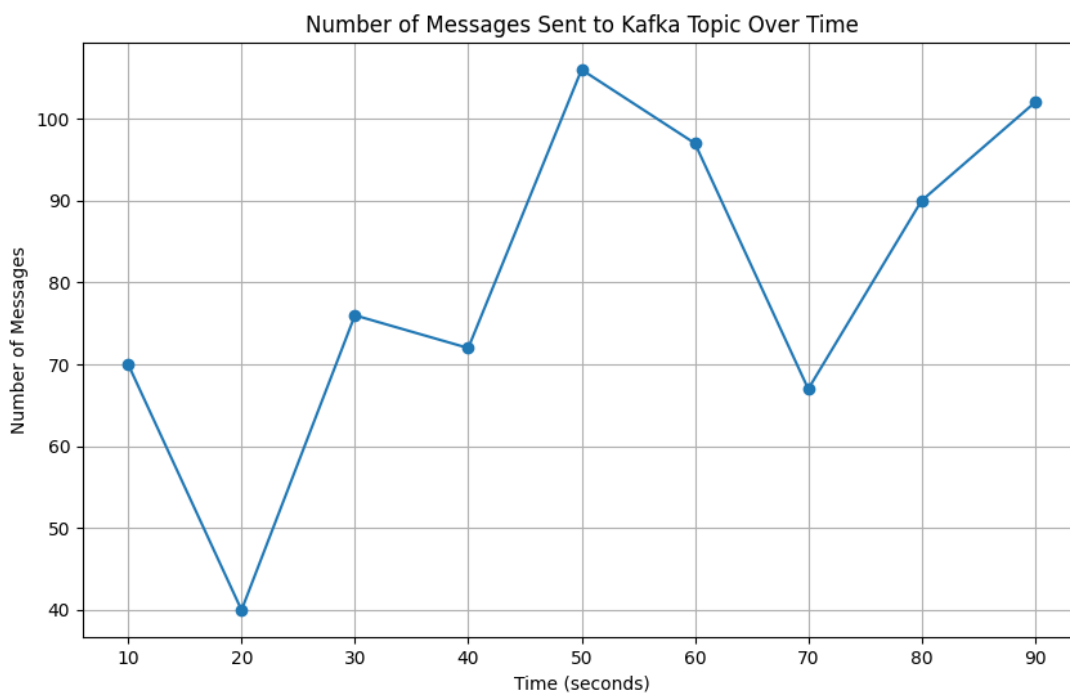


Рис.3.30. График нагрузки

График задержки:

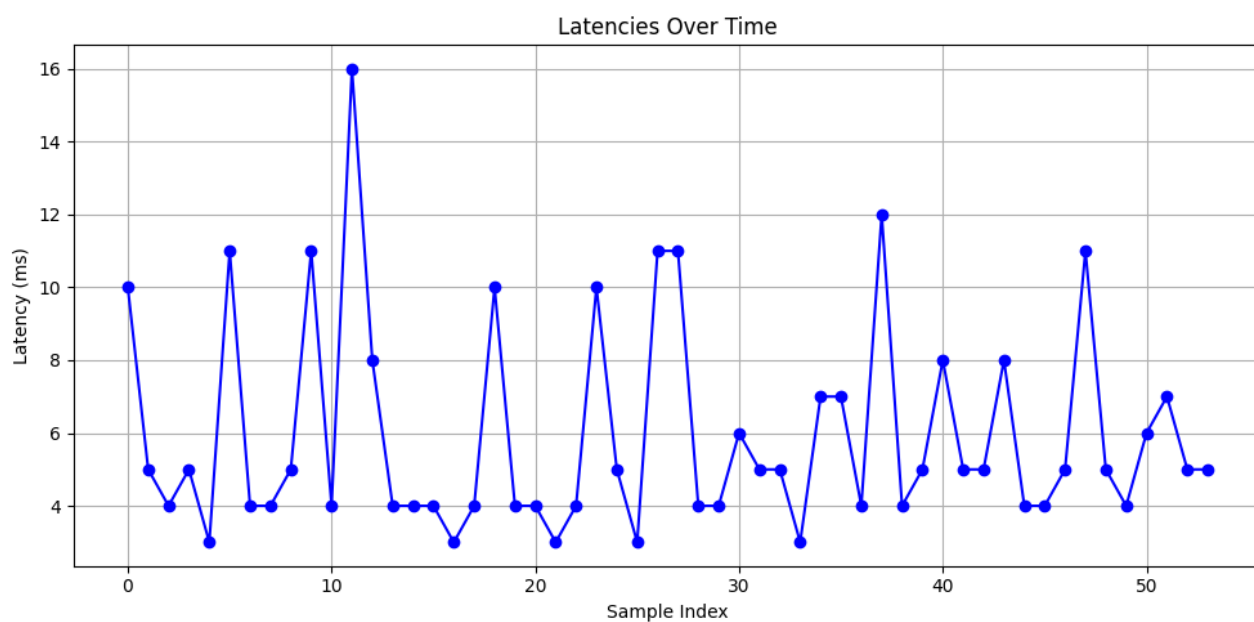


Рис.3.31. Задержка при выполнении задачи Apache Flink



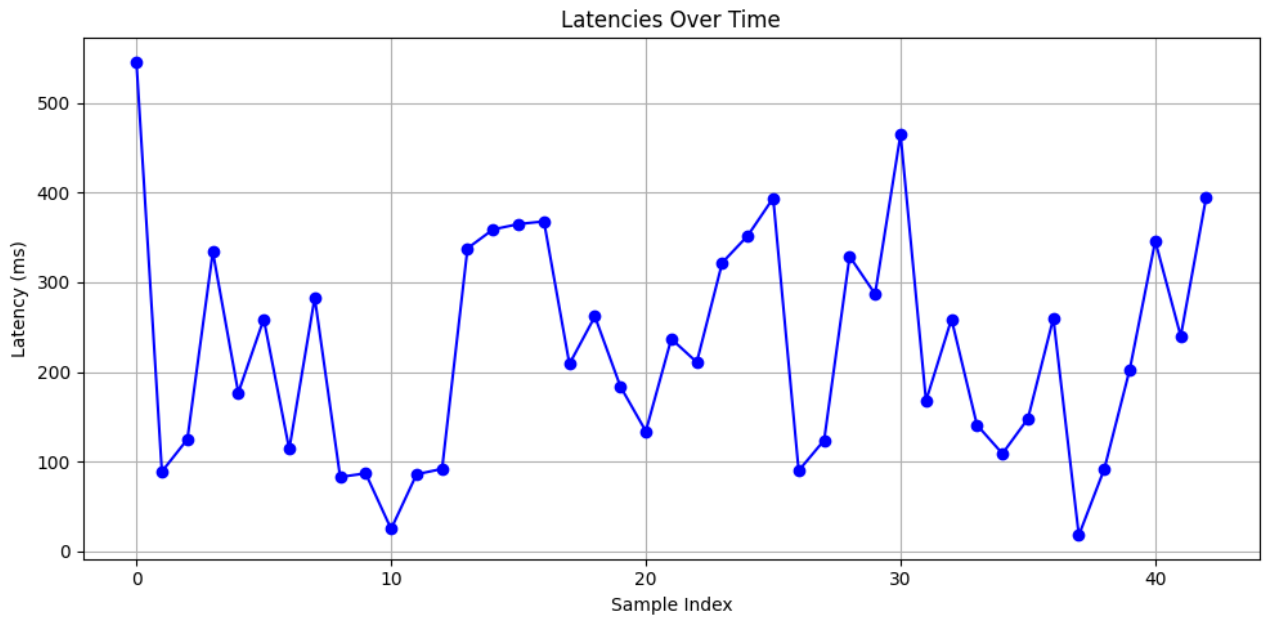


Рис.3.32. Задержка при выполнении задачи Spark Streaming

Таблица 3.5

Сравнение Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	42	6.02±3.53	2
Spark Streaming	702	205.11±124.51	4

3.2.5. Высокая нагрузка

Для формирования высокой входной нагрузки на систему помимо стримов, используемых в предыдущей стадии эксперимента использовались также сообщения с 7 дополнительных стримов, суммарная аудитория которых на момент проведения эксперимента составляла 90 тыс. человек.

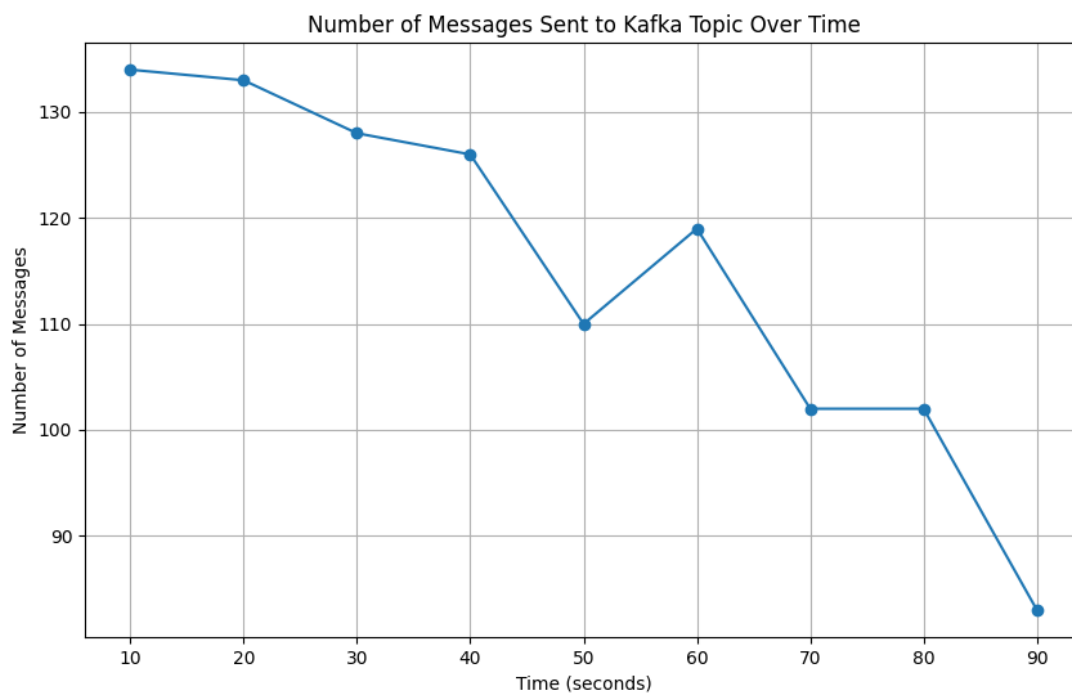


Рис.3.33. График нагрузки

График задержки:

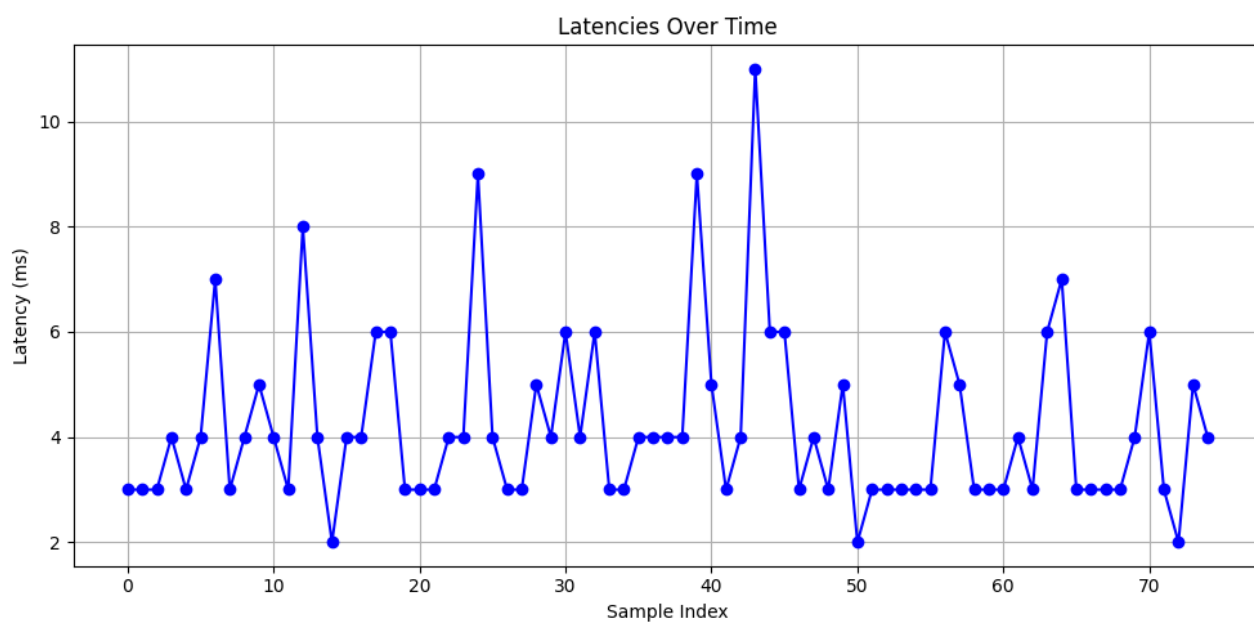


Рис.3.34. Задержка при выполнении задачи Apache Flink

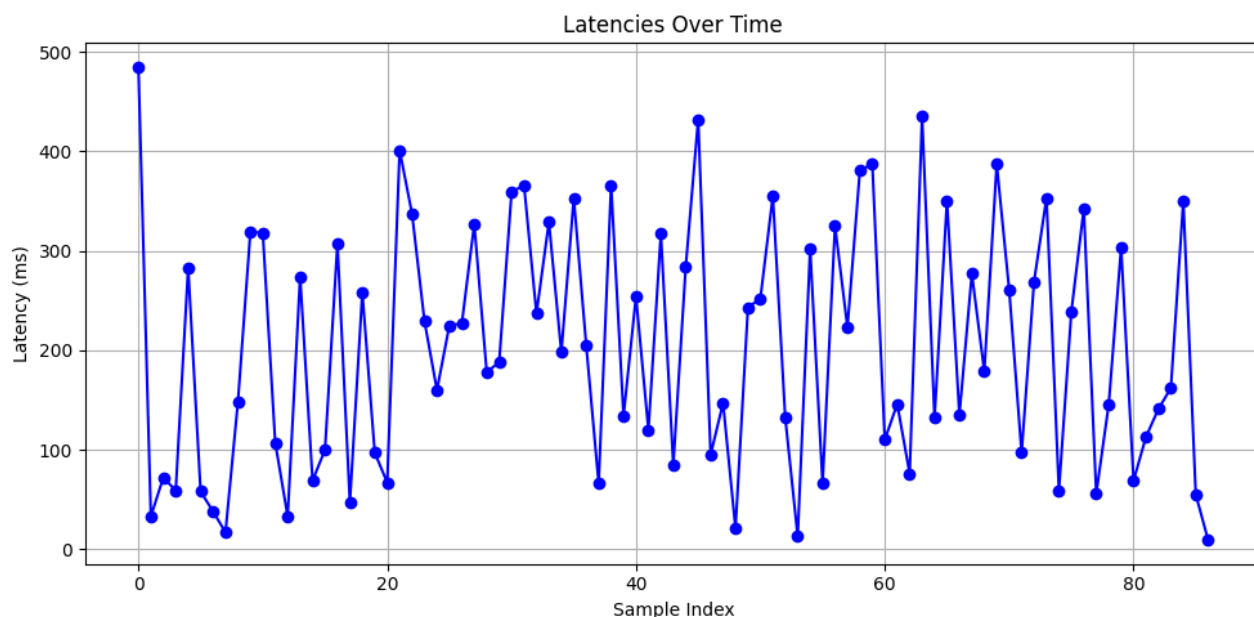


Рис.3.35. Задержка при выполнении задачи Spark Streaming

Таблица 3.6

Сравнение Spark Streaming и Apache Flink

Фреймворк	Максимальная задержка	Средняя задержка	Минимальная задержка
Apache Flink	15	$4.63 \pm 2.24$	2
Spark Streaming	743	$216.53 \pm 123.33$	4

Apache Flink и в рамках этого эксперимента показал значительно меньшую задержку, чем Spark Streaming.

Средняя задержка Spark Streaming превысила задержку, демонстрируемую Apache Flink более, чем в 50 раз, а максимальная - более, чем в 49 раз.

Данная задача была вычислительно более сложной, так как помимо разделения строки входных данных на составные части, мы также вынуждены были пробегаться по самому сообщению и искать в нем ненормативную лексику. Это отразилось на результатах эксперимента, средняя задержка, как у Apache Flink, так и у Spark Streaming оказалась выше, чем в предыдущем эксперименте. Тем не менее, доминация Apache Flink по производительности сохранилась и в этом эксперименте, что позволяет нам сделать выводы о выгоде использования Apache Flink и Spark Streaming для потоковой обработки данных.

### 3.3. Выводы по итогам экспериментов

Итак, в ходе экспериментов оба фреймворка работали с одинаковыми данными, одинаковое количество времени при этом потребляя одинаковое количество аппаратных ресурсов. По истечении каждого этапа экспериментов были проанализированы результаты работы программ и в случае совпадения результатов данные эксперимента описывались выше. Каждый этап каждого эксперимента проводился несколько раз, чтобы уменьшить влияние случайных обстоятельств на исход эксперимента. Учитывая вышесказанное, считаю, проведенные эксперименты релевантными для описания задержки при выполнении задач потоковой обработки данных с использованием Spark Streaming и Apache Flink.

Apache Flink продемонстрировал как значительно более низкую среднюю задержку между приходом сообщения в систему и завершением его обработки, так и значительно более стабильную задержку. В ходе экспериментов задержка Apache Flink не была выше 42 миллисекунд, что является отличным результатом, демонстрирующим, что Apache Flink, при объявленных в эксперименте условиях не допускает значительных задержек в обработке потоков данных.

Spark Streaming в сравнении с Apache Flink показал себя слабее, средняя задержка в 250 миллисекунд, а максимальная доходит до 702 мс. Тем не менее, даже этот результат является подходящим для потоковой обработки больших данных, если задержка 1 с является приемлемой в рамках конкретной задачи, или задержка не важна вовсе, а важна надежность системы и гарантия обработки входящих сообщений.

Так же во время экспериментов проводился анализ потребления оперативной памяти обоими фреймворками. Ввиду специфики задач, а именно потоковой обработки данных, где данные после поступления записываются в хранилище или отправляются далее в Kafka Consumer, рост потребляемой оперативной памяти в зависимости от потока данных и с течением времени не наблюдался, так как и Spark Streaming и Apache Flink обладают механизмами для автоматической очистки мусора, и как только данные обработаны, они очищаются из оперативной памяти.

## ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНОЕ СРАВНЕНИЕ ФРЕЙМВОРКОВ ДЛЯ ПАКЕТНОЙ ОБРАБОТКИ ДАННЫХ: HADOOP MAPREDUCE, APACHE HIVE И SPARK

По аналогии с экспериментальным сравнением фреймворков для потоковой обработки данных, фреймворки для пакетной обработки данных будут сравниваться в рамках решения прикладных в предметной области задач.

Каждым эксперимент будет проводиться на трех наборах данных. Они были заранее собраны с помощью скрипта, используемого для тестирования фреймворков для потоковой обработки данных.

При проведении экспериментов все ресурсы, доступные WSL выделялись на выполнения текущей задачи пакетной обработки данных, а именно 8Гб оперативной памяти и 4 ядра процессора.

Таблица 4.1

Объем датасетов

Набор данных	Количество записей	Объем(мб)
/user/hadoop/small	34650	346.21
/user/hadoop/medium	69106	683.90
/user/hadoop/large	237531	2350.70

Имея информацию о датасетах мы можем провести пробный эксперимент.

### 4.1. Задача подсчёта количества записей в датасетах

Первый эксперимент будет содержать очень простую операцию. Мы должны будем перебрать все записи в каждой из hdfs директорий и посчитать их количество.

#### 4.1.1. Реализация на Apache Hive

Реализация подобной операции на Apache Hive максимально проста и понятна. Сначала мы создаем таблицу, которая станет представлением наших данных в рамках Hive:

```
| SELECT COUNT(*) from small;
```

Рис.4.1. HiveQL код для выполнения подсчёта записей в инициализированной как таблице, HDFS директории

В данном примере указывается директория small, для других двух наборов данных нужно только изменить путь к директории, операция будет выполняться так же.

#### 4.1.2. Реализация на Map Reduce

Реализация на Map Reduce содержит три основных части: Reducer Class, Mapper Class и Driver Class.

Driver Class это административный класс, с помощью которого мы назначаем Reducer и Driver классы и также определяем ввод и вывод программы, его реализация будет почти идентичной для каждой задачи.

Mapper класс считывает входные данные, разбитые на отдельные куски (input splits), и обрабатывает их. В каждой из этих частей Mapper выполняет начальную обработку данных, преобразовывая их в пары ключ-значение, эти пары затем передаются для дальнейшей обработки в фазу сортировки и смешивания, где они группируются по ключам и отправляются в Reducer.

```
| public class LineCountMapper extends Mapper<LongWritable, Text
|     , NullWritable, LongWritable> {
|     private final static LongWritable one = new LongWritable
|         (1);
5 |     @Override
|     protected void map(LongWritable key, Text value, Context
|         context) throws IOException, InterruptedException {
|         context.write(NullWritable.get(), one);
|     }
| }
```

Рис.4.2. Реализация Mapper'a

Reducer принимает отсортированные пары ключ-значение, сгруппированные по ключам. Он выполняет агрегирующую операцию на этих сгруппированных данных, например, суммирование значений для каждого ключа. Результаты агрегации передаются обратно в Driver Class.

```

public class LineCountReducer extends Reducer<NullWritable,
    LongWritable, NullWritable, LongWritable> {
    @Override
    protected void reduce(NullWritable key, Iterable<
        LongWritable> values, Context context) throws
        IOException, InterruptedException {
5       long sum = 0;
        for (LongWritable value : values) {
            sum += value.get();
        }
        context.write(NullWritable.get(), new LongWritable(sum
10        ));
    }
}

```

Рис.4.3. Реализация Reducera

Теперь рассмотрим реализацию той же задачи на Spark.

#### 4.1.3. Реализация на PySpark

Синтаксис PySpark, а также порядок действий при инициализации работ почти ничем не отличается от Spark Streaming.

```

2     spark = SparkSession.builder.appName("Count Entities").
        getOrCreate()
3
4     hdfs_path = "hdfs://localhost:9000/user/hadoop/small"
5
6     schema = "arrival_timestamp BIGINT, user STRING, streamer
        STRING, msg STRING, processing_timestamp BIGINT"
7
8     df = spark.read.format("csv").option("header", "false").
        schema(schema).load(hdfs_path)
9
10    count = df.count()
11
12    print(f"Number of entities: {count}")
13
14    spark.stop()

```

Рис.4.4. Spark работа, выполняющая подсчет количества записей в заданной HDFS директории



Так же, как и в Spark Streaming мы работаем с программной абстракцией Data Frame. После создания сессии, объявляем схему и путь к hdfs директории. Считываем данные из формата csv и применяем агрегирующую функцию из Data Frame API - count.

#### *4.1.4. Результаты эксперимента*

Итак, реализованные задачи были запущены для трех наборов данных. В результате мы получили следующие результаты по времени выполнения задач:

Таблица 4.2

Сравнение времени выполнения задач Map Reduce, Spark и Apache Hive

<b>Фреймворк</b>	<b>Small</b>	<b>Medium</b>	<b>Large</b>
Map Reduce	127	224	768
Apache Hive on Tez	59	112	347
Spark	49	78	232

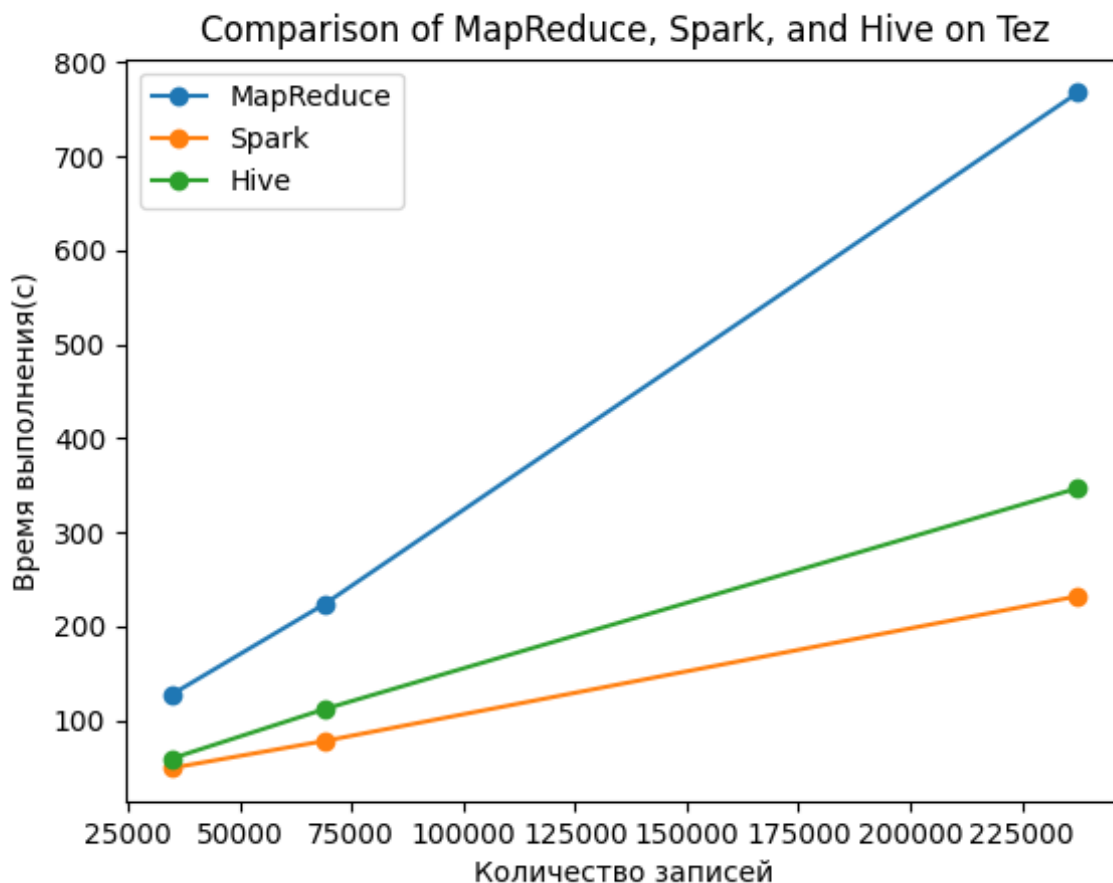


Рис.4.5. График времени выполнения Map Reduce, Spark и Apache Hive задач на разных наборах данных

По итогам эксперимента, выяснилось, что Spark демонстрирует гораздо лучшую производительность при выполнении задач пакетной обработки данных, что было ожидаемо до проведения эксперимента. Интересно, что Apache Hive показал себя немного лучше Map Reduce на среднем наборе данных, но хуже на большом наборе данных, хотя руководствуясь природой Apache Hive, как транслятора HiveQL в задачи Map Reduce, он должен был продемонстрировать худшую производительность за счет времени компиляции. Природа этого происшествия будет выяснена в дальнейших экспериментах.

## 4.2. Задача фильтрации брани в хранимых записях

Мы провели первый эксперимент, но, как упомянуто выше, для того, чтобы делать выводы о производительности фреймворков нужно провести несколько экспериментов. Далее мы напишем новые задачи, относящиеся к пакетному



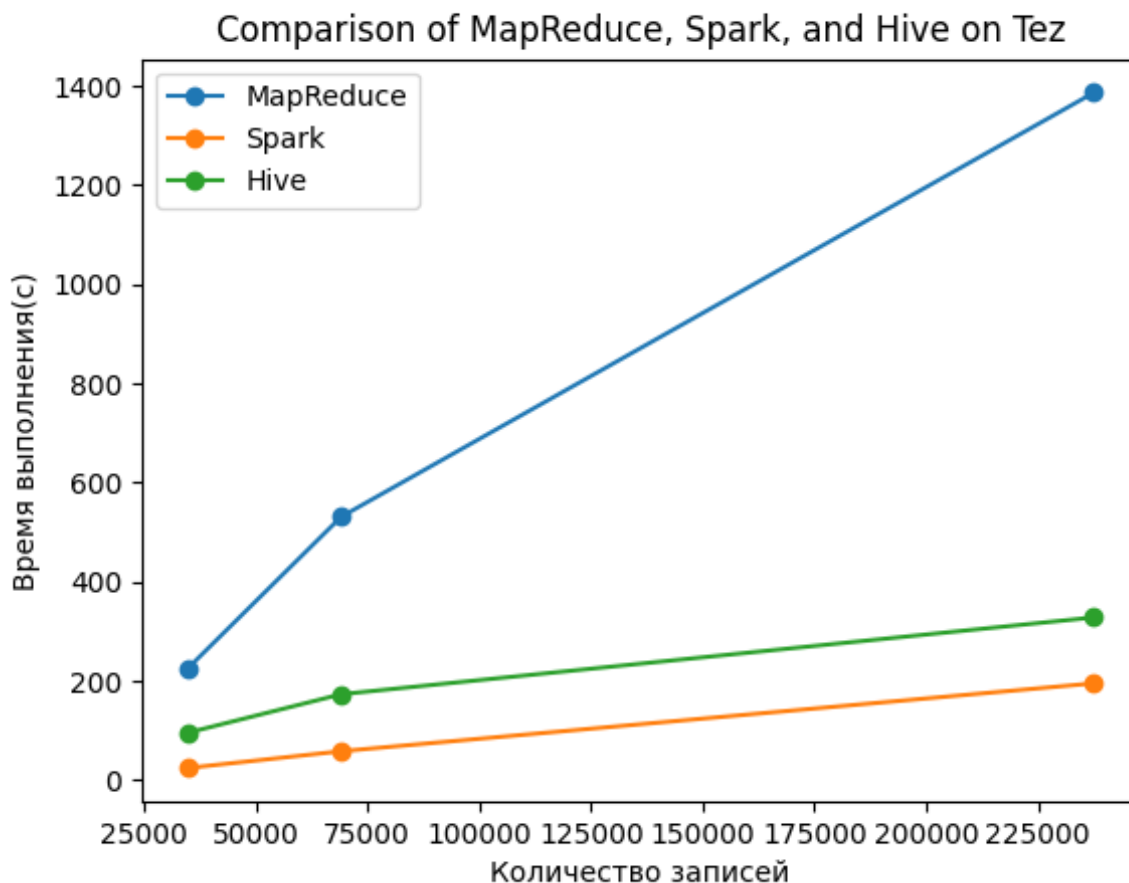


Рис.4.7. График времени выполнения Map Reduce, Spark и Apache Hive задач на разных наборах данных

Spark и в этом эксперименте показал себя производительнее всех. Стоит отметить, что разница во времени выполнения задачи между Hive on Tez и Spark не так значительна, как разница между Map Reduce и Hive on Tez, Map Reduce и Spark. Несмотря на это, характер зависимости, судя по графику, у всех трех фреймворков одинаковый, близок к линейному.

#### 4.3. Задача формирования топа стримеров по количеству сообщений

Следующая задача - формирование топа стримеров по количеству сообщений. Для системы эта задача является достаточно комплексной, вовлекая в себя, как группировку, так и сортировку большого количества данных.

Реализации задач на HiveQL, PySpark и MapReduce содержится в приложении 17.

Сообщения группируются по имени стримера, а затем упорядочиваются, в результате получается таблица следующего вида:

```

+-----+-----+
|streamer|count|
+-----+-----+
|silvername|947|
|c_a_k_e|654|
|dreadztv|431|
|iltw1|410|
|betboom_ru|122|
|caedrel|21|
+-----+-----+

```

Рис.4.8. Пример результата выполнения задачи по составлению топа стримеров по количеству сообщений

После запуска Hive on Tez, Spark и MapReduce работ получаем следующие результаты эксперимента:

Таблица 4.4

Сравнение времени выполнения задач Map Reduce, Spark и Apache Hive

Фреймворк	Small	Medium	Large
Map Reduce	160	361	1514
Apache Hive on Tez	89	139	806
Spark	72	116	673

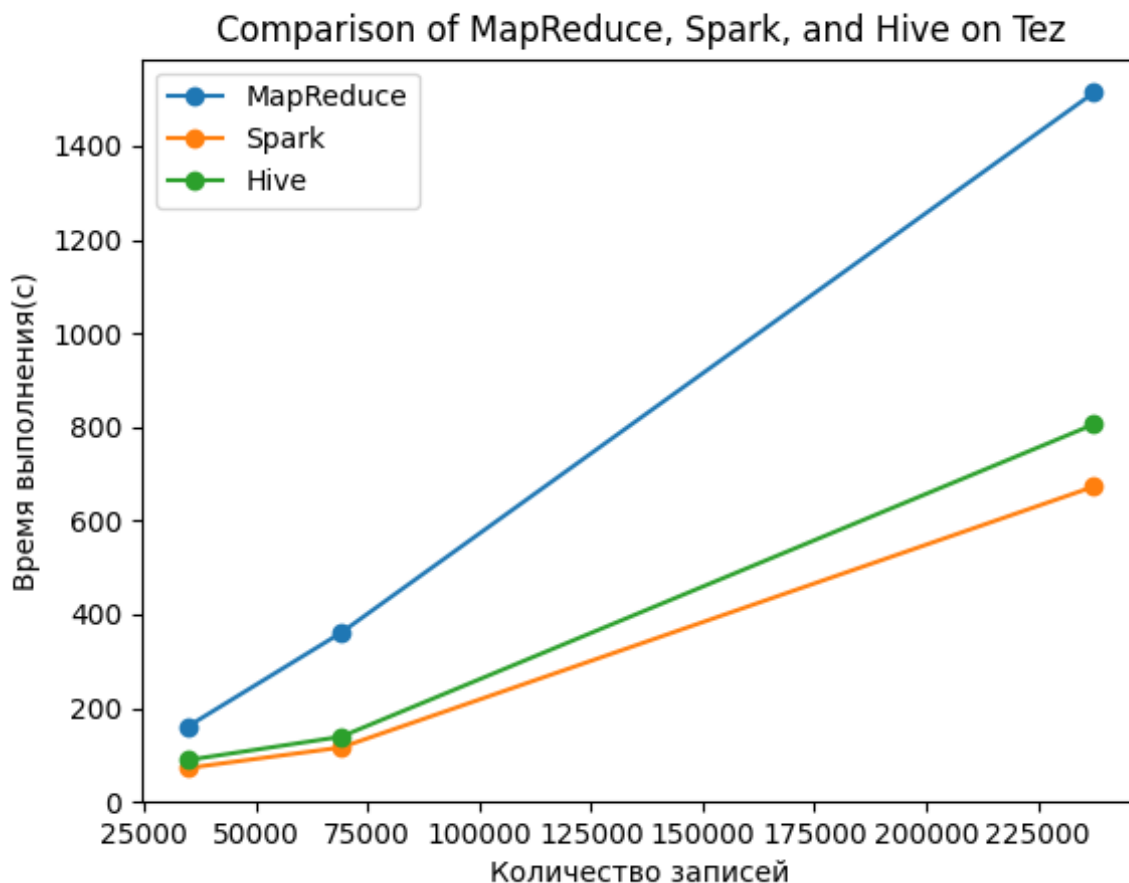


Рис.4.9. График времени выполнения Map Reduce, Spark и Apache Hive задач на разных наборах данных

Spark и Hive on Tez продемонстрировали значительно лучшую производительность, чем Map Reduce. Тем не менее, разница во времени выполнения не настолько значительная, как в предыдущей задаче. Время выполнения на самом наборе данных у Map Reduce отличается чуть менее чем в 2 раза от времени выполнения Spark, и чуть более, чем в 2 раза от Hive on Tez.

#### 4.4. Задача формирования топа пользователей по количеству сообщений

Следующая задача - формирование топа пользователей по количеству отправленных сообщений. Задача вовлекает те же функции фреймворков, что и предыдущая. Реализации работ для Map Reduce, Hive и PySpark также похожи, их можно найти в приложении 18.

Сообщения группируются по имени пользователя, а затем упорядочиваются, в результате получается таблица следующего вида:

user	count
cjlakc	471
yoshkinkrot	271
botblin	269
senyasv	218
ya_morphe_b_kedax	205
obuzameduza	201
lin_oq	196
moobot	182
sanchoszz	182
nickls_	170

Рис.4.10. Пример результата выполнения задачи по составлению топа пользователей по количеству сообщений

Примечательно, что в топе пользователей окажется moobot - бот, отправляющий раз в определенный промежуток времени заданные стримером сообщения. Это достаточно распространенный бот, которым пользуются многие стримеры. Остальные пользователи, оказавшиеся в топе скорее всего "спамили" сообщениями.

После запуска Hive on Tez, Spark и MapReduce работ получаем следующие результаты эксперимента:

Таблица 4.5

Сравнение времени выполнения задач Map Reduce, Spark и Apache Hive

Фреймворк	Small	Medium	Large
Map Reduce	158	260	1415
Apache Hive on Tez	87	144	701
Spark	73	123	620



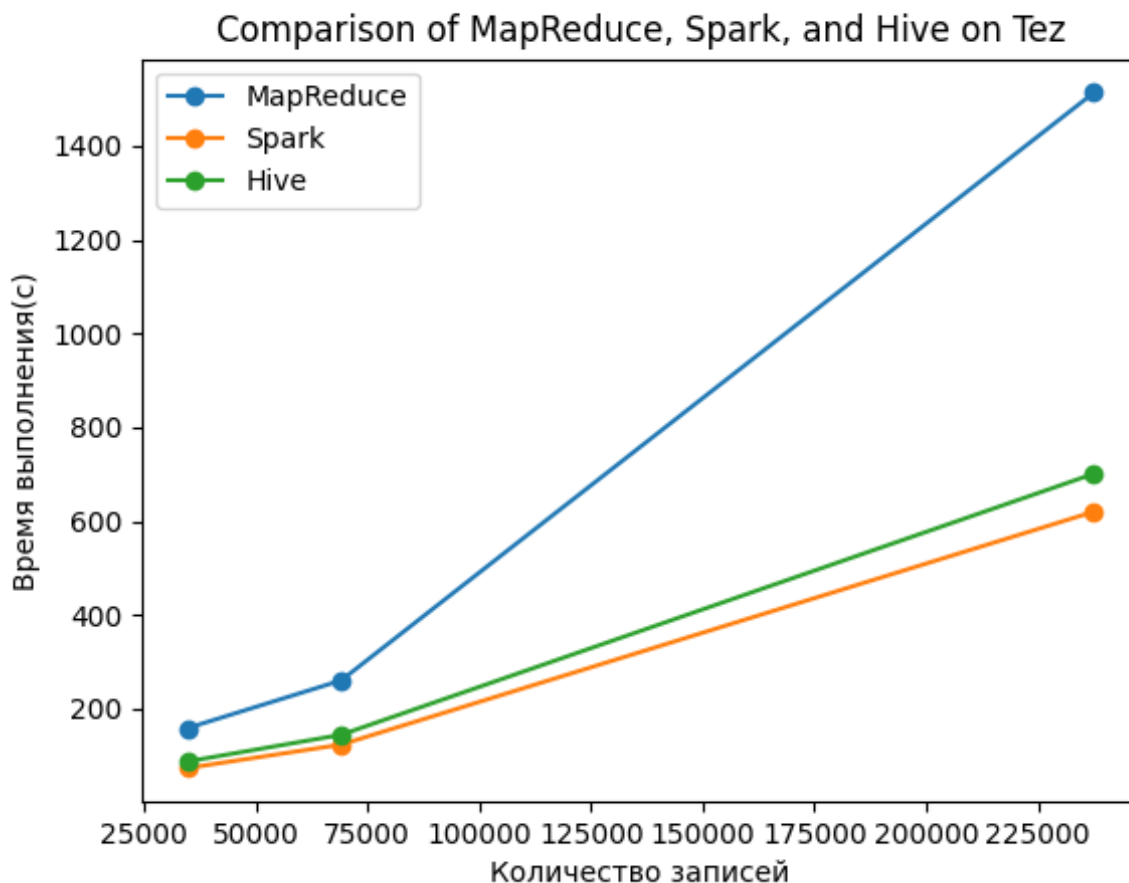


Рис.4.11. График времени выполнения Map Reduce, Spark и Apache Hive задач на разных наборах данных

Время выполнения задачи получилось в целом больше, чем для предыдущей задачи, но характер зависимости времени выполнения от количества записей синонимичен предыдущей задаче. Это вполне объяснимо, так как по сути для этой задачи выполняются по сути те же внутренние операции, что и для предыдущей.

#### 4.5. Выводы

В ходе экспериментов, мы убедились, что MapReduce неэффективен для обработки больших объемов данных из-за своей архитектуры, предполагающей высокие накладные расходы на этапы Map и Reduce. Каждая задача требует записи промежуточных данных на диск, что увеличивает время выполнения. Tez позволяет улучшить производительность Hive за счет DAG (Directed Acyclic Graph) выполнения задач, что уменьшает накладные расходы на обработку данных, это архитектурное отличие приближает производительность Hive on Tez к Spark. Spark является наиболее производительным фреймворком среди рассмотренных

благодаря своей архитектуре, использующей вычисления в памяти. Это позволяет избежать накладных расходов на запись промежуточных данных на диск, что значительно ускоряет обработку данных.

Само ранжирование по времени выполнения во всех задачах получилось одинаковым, однако Spark справился гораздо лучше других фреймворков с задачей фильтрации брани, которая подразумевает лишь итерирование по сообщениям и обработку содержания каждого сообщения, что является весьма необычной задачей в рамках пакетной обработки данных.

В классических для пакетной обработки данных задачах, подразумевающих группировку и ранжирование, таких как в третьем и четвертом экспериментах, Hive on Tez и Spark оказываются очень близки по времени выполнения задач. Добавив к этому удобство написания Hive задач, которые пишутся на HiveQL, Hive on Tez является конкурентоспособным инструментом для пакетной обработки данных. Характер зависимости времени выполнения от объема данных, используемых в эксперименте для Hive on Tez и Spark во всех экспериментах остается одинаковым, хотя и сохраняется небольшое преимущество во времени выполнения у Spark.

Map Reduce же, как можно видеть по результатам эксперимента, хотя и не демонстрирует невероятно низкой производительности (531 секунда в итеративной задаче, 260 секунд в классических), но, к сожалению, его уже нельзя считать конкурентоспособным для пакетной обработки больших данным инструментом.

## ЗАКЛЮЧЕНИЕ

В первой главы работы был проведен сравнительный анализ средств работы с большими данными в веб-приложениях, а именно: средств хранения данных(распределенная файловая система HDFS), средств для пакетных вычислений(Hadoop Map Reduce, Apache Spark, Apache Hive), средств для потоковых вычислений(Spark Streaming, Apache Flink). Были рассмотрены архитектурные особенности каждого инструмента, а также составлены сравнительные таблицы, обобщающие характеристики каждого из инструментов.

В второй главе работы описаны шаги по проектированию системы для проведения экспериментального сравнения средств работы для потоковой и пакетной обработки данных.

Также был выбран источник данных - текстовые сообщения из чатов стриминговой платформы Twitch. Источник данных был предварительно проанализирован на соответствие требованиям к источникам больших данных, в ходе чего выяснилось, что он обладает необходимыми признаками.

Были сформулированы требования к системе. После чего были выбраны вспомогательные средства, предусматривающие тесную интеграцию со сравниваемыми средствами обработки данных. Далее была развернута среда исполнения на базе WSL2, перечислены аппаратные характеристики системы и установлены все необходимые для функционирования компонентов системы языковые пакеты (Java, Python, Scala). Были установлены и сконфигурированы компоненты Apache Hadoop, используемые в ходе экспериментального анализа. Далее были установлены и сконфигурированы(под конфигурацией понимается редактирование системных и программных файлов с целью достижения функционирования с установленным количеством аппаратных ресурсов, а также интеграция с СХД, брокером сообщений и другими компонентами системы) фреймворки, которые будут подвергаться экспериментальному сравнению, а именно Apache Flink и Spark(Spark Streaming) для потоковой обработки данных и Spark, Hive on Tez и Map Reduce для пакетной обработки данных.

Далее были самостоятельно разработаны дополнительные компоненты для проведения экспериментального анализа: сервер модификации нагрузки, приходящей с IRC сокетов Twitch, а также набор средств для конфигурации и отслеживания хода экспериментов для потоковой обработки данных, позволяющий провести запуск эксперимента с основной Windows машины и подвести его итоги, измерив

входную нагрузку на систему в ходе эксперимента и оценив задержку, которую демонстрируют сравниваемые средства для потоковой обработки данных.

После чего, в главе 3 было проведено экспериментальное сравнение средств потоковой обработки больших данных, а именно Apache Flink и Spark Streaming. В ходе решения двух задач при различной входной нагрузке на систему (максимально - 126 сообщений в секунду) было установлено, что Apache Flink демонстрирует значительно меньшую среднюю и пиковые задержки при потоковой обработке данных (4.63 мс при решении задачи обработки и записи данных и 4.11 мс при решении задачи фильтрации нецензурной лексики) в сравнении с Spark Streaming (134.21 мс и 216.63 мс соответственно). Также было установлено, что Flink выполняет операции с более стабильной задержкой, что демонстрируется графиками задержки, собранными в ходе экспериментов и данными о максимальной задержке, которая у Apache Flink достигала 42 мс, а у Spark Streaming 743 мс. При этом нагрузка на оперативную память у обоих фреймворков находилась на одном уровне и не менялась с течением эксперимента, так как оба средства обладают развитыми сборщиками мусора и не хранят уже записанную или обработанную информацию в оперативной памяти. Полученные результаты можно объяснить превосходством архитектуры Apache Flink, которые в отличие от Spark Streaming реализует True Stream Processing (истинную потоковую обработку). Spark Streaming же подразделяет приходящие данные на микро-batchи, которые обрабатываются с фиксированным интервалом времени, поэтому задержка у прибывшего позднее элемента данных будет меньше, чем задержка у прибывавшего раньше, но попавшего в тот же микро-batch.

В главе 4 были приведено экспериментальное сравнение фреймворков для пакетной обработки данных, а именно Spark, Hive on Tez и Hadoop Map Reduce. Предварительно были собраны наборы данных различного объема и в ходе каждого эксперимента измерялось время выполнения задач на собранных наборах данных разного объема. Всего было проведено 4 эксперимента. При любом размере дата-сета наблюдалось превосходство Spark и Hive on Tez над Map Reduce. При этом Spark сохранял первенство по времени выполнения. Особенно ярко выражено это проявилось в эксперименте, решающем задачу фильтрации нецензурных выражений из хранимых сообщений. Это нетипичная задача для пакетной обработки данных, и Spark справился с ней гораздо лучше Hive On Tez. В более типичных для пакетной обработки данных задачах, подразумевающих группировку и сортировку большого количества данных, отставание Hive on Tez от Spark составляло от 10%

до 29.3%, в зависимости от конкретной задачи и объема обрабатываемых данных. Также стоит отметить, что у всех фреймворков присутствовала положительная зависимость времени выполнения от размера набора данных.

На основании результатов экспериментов и исследования архитектуры фреймворков, проведенного в первой главе можно сделать вывод, что как Spark Streaming, так и Apache Flink являются актуальными и достаточно производительными средствами для потоковой обработки данных. Spark Streaming является частью наиболее популярного фреймворка для обработки больших данных, в связи с чем обладает большим количеством интерфейсов для написания задач, чем Apache Flink. Также, в ходе разработки и конфигурации становится очевидным, что Spark Streaming обладает более подробной документацией, нежели Apache Flink. Рассматривая фреймворки исключительно с точки зрения производительности, архитектурные особенности Spark Streaming не позволяют ему конкурировать с Apache Flink на поприще истинной потоковой обработки. Задержка, демонстрируемая Spark Streaming в среднем превышает задержку, демонстрируемую Apache Flink в 36 раз. Поэтому, если для системы очень важно обрабатывать информацию как можно быстрее и задержка - наиболее важная характеристика системы, то выбор разработчика должен склоняться в сторону Apache Flink. Если же допустимы задержки в районе секунды, и конечный выходной источник системы может воспринимать информацию с такой скоростью, либо система нацелена на агрегацию данных и обрабатывает их потоково лишь для изначальной фильтрации или преобразования в формат, удобный для хранения, то Spark Streaming будет более предпочтителен из-за более развитой документации и более широкого набора API для написания работ и процедур.

При выборе инструмента для пакетной обработки данных стоит в первую очередь обратить внимание не на Apache Spark. В ходе экспериментов было выявлено, что он справляется как с классическими для пакетной обработки данных задачами, так и с задачами, подразумевающими итерационную обработку текстовых фрагментов данных быстрее Hive on Tez и значительно лучше Map Reduce. Hive on Tez также является актуальным инструментом для пакетной обработки данных, так как в более классических для пакетной обработки данных задачах он демонстрирует небольшое отставание по скорости выполнения от Spark (~10%), при этом позволяя писать запросы на диалекте знакомого почти всем разработчикам SQL. Map Reduce же, сегодня, является устаревшим инструментом. Он был первым массовым инструментом для пакетной обработки больших данных, и Tez, как и

движок Spark, реализуют технологический подход схожий с Map Reduce, при этом избавляясь от записи промежуточных результатов запросов в хранилище после каждого этапа обработки и значительно оптимизируя его.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Aleksiyan A., Borisenko O., Turdakov D., Sher A., Kuznetsov S. Implementing Apache Spark jobs execution and Apache Spark cluster creation for Openstack Sahara // Труды ИСП РАН. 2015. №5. URL: <https://cyberleninka.ru/article/n/implementing-apache-spark-jobs-execution-and-apache-spark-cluster-creation-for-openstack> (дата обращения: 28.03.2024).
2. Apache Flink Concepts - URL: <https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/try-flink/flink-operations-playground/#flink-webui> (дата обращения: 12.04.2024).
3. Apache Flink Web UI documentation - URL: <https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/try-flink/flink-operations-playground/#flink-webui> (дата обращения: 11.04.2024)
4. Apache Spark RDD programming guide - URL: <https://spark.apache.org/docs/latest/rdd-programming-guide.html> (дата обращения: 11.04.2024).
5. Apache Spark Streaming programming guide - URL: <https://spark.apache.org/docs/latest/streaming-programming-guide.html> (дата обращения: 14.04.2024).
6. Apache Spark Web UI Documentation - URL: <https://spark.apache.org/docs/latest/web-ui.html> (дата обращения: 8.04.2024).
7. Gimaletdinova A. AN IN-DEPTH COMPARATIVE STUDY OF DISTRIBUTED DATA PROCESSING FRAMEWORKS: APACHE SPARK, APACHE FLINK, AND HADOOP MAPREDUCE // Вестник науки. 2024. №4 (73). URL: <https://cyberleninka.ru/article/n/an-in-depth-comparative-study-of-distributed-data-processing-frameworks> (дата обращения: 4.04.2024).
8. Namiot D. On Big Data Stream Processing // International Journal of Open Information Technologies. 2015. №8. URL: <https://cyberleninka.ru/article/n/on-big-data-stream-processing> (дата обращения: 13.04.2024).
9. StreamElements Chat Stats - URL: <https://stats.streamelements.com> (дата обращения: 8.04.2024).
10. Twitch Chat Modes - URL: [https://safety.twitch.tv/s/article/Chat-Tools?language=en\\_US#9ChatModes](https://safety.twitch.tv/s/article/Chat-Tools?language=en_US#9ChatModes) (дата обращения: 12.04.2024).
11. Twitter Blog. Record on tweets per second - URL: [blog.twitter.com/2010/06/big-goals-big-game-big-records.html](https://blog.twitter.com/2010/06/big-goals-big-game-big-records.html) (дата обращения: 8.04.2024).
12. WebHCat documentation - URL: <https://cwiki.apache.org/confluence/display/Hive/WebHCat+UsingWebHCat> (дата обращения: 14.04.2024).



13. Борисенко О. Д., Турдаков Д. Ю., Кузнецов С. Д. Автоматическое создание виртуальных кластеров Apache Spark в облачной среде Openstack // Труды ИСП РАН. 2014. №4. URL: <https://cyberleninka.ru/article/n/avtomaticheskoe-sozdanie-virtualnyh-klasterov-apache-spark-v-oblachnoy-srede-openstack> (дата обращения: 3.04.2024).
14. Белов В.А. Методики анализа форматов хранения и глобально распределенной обработки больших объемов данных. Дисс. на соиск. уч. ст. к.т.н. Диссертационный совет 24.2.326.09. Размещено 19.06.2023. URL: [https://www.mirea.ru/upload/medialibrary/923/krk0nugmcxc6j7mnonmah53kf089lnwe/dissertatsiya\\_belov\\_v1.pdf](https://www.mirea.ru/upload/medialibrary/923/krk0nugmcxc6j7mnonmah53kf089lnwe/dissertatsiya_belov_v1.pdf) (дата обращения 30.04.2024).
15. Белов В. А., Никульчев Е. В. ЭКСПЕРИМЕНТАЛЬНАЯ ОЦЕНКА ВРЕМЕННОЙ ЭФФЕКТИВНОСТИ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ В ЗАДАННЫХ ФОРМАТАХ ХРАНЕНИЯ // International Journal of Open Information Technologies. 2021. №9. URL: <https://cyberleninka.ru/article/n/eksperimentalnaya-otsenka-vremennoy-effektivnosti-obrabotki-bolshih-dannyh-v-zadannykh-formatakh-hraneniya> (дата обращения: 27.03.2024).
16. Вишератин А. А. Семантические технологии больших данных для многомасштабного моделирования в распределенных вычислительных средах, диссертационная работа: дис. - С.-Петерб. нац. исслед. ун-т информац. технологий, механики и оптики, 2017.
17. Горшков Н.А, Денисов В.С. Анализ сообщений социальной сети Twitter с использованием систем обработки потоковых данных Apache Spark и Apache Storm // International Journal of Open Information Technologies. 2016. №11. URL: <https://cyberleninka.ru/article/n/analiz-soobscheniy-sotsialnoy-seti-twitter-s-ispolzovaniem-sistem-obrabotki-potokovykh-dannyh-apache-spark-i-apache-storm> (дата обращения: 2.04.2024).
18. Гусейнов А.А., Бочкова И.А. Исследование распределенной обработки данных на примере системы Hadoop // Актуальные проблемы авиации и космонавтики. 2016. №12. URL: <https://cyberleninka.ru/article/n/issledovanie-raspredelennoy-obrabotki-dannyh-na-primere-sistemy-hadoop> (дата обращения: 8.04.2024).
19. Дзидзава Э.Т., Ахмедов К.М. БОЛЬШИЕ ДАННЫЕ И HADOOP: ОБЗОРНЫЙ ДОКЛАД // Вестник магистратуры. 2021. №1-1 (112). URL: <https://cyberleninka.ru/article/n/bolshie-dannye-i-hadoop-obzornyy-doklad> (дата обращения: 4.04.2024).



20. Документация Apache Spark: конфигурация Spark History Server - <https://spark.apache.org/docs/latest/monitoring.html#spark-history-server-configuration-options> (дата обращения: 12.04.2024).
21. Документация Hadoop HDFS - URL: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> (дата обращения: 8.04.2024).
22. Документация модуля Socket - <https://docs.python.org/3/library/socket.html> (дата обращения: 12.04.2024).
23. Ермагамбетов Р.Т., Киселев Е.С. СОВРЕМЕННЫЕ СИСТЕМЫ ХРАНЕНИЯ И ОБРАБОТКИ БОЛЬШИХ ДАННЫХ: HADOOP И АПАЧЕ SPARK // Форум молодых ученых. 2018. №8 (24). URL: <https://cyberleninka.ru/article/n/sovremennye-sistemy-hraneniya-i-obrabotki-bolshih-dannyh-hadoop-i-apache-spark> (дата обращения: 5.04.2024).
24. Загребав Дмитрий Кириллович Мониторинг кластера анализа больших данных Apache Spark на основе Kubernetes // Достижения науки и образования. 2019. №5 (46). URL: <https://cyberleninka.ru/article/n/monitoring-klastera-analiza-bolshih-dannyh-apache-spark-na-osnove-kubernetes> (дата обращения: 5.04.2024).
25. Манев Дмитрий Валерьевич, Сальников Вячеслав Юрьевич Информационная система обработки и хранения больших объемов измерительных данных // SAEC. 2019. №1. URL: <https://cyberleninka.ru/article/n/informatsionnaya-sistema-obrabotki-i-hraneniya-bolshih-obemov-izmeritelnyh-dannyh> (дата обращения: 5.04.2024).
26. Открытый репозиторий Apache Hadoop - <https://hadoop.apache.org/release/3.4.0.html> (дата обращения: 12.04.2024).
27. Открытый репозиторий Apache Spark - <https://spark.apache.org/downloads.html> (дата обращения: 12.04.2024).
28. Пример подключения к IRC сокету чата Twitch - URL: <https://dev.twitch.tv/docs/irc/#connecting-to-the-twitch-irc-server> (дата обращения: 12.04.2024).
29. Самарев Р.С. Обзор состояния области потоковой обработки данных // Труды ИСП РАН. 2017. №1. URL: <https://cyberleninka.ru/article/n/obzor-sostoyaniya-oblasti-potokovoy-obrabotki-dannyh> (дата обращения: 6.04.2024).
30. Стриминговая платформа Twitch - URL: <https://www.twitch.tv/> (дата обращения: 12.04.2024).

31. Стариков А.Е., Намиот Д.Е. Система выполнения моделей машинного обучения на потоке событий // International Journal of Open Information Technologies. 2020. №7. URL: <https://cyberleninka.ru/article/n/sistema-vypolneniya-modeley-mashinnogo-obucheniya-na-potoke-sobytiy> (дата обращения: 3.04.2024).

## Приложение 1

## Отличные от переменных по умолчанию системные переменные в WSL

```

export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export HADOOP_HOME=/home/hadoop/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
5 export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
10 export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/
    native"
export HIVE_HOME=/home/hadoop/hadoop/apache-hive-4.0.0-bin
export PATH=$PATH:/home/hadoop/hadoop/apache-hive-4.0.0-bin/
    bin
export FLINK_HOME=~/.hadoop/flink
15 export PATH=$PATH:$FLINK_HOME/bin
export HADOOP_CLASSPATH='$HADOOP_HOME/bin/hadoop classpath'
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export LANG=ru_RU.UTF-8
export LANGUAGE=ru_RU:ru
20 export LC_ALL=ru_RU.UTF-8

alias jupyter-notebook="~/local/bin/jupyter-notebook --no-
    browser"

export SPARK_HOME="/home/hadoop/spark-3.5.1-bin-without-hadoop
    "
25 export PATH=$PATH:$SPARK_HOME/bin
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_CLASSPATH=($HADOOP_HOME/bin/hadoop classpath)
export PYSARK_PYTHON=python3
30 export PYSARK_DRIVER_PYTHON=python3
export HADOOP_CLASSPATH=($HADOOP_HOME/bin/hadoop classpath)
export SPARK_CLASSPATH=$SPARK_HOME/jars/jsr305-3.0.0.jar

```

## IRC Socket server

```
2 from kafka import KafkaProducer
3 import socket
4 import threading
5 import time
6 from queue import Queue
7 from flask import Flask, request, jsonify
8
9 app = Flask(__name__)
10
11 class IRCClient:
12     def __init__(self, server, port, nickname, token,
13                 kafka_server, kafka_topic):
14         self.server = server
15         self.port = port
16         self.nickname = nickname
17         self.token = token
18         self.channels = []
19         self.kafka_server = kafka_server
20         self.kafka_topic = kafka_topic
21         self.command_queue = Queue()
22         self.producer = KafkaProducer(bootstrap_servers=[
23             kafka_server])
24         self.stop_event = threading.Event()
25         self.threads = []
26
27         threading.Thread(target=self.process_commands, daemon=
28             True).start()
29
30     def process_commands(self):
31         while True:
32             command, args = self.command_queue.get()
33             if command == "add_channel":
34                 self._add_channel(args)
35             elif command == "clear_channels":
36                 self._clear_channels()
37             self.command_queue.task_done()
38
39     def irc_connection(self, channel):
40         irc_sock = socket.socket()
```

```

38     irc_sock.connect((self.server, self.port))
39     irc_sock.send(f"PASS {self.token}\n".encode('utf-8'))
40     irc_sock.send(f"NICK {self.nickname}\n".encode('utf-8'
41         ))
42     irc_sock.send(f"JOIN {channel}\n".encode('utf-8'))
43     print(f"Connected to IRC channel {channel}")
44
45     while not self.stop_event.is_set():
46         try:
47             resp = irc_sock.recv(2048).decode('utf-8')
48             if resp.startswith('PING'):
49                 irc_sock.send("PONG\n".encode('utf-8'))
50             elif "PRIVMSG" in resp:
51                 arrival_timestamp = str(int(float(time.
52                     time()) * 1000))
53                 formatted_resp = f"{arrival_timestamp} {
54                     resp}"
55                 print(f"Received message: {formatted_resp}
56                     ")
57                 self.producer.send(self.kafka_topic,
58                     formatted_resp.encode('utf-8'))
59             except socket.error:
60                 break
61
62     irc_sock.close()
63     print(f"Disconnected from IRC channel {channel}")
64
65     def _add_channel(self, channel):
66         self.channels.append(channel)
67         thread = threading.Thread(target=self.irc_connection,
68             args=(channel,))
69         thread.daemon = True
70         thread.start()
71         self.threads.append(thread)
72         print(f"Added channel {channel}")
73
74     def _clear_channels(self):
75         self.stop_event.set()
76         for thread in self.threads:
77             thread.join()
78         self.threads = []
79         self.channels = []
80         self.stop_event.clear()

```

```

75         print("Cleared all channels and stopped threads")
76
77     def add_channel(self, channel):
78         self.command_queue.put(("add_channel", channel))
79         return f"Adding channel {channel}"
80
81     def clear_channels(self):
82         self.command_queue.put(("clear_channels", None))
83         return "Clearing all channels"
84
85     irc_client = IRCClient(
86         server="irc.chat.twitch.tv",
87         port=6667,
88         nickname="FrolovGeorgiy",
89         token="oauth:ryh3hgq656pi5c96ki8jjkqfjakh4f",
90         kafka_server="localhost:9092",
91         kafka_topic="irc_messages"
92     )
93
94     @app.route('/add_channel', methods=['POST'])
95     def add_channel():
96         data = request.json
97         channel = data.get('channel')
98         if channel:
99             response = irc_client.add_channel(channel)
100             return jsonify({"message": response}), 200
101         return jsonify({"message": "Channel is required"}), 400
102
103     @app.route('/', methods=['GET'])
104     def root():
105         return jsonify({}), 200
106
107     @app.route('/clear_channels', methods=['POST'])
108     def clear_channels():
109         response = irc_client.clear_channels()
110         return jsonify({"message": response}), 200
111
112 if __name__ == '__main__':
113     app.run(host='0.0.0.0', port=5000)

```

## Приложение 3

# Скрипт для выполнения экспериментов в рамках потоковой обработки данных

```

2 from utils import try_connect, load_config, save_config
3 from connections_checker import check_connections
4 from logs import config_logger, logger,
    critical_log_with_error
5 import argparse
6 import os
7 import time
8 from datetime import datetime
9 from payload_debugging import PayloadDebug
10 from latency_debugging import LatencyAnalyzer
11 import threading
12 from job_runner import run_flink_job
13 from channel_management import add_channel, clear_channels
14
15
16 def main():
17     try:
18         parser = argparse.ArgumentParser(
19             description='Experiment directory')
20         parser.add_argument('experiment_directory_path', type=
21                               str,
22                               help='Path to the experiment
23                                     directory')
24         path = parser.parse_args().experiment_directory_path
25         config = load_config(f"{path}/config.json")
26
27         logger.success(
28             "Successfully got configuration. Initializing
29             current experiment directory!")
30     except Exception as e:
31         critical_log_with_error(
32             "Unable to find path provided, or it doesn't
33             contain config.json, please provide the correct
34             path for the experiment directory!", e)
35     exit()
36
37     try:

```

```

33         current_time = datetime.now().strftime('%Y-%m-%d_%H-%M
           -%S')
34         new_directory_path = os.path.join(path, current_time)
35         os.makedirs(new_directory_path, exist_ok=True)
36
37         old_config_path = os.path.join(new_directory_path, '
           config_old.json')
38         save_config(config, old_config_path)
39
40         config_logger(new_directory_path)
41
42         logger.success(
43             "Successfully initialized current try directory
              and saved old config!")
44     except Exception as e:
45         critical_log_with_error("Can't create current try
              directory!", e)
46         exit()
47
48     REQUIRED_SERVICES = [
49         {"title": "Spark Master", "port": 8080},
50         {"title": "Flink Web UI", "port": 8085},
51         {"title": "IRC Socket Server", "port": 5000},
52     ]
53
54     try:
55         check_connections(REQUIRED_SERVICES)
56     except Exception as e:
57         critical_log_with_error("Required services are not
              available!", e)
58         exit()
59
60     flink_thread = threading.Thread(
61         target=run_flink_job)
62     flink_thread.start()
63
64     time.sleep(30)
65
66     logger.success("Successfully submitted job!")
67
68     clear_channels()
69
70     for streamer in config['streamers']:

```



```

71         add_channel(streamer)
72
73     kafka_bootstrap_servers = "localhost:9092"
74     payload_topic = "irc_messages"
75     latency_topic = "flink_latency"
76     duration = config['experiment_duration']
77
78     payload_debugger = PayloadDebug(
79         topic=payload_topic, server=kafka_bootstrap_servers,
80         output_path=new_directory_path)
81     latency_analyzer = LatencyAnalyzer(
82         kafka_bootstrap_servers=kafka_bootstrap_servers,
83         kafka_topic=latency_topic, output_path=
84         new_directory_path)
85
86     try:
87         payload_debugger_thread = threading.Thread(
88             target=payload_debugger.start)
89         latency_analyzer_thread = threading.Thread(
90             target=latency_analyzer.start)
91
92         payload_debugger_thread.start()
93         latency_analyzer_thread.start()
94
95         time.sleep(duration)
96
97         payload_debugger.stop()
98         latency_analyzer.stop()
99
100        payload_debugger_thread.join()
101        latency_analyzer_thread.join()
102        flink_thread.join()
103    except Exception as e:
104        critical_log_with_error("Error during the experiment!"
105                                , e)
106        exit()
107
108    logger.success("Experiment completed successfully!")
109
110    if __name__ == "__main__":
111        main()

```

## Приложение 4

## Класс для анализа задержки во процессе потоковой обработки данных

```
2
3 import time
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from kafka import KafkaConsumer
7 import os
8
9
10 class LatencyAnalyzer:
11     def __init__(self, kafka_bootstrap_servers, kafka_topic,
12                 output_path):
13         self.kafka_bootstrap_servers = kafka_bootstrap_servers
14         self.kafka_topic = kafka_topic
15         self.output_path = output_path
16         self.consumer = KafkaConsumer(
17             kafka_topic,
18             bootstrap_servers=kafka_bootstrap_servers,
19             auto_offset_reset='latest',
20             group_id="joker_s2",
21             value_deserializer=lambda x: int(x.decode('utf-8'))
22         )
23         self.latencies = []
24
25         os.makedirs(self.output_path, exist_ok=True)
26
27     def start(self):
28         print("Starting latency analyzer...")
29         for message in self.consumer:
30             latency = message.value
31             self.latencies.append(latency)
32             print(f"Received latency: {latency}")
33
34     def stop(self):
35         print("Stopping latency analyzer...")
36         self.consumer.close()
37
38         if not self.latencies:
39             print("No latencies recorded.")
```

```

39         return
40
41         min_latency = np.min(self.latencies)
42         max_latency = np.max(self.latencies)
43         mean_latency = np.mean(self.latencies)
44         std_latency = np.std(self.latencies)
45
46         stats_file_path = os.path.join(
47             self.output_path, 'latency_statistics.txt')
48         with open(stats_file_path, 'w') as f:
49             f.write(f"Min Latency: {min_latency}\n")
50             f.write(f"Max Latency: {max_latency}\n")
51             f.write(f"Mean Latency: {mean_latency}\n")
52             f.write(f"Standard Deviation: {std_latency}\n")
53
54         print(f"Statistics saved to {stats_file_path}")
55
56         step = max(len(self.latencies) // 100, 1)
57         sampled_latencies = self.latencies[::step]
58
59         plt.figure(figsize=(10, 5))
60         plt.plot(sampled_latencies, marker='o', linestyle='-',
61                 color='b')
62         plt.xlabel('Sample Index')
63         plt.ylabel('Latency (ms)')
64         plt.title('Latencies Over Time')
65         plt.grid(True)
66         plt.tight_layout()
67         graph_file_path = os.path.join(self.output_path, '
68             latency_graph.png')
69         plt.savefig(graph_file_path)
70         plt.show()
71
72         print(f"Graph saved to {graph_file_path}")
73
74 # Example usage
75 if __name__ == "__main__":
76     output_path = "./latency_output" # specify your output
77     directory
78     analyzer = LatencyAnalyzer(
79         kafka_bootstrap_servers="localhost:9092", kafka_topic
80         ="flink_latency", output_path=output_path)

```

```
78  
79     try:  
80         analyzer.start()  
81     except KeyboardInterrupt:  
82         analyzer.stop()
```

## Приложение 5

## Класс для анализа нагрузки во время потоковой обработки данных

```
2 import time
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from kafka import KafkaConsumer
6 import os
7
8
9 class PayloadDebug:
10     def __init__(self, topic, server, output_path, interval=10
11         ):
12         self.topic = topic
13         self.server = server
14         self.interval = interval
15         self.output_path = output_path
16         self.consumer = KafkaConsumer(
17             self.topic, bootstrap_servers=self.server,
18             auto_offset_reset='latest')
19         self.message_counts = []
20         self.running = False
21
22         os.makedirs(self.output_path, exist_ok=True)
23
24     def start(self):
25         self.running = True
26         while self.running:
27             interval_start_time = time.time()
28             count = 0
29             interval_end_time = interval_start_time + self.
30                 interval
31             while time.time() < interval_end_time:
32                 msg_pack = self.consumer.poll(timeout_ms=500)
33                 count += sum(len(v) for v in msg_pack.values()
34                     )
35                 self.message_counts.append(count)
36                 print(f"Messages in last {self.interval} seconds:
37                     {count}")
38
39             elapsed_time = time.time() - interval_start_time
40             remaining_time = self.interval - elapsed_time
```

```

36
37         if remaining_time > 0:
38             for _ in range(int(remaining_time)):
39                 if not self.running:
40                     break
41                 time.sleep(1)
42             if self.running:
43                 time.sleep(remaining_time - int(
44                     remaining_time))
45
46 def stop(self):
47     print("Stopping payload debugger...")
48     self.running = False
49     if not self.message_counts:
50         print("No messages recorded.")
51         return
52
53     max_val = max(self.message_counts)
54     min_val = min(self.message_counts)
55     mean_val = sum(self.message_counts) / len(self.
56         message_counts)
57     std_val = np.std(self.message_counts)
58
59     stats_file_path = os.path.join(
60         self.output_path, 'payload_statistics.txt')
61     with open(stats_file_path, 'w') as f:
62         f.write(f"Max messages: {max_val}\n")
63         f.write(f"Min messages: {min_val}\n")
64         f.write(f"Mean messages: {mean_val}\n")
65         f.write(f"Standard Deviation: {std_val}\n")
66
67     print(f"Statistics saved to {stats_file_path}")
68
69     times = [(i+1) * self.interval for i in range(len(self
70         .message_counts))]
71
72     plt.figure(figsize=(10, 6))
73     plt.plot(times, self.message_counts, marker='o')
74     plt.title('Number of Messages Sent to Kafka Topic Over
75         Time')
76     plt.xlabel('Time (seconds)')
77     plt.ylabel('Number of Messages')
78     plt.grid(True)

```

```
75         graph_file_path = os.path.join(  
76             self.output_path, 'payload_message_counts.png')  
77         plt.savefig(graph_file_path)  
78         plt.show()  
79  
80         print(f"Graph saved to {graph_file_path}")
```

## Приложение 6

## Функции, запускающие Spark Steaming и Apache Flink работы извне WSL

```

2 import subprocess
3
4
5 def run_flink_job():
6     try:
7         result = subprocess.run(
8             ['wsl', '-u', 'hadoop', 'bash', '-c', 'source ~/.
              profile && \${FLINK_HOME}/bin/flink run -c
              IRCCClientFlinkJob \${FLINK_HOME}/jobs/
              IRCCClientFlinkJob/target/IRCCClientFlinkJob-1.0-
              SNAPSHOT.jar'], capture_output=True, text=True)
9         if result.returncode == 0:
10             print(result.stdout)
11             return result.stdout
12         else:
13             raise Exception(result.stderr)
14     except Exception as e:
15         print(f"Error running command:\nException: {e}")
16         return None
17
18
19 def run_spark_job():
20     """
21     Run a command in WSL as a specified user and return the
22     output.
23     """
24     try:
25         result = subprocess.run(
26             ['wsl', '-u', 'hadoop', 'bash', '-c', "$PWD"],
27             capture_output=True, text=True)
28         if result.returncode == 0:
29             print(result.stdout)
30             return result.stdout
31         else:
32             raise Exception(result.stderr)
33     except Exception as e:
34         print(f"Error running command: {"$PWD"}\nException: {e
35             }")
36         return None

```





## Приложение 7

## Модуль utils для скрипта, автоматизирующего проведение экспериментов

```
2 import time
3 from logs import logger
4 import socket
5 import requests
6 import json
7
8
9 def submit_func(data):
10     logger.success(f'''Successfully connected to {
11                     data['title']} on port {data['port']}''')
12
13
14 def reject_func(data):
15     logger.critical(f'''Unable to connect to {data['title']}
16                     on port {
17                         data['port']}, make sure the service is
18                         running''')
19
20
21 def unsuccess_func(data):
22     logger.error(f'''Can't connect to {data['title']} on port
23                 {
24                     data['port']}, trying again''')
25
26
27 def load_config(filename):
28     with open(filename, 'r') as file:
29         config = json.load(file)
30     return config
31
32
33 def save_config(config, filename):
34     with open(filename, 'w') as file:
35         json.dump(config, file, indent=4)
36
37
38 def try_connect(n, i, func, args):
39     for i in range(n):
40         res = func(args)
```

```
38         if (res):
39             submit_func(args)
40             return
41         else:
42             unsuccess_func(args)
43             time.sleep(i)
44
45     reject_func(args)
46     raise Exception(f"Unable to connect to {args['name']}")
47
48
49 def is_http_service_running(args):
50     url = f"http://localhost:{args['port']}"
51     try:
52         response = requests.get(url, timeout=2)
53         if response.status_code == 200:
54             return True
55         else:
56             return False
57     except requests.RequestException:
58         return False
```

## Модуль-логгер

```
2 from loguru import logger
3
4
5 def config_logger(path):
6     logger.add(f"{path}/file.log",
7               format="{time} {level} {message}", level="INFO",
8               ")
9
10 def critical_log_with_error(message, error):
11     logger.critical(
12         f"{message} \n Error is: {error}")
```

## Приложение 9

**Apache Flink Job, выполняющая чтение сообщений из Kafka Producer,  
инициированного сервером, слушающим IRC сокеты и сохраняющая записи  
в HDFS в формате csv**

```

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.
    SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
5 import org.apache.flink.streaming.api.environment.
    StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.
    RichSinkFunction;
import org.apache.flink.streaming.connectors.kafka.
    FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.
    FlinkKafkaProducer;
import org.apache.hadoop.conf.Configuration;
10 import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.KafkaFuture;
15
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
import java.io.IOException;
import java.io.Serializable;
20 import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class IRCClientFlinkJob {
25     public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment
                ();

        String kafkaBootstrapServers = "localhost:9092";
        String kafkaTopic = "irc_messages";
30     String latencyTopic = "flink_latency";

```

```

        createKafkaTopic(kafkaBootstrapServers, latencyTopic);

        Properties properties = new Properties();
35 properties.setProperty("bootstrap.servers",
            kafkaBootstrapServers);
        properties.setProperty("group.id", "flink-group");
        properties.setProperty("auto.offset.reset", "latest");

        FlinkKafkaConsumer<String> kafkaConsumer = new
            FlinkKafkaConsumer<>(kafkaTopic, new
                SimpleStringSchema(), properties);
40 FlinkKafkaProducer<String> kafkaProducer = new
            FlinkKafkaProducer<>(kafkaBootstrapServers,
                latencyTopic, new SimpleStringSchema());

        DataStream<String> text = env.addSource(kafkaConsumer)
            ;

        DataStream<Message> messages = text.map(new
            MapFunction<String, Message>() {
45         @Override
            public Message map(String value) {
                try {
                    // Assuming the message format is "
                        arrival_timestamp :user!user@user.tmi.
                        twitch.tv PRIVMSG #streamer :msg"
                    String[] parts = value.split(" ");
50 if (parts.length < 5) {
                        throw new IllegalArgumentException("
                            Unexpected message format: " +
                                value);
                    }

                    long arrivalTimestamp = Long.parseLong(
                        parts[0]);
55 String user = parts[1].split("!")[0].
                        substring(1);
                    String streamer = parts[3].substring(1);
                    String msg = value.split(":", 3)[2];

                    long processingTimestamp = System.
                        currentTimeMillis();
60

```

```

        Message message = new Message(
            arrivalTimestamp, user, streamer, msg,
            processingTimestamp);
        System.out.println("Processed message: " +
            message);
        return message;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

}).filter(message -> message != null); // Filter out
null messages

messages.addSink(new HDFSSink());

messages.map(new MapFunction<Message, String>() {
    @Override
    public String map(Message message) {
        long latency = message.getProcessingTimestamp
            () - message.getArrivalTimestamp();
        return Long.toString(latency);
    }
}).addSink(kafkaProducer);

env.execute("IRC Client Flink Job");
}

public static class Message implements Serializable {
    private long arrivalTimestamp;
    private String user;
    private String streamer;
    private String msg;
    private long processingTimestamp;

    public Message() {
    }

    public Message(long arrivalTimestamp, String user,
        String streamer, String msg, long
        processingTimestamp) {
        this.arrivalTimestamp = arrivalTimestamp;
        this.user = user;
        this.streamer = streamer;
        this.msg = msg;
    }
}

```

```
        this.processingTimestamp = processingTimestamp;
100    }

    public long getArrivalTimestamp() {
        return arrivalTimestamp;
    }
105

    public void setArrivalTimestamp(long arrivalTimestamp)
    {
        this.arrivalTimestamp = arrivalTimestamp;
    }

110    public String getUser() {
        return user;
    }

    public void setUser(String user) {
115        this.user = user;
    }

    public String getStreamer() {
        return streamer;
120    }

    public void setStreamer(String streamer) {
        this.streamer = streamer;
    }
125

    public String getMsg() {
        return msg;
    }

130    public void setMsg(String msg) {
        this.msg = msg;
    }

    public long getProcessingTimestamp() {
135        return processingTimestamp;
    }

    public void setProcessingTimestamp(long
        processingTimestamp) {
        this.processingTimestamp = processingTimestamp;
140    }
```



```

@Override
public String toString() {
    return arrivalTimestamp + "," + user + "," +
        streamer + "," + msg + "," +
        processingTimestamp;
145     }
}

public static class HDFSSink extends RichSinkFunction<
    Message> {
150     private transient FileSystem fs;

    @Override
    public void open(org.apache.flink.configuration.
        Configuration parameters) throws Exception {
        super.open(parameters);
        Configuration hadoopConf = new Configuration();
155     hadoopConf.set("fs.defaultFS", "hdfs://localhost
        :9000");
        fs = FileSystem.get(hadoopConf);
    }

    @Override
160     public void invoke(Message value, Context context)
        throws Exception {
        String filePath = "/user/hadoop/flink-data/" +
            System.currentTimeMillis() + ".csv";
        org.apache.hadoop.fs.Path path = new org.apache.
            hadoop.fs.Path(filePath);

        try (FSDataOutputStream outputStream = fs.create(
            path, true);
165             BufferedWriter writer = new BufferedWriter(
                new OutputStreamWriter(outputStream))) {
            writer.write(value.toString());
            writer.newLine();
        } catch (IOException e) {
            e.printStackTrace();
170        }
    }
}

private static void createKafkaTopic(String
    bootstrapServers, String topicName) {
175     Properties properties = new Properties();

```

```
properties.put("bootstrap.servers", bootstrapServers);
properties.put("client.id", "admin-client");

try (AdminClient adminClient = AdminClient.create(
    properties)) {
    180     NewTopic newTopic = new NewTopic(topicName, 1, (
        short) 1);
        KafkaFuture<Void> future = adminClient.
            createTopics(Collections.singletonList(newTopic
            )).all();
        future.get();
        System.out.println("Topic " + topicName + "
            created successfully.");
    } catch (InterruptedException | ExecutionException e)
    185     {
        e.printStackTrace();
    }
}
}
```

## Приложение 10

## Файл зависимостей(pom.xml) для Apache Flink Job

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
    //maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>IRCCClientFlinkJob</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka_2.12</artifactId>
      <version>1.14.4</version>
    </dependency>

    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>3.4.1</version>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java_2.12</artifactId>
      <version>1.12.2</version>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients_2.12</artifactId>
      <version>1.12.0</version>
    </dependency>

    <dependency>
      <groupId>org.apache.flink</groupId>

```

```

    <artifactId>flink-connector-filesystem_2.12</artifactId>
    <version>1.11.2</version>
40 </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table-api-java-bridge_2.12</artifactId>
        >
        <version>1.14.5</version>
45 </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-core</artifactId>
        <version>1.15.0</version>
50 </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-java</artifactId>
        <version>1.15.0</version>
55 <exclusions>
        <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
        </exclusion>
60 <exclusion>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </exclusion>
        <exclusion>
65 <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
        </exclusion>
    </exclusions>
    </dependency>
70 <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-avro</artifactId>
        <version>1.15.0</version>
    </dependency>
75 <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-parquet</artifactId>
        <version>1.15.0</version>
    </dependency>
80 <dependency>
        <groupId>org.apache.parquet</groupId>

```

```

        <artifactId>parquet-avro</artifactId>
        <version>1.12.0</version>
    </dependency>
85 <dependency>
        <groupId>org.apache.parquet</groupId>
        <artifactId>parquet-hadoop</artifactId>
        <version>1.12.0</version>
    </dependency>
90 <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>3.4.0</version>
    </dependency>
95 <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>3.4.0</version>
    </dependency>
100 <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.12.7</version>
    </dependency>
105 <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.7</version>
    </dependency>
110 <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.12.7</version>
    </dependency>
115 <dependency>
        <groupId>org.apache.parquet</groupId>
        <artifactId>parquet-hadoop</artifactId>
        <version>1.11.0</version>
    </dependency>
120 </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
125 <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
            </plugin>
        </plugins>
    </build>

```

```

130     <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
135                 <transformer implementation="org.apache.maven.
                    plugins.shade.resource.
                    ManifestResourceTransformer">
                    <mainClass>IRCCClientFlinkJob</mainClass>
                </transformer>
            </transformers>
            <filters>
140                 <filter>
                    <artifact>*:*</artifact>
                    <excludes>
                        <exclude>META-INF/*.SF</
                            exclude>
                        <exclude>META-INF/*.DSA</
                            exclude>
145                        <exclude>META-INF/*.RSA</
                            exclude>
                    </excludes>
                </filter>
            </filters>
        </configuration>
    </execution>
150 </executions>
</plugin>
</plugins>
</build>
155 </project>

```

## Приложение 11

**Spark Streamig Job, выполняющая чтение сообщений из Kafka Producer, инициированного сервером, слушающим IRC сокеты и сохраняющая записи в HDFS в формате csv**

```

2 from pyspark.sql import SparkSession
3 from pyspark.sql.functions import split, col, expr, substring,
  current_timestamp, unix_millis
4 from kafka import KafkaProducer
5
6 spark = SparkSession.builder \
7     .appName("KafkaToHDFS") \
8     .config("spark.jars.packages", "org.apache.spark:spark-sql
  -kafka-0-10_2.12:3.5.1") \
9     .master("spark://LAPTOP-AM1MP0U8:7077") \
10    .config("spark.executor.memory", "2g") \
11    .config("spark.driver.memory", "2g") \
12    .config("spark.executor.cores", "2") \
13    .config("spark.driver.cores", "1") \
14    .config("spark.hadoop.fs.defaultFS", "hdfs://localhost:900
  0") \
15    .getOrCreate()
16
17 kafka_bootstrap_servers = "localhost:9092"
18 kafka_topic = "irc_messages"
19 kafka_metrics_topic = "spark_latency_data"
20
21 producer = KafkaProducer(
22     bootstrap_servers=kafka_bootstrap_servers,
23     value_serializer=lambda v: str(v).encode('utf-8')
24 )
25
26 lines = spark.readStream \
27     .format("kafka") \
28     .option("kafka.bootstrap.servers", kafka_bootstrap_servers
  ) \
29     .option("subscribe", kafka_topic) \
30     .option("startingOffsets", "latest") \
31     .option("auto.offset.reset", "latest") \
32     .option("failOnDataLoss", "false") \
33     .load()

```

```

34
35 messages = lines.selectExpr("CAST(value AS STRING) as value")
    \
36     .select(split(col("value"), " ", 5).alias("parts")) \
37     .filter(expr("size(parts) > 3")) \
38     .selectExpr(
39         "parts[0] as arrival_timestamp",
40         "split(parts[1], '!')[0] as raw_username",
41         "parts[3] as channel",
42         "parts[4] as message"
43     ) \
44     .selectExpr(
45         "CAST(arrival_timestamp AS LONG) as arrival_timestamp"
46         ,
47         "substring(raw_username, 2, length(raw_username) - 1)
48         as user",
49         "substring(channel, 2, length(channel) - 1) as
50         streamer",
51         "substring(message, 2, length(message) - 1) as msg"
52     ) \
53     .withColumn("processing_timestamp", (unix_millis(
54         current_timestamp()))).cast("long"))
55
56 def send_to_kafka(df, epoch_id):
57     for row in df.collect():
58         latency = row['processing_timestamp'] - row['
59             arrival_timestamp']
60         producer.send(kafka_metrics_topic, latency)
61     producer.flush()
62
63 query = messages.writeStream \
64     .outputMode("append") \
65     .format("csv") \
66     .option("path", "/user/hadoop/data") \
67     .option("checkpointLocation", "/user/hadoop/checkpoints")
68     \
69     .foreachBatch(send_to_kafka) \
70     .start()
71
72 query.awaitTermination()

```



## Приложение 12

## Конфигурация первого эксперимента по потоковой обработке данных

```
2 //
3 {
4     "experiment_duration": 100,
5     "streamers": [
6         "tarik"
7     ],
8     "flink_job_name": "IRCClientFlinkJob",
9     "spark_job_name": "main"
10 }
11 //
12 {
13     "experiment_duration": 100,
14     "streamers": [
15         "tarik",
16         "iltw1",
17         "leva2k"
18     ],
19     "flink_job_name": "IRCClientFlinkJob",
20     "spark_job_name": "main"
21 }
22 //
23 {
24     "experiment_duration": 100,
25     "streamers": [
26         "tarik",
27         "valorant",
28         "summit1g",
29         "leva2k",
30         "limitlessqt",
31         "resolut1ontv",
32         "vovapain",
33         "9class",
34         "r0xieee"
35     ],
36     "flink_job_name": "IRCClientFlinkJob",
37     "spark_job_name": "main"
38 }
```

## Приложение 13

**Flink Job для фильтрации нецензурной лексики из сообщений**

```

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.
    SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
5 import org.apache.flink.streaming.api.environment.
    StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.
    RichSinkFunction;
import org.apache.flink.streaming.connectors.kafka.
    FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.
    FlinkKafkaProducer;
import org.apache.kafka.clients.admin.AdminClient;
10 import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.KafkaFuture;

import java.io.BufferedReader;
import java.io.BufferedWriter;
15 import java.io.FileReader;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Serializable;
import java.util.Collections;
20 import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutionException;
import java.util.stream.Collectors;

25 public class CensorCurseWords {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment
                ();

        String kafkaBootstrapServers = "localhost:9092";
30 String kafkaTopic = "irc_messages2";
String filteredTopic = "filtered_messages";
String latencyTopic = "flink_latency";

    // Create Kafka topics

```

```

35     createKafkaTopic(kafkaBootstrapServers, filteredTopic)
        ;
        createKafkaTopic(kafkaBootstrapServers, latencyTopic);

        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers",
40             kafkaBootstrapServers);
        properties.setProperty("group.id", "flink-group");

        FlinkKafkaConsumer<String> kafkaConsumer = new
            FlinkKafkaConsumer<>(kafkaTopic, new
                SimpleStringSchema(), properties);
        FlinkKafkaProducer<String> filteredKafkaProducer = new
            FlinkKafkaProducer<>(kafkaBootstrapServers,
                filteredTopic, new SimpleStringSchema());
        FlinkKafkaProducer<String> latencyKafkaProducer = new
            FlinkKafkaProducer<>(kafkaBootstrapServers,
45             latencyTopic, new SimpleStringSchema());

        DataStream<String> text = env.addSource(kafkaConsumer)
            ;

        List<String> curseWords = loadCurseWords("curse_words.
            txt");

50        DataStream<Message> messages = text.map(new
            MapFunction<String, Message>() {
                @Override
                public Message map(String value) {
                    try {
                        // Assuming the message format is "
                            arrival_timestamp :user!user@user.tmi.
                            twitch.tv PRIVMSG #streamer :msg"
55                        String[] parts = value.split(" ");
                        if (parts.length < 5) {
                            throw new IllegalArgumentException("
                                Unexpected message format: " +
                                    value);
                        }
                    }

60                    long arrivalTimestamp = Long.parseLong(
                        parts[0]);
                    String user = parts[1].split("!")[0].
                        substring(1);

```

```

        String msg = value.split(":", 3)[2].
            replace("\n", "").replace("\r", "").
            trim();

        // Censor the message
65      msg = censorText(msg, curseWords);

        long processingTimestamp = System.
            currentTimeMillis();

        return new Message(arrivalTimestamp, user,
            msg, processingTimestamp);
70      } catch (Exception e) {
        e.printStackTrace();
        return null;
      }
    }
75  }).filter(message -> message != null); // Filter out
    null messages

    // Send filtered messages to Kafka
    messages.map(new MapFunction<Message, String>() {
        @Override
80      public String map(Message message) {
        return message.getUser() + "::-" + message.
            getMsg();
      }
    }).addSink(filteredKafkaProducer);

    // Calculate latency and send to Kafka
85  messages.map(new MapFunction<Message, String>() {
        @Override
        public String map(Message message) {
            long latency = message.getProcessingTimestamp
                () - message.getArrivalTimestamp();
90            return Long.toString(latency);
        }
    }).addSink(latencyKafkaProducer);

    env.execute("IRC Client Flink Job");
95  }

    public static class Message implements Serializable {
        private long arrivalTimestamp;
        private String user;
    }

```

```
100     private String msg;
        private long processingTimestamp;

        public Message() {
105     }

        public Message(long arrivalTimestamp, String user,
            String msg, long processingTimestamp) {
            this.arrivalTimestamp = arrivalTimestamp;
            this.user = user;
            this.msg = msg;
110         this.processingTimestamp = processingTimestamp;
        }

        public long getArrivalTimestamp() {
            return arrivalTimestamp;
115     }

        public void setArrivalTimestamp(long arrivalTimestamp)
        {
            this.arrivalTimestamp = arrivalTimestamp;
        }

120     public String getUser() {
            return user;
        }

        public void setUser(String user) {
125         this.user = user;
        }

        public String getMsg() {
130         return msg;
        }

        public void setMsg(String msg) {
            this.msg = msg.strip().trim();
135     }

        public long getProcessingTimestamp() {
            return processingTimestamp;
        }

140     public void setProcessingTimestamp(long
        processingTimestamp) {
```

```

        this.processingTimestamp = processingTimestamp;
    }

145    @Override
    public String toString() {
        return arrivalTimestamp + "$" + user + "$" + msg +
            "$" + processingTimestamp;
    }
}

150    private static String censorText(String text, List<String>
        curseWords) {
        for (String word : curseWords) {
            text = text.replaceAll("(?i)" + word, "*".repeat(
                word.length()));
        }
155    return text;
}

    private static List<String> loadCurseWords(String filePath
    ) {
        try (java.io.BufferedReader br = new java.io.
            BufferedReader(new java.io.FileReader(filePath))) {
160            return br.lines().collect(Collectors.toList());
        } catch (IOException e) {
            e.printStackTrace();
            return Collections.emptyList();
        }
165    }

    private static void createKafkaTopic(String
        bootstrapServers, String topicName) {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", bootstrapServers);
170        properties.put("client.id", "admin-client");

        try (AdminClient adminClient = AdminClient.create(
            properties)) {
            NewTopic newTopic = new NewTopic(topicName, 1, (
                short) 1);
            KafkaFuture<Void> future = adminClient.
                createTopics(Collections.singletonList(newTopic
                )).all();
175            future.get(); // Ensure topic creation is
                           complete

```

```
        System.out.println("Topic " + topicName + "  
        created successfully.");  
    } catch (InterruptedException | ExecutionException e)  
    {  
        e.printStackTrace();  
    }  
    }  
}
```

## Приложение 14

## Spark Job для фильтрации нецензурной лексики из сообщений

```

2 from pyspark.sql import SparkSession
3 from pyspark.sql.functions import split, col, expr,
    current_timestamp, unix_millis, regexp_replace, udf, trim
4 from pyspark.sql.types import StringType, LongType
5 from kafka import KafkaProducer
6
7 spark = SparkSession.builder \
8     .appName("FilterCurseWords") \
9     .config("spark.jars.packages", "org.apache.spark:spark-sql
    -kafka-0-10_2.12:3.5.1") \
10     .getOrCreate()
11
12 kafka_bootstrap_servers = "localhost:9092"
13 kafka_topic = "irc_messages2"
14 filtered_topic = "filtered_messages"
15 latency_topic = "spark_latency"
16
17 producer = KafkaProducer(
18     bootstrap_servers=kafka_bootstrap_servers,
19     value_serializer=lambda v: v if isinstance(v, bytes) else
        v.encode('utf-8')
20 )
21
22 with open("curse_words.txt", "r", encoding='utf-8') as file:
23     curse_words = file.read().splitlines()
24
25 def censor_text(text, curse_words):
26     for word in curse_words:
27         text = text.replace(word, '*' * len(word))
28     return text
29
30 censor_udf = udf(lambda text: censor_text(text, curse_words),
    StringType())
31
32 lines = spark.readStream \
33     .format("kafka") \
34     .option("kafka.bootstrap.servers", kafka_bootstrap_servers
        ) \
35     .option("subscribe", kafka_topic) \

```



```

36     .option("startingOffsets", "latest") \
37     .option("failOnDataLoss", "false") \
38     .load()
39
40 messages = lines.selectExpr("CAST(value AS STRING) as value")
41     \
42     .select(split(col("value"), " ", 5).alias("parts")) \
43     .selectExpr(
44         "parts[0] as arrival_timestamp",
45         "split(parts[1], '!')[0] as raw_username",
46         "parts[4] as message"
47     ) \
48     .selectExpr(
49         "CAST(arrival_timestamp AS LONG) as arrival_timestamp"
50         ,
51         "substring(raw_username, 2, length(raw_username) - 1)
52         as user",
53         "substring(message, 2, length(message) - 1) as msg"
54     ) \
55     .withColumn("msg", regexp_replace(col("msg"), "\n|\r", ""))
56     \
57     .withColumn("msg", censor_udf(col("msg"))) \
58     .withColumn("msg", trim(col("msg"))) \
59     .withColumn("filtered_msg", censor_udf(col("msg"))) \
60     .withColumn("processing_timestamp", (unix_millis(
61         current_timestamp()).cast(LongType()))
62
63 def send_to_kafka(df, epoch_id):
64     for row in df.collect():
65         filtered_message = f"{row['user']}:{row['filtered_msg']}
66         ]}"
67         latency = row['processing_timestamp'] - row['
68         arrival_timestamp']
69         if '*' in filtered_message:
70             producer.send(filtered_topic, filtered_message.
71                 encode('utf-8'))
72             producer.send(latency_topic, str(latency).encode('utf-
73                 8'))
74     producer.flush()
75
76 query = messages.writeStream \
77     .outputMode("append") \
78     .foreachBatch(send_to_kafka) \

```

```
70         .start()  
71  
72 query.awaitTermination()
```

## Приложение 15

## Конфигурация второго эксперимента по потоковой обработке данных

```
2 //
3 {
4     "experiment_duration": 100,
5     "streamers": [
6         "tarik"
7     ],
8     "flink_job_name": "CensorCurseWords",
9     "spark_job_name": "curse"
10 }
11 //
12 {
13     "experiment_duration": 100,
14     "streamers": [
15         "tarik",
16         "iltw1",
17         "leva2k"
18     ],
19     "flink_job_name": "CensorCurseWords",
20     "spark_job_name": "curse"
21 }
22 //
23 {
24     "experiment_duration": 100,
25     "streamers": [
26         "tarik",
27         "valorant",
28         "summit1g",
29         "leva2k",
30         "limitlessqt",
31         "resolut1ontv",
32         "vovapain",
33         "9class",
34         "r0xieee"
35     ],
36     "flink_job_name": "CensorCurseWords",
37     "spark_job_name": "curse"
38 }
```

## Приложение 16

## Работы для второго эксперимента по пакетной обработке данных

```

2 //Spark
3 from pyspark.sql import SparkSession
4 from pyspark.sql.functions import udf
5 from pyspark.sql.types import StructType, StringType,
    StructField
6 import re
7 import os
8
9 def censor_text(text, curse_words):
10     if text is None:
11         return text
12     for word in curse_words:
13         regex_pattern = re.compile(re.escape(word), re.
            IGNORECASE)
14         text = regex_pattern.sub('*' * len(word), text)
15     return text
16
17 def main():
18     spark = SparkSession.builder.appName("Censor Curse Words")
        .getOrCreate()
19
20     current_dir = os.path.dirname(os.path.abspath(__file__))
21     curse_words_path = os.path.join(current_dir, "curse_words.
        txt")
22
23     with open(curse_words_path, 'r') as file:
24         curse_words = [line.strip() for line in file.readlines
            ()]
25
26     broadcast_curse_words = spark.sparkContext.broadcast(
        curse_words)
27
28     censor_udf = udf(lambda text: censor_text(text,
        broadcast_curse_words.value), StringType())
29
30     input_path = "hdfs://localhost:9000/user/hadoop/medium"
31
32     schema = StructType([
33         StructField("arrival_timestamp", StringType(), True),

```

```

34         StructField("user", StringType(), True),
35         StructField("streamer", StringType(), True),
36         StructField("msg", StringType(), True),
37         StructField("processing_timestamp", StringType(), True
38             )
39     ])
40     df = spark.read.format("csv").option("header", "false").
41         option("delimiter", "$").schema(schema).load(input_path
42             )
43
44     censored_df = df.withColumn("msg", censor_udf(df.msg))
45
46     censored_df.show(truncate=False)
47
48     spark.stop()
49
50 if __name__ == "__main__":
51     main()
52
53 //Hive
54 ADD FILE /hive/functions/censor_text_udf.py;
55 CREATE TEMPORARY FUNCTION censor_text AS 'censor_text_udf'
56 USING 'curse_words.txt';
57
58 INSERT OVERWRITE TABLE processed_data
59 SELECT
60     arrival_timestamp,
61     user,
62     streamer,
63     censor_text(msg, 'curse_words.txt') AS msg,
64     processing_timestamp
65 FROM raw_data;
66
67 //Map Reduce
68
69 import org.apache.hadoop.conf.Configuration;
70 import org.apache.hadoop.fs.Path;
71 import org.apache.hadoop.io.LongWritable;
72 import org.apache.hadoop.io.Text;
73 import org.apache.hadoop.mapreduce.Job;
74 import org.apache.hadoop.mapreduce.Mapper;
75 import org.apache.hadoop.mapreduce.Reducer;

```

```

73 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
74 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
    ;
75
76 import java.io.BufferedReader;
77 import java.io.FileReader;
78 import java.io.IOException;
79 import java.util.HashSet;
80 import java.util.Set;
81 import java.util.regex.Matcher;
82 import java.util.regex.Pattern;
83
84 public class CensorWordsMapper extends Mapper<LongWritable,
    Text, LongWritable, Text> {
85     private Set<String> curseWords = new HashSet<>();
86
87     @Override
88     protected void setup(Context context) throws IOException,
        InterruptedException {
89         Path[] paths = context.getLocalCacheFiles();
90         for (Path path : paths) {
91             BufferedReader reader = new BufferedReader(new
                FileReader(path.toString()));
92             String line;
93             while ((line = reader.readLine()) != null) {
94                 curseWords.add(line.trim().toLowerCase());
95             }
96             reader.close();
97         }
98     }
99
100    @Override
101    protected void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
102        String[] fields = value.toString().split("\\$");
103        if (fields.length == 5) {
104            String msg = fields[3];
105            String censoredMsg = censorText(msg);
106            fields[3] = censoredMsg;
107            context.write(key, new Text(String.join("$",
                fields)));
108        }
109    }

```

```

110
111     private String censorText(String text) {
112         if (text == null) {
113             return null;
114         }
115         for (String word : curseWords) {
116             Pattern pattern = Pattern.compile(Pattern.quote(
117                 word), Pattern.CASE_INSENSITIVE);
118             Matcher matcher = pattern.matcher(text);
119             text = matcher.replaceAll(repeat('*', word.length
120                 (())));
121         }
122         return text;
123     }
124
125     private String repeat(char c, int times) {
126         char[] chars = new char[times];
127         for (int i = 0; i < times; i++) {
128             chars[i] = c;
129         }
130         return new String(chars);
131     }
132 }
133
134 public class CensorWordsReducer extends Reducer<LongWritable,
135     Text, LongWritable, Text> {
136     @Override
137     protected void reduce(LongWritable key, Iterable<Text>
138         values, Context context) throws IOException,
139         InterruptedException {
140         for (Text value : values) {
141             context.write(key, value);
142         }
143     }
144 }
145
146 public class CensorWordsJob {
147     public static void main(String[] args) throws Exception {
148         if (args.length != 3) {
149             System.err.println("Usage: CensorWordsJob <input
150                 path> <output path> <curse words file>");
151             System.exit(-1);
152         }
153     }
154 }

```

```
147
148     Configuration conf = new Configuration();
149     Job job = Job.getInstance(conf, "Censor Words Job");
150     job.setJarByClass(CensorWordsJob.class);
151     job.setMapperClass(CensorWordsMapper.class);
152     job.setReducerClass(CensorWordsReducer.class);
153     job.setOutputKeyClass(LongWritable.class);
154     job.setOutputValueClass(Text.class);
155
156     FileInputFormat.addInputPath(job, new Path(args[0]));
157     FileOutputFormat.setOutputPath(job, new Path(args[1]))
158         ;
159
160     // Add curse words file to the distributed cache
161     job.addCacheFile(new Path(args[2]).toUri());
162
163     System.exit(job.waitForCompletion(true) ? 0 : 1);
164 }
```



## Приложение 17

## Работы для третьего эксперимента по пакетной обработке данных

```

2 //Spark
3 from pyspark.sql import SparkSession
4 from pyspark.sql.functions import col
5 from pyspark.sql.types import StructType, StringType,
  StructField
6
7 def main():
8     spark = SparkSession.builder.appName("Top Ten Streamers").
  getOrCreate()
9
10    input_path = "hdfs://localhost:9000/user/hadoop/large"
11    schema = StructType([
12        StructField("arrival_timestamp", StringType(), True),
13        StructField("user", StringType(), True),
14        StructField("streamer", StringType(), True),
15        StructField("msg", StringType(), True),
16        StructField("processing_timestamp", StringType(), True
17    ])
18
19    df = spark.read.format("csv").option("delimiter", "$").
  option("header", "false").schema(schema).load(
  input_path)
20
21    streamer_counts = df.groupBy("streamer").count()
22    top_ten_streamers = streamer_counts.orderBy(col("count").
  desc()).limit(10)
23
24    top_ten_streamers.show(truncate=False)
25    spark.stop()
26
27 if __name__ == "__main__":
28     main()
29
30 //Hive
31 SELECT streamer, COUNT(*) AS message_count
32 FROM large
33 GROUP BY streamer
34 ORDER BY message_count DESC;

```

```

35
36 //Map Reduce
37
38 import org.apache.hadoop.conf.Configuration;
39 import org.apache.hadoop.fs.Path;
40 import org.apache.hadoop.io.IntWritable;
41 import org.apache.hadoop.io.LongWritable;
42 import org.apache.hadoop.io.Text;
43 import org.apache.hadoop.mapreduce.Job;
44 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
45 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
    ;
46
47 public class TopTenStreamersJob {
48     public static void main(String[] args) throws Exception {
49         if (args.length != 3) {
50             System.err.println("Usage: TopTenStreamersJob <
                input path> <intermediate path> <output path>")
                ;
51             System.exit(-1);
52         }
53
54         // First job: count the occurrences of each streamer
55         Configuration conf1 = new Configuration();
56         Job job1 = Job.getInstance(conf1, "Count Streamers");
57         job1.setJarByClass(TopTenStreamersJob.class);
58         job1.setMapperClass(StreamerMapper.class);
59         job1.setCombinerClass(StreamerReducer.class);
60         job1.setReducerClass(StreamerReducer.class);
61         job1.setOutputKeyClass(Text.class);
62         job1.setOutputValueClass(IntWritable.class);
63
64         FileInputFormat.addInputPath(job1, new Path(args[0]));
65         FileOutputFormat.setOutputPath(job1, new Path(args[1])
            );
66
67         if (!job1.waitForCompletion(true)) {
68             System.exit(1);
69         }
70
71         // Second job: sort the streamers by count and get the
            top ten
72         Configuration conf2 = new Configuration();

```

```

73         Job job2 = Job.getInstance(conf2, "Top Ten Streamers")
74         ;
75         job2.setJarByClass(TopTenStreamersJob.class);
76         job2.setMapperClass(SortMapper.class);
77         job2.setReducerClass(TopTenReducer.class);
78         job2.setSortComparatorClass(IntWritable.
79             DecreasingComparator.class);
80         job2.setOutputKeyClass(IntWritable.class);
81         job2.setOutputValueClass(Text.class);
82         FileInputFormat.addInputPath(job2, new Path(args[1]));
83         FileOutputFormat.setOutputPath(job2, new Path(args[2])
84             );
85         System.exit(job2.waitForCompletion(true) ? 0 : 1);
86     }
87 }
88 //First Mapper
89
90 import org.apache.hadoop.io.IntWritable;
91 import org.apache.hadoop.io.LongWritable;
92 import org.apache.hadoop.io.Text;
93 import org.apache.hadoop.mapreduce.Mapper;
94
95 import java.io.IOException;
96
97 public class StreamerMapper extends Mapper<LongWritable, Text,
98     Text, IntWritable> {
99     private static final IntWritable ONE = new IntWritable(1);
100     private Text streamer = new Text();
101
102     @Override
103     protected void map(LongWritable key, Text value, Context
104         context) throws IOException, InterruptedException {
105         String[] fields = value.toString().split("\\$");
106         if (fields.length >= 3) {
107             streamer.set(fields[2]);
108             context.write(streamer, ONE);
109         }
110     }

```

```

111 //First Reducer
112 import org.apache.hadoop.io.IntWritable;
113 import org.apache.hadoop.io.Text;
114 import org.apache.hadoop.mapreduce.Reducer;
115
116 import java.io.IOException;
117
118 public class StreamerReducer extends Reducer<Text, IntWritable
    , Text, IntWritable> {
119     private IntWritable result = new IntWritable();
120
121     @Override
122     protected void reduce(Text key, Iterable<IntWritable>
        values, Context context) throws IOException,
        InterruptedException {
123         int sum = 0;
124         for (IntWritable val : values) {
125             sum += val.get();
126         }
127         result.set(sum);
128         context.write(key, result);
129     }
130 }
131
132 //Second Mapper
133
134 import org.apache.hadoop.io.IntWritable;
135 import org.apache.hadoop.io.Text;
136 import org.apache.hadoop.mapreduce.Mapper;
137
138 import java.io.IOException;
139
140 public class SortMapper extends Mapper<LongWritable, Text,
    IntWritable, Text> {
141     private IntWritable count = new IntWritable();
142     private Text streamer = new Text();
143
144     @Override
145     protected void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
146         String[] fields = value.toString().split("\\t");
147         if (fields.length == 2) {
148             streamer.set(fields[0]);

```

```

149         count.set(Integer.parseInt(fields[1]));
150         context.write(count, streamer);
151     }
152 }
153 }
154
155 //Second Reducer
156
157 import org.apache.hadoop.io.IntWritable;
158 import org.apache.hadoop.io.Text;
159 import org.apache.hadoop.mapreduce.Reducer;
160
161 import java.io.IOException;
162 import java.util.PriorityQueue;
163 import java.util.Comparator;
164
165 public class TopTenReducer extends Reducer<IntWritable, Text,
    Text, IntWritable> {
166     private PriorityQueue<StreamerCount> topTen = new
        PriorityQueue<>(10, Comparator.comparingInt(sc -> sc.
            count));
167
168     @Override
169     protected void reduce(IntWritable key, Iterable<Text>
        values, Context context) throws IOException,
        InterruptedException {
170         for (Text val : values) {
171             topTen.add(new StreamerCount(val.toString(), key.
                get()));
172             if (topTen.size() > 10) {
173                 topTen.poll();
174             }
175         }
176     }
177
178     @Override
179     protected void cleanup(Context context) throws IOException
        , InterruptedException {
180         while (!topTen.isEmpty()) {
181             StreamerCount sc = topTen.poll();
182             context.write(new Text(sc.streamer), new
                IntWritable(sc.count));
183         }

```

```
184     }
185
186     private static class StreamerCount {
187         String streamer;
188         int count;
189
190         StreamerCount(String streamer, int count) {
191             this.streamer = streamer;
192             this.count = count;
193         }
194     }
195 }
```

## Приложение 18

## Работы для четвертого эксперимента по пакетной обработке данных

```

2 //Spark
3 from pyspark.sql import SparkSession
4 from pyspark.sql.functions import col
5 from pyspark.sql.types import StructType, StringType,
  StructField
6
7 def main():
8     spark = SparkSession.builder.appName("Top Ten Users").
        getOrCreate()
9
10    input_path = "hdfs://localhost:9000/user/hadoop/medium"
11    schema = StructType([
12        StructField("arrival_timestamp", StringType(), True),
13        StructField("user", StringType(), True),
14        StructField("streamer", StringType(), True),
15        StructField("msg", StringType(), True),
16        StructField("processing_timestamp", StringType(), True
17    ])
18
19    df = spark.read.format("csv").option("delimiter", "$").
        option("header", "false").schema(schema).load(
            input_path)
20
21    user_counts = df.groupBy("user").count()
22    top_ten_users = user_counts.orderBy(col("count").desc()).
        limit(10)
23
24    top_ten_users.show(truncate=False)
25    spark.stop()
26
27 if __name__ == "__main__":
28     main()
29
30 //Hive
31 SELECT user, COUNT(*) AS message_count
32 FROM large
33 GROUP BY user
34 ORDER BY message_count DESC;

```

```

35
36 //Map Reduce
37
38 import org.apache.hadoop.conf.Configuration;
39 import org.apache.hadoop.fs.Path;
40 import org.apache.hadoop.io.IntWritable;
41 import org.apache.hadoop.io.LongWritable;
42 import org.apache.hadoop.io.Text;
43 import org.apache.hadoop.mapreduce.Job;
44 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
45 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
    ;
46
47 public class TopTenUsersJob {
48     public static void main(String[] args) throws Exception {
49         if (args.length != 3) {
50             System.err.println("Usage: TopTenUsersJob <input
                path> <intermediate path> <output path>");
51             System.exit(-1);
52         }
53
54         // First job: count the occurrences of each user
55         Configuration conf1 = new Configuration();
56         Job job1 = Job.getInstance(conf1, "Count Users");
57         job1.setJarByClass(TopTenUsersJob.class);
58         job1.setMapperClass(UserMapper.class);
59         job1.setCombinerClass(UserReducer.class);
60         job1.setReducerClass(UserReducer.class);
61         job1.setOutputKeyClass(Text.class);
62         job1.setOutputValueClass(IntWritable.class);
63
64         FileInputFormat.addInputPath(job1, new Path(args[0]));
65         FileOutputFormat.setOutputPath(job1, new Path(args[1])
            );
66
67         if (!job1.waitForCompletion(true)) {
68             System.exit(1);
69         }
70
71         // Second job: sort the users by count and get the top
            ten
72         Configuration conf2 = new Configuration();
73         Job job2 = Job.getInstance(conf2, "Top Ten Users");

```



```

74         job2.setJarByClass(TopTenUsersJob.class);
75         job2.setMapperClass(SortMapper.class);
76         job2.setReducerClass(TopTenReducer.class);
77         job2.setSortComparatorClass(IntWritable.
            DecreasingComparator.class);
78         job2.setOutputKeyClass(IntWritable.class);
79         job2.setOutputValueClass(Text.class);
80
81         FileInputFormat.addInputPath(job2, new Path(args[1]));
82         FileOutputFormat.setOutputPath(job2, new Path(args[2])
            );
83
84         System.exit(job2.waitForCompletion(true) ? 0 : 1);
85     }
86 }
87
88
89 //First Mapper
90
91 import org.apache.hadoop.io.IntWritable;
92 import org.apache.hadoop.io.LongWritable;
93 import org.apache.hadoop.io.Text;
94 import org.apache.hadoop.mapreduce.Mapper;
95
96 import java.io.IOException;
97
98 public class UserMapper extends Mapper<LongWritable, Text,
    Text, IntWritable> {
99     private static final IntWritable ONE = new IntWritable(1);
100     private Text user = new Text();
101
102     @Override
103     protected void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
104         String[] fields = value.toString().split("\\$");
105         if (fields.length >= 3) {
106             user.set(fields[1]); // Set user field
107             context.write(user, ONE);
108         }
109     }
110 }
111
112

```

```

113 //First Reducer
114 import org.apache.hadoop.io.IntWritable;
115 import org.apache.hadoop.io.Text;
116 import org.apache.hadoop.mapreduce.Reducer;
117
118 import java.io.IOException;
119
120 public class UserReducer extends Reducer<Text, IntWritable,
    Text, IntWritable> {
121     private IntWritable result = new IntWritable();
122
123     @Override
124     protected void reduce(Text key, Iterable<IntWritable>
        values, Context context) throws IOException,
        InterruptedException {
125         int sum = 0;
126         for (IntWritable val : values) {
127             sum += val.get();
128         }
129         result.set(sum);
130         context.write(key, result);
131     }
132 }
133
134
135 //Second Mapper
136
137 import org.apache.hadoop.io.IntWritable;
138 import org.apache.hadoop.io.Text;
139 import org.apache.hadoop.mapreduce.Mapper;
140
141 import java.io.IOException;
142
143 public class SortMapper extends Mapper<LongWritable, Text,
    IntWritable, Text> {
144     private IntWritable count = new IntWritable();
145     private Text user = new Text();
146
147     @Override
148     protected void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
149         String[] fields = value.toString().split("\\t");
150         if (fields.length == 2) {

```

```

151         user.set(fields[0]);
152         count.set(Integer.parseInt(fields[1]));
153         context.write(count, user);
154     }
155 }
156 }
157
158
159 //Second Reducer
160
161 import org.apache.hadoop.io.IntWritable;
162 import org.apache.hadoop.io.Text;
163 import org.apache.hadoop.mapreduce.Reducer;
164
165 import java.io.IOException;
166 import java.util.PriorityQueue;
167 import java.util.Comparator;
168
169 public class TopTenReducer extends Reducer<IntWritable, Text,
    Text, IntWritable> {
170     private PriorityQueue<UserCount> topTen = new
        PriorityQueue<>(10, Comparator.comparingInt(uc -> uc.
            count));
171
172     @Override
173     protected void reduce(IntWritable key, Iterable<Text>
        values, Context context) throws IOException,
        InterruptedException {
174         for (Text val : values) {
175             topTen.add(new UserCount(val.toString(), key.get()
                ));
176             if (topTen.size() > 10) {
177                 topTen.poll();
178             }
179         }
180     }
181
182     @Override
183     protected void cleanup(Context context) throws IOException
        , InterruptedException {
184         while (!topTen.isEmpty()) {
185             UserCount uc = topTen.poll();

```

```
186         context.write(new Text(uc.user), new IntWritable(
187             uc.count));
188     }
189
190     private static class UserCount {
191         String user;
192         int count;
193
194         UserCount(String user, int count) {
195             this.user = user;
196             this.count = count;
197         }
198     }
199 }
```