



**Universidade de Coimbra**  
**Faculdade de Ciências e Tecnologia**  
**Departamento de Engenharia Informática**

**Mestrado em Engenharia Informática**  
**Sistema de Gestão de Dados**  
**Projeto 2, 2020/2021, 2º Semestre**

**Docente**

**Professor Pedro Furtado**  
**(PNF@DEI.UC.PT)**

David Silva de Paiva	davidpaiva@student.dei.uc.pt	2020178529
Ricardo David Da Silva Briceño	briceno@student.dei.uc.pt	2020173503



## Registo de Trabalhos

### Lista de funcionalidades/objetivos implementados

Funcionalidades/Objetivos	Responsável	Esforço (horas)
Pesquisa	Ambos	32
Comunicação Kafka	Ambos	8
ETL	Ambos	16
Avaliação de Performance	David	6
Relatório	Ambos	10

*Tabela 1 – Objetivos Alcançados*

### Autoavaliação individual e global do projeto

O aluno, David Paiva, autoavalia o desempenho do grupo com uma nota de dezanove valores - numa escala de zero a vinte. Relativamente a uma autoavaliação individual, avalia o seu desempenho e o do colega, Ricardo Briceño, com dezanove valores para ambos.

O aluno, Ricardo Briceño, autoavalia o desempenho do grupo com uma nota de dezanove valores - numa escala de zero a vinte. Relativamente a uma autoavaliação individual, avalia o seu desempenho e o do colega, David Paiva, com dezanove valores para ambos.

## Índice de Figuras

Figura 1 - Diagrama das API disponibilizadas pelo Kafka [4].....	5
Figura 2 - Algumas das propriedades de configuração do Produtor .....	10
Figura 3 - Algumas das propriedades de configuração do Consumidor definidas em Runtime .....	10
Figura 4 - Componentes da Solução Desenvolvida.....	11
Figura 5 - Packages e classes dos projetos .....	12
Figura 6 - Classe ProducerRunnable.java .....	13
Figura 7 - Classe Producer.java.....	14
Figura 8 - Propriedades da classe Record.java.....	15
Figura 9 - Propriedades e construtor da classe Consumer.java .....	16
Figura 10 - Método run da classe Consumer.java .....	17
Figura 11 - Lookup dos dados nas tabelas auxiliares .....	18
Figura 12 - Parte do código responsável pela transformação da linha de update.....	20
Figura 13 - Tempo de processamento total de cada um dos ficheiros, quando recebidos separadamente .....	22
Figura 14 - Throughput do consumidor Kafka no processamento de cada um dos ficheiros .....	22

## Índice de Tabelas

Tabela 1 – Objetivos Alcançados .....	i
Tabela 2 - Comparação entre o valor bruto e o valor de um registo no SSB .....	19
Tabela 3 - Configuração da máquina onde foram realizados os testes.....	21
Tabela 4 - Tempo máximo de processamento .....	21
Tabela 5 - Número linhas por segundo e linhas processadas num período de 6h .....	23

## Índice

Registo de Trabalhos .....	i
Lista de funcionalidades/objetivos implementados .....	i
Autoavaliação individual e global do projeto .....	i
Índice de Figuras .....	ii
Índice de Tabelas .....	ii
1. Enquadramento do Trabalho .....	1
2. Estado da arte: Kafka .....	3
2.1 Apresentação Kafka .....	3
2.2 Abordagem escolhida .....	6
2.3 Deliberação da abordagem.....	7
3. Receção de linhas de update para atualização de dados do SSB .....	9
3.1 Setup .....	9
3.2 Implementação .....	10
3.3 Transformação dos dados .....	18
3.3.1 Transformação – Customer e Supplier .....	19
3.3.2 Transformação – Part.....	19
3.3.3 Transformação – Date .....	19
3.4 Performance e capacidades .....	20
4. Considerações Finais.....	26
Referências .....	28

Esta página foi deixada propositadamente em branco.

# 1. Enquadramento do Trabalho

Nesta secção é apresentado o objetivo do segundo projeto prático da unidade curricular de Sistemas e Gestão de Dados.

Durante as aulas práticas da unidade curricular houve a oportunidade de interação com diferentes tecnologias utilizadas no processamento de dados. Com isso em mente, neste projeto é pedido que se explore uma das ferramentas em detalhe, através da sua utilização num cenário prático de exemplo. Deste modo, foi escolhida a framework Apache Kafka para receção de linhas de update de SSB para integração nos dados da data warehouse SSB [1].

Este projeto divide-se em dois momentos distintos. Um primeiro momento onde foi realizado um trabalho de pesquisa que visou aprofundar o conhecimento em relação ao Kafka. Assim sendo, procurou-se analisar e entender as diferentes aplicações reais da framework e quais as suas vantagens quando ao processamento de dados diz respeito. Este breve estado da arte está documentado na secção dois deste documento. Por fim, num segundo momento, foram implementadas as funcionalidades de receção de linhas de update com o Kafka, a transformação das mesmas e, consequentemente, a sua integração na data warehouse. As decisões e escolhas tomadas durante a implementação são apresentadas na secção três do presente deste documento.

Esta página foi deixada propositadamente em branco.



## 2. Estado da arte: Kafka

Nesta secção é apresentado um breve estado da arte relativo à framework Apache Kafka. Antes de iniciar o desenvolvimento da solução e do desafio proposto no segundo projeto prático foi necessário conhecer em mais detalhe a framework, assim como, também foi importante avaliar o nível de complexidade da mesma. Com isso em mente, são aqui apresentadas as principais características da ferramenta.

### 2.1 Apresentação Kafka

A equipa do Apache Kafka, descreve a framework como uma plataforma de streaming de eventos usada por milhares de empresas para a construção de pipelines de dados de alta performance, análises de streams, integração de dados e aplicações críticas [2].

Em primeiro lugar é fundamental entender o que é uma streaming de eventos. Na prática, consiste na captura de dados, em tempo real, vindos de fontes de eventos como sensores, dispositivos móveis, serviços cloud e aplicações de software. Estes dados são armazenados de forma durável e com a garantia que não são perdidos até que sejam manipulados e processados. De uma forma geral, uma streaming de eventos assegura um caminho e uma interpretação contínua dos dados de forma que a informação certa, esteja no sítio certo, no tempo certo [3].

O Kafka combina três capacidades chaves que possibilitam o streaming de eventos:

- É possível publicar (escrever) e subscrever (ler) uma streaming de eventos, permitindo que se importe e exporte dados de e para outros sistemas. Isto segue um paradigma de produtor consumidor, mas com algumas vantagens, nomeadamente, o facto de as mensagens poderem ficar armazenadas num tópico o tempo que for necessário.
- Tal como referido, os eventos são armazenados o tempo que for necessário, sem que sejam perdidas as mensagens, permitindo assim que os projetos que usem Kafka cumpram com elevados requisitos de *reliability* e *availability*.
- Permite ainda que os eventos sejam processados em tempo real.

Todas as funcionalidades mencionadas do sistema são providenciadas de uma forma distribuída, altamente escalável, tolerante a falhas e segura.

Feita esta contextualização, é importante referir o que são eventos. Basicamente, um evento representa uma mensagem de algo que aconteceu no mundo real ou num negócio. Pode ser telemetria de sensores, a compra de um produto numa *Marketplace*, partilha de dados entre drones, entre muitos outros cenários possíveis. Na prática, um evento é um conjunto de três informações: uma chave, um valor, registo temporal em que a mensagem foi criada e cabeçalhos opcionais de meta dados.

Como já foi referido, os eventos são produzidos e consumidos. Os produtores são aplicações que produzem e publicam eventos num tópico Kafka e, os consumidores são aqueles que subscrevem um determinado tópico e leem e processam os eventos disponíveis. Para além disso, em Kafka, as aplicações produtores e consumidores são totalmente independentes umas das outras, sendo isto, um elemento-chave para que se consiga alcançar escalabilidade. Esta é uma das vantagens apresentadas pelos criadores do Kafka [3], uma vez que os produtores nunca esperam pelos consumidores e as mensagens uma vez enviadas para o tópico, nunca são perdidas.

Já foi referido mais do que uma vez a palavra tópico. Na prática, é possível olhar para um tópico como uma pasta no sistema de ficheiros e, os eventos são os ficheiros dentro dessa pasta. Em Kafka os tópicos são sempre multi-processador e multi-subscritor [3], o que significa que um tópico pode ter zero, um ou vários processadores que escrevem para ele e, consequentemente, vários consumidores que subscrevem esse mesmo tópico. Importa realçar que os eventos num tópico podem ser lidos as vezes que forem necessários. Em contraste com os sistemas tradicionais de mensagens, os eventos não são eliminados após serem consumidos, em vez disso, o programador define quanto tempo o Kafka os deve guardar, através da configuração do tópico, e só findo esse tempo é que os eventos são eliminados. Também, segundo os criadores do Kafka, a performance do Kafka é constante, o que significa que armazenar dados por um longo período de tempo é totalmente aceitável.

O Kafka pode ser acedido e configurado através da linha de comandos, mas para além disso, possui cinco API's fundamentais para Java e Scala:

- Admin API: Disponibiliza funcionalidades de gestão de tópicos e outros objetos do Kafka.

- **Producer API:** Disponibiliza funcionalidades de escrita de eventos em um ou mais tópicos.
- **Consumer API:** Disponibiliza funcionalidades de leitura de um ou mais tópicos e funcionalidades de processamento dos eventos que chegam aos tópicos.
- **Kafka Streams API:** Disponibiliza funcionalidades que permitem a implementação de aplicações de processamento de streams e microserviços. Na prática os eventos são lidos de um tópico, são processados e de seguida, são enviados para um tópico de output.
- **Kafka Connect API:** Disponibiliza a possibilidade de construção e execução de conectores que consomem ou produzem streams de eventos de ou para sistemas externos de modo que sejam integrados com o Kafka. Os criadores do Kafka, dão o exemplo de um conector, que se liga a uma base de dados, e que capta quaisquer alterações realizadas num conjunto de tabelas.

Esta breve introdução sobre o Kafka pode ser resumida num simples diagrama como o apresentado na Figura 1. O objetivo desta introdução é ser o mais simples e esclarecedora possível em relação ao que é o Kafka. Posto isto, foi necessário decidir qual a API do Kafka que pode ser útil para resolver o problema proposto.

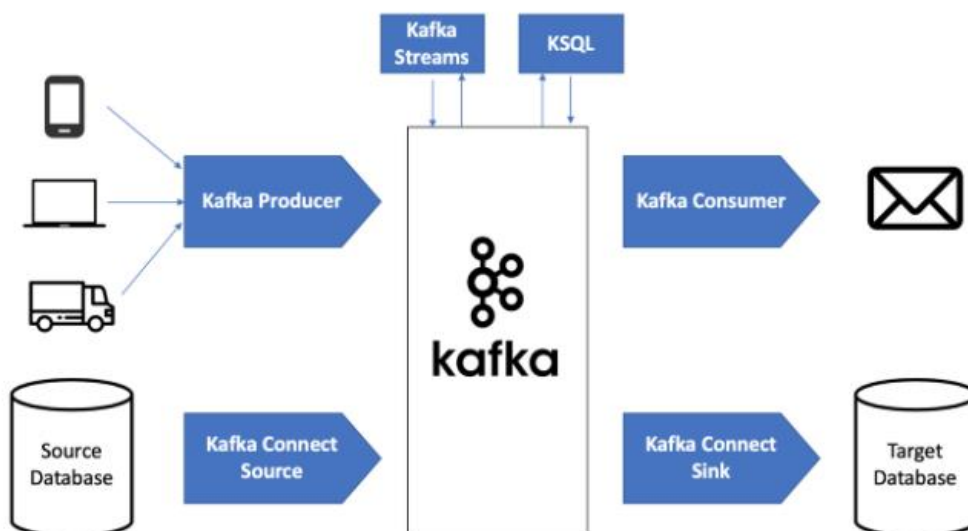


Figura 1 - Diagrama das API disponibilizadas pelo Kafka [4]

## 2.2 Abordagem escolhida

De modo a cumprir o objetivo do segundo projeto prático foi necessário entender, na prática, como é que as API's disponibilizadas pelo Kafka poderiam ser úteis na extração, transformação e carregamento (ETL) dos dados do SSB.

Uma vez que o objetivo do trabalho é a realização de ETL com Kafka, foi feita pesquisa no sentido de encontrar soluções e projetos semelhantes. Não foram encontradas soluções ou projetos semelhantes no que toca ao uso da API do Kafka para a realização de ETL. No entanto, foi encontrado um artigo onde é feita uma comparação entre o Kafka e o Spark [5]. É referido que o Kafka não suporta a transformação de dados (ETL) ao invés do Spark que disponibilizada uma panóplia de bibliotecas para esse fim. Algo que também foi referido neste artigo é o facto de ser possível usar o Kafka para envio e receção de dados – aproveitando assim todas as vantagens já mencionadas acima – e utilizar o Spark para o ETL dos dados.

Apesar de existir a possibilidade de utilizar o Spark para a realização de ETL, esse não foi o caminho seguido para a realização do trabalho. Uma vez que era solicitado a utilização do Kafka, tentou-se investir o máximo de esforço em encontrar uma solução para de ETL onde apenas se utilizasse Kafka e a linguagem Java. Com isso em mente, foi utilizada as API's Kafka Producer e Kafka Consumer para o envio e receção dos dados, respetivamente. Do lado do consumidor, os dados são tratados em tempo real. Este tratamento foi programado em Java e foi baseado num exemplo tutorial disponibilizado pela Confluent [6]. Nesse tutorial é feita uma transformação dos dados em Java, assim que o evento é consumido, e imediatamente a seguir, a informação transformada é enviada para um novo tópico. No cenário desenvolvido para este trabalho, foi feita a mesma abordagem, com a única diferença de que no fim do processamento, a informação transformada é persistida na data warehouse.

Para além disso, foram analisados dois outros exemplos disponíveis no repositório de código Github. O primeiro exemplo, foi disponibilizado no decorrer das aulas práticas da unidade curricular e consiste, na produção de dados de telemetria e o seu consumo, tratamento e carregamento para uma base de dados [7]. O segundo exemplo usa as API's Kafka Producer e Consumer para simular a receção de informação a compras [8], sendo muito semelhante ao solicitado neste trabalho.

### 2.3 Deliberação da abordagem

Concluímos, com a pesquisa, que o Kafka é uma boa solução para o processamento de dados e possui ferramentas robustas para o transporte desses dados para os destinos pretendidos. Porém, apesar do Kafka ter variedade de API's de processamento, não é fácil realizar transformações de dados dinamicamente. Precisamos de construir condições complexas entre produtores e consumidores. É necessário muito trabalho e esforço, portanto, decidimos usar o Kafka para a fase de extração dos dados e transporte dos dados, servindo apenas de intermediário entre as fontes de dados e os utilizadores finais. Para tarefas de transformação decidimos optar por o fazer com Java. Esta decisão foi baseada na falta de ferramentas para a transformação dos dados, decisão esta que viemos reforçar com os esclarecimentos e consentimento do docente.

Esta página foi deixada propositadamente em branco.

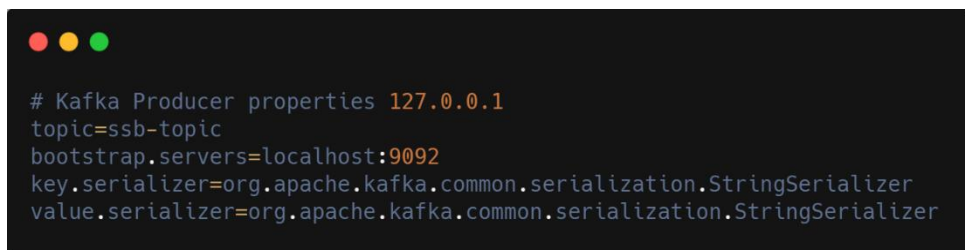
### 3. Receção de linhas de update para atualização de dados do SSB

Nesta secção é apresentado o setup construído para a implementação do trabalho proposto, assim como as instalações, configurações e outros aspetos fundamentais para a implementação da solução. Por fim, é feita uma avaliação da performance e das capacidades do Kafka na receção e tratamento dos eventos.

#### 3.1 Setup

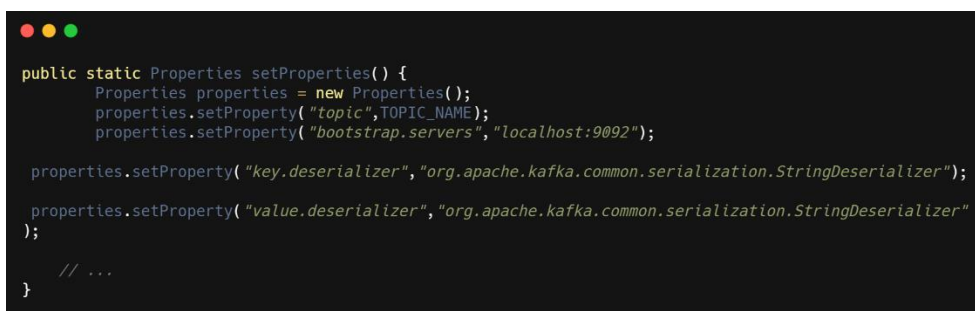
O Kafka foi usado como mecanismo de transporte dos dados e para isso foram utilizadas as API's Kafka Producer e Consumer. Para além disso, a solução desenvolvida para enviar, receber, transformar e carregar as linhas de update do SSB foi baseada em dois tutorias, já mencionados acima. Um primeiro que simula o envio de dados de telemetria [7] e, um segundo, onde é feita uma transformação de dados relativos a informações de filmes [6].

Em primeiro lugar foi necessário definir as configurações dos projetos, de modo que o produtor e consumidor saibam para que tópico devem enviar e subscrever os eventos, respetivamente. As configurações são muito semelhantes em ambas as partes, no entanto no produtor foram definidas num ficheiro de configuração e carregadas no início da execução da aplicação produtora - Figura 2 – e na aplicação consumidora as configurações foram definidas num método que é chamado no início da execução da aplicação - Figura 3. Sucintamente, estas configurações indicam onde está localizado o servidor/broker Kafka que contém os tópicos, através do endereço de IP – neste caso o localhost, porque o desenvolvimento e teste foi realizado sempre na mesma máquina – e o porto de escuta do broker. Para além disso, é definido o tópico para o qual se vai enviar os eventos, assim como, do lado do consumidor, é definido o nome do tópico que se pretende subscrever. Por fim, foi ainda definido o formato da key e do value transportados em cada evento, que neste caso são ambas do tipo `StringSerializer` – representam Strings.



```
# Kafka Producer properties 127.0.0.1
topic=ssb-topic
bootstrap.servers=localhost:9092
key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.apache.kafka.common.serialization.StringSerializer
```

Figura 2 - Algumas das propriedades de configuração do Produtor



```
public static Properties setProperties() {
    Properties properties = new Properties();
    properties.setProperty("topic", TOPIC_NAME);
    properties.setProperty("bootstrap.servers", "localhost:9092");

    properties.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    properties.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
}

// ...
}
```

Figura 3 - Algumas das propriedades de configuração do Consumidor definidas em Runtime

Feita a configuração foi necessário programar o envio e a receção das linhas de update, assim como todas as transformações necessárias de modo que, os dados possam ser inseridos e integrados na data warehouse SSB.

### 3.2 Implementação

Ao nível do produtor, pretendeu-se simular o envio de linhas de update com um intervalo de tempo pré-definido. Na prática, as linhas de update são lidas dos ficheiros em runtime e são armazenadas em memória para que, de seguida, sejam enviadas para um tópico. As linhas que pertencem ao mesmo cliente são enviadas, imediatamente, umas a seguir às outras, uma vez que representam o conjunto de artigos adquiridos numa compra. Cada compra – conjunto de linhas pertencentes ao mesmo cliente – é colocada no tópico com um intervalo de tempo predefinido pelo programador.

Em contraste, do lado do consumidor as transformações são realizadas em tempo real, o que significa que, assim que é recebida um evento com uma linha de update, a mesma é transformada e carregada/integrada na data warehouse. São realizadas várias transformações, desde o momento em que é recebida uma linha de update até que a mesma seja integrada. As primeiras transformações visam extrair todos os campos que estão na linha de update para que de seguida sejam tratados. Algumas destas transformações são explicadas em mais detalhe na subsecção 3.3. Feitas as



transformações necessárias, é feito o lookup dos dados das dimensões. No caso, de não existir um registo na dimensão Date correspondente às datas que compõem os dados da compra, são gerados os dados equivalentes a todos os dias do ano correspondente. Por sua vez, quando uma compra possui um novo cliente, um novo fornecedor, ou um novo produto, são criadas essas linhas nas respetivas dimensões e só depois é que é criada o registo na tabela *lineorder*. Alguns dos campos não são preenchidos, devido à falta de informação, sendo que nesses campos é colocado o valor “ND” que significa *not defined*. Por fim, feitas todas as transformações e associações com as tabelas envolvidas, a linha é carregada e integrada na data warehouse.

A Figura 4 sintetiza de uma forma gráfica o *workflow* da solução desenvolvida para resolver o problema.

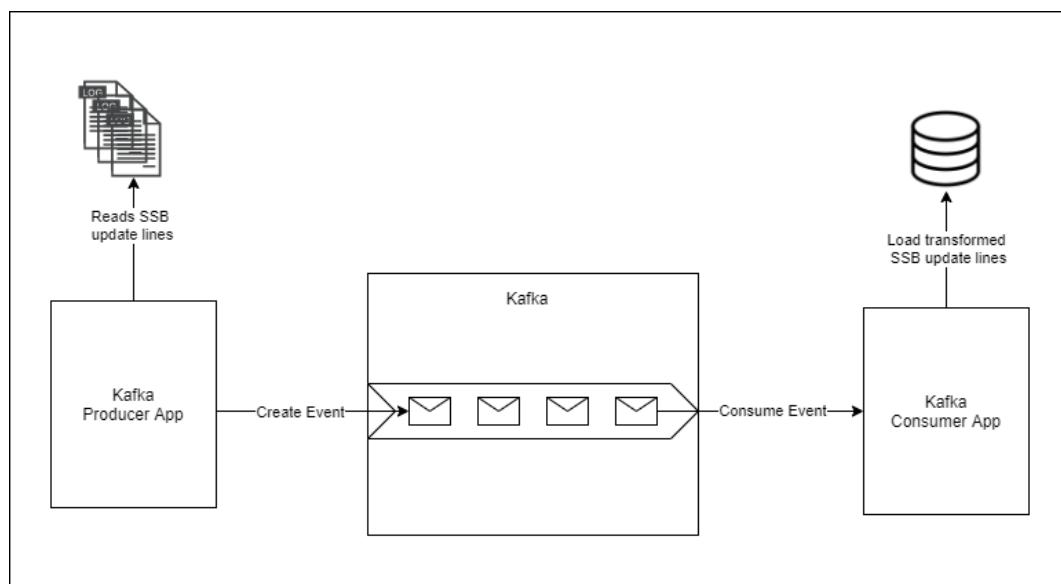


Figura 4 - Componentes da Solução Desenvolvida

As aplicações produtor e consumidor são totalmente independentes e não precisam de estar a correr em simultâneo. Uma vez criados eventos, estes ficam armazenados de forma segura no tópico até que sejam consumidos. Deste modo, o Kafka garante que nenhuma linha de update é perdida.

Para além disso é importante apresentar a organização das classes e do código. Com isso em mente, a Figura 5 apresenta os projetos, as packages e respetivas classes.



Figura 5 - Packages e classes dos projetos

Relativamente ao projeto Produtor Kafka, a classe *NewLine.java* representa a linha linda – que é armazenada numa String – e o nome do customer dessa linha de modo que seja possível saber se um conjunto de linhas pertencem a um mesmo customer no momento do envio das mesmas. A classe *ReadRecordsFromFile.java* é responsável por ler todas as linhas que estão nos quatro ficheiros fornecidos pelo docente. Já a classe *ProducerRunnable.java* implementa a interface *Runnable* e envia as linhas lidas ao consumidor, como é possível perceber pelo código apresentado na Figura 6.

```

public class ProducerRunnable implements Runnable{

    Logger logger = LoggerFactory.getLogger(producer.ProducerRunnable.class);
    KafkaProducer<String, String> producer;
    String topic;
    List<NewLine> records;

    /**
     * Set the properties in order to create the producer
     * @param propsPath - Path to the file that contains the properties
     * @param records
     */
    public ProducerRunnable(String propsPath, List<NewLine> records) {
        try {
            this.records = records;
            FileInputStream in = new FileInputStream(propsPath);
            Properties properties = new Properties();
            properties.load(in);
            this.topic = properties.getProperty("topic");
            in.close();
            producer = new KafkaProducer<String, String>(properties);
        } catch (IOException e) {
            logger.error("Kafka Producer properties file read failure!");
        }
    }

    /**
     * Thread responsible to send the lines to the consumer
     */
    @Override
    public void run() {
        boolean firstTime = true;
        String lastCostumer = "";
        int performanceTest = 0;
        for(NewLine rec : records){
            // If the customer is diferent... wait 1 second until send data related to ohter customer
            if(!lastCostumer.equals(rec.getCostumer_name()) && !firstTime ){
                try { Thread.sleep(1000); } catch (InterruptedException ex) { }
                lastCostumer = rec.getCostumer_name();
            }
            // Send the data to the Kafka consumer
            producer.send(new ProducerRecord<String, String>(topic, rec.getLine()));
            producer.flush();
            if(firstTime){
                lastCostumer = rec.getCostumer_name();
                firstTime = false;
            }
        }
    }

    /**
     * Shutdown producer
     */
    public void shutdown() {
        producer.close();
        logger.info("Kafka Producer has closed.");
    }
}

```

Figura 6 - Classe ProducerRunnable.java

Por fim, a classe *Producer.java* cria quatro threads que representam quatro produtores de linhas de update. Estes produtores vão estar em simultâneo a enviar linhas para o consumidor, como é possível observar na Figura 7.

```

public class Producer {

    public static void main(String[] args) {
        new Producer().run();
    }

    public void run(){

        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
        String propsPath = rootPath + "config.properties";

        //Get update data from files
        ReadRecordsFromFile obj1 = new ReadRecordsFromFile();
        ReadRecordsFromFile obj2 = new ReadRecordsFromFile();
        ReadRecordsFromFile obj3 = new ReadRecordsFromFile();
        ReadRecordsFromFile obj4 = new ReadRecordsFromFile();
        List<NewLine> rec1 = null;
        List<NewLine> rec2 = null;
        List<NewLine> rec3 = null;
        List<NewLine> rec4 = null;
        try {
            obj1.readFromFile(ReadRecordsFromFile.PATH_FILE_1);
            obj2.readFromFile(ReadRecordsFromFile.PATH_FILE_2);
            obj3.readFromFile(ReadRecordsFromFile.PATH_FILE_3);
            obj4.readFromFile(ReadRecordsFromFile.PATH_FILE_4);

            rec1 = obj1.getRecords();
            rec2 = obj2.getRecords();
            rec3 = obj3.getRecords();
            rec4 = obj4.getRecords();
        } catch (IOException ex) {
            return;
        }

        // Create a kafka producer 1
        ProducerRunnable producerRunnable1 = new ProducerRunnable(propsPath, rec1);
        Thread producerThread1 = new Thread(producerRunnable1);
        producerThread1.start();

        // Create a kafka producer 2
        ProducerRunnable producerRunnable2 = new ProducerRunnable(propsPath, rec2);
        Thread producerThread2 = new Thread(producerRunnable2);
        producerThread2.start();

        //Create a kafka producer 3
        ProducerRunnable producerRunnable3 = new ProducerRunnable(propsPath, rec3);
        Thread producerThread3 = new Thread(producerRunnable3);
        producerThread3.start();

        // Create a kafka producer 4
        ProducerRunnable producerRunnable4 = new ProducerRunnable(propsPath, rec4);
        Thread producerThread4 = new Thread(producerRunnable4);
        producerThread4.start();

        try {
            producerThread1.join();
            producerThread2.join();
            producerThread3.join();
            producerThread4.join();
        } catch (InterruptedException ex) {
            Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}

```

Figura 7 - Classe Producer.java

Relativamente ao projeto do Consumidor Kafka, a package *calendar* agrega os métodos que realizam a transformação de uma data recebida num evento Kafka numa data no formato do SSB, gerando os restantes campos. A package *db.communication* contém apenas uma classe que é responsável pela comunicação com a base de dados – realiza as operações de leitura e escrita na base de dados. Já a package *db.tables* contém objetos que representam as tabelas da data warehouse de modo a que se possa fazer um mapeamento tabela <-> objeto. Por fim, a package *consumer* contém o motor do

consumidor Kafka. A classe *Record.java* representa os campos de uma linha de update recebida, já com os respectivos dados transformados. A Figura 8 apresenta as propriedades da classe.



```
public class Record {  
  
    // ... more information ...  
  
    // Fields from the file that matches with the SSB DB fields  
    private String p_name;           // Table Part  
    private String p_color;         // Table Part  
    private String c_name;          // Table Customer  
    private String d_date;          // Table Date  
    private String s_name;          // Table Supplier  
    private String lo_orderpriority; // Table Lineorder  
    private String lo_shippriority; // Table Lineorder  
    private int lo_quantity;        // Table Lineorder  
    private int lo_extendedprice;    // Table Lineorder  
    private int lo_ordertotalprice;  // Table Lineorder  
    private int lo_discount;        // Table Lineorder  
    private int lo_supplycost;      // Table Lineorder  
    private int lo_tax;             // Table Lineorder  
    private String lo_shipmode;     // Table Lineorder  
  
    //... continue ...  
}
```

Figura 8 - Propriedades da classe *Record.java*

A classe *Consumer.java* é responsável por atender os pedidos e coordenar o processo de ETL. No construtor da classe são inicializadas as propriedades da classe e são carregados para memória os dados das tabelas mais pequenas da data warehouse, como mostra a Figura 9.

```

public class Consumer {

    public static final String TOPIC_NAME = "ssb-topic";           // Topic Name

    private final DbCommunication ssbLT;                          // DataBase communication
    private HashMap<String, Supplier> supplierTable;              // Supplier Table in memory
    private HashMap<String, Customer> customerTable;              // Customer Table in memory
    private HashMap<String, Part> partTable;                      // Part Table in memory
    private HashMap<String, Date> dateTable;                     // Date Table in memory
    private int lo_orderKey;                                       // Last OrderKey
    private int p_partkey;                                         // Last PartKey
    private int c_custkey;                                         // Last CustKey
    private int s_supkey;                                         // Last SupKey

    public Consumer() {
        ssbLT = new DbCommunication();
        supplierTable = ssbLT.getSupplierTable();
        customerTable = ssbLT.getCustomerTable();
        partTable = ssbLT.getPartTable();
        dateTable = ssbLT.getDateTable();
        lo_orderKey = ssbLT.getLastLineorderKey();
        p_partkey = ssbLT.getLastPartKey();

        //Get last Customer ID Value
        final Set<Entry<String, Customer>> cMapValues = customerTable.entrySet();
        final Entry<String, Customer>[] cArray = new Entry[customerTable.size()];
        cMapValues.toArray(cArray);
        c_custkey = cArray[customerTable.size()-1].getValue().getC_custkey();

        //Get last Supplier ID Value
        final Set<Entry<String, Supplier>> sMapValues = supplierTable.entrySet();
        final Entry<String, Supplier>[] sArray = new Entry[supplierTable.size()];
        sMapValues.toArray(sArray);
        s_supkey = sArray[supplierTable.size()-1].getValue().getS_supkey();
    }

    //... continue ...
}

```

Figura 9 - Propriedades e construtor da classe *Consumer.java*

Tal como já foi referido, a classe *Consumer.java* é o motor do consumidor Kafka e, portanto, está sempre à escuta de novos eventos no método *run* – Figura 10. Este método, assim que recebe uma linha de update, começa por a transformar, depois realiza os lookups necessários nas outras tabelas e, antes de colocar a o novo registo na tabela *lineorder*, preenche os restantes campos que não foram preenchidos anteriormente.

```

public class Consumer {

    // .... more before ...

    public void run() throws SQLException {

        Transformation tf = new Transformation();
        Logger logger = java.util.logging.Logger.getLogger(consumer.Consumer.class.getName());
        String lastCustomer = "";
        boolean firstTime = true;
        int sameCustomerCounter = 0;
        boolean sameCustomerFlag = false;

        // create kafka consumer
        KafkaConsumer<String, String> consumer = null;
        try {
            consumer = createConsumer();
        } catch (IOException ex) {
            logger.log(Level.SEVERE, null, ex);
        }

        logger.info("Ready to work;");
        // Receive data from the producer and save into the database
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                // logger.info("Consumer reads: " + record.value());
                logger.info("");
                // Transform data received into fields
                Record r = tf.transformLineToRecord(record.value());
                // Lookup data
                Lineorder newLine = lookupData(r);
                // Feel the other fields
                newLine = finishingLineorderCreation(newLine, r);
                // Verify if the order belongs to the same customer
                if (!r.getC_name().equals(lastCustomer) && !firstTime) {
                    lastCustomer = r.getC_name();
                    sameCustomerCounter = 0;
                    sameCustomerFlag = false;
                }
                // Set field lo_linenumbr and lo_orderkey
                newLine.setLo_linenumbr(++sameCustomerCounter);
                if (!sameCustomerFlag) {
                    newLine.setLo_orderkey(++lo_orderKey);
                } else {
                    newLine.setLo_orderkey(lo_orderKey);
                }
                if (firstTime) {
                    firstTime = false;
                    lastCustomer = r.getC_name();
                }
                // Write to DataBase
                ssbLT.write(newLine.sqlInsert());
                lastCustomer = r.getC_name();
                sameCustomerFlag = true;
            }

            if (records.count() > 0) {
                consumer.commitSync();
            }
        } // End While

        //... continue ...
    }
}

```

Figura 10 - Método run da classe Consumer.java

Por fim, tal como foi referido acima, os dois passos da transformação da linha de update passa pelo *lookup* de dados noutras tabelas. Com isso em mente, é realizada uma pesquisa em memória através de um identificador de cada uma das tabelas. Quando ainda não existe um determinado registo numa tabela, o mesmo é criado em tempo real. A Figura 11 mostra o processo de *lookup* nas tabelas *date*, *supplier*, *customer* e *part*.

```

public class Consumer {

    // .... more before ...

    /**
     * Lookup for the data in the others tables and create the registers if they
     * don't exists
     *
     * @param r
     * @return
     */
    public Lineorder lookupData(Record r) {
        Lineorder newLine = new Lineorder();
        boolean createPart = true;
        boolean createCostumer = true;
        boolean createSupplier = true;

        // Data related with Part Table
        if (partTable.containsKey(r.getP_name() + r.getP_color())) {
            newLine.setLo_partkey(partTable.get(r.getP_name()+r.getP_color()).getP_partkey());
            createPart = false;
        }
        // Part registry dont exists in the DB. Create a new one
        if (createPart) {
            Part newPart = new Part(++this.p_partkey, r.getP_name(), r.getP_color());
            newPart.generateRandomData();
            ssbLT.write(newPart.toSqlInsert());
            newLine.setLo_partkey(p_partkey);
            this.partTable.put(newPart.getP_name() + newPart.getP_color(), newPart);
        }

        // Data related with Supplier Table
        if (supplierTable.containsKey(r.getS_name())) {
            newLine.setLo_supkey(supplierTable.get(r.getS_name()).getS_supkey());
            createSupplier = false;
        }
        // Supplier dont exists in the DB. Create a new one
        if (createSupplier) {
            Supplier newSupp = new Supplier(++this.s_supkey);
            ssbLT.write(newSupp.toSqlInsert());
            newLine.setLo_supkey(this.s_supkey);
            this.supplierTable.put(newSupp.getS_name(), newSupp);
        }

        // Data related with Costumer Table
        if (customerTable.containsKey(r.getC_name())) {
            newLine.setLo_custkey(customerTable.get(r.getC_name()).getC_custkey());
            createCostumer = false;
        }
        // Customer dont exists in the DB. Create a new one
        if (createCostumer) {
            Customer newCust = new Customer(++this.c_custkey);
            ssbLT.write(newCust.toSqlInsert());
            newLine.setLo_custkey(this.c_custkey);
            this.customerTable.put(newCust.getC_name(), newCust);
        }

        // Data related with Date Table
        if (dateTable.containsKey(r.getD_date())) {
            newLine.setLo_orderdate(dateTable.get(r.getD_date()).getD_datekey());
            return newLine;
        }
        //If the date dont exists
        createDataInTheDataBase(r.getD_date());
        // Data related with Date Table
        if (dateTable.containsKey(r.getD_date())) {
            newLine.setLo_orderdate(dateTable.get(r.getD_date()).getD_datekey());
        }
        return newLine;
    }

    //... continue ...
}

```

Figura 11 - Lookup dos dados nas tabelas auxiliares

### 3.3 Transformação dos dados

As transformações efetuadas nos dados brutos extraídos pelo Kafka Consumer do tópico, facilita a manipulação e preparação dos mesmos de modo que seja feita uma integração com a data warehouse SSB.



### 3.3.1 Transformação – Customer e Supplier

A transformação para a dimensão *Customer* e *Supplier* do SSB foi muito simples visto que apenas foi interpretado um valor de todos os disponíveis para cada uma das dimensões. Notamos que os dados brutos já nos concediam o identificador do cliente (*c\_custkey*) e o identificador do fornecedor (*s\_suppkey*) com a particularidade de não ter os 9 dígitos pré-definidos:

	Customer	Supplier
Valor bruto	Customer#16067	Supplier#1275
Valor pretendido	Customer#000016067	Supplier #000001275

Tabela 2 - Comparação entre o valor bruto e o valor de um registo no SSB

Desta forma, foi apenas necessário fazer um split entre o prefixo (Customer/Supplier) e os dígitos, sendo o delimitador o símbolo # e aplicar a função *String.format("%09d", identificador)*; Ao realizar esta pequena transformação só restava voltar a concatenar a separação.

### 3.3.2 Transformação – Part

As transformações dos dados brutos para a dimensão Part também não apresentaram nenhuma complexidade dado que, a maioria estão diretamente relacionados com valores random, como por exemplo *p\_mfgr* e o *p\_category*, na qual foi apenas necessário usar um prefixo *MFGR#* e concatenar com um número random entre 1 e 5, em ambos os casos. O atributo *p\_brand1* é gerado a partir do valor *p\_category* e um valor random entre 1 e 40. O atributo *p\_type* e *p\_container*, por simplicidade optamos por definir que o seu valor seria “ND” e por último o atributo *p\_size* tem um valor random entre 1 e 5. Tendo-se realizado estas transformações, estamos em condições de criar um registo completo nesta dimensão seguindo a estrutura do SSB.

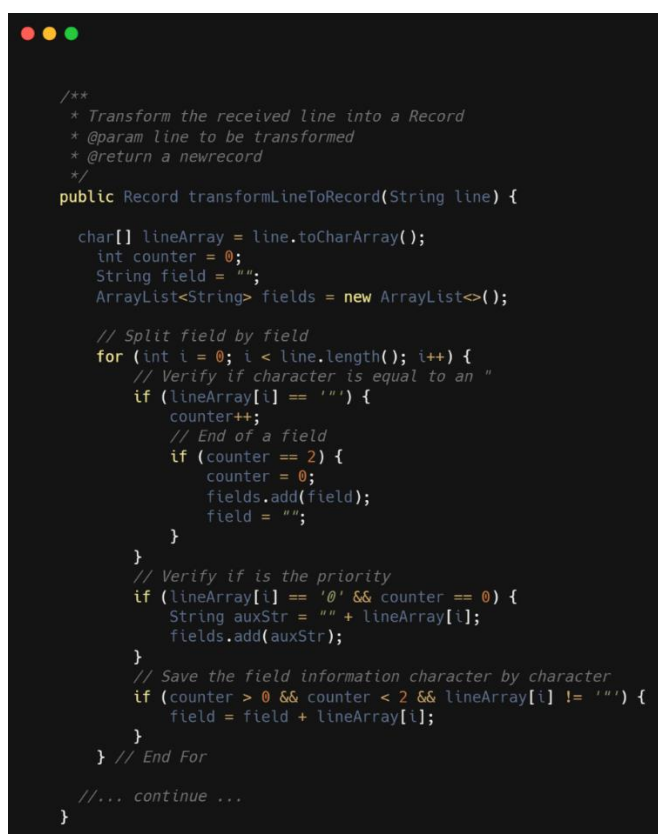
### 3.3.3 Transformação – Date

As transformações dos dados brutos para a dimensão Date não demonstraram ser complexas dado que foram utilizadas bibliotecas para a manipulação destes dados como o *java.util.Calendar* e o *java.util.GregorianCalendar*. Foi notado que a partir dos dados fornecidos pela fonte, podíamos ter muita informação como o número do dia numa semana, num mês e num ano, entre outras informações úteis suportadas pela biblioteca [9].

De modo a preparar o SSB para updates futuros, automatizamos as transformações para esta dimensão para um ano inteiro, isto significa que ao receber um determinado ano as transformações deverão ocorrer para os 365 dias desse ano.

Quanto a transformação propriamente dita, muitos dos atributos eram gerados a partir de concatenação do dia, mês e ano, como o identificador *d\_datekey*, *d\_date*, *d\_yearmonthnum* e *d\_yearmonth*. Os restantes atributos, em exceção a estação do ano *d\_sellingseason*, foram gerados a partir dos métodos das bibliotecas acima referidas.

Todas estas transformações apresentadas e explicadas acima são realizadas no método *transformeLineToRecord* na classe *Transformation.java*. Na Figura 12 é possível observar um pedaço desse código.



```
/**
 * Transform the received line into a Record
 * @param line to be transformed
 * @return a new record
 */
public Record transformLineToRecord(String line) {

    char[] lineArray = line.toCharArray();
    int counter = 0;
    String field = "";
    ArrayList<String> fields = new ArrayList<>();

    // Split field by field
    for (int i = 0; i < line.length(); i++) {
        // Verify if character is equal to an "
        if (lineArray[i] == '"') {
            counter++;
            // End of a field
            if (counter == 2) {
                counter = 0;
                fields.add(field);
                field = "";
            }
        }
        // Verify if is the priority
        if (lineArray[i] == '@' && counter == 0) {
            String auxStr = "" + lineArray[i];
            fields.add(auxStr);
        }
        // Save the field information character by character
        if (counter > 0 && counter < 2 && lineArray[i] != '"') {
            field = field + lineArray[i];
        }
    } // End For

    //... continue ...
}
```

Figura 12 - Parte do código responsável pela transformação da linha de update

### 3.4 Performance e capacidades

Após a configuração e desenvolvimento da solução, torna-se fundamental avaliar a performance da mesma. Para isso foram realizadas algumas medições e, por sua vez, alguns cálculos de modo a avaliar o tempo de processamento das linhas de update, assim como a capacidade do trabalho do Kafka Consumer.

Antes da apresentação dos resultados obtidos, são apresentadas na Tabela 3 as características da máquina onde foram realizados os testes. Uma vez que o produtor e o consumidor estão a executar na mesma máquina, é expectável que a performance de ambos seja afeta. No entanto, ficam registados os valores para que se possam retirar algumas conclusões.

<b>Processador</b>	Intel® Core™ i7 7700HQ
<b>Nº de Núcleos</b>	4 Núcleos 4 Threads
<b>RAM</b>	16.0 GB
<b>Disco</b>	Samsung SSD 970 EVO Plus 1TB
<b>Sistema Operativo</b>	Windows 10 Home 64bits
<b>Postgres</b>	Versão 10

*Tabela 3 - Configuração da máquina onde foram realizados os testes*

Numa primeira fase de testes, procurou-se saber quanto tempo é que o consumidor Kafka demorava a receber, transformar e carregar as linhas de update de cada um dos ficheiros em separado. Para isso, foram realizadas 10 repetições de envio das linhas de update, para cada um dos ficheiros. Feitas essas repetições, rapidamente se percebeu que o consumidor Kafka tinha um comportamento muito homogéneo, o que significa, que os tempos de tratamento das linhas era muito semelhante a cada execução. Na Tabela 4 ficam registados os valores obtidos após a execução dos testes com duas versões do consumidor. Uma primeira versão implementada com ArrayList e uma segunda com HashMap, no momento de fazer lookup a tabelas em memória. Já na Figura 13 são apresentados os tempos de tratamento de cada um dos ficheiros para cada um dos casos referidos.

<b>Ficheiro</b>	<b>Nº linhas</b>	<b>Tempo de processamento com ArrayList</b>	<b>Tempo de processamento com HashMap</b>
1	3555	17m22s	12s
2	257	1m25s	3s
3	347	1m37s	3s
4	3502	15m11s	9s

*Tabela 4 - Tempo de processamento dos ficheiros*

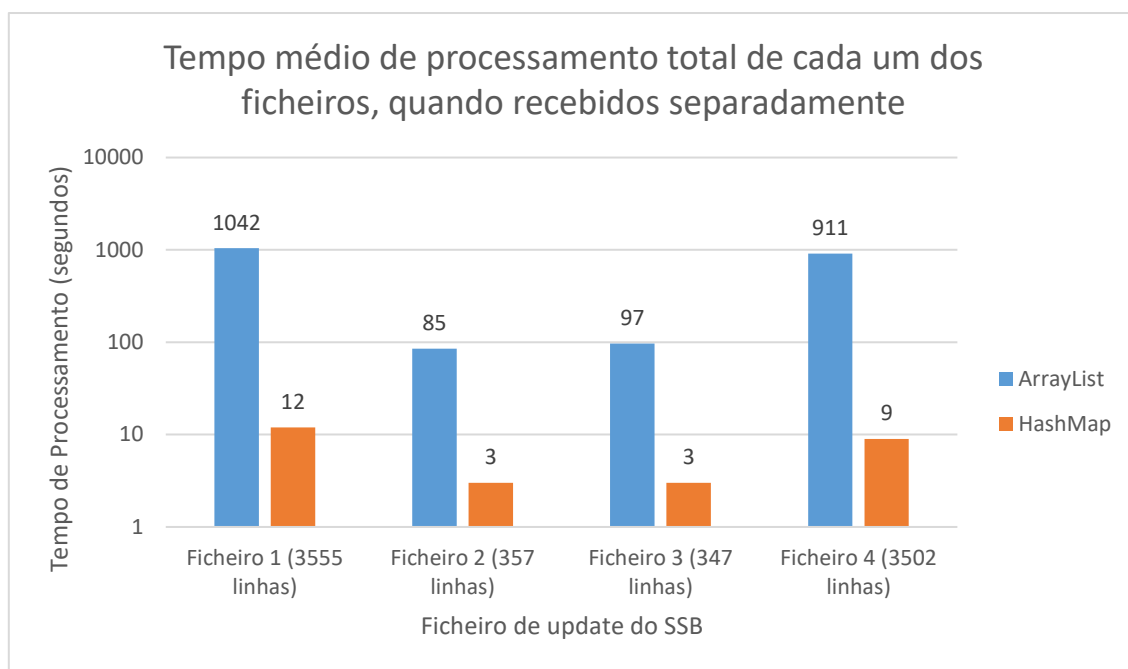


Figura 13 - Tempo de processamento total de cada um dos ficheiros, quando recebidos separadamente

De seguida, com base nos resultados obtidos, calculou-se o número de linhas processadas por segundo – também conhecido por *throughput*. Analisando a gráfico apresentado na Figura 14 é possível concluir que o número de linhas recebidas não influência diretamente a resposta do consumidor Kafka, uma vez que para ambos os ficheiros apresentam valores muito próximos – relativamente ao número de linhas processadas em um segundo.

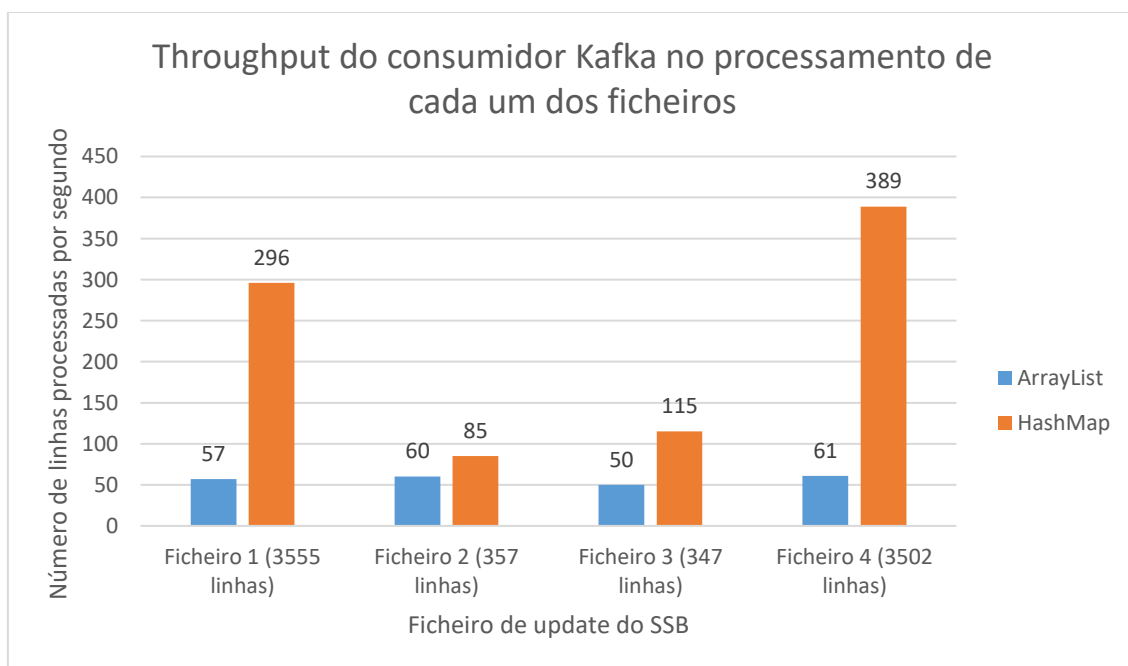


Figura 14 - Throughput do consumidor Kafka no processamento de cada um dos ficheiros

Tendo em conta os resultados obtidos na Tabela 4 e dependendo do tipo de negócio que esteja a ser considerado, é importante avaliar se estes tempos satisfazem as necessidades de uma empresa. Assumindo o caso de um Supermercado (relacionando com o SSB), podemos assumir que o processamento das vendas possa ser realizado entre as 00h00 e as 06h00 com a implementação usando HashMap para a realização de lookup tables. Desta forma, computou-se o número máximo de linhas que o consumidor Kafka é capaz de consumir num intervalo de, como se pode observar na Tabela 5.

Tempo disponibilizado (horas)	Nº linhas processadas
2	2,462,400
3	3,693,600
4	4,924,800
6	7,387,200
8	9,849,600

Tabela 5 – Capacidade de processamento do consumidor Kafka num determinado intervalo de tempo

Como se pode verificar, o número de registos processados em 6 horas

Uma forma de melhorar a performance dos registos processados passará, por exemplo, explorar o paralelismo, de forma a garantir que os dados são processados em sistemas distribuídos. Para tal, poder-se-iam usar ferramentas como *Spark* ou *Hadoop* com *MapReduce* facilitando assim o processamento distribuído e de grandes quantidades de dados. Também é importante referir que as máquinas que processam os dados serão máquinas próprias, desenhadas propositadamente para o efeito, assumimos que os nossos laptops não tenham CPU suficiente para assumir este *Data Overload*.

Numa segunda e última fase, procurou-se saber quanto tempo demoraria o consumidor Kafka a processar todas as linhas, dos quatro ficheiros em simultâneo. Para isso foram criadas quatro *threads* que simulam quatro produtores, onde cada um deles envia constantemente uma linha a seguir à outra. Neste cenário, o consumidor demorou em média 2 minutos e 13 segundos a receber, transformar e carregar as linhas de update na data warehouse. Consecutivamente, dá um *throughput* média de 58 linhas carregadas por segundos. Com isto, é possível concluir que a performance do consumidor Kafka, não é influenciada pelo número de pedidos que são recebidos, uma vez que o *throughput* médio é muito constante sejam recebidas muitas ou poucas linhas de update.

Por fim, concluiu-se que a performance não é afetada pelo número de eventos que chegam ao consumidor Kafka, mas sim pelo tipo de evento. Caso seja recebido um evento que contenha a compra de um artigo (tabela part) que não esteja registado na tabela correspondente, é necessário criar essa linha e isso prejudica em até 15 vezes mais o tempo de processamento, como já foi referido anteriormente.

Feita a avaliação de performance do consumidor Kafka, a solução final irá enviar compras relativas a um cliente de segundo a segundo. Ou seja, quando as linhas lidas dos ficheiros pertencerem ao mesmo cliente são enviadas sem nenhum intervalo de tempo entre elas, face que quando uma linha não corresponde ao cliente anterior, é feita uma pause de um segundo até que seja enviada.

Esta página foi deixada propositadamente em branco.

## 4. Considerações Finais

Dado por terminado o projeto é necessário fazer um balanço dos objetivos alcançados, assim como as competências adquiridas.

As aulas práticas de introdução ao Kafka e principalmente com o esclarecimento e ajuda do docente podemos afirmar que de uma forma geral, o desenvolvimento da solução com a framework Kafka e a transformação dos dados brutos foram implementados com sucesso.

Com isto em mente é importante realçar que os objetivos de aprendizagem foram alcançados. Nomeadamente, foram conhecidas um conjunto de ferramentas novas como o Kafka, que permitiu conhecer algumas das suas características e limitações. Foram também conhecidas as dificuldades que podem ser encontradas com o processo de ETL, sendo importante mencionar que não foram utilizadas ferramentas típicas para este processo como o Pentaho, Talend ou o Spark.

Relativamente à framework Kafka é possível afirmar que este possui ferramentas robustas para o desenvolvimento de aplicações que tenham uma comunicação baseada em stream de eventos em tempo real. Estas aplicações, por sua vez têm ao seu dispor uma série de recursos úteis para o processamento e transporte de informação. Contudo, na fase de transformação de dados é apresentada uma complexidade elevada pois, o Kafka não foi idealizado para este tipo de tarefa. É necessário realizar manipulações muito complexas entre as aplicações para alcançar uma transformação dos dados que facilmente poderia ser feita em Spark, por exemplo.

Em relação ao processo de ETL, típico em sistemas de gestão de dados, permitiu-nos compreender a complexidade com que os dados brutos podem aparecer, tornando este processo muito delicado, no sentido de podermos comprometer a performance da Data Warehouse. Devemos ter especial cuidado na construção de queries de forma a evitar que as pesquisas se tornem “pesadas” e possam exceder o tempo tolerado pelos utilizadores finais, isto porque as Data Warehouse tipicamente têm centenas de milhares de registos, contudo, não nos preocupamos muito com este detalhe visto estarmos a usar o SSB, um exemplo típico e construído de forma a ter uma performance adequada. Outro ponto importante é a construção de metadados, realizados no projeto. A construção deve ser feita de forma cuidada e que respeite à estrutura definida inicialmente (datatypes, formatação, constituição), visto que transformações inadequadas possam comprometer severamente a estrutura da base de dados, assim como do projeto. Por último, ao realizarmos as inserções e updates, devemos ter especial atenção nas condições que se realizam, de modo a evitar redundâncias nos dados



podendo comprometer seriamente o tamanho da data warehouse, as pesquisas e futuramente a análise dos dados por parte dos analistas.

Consideramos, ainda, que as nossas soluções podem não estar da forma mais eficiente possível, pois a utilização de uma ferramenta de ETL – como o Spark – provavelmente seria mais eficiente. Contudo, tentamos procurar sempre soluções ajustadas aquilo que nos é exigido e esperado no âmbito da disciplina de Sistemas de Gestão de Dados.

## Referências

- [1] O. Pat, O. Betty and C. Xuedong, "Star Schema Benchmark," vol. Revision 3, 2009.
- [2] Apache, "APACHE KAFKA," [Online]. Available: <https://kafka.apache.org/>. [Accessed 23 04 2021].
- [3] APACHE, "Documentation," [Online]. Available: <https://kafka.apache.org/documentation/#gettingStarted>. [Accessed 23 04 2021].
- [4] M. Stéphane, "The Kafka API Battle: Producer vs Consumer vs Kafka Connect vs Kafka Streams vs KSQL !," 29 10 2018. [Online]. Available: <https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql-ef584274c1e>. [Accessed 24 04 2021].
- [5] P. Priya, "Differences Between Kafka vs Spark," [Online]. Available: <https://www.educba.com/kafka-vs-spark/>. [Accessed 24 04 2021].
- [6] Confluent, "How to transform a stream of events," [Online]. Available: <https://kafka-tutorials.confluent.io/transform-a-stream-of-events/kafka.html>. [Accessed 28 04 2021].
- [7] musa-atlihan, "Simulate IoT sensor, use Kafka to process data in real-time, save to Elasticsearch.," [Online]. Available: <https://github.com/musa-atlihan/IoT-voltage>. [Accessed 28 04 2021].
- [8] ruilinhares, "Kafka-Streams-Project," [Online]. Available: <https://github.com/ruilinhares/Kafka-Streams-Project>. [Accessed 28 04 2021].
- [9] Oracle, "Class Calendar," [Online]. Available: <https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html>. [Accessed 29 04 2021].