ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Математико-механический факультет
Кафедра прикладной кибернетики
Выпускная квалификационная работа

Система управления выгрузкой данных в облачное хранилище

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:

Тюльков Никита Андреевич

Научный руководитель:
Профессор кафедры прикладной кибернетики,
д.ф.-м.н. Кузнецов Николай Владимирович

Рецензент:
Ведущий научный сотрудник кафедры прикладной кибернетики,
д.ф.-м.н. Киселева Мария Алексеевна

Санкт-Петербург

2021

Saint Petersburg State University

Faculty of Mathematics and Mechanics

DEPARTMENT OF APPLIED CYBERNETICS

Bachelor's Thesis

Management system for uploading data to cloud storage

Student:

Nikita Tiulkov

Scientific Supervisor:

Associate Professor, Department of Applied Cybernetics,

Dr. of Sci.

Nikolay Kuznetsov

Reviewer:

Leading Researcher, Department of Applied Cybernetics,

Dr. of Sci.

Mariia Kiseleva

Saint Petersburg

2021

Содержание

1 Введение

В современном мире появляется все больше и больше данных. Начиная со сбора метрик на интернет сайтах для анализа активности пользователей, улучшения дизайна или функциональности и заканчивая данными, получаемыми из сенсоров, GPS сигналов телефонов и так далее [1, 2].

Такой быстрый рост данных порождает различные проблемы, например, как анализировать эти данные [2]. Обычно данные хранятся в источниках, таких как реляционные и нереляционные базы данных. Как правило, аналитические запросы влияют на скорость работы БД, поскольку создают избыточную нагрузку, вследствие чего требуется переместить данные в аналитическое хранилище, прежде чем как-то взаимодействовать с ними [3].

В результате, появляется необходимость в инструменте, способном получать из источника, например, нереляционной базы данных, информацию, трансформировать ее в формат, соответствующий бизнес-требованиям, не нарушая правила конфиденциальности, и затем выгружать в виде файла в хранилище [4]. Такая последовательность действий называется ETL процессом — Extract, Transform, Load, в переводе — “извлечение, преобразование и загрузка”. В целом, процессы ETL позволяют автоматизировать операцию выбора, сбора и обработки данных из хранилища данных, а также производить выходные данные в наилучшем формате для последующей обработки или для задач бизнеса [5, 10].

При этом необходимо, чтобы инструменты по извлечению, преобразованию и загрузке данных могли выполнять свою задачу быстро и надежно. Это приводит к таким понятиям как: отказоустойчивость, “хотя бы один раз” (“at least once”) семантика доставки данных и масштабируемость.

Под отказоустойчивостью предполагается, что приложение является устойчивым к сбоям. Под сбоем понимается отклонение в выполнении одной компонентой системы предназначенных ей задач [6]. Добиться полной отказо-

устойчивости невозможно, тем не менее вероятность сбоя можно снизить с помощью правильного проектирования механизмов устойчивости к сбоям [6].

Может возникнуть ситуация, когда одна машина выгружает данные, например, какие-либо сообщения, но происходит сбой. В таких случаях система может обеспечивать “хотя бы один раз”, “как максимум один раз”, “ровно один раз” семантики доставки данных [7].

Увеличение нагрузки на систему или ограниченность данных, проходящих через обработку системой, приводит к понятию масштабируемости. Масштабируемость — это способность системы справляться с увеличенной нагрузкой [6]. Один из способов обеспечения масштабируемости является виртуализация, то есть абстрагирование от аппаратной реализации [8].

Итак, процесс выгрузки из источника данных, преобразование этих данных, загрузка в хранилище вместе с обеспечением “хотя бы один раз” семантики доставки, отказоустойчивости и масштабируемости является актуальной задачей в современном мире больших данных.

2 Постановка задачи

Так как хранилище данных обычно использует систему извлечения, преобразования и загрузки данных [9], то целью данной работы является создание системы (приложения), выполняющей выгрузку из MongoDB, преобразование данных и загрузку в аналитическое хранилище HDFS/Amazon S3 (Hadoop Distributed File System/Amazon Simple Storage Service), то есть обеспечивающей ETL процесс [10]. Система должна опрашивать источник в многопоточном режиме, выгружать данные по частям и обеспечивать:

– “хотя бы один раз” семантику доставки данных, то есть при сбое на одной из машин передаваемые данные не должны пропасть, а должны быть доставлены в указанное место, даже если это приводит к дублированию этих данных;

– отказоустойчивость (fault-tolerance);

– автоматическое масштабирование (automatic scaling) засчет контейнеризации, зависящее от нагрузки на систему;

– фиксация версии системы в случае изменений конфигураций, то есть при создании новой версии системы с новыми конфигурациями или новыми исполняемыми файлами должно быть выполнено сохранение предыдущей версии конфигурации или исполняемого файла.

Выгрузка обновленных или добавленных в момент работы системы данных не поддерживается.

3 Обзор литературы и аналогов

Рассмотрим готовые решения, функции которых схожи с поставленной задачей. Для этого была изучена документация некоторых бесплатных (open-source) и платных аналогов. В выборку вошли наиболее известные системы такие как: Apache NiFi [11], Apache Flume [12], SteamSets Data Collector [13], Apache Airflow [14], AWS Glue [15].

Apache NiFi

Преимущества

+ Бесплатный (open-source).

+ Не зависит от Hadoop.

+ Множество доступных процессоров (возможностей работы с данными, например, получение данных из источника, преобразование, выгрузка в источник).

+ При отказе узла возможно восстановить потерянные данные (но не из очереди).

+ Возможность создания пользовательского процессора.

+ Кластеризация.

+ “Хотя бы раз” семантика доставки.

+ Удобный графический интерфейс.

+ Поддержка различных форматов файлов.

Недостатки

– Отсутствие механизма репликации данных.

– При отказе источника данных в узле происходит потеря данных.

– При отказе узла (жесткого диска) происходит потеря данных в очереди.

– При отказе узла необходимо восстанавливать данные ручным образом.

– При отказе узла алгоритм Round Robin будет выполнять балансировку медленно, ожидая восстановления потерянного узла.

Apache Flume

Преимущества

+ Бесплатный (open-source).

+ Возможность добавления пользовательских интеграций.

+ Удобен для перемещения потоковых данных (streaming data).

+ Кластеризация.

+ “Хотя бы раз” семантика доставки данных.

+ Отказоустойчивость в Hadoop.

+ Ручная конфигурация всего процесса.

Недостатки

– Малое количество встроенных интеграций.

– Неудобен для работы с данными в БД.

– Нет явной возможности записи файлов колоночного формата (только если использовать сторонние инструменты типа Impala).

– Упор на HDFS в качестве источника доставки данных.

– Необходимо следить за отказоустойчивостью для пользовательских интеграций.

– Необходимость ручным образом выстраивать топологию кластера.

– Ручная конфигурация всего процесса.

SteamSets Data Collector

Преимущества

+ Бесплатный (open-source).

+ Интеграция со множеством сторонних систем.

+ “Хотя бы раз” семантика доставки данных.

+ Кластеризация (с использованием кластеров Kafka, MapR, HDFS, Amazon S3).

+ Подробный мониторинг записей для каждого процессора.

+ Пользовательские процессоры.

+ Удобный пользовательский интерфейс.

Недостатки

– Для конфигурации одного процессора необходимо останавливать весь процесс.

– При ошибке отладка проводится для всего потока данных, а не для отдельного процессора.

– Кластеризация только при чтении из HDFS, Amazon S3, Kafka cluster, MapR cluster.

Apache AirFlow

Преимущества

+ Бесплатный (open-source).

+ Разработка, планирование и осуществление мониторинга сложных рабочих процессов, в том числе извлечения-преобразования-загрузки на Python.

+ Масштабируемость.

+ Интеграция с множеством систем и сервисов.

Недостатки

– Автоматизация.

– Большие накладные расходы (временная задержка 5-10 секунд) на постановку компонент системы в очередь и приоритизацию задач при запуске.

– Пост-фактум оповещения о сбоях в конвейере данных.

AWS Glue

Преимущества

+ Масштабирование, подготовка и настройка полностью управляются в Apache Spark (работает на бессерверной платформе). Лёгкость в обслуживании и развертывании.

+ Отказоустойчивость.

+ Возможность отладки логов.

+ Автоматическое обнаружение схемы. То есть позволяет разработчикам автоматизировать поисковые crawler'ы для получения информации, связанной со схемой, и сохранения ее в каталоге данных, который затем можно использовать для управления задачей (job).

+ Генерация кода задачи.

+ Developer endpoints - возможность изменить код задачи для своих нужд, импортировать этот код в кастомную библиотеку.

Недостатки

– Платный.

– Недостаточная интеграция с другими платформами/системами/источниками (только AWS и несколько реляционных баз данных таких как: MySQL, Oracle, PostgreSQL).

– Не самые гибкие возможности ручной настройки окружения.

– Нет возможности хранить временные файлы, исполняемые файлы на своей стороне из-за бессерверной инфраструктуры, что влияет на производительность системы.

– Нет дополнительной синхронизации данных. Все данные сначала размещаются на S3, поэтому платформа не является лучшим вариант для задач ETL в реальном времени (real-time ETL jobs).

4 Архитектура системы

4.1 Архитектура кластера

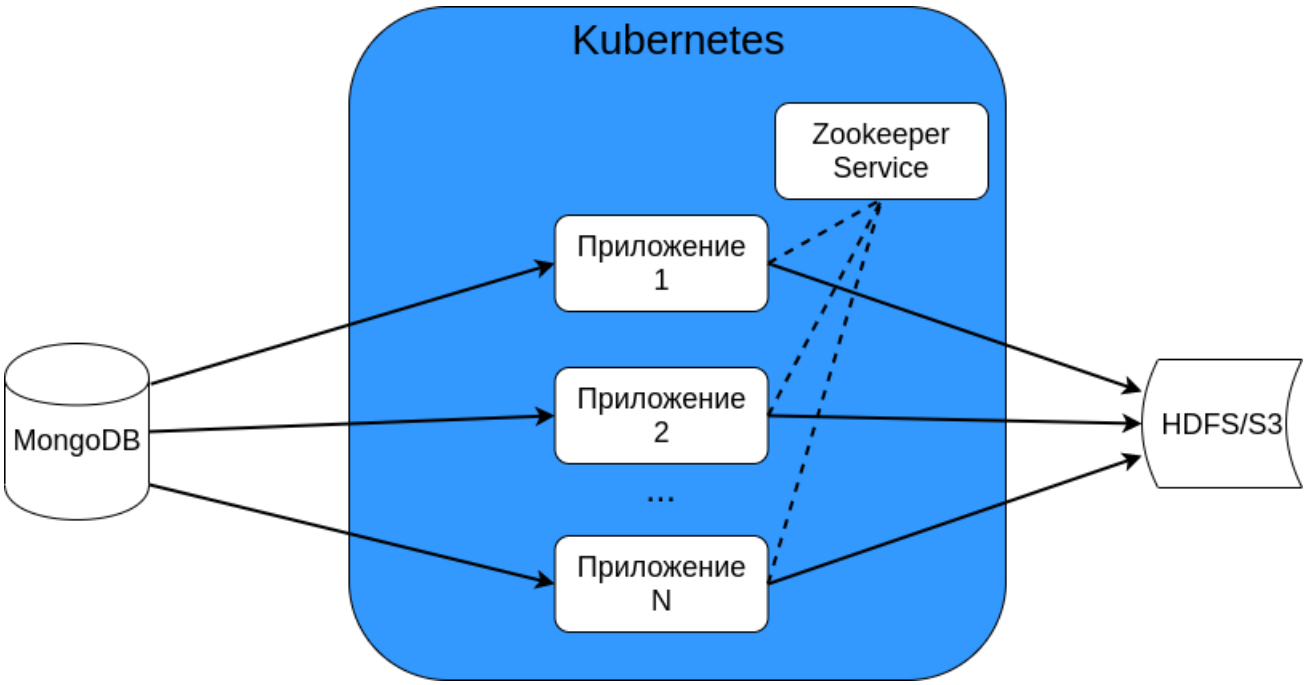

Рис. 1: Архитектура кластера

На Рис. 1 представлена архитектура кластера с N серверами, где N = 3. В ней можно выделить следующие элементы:

- Источник — место, откуда данные извлекаются. В реализации взята нереляционная (NoSQL) база данных MongoDB [16].
- Кластер (группа серверов [17]), управляемый Kubernetes (платформа для работы с множеством контейнеров [18]) — в него входят N серверов, а также сервис Zookeeper (для координации серверов в данной распределенной системе [19]; на основе информации, записанной с помощью модуля Статистика (см. описание ниже) в Zookeeper, сервера берут на себя задачи и указывают это в Zookeeper.

  Kubernetes обеспечивает системе отказоустойчивость и масштабируемость.

- Хранилище — место, куда будут выгружены данные. В реализации взята распределенная отказоустойчивая файловая система Hadoop (HDFS) [17, 20], а также облачное хранилище Amazon S3 (Amazon Simple Storage Service) [21], для взаимодействия с которым используется такой же API, как и с HDFS.

4.2 Архитектура системы на одном сервере

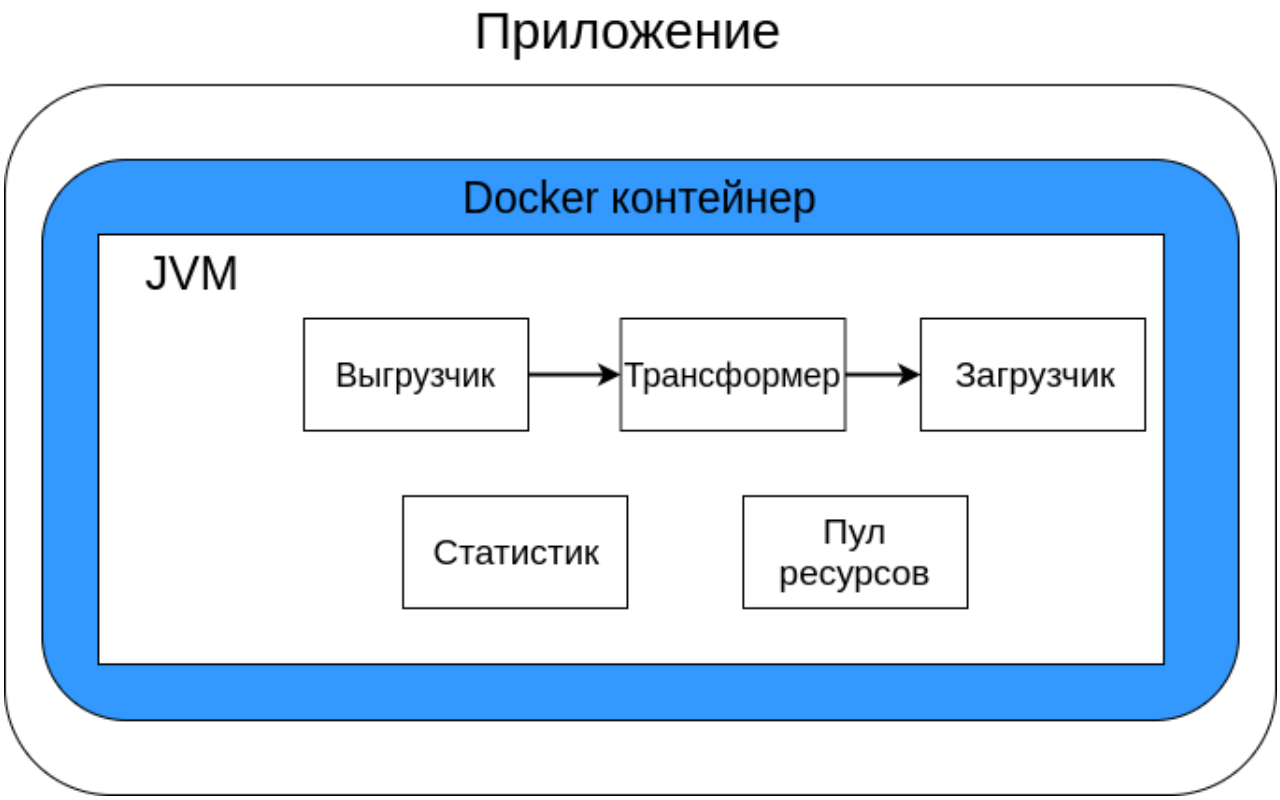

Рис. 2: Архитектура системы на одном сервере

На Рис. 2 представлена архитектура самого ETL процесса на одном сервере. В ней можно выделить следующие элементы:

- Docker (инструмент для создания образа приложения и зависимостей для работы этого приложения [27]) контейнер с JVM процессом:
  - Выгрузчик - получает данные из хранилища.
  - Трансформер - обрабатывает данные в зависимости от бизнес-логики.
  - Загрузчик - загружает данные в HDFS/S3 хранилище.
  - Статистик - выполняет разбиение данных на порции и запись этих порций в ZooKeeper; создает описание того, какой сервер выполняет задачу с конкретными данными (порциями) для того, чтобы при падении сервера была возможность обработать потерянные данные;

предоставляет информацию по доступным, выполняющимся и готовым задачам; выдает статистику выполнения задачи.

Данный модуль обеспечивает системе “хотя бы один раз” семантику доставки данных.

– Пул ресурсов - выделяет ресурсы процессам (многопоточность).

5 Сценарии восстановления системы

Считается, что среднее время наработки на отказ (mean time to failure, MTTF) жесткого диска составляет от 10 до 50 лет [6]. Таким образом, нельзя с уверенностью сказать, что приложение будет работать вечно. Также могут возникнуть неполадки с сетью или с самим приложеним. Для обеспечения отказоустойчивости необходимо предусмотреть такие отказы в разрабатываемой системе. В основном предполагается, что система будет способна переключиться на другую, работающую машину, вследствие чего работа системы будет продолжена [22].
Далее приведены возможные сбои и способы восстановления системы.

- При отказе источника, то есть базы данных MongoDB, производится вывод предупреждения оператору (в консоль выводится предупреждения о том, что база данных MongoDB недоступна) и повторная попытка соединения с базой данных.

- При отказе приложения, то есть самого ETL (Java) процесса, происходит логирование и автоматический перезапуск docker контейнера (то есть установлена политика перезапуска при ошибке - restart on-failures). При этом незавершенные задачи перераспределяются между работающими ETL процессами (серверами).

- При отказе файловой системы необходим полный перезапуск сервера. При этом незавершенные задачи перераспределяются между работающими ETL процессами (серверами).

- При отказе сервера необходим его полный перезапуск. При этом незавершенные задачи перераспределяются между работающими ETL процессами (серверами).

- При отказе сети у одного из серверов происходит попытка подключения

сервера к сети. При этом незавершенные задачи перераспределяются между работающими ETL процессами (серверами). Необходим перезапуск сервера.

- При отказе Zookeeper или Kubernetes необходим полный перезапуск кластера.

6 Программная реализация

Приложение состоит из нескольких пакетов, таких как:

"*commons*" - отвечает за вспомогательные классы:

- Option - класс, который содержит параметры (ключи) командной строки при запуске приложения. Пример: ключ "-f", после которого идет путь к файлу конфигураций.
- Options - класс, который содержит список всех опций.
- CommandLineParser - класс, который проводит парсинг командной строки и заносит ключи и их значения в Option.
- CommandLine - класс, содержащий Map с Option, позволяет получить по ключу значение опции.
- Configuration - класс, имеющий поле Map, в котором хранятся конфигурации для работы программы. Эти конфигурации указываются в файле, путь которого указывается после ключа "-f".
- ConfigurationParser - класс, который проводит парсинг конфигурационного файла и заносит данные, как ключ-значение в Map.
- ZookeeperConf - класс, отвечающий за первоначальную инициализацию узлов ZooKeeper, за предоставление данных по пути узла и за предоставление свободной порции данных, которую может получить машина. Данные в ZooKeper представлены в виде узлов, которые объединены в древовидную структуру. То есть каждый узел может содержать данные и иметь дочерние узлы [23].

"*concurrency*" - отвечает за классы для работы с многопоточностью:

- PoolWorker - класс, представляющий собой задачу, которая извлекается из очереди всех задач пула ресурсов и выполняется одним потоком.

- ResourcePool - класс, который регулирует работу потоков приложения, а именно выделяет задачи для потоков, позволяет добавить задачи в очередь, запускать и останавливать потоки.

"*partitioner*" - отвечает за классы, связанные с разделением данных на отдельные задачи:

- Partitioner - интерфейс, который содержит один метод - *partition*. Этот метод должен быть определен в классах, реализующих этот интерфейс.

- MongoPartitioner - класс, который реализует интерфейс Partitioner. Позволяет разделить данные из источника MongoDB на отдельные порции и затем записать их в ZooKeeper. Далее процессы и потоки будут брать себе в работу эти порции, как отдельные задачи.

"*extractor*" - отвечает за классы, связанные с получение данных из источника:

- Extractor - интерфейс, который содержит один метод - *extract*. Этот метод должен быть определен в классах, реализующих этот интерфейс.

- MongoExtractor - класс, который реализует интерфейс Extractor. Позволяет получать данные из MongoDB, записывать их в список, который далее будет обработан трансформером (Transformer).

"*transformer*" - отвечает за классы, связанные с обработкой полученных данных:

- Transformer - интерфейс, который содержит один метод - *process*. Этот метод должен быть определен в классах, реализующих этот интерфейс.

- AbstractTransformer - класс, который реализует интерфейс Transformer. Этот класс не обрабатывает данные, так как не было каких-либо бизнес-требований по обработке получаемых данных. Соответственно, класс

принимает список задач и возвращает его без каких-либо преобразований.

"*loader*" - отвечает за классы, связанные с выгрузкой преобразованных данных:

- Loader - интерфейс, который содержит один метод - *load*. Этот метод должен быть определен в классах, реализующих этот интерфейс.
- HDFSLoader - класс, который реализует интерфейс Loader. Этот класс выгружает преобразованные после Transformer'а данные в файл, расположенный в хранилище HDFS (Hadoop Distributed File System) или Amazon S3 (Amazon Simple Storage Service).

"*core*" - отвечает за основные классы программы:

- BatchProcess - класс, реализующий интерфейс Runnable. Можно сказать, что это класс, который является одной полноценной задачей по всему процессу извлечения, преобразования и выгрузки данных. В нем создаются экземляры классов MongoExtractor, AbstractTransformer, HDFSLoader и вызываются соответсвующие методы для работы с данными.
- Application - главный класс, содержащий метод *main*. В нем происходит первоначальная настройка ZooKeeper'а (создание соотвествующих путей в ZooKeeper), партицирование (для этого имеется экземпляр класса MongoPartitioner, который записывает задачи в ZooKeeper), создание задач и ResourcePool'а для работы с несколькими задачами, в зависимости от указанного числа потоков. При завершении всех задач из пула ресурсов происходит проверка на то, остались ли еще свободные задачи и есть ли еще задачи в работе. При их отсуствии работа приложения завершается. Иначе создается новая задача для текущего процесса.

- TaskChecker - класс, проверящий, остались ли свободные задачи и задачи, находящиеся в работе.

Работа приложения начинается с класса *Application*, в котором для первоначальной настройки ZooKeeper'а (с помощью API [23]) и партицирования используется фреймвор Apache Curator [24]. Существует проблема: хоть потоки приложения синхронизированы и могут получать данные из ZooKeeper, не мешая друг другу, между репликами приложения при работе кластера может возникать конфронтация, например, две реплики могут начать выполнять одну и ту же задачу, так как они не будут синхронизированы. Эту проблему решает фреймворк Apache Curator [24], который предоставляет средства для работы с распределенными системами.

Далее для многопоточной работы создается пул ресурсов и задачи для него. Задача - это процесс извлечения из MongoDB (с помощью API [26]) и загрузка в HDFS/S3 (с помощью API [25]) одной порции данных, которая была сформирована и записана в ZooKeeper с помощью класса *MongoPartitioner*. При отработке всех задач из пула ресурсов, приложение обращается к ZooKeeper на предмет свободных и невыполненных задач. При их отсутствии приложение спустя определенное продолжительное время проверяет список задач, находящихся в работе, и если таковые существуют (возможно, что у одного из сервером произошел сбой), то приложение начинает выполнять эту же задачу для обеспечения "хотя бы один раз" семантики доставки данных. Когда все задачи в ZooKeeper переходят в статус завершенных задач, работа системы прекращается.

Ввиду того, что система использует Docker и Kubernetes, появляется возможность отказоустойчивости. Для этого необходимо в конфигурационном файле, сформированном для работы приложения, проставить необходимые параметры, например, политика перезапуска приложения будет имеет значение "Перезапуск при сбое". Для автоматического масштабирования в том же файле проставляются значения метрик нагрузки на центральный про-

цессор и количество используемой оперативной памяти, при превышении указанных параметров Kubernetes будет создавать реплику приложения.

Также Docker решает задачу сохранения версий конфигурационных файлов при их изменении. Эта возможность появляется благодаря загрузке docker образа на сервис Docker Hub [28], который содержит репозитории с образами и их версиями. При обновлении конфигураций и загрузке в репозиторий пользователь получает возможность указать версию приложения. Все предыдущие версии конфигураций так же сохраняются.

7 Заключение

В результате проделанной работы было разработано приложение по многопоточной пакетной выгрузке данных из хранилища MongoDB в распределенное аналитическое хранилище HDFS или в облачное хранилище Amazon S3 с выполнением требований:

– “хотя бы один раз” семантика доставки данных;

– отказоустойчивость;

– автоматическое масштабирование;

– фиксация версии системы в случае изменений конфигураций.

Данная система, находящаяся в открытом доступе, в поставленной задаче является уникальной и обходит аналогичные продукты по одному или более критериям.

Список литературы

[1] Rajendra Akerkar - Big Data Computing - CRC Press, 2013

[2] Qi, Chong-chong - Big data management in the mining industry. International Journal of Minerals, Metallurgy and Materials - 2020

[3] Michael Collins - Network Security Through Data Analysis: Building Situational Awareness - 1st Edition, 2014

[4] Castellanos, Malu; Dayal, Umeshwar; Miller, Renée J. - Enabling Real-Time Business Intelligence Volume 41 || Near Real-Time Data Warehousing Using State-of-the-Art ETL Tools - (2010). [Lecture Notes in Business Information Processing]

[5] Shaker H. Ali El-Sappagh; Abdeltawab M. Ahmed Hendawi; Ali Hamed El Bastawissy. A proposed model for data warehouse ETL processes - 2011

[6] Martin Kleppmann — Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems — First Edition — O'Reilly Media, 2017

[7] Bhimani, P., Panchal, G. - Message Delivery Guarantee and Status Update of Clients Based on IoT-AMQP. Lecture Notes in Networks and Systems - 2017

[8] Kon, Joichiro; Mizusawa, Naoki; Umezawa, Ayaka; Yamaguchi, Saneyasu; Tao, Jian - Highly consolidated servers with container-based virtualization - [IEEE 2017 IEEE International Conference on Big Data (Big Data) - Boston, MA, USA (2017.12.11-2017.12.14)] 2017 IEEE International Conference on Big Data (Big Data)

[9] Sabtu, Adilah; Azmi, Nurulhuda Firdaus Mohd; Sjarif, Nilam Nur Amir; Ismail, Saiful Adli; Yusop, Othman Mohd; Sarkan, Haslina; Chuprat,

Suriayati - The challenges of Extract, Transform and Loading (ETL) system implementation for near real-time environment - [IEEE 2017 5th International Conference on Research and Innovation in Information Systems (ICRIIS) - Langkawi, Malaysia (2017.7.16-2017.7.17)] 2017 International Conference on Research and Innovation in Information Systems (ICRIIS)

[10] Galici, Roberta; Ordile, Laura; Marchesi, Michele; Pinna, Andrea; Tonelli, Roberto. - Applying the ETL Process to Blockchain Data. Prospect and Findings. Information - 2019

[11] Gerardus Blokdyk — Apache Nifi — Third Edition — 5STARCooks, 2018

[12] Hari Shreedharan — Using Flume: Flexible, Scalable and Reliable Data Streaming — First Edition — O'Reilly Media, 2014

[13] Quinto, Butch - Next-Generation Big Data || Batch and Real-Time Data Ingestion and Processing - 2018

[14] Singh, Pramod - Learn PySpark (Build Python-based Machine Learning and Deep Learning Models) - 2019

[15] Sudhakar Kalyan - Amazon Web Services (AWS) GLUE - 2018, International Journal of Management, IT and Engineering

[16] Shannon Bradshaw, Eoin Brazil, Kristina Chodorow - MongoDB: The Definitive Guide: Powerful and Scalable Data Storage - O'Reilly Media, 2019

[17] Jan Kunigk, Ian Buss, Paul Wilkinson, Lars George - Architecting Modern Data Platforms: A Guide to Enterprise Hadoop at Scale - 1st Edition, 2019

[18] Muddinagiri, Ruchika; Ambavane, Shubham; Bayas, Simran - Self-Hosted Kubernetes: Deploying Docker Containers Locally With Minikube. - [IEEE 2019 International Conference on Innovative Trends and Advances in

Engineering and Technology (ICITAET) - SHEGAON, India (2019.12.27-2019.12.28)] 2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)

[19] Supun Kamburugamuve - Survey of Apache Big Data Stack - 2013

[20] Mahmoud, Hadeer; Hegazy, Abdelfatah; Khafagy, Mohamed H. - An approach for big data security based on Hadoop distributed file system. - [IEEE 2018 International Conference on Innovative Trends in Computer Engineering (ITCE) - Aswan, Egypt (2018.2.19-2018.2.21)] 2018 International Conference on Innovative Trends in Computer Engineering (ITCE)

[21] Palankar, Mayur R.; Iamnitchi, Adriana; Ripeanu, Matei; Garfinkel, Simson - Amazon S3 for science grids - [ACM Press the 2008 international workshop - Boston, MA, USA (2008.06.24-2008.06.24)] Proceedings of the 2008 international workshop on Data-aware distributed computing - DADC '08

[22] Khan, Maqbool; Wu, Xiaotong; Xu, Xiaolong; Dou, Wanchun - Big data challenges and opportunities in the hype of Industry 4.0. - [IEEE ICC 2017 - 2017 IEEE International Conference on Communications - Paris, France (2017.5.21-2017.5.25)] 2017 IEEE International Conference on Communications (ICC)

[23] ZooKeeper: Distributed Process Coordination, https://zookeeper.apache.org/

[24] Curator zookeeper recipes, http://curator.apache.org/ (Apache Software Foundation)

[25] Apache Hadoop API 3.3.0, https://hadoop.apache.org/docs/current/api/overview-summary.html

[26] MongoDB Java Driver, https://mongodb.github.io/mongo-java-driver/

[27] Miika Moilanen - Deploying an application using Docker and Kubernetes - 2018

[28] Docker Hub, https://index.docker.io/

A Программный код

ZookeeperConf.java

```
package ru.franticlol.fff.commons;

import org.apache.zookeeper.*;
import org.apache.zookeeper.data.ACL;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.List;

public class ZookeeperConf
{
    Configuration configuration;
    ZooKeeper zooKeeper;

    public ZookeeperConf(Configuration configuration) {
        this.configuration = configuration;
        String server = configuration.getConfigurationMap()
        .get("zookeeper");
        Object lock = new Object();

        Watcher connectionWatcher = new Watcher() {
            public void process(WatchedEvent we) {
                if (we.getState() ==
                Watcher.Event.KeeperState.SyncConnected)
                {
                    synchronized (lock) {
                        lock.notifyAll();
                    }
                }
            }
        };

        int sessionTimeout = 2000;
```

```
        this.zooKeeper = null;

        try {
            synchronized (lock) {
                this.zooKeeper =
                new ZooKeeper(server, sessionTimeout,
                connectionWatcher);
                lock.wait();
            }
        } catch (IOException | InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    public Configuration getConfiguration() {
        return configuration;
    }

    public void startZookeeperConfiguration() {
        try {

            List<ACL> acls = ZooDefs.Ids.OPEN_ACL_UNSAFE;
            if (zooKeeper.exists("/conf", true) == null) {
                zooKeeper.create("/conf", null, acls,
                CreateMode.PERSISTENT);
            }

            for (String key : configuration.getConfigurationMap()
            .keySet()) {
                String znodePath = "/" + key;
                if (zooKeeper.exists("/conf" + znodePath, true)
                == null) {
                    zooKeeper.create("/conf" + znodePath,
                    configuration.getConfigurationMap().get(key)
                    .getBytes(StandardCharsets.UTF_8), acls,
```

```
                CreateMode.PERSISTENT);
        } else {
            zooKeeper.setData("/conf" + znodePath,
            configuration.getConfigurationMap().get(key)
            .getBytes(StandardCharsets.UTF_8),
            zooKeeper.exists("/conf" + znodePath, true)
            .getVersion());
        }
    }

    if (zooKeeper.exists("/conf/freePartition", true)
    == null) {
        zooKeeper.create("/conf/freePartition", null,
        acls, CreateMode.PERSISTENT);
    }

    if (zooKeeper.exists("/conf/partitionInWork", true)
    == null) {
        zooKeeper.create("/conf/partitionInWork", null,
        acls, CreateMode.PERSISTENT);
    }

    if (zooKeeper.exists("/conf/finishedTasks", true)
    == null) {
        zooKeeper.create("/conf/finishedTasks", null,
        acls, CreateMode.PERSISTENT);
    }

    if (zooKeeper.exists("/conf/workStartedFlag", true)
    == null) {
        zooKeeper.create("/conf/workStartedFlag", null,
        acls, CreateMode.PERSISTENT);
    }

    if (zooKeeper.exists("/conf/workEndedFlag", true)
```

```
            == null) {
                zooKeeper.create("/conf/workEndedFlag", null,
                acls, CreateMode.PERSISTENT);
            }
        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }
    }

    public void addZookeeperConfiguration(String key, byte[] data)
    {
        try {
            List<ACL> acls = ZooDefs.Ids.OPEN_ACL_UNSAFE;

            String znodePath = "/" + key;
            if (zooKeeper.exists("/conf" + znodePath, false)
            == null) {
                zooKeeper.create("/conf" + znodePath, data,
                acls, CreateMode.PERSISTENT);
            }
        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }

    }

    public synchronized Long findFirstFreeTask() {
        try {
            List<String> freePartitionsList = zooKeeper
            .getChildren("/conf/freePartition", null);
            if (freePartitionsList.stream()
            .anyMatch(str -> str.matches("[0-9]+"))) {
                String firstPartition = freePartitionsList.stream()
                .filter(str -> str.matches("[0-9]+"))
                .findFirst().get();
```

```
                Long freePartition = Long.valueOf(
                getData("/conf/freePartition/" + firstPartition));
                moveConfiguration("freePartition/" +
                firstPartition, "partitionInWork/"
                + firstPartition);
                return freePartition;
            } else {
                return -1L;
            }
        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }

        return null;
    }

    public void moveConfiguration(String oldKey, String newKey)
    {
        try {
            List<ACL> acls = ZooDefs.Ids.OPEN_ACL_UNSAFE;
            String znodeOldPath = "/" + oldKey;
            String znodeNewPath = "/" + newKey;
            if (zooKeeper.exists("/conf" + znodeNewPath, false)
            == null) {
                zooKeeper.create("/conf" + znodeNewPath,
                getData("/conf" + znodeOldPath)
                .getBytes(StandardCharsets.UTF_8),
                acls, CreateMode.PERSISTENT);
            } else {
                zooKeeper.setData("/conf" + znodeNewPath,
                getData("/conf" + znodeOldPath)
                .getBytes(StandardCharsets.UTF_8),
                zooKeeper.exists("/conf" + znodeNewPath,
                true).getVersion());
            }
```

```
            zooKeeper.delete("/conf" + znodeOldPath,
            zooKeeper.exists("/conf" + znodeOldPath, true)
            .getVersion());

        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }

    }

    public String getData(String nodePath) {
        try {
            byte[] data = zooKeeper.getData(nodePath, null, null);
            return new String(data);
        } catch (InterruptedException | KeeperException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

MongoExtractor.java

```
package ru.franticlol.fff.extractor;

import com.mongodb.client.*;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.recipes.
locks.InterProcessSemaphoreMutex;
import org.apache.curator.retry.ExponentialBackoffRetry;
import org.bson.Document;
import ru.franticlol.fff.commons.ZookeeperConf;

import java.util.ArrayList;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;


public class MongoExtractor<K, T> implements Extractor<K, T>
{
    ZookeeperConf zookeeperConf;

    public MongoExtractor(ZookeeperConf zookeeperConf) {
        this.zookeeperConf = zookeeperConf;
    }

    @Override
    public Map<K, T> extract() {
        System.out.println("Extracting...");
        Map<String, List<String>> documents =
        new HashMap<>();
        List<String> documentsList = new ArrayList<>();
        Long batchSize = Long.valueOf(zookeeperConf
        .getData("/conf/batchSize"));
        MongoClient mongoClient = MongoClients
        .create(zookeeperConf.getData("/conf/mongo"));
        MongoDatabase database = mongoClient
        .getDatabase(zookeeperConf.getData("/conf/dbName"));
        MongoCollection<Document> collection = database
        .getCollection(zookeeperConf
        .getData("/conf/collectionName"));

        CuratorFramework client = CuratorFrameworkFactory
        .newClient(zookeeperConf.getConfiguration()
        .getConfigurationMap().get("zookeeper"),
        new ExponentialBackoffRetry(1000, 3));
        client.start();
        InterProcessSemaphoreMutex sharedLock =
```

```
        new InterProcessSemaphoreMutex(client, "/conf");

        Long skipCount = null;
        try {
            sharedLock.acquire();
            skipCount = zookeeperConf.findFirstFreeTask();
            System.out.println("SkipCount is " + skipCount);
            sharedLock.release();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        if (skipCount != null) {
            if(skipCount != -1L) {
                for (Document document : collection.find()
                .skip(Math.toIntExact(skipCount))
                .limit(Math.toIntExact(batchSize))) {
                    documentsList.add(document.toJson());
                }
                documents.put(Long.toString(
                skipCount / batchSize), documentsList);
            }
        } else {
            System.out.println("SkipCount is null. Check
            MongoExtractor");
        }

        return (Map<K, T>) documents;
    }
}
```

HDFSLoader.java

```
package ru.franticlol.fff.loader;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
```

```
import com.google.gson.JsonElement;
import com.google.gson.JsonParser;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import ru.franticlol.fff.commons.ZookeeperConf;

import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.net.InetAddress;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.Map;

public class HDFSLoader<K, T> implements Loader<K, T>
{
    ZookeeperConf zookeeperConf;

    public HDFSLoader(ZookeeperConf zookeeperConf) {
        this.zookeeperConf = zookeeperConf;
    }

    @Override
    public void load(Map<K, T> objects) throws IOException {
        if(objects.isEmpty()) {
            return;
        }

        System.out.println("Loading started");

        Configuration configuration = new Configuration();
        configuration.set("fs.hdfs.impl",
                org.apache.hadoop.hdfs
```

```
            .DistributedFileSystem.class.getName()
    );
    configuration.set("fs.file.impl",
            org.apache.hadoop.fs.LocalFileSystem.class
            .getName()
    );
    configuration.addResource(new Path(
    "/home/nikita/Apache/hadoop/etc/hadoop/core-site.xml"));
    configuration.addResource(new Path(
    "/home/nikita/Apache/hadoop/etc/hadoop/hdfs-site.xml"));

    FileSystem fileSystem = FileSystem.get(configuration);

    String fileName = InetAddress.getLocalHost()
    .getCanonicalHostName() + "_" + Thread.currentThread()
    .getName() + "_" + objects.keySet().stream()
    .findFirst().get() + ".json";
    Path hdfsWritePath = new Path("/user/" + fileName);
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    JsonParser jp = new JsonParser();
    try {
        FSDataOutputStream fsDataOutputStream = fileSystem
        .create(hdfsWritePath, true);
        BufferedWriter bufferedWriter = new BufferedWriter(
        new OutputStreamWriter(fsDataOutputStream,
        StandardCharsets.UTF_8));
        for (T documentList : objects.values()) {
            for(String document : (List<String>) documentList)
            {
                JsonElement je = jp.parse(document);
                String prettyJsonString = gson.toJson(je);
                bufferedWriter.write(prettyJsonString);
                bufferedWriter.newLine();
            }
        }
```

```
            System.out.println("Loading finished");
            bufferedWriter.close();
            fileSystem.close();

            if (objects.keySet().stream().findFirst()
            .isPresent()) {
                String taskKey = (String) objects.keySet().stream()
                .findFirst().get();
                zookeeperConf.setEndTask(taskKey);
            }

        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

MongoPartitioner.java

```
package ru.franticlol.fff.partition;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import ru.franticlol.fff.commons.ZookeeperConf;

import java.nio.charset.StandardCharsets;

public class MongoPartitioner implements Partitioner
{
    ZookeeperConf zookeeperConf;

    public MongoPartitioner(ZookeeperConf zookeeperConf)
    {
        this.zookeeperConf = zookeeperConf;
```

```
    }

    @Override
    public void partition() {
        Long batchSize = Long.valueOf(zookeeperConf
        .getData("/conf/batchSize"));
        System.out.println("Connecting to MongoDB");
        MongoClient mongoClient = MongoClients
        .create(zookeeperConf.getData("/conf/mongo"));
        MongoDatabase database = mongoClient
        .getDatabase(zookeeperConf.getData("/conf/dbName"));
        MongoCollection<Document> collection = database
        .getCollection(zookeeperConf
        .getData("/conf/collectionName"));
        Long docCount = collection.countDocuments();
        Long partitionCount = docCount % batchSize == 0
        ? docCount / batchSize : (docCount / batchSize) + 1;
        for(long i = 0; i < partitionCount; ++i) {
            System.out.println(i * batchSize);
            zookeeperConf.addZookeeperConfiguration(
            "freePartition/" + i , String.valueOf(i * batchSize)
            .getBytes(StandardCharsets.UTF_8));
        }
    }
}
```

BatchProcess.java

```
package ru.franticlol.fff.core;

import ru.franticlol.fff.commons.ZookeeperConf;
import ru.franticlol.fff.extractor.Extractor;
import ru.franticlol.fff.loader.Loader;
import ru.franticlol.fff.transformer.Transformer;

import java.io.FileNotFoundException;
import java.io.IOException;
```

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.Map;

public class BatchProcess implements Runnable
{
    ZookeeperConf zookeeperConf;

    BatchProcess(ZookeeperConf zookeeperConf) {
        this.zookeeperConf = zookeeperConf;
    }

    @Override
    public void run() {
        try {
            Class<?> extractorClass = Class.forName(
            zookeeperConf.getData("/conf/extractor"));
            Constructor<?> extractorClassConstructor =
            extractorClass.getConstructor(ZookeeperConf.class);
            Extractor extractor = (Extractor)
            extractorClassConstructor.newInstance(
            new Object[]{zookeeperConf});

            Class<?> transformerClass = Class.forName(
            zookeeperConf.getData("/conf/transformer"));
            Constructor<?> transformerClassConstructor =
            transformerClass.getConstructor(ZookeeperConf.class);
            Transformer transformer = (Transformer)
            transformerClassConstructor.newInstance(
            new Object[]{zookeeperConf});

            Class<?> loaderClass = Class.forName(
            zookeeperConf.getData("/conf/loader"));
            Constructor<?> loaderClassConstructor =
            loaderClass.getConstructor(ZookeeperConf.class);
```

```
            Loader loader = (Loader) loaderClassConstructor
            .newInstance(new Object[]{zookeeperConf});

            Map<String, String> objects = extractor.extract();
            loader.load(transformer.process(objects));
            System.out.println(Thread.currentThread().getName()
            + " has done a job.");
        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException | InstantiationException |
        IllegalAccessException | NoSuchMethodException |
        ClassNotFoundException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Application.java

```
package ru.franticlol.fff.core;

import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.recipes.locks
.InterProcessSemaphoreMutex;
import org.apache.curator.retry.ExponentialBackoffRetry;
import ru.franticlol.fff.commons.*;
import ru.franticlol.fff.concurrency.ResourcePool;
import ru.franticlol.fff.partition.Partitioner;

import java.io.File;
import java.io.FileNotFoundException;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.List;

public class Application
```

```
{

    public static void main(String[] args) {
        System.out.println("Starting application...");
        CommandLine commandLine = CommandLineParser.parse(args);
        File configFile = new File(commandLine.getOption("f")
        .getOptionValue());
        try {
            Configuration configuration = new Configuration(
            ConfigurationParser.parse(configFile));
            ZookeeperConf zookeeperConf =
            new ZookeeperConf(configuration);

            CuratorFramework client = CuratorFrameworkFactory
            .newClient(configuration
            .getConfigurationMap().get("zookeeper"),
            new ExponentialBackoffRetry(1000, 3));
            client.start();
            InterProcessSemaphoreMutex sharedLock =
            new InterProcessSemaphoreMutex(
                    client, "/conf");

            sharedLock.acquire();
            if (!TaskChecker.checkTaskStartedFlag(zookeeperConf))
            {
                System.out.println(TaskChecker
                .checkTaskStartedFlag(zookeeperConf));
                System.out.println("Starting configuration...");
                zookeeperConf.startZookeeperConfiguration();
                System.out.println("Configuration ended");

                System.out.println("Start partitioning");
                Class<?> partitionClass = Class
                .forName(zookeeperConf
                .getData("/conf/partitioner"));
```

```
                Constructor<?> partitionClassConstructor =
                partitionClass.getConstructor(ZookeeperConf.class);
                Partitioner partitioner = (Partitioner)
                partitionClassConstructor
                .newInstance(zookeeperConf);

                partitioner.partition();
                zookeeperConf.setTaskStartedFlag();
                System.out.println(String.valueOf(zookeeperConf
                .getData("/conf/workStartedFlag").toCharArray()));
                System.out.println("Partitioning ended");
            } else {
                System.out.println("Configuration was ended by
                ~~~~~~~~~~~~~~~~another process");
            }
            sharedLock.release();

            List<BatchProcess> tasksList = new ArrayList<>();

            Integer threadCount = Math.toIntExact(Long
            .parseLong(zookeeperConf
            .getData("/conf/threadCount")));

            for (int i = 0; i < threadCount; ++i) {
                tasksList.add(new BatchProcess(zookeeperConf));
            }

            ResourcePool<BatchProcess> pool =
            new ResourcePool<>(tasksList, threadCount);


            System.out.println("Starting pool working...");
            pool.start();
            while (true) {
                if (pool.tasksIsEmpty()) {
```

```
                    Thread.sleep(3000);
                    if (!TaskChecker
                        .isFreeTasksEmpty(zookeeperConf))
                        {
                        for (int i = 0; i < threadCount; ++i)
                        {
                            pool.addTask(
                            new BatchProcess(zookeeperConf));
                        }
                        System.out.println(
                        "New task was added to poll");
                        pool.run();
                    }

                    if(TaskChecker.isFreeTasksEmpty(zookeeperConf)
                    && !TaskChecker.checkWorkIsOver(zookeeperConf))
                    {
                        Thread.sleep(60000);
                        if(TaskChecker
                            .isFreeTasksEmpty(zookeeperConf)
                        && !TaskChecker
                            .checkWorkIsOver(zookeeperConf))
                        {
                            zookeeperConf.moveAllWorkTasksToFree();
                        }
                    }

                    if (TaskChecker.isFreeTasksEmpty(zookeeperConf)
                    && TaskChecker.checkWorkIsOver(zookeeperConf))
                    {
                        System.out.println("Poll has stopped due to
                        no new tasks.");
                        pool.stop();
                        zookeeperConf.setTaskEndedFlag();
                        return;
```

```
                    }
                }
            }
        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

B Инструкция по запуску

B.1 Запуск на одной машине

Первоначальная версия приложения работает на одной локальной машине (Ubuntu 20.04 LTS), на которой установлена JDK 11 и Gradle 6.7. Данный вариант включает в себя выгрузку из хранилища в многопоточном режиме, обработку данных, полученных из хранилища и загрузку их в целевое хранилище. При этом всей конфигурацией, порционированием и синхронизацией задач занимается Apache Zookeeper (версия 3.6.2). Для работы приложения необходимо иметь JDK 11 (или JRE 11, если имеется готовый jar файл), Apache Zookeeper (версии 3.6.2, достаточно Standalone), Apache Hadoop (версии 3.2.2), база данных MongoDB (версии 4.4.3).
Для начала необходимо запустить Hadoop с помощью команды (где hadoop — основной каталог, в котором расположены все файлы):

```
./hadoop/sbin/start-all.sh
```

Запустить Zookeeper сервер командой (где zookeeper — основной каталог, в котором расположены все файлы):

```
./zookeeper/bin/zkServer.sh start
```

Запустить mongoDB командой:

```
mongo
```

Далее необходимо создать .jar файл, если его изначально нет. Для этого понадобится система автоматической сборки Gradle (версии 6.7).
Для создания .jar файла необходимо выполнить команду (где 3F — каталог, в котором расположен проект):

```
./gradlew :shadowJar
```

В данном случае используется команда gradle-wrapper. Сам gradle-wrapper создается после сборки проекта.

Далее необходимо создать конфигурационный файл, в котором будут храниться ключи-значения необходимых программе конфигураций.
Разберемся по порядку с ключами, некоторые из них являются обязательными при создании конфигурационного файла, некоторые опциональные и могут быть заданы пользователем по необходимости:

1. “mongo” — ключ, значение которого указывает адрес базы данных MongoDB, из которой будет происходить выгрузка

2. “dbName” — ключ, значение которого указывает на название базы данных, из которой будет происходить выгрузка

3. “collectionName” — ключ, значение которого указывает на название коллекции, из которой будут выгружаться данные

4. “extractor” (обязательный) — ключ, который указывает на полное название (вместе с пакетом) класса, который будет выполнять выгрузку из хранилища

5. “transformer” (обязательный) — ключ, который указывает на полное название (вместе с пакетом) класса, который будет выполнять обработку полученных из хранилища данных

6. “loader” (обязательный) — ключ, который указывает на полное название (вместе с пакетом) класса, который будет выполнять загрузку данных в хранилище

7. “partitioner” (обязательный) — ключ, который указывает на полное название (вместе с пакетом) класса, который будет разбивать изначальную базу данных на порции и создавать в Zookeeper задачи

8. “zookeeper” — ключ, значение которого указывает на адрес сервера zookeeper

9. “threadCount” — количество потоков процесса

10. “batchSize” - размер порции данных, которые будут выгружены одним потоком одного процесса

После создания конфигурационного файла можно приступать к работе приложения. В терминале необходимо выполнить следующую команду:
$ java -jar <путь к полученному .jar файлу + его название> -f <путь к конфигурационному файлу + его название>
Таким образом можно запустить приложение на одной машине.

B.2 Запуск на кластере

Для работы приложения на нескольких машинах необходимо иметь либо несколько реальных машин, либо виртуальных. В данной работе для создания нескольких машин используется технология конвейеризации. А именно несколько docker-контейнеров с приложением создаются и управляются технологией Kubernetes. Таким образом, для корректной работы необходим установленный Docker (версии 20.10.4). А также Kubernetes. В данной работе был использован локальный кластер minikube [18].
После установки docker необходимо убедиться, что в конфигурационных файлах верно указаны адреса для MongoDB и HDFS. На локальной машине они будут связаны с адресом, который указывает на Docker. Также необходимо убедиться, что порты для работы с MongoDB и HDFS открыты, а в HDFS так же выставлен доступ на запись файлов.
Для создания контейнера из приложения необходимо создать Dockerfile, в котором должны быть указаны следующие необходимые команды:

```
FROM openjdk:11
COPY 3F-1.0-all.jar /
COPY config.txt /config/
COPY hdfs-site.xml
```

```
/home/nikita/Apache/hadoop/etc/hadoop/hdfs-site.xml
COPY core-site.xml
/home/nikita/Apache/hadoop/etc/hadoop/core-site.xml
CMD ["java", "-jar", "/3F-1.0-all.jar",
"-f", "/config/config.txt"]
```

Данный Dockerfile загружает JDK 11, копирует из каталога в контейнер .jar и конфигурационный файлы, а затем выполняет команду по запуску приложения.

Теперь необходимо создать docker-образ следующей командой (находясь в директории с Dockerfile, confix.txt и с 3F-1.0-all.jar):

```
docker build -t app .
```

Затем необходимо загрузить docker образ на сайт docker.io для возможности сохранения конфигурации системы (четвертый пункт в постановке задачи). Это выполняется следующими командами:

```
docker tag app docker.io/$user$/app
docker push docker.io/$user$/app
```

Далее необходимо использовать docker-compose файл для полной настройки окружения, а именно количества реплик приложения и запуска сервиса ZooKeeper. Получившийся docker-compose.yml:

```
version: "3.3"
services:
   zookeeper:
      image: "zookeeper"
      ports:
      - "2181:2181"
   my-app:
      image: "docker.io/$user$/app"
      deploy:
```

```
        replicas: 2
    depends_on:
    - zookeeper
```

Далее необходимо использовать утилиту, переводящую docker-compose.yml в конфигурационные файлы для работы с Kubernetes. Такой возможностью обладает утилита Kompose [27]. Используем команду:

```
kompose convert
```

И получим 3 файла: "app-deployment.yaml "zookeeper-deployment.yaml "zookeeper-service.yaml". В первом файле можно изменить количество реплик приложения и путь к docker образу.

Следующим шагом является запуск сервиса ZooKeeper и приложения командой:

```
kubectl apply -f zookeeper-service.yaml,
zookeeper-deployment.yaml,app-deployment.yaml
```

Таким образом можно запустить приложение в кластере Kubernetes.