

# SKRIPSI

## STUDI DAN IMPLEMENTASI SPARK STREAMING UNTUK MENGUMPULKAN *BIG DATA STREAM*



Muhammad Ravi

NPM: 2016730041

PROGRAM STUDI TEKNIK INFORMATIKA  
FAKULTAS TEKNOLOGI INFORMASI DAN SAINS  
UNIVERSITAS KATOLIK PARAHYANGAN  
2019



# UNDERGRADUATE THESIS

## STUDY AND IMPLEMENTATION OF SPARK STREAMING TO COLLECT BIG DATA STREAM



Muhammad Ravi

NPM: 2016730041

DEPARTMENT OF INFORMATICS  
FACULTY OF INFORMATION TECHNOLOGY AND SCIENCES  
PARAHYANGAN CATHOLIC UNIVERSITY  
2019



# LEMBAR PENGESAHAN

## STUDI DAN IMPLEMENTASI SPARK STREAMING UNTUK MENGUMPULKAN *BIG DATA STREAM*

Muhammad Ravi

NPM: 2016730041

Bandung, 6 November 2019

Menyetujui,

Pembimbing Utama

Pembimbing Pendamping

Dr. Veronica Sri Moertini

Ketua Tim Penguji

Anggota Tim Penguji

Mengetahui,

Ketua Program Studi

Mariskha Tri Adithia, P.D.Eng



## PERNYATAAN

Dengan ini saya yang bertandatangan di bawah ini menyatakan bahwa skripsi dengan judul:

### **STUDI DAN IMPLEMENTASI SPARK STREAMING UNTUK MENGUMPULKAN *BIG DATA STREAM***

adalah benar-benar karya saya sendiri, dan saya tidak melakukan penjiplakan atau pengutipan dengan cara-cara yang tidak sesuai dengan etika keilmuan yang berlaku dalam masyarakat keilmuan.

Atas pernyataan ini, saya siap menanggung segala risiko dan sanksi yang dijatuhkan kepada saya, apabila di kemudian hari ditemukan adanya pelanggaran terhadap etika keilmuan dalam karya saya, atau jika ada tuntutan formal atau non-formal dari pihak lain berkaitan dengan keaslian karya saya ini.

Dinyatakan di Bandung,  
Tanggal 6 November 2019

Meterai Rp. 6000
---------------------

Muhammad Ravi  
NPM: 2016730041





## ABSTRAK

«Tuliskan abstrak anda di sini, dalam bahasa Indonesia»

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

**Kata-kata kunci:** «Tuliskan di sini kata-kata kunci yang anda gunakan, dalam bahasa Indonesia»



## ABSTRACT

«Tuliskan abstrak anda di sini, dalam bahasa Inggris»

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

**Keywords:** «Tuliskan di sini kata-kata kunci yang anda gunakan, dalam bahasa Inggris»



*«kepada siapa anda mempersembahkan skripsi ini...?»*



## KATA PENGANTAR

«Tuliskan kata pengantar dari anda di sini ...»

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Bandung, November 2019

Penulis





# DAFTAR ISI

<b>KATA PENGANTAR</b>	<b>xv</b>
<b>DAFTAR ISI</b>	<b>xvii</b>
<b>DAFTAR GAMBAR</b>	<b>xix</b>
<b>DAFTAR TABEL</b>	<b>xxi</b>
<b>1 PENDAHULUAN</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Rumusan Masalah . . . . .	1
1.3 Tujuan . . . . .	1
1.4 Batasan Masalah . . . . .	2
1.5 Metodologi . . . . .	2
1.6 Sistematika Pembahasan . . . . .	2
<b>2 LANDASAN TEORI</b>	<b>5</b>
2.1 Big Data . . . . .	5
2.2 Big Data Stream . . . . .	6
2.2.1 Pengertian Stream Processing . . . . .	7
2.2.2 Pemodelan Stream Processing . . . . .	7
2.2.3 Pola Pemrosesan Data Stream . . . . .	8
2.2.4 Arsitektur Stream Processing . . . . .	12
2.3 Sistem Terdistribusi Hadoop . . . . .	13
2.4 Scala . . . . .	17
2.4.1 variable . . . . .	18
2.4.2 Fungsi . . . . .	18
2.4.3 Kelas . . . . .	19
2.4.4 Kelas option . . . . .	20
2.4.5 Trait . . . . .	21
2.4.6 Tuple . . . . .	21
2.4.7 Koleksi . . . . .	22
2.4.8 Percabangan . . . . .	23
2.4.9 Pengulangan . . . . .	24
2.4.10 Operasi Baca Tulis File . . . . .	25
2.5 Sistem Terdistribusi Spark . . . . .	26
2.5.1 Susunan Spark . . . . .	27
2.5.2 Application Programming Interface (API) Spark . . . . .	28
2.5.3 Arsitektur Apache Spark . . . . .	29
2.5.4 Spark Streaming . . . . .	30
2.6 Input Sources . . . . .	35
2.6.1 Twitter API . . . . .	35
2.6.2 Kafka . . . . .	36

<b>3</b>	<b>STUDI EKSPLORASI</b>	<b>41</b>
3.1	Konfigurasi Kluster . . . . .	41
3.1.1	Konfigurasi Hadoop . . . . .	41
3.1.2	Konfigurasi Spark . . . . .	42
3.2	Konfigurasi API dan Data Collector . . . . .	43
3.2.1	Konfigurasi TCP Socket . . . . .	43
3.2.2	Konfigurasi Twitter API . . . . .	43
3.2.3	Konfigurasi Kafka . . . . .	44
3.3	Studi Eksplorasi . . . . .	46
3.3.1	Eksplorasi Spark Streaming dengan TCP Socket . . . . .	46
3.3.2	Eksplorasi dengan Twitter API . . . . .	48
<b>4</b>	<b>ANALISIS DAN PERANCANGAN</b>	<b>51</b>
4.1	Analisis Perangkat Lunak . . . . .	51
4.1.1	Analisis Set Data . . . . .	51
4.1.2	Analisis Masukan dan Keluaran . . . . .	51
4.2	Perancangan Penghitung Hashtag . . . . .	51
<b>A</b>	<b>KODE PROGRAM</b>	<b>53</b>
<b>B</b>	<b>HASIL EKSPERIMEN</b>	<b>57</b>

## DAFTAR GAMBAR

2.1	Gambar Pemetaan <i>Time-Domain</i> . . . . .	8
2.2	Gambar <i>fixed-window</i> . . . . .	9
2.3	Gambar <i>Session-batch</i> . . . . .	9
2.4	Gambar <i>Filtering-unbounded-data</i> . . . . .	10
2.5	Gambar <i>inner-join</i> . . . . .	10
2.6	Gambar <i>approximation-algorithm</i> . . . . .	10
2.7	Gambar <i>Windowing</i> . . . . .	11
2.8	Gambar Arsitektur <i>Lambda</i> . . . . .	12
2.9	Gambar Arsitektur <i>Hadoop</i> . . . . .	14
2.10	Gambar Arsitektur HDFS . . . . .	15
2.11	Gambar Arsitektur MapReduce . . . . .	16
2.12	Gambar Proses MapReduce . . . . .	17
2.13	Gambar <i>Spark unified stack</i> . . . . .	27
2.14	Gambar Arsitektur <i>Spark</i> . . . . .	29
2.15	Gambar Arsitektur <i>Spark Streaming</i> . . . . .	31
2.16	Gambar Arsitektur <i>Spark Streaming</i> . . . . .	31
2.17	Gambar Alur <i>Dstream</i> . . . . .	32
2.18	Gambar mengubah <i>Data Stream</i> dari lines ke words . . . . .	32
2.19	Gambar eksekusi <i>Spark Streaming</i> pada komponen <i>Spark</i> . . . . .	33
2.20	Gambar cara kerja <i>Windowed Transformation</i> . . . . .	34
2.21	Gambar Twitter Object . . . . .	35
2.22	Gambar Twitter Object . . . . .	35
2.23	Gambar Twitter Object . . . . .	36
2.24	Gambar Publisher/Subscriber . . . . .	36
2.25	Gambar Topic pada Kafka . . . . .	37
2.26	Gambar stream topic . . . . .	38
2.27	Gambar Kafka Broker . . . . .	38
3.1	Gambar Instalasi Java . . . . .	42
3.2	Gambar HADOO PHOME . . . . .	42
3.3	Gambar Spark . . . . .	43
3.4	Gambar Twitter Tokens . . . . .	44
3.5	Gambar Instalasi Java . . . . .	45
3.6	Gambar ZOOKEEPER _HOME . . . . .	45
3.7	Gambar menjalankan server . . . . .	45
3.8	Gambar menjalankan topic . . . . .	46
3.9	Gambar <i>consumer-producer</i> . . . . .	46
3.10	Gambar File input . . . . .	46
3.11	Gambar pengaturan spark streaming . . . . .	47
3.12	Gambar perhitungan url . . . . .	47
3.13	Gambar File input . . . . .	47
3.14	Gambar Output Web Log . . . . .	48

3.15	Gambar Setup Twitter . . . . .	48
3.16	Gambar Setup Spark Streaming . . . . .	48
3.17	Gambar transformasi twitter . . . . .	49
3.18	Gambar folder output . . . . .	49
3.19	Gambar file output . . . . .	49
3.20	Gambar file output . . . . .	49
4.1	Twitter Obejct . . . . .	52
B.1	Hasil 1 . . . . .	57
B.2	Hasil 2 . . . . .	57
B.3	Hasil 3 . . . . .	57
B.4	Hasil 4 . . . . .	57

## DAFTAR TABEL



# BAB 1

## PENDAHULUAN

### 1.1 Latar Belakang

Dalam beberapa tahun terakhir, Perkembangan data melonjak secara cepat. Hal ini disebabkan karena semakin banyak orang yang terhubung secara digital. Website yang diakses, media sosial yang dijelajahi, atau sensor-sensor dari barang-barang elektronik yang terhubung ke internet semua meninggalkan jejak digital berupa data. Data yang terakumulasi ini berukuran besar dengan format yang bervariasi dan berkembang dengan sangat cepat.

Jika data yang terakumulasi tersebut diolah dan dianalisis, banyak informasi-informasi bermanfaat yang bisa didapat. Contohnya, data bisa menjadi bahan pertimbangan untuk pengambilan keputusan bisnis. Tetap, Tidak semua data memiliki nilai dan sifat yang sama. Ada data yang memiliki nilai lebih ketika bisa langsung dianalisis ketika didapatkan. Kebutuhan untuk langsung mendapatkan dan menganalisis data secara *real-time* menjadi sangat penting. Selain itu, teknik pengumpulan data yang digunakan untuk pola data yang datang secara terus menerus berbeda dengan teknik yang digunakan untuk mengumpulkan dan mengolah data biasa. *Big Data* yang perlu diakses secara *real-time* adalah page views pada sebuah website, sensor pada IoT (*Internet of Things*).

Selain itu kebutuhan untuk mengolah data dengan cepat semakin penting karena nilai suatu data cenderung menurun secara eksponensial seiring bertambahnya waktu. Banyak Perusahaan dan Organisasi yang membutuhkan data untuk diolah secara cepat. Semakin cepat data bisa diambil, dianalisis, dimanipulasi, dan semakin banyak throughput yang bisa dihasilkan maka sebuah organisasi akan lebih *agile* dan responsif. Semakin sedikit waktu yang digunakan untuk ETL (*Extract, Load, Transform*) pekerjaan akan semakin fokus untuk melakukan analisis bisnis.

Untuk menjawab masalah di atas, *Spark Streaming* merupakan teknologi yang menjadi salah satu solusi terhadap adanya kebutuhan untuk menganalisis *big data* secara *real time*. Data hasil streaming kemudian dapat dianalisis dengan teknik-teknik analisis data berbasis statistik maupun *machine learning/data mining* dan divisualisasikan agar lebih mudah dimengerti.

### 1.2 Rumusan Masalah

- Bagaimana Karakteristik *data stream* dan contoh-contoh analisisnya?
- Bagaimana cara kerja *Spark Streaming*?
- Bagaimana cara mengintegrasikan *Spark Streaming* untuk mengumpulkan data?
- Bagaimana cara menganalisis data yang telah terkumpul?

### 1.3 Tujuan

- Melakukan studi tentang definis, pola-pola, arsitektur, dan manfaat analisis dari data stream

- Mempelajari konsep, arsitektur, cara kerja Spark Streaming dan integrasinya dengan teknologi-teknologi lain
- Mengimplementasikan *Spark Streaming* pada sebuah sistem untuk mengumpulkan data stream dengan kasus-kasus tertentu.
- Menganalisis dan mempresentasikan data

## 1.4 Batasan Masalah

1. Data uji yang digunakan akan berupa data yang didapatkan dari Twitter API dan API lain yang didapatkan dari kafka.
2. Pengembangan perangkat lunak untuk pemrosesan data dilakukan dengan menggunakan library *Spark* dan menggunakan bahasa pemrograman *Scala*.
3. Data yang diolah bisa berubah dan memiliki batasan akses sesuai penyedia data tersebut

## 1.5 Metodologi

1. Mempelajari pola, arsitektur, dan sumber dari *Big Data Stream*.
2. Mempelajari arsitektur, cara kerja, dan komponen-komponen *Spark*.
3. Mempelajari Distribusi data pada *Hadoop distributed file System*.
4. Mempelajari arsitektur dan cara kerja *Spark Streaming* pada *Spark*.
5. Mempelajari Bahasa pemrograman *Scala*.
6. Mempelajari *Kafka* dan *Twitter Analysis*.

## 1.6 Sistematika Pembahasan

1. Bab Pendahuluan  
Bab 1 membahas tentang latar belakang, rumusan masalah, tujuan, Batasan masalah, metodologi penelitian, dan sistematika pembahasan.
2. Bab Landasan Teori  
Bab 2 membahas tentang teori-teori mengenai *Big Data*, *Big Data Stream*, Sistem terdistribusi *Spark*, *Spark Streaming*, *Kafka*, dan *Twitter Analysis*.
3. Bab Studi Eksplorasi  
Bab 3 membahas tentang langkah-langkah untuk melakukan konfigurasi kluster pada *hadoop*, konfigurasi kluster untuk *Spark*, hasil studi eksplorasi *Spark Streaming*.
4. Bab Analisis dan Perancangan  
Bab 4 membahas tentang analisis perangkat lunak *Spark*, analisis data uji, analisis masukan dan keluaran, analisis antar muka, diagram *use case*, skenario *use case*, rancangan proses praolah, rancangan proses analisis, diagram kelas, dan rancangan antarmuka.
5. Bab Implementasi dan Eksperimen  
Bab 5 membahas tentang implementasi perangkat lunak, eksperimen performansi, perintah-perintah *Spark Streaming* yang diimplementasikan, dan analisis hasil eksperimen.



#### 6. Bab kesimpulan dan Saran

Bab 6 membahas tentang kesimpulan yang disampaikan penulis setelah melakukan penelitian ini dan saran-saran untuk pengembangan lanjut.



## BAB 2

### LANDASAN TEORI

Pada Bab ini akan dijelaskan dasar-dasar teori tentang Big Data, Data Stream beserta contoh-contohnya, *Apache Spark*, *Spark Streaming*, *Flume*, *Kafka*, dan *Twitter Analytics*. Serta Bahasa pemrograman yang akan digunakan yaitu *Scala*.

#### 2.1 Big Data

*Big Data* merupakan data yang melebihi kapasitas pemrosesan dari sistem basis data konvensional. Data Tersebut berukuran terlalu besar, tumbuh dengan sangat cepat, memiliki banyak variasi tipe data, dan tidak cukup pada arsitektur basis data konvensional.

*Big Data* memberikan dua kegunaan untuk sebuah organisasi, yaitu untuk keperluan analisis dan keperluan bisnis. *Big Data* dapat dianalisis untuk mendapatkan informasi seperti hubungan antar pelanggan. Hal ini dapat dilihat dari hasil analisis transaksi setiap pelanggan, graf sosial, dan graf geografis.

suatu set data dapat dikatakan sebagai big data jika set data tersebut memenuhi salah satu dari lima karakteristik big data. Kelima karakteristik *Big Data* yang sering disebut dengan 5V adalah *volume*, *velocity*, *variety*, *veracity*, dan *value*.

1. *Volume*

*volume* merupakan istilah untuk menggambarkan ukuran dari data. Data dengan ukuran besar mempunyai kebutuhan penyimpanan dan pemrosesan yang berbeda serta tambahan dalam persiapan, pengolahan pemrosesan data. Data berukuran besar tersebut dapat berasal dan transaksi online, eksperimen penelitian ilmiah, sensor, dan media sosial.

2. *Velocity*

*Velocity* dari data merupakan waktu yang diperlukan untuk melakukan pengolahan data ketika data tersebut masuk ke penyimpanan. Untuk menangani data yang masuk dengan cepat, perusahaan atau organisasi memerlukan solusi pemrosesan data yang elastis dan terbuka, dan sesuai.

Kecepatan dari data tidak selalu tinggi dan tergantung pada sumber data. Kecepatan data dapat dipertimbangkan ketika data berukuran besar dan dapat dihasilkan dalam waktu yang singkat. Seperti; 350.000 cuitan, 300 jam video, 171 juta surat elektronik, dan 330 *gigabytes* sensor.

3. *Variety*

*Variety* atau variasi data mengacu pada banyaknya format dan tipe data yang perlu didukung oleh solusi *Big Data*. Variasi data ini merupakan tantangan bagi yang ingin melakukan integrasi, transformasi, pemrosesan, dan penyimpanan data.

4. *Veracity*

*Veracity* mengacu pada kualitas atau akurasi data. Data yang masuk akan diperiksa untuk menentukan kualitas dan menghindari adanya data yang tidak valid serta menghilangkan *Noise*.

Data yang masuk tersebut dapat terdiri dari sinyal dengan informasi tersimpan dan *noise* dari suatu set data. Noise tersebut tidak menyimpan informasi bermakna dan tidak bernilai. Oleh karena itu, data dengan perbandingan sinyal terhadap noise yang besar, memiliki veracity yang lebih besar.

#### 5. Value

*Value* atau nilai didefinisikan sebagai kegunaan data tersebut bagi perusahaan atau organisasi. Karakteristik value berhubungan *linear* dengan karakteristik *veracity* karena besarnya veracity tersebut menentukan besarnya nilai yang dimiliki suatu data dapat bernilai hanya pada rentang waktu tertentu saja dan tidak bernilai di luar rentang waktu tersebut.

*Big Data* yang diolah dapat dihasilkan oleh manusia maupun mesin. Data yang dihasilkan manusia berupa hasil interaksi antara manusia dengan sistem. Data yang dihasilkan oleh mesin dapat berupa hasil dari perangkat lunak maupun perangkat keras sebagai respon dari aktivitas dunia nyata. Data yang dihasilkan tersebut dapat dikelompokkan menjadi tiga tipe dasar yaitu; data terstruktur, data tidak terstruktur, dan data semi terstruktur.

#### 1. Data Terstruktur

Data Terstruktur adalah data yang dimodelkan dalam sebuah model data atau skema data dan biasanya disimpan dalam bentuk tabel relasional. Data Tersebut digunakan untuk melihat hubungan antar entitas sehingga pada umumnya tersimpan dalam basis data relasional. Data terstruktur biasanya dihasilkan dari aplikasi perusahaan dan sistem informasi. Data terstruktur tidak memerlukan pertimbangan khusus. Terkait penyimpanan maupun pemrosesan karena basis data sudah mendukung tipe data terstruktur.

#### 2. Data Tidak Terstruktur

Data Tidak terstruktur merupakan data yang tidak dimodelkan dalam model data atau skema data. Sebagian besar dari data perusahaan merupakan data yang tidak terstruktur. Data tidak terstruktur pada umumnya berupa file teks atau media yang masing-masing tidak tergantung pada file-file lain. Oleh karena itu, pengolahan data tidak terstruktur memerlukan penanganan khusus. contoh: video memerlukan *coder* dan *encoder*

#### 3. Data Semi Terstruktur

Data semi terstruktur mempunyai struktur tertentu dan konsisten, tetapi tidak bersifat relasional. Struktur data pada tipe ini berupa struktur hirarkis atau dalam bentuk graf. umumnya berbentuk XML atau JSON. Adanya struktur dalam data membuat data semi terstruktur lebih mudah untuk diproses dibandingkan data tidak terstruktur.

## 2.2 Big Data Stream

*Data Stream* adalah urutan rekaman atau kejadian (*events*) yang tidak pernah berhenti. Serangkaian *data stream* bersifat tidak terbatas dan akan terus dihasilkan (*in-motion*) sedangkan waktu dan tempat yang dialokasikan untuk mengolah data terbatas. Selain itu, sifat khusus dari *data stream* adalah interaksi yang terbatas dengan sumber data, *data stream* hanya bisa menerima data dari sumber dan tidak bisa mengirim informasi kembali dan data yang dihasilkan hanya bisa di akses dan diproses sekali saja sebelum ditumpuk dan dikumpulkan di tempat penyimpanan. Sehingga, hanya satu atau beberapa elemen terbaru saja yang bisa diproses.

*Big Data Stream* adalah gabungan dua sifat dari *Big Data* dan *Data Stream*. Selain memiliki atribut-atribut *Big Data* yang besar, berkembang dengan sangat cepat, dan bervariasi. *Big Data Stream* terus menerus dihasilkan sedangkan waktu dan tempat untuk mengolah data tersebut sangat terbatas.

*Data stream* banyak digunakan untuk proses agregasi, korelasi, dan *filtering* secara *real-time*. *Data Stream* memungkinkan pengguna untuk melihat *insights*, menganalisisnya, dan menarik kesimpulan dari *insights* tersebut. Beberapa contoh analisis adalah; memantau informasi dari suatu sosial media seperti *twitter* untuk mendapatkan informasi untuk mengetahui tingkah pengguna, memantau aktivitas web dengan mencatat pembaharuan *web logs* setiap detiknya, mendeteksi anomali dari suatu jaringan atau sensor yang terus menerus mengirimkan data untuk dianalisis. Contoh beberapa kasus nyata adalah:

- institusi keuangan yang memantau perubahan pasar saham. Sehingga bisa mengubah portofolio berdasarkan batasan batasan tertentu (menjual saham ketika saham mencapai titik nilai tertentu)
- Memantau suatu jaringan listrik berdasarkan *throughput* dan akan memberi peringatan jika melebihi batas tertentu.
- Kantor berita yang menggunakan *clickstream* dari berbagai platform untuk memperkaya data dengan informasi demografik sehingga bisa merekomendasikan artikel ke pembaca yang relevan.
- perusahaan *e-commerce* menggunakan data stream untuk melihat apakah ada anomali pada *datastream* dan akan mengirimkan peringatan jika terjadi anomali.

### 2.2.1 Pengertian Stream Processing

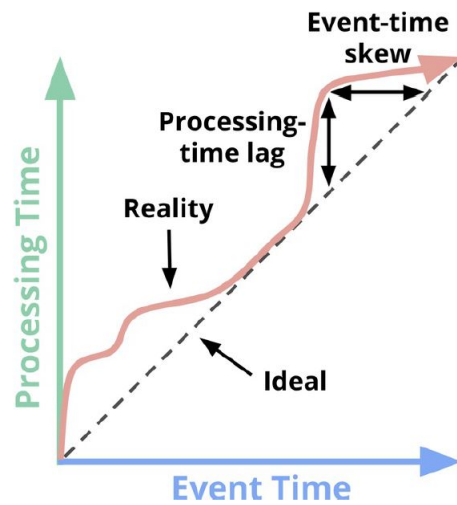
Stream Processing adalah proses yang dirancang untuk mengatasi untaian data yang tidak terbatas atau tidak memiliki awalan dan akhiran yang jelas. Stream Processing menawarkan pemrosesan data dengan latency yang rendah dan hasil yang spekulatif. Untuk memproses suatu data, stream processing mempunyai dua dimensi penting yaitu; cardinality dan constitution. cardinality adalah ukuran dari data tersebut; terbatas (bounded) atau tidak terbatas (unbounded). Constitution adalah bagaimana data ditampilkan; adalah bagaimana data ditampilkan; tabel yang menampilkan data secara menyeluruh seperti SQL atau potongan-potongan le pada HDFS seperti Map Reduce.

Walaupun Stream Processing menawarkan Latency yang rendah, namun hasil dari data yang diproses kurang akurat. Hal ini disebabkan karena Stream Processing menggunakan algoritma pendekatan. Salah satu cara mengatasi masalah ini adalah dengan perancangan arsitektur Lambda. Pembahasan arsitektur Lambda dan Kappa akan dibahas lebih lengkap di 2.2.4. Stream processing memodelkan waktu menjadi dua domain; Event time dan Processing Time. *Event-Time* adalah waktu saat data sedang dihasilkan. Setiap data yang dihasilkan akan diberi *time stamp* agar semua data dari sumber yang sama bisa diurutkan secara kronologis. *Event-time* digunakan diantaranya untuk menggolongkan perilaku pengguna dari waktu ke waktu. Processing time adalah waktu ketika data diobservasi pada Stream-Processing. Secara ideal, *event time* dan *processing time* bernilai sama. Artinya, sistem langsung mengobservasi data saat data itu sedang dihasilkan. Namun pada nyatanya, nilai processing time dan event time tidak selalu sama karena dipengaruhi sumber data, waktu eksekusi, dan performa mesin hardware. Hubungan antara event time dan processing-time bisa dilihat di Gambar 1:

### 2.2.2 Pemodelan Stream Processing

Pada *Stream Processing* waktu dimodelkan menjadi dua domain yaitu; *Event Time* dan *Processing Time*. *Event Time* digunakan diantaranya untuk menggolongkan perilaku pengguna dari waktu ke waktu, aplikasi tagihan, mendeteksi suatu anomali yang terjadi. Secara ideal, *event time* dan *Processing time* bernilai sama. Artinya, sistem langsung mengobservasi data saat data itu sedang dihasilkan.

Tetapi pada nyatanya tidak semulus itu, nilai *event time* dan *processing time* tidak selalu sama karena dipengaruhi sumber data, waktu eksekusi, dan performa mesin serta *hardware*. Hubungan antara *event time* dan *processing time* bisa dilihat pada gambar



Gambar 2.1: Pemetaan *Time-Domain*

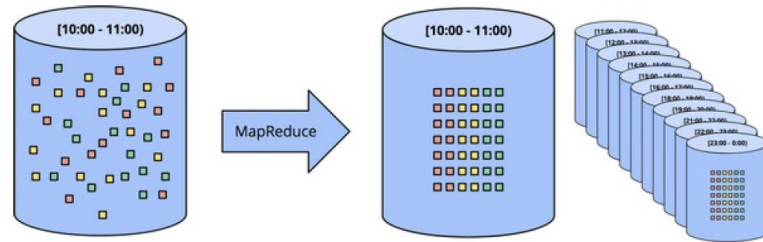
Sumbu X adalah *event time* atau completeness dalam sistem. Sumbu Y adalah Processing time waktu jam biasa yang diamatai oleh sistem data processing ketika sedang dieksekusi. Processing-time lag adalah jarak vertikal antara garis ideal dengan garis merah yang menunjukkan ada berapa banyak waktu delay yang sedang diobservasi di antara kejadian pada waktu tertentu dan kapan delay itu terjadi. event-time skew adalah garis horizontal dari garis ideal dengan garis merah yang menunjukkan banyaknya distribusi data di pipeline pada saat itu dan menunjukkan seberapa ketinggalan suatu pipeline dari pipeline ideal.

### 2.2.3 Pola Pemrosesan Data Stream

Pola-pola teknik pemrosesan stream processing dibagi menjadi dua yaitu batch dan streaming. Pola yang tergolong ke dalam batch tidak dirancang untuk data yang tidak terbatas. Tetapi pemrosesan data tidak terbatas bisa dilakukan dengan membagi dataset menjadi beberapa bagian yang berisi potongan-potongan dari dataset tersebut yang lebih kecil dan terbatas. Sehingga bisa diproses dengan batch processing. Beberapa jenis batch processing adalah *fixed windows* dan *Session*:

#### Fixed Windows

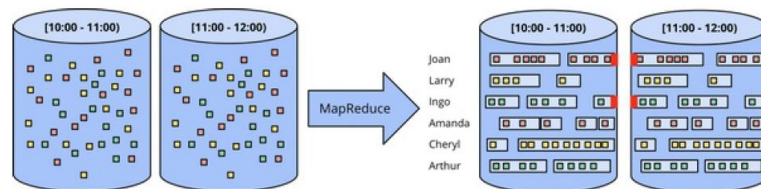
*Fixed Windows* adalah pola pemrosesan yang paling umum. *Fixed Windows* memproses dataset tidak terbatas menggunakan mesin *batch* yang dijalankan dengan cara membagi input data ke beberapa *window* dan memproses setiap *window* tersebut secara terpisah. proses ini dilakukan secara berulang-ulang. Seperti pada gambar 2.2. Metode ini digunakan untuk data yang berbentuk logs karena data bisa ditulis pada directory dan hirarki file dengan nama yang sama dengan window. Jika data terkena delay gara-gara partisi *network* perlu dilakukan mitigasi yang memperlambat proses sampai semua data terkumpul ketika data datang terlambat sistem telah memproses seluruh *batch* sebelum dipindahkan ke *windows*.

Gambar 2.2: *data processing* dengan menggunakan fixed window

### Session

memiliki cara kerja yang hampir sama dengan *fixed windows* bedanya pemotongan data dipisah berdasarkan session sehingga pembagian data jadi tidak seimbang data yang sama mungkin berakhir di batch berbeda.

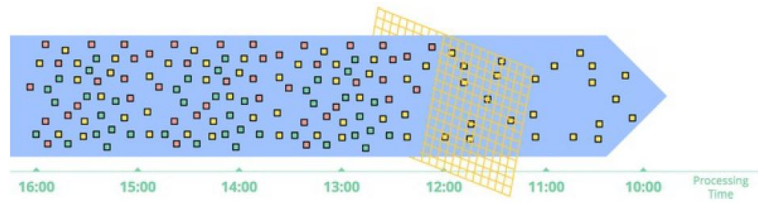
Session adalah aktivitas atau periode spesifik yang akan dihentikan bila diselingi oleh suatu ketidakaktifan Session dihitung oleh batch processing dan dibagi ke pada seluruh batch seperti pada gambar 4. Banyaknya split berbanding terbalik dengan latency. Jadi, bila jumlah split dikurangi maka latency akan meningkat. Sebaliknya jika split bertambah, latency akan berkurang.

Gambar 2.3: *Unbounded Data Processing* dengan *Sessions*

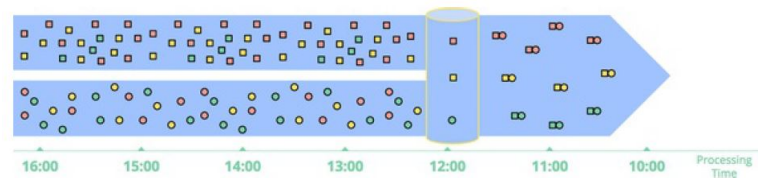
Pola-pola yang dikelompokkan menjadi *Streaming* khusus dibangun untuk memproses data tidak terbatas (unbounded). Karena pada kasus nyata banyak data yang tidak teratur atau berbentuk dan tidak sekuensial. Sehingga jika data ingin dianalisis dalam konteks data itu masih baru, harus ada sebuah metode pada pipeline untuk mengurutkan data berdasarkan waktu. Ada empat pola yang digunakan sebagai pendekatan terhadap dataset yang mempunyai karakteristik-karakteristik ini yaitu; *filtering*, *approximation algorithm*, dan *windowing*.

### filtering

*filtering* adalah operasi mendasar dan dilakukan secara *Time-Agnostic* yang memilah data yang masuk. Pola *Time-Agnostic* digunakan untuk kasus-kasus dimana waktu tidak relevan. Artinya, semua logika dan informasi pada data yang relevan ada pada data dan yang lebih menentukan relevansi dari suatu data adalah urutan kedatangan data tersebut. Jadi, pola ini hanya membutuhkan streaming engine yang mendukung pengiriman data yang sederhana karena itu semua sistem streaming bisa menggunakan pola *Time-Agnostic*. Sistem melihat setiap record yang datang dan melihat apakah domain data sama dengan domain tujuan. Bila tidak data akan dibuang karena itu proses ini hanya bergantung pada urutan kedatangan data bukan dari *event-time*.

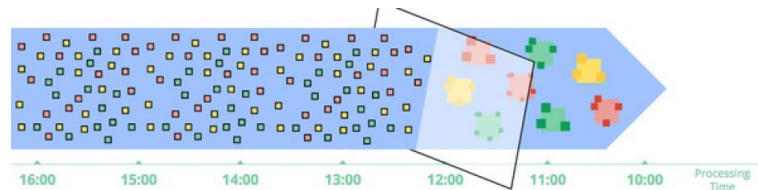
Gambar 2.4: *Filtering data*

proses di atas adalah proses *filtering data* dari kumpulan data yang heterogen menjadi homogen dengan tipe yang sama dan diletakan pada klaster yang sama. salah satu cara untuk mengelompokkan data yang bersifat sama adalah dengan melakukan *Inner Join*. *Inner Join* adalah salah satu bagian dari filtering dimana proses menggabungkan dua sumber data yang tidak terbatas. Jika ada suatu data datang sistem akan menyimpan data tersebut ke *persistent state* ketika data berikutnya datang sistem akan menggabungkan data tersebut dengan data yang ada di *persistent state*.

Gambar 2.5: *inner join pada unbounded data*

### Approximation Algorithm

*Approximation algorithm* adalah algoritma pendekatan yang menerima input data tidak terbatas dan mengelompokkan data tersebut menjadi berdekatan jika memiliki sesuatu kesamaan. seperti pada gambar 2.5. Algoritma pendekatan ini memang dirancang untuk data tidak terbatas. Tetapi, algoritma pendekatan merupakan algoritma yang rumit sehingga algoritma pendekatan susah untuk dipanggil ketika dibutuhkan, dan pendekatan dari algoritma ini cenderung terbatas.

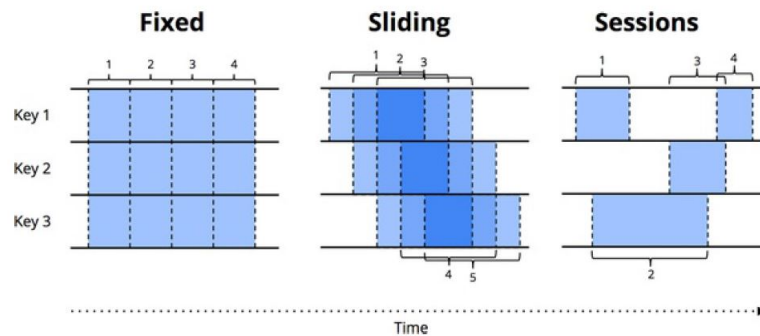
Gambar 2.6: Algoritma pendekatan pada *unbounded data*

Data dijalankan melewati algoritma yang kompleks dan menghasilkan data output yang terlihat lebih mirip hasil yang diinginkan. Algoritma pendekatan langsung memproses data yang datang karena itu melibatkan *processing time*. *Processing-time* digunakan algoritma sebagai pengecek *error* pada data dengan membaca waktu urutan kedatangan dari data.

### Windowing

*Windowing* adalah fungsi menerima *data source* sebagai input. Lalu, membagi data tersebut menjadi beberapa bagian dan memberi batasan pada potongan data tersebut. Bisa dilihat pada gambar 2.6. Windowing memiliki dua variasi *Fixed Windows* dan *Sliding Windows*





Gambar 2.7: Teknik Windowing

Windowing memiliki beberapa pola yang sering digunakan beberapa contohnya adalah:

- Fixed Windows (Tumbling Windows) bekerja seperti Fixed windows pada batch processing. Fixed windows membagi waktu menjadi segmen-segmen dengan ukuran yang tetap. Seperti pada gambar 2.6. Segmen untuk fixed window diterapkan secara seragam pada seluruh dataset. Pembagian segmen dengan ukuran yang sama disebut aligned window. Tetapi, terkadang window tidak dibagi dengan ukuran yang sama. Dalam beberapa kasus, pembagian data bergantung pada ukuran dataset dan bervariasi tiap dataset. Pembagian waktu yang tidak merata, unaligned window, membantu meratakan penyebaran waktu penyelesaian.
- Sliding Windows (Hopping Windows) adalah fixed window yang lebih umum. Sliding windows memiliki panjang dan periode yang tetap. Jika periode lebih kecil dari length maka terjadi overlap pada windows. Jika periode sama dengan waktu maka window akan menjadi fixed window. Jika periode lebih besar dari length maka akan menjadi sampling window yang hanya akan melihat suatu subset data dengan waktu yang lama. Seperti pada gambar 2.6
- Dynamic sessions biasanya digunakan untuk menganalisa perilaku pengguna secara berkala dengan mengelompokkan suatu rantai peristiwa yang berhubungan. contohnya adalah berapa banyak video yang ditonton oleh pengguna dalam sekali duduk. Panjang dari suatu sesi tidak bisa ditentukan terlebih dahulu. Panjang sesi tergantung dari seberapa banyak data yang terlibat. Dynamic Session merupakan salah satu penerapan unaligned window karena untuk setiap session subset pada suatu dataset tidak pernah identik.

Selain itu Windowing bekerja dengan dua cara berbeda. Seperti *Processing-time Windowing* dimana Sistem menyimpan sementara data yang datang pada window untuk beberapa saat sampai processing time telah lewat. Contohnya; misalkan ada fixed windows dengan durasi 5 menit, sistem akan menyimpan sementara data selama lima menit waktu pemrosesan. Lalu, sistem akan mengirim data yang telah diobservasi selama lima menit tersebut ke *downstream* untuk diproses.

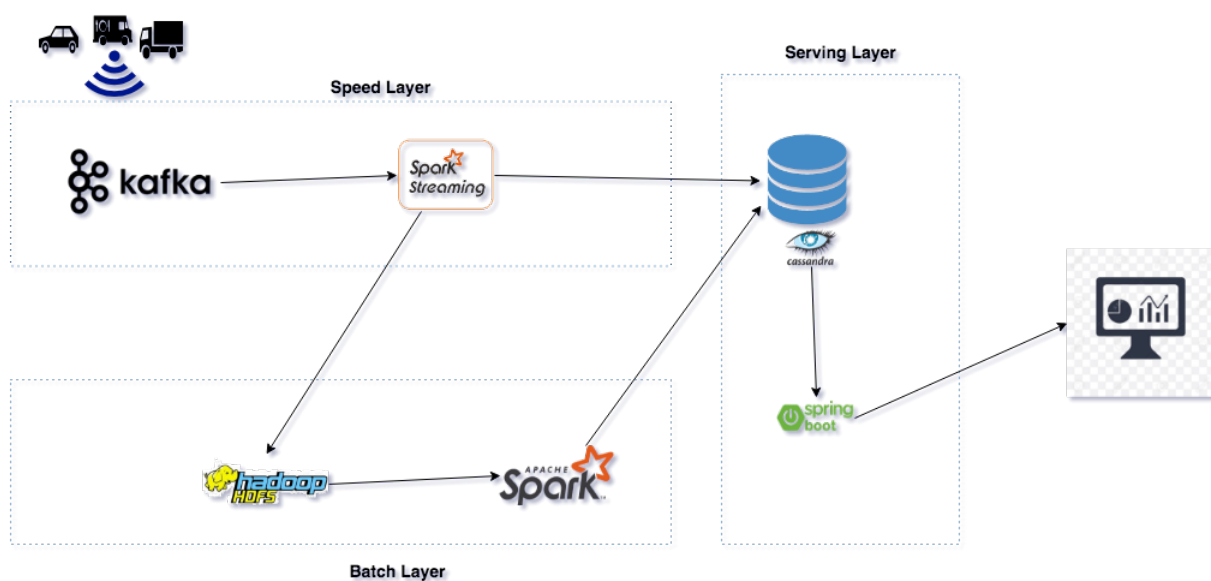
Proses windowing ini sangat simpel dan implementasinya mudah karena sistem tidak harus mengatur data sesuai waktu. data hanya akan disimpan sementara ketika datang dan langsung dilempar ke downstream ketika processing time selesai. Karena sistem bisa mengetahui semua input karena telah dilihat oleh window. Sehingga sistem bisa dengan baik memprediksi kapan suatu window akan selesai. metode yang kedua adalah *Event-time Windowing* Event-time Windowing digunakan ketika sistem mengobservasi sumber data yang tidak terbatas dalam potongan-potongan data yang terbatas berdasarkan kapan data itu terjadi.

## 2.2.4 Arsitektur Stream Processing

### Lambda Architecture

Arsitektur Lambda adalah teknik pemrosesan data yang bisa menangani data yang besar dengan cara menggabungkan metode batch dan stream processing. Teknik ini menyeimbangkan antara latency, throughput, dan fault-tolerance dengan menggunakan batch processing yang menyediakan penyimpanan data yang akurat dan komprehensif. Juga memanfaatkan stream processing agar mendapat data tidak terbatas secara realtime.

Banyak perusahaan yang menggunakan metode stream processing untuk memprediksi updates dari model dan menyimpan event yang berbeda yang digunakan sebagai bahan untuk memprediksi. Untuk menangani kejadian seperti itu, Arsitektur Lambda memiliki tiga layer; *Batch layer*, *speed layer (stream layer)*, dan *Serving layer*.



Gambar 2.8: Arsitektur *Lambda*

#### *batch layer*

layer ini terlebih dahulu memproses data dengan menggunakan sistem terdistribusi yang bisa menangani data yang besar. tujuan dari batch layer adalah untuk meningkatkan akurasi dengan cara memproses semua data yang ada ketika membangun view. Artinya, batch layer bisa memperbaiki error pada data dengan memproses data kembali berdasarkan dataset yang sudah lengkap.

Setiap data yang terus menerus datang ke sistem akan diteruskan ke batch layer dan stream layer secara bersamaan. Data Stream yang baru masuk ke batch layer langsung diproses pada data lake. Data disimpan pada data lake menggunakan in-memory database atau long term persistent database seperti NoSQL. Data akan diproses menggunakan MapReduce atau machine- learning. Apache hadoop digunakan di layer ini karena memiliki throughput yang paling tinggi.

#### *Speed Layer(Stream Layer)*

Layer ini memproses data stream secara real-time tanpa memperdulikan completeness atau akurasi dari data. layer ini mengurangi throughput untuk mengurangi latency. Sehingga data yang terbaru bisa langsung dilihat. Speed layer digunakan untuk mengisi jarak yang

disebabkan oleh batch layer dengan memberikan informasi tentang data terkini. View yang dihasilkan dari layer ini belum tentu akurat namun bisa langsung dilihat dan diakses. Data yang lebih akurat akan disediakan dan diganti nanti oleh hasil data yang telah diolah oleh batch layer ketika sudah tersedia.

#### *Serving Layer*

Output dari batch dan Speed layer diteruskan dan disimpan di layer ini dan proses ad-hoc queries akan dilakukan di layer ini dengan mengembalikan views dari data yang telah diproses.

Arsitektur Lambda dapat dianggap sebagai arsitektur pemrosesan data yang real-time. Seperti disebutkan di atas, dapat menahan kesalahan serta memungkinkan skalabilitas. Arsitektur ini menggunakan fungsi-fungsi batch dan stream lalu menambahkan data baru ke penyimpanan utama sambil memastikan bahwa data yang ada akan tetap utuh. Perusahaan seperti Twitter, Netflix, dan Yahoo menggunakan arsitektur ini untuk memenuhi kualitas standar layanan.

Keuntungan dari Arsitektur Lambda antara lain adalah; *Batch Layer* dari arsitektur ini mengatur histori data dengan penyimpanan terdistribusi yang *fault tolerant* yang mana akan memperkecil terjadinya error walaupun sistem *crash*, seimbang antara kecepatan dan keandalan, Scalable dan fault tolerant untuk data processing. Tetapi, arsitektur ini memiliki kelemahan yaitu; penerapan yang cukup sulit karena melibatkan *batch* dan *stream processing*, memproses setiap batch pada beberapa *cycle* tidak menguntungkan untuk beberapa skenario, data yang dimodelkan dengan arsitektur ini susah untuk dimigrasi dan diorganisir ulang.

### Kappa Architecture

*Kappa Architecture* adalah simplifikasi dari arsitektur lambda. Susunan arsitektur ini hampir sama dengan sistem arsitektur lambda namun tidak memiliki *batch layer*. Untuk mengganti *batch processing* data langsung diteruskan ke sistem streaming. Arsitektur ini digunakan model data berupa; beberapa *event* atau *query* data dicatat dalam suatu antrian untuk disesuaikan dengan penyimpanan atau riwayat sistem file terdistribusi, urutan *event* dan *query* tidak ditentukan sebelumnya, platform *stream processing* dapat berinteraksi dengan basis data kapan saja. model ini sangat *resilient* dan bisa menangani beberapa terabyte data untuk penyimpanan yang diperlukan untuk setiap sistem node yang mendukung replikasi.

Skenario data yang disebutkan di atas ditangani oleh *Apache kafka* yang cepat, toleran terhadap kesalahan dan dapat diskalakan secara horizontal. Hal ini memungkinkan mekanisme yang lebih baik untuk mengatur aliran data. karena kontrol yang seimbang pada *stream processing* dan database maka hal ini memungkinkan aplikasi untuk bekerja sesuai ekspektasi. Kafka menyimpan data yang diminta untuk jangka waktu yang lebih lama dan meyakini *queries* analog dengan menautkannya ke posisi yang sesuai dari log yang disimpan. Manfaat dari arsitektur ini adalah bisa mempertahankan sejumlah besar data untuk menyelesaikan *queries*

keuntungan dari arsitektur ini adalah dapat digunakan untuk mengembangkan sistem data yang merupakan yang tidak membutuhkan *batch layer*, Pemrosesan ulang hanya diperlukan saat kode berubah dan dapat digunakan dengan memori tetap, Lebih sedikit sumber daya yang diperlukan karena pembelajaran mesin dilakukan secara *real-time*. Tetapi, tidak adanya **batch layer** dapat mengakibatkan kesalahan selama pemrosesan data atau saat memperbarui database yang mengharuskan untuk ada pemrosesan ulang atau rekonsiliasi.

## 2.3 Sistem Terdistribusi Hadoop

*Apache Hadoop* merupakan sebuah *framework* yang bersifat *open-source* untuk menulis dan menjalankan aplikasi terdistribusi untuk mengolah data dalam ukuran besar. Proyek Hadoop disusun dengan tujuan untuk mengatasi masalah skalabilitas pada *nutch*, sebuah *open-source crawler* dan

mesin pencarian. Hadoop merupakan bagian dari implementasi hasil riset google mengenai sistem file terdistribusi dan komputasi paralel.

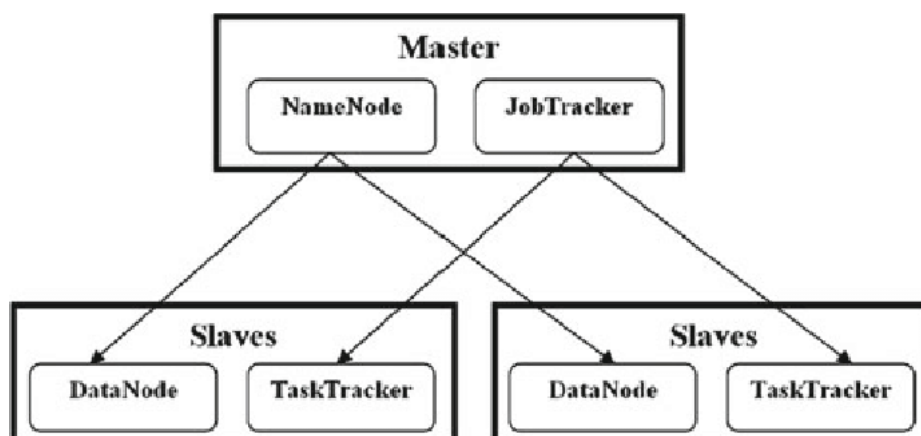
Hadoop dapat berjalan di atas kluster mesin-mesin dengan kekuatan pemrosesan yang setara dengan komputer komersial maupun berjalan dengan layanan komputasi *cloud* yang dapat diakses dengan mudah oleh klien. Sistem terdistribusi Hadoop tidak perlu menggunakan mesin-mesin berspesifikasi tinggi, karena beban pemrosesan akan didistribusikan ke masing-masing mesin di dalam kluster. Sistem tersebut dapat memudahkan pengguna ketika sistem diperlukan untuk pengolahan data dengan ukuran lebih besar, karena penambahan ukuran data hanya memerlukan tambahan mesin ke dalam kluster dengan spesifikasi yang sama atau mendekati mesin-mesin lain dalam kluster.

Mesin-mesin dalam kluster dapat menjadi lebih ekonomis dibandingkan dengan menggunakan sebuah mesin dengan spesifikasi yang lebih baik. Hal ini terkait dengan penambahan ukuran data yang akan diproses serta kerusakan perangkat keras yang mungkin terjadi. Hadoop dapat mengatasi kasus kerusakan tersebut tanpa ada kehilangan data, tetapi penggunaan sebuah mesin saja memiliki resiko kehilangan data saat ada kerusakan. Selain itu, penggantian sebuah mesin yang rusak di dalam kluster akan memerlukan biaya yang lebih kecil dibanding mengganti mesin dengan spesifikasi tinggi.

Hadoop memerlukan pengolahan data dengan ukuran yang sangat besar, Sehingga pemindahan data berukuran besar tersebut melalui jaringan akan memperlambat jalannya proses. Oleh karena itu, data tersebut diperkecil dengan membagi data tersebut menjadi blok-blok data dan mendistribusikannya ke masing-masing mesin dalam kluster. Kode aplikasi yang merupakan sebuah pekerjaan di lingkungan Hadoop dan berukuran lebih kecil dibandingkan dengan data akan didistribusikan ke dalam mesin-mesin kluster. Dengan demikian, pekerjaan pemrosesan data akan terjadi di setiap mesin di dalam kluster terhadap blok-blok data yang ada pada masing-masing mesin kluster.

### Arsitektur Hadoop

Sebuah kluster hadoop terdiri dari mesin-mesin yang saling terhubung. Kluster hadoop mempunyai arsitektur master/slave dengan sebuah mesin sebagai master node dan mesin-mesin lain sebagai slave node seperti yang ditunjukkan pada gambar. Setiap node tersebut akan mempunyai komponen penyimpanan berupa HDFS dan komponen komputasi berupa MapReduce. Seperti yang ditunjukkan pada gambar dapat dilihat bahwa HDFS dan MapReduce pada node mempunyai *daemon* dan tugas yang berbeda dengan komponen HDFS dan MapReduce yang ada pada slave node. Perbedaan tugas-tugas tersebut akan dijelaskan dibagian.

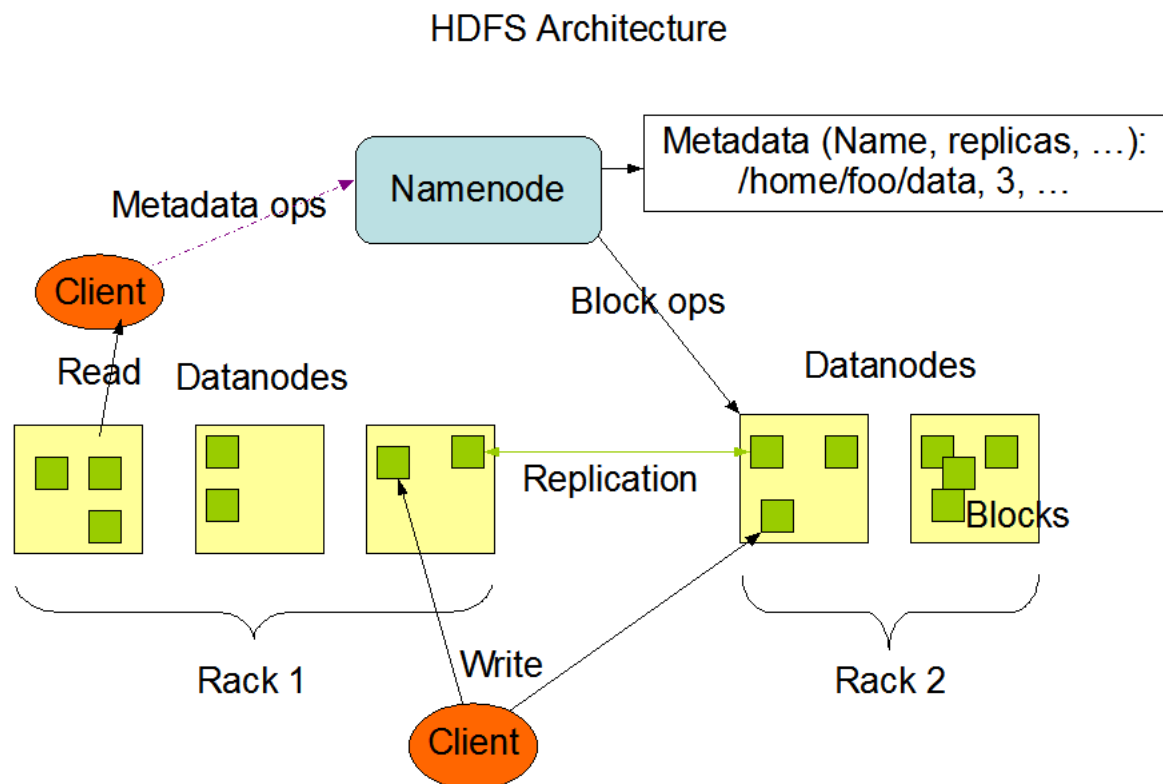


Gambar 2.9: Arsitektur *Hadoop*

### Komponen-Komponen penting Hadoop

#### *Hadoop Distributed File System(HDFS)*

Hadoop distributed file system atau HDFS adalah komponen penyimpanan data dalam Hadoop. HDFS dirancang untuk memiliki *throughput* tinggi dan cocok untuk melakukan operasi baca dan tulis pada file dengan ukuran yang sangat besar. Untuk mendukung hal tersebut, HDFS memanfaatkan ukuran blok data yang besar dan optimasi lokalitas data untuk mengurangi input output jaringan. Selain itu, data yang tersimpan di dalam HDFS tidak akan hilang ketika ada kerusakan pada salah satu mesin. Hal ini disebabkan adanya replikasi untuk setiap blok data yang terdistribusi di dalam klaster. Banyaknya replikasi yang terjadi pada awalnya adalah tiga, tetapi angka ini dapat dikonfigurasi ulang menjadi lebih sedikit maupun lebih banyak.



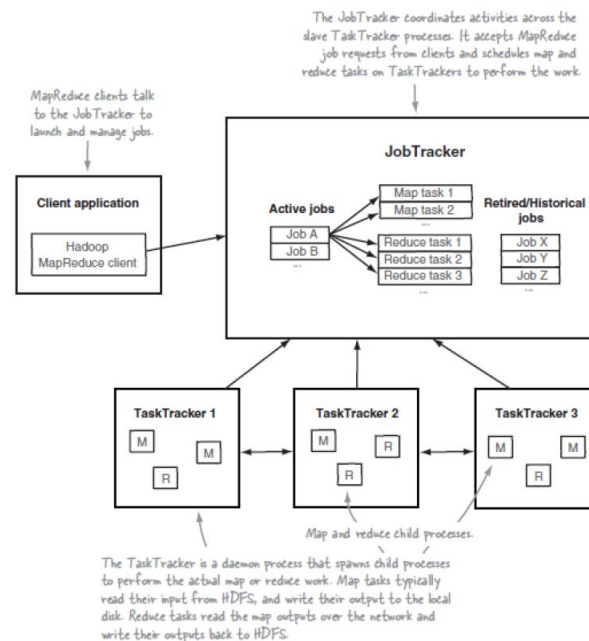
Gambar 2.10: Arsitektur HDFS

Komponen HDFS pada master node menjalankan sebuah daemon yang disebut dengan NameNode. NameNode bertugas untuk mengatur pembagian blok-blok data ke slave node dan mencatat lokasi masing-masing blok data tersebut. NameNode Merupakan komponen yang penting dalam eksekusi pemrosesan data dalam klaster. Berbeda dengan Master Node, komponen HDFS pada slave node menjalankan daemon yang disebut dengan DataNode. Data Node bertugas untuk melakukan proses baca tulis blok-blok data pada file asli yang terdapat sistem file lokal. Komunikasi pada awal operasi tulis atau baca terjadi diantara klien dan NameNode untuk mendapatkan lokasi blok-blok data yang akan diproses. setelah itu klien dapat berkomunikasi dengan DataNode lain untuk melakukan replikasi blok-blok data yang ada.

#### *MapReduce*

MapReduce merupakan komponen komputasi dalam Hadoop. Model pemrograman yang dimiliki oleh MapReduce memungkinkan seorang programmer mengimplementasikan sebuah

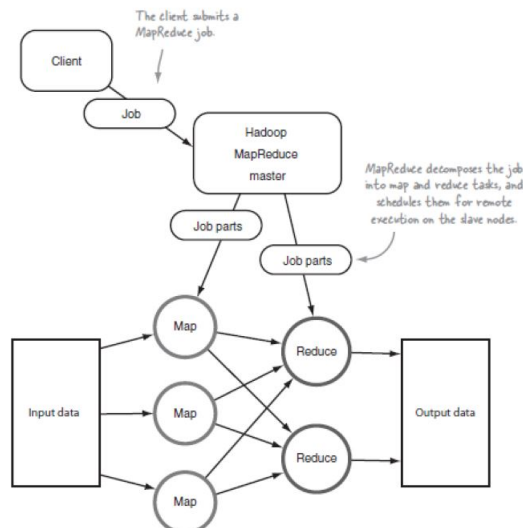
aplikasi yang berjalan paralel dengan mudah. Konfigurasi mengenai paralelisasi komputasi, distribusi pekerjaan, dan cara mengatasi kegagalan perangkat lunak maupun keras sudah ditangani oleh Hadoop, sehingga programmer hanya perlu mengimplementasikan pekerjaan pemrosesan apa saja yang perlu dilakukan



Gambar 2.11: Arsitektur MapReduce

Komponen MapReduce pada master node menjalankan daemon JobTracker yang merupakan daemon yang menghubungkan proses pada Hadoop dengan aplikasi. JobTracker bertugas melakukan pemecahan pekerjaan yang dikirimkan klien menjadi unit-unit pekerjaan map dan reduce. JobTracker akan mendistribusikan unit-unit pekerjaan tersebut ke slave node, melakukan penjadwalan komputasi, dan melakukan pengawasan terhadap pemrosesan yang dilakukan dalam slave node. Komponen MapReduce pada setiap slave node menjalankan daemon TaskTracker yang bertugas untuk melakukan eksekusi pemrosesan di dalam slave node. TaskTracker akan selalu berkomunikasi dengan JobTracker untuk memantau jalannya proses. Jika komunikasi tersebut terputus, dapat diasumsikan proses pada TaskTracker tersebut gagal dan unit pekerjaan yang sesuai akan dikirimkan oleh JobTracker ke slave node lain yang terdapat di dalam kluster.

MapReduce terdiri dari komponen-komponen mapper dan reducer. Pekerjaan yang dikirimkan ke dalam kluster akan dipecah menjadi pekerjaan map dan reduce yang berjalan secara paralel. Setiap node dalam map dan reduce dapat berdiri sendiri dan tidak tergantung pada node-node map atau reduce lainnya. Ketergantungan yang ada hanya ketergantungan node-node reduce terhadap node-node map. Pemrosesan ini dilakukan dengan model share-nothing, yaitu data yang diolah tidak dibagikan antarnode untuk mencegah node-node tersebut saling menunggu untuk memakai sumber daya. Implementasi aplikasi dengan MapReduce dapat dilakukan dengan mendefinisikan fungsi map dan reduce. Fungsi map menerima masukan berupa pasangan-pasangan key dan value dan memberikan keluaran berupa list key dan value. Fungsi reduce menerima masukan berupa key dan list value dan memberikan keluaran berupa pasangan key dan value.



Gambar 2.12: Proses MapReduce

Gambar 2.10 merupakan gambaran proses yang terjadi ketika klien mengirimkan sebuah pekerjaan MapReduce ke dalam kluster. Pekerjaan yang dikirimkan oleh klien dapat berupa file jar atau xml. Pekerjaan tersebut akan dipecah menjadi unit-unit pekerjaan map dan reduce oleh komponen MapReduce yang terdapat pada master node sebelum didistribusikan ke slave node. Data masukan akan diproses menggunakan komponen mapper pada MapReduce dan format data harus berupa pasangan key dan value. Untuk setiap pasang key dan value tersebut, dilakukan pemrosesan menggunakan fungsi map dan mengembalikan keluaran berupa list pasangan key dan value yang baru. Nilai key pada tahap pemrosesan ini pada umumnya tidak diperhitungkan. Hasil keluaran dari mapper akan diproses terlebih dahulu sebelum dijadikan masukan untuk komponen reducer. Tahap pemrosesan ini disebut sebagai tahap shuffle and sort. Setiap pasangan

key dan value akan diurutkan berdasarkan key yang dimiliki dan mengelompokkan semua value yang mempunyai key yang sama untuk dimasukkan ke node reducer yang sama. Proses ini akan menjadi kompleks ketika semua key yang ada dalam list keluaran mempunyai nilai yang berbeda-beda. Komponen reducer menerima masukan yang berasal dari hasil pada tahap shuffle and sort yang berupa pasangan key dan list value. Untuk setiap nilai key yang ada, fungsi reduce pada reducer akan dipanggil satu kali dan memberikan keluaran berupa pasangan key dan value yang baru. Keluaran dari proses reduce tersebut dapat ditulis ke file yang berada di HDFS maupun ke dalam basis data.

## 2.4 Scala

Scala merupakan bahasa pemrograman fungsional dan berorientasi objek. Sebagai bahasa pemrograman berorientasi objek, setiap sifat dan kemampuan objek Scala dideskripsikan dalam kelas-kelas Scala. Sebagai bahasa pemrograman fungsional, penggunaan fungsi high-order dan pendefinisian fungsi anonim dapat dilakukan pada Scala. Fungsi high-order merupakan fungsi yang menerima fungsi lain sebagai parameter. Fungsi anonim merupakan fungsi yang didefinisikan tanpa memerlukan nama fungsi dan hanya langkah-langkah yang dilakukan oleh fungsi tersebut. Scala merupakan bahasa pemrograman yang berdasar pada Java Virtual Machine, sehingga Scala dapat dikompilasi menjadi Java Byte Code dan dapat dijalankan pada Java Virtual Machine. Oleh karena itu, Scala dapat beroperasi dengan baik bersamaan dengan Java. Library Scala dapat digunakan pada aplikasi berbasis Java dan demikian pula sebaliknya.

### 2.4.1 variable

Scala mempunyai dua jenis variabel, yaitu variabel yang dapat diubah dan tidak dapat diubah. Deklarasi variabel yang dapat diubah dilakukan dengan menggunakan kata kunci `var`, sedangkan variabel yang tidak dapat diubah atau konstanta menggunakan kata kunci `val`. Pengubahan nilai konstanta akan menyebabkan terjadinya error. Contoh deklarasi variabel dapat dilihat pada potongan kode berikut.

### 2.4.2 Fungsi

Deklarasi fungsi pada Scala dapat dilakukan dengan menggunakan kata kunci `def`. Header dari fungsi dapat dituliskan dengan format `def namaFungsi(namaParameter: TipeData): TipeKeluaran`.

Listing 2.1: Deklarasi Fungsi

```
def add( firstInput : Int , secondInput : Int ): Int = {
  val sum = firstInput + secondInput
  return sum
}
```

Bahasa pemrograman Scala mempunyai filosofi kode yang ringkas, sehingga deklarasi fungsi dapat diringkas lebih lanjut. Jika suatu fungsi menggunakan sebuah variabel sebagai penampung hasil komputasi singkat, maka penulisan deklarasi tersebut dapat diringkas menjadi seperti deklarasi sebuah variabel. Penulisan deklarasi tersebut dapat diringkas menjadi seperti berikut ini.

Listing 2.2: Deklarasi Fungsi Ringkas

```
def add( firstInput : Int , secondInput : Int ) = firstInput +
secondInput
```

### Fungsi Lokal

Sebuah fungsi dapat didefinisikan di dalam sebuah fungsi lain. Fungsi yang didefinisikan di dalam fungsi lain tersebut disebut dengan fungsi lokal. Fungsi lokal tersebut dapat mengakses semua variabel yang ada pada fungsi tempat fungsi lokal tersebut didefinisikan, tetapi fungsi lokal tersebut hanya dapat diakses oleh fungsi yang mendefinisikan fungsi lokal tersebut.

### Fungsi Highorder

Fungsi yang menerima fungsi lain sebagai parameter disebut dengan fungsi high-order. Fungsi high-order tersebut dapat membantu mengurangi duplikasi pada kode program dan membuat kode yang lebih ringkas.

Listing 2.3: Contoh fungsi *high-order*

```
def encode (n: Int , f: (Int) => Long ): Long = {
  val x = n * 10
  f(x)
}
```

### Fungsi Anonim

Selain menggunakan cara-cara yang sudah dijabarkan sebelumnya, fungsi Scala dapat dideklarasikan dengan menggunakan hanya parameter fungsi dan langkah-langkah yang perlu dilakukan dalam fungsi. Fungsi yang dideklarasikan dengan cara tersebut dapat dijadikan sebagai masukan dari sebuah fungsi high-order atau dimasukkan sebagai nilai dari variabel. Fungsi yang dideklarasikan dengan cara seperti ini disebut dengan fungsi anonim.



Listing 2.4: Contoh fungsi anonim

```
(x: Int) => {
    x + 100
}
```

Bagian kiri dari karakter panah merupakan parameter dari fungsi, sedangkan bagian kanan merupakan isi fungsi atau operasi yang dilakukan oleh fungsi. Isi fungsi tersebut dimasukkan ke dalam tanda kurung kurawal. Jika fungsi tersebut hanya berisi satu baris perintah saja, maka penulisan isi fungsi tidak memerlukan kurung kurawal. Penulisan fungsi tanpa kurung kurawal tersebut dapat diringkas menjadi seperti berikut ini.

Listing 2.5: Penulisan ringkas fungsi anonim

```
(x: Int) => x + 100
```

Fungsi anonim tersebut dapat digunakan sebagai masukan dari fungsi high-order encode yang sudah didefinisikan sebelumnya.

Listing 2.6: Penggunaan fungsi anonim pada fungsi *high-order*

```
val code = encode (10 , (x: Int) => x + 100)
```

### 2.4.3 Kelas

Kelas merupakan sebuah konsep pemrograman berbasis objek. Pada tingkat dasar, penggunaan kelas merupakan sebuah teknik penyusunan kode untuk mengelompokkan data dan operasi operasinya. Secara konsep, kelas merepresentasikan sebuah entitas dengan sifat-sifat dan kemampuannya. Kelas pada Scala mempunyai kemiripan seperti kelas pada bahasa pemrograman berorientasi objek yang lain. Kelas terdiri dari sejumlah field dan method dengan field adalah variabel yang menyimpan data dan method menyimpan kode yang dapat dieksekusi. Method merupakan fungsi yang didefinisikan di dalam kelas dan mempunyai akses pada semua variabel yang ada pada kelas tersebut. Sebuah kelas merupakan cetakan untuk membuat objek pada saat runtime. Kelas didefinisikan pada kode sumber, sedangkan objek didefinisikan pada saat runtime. Sebuah kelas didefinisikan dengan kata kunci `class`, diikuti dengan nama kelas, sejumlah parameter kelas yang dimasukkan ke dalam sebuah kurung, dan field beserta method yang dimasukkan ke dalam kurung kurawal.

Listing 2.7: Contoh definisi kelas

```
class Car(mk: String , ml: String , cr: String ){
    val make = mk
    val model = ml
    val color = cr

    def repaint ( newColor : String ) = {
        color = newColor
    }
}
```

Berdasarkan definisi kelas di atas, instansi dari kelas tersebut dapat dibuat dengan kata kunci `new`.

Listing 2.8: Pembuatan instansi kelas

```
val mustang = new Car("Ford" , "Mustang" , "Red")
val corvette = new Car("GM" , "Corvette" , "Black")
```

Pada umumnya, sebuah kelas digunakan sebagai struktur data yang mutable atau dapat diubah. Setiap objek yang menjadi instansi sebuah kelas mempunyai state yang dapat berubah-ubah. Oleh karena itu, sebuah kelas dapat mempunyai field yang didefinisikan dengan menggunakan kata kunci `var`. Penghapusan objek-objek yang sudah dibuat tidak perlu ditangani karena Scala berjalan pada JVM dan garbage collector Java sudah mengatasi hal tersebut.

### *Singleton*

Terkait dengan penggunaan struktur kelas, salah satu pola perancangan yang ada pada pemrograman berorientasi objek adalah penggunaan kelas yang hanya dapat diinstansiasi sebanyak satu kali saja. Kelas yang mempunyai sifat tersebut adalah singleton. Definisi kelas singleton pada Scala dilakukan dengan menggunakan kata kunci `object`.

Listing 2.9: Contoh definisi kelas singleton

```
object DatabaseConnection {
def open ( name : String ): Int = {
  // isi method open ()
}

def read ( streamId : Int ): Array [ Byte ] = {
  isi method read ()
}

def close (): Unit = {
  // isi method close ()
}
}
```

### Case Class

Sebuah kelas yang didefinisikan dengan kata kunci `class` akan bersifat mutable. Case class merupakan kelas yang didefinisikan dengan penambahan kata kunci `case`.

Listing 2.10: Definisi *case class*

```
case class Message ( from : String , to: String , content : String )
```

Penggunaan kata kunci tersebut memberikan beberapa keuntungan, seperti pembuatan method yang memiliki nama sama dengan nama kelas tersebut. Hal ini memungkinkan pembuatan instansi case class tanpa menggunakan kata kunci `new`.

Listing 2.11: Pembuatan instansi *case class*

```
val request = Message ( "harry" , "sam" , "fight" )
```

Selain itu, semua parameter yang ada pada definisi kelas akan menjadi field kelas yang nonmutable atau tidak dapat diubah. Sifat ini memungkinkan field kelas tersebut diakses dari luar kelas. Definisi parameter tersebut dengan penggunaan kata kunci `val` pada setiap parameter. Penggunaan case class juga memungkinkan akses pada tambahan method `toString`, `hashCode`, `equals`, dan `copy`.

#### 2.4.4 Kelas option

Kelas Option merupakan sebuah kelas yang digunakan sebagai tipe data keluaran fungsi. Kelas ini dapat menangani keluaran fungsi yang dapat berupa sebuah nilai tertentu atau berupa null. Masalah ini ditangani dengan dua kelas turunan dari Option, yaitu kelas `Some` dan kelas `None`.

Sebagai contoh, fungsi untuk mengubah tipe data `String` menjadi `Int` sebaiknya memiliki tipe keluaran berupa `Option` dan bukan menggunakan tipe keluaran `Int`. Hal ini disebabkan nilai `String` yang diberikan dapat berupa angka maupun angka yang bercampur dengan huruf. `String` yang berupa angka bercampur huruf akan menimbulkan error jika digunakan sebagai masukan fungsi perubahan dengan tipe keluaran `Int`.

Listing 2.12: Fungsi pengubah `String` menjadi `Int`

```
val request = Message ("harry", "sam", "fight")
```

Fungsi tersebut dapat menangani `String` masukan dengan format yang salah, yaitu dengan mengeluarkan nilai berupa `None`. Jika masukan tersebut benar, maka nilai yang dikembalikan tersebut adalah `Some(angka)`.

Listing 2.13: Contoh pemanfaatan kelas `Some` dan `None`

```
def toInt2 (str: String ): Option [Int] = {
  try {
    Some ( Integer . parseInt (str . trim ))
  }
  catch {
    case e: NumberFormatException => None
  }
}
```

### 2.4.5 Trait

Trait merepresentasikan antarmuka yang didukung oleh hirarki kelas yang terhubung. Trait pada Scala mempunyai kemiripan dengan interface pada Java. Akan tetapi, interface Java hanya mempunyai nama, parameter, dan tipe keluaran method sedangkan trait Scala dapat mempunyai implementasi dari method. Trait mempunyai kemiripan dengan kelas abstrak, hanya saja kelas dapat diturunkan dari satu buah kelas tetapi dapat diturunkan dari beberapa trait.

Listing 2.14: Contoh penggunaan trait

```
trait Shape {
  def area (): Int
}

class Square ( length : Int) extends Shape {
  def area = length * length
}

class Rectangle ( length : Int , width : Int) extends Shape {
  def area = length * width
}

val square = new Square (10)
```

### 2.4.6 Tuple

Pengembalian hasil dari sebuah fungsi hanya dapat berupa satu nilai saja, tetapi Scala mempunyai struktur data yang dapat mengembalikan lebih dari satu nilai. Tuple merupakan salah satu struktur data dalam Scala yang berupa wadah untuk menyimpan dua atau lebih elemen yang dapat mempunyai tipe berbeda. Tuple bersifat immutable atau tidak dapat diubah setelah dibuat. Scala

menyediakan kelas-kelas untuk setiap tuple yang mempunyai 2 sampai 22 elemen, yaitu Tuple2, Tuple3, sampai dengan Tuple22. Setiap elemen dapat diakses dengan menggunakan indeks.

Listing 2.15: Deklarasi dan penggunaan tuple

```
val tuple2 = ("Rod" , 3)
val tuple3 = ("10" , "Wombat" , true )
println ( tuple2 ._1 + "has" + tuple2 ._2 + "coconuts" )
```

### 2.4.7 Koleksi

Koleksi merupakan struktur data yang berupa tempat penyimpanan yang berisi nol atau lebih elemen. Setiap jenis koleksi pada Scala mempunyai antarmuka yang sama, sehingga penguasaan pada salah satu jenis koleksi dapat memudahkan penggunaan koleksi-koleksi Scala lainnya. Koleksi pada Scala dapat dikelompokkan menjadi tiga kategori, yaitu sequence, set, dan map.

#### Sequence

Sequence merupakan koleksi elemen dengan keterurutan tertentu. Oleh karena keterurutan tersebut, setiap elemen dapat diakses melalui posisi elemen-elemen tersebut di dalam koleksi. Array merupakan urutan elemen yang terindeks dengan tipe data yang sama. Struktur data array merupakan struktur data yang bersifat mutable karena setiap elemen dalam array dapat diubah. Panjang array bersifat tetap, sehingga tidak memungkinkan penambahan elemen baru setelah array dibuat. Indeks pada array dimulai dari 0. Untuk mengambil nilai atau melakukan perubahan pada sebuah elemen dalam array, indeks elemen tersebut perlu dicantumkan di dalam tanda kurung.

Listing 2.16: Contoh penggunaan array

```
val arr = Array (10 , 20, 30, 40) // definisi array
arr (0) = 50 // mengubah elemen indeks 0 menjadi 50
val first = arr (0) // mengambil elemen indeks 0
```

List merupakan urutan elemen linier yang mempunyai tipe data yang sama. Berbeda dengan array, list merupakan struktur data dengan nilai-nilai yang tidak bisa diubah. Elemen-elemen dari sebuah list dapat diakses dengan menggunakan indeks, tetapi list bukan struktur data yang efisien untuk melakukan akses data berdasarkan nomor indeks elemen. Hal ini disebabkan waktu akses indeks sebanding dengan banyak elemen di dalam list.

Listing 2.17: Macam-macam cara pembuatan list

```
val xs = List (10 , 20, 30, 40)
val ys = (1 to 100) . toList
val zs = someArray . toList
```

Vector adalah kelas yang merupakan gabungan dari kelas list dan array. Kelas Vector menggabungkan karakteristik kinerja dari kedua kelas tersebut dan waktu akses melalui indeks serta waktu akses linier menjadi konstan. Selain itu, kelas Vector memungkinkan akses elemen secara acak dan pengubahan elemen dengan cepat.

Listing 2.18: Contoh penggunaan Vector

```
val v1 = Vector (0, 10, 20, 30, 40)
val v2 = v1 :+ 50
val v3 = v2 :+ 60
val v4 = v3 (4)
val v5 = v3 (5)
```

### Set

Set merupakan sebuah koleksi yang terdiri dari elemen-elemen yang berbeda dan tidak terurut. Set tidak menggunakan indeks, sehingga akses elemen berdasarkan indeks tidak dimungkinkan. Set mendukung dua buah operasi dasar, yaitu `contains` untuk memeriksa apakah set tersebut mempunyai elemen yang menjadi parameter dan `isEmpty` untuk memeriksa apakah set tersebut kosong. Kedua operasi dasar tersebut akan mengembalikan hasil berupa `true` atau `false`.

Listing 2.19: Contoh deklarasi set

```
val fruits = Set("apple", "orange", "pear", "banana")
```

### Map

Map merupakan koleksi yang terdiri dari pasangan key-value. Map merupakan struktur data yang efisien untuk melakukan akses suatu nilai dengan menggunakan kunci atau key. Pada bahasa-bahasa lain, map dikenal dengan istilah kamus, associative array, atau hash map.

Listing 2.20: Contoh deklarasi dan penggunaan map

```
val capitals = Map("USA" -> "Washington_D.C.", "UK" -> "London",
                  "India" -> "New_Delhi")
val indiaCapital = capitals ("India")
```

### 2.4.8 Percabangan

Percabangan atau ekspresi bersyarat mengarahkan jalannya program berdasarkan hasil dari evaluasi syarat percobaan. Jika hasil tersebut mengembalikan nilai benar maka satu cabang kode akan dijalankan, jika tidak maka cabang kode lain yang akan dijalankan.

Listing 2.21: Contoh dasar percabangan

```
if ( inputNumber < 5)
    println ("Number is smaller than 5")
else
    println ("Number is greater than or equal to 5")
```

Percabangan dengan cabang lebih dari dua cabang dapat dilakukan dengan menggunakan kata kunci `else if`. Kata kunci tersebut dapat digunakan lebih dari satu kali dalam sebuah percabangan, berbeda dengan penggunaan `if` dan `else` yang hanya dapat digunakan masing-masing satu kali dalam sebuah percabangan pada tingkat yang sama. Selain itu, kode yang dijalankan pada percabangan perlu dimasukkan ke dalam kurung kurawal jika kode tersebut lebih dari satu baris pernyataan.

Listing 2.22: Penggunaan `else if` dalam percabangan

```
if (( executeFlag == true ) && ( firstNumber - secondNumber ) > 0) {
    positiveDiff = firstNumber - secondNumber
    println ("The positive difference : "+ positiveDiff )
} else if (( executeFlag == true ) && ( secondNumber - firstNumber ) > 0){
    positiveDiff = secondNumber - firstNumber
    println ("The positive difference : "+ positiveDiff )
} else if( executeFlag == true ) {
    println ("Two numbers are equal ")
} else {
    println ("The execution flag is not set")
}
```

### 2.4.9 Pengulangan

Pengulangan dapat dilakukan dengan menggunakan kata kunci `for` atau menggunakan `while`.

#### Pengulangan dengan `for`

Pengulangan dengan menggunakan `for` merupakan pengulangan yang dikendalikan dengan menggunakan dua buah variabel, yaitu variabel yang menyatakan titik mulai dan variabel yang menyatakan titik akhir pengulangan.

Listing 2.23: Contoh dasar penggunaan `for`

```
for (i <- 1 until 5)
  print (i+", ")
// Keluaran : 1, 2, 3, 4
```

Selain itu, `for` dapat digunakan untuk melakukan iterasi pada sebuah array.

Listing 2.24: Iterasi pada array

```
for (i <- (1 to 5).reverse)
  println (i+", ")
// Keluaran : 5, 4, 3, 2, 1
```

Pengulangan dapat dilakukan untuk menampilkan angka secara terurut menurun tanpa menggunakan array. Besarnya lompatan antara angka yang satu dengan yang lain dinyatakan dengan menggunakan kata kunci `by`.

Listing 2.25: Pengulangan angka secara terurut menurun

```
for (i <- 5 to 1 by -1)
  print (i+", ")
// Keluaran : 5, 4, 3, 2, 1
```

Pengulangan dapat dilakukan secara bertingkat dengan menggunakan titik awal yang mempunyai nama variabel berbeda. Setiap tingkatan tersebut dipisahkan dengan simbol titik koma (;) dan pengulangan dimulai dari tingkatan yang berada pada bagian paling kanan.

Listing 2.26: Pengulangan bertingkat

```
for (i <- 1 to 3; j <- 1 to 2)
  print (i+", "+j+", ")
// Keluaran : 1 ,1; 1 ,2; 2 ,1; 2 ,2; 3 ,1; 3 ,2;
```

#### Pengulangan dengan `while`

Pengulangan dengan menggunakan `while` memerlukan syarat pengulangan tersebut dihentikan dan blok kode yang akan dijalankan selama pengulangan berlangsung. Jika hasil pemeriksaan syarat tersebut mengembalikan nilai benar, maka blok kode yang ada di dalam kurung kurawal akan dijalankan. Hal tersebut akan terus berulang sampai hasil pemeriksaan syarat tersebut mengembalikan nilai salah.

Listing 2.27: Pengulangan menggunakan `while`

```
while (i < inputNumber){
  if (inputNumber % i == 0){
    isPrime = false
  }
  i += 1
}
```

Penggunaan `while` memungkinkan pengulangan dengan melakukan pemeriksaan syarat terlebih dahulu kemudian menjalankan blok kode jika hasil pemeriksaan berhasil benar. Hal ini berbeda dengan penggunaan `do-while` yang akan selalu menjalankan blok kode minimal satu kali sebelum pengulangan dihentikan.

Listing 2.28: Penggunaan do-while

```

var i = 1
do {
    print (i+", ")
    i += 1
} while (i < 5)

```

### 2.4.10 Operasi Baca Tulis File

Membaca Data dari File Operasi membaca data dari sebuah file pada Scala menggunakan library yang disediakan oleh Java. Scala hanya menyediakan library untuk mereferensikan alamat dari file yang akan dibaca dengan menggunakan kelas `Source`.

Listing 2.29: Membaca file teks

```

import java .io .{ IOException , FileNotFoundException }
import scala .io. Source

object ReadFromTextFile {
def main ( args : Array [ String ]): Unit = {
val fileName = "/Users/.../temp/InputFile.txt"
var source : scala.io.BufferedSource = null
try {
    source = Source.fromFile(fileName)
    for (line <- source.getLines()){
        println (line)}
    } catch {
        case e: FileNotFoundException => println ("File not found.")
        case e: IOException => println ("IO problem .")
        case e: Exception => println (" Something went wrong .")
    } finally {
        source.close ()
    }
}
}

```

### Menulis Data pada File

Seperti halnya pada operasi membaca file, operasi menulis file juga dilakukan dengan menggunakan library yang disediakan oleh Java.

Listing 2.30: Menulis file teks

```

import java .io .{ File , PrintWriter }
object WriteToFile {
def main ( args : Array [ String ]):Unit = {
val outFileName = "/Users/.../temp/OutputFile .txt"
var printWriter : PrintWriter = null
try {
    printWriter = new PrintWriter (new File(outFileName))
    printWriter.write("Scala is purely OO language.")
} catch {
    case e: Exception => println ("Something went wrong.")
} finally {
    printWriter . close ()
}
}

```

```

    }
}
}
}

```

## 2.5 Sistem Terdistribusi Spark

*Apache Spark* merupakan *platform* komputasi klaster yang dirancang untuk berjalan dengan cepat ketika mengolah data yang sangat besar dan untuk tujuan penggunaan umum. *Spark* merupakan penerus dari model pemrosesan *MapReduce* pada *Hadoop* dengan jenis komputasi lebih banyak yang dapat dilakukan. Salah satu fitur utama yang ditawarkan *Spark* untuk kecepatan adalah kemampuan untuk menjalankan komputasi di memori. Namun, sistem ini lebih efisien dari *MapReduce* untuk aplikasi kompleks yang berjalan pada disk karena *Spark* menggunakan *DAG(Directed Acyclic Graph) Engine* yang mengoptimasi workflow. *DAG* bekerja dengan cara menentukan jenis suatu flow yang akan memproses data yang masuk. *Spark* akan mencari cara pengerjaan mana yang paling optimal untuk melakukan pendekatan bagi masalah ini secara keseluruhan. *Spark* juga akan mengoptimasi workflow dari pengerjaan tersebut. *Spark* lebih bisa beradaptasi dengan pengerjaan pemrosesan data secara menyeluruh karena itu *spark* bisa bekerja dengan cepat.

*Spark* telah dioptimalkan untuk berjalan pada memori sehingga mempercepat pengolahan data dibandingkan dengan pendekatan alternatif lain seperti *MapReduce* pada *hadoop* yang menulis dan membaca data secara langsung pada *hard drive* komputer pada setiap tahap pemrosesan. Dengan demikian kecepatan pengolahan data menggunakan *spark* dapat menjadi lebih cepat dibandingkan dengan *Hadoop*.

*Spark* sering digunakan dalam pemanggilan kueri interaktif pada set data yang besar, pengolahan data streaming dari sensor maupun sistem finansial, dan tugas-tugas pembelajaran mesin. Selain itu, pengembangan juga dapat menggunakan *spark* untuk tugas-tugas pemrosesan data lainnya dengan memanfaatkan *library* pengembang dan API serta dukungan untuk bahasa pemrograman java, Python, R, dan Scala. *Spark* biasa digunakan bersama dengan *HDFS(Hadoop distributed File System)* sebagai pengganti media penyimpanan. *Spark* mempunyai lima buah fitur kunci, yaitu; *easy to use, fast, general purpose, scalable, dan fault tolerant*.

### 1. *Easy to Use*

*Spark* menyediakan lebih dari 80 jenis operator untuk melakukan proses pengolahan data. Sehingga pengolahan data yang lebih kompleks bisa dilakukan dengan mudah.

### 2. *Fast*

*Spark* meminimalisir akses data pada disk dengan menyimpan data pada *cache*. Sehingga pengolahan data menggunakan set data yang sama hanya memerlukan akses pada disk sebanyak satu kali.

### 3. *General Purpose*

*Spark* sudah mempunyai *library* sendiri untuk melakukan *batch processing*, analisis interaktif pada pengolahan *data stream*, pembelajaran mesin dan komputasi graf. Setiap mesin tersebut tidak memerlukan mesin klaster tersendiri untuk melakukan jenis pengolahan tertentu sehingga mengurangi kompleksitas operasional dan menghindari duplikasi kode maupun data.

### 4. *Scalability*

Sama seperti pada *Hadoop*, peningkatan kapasitas pengolahan data dapat dilakukan dengan menambahkan mesin ke dalam klaster. Selain itu, penambahan mesin pada *Spark* memengaruhi kode aplikasi yang sudah ada.

### 5. *Fault Tolerant*

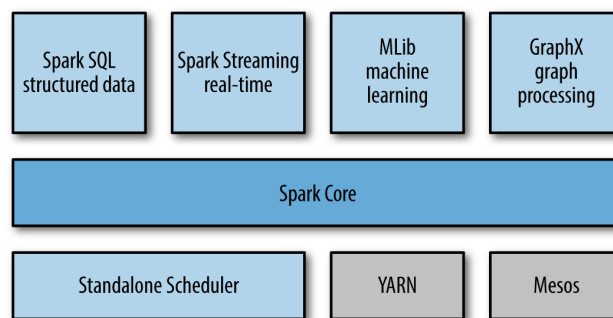
Kerusakan pada salah satu mesin pada klaster sudah ditangani oleh *spark*. Tetapi, perlu ada



kode untuk menangani hal tersebut walaupun kerusakan tersebut tidak memengaruhi kinerja aplikasi.

### 2.5.1 Susunan Spark

Sebuah proyek Spark mempunyai beberapa komponen yang terintegrasi dalam Spark. Pada intinya, Spark adalah sebuah mesin komputasi yang bertugas untuk mendistribusikan dan memantau aplikasi yang terdiri dari banyak tugas komputasi yang tersebar ke mesin pekerja atau kluster komputasi. Mesin inti Spark yang dapat berjalan cepat dengan tujuan penggunaan umum memberikan kekuatan tambahan untuk komponen dengan tingkatan yang tingkatan pada susunan yang dikhususkan untuk beban kerja yang beragam. Komponen-komponen ini dirancang untuk beroperasi dengan erat dan dapat digunakan dengan memanggil komponen ini sebagai library di dalam sebuah proyek perangkat lunak.



Gambar 2.13: *Susunan Spark*

Integrasi antar komponen yang erat tersebut memberikan beberapa keuntungan. Pertama, *library* di komponen-komponen tingkat tinggi akan mendapatkan keuntungan dari.. pada komponen-komponen ditingkat rendah. Kedua, beban untuk menjalankan susunan.. diminimalisir.

#### *Spark Core*

*Spark Core* merupakan fungsi dasar dari *Spark* dan mempunyai komponen-komponen untuk penjadwalan tugas, pengelolaan memori, pemulihan kegagalan, berinteraksi dengan sistem penyimpanan, dan lainnya. *Spark Core* juga mempunyai API untuk mendefinisikan *Resilient Distributed Dataset*(RDD) serta *Spark Context* akan dibahas lanjut pada 2.4.2

#### *Spark SQL*

Modul yang bekerja dengan data terstruktur menggunakan Hive yang memungkinkan programmer untuk menggabungkan SQL dengan bahasa pemrograman spark seperti *python*, *scala*, dan *java*.

#### *Spark Streaming*

API yang menyediakan pemrosesan data secara *real-time*.komponen-komponen dari *Spark Streaming* hampir sama dengan *Spark Core*. Seperti pengelolaan memori, pemulihan kegagalan, dan skalabilitas. *Spark Streaming* mempunyai abstraksi dan API berupa *Dstream* dan *Streaming Context*. Spark Streaming akan dibahas lanjut pada 2.4.3

#### *MLib*

Sebuah *library* untuk *machine learning* yang menyediakan beberapa tipe algoritma pembelajaran mesin yang dirancang untuk bekerja lintas kluster.

API untuk pemrosesan *graph* dan komputasi *graph-parallel*.

### *Cluster Manager*

Spark Dirancang untuk tetap efisien dalam peningkatan mesin dari satu hingga ribuan mesin. Untuk mencapai efisiensi tersebut sekaligus memaksimalkan fleksibilitas. Spark dapat menjalankan *cluster manager* termasuk Hadoop dan YARN, Apache Mesos, dan *Cluster Manager* yang sudah termasuk dalam spark yaitu Standalone Scheduler.

## 2.5.2 Application Programming Interface (API) Spark

Kemampuan komputasi pada aplikasi spark ada dalam bentuk *library*. *library* tersebut ditulis dalam bahasa scala. Tetapi, menyediakan *Applicataion Programming Interface* atau API dalam berbagai bahasa. Spark API mempunyai dua abstraksi penting, yaitu *Spark Context* dan *Resilient Distributed Datasets*(RDD). Kedua Abstraksi ini memungkinkan sebuah aplikasi untuk berinteraksi dengan Spark, terhubung dengan klaster, dan menggunakan sumber daya dalam *Cluster*.

### Spark Context

Spark Context Merupakan sebuah kelas yang didefinisikan dalam *library Spark*. Spark Context merepresentasikan sebuah koneksi ke cluster spark dan diperlukan untuk membuat objek-objek lain yang disediakan oleh *Spark API*. Sebuah aplikasi harus mempunyai objek *Spark Context* yang aktif. Objek *Spark Context* tersebut harus mempunyai konfigurasi untuk alamat spark master dan nama aplikasi. Spark Master merupakan cara SparkContext terkoneksi dengan klaster. Penggunaan kata kunci lokal menjalankan Spark dengan menggunakan sebuah thread pada satu mesin saja. Penggunaan thread tersebut dapat dikonfigurasi menjadi `local[n]` untuk n buah thread atau `local[*]` untuk menggunakan thread sejumlah core.

### Resilient Distributed Dataset(RDD)

RDD adalah abstraksi dasar untuk merepresentasikan kumpulan objek yang dapat didistribusikan pada mesin-mesin dalam sebuah klaster. RDD dapat dibuat dengan menggunakan data yang bersumber dari luar seperti file dalam HDFS, tabel basis data, atau kumpulan objek lokal dan hasil transformasi yang dilakukan pada RDD yang sudah ada. Pembuatan RDD dengan data dari sumber luar memerlukan objek *spark context*. karakteristik RDD.

#### 1. *Immutable*

RDD merupakan sebuah struktur data yang permanen. RDD yang dibuat tidak dapat dimodifikasi lebih lanjut. Sehingga operasi yang mengubah RDD akan mengembalikan RDD yang baru.

#### 2. *Partitioned*

Data yang direpresentasikan oleh RDD terbagi menjadi partisi-partisi yang didistribusikan pada klaster mesin-mesin. Akan tetapi, partisi-partisi tersebut akan berada pada sebuah mesin yang sama jika *Spark* hanya berjalan pada satu mesin saja. Di antara partisi RDD dengan partisi fisik set data terdapat pemetaan. Jenis Pemetaan tergantung pada sumber data seperti, blok-blok data HDFS mempunyai pemetaan satu ke satu dengan partisi-partisi RDD dan berapa partisi-partisi RDD dan beberapa partisi *cassandra* dipetakan ke satu buah partisi RDD.

#### 3. *Fault Tolerant*

RDD mengatasi kegagalan dari mesin klaster secara otomatis. Partisi RDD yang hilang pada mesin yang rusak tersebut akan dibuat ulang pada mesin lain. Hal ini dapat dilakukan karena spark menyimpan informasi keterhubungan antara RDD dan informasi tersebut digunakan untuk memulihkan bagian atau keseluruhan RDD yang hilang.

#### 4. *Interface*

RDD merupakan sebuah antar muka untuk pemrosesan data yang didefinisikan sebagai kelas abstrak dalam *library Spark*. RDD menyediakan antarmuka yang seragam untuk pemrosesan data yang berasal dari berbagai sumber. Selain itu RDD juga menyediakan kelas-kelas implementasi konkret untuk sumber data yang berbeda.

#### 5. *Strongly typed*

Definisi kelas RDD mempunyai parameter tipe yang memungkinkan RDD untuk merepresentasikan data dengan tipe berbeda. RDD merupakan kumpulan elemen homogen yang terdistribusi dan elemen-elemen tersebut dapat bertipe *Integer*, *Long*, *Float*, *String*, atau tipe yang didefinisikan oleh pengembang aplikasi.

#### 6. *In Memory*

Kelas RDD menyediakan API untuk komputasi kluster dalam memori. *Spark* memungkinkan RDD untuk di-cache atau dipertahankan dalam memori. Operasi pada RDD yang ada dalam cache berjalan lebih cepat dibandingkan dengan operasi pada RDD yang tidak berada dalam cache.

Data yang sudah direpresentasikan dalam RDD dapat dioperasikan dengan menggunakan dua jenis operasi dasar, yaitu transformasi dan aksi.

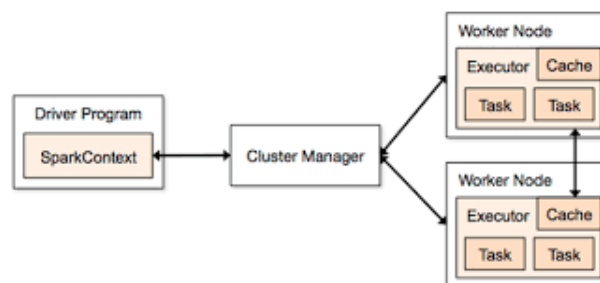
##### 1. *transformasi*

RDD yang menjadi masukan dari sebuah operasi transformasi akan mengalami perubahan struktur atau nilai yang ada pada RDD. karena RDD bersifat *immutable*, maka perubahan dari operasi ini secara konseptual akan dikembalikan dalam bentuk RDD baru.

##### 2. *Aksi*

Operasi menjadi titik awal dari komputasi-komputasi yang telah terjadi pada RDD masukan. Pemanggilan operasi aksi akan memulai pembentukan RDD yang diperlakukan untuk komputasi. Operasi ini menerima masukan berupa RDD dan memberikan hasil berupa sebuah nilai. Operasi yang dilakukan pada RDD bersifat *lazy* yang berarti operasi tersebut tidak akan dieksekusi oleh spark sampai ada operasi aksi. Evaluasi *lazy* berarti ketika ada pemanggilan operasi transformasi pada RDD. Operasi tersebut tidak langsung dilakukan spark mencatat metadata untuk menandakan bahwa operasi tersebut sudah pernah diminta. Dengan demikian, RDD dapat disebut sebagai kumpulan instruksi untuk melakukan komputasi data yang dibuat dari kumpulan operasi transformasi.

### 2.5.3 Arsitektur Apache Spark



Gambar 2.14: Arsitektur *Spark*

Berdasarkan pada gambar sebuah aplikasi melibatkan lima entitas penting yaitu; driver program, cluster manager, worker, executor, dan task.

1. *Driver Program*

Driver Program merupakan bagian dari aplikasi yang memulai menjalankan proses pengolahan data yang akan dilakukan. Driver program terhubung dengan komponen-komponen lain melalui objek *Spark Context* yang terdapat dalam komponen ini. Objek *Spark Context* akan melakukan koneksi dengan *cluster manager*. Setiap klaster hanya akan mempunyai satu buah driver program atau satu buah objek spark context.

2. *Cluster Manager*

*Cluster Manager* berfungsi untuk mengelola sumber daya yang digunakan untuk proses pengolahan. Spark Context pada komponen driver program dapat terhubung dengan salah satu dari jenis-jenis cluster manager yang didukung oleh Spark, yaitu *cluster manager* seperti *Apache mesos* dan *Yarn*. Setiap Cluster hanya memiliki satu buah *cluster manager*.

3. *Worker*

Worker merupakan komponen yang menyediakan unit pemrosesan dan alokasi memori untuk menyimpan sumber daya yang digunakan dalam proses yang berjalan. Setiap klaster dapat memiliki lebih dari satu buah worker dan setiap worker tersebut akan menjalankan aplikasi sebagai proses yang terdistribusi pada sebuah klaster.

4. *Executor*

*Executor* merupakan proses yang dibuat oleh *spark* pada setiap node worker untuk menjalankan aplikasi. *Executor* yang terdapat pada setiap worker hanya dapat menangani proses untuk sebuah aplikasi saja, sehingga aplikasi yang berbeda akan mempunyai eksekutor yang berbeda. Setiap eksekutor mempunyai lama hidup yang sama dengan aplikasi sehingga akan berhenti ketika aplikasi berhenti berjalan. Setiap Klaster dapat mempunyai lebih dari satu eksekutor dan jumlah tergantung pada banyak aplikasi.

5. *Task*

*Task* merupakan unit pekerjaan terkecil yang dikirimkan ke executor. Unit pekerjaan tersebut akan dijalankan pada sejumlah *thread* yang terdapat pada executor. Banyak *Thread* yang digunakan berbanding lurus dengan banyak partisi data yang diolah.

#### 2.5.4 Spark Streaming

*Spark Streaming* adalah ekstensi dari API *Spark Core* yang menyediakan pemrosesan *Data Stream* yang bisa ditingkatkan performanya dengan menambah *hardware baru*, bisa memproses data dengan banyak dan cepat, dan sistem masih bisa beroperasi ketika terjadi kegagalan. Data bisa dikumpulkan dari berbagai sumber seperti *Kafka*, *Flume*, *Kinesis*, atau *TCP Socket*. Data yang terkumpul akan langsung diproses dengan algoritma yang kompleks seperti *Map*, *Reduce*, *Join* dan *Windowing*. Terakhir data yang telah diproses langsung dikirim ke *File Systems*, *database* dan *live dashboard*. Penjelasan lebih jelas ada pada gambar 2.12 berikut:

Gambar 2.15: Arsitektur *Spark Streaming*

### Arsitektur Spark Streaming

*Spark Streaming* bekerja dengan cara menerima input *data streams* secara langsung dan membagi data tersebut menjadi beberapa potongan-potongan (*batches*), yang nanti akan diproses oleh mesin *Spark* untuk menghasilkan *stream* akhir pada *batches*. *Spark Streaming* tidak memproses data secara periodik, hanya memproses yang duluan datang dan memutakhirkan hasil dari *Spark Streaming* dari waktu ke waktu. Data langsung bisa dianalisis ketika datang dengan mengelompokkannya ke beberapa bagian kecil dan langsung melakukan agregasi pada potongan data tersebut.

Gambar 2.16: Arsitektur *Spark Streaming*

Berdasarkan gambar 2.13 *Data Streams* yang masuk akan diterima oleh *reciever* lalu potongan-potongan data yang masuk pada selang waktu tertentu akan dihasilkan secara terus menerus dengan kata lain proses transformasi akan terus berlangsung ketika program dihentikan. Lalu data yang telah dihasilkan dapat dikirimkan langsung ke *external database* dengan data yang telah diagregasi sebelumnya.

transformasi dan aksi pada RDD bisa terjadi secara paralel pada node *worker*. Artinya, proses RDD dibagi menjadi potongan kecil dan didistribusikan, potongan yang berbeda akan dikirim ke node yang berbeda. Potongan RDD tersebut akan didistribusikan ke seluruh kluster.

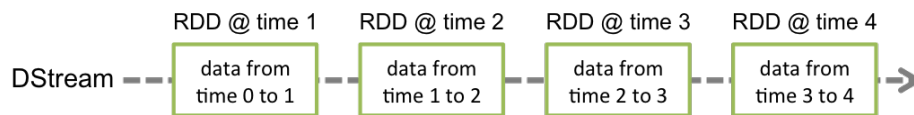
Aplikasi *Spark Streaming* membutuhkan pengaturan tambahan untuk beroperasi tanpa henti. Aturan yang dimaksud adalah *checkpointing* yang merupakan mekanisme utama pada *Spark Streaming*. *Checkpointing* memungkinkan penyimpanan data pada *file system* seperti HDFS dan yang membuat *Spark Streaming* menjadi *fault tolerant*

### Abstraksi Spark Streaming

Abstraksi dasar yang disediakan oleh *Spark Streaming* disebut *Discretized Streams (DStreams)*. *Dstream* merupakan seluruh alur data yang datang dari *receivers*. Setiap *Dstream* dibuat pada potongan-potongan RDD yang merepresentasikan aliran data yang kontinu. *Dstream* memiliki dua buah operasi; transformasi dan *output operation*. Transformasi bertugas untuk menghasilkan *Dstream* baru dan *Output operation* bertugas untuk menuliskan data ke sistem eksternal. *Dstream*

menyediakan operasi yang hampir sama dengan operasi RDD dan mempunyai operasi sendiri yang digunakan untuk mengatur waktu seperti *sliding window*.

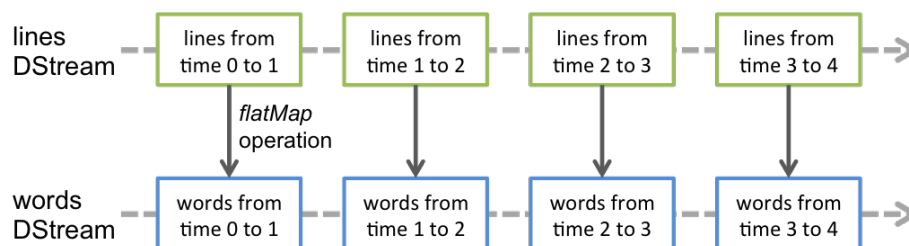
setiap RDD yang ada pada Dstream mempunyai data dari interval tertentu yang akan ditunjukkan pada gambar 2.14



Gambar 2.17: Alur *Dstream*

Pada setiap awal interval, *batch* baru selalu dibuat dan setiap data yang muncul pada interval tersebut akan dimasukan ke *batch* tersebut. Saat interval berakhir *batch* telah selesai berkembang. Ukuran dari suatu interval ditentukan oleh sebuah parameter yang disebut *batch interval*. Biasanya, ukurandari interval berkisar antara 500 milidetik sampai beberapa detik. Setiap input yang ada di dalam batch membentuk RDD dan diproses menggunakan *Spark Jobs* untuk membuat RDD lain. RDD akan terus dibuat dan ditransformasi terus menerus sampai ada aksi yang memberhentikan *Dstream*. Hasil dari transformasi *Dstream* akan langsung dikirimkan ke sistem eksternal dalam bentuk *batch*.

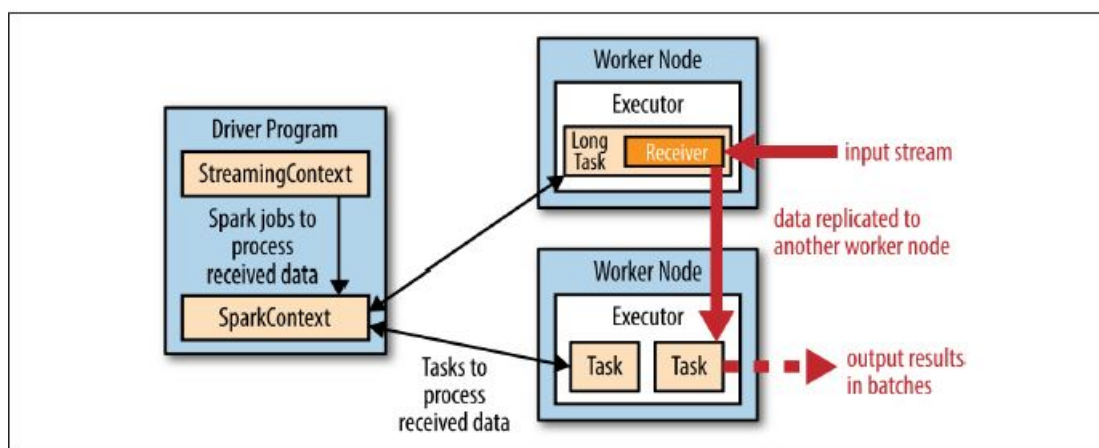
*Dstream* merupakan salah satu abstraksi yang paling memudahkan pada spark streaming karena transformasi langsung diterapkan ke *Dstream* bukan masing-masing RDD. Jadi, jika melakukan transformasi pada Dstream seluruh potongan RDD akan ikut bertransformasi. Namun, masing-masing RDD masih bisa diakses melalui *Dstream*.



Gambar 2.18: mengubah *Data Stream* dari lines ke words

*DStream* dapat dibuat dari sumber eksternal ataupun menggunakan hasil transformasi dari *Dstream* lain. *DStream* juga memiliki *Stateful transformations* yang bisa mengagregasi data pada seluruh interval yang ada. pembahasan tentang *stateful transformation* akan dibahas lebih jelas di bab berikutnya.

Untuk setiap sumber input, *spark streaming* meluncurkan *recievers* yang mana adalah *task* yang berjalan pada eksekutor yang mengumpulkan data dari sumber input dan menyimpannya sebagai RDD. Selain menyimpan data, *recievers* juga mereplikasi data ke eksekutor lain untuk mencapai *fault-tolerance*. Data akan disimpan di memori eksekutor sama seperi *cache* pada RDD. *Receivers* juga bisa mereplikasi data ke HDFS. *Streaming Context* pada *driver* than secara periodik menjalankan *Spark Jobs* untuk memproses data dan menggabungkannya dengan RDD sebelumnya.



Gambar 2.19: eksekusi *Spark Streaming* pada komponen *Spark*

*Spark Streaming* memiliki sifat *fault-tolerant* yang sama dengan *Spark* untuk RDD selama replika input data masih tersedia. *Spark Streaming* bisa mengkomputasi ulang setiap *state* yang diturunkan dari *lineage* suatu RDD dengan cara menjalankan kembali operasi yang memproses RDD tersebut. Biasanya, data yang diterima direplika dalam dua node sehingga *spark streaming* bisa mentoleransi satu worker yang gagal. Namun, jika menggunakan *lineage* penghitungan ulang bisa memerlukan waktu yang lama karena datanya telah dibuat duluan. Karena itu, *Spark Streaming* menyediakan mekanisme yang disebut *checkpointing* yang akan menyimpan state secara berkala ke suatu file system seperti HDFS. *checkpointing* akan dijalankan setiap lima atau sepuluh *batch*. Ketika terjadi ingin memperbaiki data yang gagal *Spark Streaming* hanya perlu kembali ke *checkpoint* paling baru.

## Transformasi

Transformasi pada *Spark Streaming* dikelompokkan menjadi dua yaitu; *stateless* atau *Stateful*:

- Pada transformasi *stateless*, pemrosesan setiap batch tidak bergantung pada data di batch sebelumnya. Transformasi ini memiliki transformasi RDD seperti `map()`, `reduce()`, dan `reduceByKey()`
- Transformasi *stateful* menggunakan data yang dihasilkan oleh *batch* sebelumnya untuk menghitung hasil dari *batch* saat ini. Transformasi ini memiliki *sliding windows* dan bisa mengecek waktu pada seluruh interval.

### Stateless Transformations

Transformasi *Stateless* adalah transformasi RDD sederhana yang diterapkan kepada setiap RDD pada *Dstreams*. Walaupun setiap fungsi terlihat diterapkan ke pada seluruh aliran data. Namun, secara internal setiap *DStream* tersusun dari beberapa RDD (*batches*) dan setiap transformasi *stateless* diterapkan secara terpisah untuk setiap RDD. Contohnya, `reduceByKey()` akan melakukan *reduce* pada data pada setiap *batch interval*. Untuk menggabungkan data pada seluruh interval diperlukan *Stateful Transformation*.

### Stateful Transformation

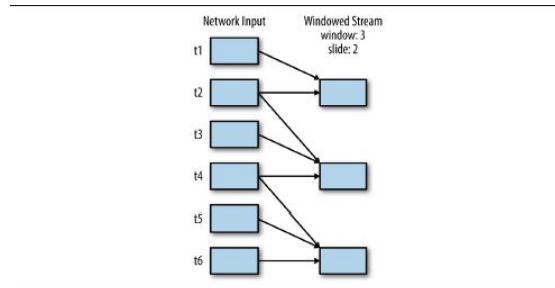
*Stateful Transformation* adalah sebuah operasi pada *Dstream* yang bisa menelusuri waktu pada semua interval sehingga data pada *batch* sebelumnya bisa digunakan untuk *batch* saat ini.

*Spark Streaming* memerlukan *checkpointing* untuk bisa diaktifkan di *streaming context* sebagai upaya menghindari kesalahan *fault*.

### Windowed Transformation

Transformasi ini menghitung hasil pada interval yang lebih lama dari *streaming context* dengan

menggabungkan beberapa *batch* pada interval tertentu. pada transformasi *windowing* ada tiga interval yang digunakan: *batch*, *slide*, dan *window interval*. *Batch Interval* adalah seberapa sering suatu data diambil ke dalam *Dstream*. Durasi dari batch interval sangat sebentar setengah sampai satu detik. *Batch time* tidak berkorelasi dengan apa yang akan dianalisis nanti. *Batch Interval* hanya mengambil data sebanyak dan secepat mungkin dari suatu sumber data.



Gambar 2.20: Gambar cara kerja *Windowed Transformation*

*Slide Interval* adalah titik acuan seberapa sering suatu informasi ingin dikomputasi, dan *Window Interval* adalah bagaimana *Spark Streaming* melihat ke belakang setiap kali bertemu dengan *slide Interval*.

## Output Operations

*Output Operation* adalah fungsi akhir dari rantai transformasi. Fungsi ini menentukan apa yang perlu dilakukan dengan data yang ditransformasikan pada akhir aliran. contoh: tampilkan ke layar atau simpan di database eksternal.

fungsi *Output Operation* yang paling umum adalah `print()`. Fungsi ini menampilkan 10 element pada tiap *batch* sebagai hasil. selain itu ada `saveAsTextFile("dir",name)` fungsi ini akan menyimpan hasil transformasi ke direktori yang telah ditentukan. Terakhir, ada `foreachRDD()`. Fungsi ini hampir sama dengan `transform` dimana kita memiliki akses pada setiap RDD. Sebagai contoh jika ingin menyimpan ke eksternal tabel *MySQL* tidak bisa menggunakan *SaveAs* tetapi harus mengakses tiap RDD dan memasukannya ke tabel *MySQL* secara manual.

## Checkpointing

*Checkpointing* adalah mekanisme utama sebagai penyedia *fault tolerance* untuk *Spark Streaming*. Mekanisme ini memungkinkan *sparkstreaming* untuk menyimpan data secara periodik pada suatu *storage system* seperti HDFS. Data yang disimpan ini adalah *backup dari data asli*. Tujuan menyimpan data backup adalah:

- mengurangi transformasi atau state yang harus dikomputasi ulang jika terjadi kegagalan. *Spark Streaming* bisa mengkomputasi ulang data yang hilang dengan lineage graph. Tetapi, *checkpointing* yang menentukan seberapa jauh lineage graph harus mengambil data.

Listing 2.31: contoh checkpointing

```
ssc.checkpoint("hdfs://...")
```



## 2.6 Input Sources

*Input Source* adalah penyedia data yang bisa terhubung dengan *Spark Streaming*. *Input Source* adalah komponen terpisah dari *Spark Streaming* walaupun masih bagian dari *Spark*. Untuk mengintegrasikan dengan *Spark Streaming* dibutuhkan beberapa tambahan *package* yang harus diikutsertakan pada *build* file. Beberapa *Input Source* adalah Twitter dan Kafka:

### 2.6.1 Twitter API

Twitter API adalah sekumpulan URL yang digunakan untuk mengakses data pada twitter tanpa melewati antar muka web. URL akan digunakan sebagai parameter kode program nanti. data yang diambil pada twitter berupa objek. seperti contoh gambar:



Gambar 2.21: *Twitter Object*

objek yang diakses akan disimpan dengan format JSON. Objek terdiri dari informasi-informasi yang membangun tweet seperti; tanggal berapa suatu tweet dibuat, id twitter yang menunggah tweet tersebut, isi pesan yang diunggah oleh pengguna(status), informasi pengguna itu sendiri, dan entitas luar yang ikut di dalam suatu tweet seperti link url atau mention. Berikut adalah contoh format JSON yang membangun suatu Tweet:

```
{
  "created_at": "Wed Oct 10 20:19:24 +0000 2018",
  "id": 1050118621198921728,
  "id_str": "1050118621198921728",
  "text": "To make room for more expression, we will
tps://t.co/MkGjXf9aXm",
  "user": {},
  "entities": {}
}
```

Gambar 2.22: *Twitter Object JSON*

Informasi tentang pengguna terdiri dari beberapa objek lagi. Objek yang dimuat berupa id user, nama, lokasi, deskripsi, status verifikasi, jumlah follower, jumlah following, dan lokasi.

```

{ "user": {
  "id": 6253282,
  "id_str": "6253282",
  "name": "Twitter API",
  "screen_name": "TwitterAPI",
  "location": "San Francisco, CA",
  "url": "https://developer.twitter.com",
  "description": "The Real Twitter API. Tweets about API cl
e issues and our Developer Platform. Don't get an answer? It
te.",
  "verified": true,
  "followers_count": 6129794,
  "friends_count": 12,
  "listed_count": 12899,
  "favourites_count": 31,
  "statuses_count": 3658,
  "created_at": "Wed May 23 06:01:13 +0000 2007",

```

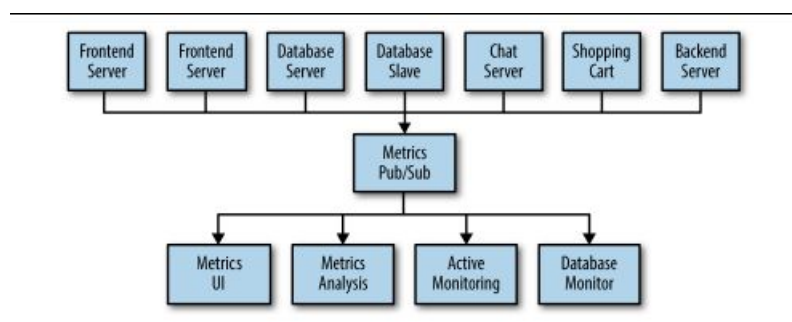
Gambar 2.23: *User Object JSON*

Namun dari beberapa banyak informasi yang terdapat pada twitter seseorang tidak semuanya bisa diakses hal ini bergantung ke pada kebijakan dari twitter dan persetujuan dari user. salah satu contohnya adalah lokasi seseorang. Lokasi seseorang hanya bisa diakses ketika user bersedia untuk menampilkan lokasi tersebut.

## 2.6.2 Kafka

### Konsep Publish/Messaging

Sebelum membahas Kafka, penting untuk mengerti tentang sistem pengiriman *publish/messaging* dan mengapa konsep ini sangat penting. *publish/subscribe messaging* adalah *pattern* yang dicirikan oleh pengirim *publischer* tidak langsung mengirimkannya ke penerima. Tetapi, pengirim mempublikasikan data yang dimiliki ke sebuah sistem lain yang disebut *broker*, sebuah titik sentral di mana suatu pesan selalu diunggah. Sehingga penerima pesan bisa langsung mengikuti perkembangan data dan informasi yang dimiliki *publisher* di broker. Penerima disebut *Subscriber*.

Gambar 2.24: *Publisher/Subscriber*

seperti gambar di atas semua server mengirimkan data ke broker *metrics* dan sistem-sistem yang ingin memiliki data yang ingin diakses harus diintegrasikan dengan broker. Sistem-sistem tersebut mengikuti perkembangan dan perubahan data yang terjadi melalui broke (subscribe).

## Pengertian Kafka

kafka adalah sebuah sistem pengiriman data *Publish/subscribe* yang didesain untuk menyelesaikan masalah yang sering disebut dengan *distributed commit log* yang mana suatu *filesystem* atau database didesain untuk menyediakan data record-record yang disimpan dengan lama dan bisa diakses kembali secara berkala pada sistem yang stabil. Data pada kafka disimpan dengan lama dan terurut. Berikut adalah komponen-komponen dari kafka:

### Messages and Batches

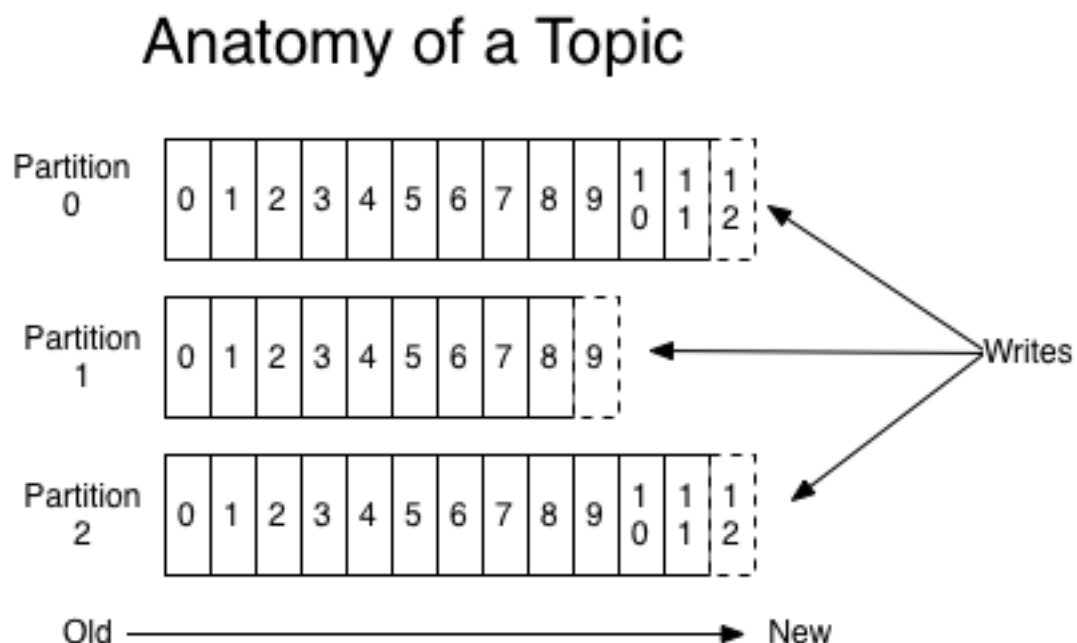
Satuan data di kafka disebut dengan *message*. *Message* hampir sama seperti *row* atau *rekord*. Sebuah *message* adalah array dari sekumpulan bytes karena itu *message* pada kafka tidak memiliki format spesifik atau arti bagi kafka. Suatu *message* bisa memiliki metadata yang disebut dengan *key*. Untuk lebih efisien semua *messages* ditulis pada *batch*. *batch* adalah sekumpulan pesan yang diproduksi oleh topik dan partisi yang sama.

### Schemas

Kafka mengetahui *message* adalah sebuah *array of bytes*. Sehingga kafka membutuhkan skema atau struktur yang diterapkan pada pesan sehingga bisa mudah dimengerti. Ada banyak cara untuk menerapkan skema tergantung kebutuhan aplikasi. Contoh dari skema bisa berbentuk JSON atau XML sehingga manusia bisa membaca pesan yang ada pada kafka.

### Topics

*Message* pada kafka disebut sebagai *topics*. *Topics* adalah sebuah kategori atau sebuah nama *feed* dimana suatu record dipublikasi. Analoginya, topik adalah tabel basis data atau suatu folder di filesystem. Suatu record bisa disimpan di folder atau basis data dengan identifikasi pengenalan.

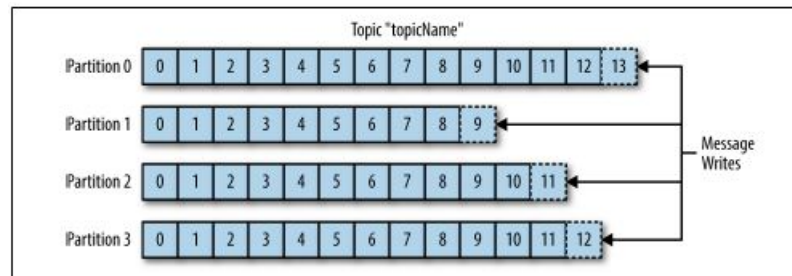


Gambar 2.25: *Topic* pada *kafka*

Setiap partisi adalah sebuah sekuensi yang terurut, tidak bisa diubah-ubah *immutable* yang terus ada pada *commit log*. Setiap record pada *ppartition* diberi tanda dengan nomer sekuensial yang

disebut offset yang menandai rekord secara unik. Partition adalah tempat dimana topic disimpan.

Klaster pada kafka akan terus menyimpan topic terlepas topik itu digunakan atau tidak selama waktu yang telah ditentukan (*retention period*). Contoh jika suatu *retention period* pada topic diatur menjadi 2 hari maka topic tersebut akan ada selama dua hari dan tidak bisa dihapus pada interval waktu itu. Sehingga setiap *subscriber* masih bisa mengakses data tersebut. Setelah itu baru dihapus.



Gambar 2.26: *stream topic*

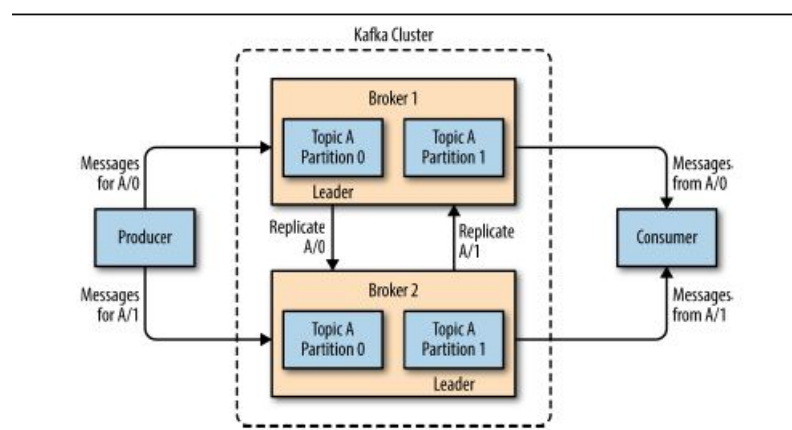
Begitu juga dengan *data stream*. Pada kafka, *Data Stream* dianggap sebagai satu topic terlepas dari banyak partisi. Sebagai contoh keempat partisi pada gambar di atas masih disebut sebagai satu stream.

## Producers and Consumers

Pengguna yang menggunakan sistem kafka disebut dengan klien. Ada dua tipe kline *producer* dan *consumer*. Bisa juga kakfa diintegrasikan dengan sistem API lain. Tugas dari *producer* adalah membuat *membuat message* dan mengirimnya ke *topic* tertentu pada broker. *Consumer* adalah yang membaca pesan degan cara mengakses topic-topic tertentu pada broker.

## Broker

Sebuah kafka server disebut dengan broker. Sebuah broker menerima pesan dari *producer* memberi *offset* pada pesan tersebut dan menyimpannya pada sebuah disk. Broker juga berinteraksi dengan *consumer* ketika konsumen meminta akses pada suatu partisi dan membalas *consumer* dengan pesan yang diinginkan dan ada di disk.



Gambar 2.27: Kafka Broker

*Broker* dirancang untuk berjalan pada klaster. Dari rangkaian *broker* pada klaster satu *broker* akan bertindak sebagai *controller* yang dipilih secara otomatis dan berfungsi untuk mengatur operasi

---

administratif seperti; menentukan partisi mana topic akan disimpan atau mengawasi jika terjadi *failure* pada broker lain. Sebuah partisi bisa disimpan pada dua broker secara bersamaan.



## BAB 3

### STUDI EKSPLORASI

Pada Bab ini akan dibahas Konfigurasi Spark, Twitter API, dan Kafka pada perangkat dengan sistem operasi windows. Selain itu, pada bab ini akan dijelaskan contoh eksekusi program Spark Streaming pada TCP Socket dan Twitter API. Serta, mengambil data dari Kafka.

### 3.1 Konfigurasi Klaster

#### 3.1.1 Konfigurasi Hadoop

Hadoop merupakan framework yang dibuat untuk berjalan pada sistem operasi berbasis Linux, sehingga versi-versi awal Hadoop tidak dapat digunakan untuk sistem operasi Windows. Penggunaan Hadoop untuk Windows dapat dilakukan mulai Hadoop versi 2.x dengan menggunakan file-file tambahan. Sebelum melakukan konfigurasi, berikut ini adalah komponen-komponen yang diperlukan untuk dapat melakukan konfigurasi dan menjalankan Hadoop pada sistem operasi Windows 10 x64.

- Java JDK 8
- Paket biner Hadoop versi 3.x.
- Paket winutils dengan versi yang sama dengan versi Hadoop yang digunakan
- Microsoft Visual C++ 2010 Redistributable

Hadoop berjalan dengan menggunakan virtual machine milik Java, sehingga instalasi Java perangkat diperlukan terlebih dahulu. Berdasarkan gambar 3.1, pengguna disarankan mengganti tempat instalasi Java yang digunakan dengan mencentang pilihan Change destination folder. Tempat instalasi awal Java pada umumnya akan berada di `C:\Program\Files\Java` atau `C:\Program\Files\ (x86)\Java`. Hal ini dapat menimbulkan masalah pada Hadoop karena Hadoop tidak mendukung penggunaan karakter spasi pada nama direktori. Oleh karena itu, direktori tempat instalasi Java diubah menjadi direktori lain dengan nama yang tidak menggunakan karakter spasi.



Gambar 3.1: Instalasi Java

Setelah instalasi selesai dilakukan, environment variable untuk JAVA \_HOME perlu ditambahkan. Nilai untuk *environment variable* tersebut merupakan direktori instalasi Java pada perangkat. Seperti yang sudah disebutkan sebelumnya, direktori tersebut disarankan tidak menggunakan karakter spasi sesuai dengan yang ditunjukkan pada gambar 3.2.

Variable	Value
HADOOP_HOME	C:\Hadoop\hadoop-3.1.2
JAVA_HOME	C:\java\jdk-8.0.222.10-hotspot\
OneDrive	C:\Users\S430FN\OneDrive
Path	C:\Users\S430FN\AppData\Local\Microsoft\WindowsApps;C:\...

Gambar 3.2: *Environment variable* untuk JAVA \_HOME dan HADOOP \_HOME

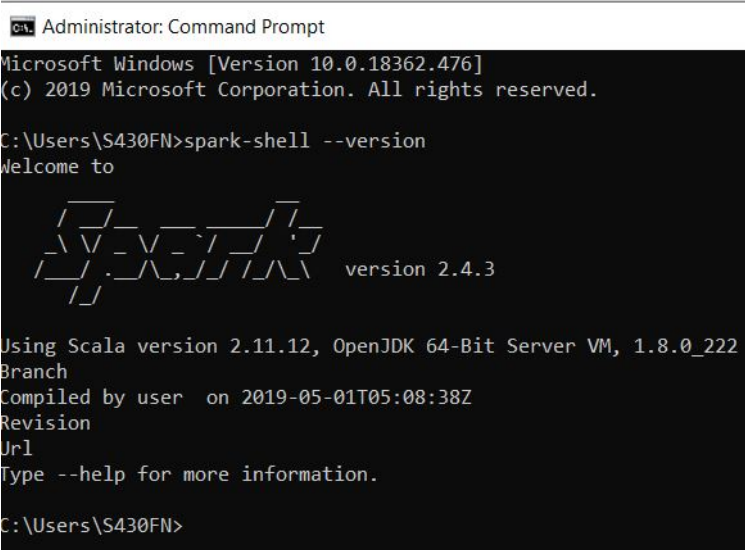
Setelah instalasi dan penambahan environment variable, file-file biner Hadoop dapat diekstraksi pada sebuah direktori. Direktori tersebut sebaiknya memiliki nama yang cukup ringkas dan tidak menggunakan karakter spasi. Untuk memudahkan penggunaan Hadoop, alamat direktori tersebut dapat ditambahkan sebagai environment variable dengan nama HADOOP \_HOME seperti yang ditunjukkan pada gambar 3.2. Konfigurasi klaster yang perlu dilakukan berupa konfigurasi untuk HDFS dan MapReduce untuk klaster single node. Konfigurasi tambahan perlu dilakukan pada file masters dan slaves untuk penggunaan klaster multi node.

Konfigurasi HDFS dilakukan dengan membuat atau mengisi file-file `hadoop-env.cmd`, `core-site.xml`, dan `hdfs-site.xml`.

### 3.1.2 Konfigurasi Spark

Klaster yang digunakan Spark pada skripsi ini adalah klaster yang sama dengan Hadoop. Versi Spark yang digunakan untuk pengembangan aplikasi adalah Spark 2.4.3 dengan sistem operasi Windows 10 x64. Berikut ini adalah langkah-langkah melakukan konfigurasi Spark pada komputer-komputer bagian klaster. Instalasi Spark hanya mengatur *environment variable* sesuai dengan direktori Spark dan menambah environment ke path. Namun harus terdapat Hadoop yang telah terinstal sebelumnya. Setelah semua terinstal dan perintah `Spark-shell -version` dipanggil akan muncul layar seperti ini:





```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.18362.476]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\S430FN>spark-shell --version
Welcome to

  ____  _
 / ___|| | | |
| |___| |_| |
 \___|_____|_|

version 2.4.3

Using Scala version 2.11.12, OpenJDK 64-Bit Server VM, 1.8.0_222
Branch
Compiled by user on 2019-05-01T05:08:38Z
Revision
Url
Type --help for more information.

C:\Users\S430FN>
```

Gambar 3.3: Gambar Spark berhasil diinstal

Jika Spark telah terinstal perintah scala akan langsung bisa dijalankan.

## 3.2 Konfigurasi API dan Data Collector

Karena data yang dibutuhkan untuk Spark Streaming harus besar dan real-time maka dibutuhkan sistem lain yang terintegrasi sebagai penyedia data. Beberapa contoh sistem tersebut adalah Twitter API dan Kafka. Twitter API menyediakan data hanya dari twitter saja. Sedangkan, Kafka bisa menerima data hampir dari semua API.

### 3.2.1 Konfigurasi TCP Socket

Sumber data TCP Socket tidak perlu diinstalasi terlebih dahulu karena tidak ada interverensi dari pihak ketiga yaitu sang penyedia data. TCP socket langsung bisa menerima data dengan mengakses port lokal dan akan langsung terintegrasi dengan IP address dan port yang kita miliki. Mengkonfigurasi TCP Socket hanya perlu menyediakan sebuah port kosong yang nantinya akan digunakan data untuk masuk. Tetapi, konfigurasi pada kode program masih diperlukan.

### 3.2.2 Konfigurasi Twitter API

Sebelum mendapatkan hak untuk mengakses Twitter API, user perlu melakukan pendaftaran ke pihak twitter untuk menjadi *developer account* terlebih dahulu. Akan muncul survei yang menanyakan tentang privasi data pengguna twitter. Pertanyaan paling umum adalah data yang didapatkan akan digunakan untuk apa. Untuk kasus skripsi ini, data akan digunakan untuk mempelajari Spark Streaming. Berikut adalah cara membuat API Setelah dikonfirmasi menjadi *developer account*:

1. Membuat Aplikasi yang akan digunakan sebagai API.
2. Mengisi keterangan dan informasi tentang aplikasi tersebut
3. Menggunakan keys dan tokens yang didapatkan yang akan digunakan untuk integrasi dengan Spark Streaming

keys dan tokens yang didapatkan adalah beberapa nomer seri yang disebut *Consumer API* dan *Access token*. Nomor seri ini yang nantinya akan jadi parameter bagi kode program untuk mengakses

Aplikasi yang telah kita buat. Nomor seri ini bersifat rahasia jadi tidak boleh tersebar ke pihak lain karena seseorang bisa mengakses Aplikasi pengumpul data yang telah kita buat. Berikut contoh gambar dari key dan tokens:



Gambar 3.4: Gambar Token yang digunakan sebagai parameter

Parameter keys dan token nantinya akan dimuat dalam `twitterAuth.txt`. File tersebut akan dijadikan sebagai otentikasi untuk mengakses data yang ada di Twitter.

### 3.2.3 Konfigurasi Kafka

kafka adalah sebuah sistem pengiriman data *Publish/subscribe* yang didesain untuk menyelesaikan masalah yang sering disebut dengan *distributed commit log* yang mana suatu *filesystem* atau database didesain untuk menyediakan data rekord-rekord yang disimpan dengan lama dan bisa diakses kembali secara berkala pada sistem yang stabil. Kafka akan menjadi Input Sources bagi spark streaming. Sebelum instalasi kafka berikut komponen-komponen yang dibutuhkan untuk menjalankan kafka di windows 10 x64:

- Java JDK 8
- Zookeeper
- Paket winutils dengan versi yang sama dengan versi Hadoop yang digunakan

karena zookeeper berjalan di atas java maka sebelum instalasi zookeeper dan kafka harus menginstal java JDK 8. instalasi Java yang digunakan dengan mencentang pilihan **Change destination folder**. Tempat instalasi awal Java pada umumnya akan berada di `C:\Program\Files\Java` atau `C:\Program\Files\ (x86)\Java`. Hal ini dapat menimbulkan masalah pada Zookeeper karena Zookeeper tidak mendukung menggunakan karakter spasi pada nama direktori. Oleh karena itu, direktori tempat instalasi Java diubah menjadi direktori lain dengan nama yang tidak menggunakan karakter spasi.



Gambar 3.5: Instalasi Java

Setelah menginstal java harus menginstal zookeeper terlebih dahulu karena kafka menggunakan zookeeper untuk menyimpan metadata tentang klaster kafka. Zookeeper yang digunakan adalah zookeeper versi 3.5.x. Lalu, ubahlah konfigurasi pada `zoo.cfg` dan ubahlah direktori tempat metadata kafka mau disimpan. contoh `\zookeeper-3.5.6\data`. setelah itu, *environment variable* untuk `ZOOKEEPER_HOME` perlu ditambahkan. Nilai untuk *environment variable* tersebut merupakan direktori instalasi ZOOKEEPER pada perangkat. Seperti yang sudah disebutkan sebelumnya, direktori tersebut disarankan tidak menggunakan karakter spasi. sesuai dengan yang ditunjukkan pada gambar 3.2.

Path	C:\Users\S430FN\AppData\Local\Microsoft\WindowsApps\C\...
SPARK_HOME	C:\Spark\spark-2.4.3-bin-hadoop2.7
TEMP	C:\Users\S430FN\AppData\Local\Temp
TMP	C:\Users\S430FN\AppData\Local\Temp
ZOOKEEPER_HOME	c:\zookeeper-3.5.6

Gambar 3.6: ZOOKEEPER\_HOME

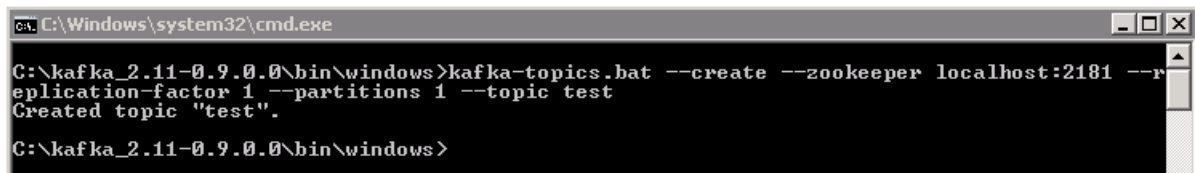
Untuk sistem *standalone* port zookeeper bisa diatur. tetapi port default adalah 2181. Untuk menjalankan Zookeeper lakukan perintah `zkserver`. Sebelum menginstal broker pastikan zookeeper berjalan terlebih dahulu. Setelah zookeeper berhasil terinstal. perlu menjalankan broker karena *consumer* dan *producer* membutuhkan *broker* untuk bisa diinstal dan saling berkomunikasi. perintah untuk menjalankan broker: `.\bin\windows\kafka-server-start.bat .\config\server.properties`. Setelah menjalankan broker dan zookeeper, membuat *topics* untuk kafka.



Gambar 3.7: menjalankan server

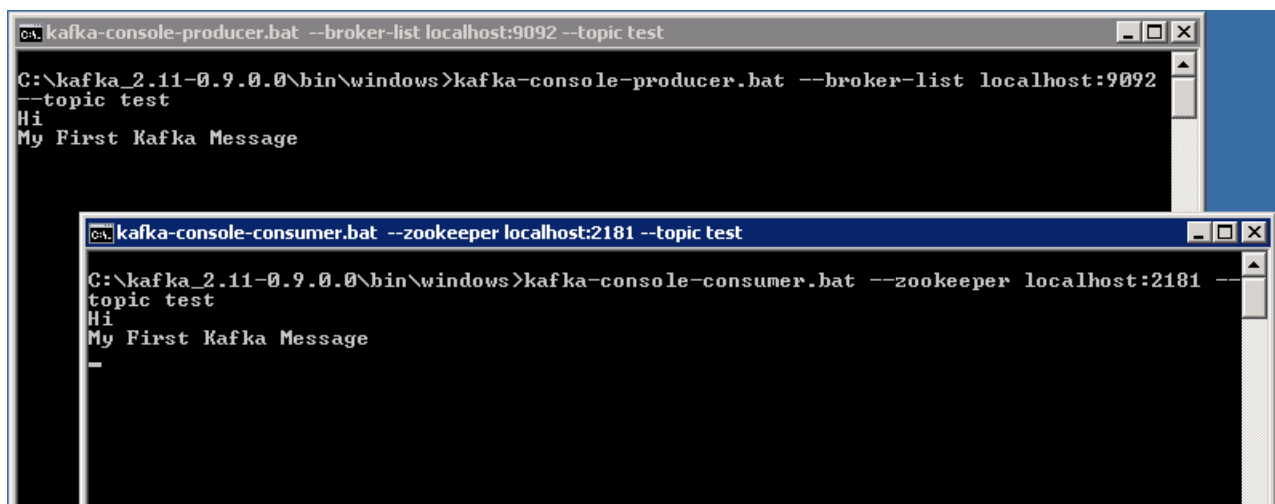
*Topics* yang dibuat dengan nama `Test` dan akan direplika sebanyak satu kali jika *standalone*. Jika memiliki klaster lebih dari satu maka bisa mereplikasi lebih dari satu dan replikasi tersebut digunakan sebagai *backup* jika terjadi kegagalan. Topik dapat dibuat dengan perintah: `kafka-topics`.

`bat--create--zookeeperlocalhost:2181--replication-factor1--partitions1--topic test`



Gambar 3.8: menjalankan topic

Untuk mengetes apakah suatu server sudah jalan perlu dilakukan pengecekan dengan membuat broker *producer* dan *consumer*. Untuk menjalankan *producer*, lakukan perintah `kafka-console-producer.bat--broker-listlocalhost:9092--topic test` dan untuk menjalankan *consumer* lakukan perintah `kafka-console-consumer.bat--zookeeperlocalhost:2181--topic test`. Jika berhasil *consumer* dan *producer* akan saling terhubung.

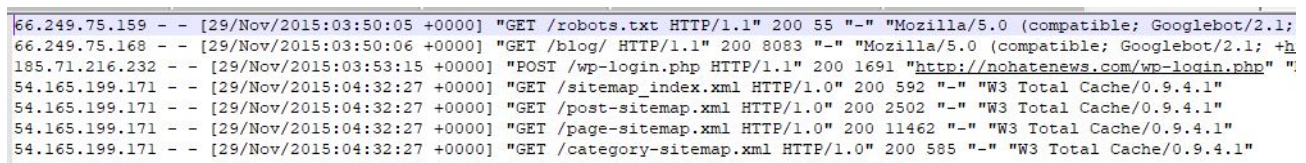


Gambar 3.9: *consumer-producer*

### 3.3 Studi Eksplorasi

#### 3.3.1 Eksplorasi Spark Streaming dengan TCP Socket

Studi Eksplorasi dilakukan dengan mencoba mengumpulkan data dari salah satu penyedia data yaitu TCP Socket. karena ini berupa simulasi, maka file input data web logs akan diunduh terlebih dahulu disimpan pada sebuah file `accesslog.txt`. Lalu menggunakan perintah `ncat -lk 9999 < accesslog.txt` yang artinya `ncat` akan memasukan data ke TCP port 9999 satu demi satu seolah-olah data yang masuk berupa aliran data.



Gambar 3.10: File input

Data yang akan dianalisis adalah simulasi web logs data stream yang datang dari aktivitas suatu website. Eksplorasi ini akan menghitung seberapa sering suatu file dibuka oleh pengguna. Berikut

cara membuat spark streaming. Contoh Gambar

```
//1. membuat streaming context dengan ukuran batch 1
val ssc = new StreamingContext("local[*]", "LogParser", Seconds(1))

setupLogging()

// 2.mengambil pattern dari satu weblog IP, Client, urls
val pattern = apacheLogPattern()

// 3.membuat socket stream untuk membaca dari netcat secara local
val lines = ssc.socketTextStream("127.0.0.1", 9999, StorageLevel.MEMORY_AND_DISK_SER)
```

Gambar 3.11: pengaturan spark streaming

```
//4.cek apakah suatu line sama dengan pola
val requests = lines.map(x => {val matcher:Matcher = pattern.matcher(x); if (matcher.matches()) matcher.group(5)})

// 5.mengambil url dari suatu pola
val urls = requests.map(x => {val arr = x.toString().split("_"); if (arr.size == 3) arr(1) else "[error]"})

// 6.Reduce Url dengan sliding windows 5 menit
val urlCounts = urls.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, _ - _, Seconds(300), Seconds(1))

// Urutkan dari yang terbesar
val sortedResults = urlCounts.transform(rdd => rdd.sortBy(x => x._2, false))
sortedResults.print()
```

Gambar 3.12: perhitungan url

Spark streaming dibuat dengan menentukan ukuran batch interval (Streaming Context) selama 1 detik. Pada interval 1 detik Spark Streaming akan mengambil data yang dihasilkan pada interval tersebut. lalu, menggunakan metode dari library ambil web log yang hanya mengikuti pola saja. Jadi, jika ada data lain yang masuk tapi tidak berbentuk web logs akan diabaikan. Menghubungkan batch interval dengan socket stream untuk mendapatkan data. Mengambil url file dari potongan web logs tersebut contoh `apachepb.gif`.Menghitung dengan `ReduceByKeyAndWindow` artinya hitung berapa banyak jumlah url pada key dan windows yang sama. Terakhir urutkan url dari yang memiliki hit paling banyak dan tampilkan 10 hasil terbaik. Contoh Gambar

```
// Jalankan Streaming
ssc.checkpoint("C:/checkpoint/")
ssc.start()
ssc.awaitTermination()
```

Gambar 3.13: Menjalankan Spark Streaming

Terakhir jalankan spark streaming yang telah dibuat. `ssc.awaitTermination()` menyatakan bahwa proses pengambilan data tidak akan berhenti sampai ada perintah dari user. Contoh Gambar

```

-----
Time: 1574436890000 ms
-----
(/wp-login.php,1758)
(/xmlrpc.php,882)
(/,43)
(/blog/,24)
(/post-sitemap.xml,19)
(/sitemap_index.xml,19)
(/category-sitemap.xml,19)
(/page-sitemap.xml,19)
(/national-headlines/,13)
(/business/,12)
...

-----
Time: 1574436891000 ms
-----
(/wp-login.php,1759)
(/xmlrpc.php,1102)
(/,46)
(/blog/,25)
(/post-sitemap.xml,20)
(/sitemap_index.xml,20)
(/category-sitemap.xml,20)
(/page-sitemap.xml,20)
(/orlando-headlines/,13)
(/national-headlines/,13)

```

Gambar 3.14: Output Web log

hasil dari eksekusi program tersebut adalah batch dengan interval yang telah kita atur dan pada tiap batch interval tersebut berisi hasil komputasi 10 file yang paling sering diakses. Dari hasil dapat disimpulkan pada batch pertama file yang paling sering diakses adalah `wp-login.php`

### 3.3.2 Eksplorasi dengan Twitter API

Studi eksplorasi dilakukan dengan mencoba mengumpulkan data dari twitter. Input berupa data stream unggahan tweet dari pengguna twitter di seluruh dunia. Data tidak akan dianalisis tapi hanya menyimpan status tweet di HDFS. Berikut Langkah-langkah penyimpanan data:

```

import scala.io.Source

for (line <- Source.fromFile("../twitter.txt").getLines) {
  val fields = line.split(" ")
  if (fields.length == 2) {
    System.setProperty("twitter4j.oauth." + fields(0), fields(1))
  }
}

```

Gambar 3.15: *Setup Twitter*

sebelum membuat batch interval harus membuat fungsi yang membaca kredensial twitter dari file `txt`.

```

setupTwitter()

//membuath batch interval ukuran 1 detik
val ssc = new StreamingContext("local[*]", "SaveTweets", Seconds(1))

setupLogging()

// Membuat aliran data
val tweets = TwitterUtils.createStream(ssc, None)

```

Gambar 3.16: *Setup Spark Streaming*

langkah pertama adalah mengatur kredensial dari twitter. Lalu membuat batch interval dengan durasi 1 detik. Tetapi, sekarang tidak dihubungkan dengan TCP socket melainkan langsung dari twitter.









## BAB 4

### ANALISIS DAN PERANCANGAN

Pada bab ini akan dibahas mengenai dan perancangan program. Bagian dari analisis terdiri dari analisis set data, analisis masukan dan keluaran, serta analisis output.

#### 4.1 Analisis Perangkat Lunak

Perangkat lunak yang akan dikembangkan adalah perangkat lunak spark. Perangkat lunak digunakan untuk mengambil data dengan sistem *spark streaming* dan melakukan analisis sederhana secara *real-time*. Program saat ini akan menghitung jumlah hashtag terbanyak pada lima menit terakhir.

##### 4.1.1 Analisis Set Data

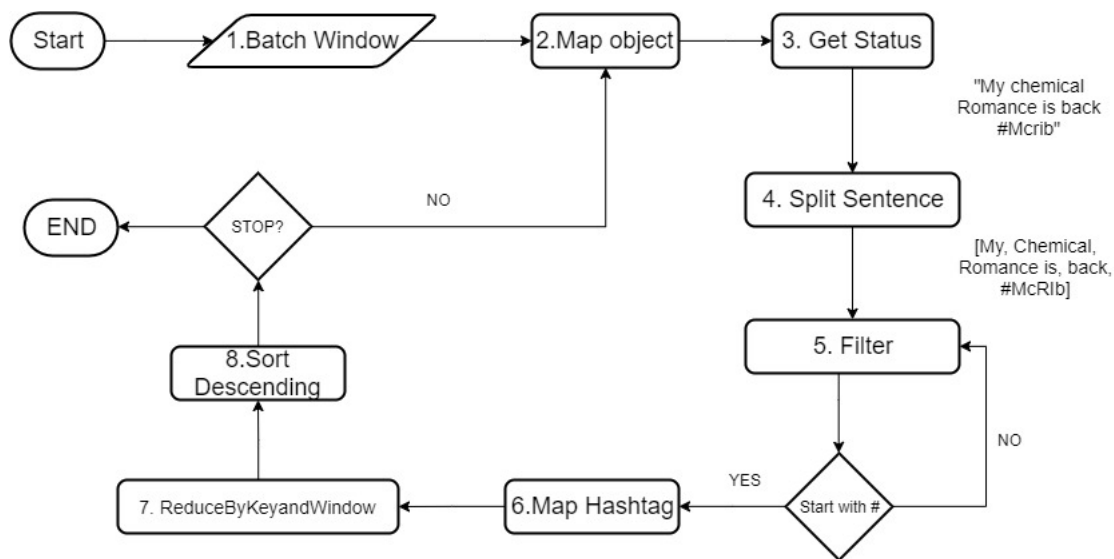
set data yang akan diambil oleh *spark streaming* bersifat *real-time*. Hal ini dilakukan dengan mengintegrasikan *spark streaming* dengan sistem eksternal yang menyediakan data melalui API. Untuk kasus ini akan digunakan data yang berasal dari twitter.

##### 4.1.2 Analisis Masukan dan Keluaran

Data yang didapatkan dari twitter adalah berupa *twitter object* dalam format JSON. Data twitter akan diambil selama 5 menit dan keluaran akan berupa pasangan key value yang bernilai (hashtag,value).

#### 4.2 Perancangan Penghitung Hashtag

Untuk membuat sistem penghitung hashtag bekerja, harus membuat batch window yang nanti akan diintegrasikan dengan Twitter. Lalu Setiap Objek akan dipetakan dan hanya diambil statusnya saja karena informasi hashtag terletak pada status seseorang. Status yang telah didapat akan dipetakan lagi dan dipisahkan perkata. Lalu fungsi filter akan dipanggil dan kita hanya mengambil simbol # yang merepresentasikan *Hashtag*. Hashtag akan dipetakan lagi dengan map hingga berisi pasangan key value. dengan nilai key adalah hashtag dan value adalah jumlahnya. Lalu akan dilakukan *reduce* berdasarkan nilai key dan batch window yang sama. Atur ukuran window menjadi 5 menit.



Gambar 4.1: Flowchart diagram analisis hashtag

1. Membuat batch window berukuran satu detik. setiap satu detik akan menangkap objek-objek dari twitter.
2. Memetakan setiap objek menjadi statusnya saja artinya dari seluruh objek tweet seperti nama atau location yang kita ambil hanya perkataan yang diposting oleh user saja.
3. Mengambil statusnya satu per satu
4. Membagi kalimat berdasarkan spasi lalu menyimpannya di array
5. Filter setiap elemen pada array jika elemen pada array tidak sama dengan # maka akan terus melakukan filter sampai status pada batch ini habis
6. memetakan hashtag menjadi  $[key, value] \Rightarrow [hashtag, 1]$
7. melakukan reduce penjumlahan berdasarkan key dan windows yang sama.
8. Mengurutkan hashtag berdasarkan value dari nilai yang paling tinggi.
9. Jika program sudah diberhentikan maka tidak akan mengumpulkan objek lagi. Tetapi, jika tidak akan terus berlanjut.

# LAMPIRAN A

## KODE PROGRAM

Listing A.1: URLCounter.scala

```
1 | package com.tcp.log
2 |
3 |
4 | import org.apache.spark.SparkConf
5 | import org.apache.spark.streaming.{Seconds, StreamingContext}
6 | import org.apache.spark.storage.StorageLevel
7 |
8 | import java.util.regex.Pattern
9 | import java.util.regex.Matcher
10 |
11 | import Utilities._
12 |
13 | object LogParser {
14 |
15 |   def main(args: Array[String]) {
16 |
17 |     //1. membuat streaming context dengan ukuran batch 1
18 |     val ssc = new StreamingContext("local[*]", "LogParser", Seconds(1))
19 |
20 |     setupLogging()
21 |
22 |     // 2.mengambil pattern dari satu weblog IP, Client, urls
23 |     val pattern = apacheLogPattern()
24 |
25 |     // 3.membuat socket stream untuk membaca dari netcat secara local
26 |     val lines = ssc.socketTextStream("127.0.0.1", 9999, StorageLevel.MEMORY_AND_DISK_SER)
27 |
28 |     //4.cek apakah suatu line sama dengan pola
29 |     val requests = lines.map(x => {val matcher:Matcher = pattern.matcher(x); if (matcher.matches()) matcher.group(5)})
30 |
31 |     // 5.mengambil url dari suatu pola
32 |     val urls = requests.map(x => {val arr = x.toString().split("_"); if (arr.size == 3) arr(1) else "[error]"})
33 |
34 |     // 6.Reduce Url dengan sliding windows 5 menit
35 |     val urlCounts = urls.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, _ - _, Seconds(300), Seconds(1))
36 |
37 |     // Urutkan dari yang terbesar
38 |     val sortedResults = urlCounts.transform(rdd => rdd.sortBy(x => x._2, false))
39 |     sortedResults.print()
40 |
41 |     // Jalankan Streaming
42 |     ssc.checkpoint("C:/checkpoint/")
43 |     ssc.start()
44 |     ssc.awaitTermination()
45 |   }
46 | }
```

Listing A.2: SaveTweets.scala

```
1 | package com.sparkstreaming.twitter
2 |
3 | import org.apache.spark._
4 | import org.apache.spark.SparkContext._
5 | import org.apache.spark.streaming._
6 | import org.apache.spark.streaming.twitter._
7 | import org.apache.spark.streaming.StreamingContext._
8 | import Utilities._
9 |
10 |
11 | object SaveTweets {
12 |
13 |
14 |   def main(args: Array[String]) {
15 |
16 |     setupTwitter()
17 |
18 |     //membuath batch interval ukuran 1 detik
19 |     val ssc = new StreamingContext("local[*]", "SaveTweets", Seconds(1))
20 |
21 |     setupLogging()
22 |
23 |     // Membuat aliran data
24 |     val tweets = TwitterUtils.createStream(ssc, None)
25 | }
```

```

26 // Ambil text status
27
28 val statuses = tweets.map(status => status.getText())
29 val names=tweets.map(user=>user.getUser().getName())
30
31 statuses.saveAsTextFiles("hdfs://localhost:50071/Twitter/Status/Output", "txt")
32
33 ssc.checkpoint("C:/checkpoint/")
34 ssc.start()
35 ssc.awaitTermination()
36 }
37 }

```

Listing A.3: HashtagsCounter.scala

```

1 package com.sparkstreaming.twitter
2
3 import org.apache.spark._
4 import org.apache.spark.SparkContext._
5 import org.apache.spark.streaming._
6 import org.apache.spark.streaming.twitter._
7 import org.apache.spark.streaming.StreamingContext._
8 import Utilities._
9
10
11 object PopularHashtags {
12
13   def main(args: Array[String]) {
14
15     // Setting twitter Credentials
16     setupTwitter()
17
18     //setup streaming context ukuran 1 detik
19     val ssc = new StreamingContext("local[*]", "PopularHashtags", Seconds(1))
20
21     // hapus spam selain error
22     setupLogging()
23
24     // Membuat Dstream dengan Streaming context
25     val tweets = TwitterUtils.createStream(ssc, None)
26
27     // ambil status
28     val statuses = tweets.map(status => status.getText())
29
30     // ambil setiap kata
31     val tweetwords = statuses.flatMap(tweetText => tweetText.split("_"))
32
33     // filter yang bukan hashtag
34     val hashtags = tweetwords.filter(word => word.startsWith("#"))
35
36     // Map setiap hastag menjadi key value (hashtag,1)
37     val hashtagKeyValues = hashtags.map(hashtag => (hashtag, 1))
38
39     //Hitung hashtag perdetik dengan sliding window selama 5 menit
40     val hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow( (x,y) => x + y, (x,y) => x - y, Seconds(300), Seconds(5))
41
42     // Sort berdasarkan banyak hashtag
43     val sortedResults = hashtagCounts.transform(rdd => rdd.sortBy(x => x._2, false))
44
45     // Print the top 10
46     sortedResults.saveAsTextFiles("hdfs://localhost:50071/Twitter/Hashtag/Output1", "txt")
47     sortedResults.print
48
49     ssc.checkpoint("C:/checkpoint/")
50     ssc.start()
51     ssc.awaitTermination()
52   }
53 }

```

Listing A.4: ErrorCounter.scala

```

1 package com.tcp.log
2
3
4 import org.apache.spark.SparkConf
5 import org.apache.spark.streaming.{Seconds, StreamingContext}
6 import org.apache.spark.storage.StorageLevel
7
8 import java.util.regex.Pattern
9 import java.util.regex.Matcher
10
11 import Utilities._
12
13 import java.util.concurrent._
14 import java.util.concurrent.atomic._
15
16
17 object LogAlarmer {
18
19   def main(args: Array[String]) {
20
21     // Membuat streaming context dengan ukuran 1 menit
22     val ssc = new StreamingContext("local[*]", "LogAlarmer", Seconds(1))
23
24     setupLogging()
25
26     // membuat pattern dari suatu log

```

```

27 | val pattern = apacheLogPattern()
28 |
29 | // Membuat socket stream yang akan menangkap data dari port 9999
30 | val lines = ssc.socketTextStream("127.0.0.1", 9999, StorageLevel.MEMORY_AND_DISK_SER)
31 |
32 | // mengambil status dari suatu lines
33 | val statuses = lines.map(x => {
34 |     val matcher:Matcher = pattern.matcher(x);
35 |     if (matcher.matches()) matcher.group(6) else "[error]"
36 | })
37 |
38 |
39 | // Map tingkat kegagalan
40 | val successFailure = statuses.map(x => {
41 |     val statusCode = util.Try(x.toInt) getOrElse 0
42 |     if (statusCode >= 200 && statusCode < 300) {
43 |         "Success"
44 |     } else if (statusCode >= 500 && statusCode < 600) {
45 |         "Failure"
46 |     } else {
47 |         "Other"
48 |     }
49 | })
50 |
51 | // hitung kegagalan pada sliding windows 5 menit
52 | val statusCounts = successFailure.countByValueAndWindow(Seconds(300), Seconds(1))
53 |
54 | // Untuk setiap batch ambil RDD di window saat ini
55 | statusCounts.foreachRDD((rdd, time) => {
56 |
57 |     var totalSuccess:Long = 0
58 |     var totalError:Long = 0
59 |
60 |     if (rdd.count() > 0) {
61 |         val elements = rdd.collect()
62 |         for (element <- elements) {
63 |             val result = element._1
64 |             val count = element._2
65 |             if (result == "Success") {
66 |                 totalSuccess += count
67 |             }
68 |             if (result == "Failure") {
69 |                 totalError += count
70 |             }
71 |         }
72 |     }
73 |
74 |     //print tingkat kesuksesan dan kegagalan pada suatu windows
75 |     println("Total_success:_" + totalSuccess + "_Total_failure:_" + totalError)
76 |
77 |     // Hanya memberi peringatan ketika rasio error dan sukses >0.5
78 |     if (totalError + totalSuccess > 100) {
79 |
80 |         val ratio:Double = util.Try( totalError.toDouble / totalSuccess.toDouble ) getOrElse 1.0
81 |         if (ratio > 0.5) {
82 |             // pada kasus nyata bisa menggunakan JavaMail atau scala's courier library
83 |             // untuk mengirim email kepada orang yang bertanggung jawab
84 |         } else {
85 |             println("All_systems_go.")
86 |         }
87 |     }
88 | })
89 |
90 | // Kick it off
91 | ssc.checkpoint("C:/checkpoint/")
92 | ssc.start()
93 | ssc.awaitTermination()
94 | }
95 | }

```

Listing A.5: Utilities.scala

```

1 | package com.sparkstreaming.twitter
2 |
3 | import org.apache.log4j.Level
4 | import java.util.regex.Pattern
5 | import java.util.regex.Matcher
6 |
7 | object Utilities {
8 |     /** Makes sure only ERROR messages get logged to avoid log spam. */
9 |     def setupLogging() = {
10 |         import org.apache.log4j.{Level, Logger}
11 |         val rootLogger = Logger.getRootLogger()
12 |         rootLogger.setLevel(Level.ERROR)
13 |     }
14 |
15 |     /** Configures Twitter service credentials using twitter.txt in the main workspace directory */
16 |     def setupTwitter() = {
17 |         import scala.io.Source
18 |
19 |         for (line <- Source.fromFile("../twitter.txt").getLines) {
20 |             val fields = line.split(",")
21 |             if (fields.length == 2) {
22 |                 System.setProperty("twitter4j.oauth." + fields(0), fields(1))
23 |             }
24 |         }
25 |     }
26 | }

```

```
27  /** Retrieves a regex Pattern for parsing Apache access logs. */
28  def apacheLogPattern():Pattern = {
29      val ddd = "\\d{1,3}"
30      val ip = s"($ddd\\. $ddd\\. $ddd\\. $ddd)?"
31      val client = "(\\S+)"
32      val user = "(\\S+)"
33      val dateTime = "(\\[[.+?\\]])"
34      val request = "\"(.*)\""
35      val status = "(\\d{3})"
36      val bytes = "(\\S+)"
37      val referer = "\"(.*)\""
38      val agent = "\"(.*)\""
39      val regex = s"$ip_$client_$user_$dateTime_$request_$status_$bytes_$referer_$agent"
40      Pattern.compile(regex)
41  }
42 }
```

## LAMPIRAN B

### HASIL EKSPERIMEN

Hasil eksperimen berikut dibuat dengan menggunakan TIKZPICTURE (bukan hasil excel yg diubah ke file bitmap). Sangat berguna jika ingin menampilkan tabel (yang kuantitasnya sangat banyak) yang datanya dihasilkan dari program komputer.



Gambar B.1: Hasil 1



Gambar B.2: Hasil 2



Gambar B.3: Hasil 3



Gambar B.4: Hasil 4