ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Математико-механический факультет

Кафедра прикладной кибернетики

Выпускная квалификационная работа

Поточный агрегатор данных

Уровень образования: бакалавриат

Направление 01.03.02 «Прикладная математика и информатика»

Основная образовательная программа СВ.5004.2017 «Прикладная математика и информатика»

Профиль «Нелинейная динамика, информатика и управление»

Студент:
Шакиров Артур Каримович

Научный руководитель:
Профессор кафедры прикладной кибернетики,
д.ф.-м.н. Мокаев Тимур Назирович

Рецензент:
Ведущий научный сотрудник кафедры прикладной кибернетики,
д.ф.-м.н. Кудряшова Елена Владимировна

Санкт-Петербург

2021

Saint Petersburg State University
Faculty of Mathematics and Mechanics
DEPARTMENT OF APPLIED CYBERNETICS

Bachelor's Thesis

Streaming Data Aggregator

Student:
Artur Shakirov

Scientific Supervisor:
Professor, Department of Applied Cybernetics,
Dr. of Sci.
Timur Mokaev

Reviewer:
Leading Researcher, Department of Applied Cybernetics,
Dr. of Sci.
Elena Kudryashova

Saint Petersburg
2021

Оглавление

1. Введение **4**

2. Постановка задачи **6**

3. Обзор средств для решения задачи **7**

3.1. Kafka Streams 7

3.2. Spark Streaming 7

3.3. Spark Structured Streaming 8

3.4. Flink 8

3.5. Сравнение описанных решений 9

3.5.1. Сравнение с Spark Streaming 9

3.5.2. Сравнение с Flink 10

3.5.3. Сравнение с Kafka Streams 10

4. Архитектукра проекта **11**

4.1. Временные окна и водяные знаки 12

5. Описание решения **15**

5.1. Запуск необходимых сервисов 15

5.2. Региональный сервер 17

5.3. Центральный сервер 20

5.4. Генерация тестовых данных 22

6. Заключение **24**

Литература **25**

А. Листинг программы **27**
А.1. SparkRegionServer 27
А.1.1. Инициализация SparkSession 27
А.1.2. Функция запуска класса 27
А.1.3. Инициализация KafkaLoader 28
А.1.4. Функции для сопоставления ip-адреса и города 28
А.1.5. Функция подготовки информации о клиентах 29
А.1.6. Подготовка географического распределения 29
А.1.7. Подготовка расчета процентиля длительности сессии . . 30
А.2. SparkMainServer 32
А.2.1. Распаковка поступающих данных для географического распределения 32
А.2.2. Окончательный расчет географического распределения . 32
А.3. KafkaGenerator 34
А.3.1. Создание потоков рассылки 34

1. Введение

Перед современными системами обработки данных встает острая необходимость обрабатывать большие объемы информации, которая постоянно обновляется и увеличивается в размерах. В связи с важностью анализа больших данных было введено понятие Big Data. Одним из первых этот термин ввел Джон Мэши на конференции 1999 года USENIX [1]. Понятие Big Data развивалось настолько быстро и беспорядочно, что нет формального общепринятого определения. В докладе, прозвучавшем на конференции IC-ININFO 2014 [2], подробно разбирается проблема неструктурированного определения для Big Data. В статье выделяются четыре основных темы: информация, технологии, методы и влияние. В настоящей работе рассматриваются методы для анализа больших данных.

Разнообразные методы и технологии разработаны и адаптированы для агрегирования, анализа, обработки и визуализации больших данных. Эти методики основаны на статистике, информатике, прикладной математике и экономике. К подобным методам можно отнести A/B тестирование, обучение ассоциативным правилам, классификация, кластерный анализ, краудсорсинг, слияние и интеграция данных. К тому же их число постоянно возрастает. Более подробную информацию об этих и других методах можно прочитать в книге McKinsey [3].

Часто для управления разнообразными системами нет необходимости рассматривать все полученные данные. Для корректной работы достаточно использовать только наиболее актуальную информацию, поскольку устаревшая может быть не только бесполезна, но и сильно повлиять на достоверность результатов. В связи с этим при анализе выбираются только те данные, ко-

торые поступили не позднее, чем за некоторое "оптимальное" время. И уже в зависимости от системы обработки выбирается, какое время наиболее оптимально.

Для работы с данными с выборкой по времени наиболее удобны системы потоковой обработки (Stream processing). При такой обработке данные выстраиваются в упорядоченную последовательность экземпляров, которые могут быть прочитаны только один или некоторое малое число раз, при ограниченных возможностях вычисления и хранения. За счет этого потоковая обработка имеет высокую скорость работы. Такие приложения позволяют обойти загрузку и хранение поступающих данных в традиционные базы данных, что экономит не только пространство в памяти, но и время на ее выделение. К примерам работы таких приложений можно отнести мониторинг сетей, моделирование пользователей в веб-приложениях, сенсорные сети в электросетях, управление данными коммуникаций, прогнозирование на фондовых рынках и т. д. В современных тенденциях для пользователей наиболее важно получить, пусть и приблизительный ответ, но за как можно более короткий срок. Больше информации о Stream Processing можно узнать в книге Жуана Гама [4]. К сервисам, позволяющим выполнять подобные вычисления, можно отнести такие системы, как Kafka Streams, Spark Streaming, Spark Structured Streaming, Flink.

2. Постановка задачи

Имеется географически распределенная, высоко нагруженная система. В каждой локации есть некоторое количество серверов, каждый из которых логирует два набора информации: информация о сессиях пользователей и о их запросах. На основе этих данных не обходимо построить поточный агрегатор, обеспечивающий близкую к реальному времени интегральную статистику по следующим измерениям:

- географическое распределение по городам,
- распределение пользователей по платформам, с помощью которой пользователь пользуется сервисом (web-приложение/ android/ ...),
- зависимость платформы от города,
- медиана 95 процентиля и 95 процентиль длительности сессии,
- процентили времени выполнения по конечным точкам.

Логирование осуществляется в распределенную очередь сообщений Apache Kafka.

3. Обзор средств для решения задачи

Рассмотрим сервисы, позволяющие вести потоковую передачу данных, а также поддерживающих вычисления, необходимые в поставленной задаче. Кроме того сервис должен обладать возможностью работать с Apache Kafka, который выступает в качестве пространства, куда поступают логи приложения.

3.1. Kafka Streams

Kafka Streams — это не самостоятельная среда, а клиентская библиотека для разработки распределенных потоковых приложений и микросервисов, которые работают с данными, хранящимися в кластерах Kafka. Эта библиотека позволяет выполнять обработку записей для каждого из события. Kafka Streams используется для работы с данными по мере их поступления и обеспечивает необходимую обработку с малой задержкой, без необходимости группировать их в микропакеты. [5]

3.2. Spark Streaming

Spark Streaming — это один из модулей Spark, предназначенный для обработки и анализа входящих данных в реальном времени. Как и многие приложения Spark работают на основе концепции RDD, Spark Streaming использует в качестве основы DStreams, или же дискретизированные потоки. Они представляют собой последовательность данных со временем, причем DStream

представляется последовательностью RDD, каждый из которых исполняется на своем определенном временном шаге. [6]

3.3. Spark Structured Streaming

Начиная со Spark 2.x, доступен Structured Streaming. Он стал еще одним способом для работы с потоковой передачей данных. Structured Streaming основан на библиотеке Spark SQL. В отличии от Spark Streaming, здесь используются Dataframe и Dataset API. Таким образом можно использовать любой запрос SQL, используя API Dataframe. [7]

3.4. Flink

Flink — это проект для распределенной потоковой и пакетной обработки данных. Flink не только обеспечивает высокую скорость при потоковой передаче в реальном времени и поддерживает механизм exactly-once, но также является механизмом пакетной обработки данных. Flink поддерживает оба вида работы с данными, благодаря тому, что пакетная обработка рассматривается, то есть обработку конечных статических данных, как особый случай потоковой обработки. [8]

3.5. Сравнение описанных решений

Среди четырех предложенных сервисов было решено использовать Spark Structured Streaming, поскольку тот превосходит остальные три решения как по производительности, так и по пропускной спсобности. ([9], [10], [11])Краткая информация по сравнениям прпедложена в Таб. 3.1.

	Kafka Streams	Spark Streaming	Spark Structured Streaming	Flink
Доступные языки программирования	Java, Scala	Java, Python, Scala	Java, Python, Scala	Java, Python, Scala, R
Потоковый режим	✓	микро-пакеты	✓	✓
Временные окна	✓	✓	✓	✓
Водяные знаки	—	✓	✓	✓
Задержка 8-80тыс. TPS	—	$<$10 сек	$<$24 сек	$<$5 сек
Задержка 88тыс. TPS	—	$\sim$ 55 сек	$\sim$ 33 сек	$\sim$ 10 сек
Пропускная способность	$\sim$ 0.7 млн. зап./с	—	$\sim$ 65 млн. зап./с	$\sim$ 33 млн. зап./с

Таблица 3.1.: Сравнение решений [9, 11, 12]

Далее предложено более подробное сравнение с каждым из трех оставшихся вариантов.

3.5.1. Сравнение с Spark Streaming

Spark Structured Streaming больше подходит для потоковой передачи в реальном времени, в то время как Spark Streaming применяется для пакетной обработки данных. К тому же в Structured Streaming используется более оптимизированные API, против устаревшего RDD, использующегося в Spark Streaming.

3.5.2. Сравнение с Flink

Если сравнивать Structured Streaming с Flink, то, хотя первый и обладает довольно низкой задержкой, но не настолько низкую как у второго, хотя в более новых версиях разница сильно сократилась. Однако, наиболее важным различием является то, что у Structured Streaming пропускная способность гораздо выше, чем у Flink.

3.5.3. Сравнение с Kafka Streams

Хотя Kafka Streams является легковесной библиотекой, благодаря чему становится очень простой в использовании и развертывании. Особенно полезно это для потоковой передачи из Kafka и последующей выгрузки обратно в Kafka. Однако, это становится и одной из его проблем, так как невозможно использование в отдельности от Kafka. К тому же Kafka при большой нагрузке не настолько хорош, как Spark Structured Streaming.

4. Архитектукра проекта

Для решения поставленной задачи предварительно мною была разработана архитекутра, в соответсвии с которой был написан проект. (см. Рис. 4.1)

Рис. 4.1.: Архитектура процесса.

Как видно по Рис. 4.1 имеется некоторое количество серверов, которое может варьироваться. Внутри каждого региона такие сервера производят логирование в Apache Kafka, и уже оттуда производится считывание для первичной обработки данных по каждой из метрик. Этот шаг существует, для того чтобы снизить объем передаваемой информации, уменьшить нагрузку на основной сервер, а также для сохранения конфеденциальности данных.

После окончания обработки на региональных серверах все данные записываются в Kafka. И уже оттуда, на основе этой подготовленной информации, производится окончательный рассчет метрик. И полученные результаты складываются в Kafka для возможной последующей обработки, или же для отправки на хранение.

4.1. Временные окна и водяные знаки

Далее рассмотрим используемые в решении при обработке входящих данных временные окна (window), а также задержки считывания или водяные знаки (watermarks).

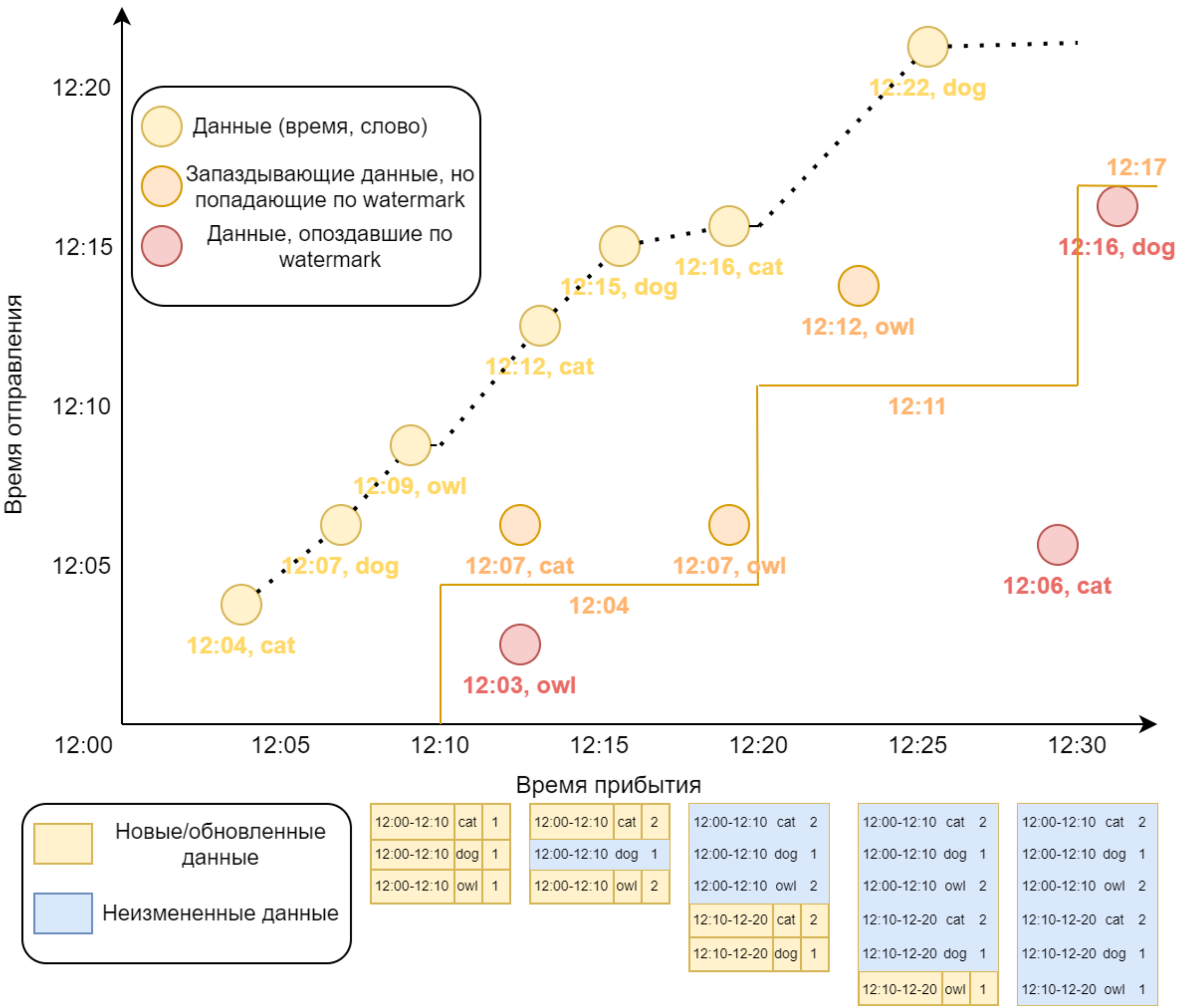

Рис. 4.2.: Окно длиной 10 минут, задержка 5 минут, сдвиг отсутствует.

Окна без смещения определяются только своей длинной, в Spark для их использования применяется команда `window('timestamp', '10 minutes')`, где первое поле соответсвует полю таблицы, для которого применяется окно, а второе поле определяет длину окна. Для задания водяных знаков используется команда `withWatermarks('5 minutes')`. С помощью окон можно группировать данные, по промежуткам времени, соответсвующим моментам их отправки.

В качестве примера рассмотрим подсчет поступающих слов по окнам (См Рис. 4.2). Здесь к 12:10 поступают только три записи, соответсвющие окну 12:00–12:10. На момент окончания окна опреляется самая поздняя пришедшая запись, и в зависимости от времени ее прибытия определяется нижняя граница ожидания, до окончания очередного окна. Таким образом в приведенном примере watermark длиной в 5 минут определяет нижную границу ожидания в 12:04, за счет этого поступают еще две записи, которые обновляют уже записанную информацию. Более ранние посылки игнорируются, так, например, **(12:03, owl)**, отправленная ранее 12:04 не идет в обработку. По аналогии происходит подсчет слов в окне 12:10–12:20, к концу окна имеется только три записи **(12:12, cat)**, **(12:15, dog)** и **(12:16, cat)**, максимальное время которых равно 12:16 и задает новую нижнюю границу в 12:11, поэтому запаздывающий **(12:12, owl)** учитвается при подсчете, но **(12:08, cat)** посланный ранее 12:11 не учитывается при подсчете слов.

Второй тип окон — окна со сдвигами, они определяются также, как предыдущие, но доваляется третий параметр, отвечающий за размер сдвига `window('timestamp', '10 minutes', '5 minutes')`. В этом случае, при длине окна 10 минут, и длине сдвига 5 минут, в дополнение к окнам, рассматриваемым в предыдущем примере, добавляются те же окна, только сдвинутые на 5 минут, и при подсчетах использоуется окна 12:00–12:10, 12:05–12:15, 12:10–12:20, 12:15–12:25, ...

При рассмотрении прошлого примера со сдвигом в 5 минут (См Рис.4.3), можно заметить, что в общих чертах ситуация похожая, но теперь каждые 5 минут обновляется нижня граница ожидания из-за чего могли пропасть

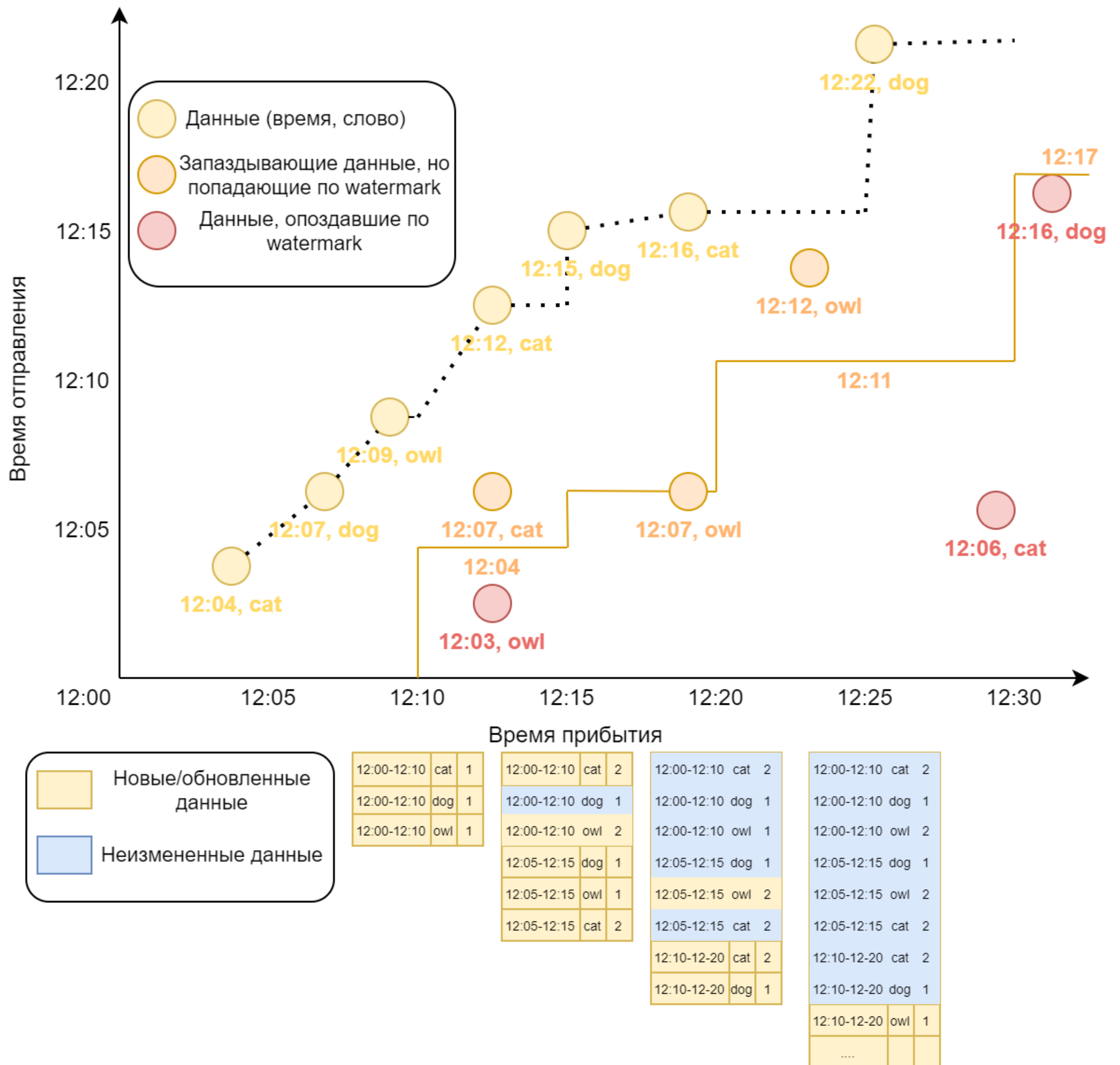

Рис. 4.3.: Окно длиной 10 минут, задержка 5 минут, сдвиг 5 минут.

некоторые данные, к примеру, **(12:07, owl)** находится на самой границе, и при времени наибольшей записи окна, равной не 12:12, а 12:13, запись бы не учитывалась.

5. Описание решения

Ниже описаны реализации каждого модуля, используемого в процессе симуляции, а также методы их настройки.

5.1. Запуск необходимых сервисов

Для выполнения работы была выбрана операционная система Windows 10.

Послькy в качестве способа передачи данных между узлами является Kafka, то для ее функционнирования необходим Zookeeper. Чтобы запустить его, необходимо в командной строке из директории Kafka выполнить следующую команду

```
start bin\windows\zookeeper-server-start.bat
config\zookeeper.properties
```

в качестве параметра здесь выступает конфигурационный файл для запуска. Для поднятия самого сервера Kafka нужно после полного запуска Zookeeper из той же директории запустить похожую команду

```
start bin\windows\kafka-server-start.bat config\server.properties
```

Здесь же в качестве параметра выступает аналогичный конфигурационный файл для запуска сервера Kafka Для простоты запуска эти команды были переписаны в один общий .bat файл.

Для корректной работы программы необходимы некоторые topic внутри Kafka, два из которых служат в качестве входных данных для передачи информации о сессиях пользователей, а также их действий. Помимо этого для

передачи первичнообработанных метрик от региональных серверов до основного для каждой метрики был введен свой topic. Чтобы создать topic с именем `kafka-topic` по адресу `localhost:9092` воспользуеумся следующей командой

```
kafka-topics.bat --create --bootstrap-server localhost:9092
--replication-factor 1 --partitions 1 --topic kafka-topic
```

По причине отсутствия реального сервера, который бы предоставлял данные для обработки, то для генерации исходных данных в Kafka используется рукописный Python класс KafkaGenerator, в качестве параметров ему передаются три обязательных поля, а именно имя топика, адрес сервера, функция для генерации данных, а также несколько опциональных: число потоков, количество отправок в минуту и время работы бота, записанное в минутах.

Для имитации регоинальных серверов создаются экземпляры класса SparkRegionServer, в качестве параметров ему передаются входящий и исходящий bootstrap-сервера. Класс отвечает за считывание данных из исходных из Kafka, предварительную обработку метрик, а также последующая отправка подготовленных данных в следующий Kafka, предназначенный для конечной обработки центральным сервером.

Для работы центрального сервера создается экземпляр класса SparkMainServer, для него передается адрес bootstrap-сервера, с которого и собираются подготовленные данные. После сбора данных происходит окончательный рассчет распределений, а также производится вычисление процентилей посредством встроенных утилит. Результаты вывчислений выгружаются в CSV.

Для запуска программы воспользуемся следующей командой. В качестве параметра передается пакет `spark-sql-kafka-0-10_2.12:3.0.0` для возможности работать с Kafka внутри Spark.

```
start ...\pyspark\bin\spark-submit
--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.0 main.py
```

5.2. Региональный сервер

Для предварительной обработки исходных данных, поступающих из Kafka, для последующего рассчета метрик, создается экземпляр класса `SparkRegionServer` со следующими обязательными параметрами:

input_kafka_bootstrap_server — bootstrap-сервер, из которого произво-диься считывание исходных данных Kafka
output_kafka_bootstrap_server — bootstrap-сервер, в который отправляются первичнообработанные данные, для последующего рассчета метрик на основном сервере.

Для потковой обработки данных используется библиотека `pyspark 3.0.0`. Кроме того, чтобы была возможность считывать исходные данные из Kafka, а также для последующей отправки первичнообработанных данных на центральный сервер используется пакет `spark-sql-kafka-0-10_2.12:3.0.0`.

Помимо того, что при инициализации экземпляра класса определяются входящие параметры, для последующей работы с сервисами Spark инициализируется SparkSession.

Для запуска первичной обработки исходных данных необходимо воспользоваться методом `start()`, в котором происходит выгрузка подготовленных данных по каждой из пяти метрик, взятых по различным окнам.

Методы `self.geographic_distribution()`, `self.platform_distribution()` и `self.platform_city_dependency()` организованы похожим образом, поэтому для рассмотрения их устройства, проанализируем работу метода, отвечающего за первичную подготовку географического распределения, `self.geographic_distribution()`. Для этого сначала рассмотрим метод `init_kafka_loader`, считывающий данные из Kafka, по `kafka_topic`, а также по `kafka_value_schema`, схеме входящих данных, для преобразования формата JSON.

В этом методе считывается поток из Kafka по bootstrap-серверу, который был определен при инициализации класса, из топика, переданному в каче-

стве параметра. Среди полей, предоставляемых Kafka, нами были использованы поля `timestamp`, хранящем время прибытия записи, а также `value`, которое хранит в себе данные в формате JSON, которые преобразуются с помощью метода `from_json`, принимающий в качестве параметров поле, в котором проходят преобразования, а также схема хранящихся там данных. Преобразованные данные складываются в поле `parsed_json`.

Поскольку считывание проходит из двух Kafka топиков, один из которых хранит информацию о сессии пользователя с полями (`user_id`, `user_platform`, `session_duration`, `user_ip_address`), а второй — о событиях с полями (`user_id`, `request_status`, `response_time`, `end_point`, `server_name`), используется дополнительные модули для распаковки преобразованный из JSON данных `prepared_client_session_stream`, `prepared_event_stream`. Для обощения рассмотрим первый из них, в котором проходит распаковка поля `parsed_json`.

В отличие от метода `prepared_event_stream` здесь на вывод подается не просто `parsed_stream`, а преобразованный с помощью метода `city_by_ip_mapping_client_session`, который сопоставляет каждому полю `user_ip_address` соответсвующий ему город, согласно карте `IP_ADDRESS_MAP`, заменяя при этом само поле на `city`.

Теперь можно перейти к описанию самого `self.geographic_distribution()`. Основная проблема для создания различных окон заключалась в том, что в Spak Structured Streming не предусмотренна возможность разветвления потока, из-за чего нельзя провести расчет метрик по различным из одного потока, поэтому в цикле по доступным длинам окон, `AVAILABLE_WINDOWS`, проходит создание нового потока для считывания персонально для каждого окна и каждой метрики. Во избежании задержек в связи с долгой передачей потока, а также, чтобы не потерять данные было принято решение рассматривать лишь те поля, которые запаздывают не более чем на пять секунд, что указывается через метод `.withWatermark("timestamp", '5 seconds')`, после чего происходит группировка по соответсвующим окнам, а также соответсвющему полю, в данном случае по `ctiy`, и происходит подсчет количества записей

внутри каждой из групп. При группировке с помощью `window` добавляется поле с соответсвующим названием, внутри которого хранятся границы окна. Для четкого определения окна были выбраны, соответсвующие границы окна, основное поле, по которому проводилась группировка, в данном случае поле `city`, а также подсчитанное количество.

После чего данные отправляются в соответсвующий topic. Здесь с помощью метода Spark `to_json` происходит преобразование каждой записи к формату JSON, после чего начинается запись потока в Kafka в формате append, наиболее подходящий для работы с оконными данными, по необходимому kafka-топику, а также ранее заданному bootstrap-серверу. В качестве checkpoint для данных выступают диерктории, предназначенные персонально для каждого способа вывода.

Методы `self.prepared_session_duration_info()` и `self.prepared_response_time_info()`, использующиеся для подготовки перед расчетом процентилей, организованы похожим образом, поэтому для рассмотрения их устройства, проанализируем работу метода, отвечающего за первичную подготовку расчета процентиля длительности сессии, `self.prepared_session_duration_info()`. Аналогично устройству ранее расмотренных методов для расссчета распределений здесь используются методы `init_kafka_loader`, а также `prepared_client_session_stream`. После загрузки исходных данных выберем только время прибытия записи и длительность сессии, во-первых, для уменьшения передаваемых данных, во-вторых, чтобы обезличить данные и не перемещать конфиденциальную информацию. И уже в таком виде данные отправляются на основной сервер.

5.3. Центральный сервер

Для окончательного рассчета метрик по данным, ранее подготовленных на региональных серверах, создается экземпляр класса `SparkMainServer` со следующим обязательным параметром:

input_kafka_bootstrap_server — bootstrap-сервер, из которого производится считывание первичнообработанных данных Kafka.

Данный модуль во многом схож с описанным ранее `SparkRegionServer`, для работы используются те же библиотеки, кроме того, в силу необходимости считывать данные из Kafka полностью дублируется метод `init_kafka_loader`.

При инициализации класса все также определяется каждый из входящих параметров.

С помощью методов `init_geographic_distribution_loader`, `init_platform_distribution_loader`, `init_platform_city_dependency_loader`, `init_prepared_session_duration_info_loader`, `init_prepared_response_time_info_loader` происходит инициализация считывателей данных из соответсвующих топиков Kafka. Все они устроенны похожим образом, поэтому, в качестве примера, достаточно рассмотерть метод `init_geographic_distribution_loader`.

В функции происходит определение схемы необходимой для корректной работы преобразователя данных из формата JSON, а также как и в аналогичном методе предыдущего класса происходит распаковка поля `parsed_json`.

Для окончательного расчета распределений исользуются функции `collect_geographic_distribution`, `collect_platform_distribution`, `collect_platform_city_dependency`. По своей сути они схожи между собой, поэтому достаточно рассмотреть только первый из них, расчитывающий географическое распределение.

Получив обработанные данные, происходит группировка по границам окон, взятых на подготовительном этапе, а также по полю, содержащему город

пользователя. Кроме того, используется Watermark длинной 5 секунд для ожидания запоздающих данных, и для его корректной работы при группировке в дополнение к ранее описанным полям добавляется окно для того, чтобы все поступающие данные по соответсвующей группировке не разбивались на несколько окон. После описанной группировки выполняется суммирование по полю `count`. Полученные результаты записываются в Kafka для возможности последущей обработки или же записи в какую-либо базу данных.

Таким образом для каждого города было рассчитано общее число сессий пользователей, выходящих в сеть из соответствующей локации, по различным промежуткам времени.

При расчете перцентелий в `collect_session_duration_percentile` и `collect_response_time_percentile` используется метод, включенный в Spark, реализующий алгоритм описанный в [13]. В качестве параметра точности $accuracy|$ используется значение по умолчанию, равное 10000, при этом относительная ошибка приближения равна $\frac{1.0}{accuracy}$. Каждый из двух методов использует одинаковую схему работы. В связи с этим в качестве примера рассматривается только расчет медианы 95 процентиля и самого 95 процентиля длительности сессии.

После получения подготовленных данных из Kafka с помощью withWatermark по полю `timestamp_arrival` отсеиваются запаздывающие данные, после чего группируется по окну времени прибытия с необходимой длиной. И производиться расчет процентиля с помощью команды аггрегирования `.agg(expr("approx_percentile(response_time, array(0.5))"))`, после чего результаты выгружаются в Kafka topic, соответсвующий рассчитываемой величине.

5.4. Генерация тестовых данных

Из-за отсутствия реального сервера, который бы записывал свои логи в Kafka, данные создает и пишет отдельное приложение, написанное на языке Python. Для возможности использовать один класс для генерации информации как о сессиях, так и о событиях одним из параметров выступает функция генерации, благодаря которой можно адаптировать исходные данные к поставленной задаче.

Для генерации исходных данных создается экземпляр класса `KafkaGenerator` со следующими параметрами:

topic_name — имя kafka-топика, на который отправляются данные
bootstrap_server — bootstrap-сервер, на котором расположена Kafka
data_generator — функция для генерации, отправляемых данных
thread_count — количество, создаваемых потоков для генерации данных в Kafka, по умолчанию создается один поток
sends_per_minute_for_thread — число отправок данных в минуту для каждого потока, по умолчанию значение равно одной посылке в минуту
working_time — время работы каждого потока, выраженное в минутах, по умолчанию время работы равно одной минуте

Для отправки данных в Kafka используется модуль `KafkaProducer` из библиотеки kafka-python 2.0.2.

При инициализации класса определяется каждый из входящих параметров. Для того, чтобы запустить генерацию данных в Kafka, необходимо воспользоваться методом `start()`. Внутри этого метода инициализируется экземпляр `KafkaProducer`, после чего в цикле создается необходимое число потоков, каждый из которых производит отправку данных, сгенерированных с помощью функции, переданной в качестве параметра, в Kafka с задержкой, рассчитанной в соответствии с указанной на входе частотой отправки сообщений.

Функции генерации данных `client_session_data_generator` и

`event_data_generator` согласно описанным ранее схемам для каждого потока информации с помощью выбора случайного элемента из заранее заготовленных массивов заполняется каждое поле. Так, например, для создания данных о сессиях пользователей `user_id` принимает случайное значение от 0 до 40, длительность сессии — от 3 до 20, платворма пользователя выбирает один из трех вариантов: android, web или компьютерное приложение. Ip-адрес выбирается из списка заранее опрделенных так, чтобы ему в соответсвии был город. Из собранных данных формируется словарь (`dict`) с ключами, соответсвующими названиям полей. и полученная структура возвращается в качестве результата функции.

6. Заключение

В ходе решения поставленной задачи, мною были рассмотрены системы с открытым кодом, предназначенные для потоковой передачи и обработки данных. На основе четырех рассмотреннных решений была выбрана наиболее производительная и удобная система Apache Spark Structured Streaming.

В условиях поставленной задачи реализована архитектура, удовлетворяющая выдвинутым требованиям. Кроме того в соответствие с ней на языке программирования Python были разработаны соответствующие модули, работающие с данным из Apache Kafka. Поточные вычисления произовдятьлся на базе Spark Structured Streaming.

Реализованная программа позволяет вести статистику близкую к реальному времени в соответствии с фиксированными метриками, которые включают в себя отыскания распределений, расчет перцентилей и их медиан. К тому же написана программа для генерации тестовых данных в Apache Kafka.

Литература

[1] Silicon Graphics/Cray Research John R. Mashey. Big data and the next wave of infrastress problems, solutions, opportunities. 1999.

[2] Michele Grimaldi Andrea De Mauro, Marco Greco. What is big data? a consensual definition and a review of key research topics. 2014.

[3] McKinsey Global Institute Brad Brown Jacques Bughin Richard Dobbs Charles Roxburgh Angela Hung Byers James Manyika, Michael Chui. *Big data: The next frontier for innovation, competition, and productivity.* McKinsey Institute, 2011.

[4] JoГJo Gama and Pedro Rodrigues. *Data Stream Processing.* Springer Verlag, 2007.

[5] William P. Bejeck Jr. *Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API.* Manning Publications, 2018.

[6] Andy Konwinski Patrick Wendell Matei Zaharia, Holden Karau. *Learning Spark: Lightning-Fast Big Data Analysis.* O'Reilly Media, 2015.

[7] Gerard Maas Francois Garillot. *Stream Processing with Apache Spark: Mastering Structured Streaming and Spark Streaming.* O'Reilly Media, 2019.

[8] Kostas Tzoumas Ellen Friedman. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond.* O'Reilly Media, 2016.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. *Structured*

Streaming: A Declarative API for Real-Time Applications in Apache Spark. SIGMOD '18. Association for Computing Machinery, New York, NY, USA, 2018.

[10] Elkhan Shahverdi, Ahmed Awad, and Sherif Sakr. *Big Stream Processing Systems: An Experimental Evaluation.* 2019.

[11] PГ©rez-Godoy M.D. GonzГЎlez P. Puentes, F. An analysis of technological frameworks for data streams. 2020.

[12] Hamed Tabesh Elham Nazari, Mohammad Hasan Shahriari. *BigData Analysis in Healthcare: Apache Hadoop , Apache spark and Apache Flink.*

[13] Michael Greenwald and Sanjeev Khanna. *Space-Efficient Online Computation of Quantile Summaries*, volume 30. Association for Computing Machinery, New York, NY, USA, May 2001.

A. Листинг программы

A.1. SparkRegionServer

A.1.1. Инициализация SparkSession

```
@staticmethod
def init_spark():
        app_name = f"SparkRegionServer"
        spark = SparkSession \
                .builder \
                .appName(app_name) \
                .getOrCreate()
        spark.sparkContext.setLogLevel('INFO')
        return spark
```

A.1.2. Функция запуска класса

```
def start(self):
        data_frames = list()
        data_frames += self.geographic_distribution()
        data_frames += self.platform_distribution()
        data_frames += self.platform_city_dependency()
        data_frames += self.prepared_session_duration_info()
        data_frames += self.prepared_response_time_info()
        return data_frames
```

A.1.3. Инициализация KafkaLoader

```
    def init_kafka_loader(self, kafka_topic, kafka_value_schema):
return self.init_spark() \
        .readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers",
                self.input_kafka_bootstrap_server) \
        .option("subscribe", kafka_topic) \
        .load() \
        .select(col('timestamp').cast('timestamp').alias('timestamp'),
                from_json(col('value').cast('string'),
                          kafka_value_schema).alias('parsed_json'))
```

A.1.4. Функции для сопоставления ip-адреса и города

```
def generate_city_map_data_frame(self):
        return self.init_spark().createDataFrame(
                prepare_ip_address_map(IP_ADDRESS_MAP),
                schema=StructType(
                        [StructField("map", MapType(StringType(),
                                                    StringType()))]))


def city_by_ip_mapping_client_session(self, parsed_stream):
        city_map_data_frame = self.generate_city_map_data_frame()
        return parsed_stream \
                .join(broadcast(city_map_data_frame
                               .select(explode("map")
                                       .alias("user_ip_address",
                                              "city"))),
                        on='user_ip_address',
                        how='left')
```

A.1.5. Функция подготовки информации о клиентах

```
def prepared_client_session_stream(self):
        parsed_stream = self \
                .init_kafka_loader(CLIENT_SESSION_TOPIC,
                                   CLIENT_SESSION_SCHEMA) \
                .select(col('timestamp').alias('timestamp'),
                        col('parsed_json').getField('user_id')
                        .alias('user_id'),
                        col('parsed_json').getField('user_platform')
                        .alias('user_platform'),
                        col('parsed_json').getField('session_duration')
                        .alias('session_duration'),
                        col('parsed_json').getField('user_ip_address')
                        .alias('user_ip_address'))
        return self.city_by_ip_mapping_client_session(parsed_stream)
```

A.1.6. Подготовка географического распределения

```
data_frames = list()
topic = f'geographic_distribution'
for window_duration in AVAILABLE_WINDOWS:
        data_frame = self.prepared_client_session_stream() \
                .withWatermark("timestamp", '10 seconds') \
                .groupBy(window('timestamp', window_duration),
                         col('city')) \
                .count() \
                .select(col('window').start
                        .cast('string').alias('start'),
                        col('window').end
                        .cast('string').alias('end'),
                        col('city'), col('count')) \
```

```
                .withColumn('window_duration', lit(window_duration))
        return_data_frame = data_frame \
                .select(to_json(struct([col(c).alias(c)
                                        for c in data_frame.columns]))
                        .alias('value')) \
                .writeStream \
                .trigger(processingTime='5 seconds') \
                .outputMode("append") \
                .format("kafka") \
                .option("kafka.bootstrap.servers",
                        self.output_kafka_bootstrap_server) \
                .option("topic", topic) \
                .option("checkpointLocation",
                        f'{CHECKPOINT_LOCATION} '
                        f'{topic} {window_duration}') \
                .start()
        data_frames.append(return_data_frame)
return data_frames
```

A.1.7. Подготовка расчета процентиля длительности сессии

```
topic = f'prepared_session_duration_info'
data_frame = self.prepared_client_session_stream() \
        .withWatermark("timestamp", '10 seconds') \
        .select(col('session_duration'),
                col('timestamp').alias('timestamp_arrival'))
        return_data_frame = data_frame \
                .select(to_json(struct([col(c).alias(c)
                                        for c in data_frame.columns]))
                        .alias('value')) \
```

```
                .writeStream \
                .trigger(processingTime='5 seconds') \
                .outputMode("append") \
                .format("kafka") \
                .option("kafka.bootstrap.servers",
                        self.output_kafka_bootstrap_server) \
                .option("topic", topic) \
                .option("checkpointLocation",
                        f'{CHECKPOINT_LOCATION} {topic}') \
                .start()
return [return_data_frame]
```

A.2. SparkMainServer

A.2.1. Распаковка поступающих данных для географического распределения

```
def init_geographic_distribution_loader(self):
        geographic_distribution_schema = StructType(
                [StructField("start", StringType()),
                 StructField("end", StringType()),
                 StructField("city", StringType()),
                 StructField("count", IntegerType()),
                 StructField("window_duration", StringType())])

        geographic_distribution_df = self.init_kafka_loader(
                self.geographic_distribution_kafka_topic,
                geographic_distribution_schema)
        return geographic_distribution_df.select(
                col('timestamp').alias('timestamp'),
                col('parsed_json').getField('start').alias('start'),
                col('parsed_json').getField('end').alias('end'),
                col('parsed_json').getField('city').alias('city'),
                col('parsed_json').getField('count').alias('count'),
                col('parsed_json').getField('window_duration')
                        .alias('window_duration'))
```

A.2.2. Окончательный расчет географического распределения

```
def collect_geographic_distribution(self):
        topic = 'geographic_distribution_results'
        data_frame = self.init_geographic_distribution_loader() \
```

```
            .withWatermark("timestamp", '10 seconds') \
            .groupBy(window(col('timestamp'), '20 seconds',
                           '10 seconds'),
                    col('start_border'), col('end_border'),
                    col('city')) \
            .sum('user_count')\
            .select(col('start_border'), col('end_border'),
                   col('city'),
            col('sum(user_count)').alias('count'))
    return_data_frame = data_frame \
            .select(to_json(struct([col(c).alias(c)
                                   for c in data_frame.columns]))
            .alias('value')) \
            .writeStream \
            .trigger(processingTime='5 seconds') \
            .outputMode("append") \
            .format("kafka") \
            .option("kafka.bootstrap.servers",
                   self.output_kafka_bootstrap_server) \
            .option("topic", topic) \
            .option("checkpointLocation",
                   f'{CHECKPOINT_LOCATION} {topic}') \
            .start()
    return return_data_frame
```

A.3. KafkaGenerator

A.3.1. Создание потоков рассылки

```
def get_thread_tuple(self):
        if self.kafka_producer is None:
        raise ValueError("'producer' is not defined")

        topic_name = self.topic_name
        producer = self.kafka_producer
        repeat_counts = self.working_time * \
                        self.sends_per_minute_for_thread
        sending_delay = 60 / self.sends_per_minute_for_thread
        data_generator = self.data_generator
        return topic_name, producer, repeat_counts, \
               sending_delay, data_generator


def init_kafka_producer(self):
        self.kafka_producer = KafkaProducer(
                bootstrap_servers=[self.bootstrap_server],
                value_serializer=lambda x:
                json.dumps(x).encode('utf-8'))


def start(self):
        def data_sender_thread(topic_name, producer, repeat_counts,
                               sending_delay, data_generator):
                for _ in range(repeat_counts):
                        producer.send(topic_name,
                                      value=data_generator())
                        time.sleep(sending_delay)
```

```
self.init_kafka_producer()
threads = list()
for _ in range(self.thread_count):
        thread_tuple = self.get_thread_tuple()
        threads.append(
        threading.Thread(target=data_sender_thread,
                         args=thread_tuple,
                         daemon=False)
        )
        threads[-1].start()
return threads
```