LAPORAN PRAKTIKUM

DATA ENGINEERING & BIG DATA SYSTEM

”Databricks Certified Associate Developer for Apache Spark Using Python”

OLEH

NAMA : MUHAMMAD ZAKY FARHAN

NIM : 105841110523

KELAS : 5AI-A INFORMATIKA

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS MUHAMMADIYAH MAKASSAR

2026

Chapter 1

Overview of the Certification Guide and Exam

Tinjauan Sertifikasi dan Standarisasi Materi (Certification Overview and Material Standardization)

Laporan implementasi ini disusun dengan mengacu pada kerangka kerja dan standar kurikulum resmi untuk sertifikasi *Databricks Certified Associate Developer for Apache Spark*. Seluruh implementasi teknis yang akan dibahas pada bagian selanjutnya didasarkan pada kompetensi inti yang dituntut dari seorang pengembang Spark profesional. Kurikulum ini dirancang untuk melatih dalam menggunakan *Spark DataFrame API* untuk melakukan tugas-tugas manipulasi data, serta pemahaman mendalam mengenai arsitektur dasar platform.

Selain aspek teknis pengkodean, standar ini juga menekankan pentingnya pemahaman konseptual. Seorang pengembang tidak hanya diharapkan mampu menulis sintaks yang benar, tetapi juga harus memahami implikasi performa dari setiap baris kode yang ditulis. Oleh karena itu, laporan ini tidak hanya sekadar menyajikan potongan kode, melainkan juga menyelaraskannya dengan *best practice* industri yang digariskan dalam bab awal buku panduan ini. Hal ini mencakup pemilihan metode yang tepat untuk efisiensi memori dan kecepatan eksekusi, memastikan bahwa solusi yang dibangun tidak hanya fungsional tetapi juga skalabel untuk menangani volume data besar (*big data*).

Chapter 2

Understanding Apache Spark and Its Applications

Pemahaman Apache Spark dan Ekosistem Terpadu (Understanding Apache Spark and Its Unified Ecosystem)

Apache Spark didefinisikan sebagai *unified analytics engine* atau mesin analitik terpadu yang dirancang khusus untuk pemrosesan data berskala besar. Bab ini menguraikan evolusi Spark dari pendahulunya, Hadoop MapReduce. Perbedaan fundamental yang membuat Spark menjadi standar industri saat ini adalah arsitektur pemrosesan *in-memory*. Jika sistem terdahulu sangat bergantung pada operasi baca-tulis (*I/O*) ke disk di setiap langkah pemrosesan, Spark mempertahankan data di dalam memori (RAM) selama mungkin. Pendekatan ini secara drastis mengurangi latensi dan meningkatkan kecepatan pemrosesan, terutama untuk algoritma iteratif yang sering digunakan dalam *machine learning* dan analisis data eksploratif. Dalam konteks laporan ini, bahasa pemrograman Python (PySpark) dipilih sebagai antarmuka utama karena fleksibilitasnya dan dukungan komunitas yang luas, menjembatani kemudahan penulisan kode dengan performa tinggi mesin Spark yang berbasis JVM (*Java Virtual Machine*).

Ekosistem Spark terdiri dari beberapa komponen yang saling terintegrasi erat, yang memungkinkan pengguna untuk menggabungkan berbagai jenis beban kerja dalam satu aplikasi. Fondasi utamanya adalah **Spark Core**, yang menyediakan fungsi dasar seperti manajemen memori, penjadwalan tugas, dan interaksi dengan sistem penyimpanan. Di atas lapisan dasar ini, terdapat modul-modul spesifik:

1. **Spark SQL**: Modul yang memungkinkan pemrosesan data terstruktur menggunakan sintaks SQL dan *DataFrame API*. Ini adalah komponen yang paling relevan dengan implementasi laporan ini.
2. **Spark Streaming**: Memungkinkan pemrosesan data secara *real-time* dengan skalabilitas tinggi dan toleransi kesalahan.
3. **MLlib**: Pustaka pembelajaran mesin yang menyediakan berbagai algoritma siap pakai yang dapat dijalankan secara terdistribusi.
4. **GraphX**: API untuk pemrosesan graf dan komputasi paralel graf.

Integrasi komponen-komponen ini memungkinkan peran data yang berbeda—seperti *Data Engineer* yang fokus pada *pipeline* data dan *Data Scientist* yang fokus pada pemodelan—untuk bekerja pada platform yang sama. Dalam laporan ini, fokus utama diberikan pada penggunaan Spark SQL dan DataFrame, karena inilah alat utama dalam proses *Extract, Transform, Load* (ETL) modern.

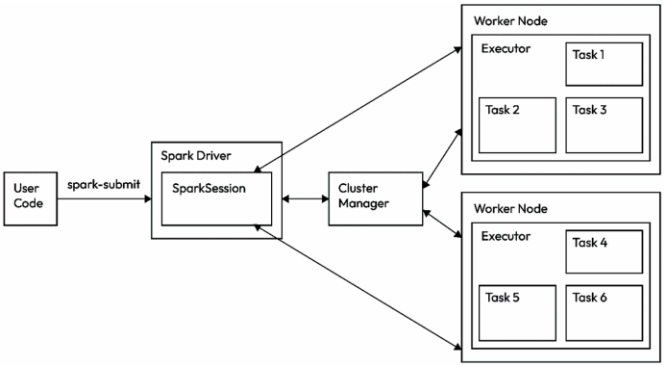
Chapter 3

Spark Architecture and Transformations

Arsitektur Klaster dan Komponen Inti (Cluster Architecture and Core Components)

Untuk memahami bagaimana kode yang diimplementasikan dalam laporan ini dieksekusi, kita perlu membedah arsitektur klaster Spark secara mendalam. Spark beroperasi menggunakan pola *Master-Slave*. Komponen pusat dari arsitektur ini adalah **Driver Program**. *Driver* bertindak sebagai "otak" dari aplikasi; di sinilah objek SparkSession dibuat. Tugas utama *Driver* adalah mengubah kode pengguna menjadi graf tugas (*tasks*) yang dapat dieksekusi, menjadwalkan tugas-tugas tersebut, dan mengoordinasikan seluruh proses dengan *Cluster Manager*. Tanpa *Driver*, tidak ada instruksi yang dapat dikirimkan ke klaster.

Di sisi lain, komponen pekerja dalam arsitektur ini disebut **Executors**. *Executors* adalah proses yang berjalan pada node-node pekerja (*worker nodes*) dalam klaster. Mereka bertanggung jawab untuk menjalankan tugas komputasi individu yang dikirim oleh *Driver* dan menyimpan data hasil pemrosesan di dalam memori atau disk lokal mereka. Hubungan antara *Driver* dan *Executors* dijembatani oleh **Cluster Manager** (seperti YARN, Mesos, Kubernetes, atau Standalone), yang bertugas mengalokasikan sumber daya fisik (CPU dan RAM). Pemahaman tentang interaksi ketiga komponen ini sangat krusial, karena performa aplikasi sangat bergantung pada seberapa efisien *Driver* membagi tugas dan seberapa optimal *Executors* memanfaatkan sumber daya yang diberikan.



Gambar 3.1 *Spark Architecture*

Dalam eksekusi program Spark, terdapat hierarki yang ketat yang menentukan bagaimana data diproses. Unit tertinggi adalah **Application**, yang mencakup keseluruhan program pengguna. Ketika sebuah *Action* dipanggil dalam kode, Spark memicu pembuatan sebuah **Job**. Sebuah *Job* kemudian dipecah menjadi unit-unit yang lebih kecil yang disebut **Stages**. Pemisahan antar-*Stage* ditentukan oleh kebutuhan untuk melakukan pertukaran data antar-node, yang dikenal sebagai batas *shuffle*. Terakhir, setiap *Stage* terdiri dari banyak **Tasks**. *Task* adalah unit kerja terkecil yang dikirim ke satu *Executor* untuk memproses satu partisi data tertentu.

Konsep partisi sangat penting di sini. Spark membagi data besar menjadi potongan-potongan kecil (partisi) agar dapat diproses secara paralel. Setiap *Task* menangani satu partisi. Jika data tidak terpartisi dengan baik (misalnya, terlalu sedikit partisi untuk jumlah CPU yang tersedia, atau data *skewed* di mana satu partisi jauh lebih besar dari yang lain), maka efisiensi paralelisasi akan berkurang drastis. Oleh karena itu, pemahaman tentang hierarki ini membantu dalam mengidentifikasi *bottleneck* performa dalam implementasi kode.

Konsep RDD, Evaluasi Tertunda, dan DAG (RDD Concepts, Lazy Evaluation, and DAG)

Meskipun implementasi modern menggunakan *DataFrame*, di balik layar Spark tetap beroperasi menggunakan struktur data fundamental yang disebut **Resilient Distributed Dataset (RDD)**. RDD adalah koleksi elemen yang *immutable* (tidak dapat diubah setelah dibuat) dan terdistribusi di seluruh klaster. Sifat *immutable* ini mendukung toleransi kesalahan (*fault tolerance*); jika satu node gagal, Spark dapat membangun kembali data yang hilang dengan mengulang langkah transformasi dari data aslinya menggunakan informasi silsilah (*lineage*) yang tersimpan.

Fitur kunci lain dari arsitektur Spark adalah **Lazy Evaluation** atau evaluasi tertunda. Ketika kita menulis kode transformasi (seperti filter, select, atau map), Spark tidak langsung mengeksekusi perintah tersebut. Sebaliknya, ia mencatat serangkaian transformasi tersebut ke dalam sebuah rencana logis yang disebut **Directed Acyclic Graph (DAG)**. Eksekusi fisik baru benar-benar terjadi ketika sebuah *Action* (seperti count, collect, atau show) dipanggil. Mekanisme ini memberi kesempatan pada **Catalyst Optimizer** Spark untuk menganalisis keseluruhan rencana kueri dan melakukan optimasi cerdas, seperti menggabungkan filter atau mengatur ulang operasi sebelum data benar-benar diproses.

Tipe Transformasi Narrow &45 Wide (Transformation Types Narrow & Wide)

Dalam memanipulasi data, Spark membedakan operasi menjadi dua kategori utama yang memiliki dampak performa berbeda:

1. **Narrow Transformation (Transformasi Sempit)**: Terjadi ketika setiap partisi output hanya bergantung pada satu partisi input. Contohnya adalah filter, map, atau union. Operasi ini sangat cepat karena dapat dilakukan secara independen di memori masing-masing *Executor* tanpa perlu berkomunikasi dengan node lain (proses ini disebut *pipelining*).
2. **Wide Transformation (Transformasi Lebar)**: Terjadi ketika perhitungan hasil membutuhkan data dari banyak partisi input yang mungkin tersebar di node yang berbeda. Contohnya adalah groupBy, orderBy, atau join. Operasi ini memicu proses **Shuffle**, yaitu perpindahan fisik data melintasi jaringan klaster untuk mengelompokkan data yang relevan. *Shuffle* adalah operasi yang mahal dan sering menjadi penyebab utama lambatnya aplikasi Spark. Dalam bab implementasi nanti, strategi untuk meminimalkan transformasi lebar yang tidak perlu akan menjadi salah satu fokus optimasi.

Chapter 4

Spark DataFrames and their Operations

Dalam bab ini, pembahasan akan difokuskan pada pengenalan berbagai API yang tersedia di dalam ekosistem Spark serta fitur-fitur unggulannya. Fokus utama akan diarahkan pada operasi Spark DataFrame, mulai dari teknik melihat data (*viewing*) hingga manipulasi data yang meliputi penyaringan (*filtering*), penambahan kolom, penggantian nama, hingga penghapusan kolom. Di akhir pembahasan, diharapkan pembaca dapat memahami cara kerja PySpark DataFrame secara menyeluruh, termasuk teknik manipulasi dan agregasi data.

Memulai dengan PySpark (*Getting Started in PySpark*)

Spark dikenal karena fleksibilitasnya dalam mendukung berbagai bahasa pemrograman utama, yaitu Scala, Python, R, dan SQL. Keunggulan arsitektur Spark terletak pada mesin eksekusi (execution engine) yang seragam di belakang layar, terlepas dari bahasa mana yang digunakan oleh pengembang. Hal ini menciptakan unifikasi yang kuat, memungkinkan pengembang untuk memilih bahasa yang paling mereka kuasai tanpa mengorbankan performa sistem. Dalam konteks laporan ini, fokus bahasa yang digunakan adalah Python, di mana implementasi Spark dengan bahasa Python dikenal dengan istilah PySpark.

Menginstal Spark dan Pustaka Pendukung (*Installing Spark and Dependencies*)

Untuk memulai pengembangan dengan Spark, langkah pertama adalah mempersiapkan lingkungan kerja lokal. Kita tidak hanya bekerja dengan Spark secara isolasi. Kita memanfaatkan ekosistem Python yang lebih luas, termasuk Pandas untuk pembuatan data awal dan PyArrow untuk optimasi pertukaran data antara Spark dan Pandas. Selain itu, karena implementasi dilakukan di sistem operasi Windows, terdapat prasyarat sistem yang ketat terkait Java (JDK) dan Hadoop binary (*winutils*) agar Spark dapat berjalan tanpa kendala. Sebelum melakukan instalasi paket Python, berikut adalah spesifikasi lingkungan pengembangan yang digunakan dalam laporan ini:

Spesifikasi Lingkungan (*Environment Specifications*):

```
Python Version : 3.10.11
Spark Version : 3.5.3
Numpy Version : 1.26.4
PyArrow Version : 14.0.1
Java Home Path : C:\Program Files\Java\jdk-21
Hadoop Path : C:\hadoop
```

Untuk menyiapkan seluruh pustaka Python yang diperlukan sesuai versi di atas, kita menjalankan perintah instalasi paket berikut melalui terminal atau command prompt:

```
%pip install pyspark==3.5.3 pandas numpy==1.26.4 pyarrow==14.0.1
```

Perintah instalasi di atas mengunduh empat komponen utama yang memiliki peran spesifik namun saling berinteraksi erat dalam kode bab ini. Paket pyspark berfungsi sebagai antarmuka inti Python menuju mesin Spark, menyediakan pustaka utama untuk pembuatan sesi dan operasi DataFrame terdistribusi. Keberadaan pustaka pandas sangat diperlukan untuk mendemonstrasikan fitur interoperabilitas, baik saat digunakan sebagai sumber data input awal maupun sebagai format output akhir. Sementara itu, numpy hadir sebagai fondasi komputasi numerik yang menjadi dependensi wajib bagi Pandas agar dapat beroperasi. Di antara komponen-komponen tersebut, pustaka pyarrow memegang peran krusial sebagai akselerator; ia memungkinkan transfer data yang efisien antara memori Java (JVM Spark) dan memori Python, yang secara signifikan mempercepat proses konversi data.

Pemilihan versi spesifik untuk setiap pustaka di atas dilakukan berdasarkan pertimbangan kompatibilitas yang sangat ketat (*dependency chain*), khususnya untuk lingkungan Windows. Kita menggunakan Java (JDK) versi 21 yang modern, namun versi ini memberlakukan kebijakan keamanan memori yang jauh lebih ketat dibanding pendahulunya. Kondisi ini memaksa kita untuk menahan versi pyarrow di 14.0.1, karena versi yang lebih baru diketahui sering mengalami konflik (*crash*) saat berinteraksi dengan alokasi memori Java 21 di dalam Spark. Efek domino ini berlanjut ke numpy; kita wajib menurunkan versinya ke 1.26.4 karena pyarrow 14.0.1 belum mendukung arsitektur numpy versi 2.0 ke atas. Seluruh rantai ketergantungan ini diikat oleh Python 3.10 dan Spark 3.5.3 yang bertindak sebagai fondasi stabil, menjembatani komponen-komponen beda generasi tersebut agar dapat bekerja secara harmonis tanpa memicu kesalahan sistem.

Selain paket Python, konfigurasi variabel lingkungan untuk Java dan Hadoop juga memegang peranan vital. Karena Spark berjalan di atas *Java Virtual Machine* (JVM), lokasi instalasi JDK 21 harus didaftarkan secara eksplisit ke dalam sistem. Khusus untuk sistem operasi Windows, variabel Hadoop Path yang mengarah ke folder bin (tempat winutils.exe berada) adalah persyaratan mutlak. Komponen ini diperlukan untuk memanipulasi izin sistem file (*file system permissions*) yang tidak dimiliki Windows secara natif. Tanpa kombinasi yang presisi antara paket Python yang telah dikalibrasi versinya dan konfigurasi sistem ini, kode implementasi tidak akan dapat dieksekusi dengan benar.

Membuat sesi Spark (*Creating a Spark session*)

Setelah instalasi berhasil, komponen terpenting dalam setiap aplikasi Spark adalah Spark Session. Sesi ini berfungsi sebagai titik masuk (entry point) atau gerbang utama bagi aplikasi untuk berkomunikasi dengan klaster Spark. Tanpa sesi yang aktif, kode tidak dapat mendistribusikan tugas atau mengakses memori klaster.

Implementasi Source Code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

Kode inisialisasi ini memiliki peran sentral dalam siklus hidup aplikasi Spark. Pertama, baris from pyspark.sql import SparkSession mengimpor kelas SparkSession dari modul SQL PySpark. Ini adalah perpustakaan inti yang menyediakan fungsionalitas untuk bekerja dengan data terstruktur. Kedua, baris spark = SparkSession.builder.getOrCreate() adalah tempat inisialisasi sebenarnya terjadi. Kode ini menggunakan pola desain *builder* (pembangun). Metode .builder menyiapkan konfigurasi sesi. Bagian paling cerdas dari kode ini adalah fungsi .getOrCreate(). Fungsi ini bekerja dengan logika ganda: jika

sebuah sesi Spark sudah ada dan aktif, ia akan mengembalikannya; namun jika belum ada, ia akan membuat sesi baru. Mekanisme ini sangat penting karena dalam satu aplikasi Spark (pada konteks JVM tunggal), **hanya diperbolehkan ada satu sesi Spark yang aktif** dalam satu waktu. Variabel spark yang dihasilkan kemudian menjadi objek utama yang akan digunakan terus-menerus untuk membaca data, membuat DataFrame, dan mengeksekusi perintah SQL. Perlu dicatat jika kode dijalankan di dalam *Spark shell*, variabel spark ini biasanya sudah dibuat secara otomatis oleh sistem.

Jika kode dijalankan di lingkungan standar atau klaster produksi berbasis Linux, inisialisasi sesi biasanya cukup dilakukan dengan perintah seperti diatas. Namun, karena kita akan menjalankan implementasi ini di lingkungan lokal berbasis Windows, kita perlu melakukan beberapa penyesuaian konfigurasi agar sistem berjalan stabil. Penyesuaian ini meliputi pengaturan variabel lingkungan untuk menjembatani sistem file Windows dengan ekosistem Hadoop, serta pengaturan parameter Java Virtual Machine (JVM) untuk menangani kebijakan akses memori pada versi Java modern.

Implementasi Source Code:

```
import os
import sys
from pyspark.sql import SparkSession
os.environ['HADOOP_HOME'] = "C:\\hadoop"
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
jvm_options = (
    "--add-opens=java.base/java.lang=ALL-UNNAMED "
    "--add-opens=java.base/java.lang.invoke=ALL-UNNAMED "
    "--add-opens=java.base/java.lang.reflect=ALL-UNNAMED "
    "--add-opens=java.base/java.io=ALL-UNNAMED "
    "--add-opens=java.base/java.net=ALL-UNNAMED "
    "--add-opens=java.base/java.nio=ALL-UNNAMED "
    "--add-opens=java.base/java.util=ALL-UNNAMED "
    "--add-opens=java.base/java.util.concurrent=ALL-UNNAMED "
    "--add-opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED "
    "--add-opens=java.base/sun.nio.ch=ALL-UNNAMED "
    "--add-opens=java.base/sun.nio.cs=ALL-UNNAMED "
    "--add-opens=java.base/sun.security.action=ALL-UNNAMED "
    "--add-opens=java.base/sun.util.calendar=ALL-UNNAMED "
    "--add-opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED"
)
spark = SparkSession.builder \
    .appName("ProyekSparkWindows") \
    .master("local[*]") \
    .config("spark.driver.bindAddress", "127.0.0.1") \
    .config("spark.driver.extraJavaOptions", jvm_options) \
    .config("spark.executor.extraJavaOptions", jvm_options) \
    .getOrCreate()
```

Pada blok kode di atas, kita melakukan serangkaian konfigurasi pra-inisialisasi. Pertama, modul os digunakan untuk menetapkan HADOOP_HOME ke direktori lokal (misalnya C:\hadoop) yang berisi binary winutils.exe, sebuah komponen vital agar Spark dapat berinteraksi dengan sistem file Windows. Kita juga secara eksplisit mengarahkan PYSPARK_PYTHON ke sys.executable untuk memastikan driver dan worker menggunakan interpreter Python yang konsisten. Selanjutnya, variabel jvm_options didefinisikan berisi deretan flag --add-opens. Flag ini sangat krusial saat menggunakan Java versi baru (Java 9 ke atas) untuk mengizinkan Spark mengakses modul internal Java melalui mekanisme reflection tanpa terhalang batasan keamanan modern.

Proses pembuatan sesi itu sendiri dilakukan menggunakan pola *builder*. Kita memberi nama aplikasi "ProyekSparkWindows" dan mengatur mode master("local[*]"), yang berarti Spark akan berjalan secara lokal memanfaatkan seluruh inti CPU yang tersedia untuk performa maksimal. Konfigurasi tambahan .config digunakan untuk menyuntikkan opsi JVM yang telah kita buat sebelumnya ke dalam *driver* dan *executor*, serta mengikat alamat *driver* ke 127.0.0.1 (localhost) untuk memastikan koneksi jaringan lokal berjalan lancar. Fungsi .getOrCreate() kemudian dipanggil untuk memfinalisasi dan mengaktifkan objek spark tersebut.

Objek spark berhasil dibuat di memori dengan konfigurasi khusus tersebut, siap digunakan sebagai gerbang utama untuk seluruh operasi DataFrame selanjutnya. Tidak ada output visual yang dicetak pada tahap ini.

API Dataset (*Dataset API*)

Spark menyediakan antarmuka Dataset yang diperkenalkan pada versi 1.6. Dataset adalah koleksi data terdistribusi yang menawarkan fitur fixed typing (pengetikan tipe data yang ketat). Namun, API ini hanya tersedia untuk bahasa pemrograman Java dan Scala, serta tidak tersedia untuk Python atau R. API Dataset dibangun di atas fondasi Resilient Distributed Datasets (RDDs), sehingga mewarisi fitur-fitur RDD namun dengan tambahan optimasi dari mesin Spark SQL untuk performa kueri yang lebih cepat.

API DataFrame (*DataFrame API*)

Mengingat popularitas Python di komunitas data science, Spark menyediakan API DataFrame sebagai ekuivalen dari Dataset. Konsep ini terinspirasi langsung dari Pandas DataFrame di Python. Secara struktural, DataFrame adalah sekumpulan data yang diorganisir ke dalam baris dan kolom yang memiliki nama, persis seperti tabel dalam basis data relasional. Format tabular ini dipilih karena telah menjadi standar lama dalam komputasi data, memudahkan pengembang yang terbiasa dengan SQL atau Excel untuk beralih ke pemrosesan data besar (big data).

Membuat operasi DataFrame (*Creating DataFrame operations*)

Meskipun DataFrame menawarkan kemudahan penggunaan seperti tabel SQL, struktur penyimpanan data yang mendasarinya tetaplah RDD. DataFrame hanyalah lapisan abstraksi yang menyembunyikan kompleksitas teknis RDD dari pengguna. Sifat utama yang diwarisi DataFrame adalah Evaluasi Malas (Lazy Evaluation) dan Immutability (sifat tidak dapat diubah).

Evaluasi malas berarti Spark tidak akan mengeksekusi perhitungan saat kode transformasi ditulis. Sebaliknya, Spark mencatat langkah-langkah tersebut sebagai sebuah rencana (*plan*). Eksekusi baru benar-benar dijalankan ketika sebuah *action* (aksi) dipanggil. Pendekatan ini memberikan keuntungan performa yang masif karena Spark dapat menyusun strategi optimasi (disebut *Catalyst Optimizer*) sebelum memproses data. Sifat *immutable* berarti setiap kali dilakukan perubahan pada DataFrame, data asli tidak diubah, melainkan menghasilkan DataFrame baru.

Di PySpark, fungsi utama untuk membuat DataFrame adalah spark.createDataFrame. Fungsi ini sangat fleksibel dan dapat menerima berbagai jenis input data mentah, mulai dari list, tuple, dictionary, Pandas DataFrame, hingga RDD.

Salah satu komponen krusial saat membuat DataFrame adalah **schema** (skema). Skema mendefinisikan struktur tabel, termasuk nama kolom dan tipe datanya. Pengguna memiliki dua opsi: mendefinisikan skema secara eksplisit (manual) atau membiarkan Spark menebaknya secara otomatis (*infer schema*) berdasarkan sampel data yang ada. Jika argumen skema tidak disertakan dalam kode, Spark akan mengambil alih tugas penentuan tipe data tersebut.

Menggunakan daftar baris (*Using a list of rows*)

Metode pertama yang diperkenalkan untuk membuat DataFrame adalah dengan menyusun data secara manual menggunakan objek baris atau rows. Secara konseptual, Anda dapat membayangkan "baris data" ini mirip dengan baris-baris dalam daftar belanjaan atau entri buku besar. Setiap baris memiliki struktur yang seragam, di mana nilai-nilai di dalamnya berbagi "judul" atau header kolom yang sama.

Implementasi Source Code:

```
import pandas as pd
from datetime import datetime, date
from pyspark.sql import Row

data_df = spark.createDataFrame([
    Row(col_1=100, col_2=200., col_3='string_test_1', col_4=date(2023, 1, 1),
     col_5=datetime(2023, 1, 1, 12, 0)),
    Row(col_1=200, col_2=300., col_3='string_test_2', col_4=date(2023, 2, 1),
     col_5=datetime(2023, 1, 2, 12, 0)),
    Row(col_1=400, col_2=500., col_3='string_test_3', col_4=date(2023, 3, 1),
     col_5=datetime(2023, 1, 3, 12, 0))
])
```

Pada blok kode di atas, proses dimulai dengan mengimpor pustaka Row dari pyspark.sql. Objek Row ini bertindak sebagai pembungkus (wrapper) untuk setiap entri data tunggal. Dalam kode tersebut, variabel data_df diinisialisasi menggunakan fungsi spark.createDataFrame. Argumen yang dimasukkan ke dalam fungsi ini adalah sebuah daftar (list) Python yang berisi sekumpulan objek Row.

Perhatikan bahwa setiap Row didefinisikan dengan pasangan kunci-nilai (*key-value*), seperti col_1=100 atau col_3='string_test_1'. Pendekatan ini memberikan label eksplisit pada setiap nilai data. Selain itu, penggunaan fungsi date dan datetime menunjukkan bahwa Spark mampu menangani tipe data waktu yang kompleks secara natif sejak awal pembuatan. Yang menarik di sini adalah tidak adanya definisi tipe data (seperti *integer* atau *string*) secara menyeluruh untuk tabel tersebut; kode ini menyerahkan sepenuhnya tugas identifikasi tipe data kepada kecerdasan internal Spark.

Output:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date, col_5: timestamp]
```

Hasil keluaran menampilkan struktur skema (schema signature) dari DataFrame yang terbentuk. Informasi di dalam kurung siku memperlihatkan bagaimana Spark menerjemahkan data mentah kita: col_1 dikenali sebagai bigint (bilangan bulat besar), col_2 sebagai double (bilangan desimal presisi ganda), dan col_3 sebagai string (teks). Keberhasilan Spark memetakan col_4 menjadi date dan col_5 menjadi timestamp membuktikan bahwa fitur inferensi skema (schema inference) berjalan dengan baik, mengenali objek waktu Python dan mengonversinya ke tipe data Spark yang sesuai tanpa campur tangan pengguna.

Menggunakan daftar baris dengan skema (*Using a list of rows with schema*)

Skema DataFrame adalah cetak biru yang mendefinisikan tipe data apa yang harus ada pada setiap kolom. Meskipun Spark bisa menebak skema (seperti pada metode sebelumnya), mendefinisikan skema secara eksplisit (explicit schema definition) adalah praktik yang lebih aman dan profesional. Hal ini berguna ketika kita ingin "memaksa" data mengikuti aturan tertentu, misalnya memastikan kode pos dibaca sebagai teks (string) agar nol di depan tidak hilang, bukan sebagai angka (integer).

Implementasi Source Code:

```
import pandas as pd
from datetime import datetime, date
from pyspark.sql import Row

data_df = spark.createDataFrame([
    Row(col_1=100, col_2=200., col_3='string_test_1', col_4=date(2023, 1, 1),
     col_5=datetime(2023, 1, 1, 12, 0)),
    Row(col_1=200, col_2=300., col_3='string_test_2', col_4=date(2023, 2, 1),
     col_5=datetime(2023, 1, 2, 12, 0)),
    Row(col_1=400, col_2=500., col_3='string_test_3', col_4=date(2023, 3, 1),
     col_5=datetime(2023, 1, 3, 12, 0))
], schema=' col_1 long, col_2 double, col_3 string, col_4 date, col_5
timestamp')
```

Kode ini hampir identik dengan contoh sebelumnya, namun memiliki satu perbedaan vital pada argumen fungsi createDataFrame. Di bagian akhir fungsi, ditambahkan parameter schema yang berisi string definisi: 'col_1 long, col_2 double...'. Ini adalah instruksi DDL (Data Definition Language) yang memberi perintah tegas kepada Spark.

Dengan menambahkan parameter ini, kita tidak lagi meminta Spark untuk "menebak", melainkan "mematuhi". Misalnya, kita menetapkan col_1 sebagai long. Jika data input memiliki angka yang bisa dianggap integer biasa, Spark akan tetap menyimpannya sebagai long sesuai instruksi. Ini menjamin konsistensi struktur data, terutama saat menangani dataset besar di mana anomali kecil pada sampel data bisa menyebabkan kesalahan inferensi tipe jika hanya mengandalkan penebakan otomatis.

Output:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date, col_5: timestamp]
```

Output ini mengonfirmasi bahwa skema yang kita paksakan telah berhasil diterapkan. Tipe data yang tertera pada output (bigint, double, string, dll.) persis sesuai dengan string skema yang kita tuliskan pada kode. Ini memverifikasi bahwa DataFrame data_df kini memiliki struktur yang kaku dan terprediksi, sesuai dengan desain yang diinginkan oleh pengembang data.

Menggunakan Pandas DataFrames (*Using Pandas DataFrames*)

Dalam alur kerja sains data, seringkali data awal diproses menggunakan pustaka Pandas karena kemudahannya untuk manipulasi data skala kecil. Spark menyadari hal ini dan menyediakan jembatan untuk mengubah Pandas DataFrame menjadi Spark DataFrame. Konsepnya adalah membuat DataFrame di Pandas terlebih dahulu, lalu mengonversinya ke lingkungan Spark yang terdistribusi.

Implementasi Source Code:

```
from datetime import datetime, date
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

rdd = spark.sparkContext.parallelize([
    (100, 200., 'string_test_1', date(2023, 1, 1), datetime(2023, 1, 1, 12, 0)),
    (200, 300., 'string_test_2', date(2023, 2, 1), datetime(2023, 1, 2, 12, 0)),
    (300, 400., 'string_test_3', date(2023, 3, 1), datetime(2023, 1, 3, 12, 0))
])
data_df = spark.createDataFrame(rdd, schema=['col_1', 'col_2', 'col_3', 'col_4',
'col_5'])
```

Pada kode ini, sparkContext.parallelize digunakan untuk memecah daftar data menjadi RDD yang tersebar di berbagai *node* klaster. Kemudian, fungsi spark.createDataFrame dipanggil dengan input RDD tersebut. Perbedaan penting di sini adalah pada parameter schema. Kode hanya memberikan daftar nama kolom ['col_1', 'col_2', ...] tanpa tipe data. Ini berarti Spark melakukan "penebakan hibrida": nama kolom diambil dari daftar tersebut, sementara tipe datanya disimpulkan (*inferred*) dengan memindai isi data RDD. Perlu diingat, konversi dari Pandas/Lokal ke Spark (toPandas atau createDataFrame dari lokal) melibatkan transfer data dari satu mesin (*driver*) ke banyak mesin (*workers*), yang bisa memakan memori jika datanya sangat besar.

Output:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date, col_5: timestamp]
```

Hasilnya konsisten dengan metode-metode sebelumnya. Meskipun input awalnya berbeda (dalam hal ini RDD/koleksi terdistribusi), hasil akhirnya tetaplah sebuah Spark DataFrame dengan tipe data yang terdefinisi dengan benar (bigint, double, dll.). Ini menunjukkan fleksibilitas fungsi createDataFrame dalam menelan berbagai jenis sumber data mentah dan mengubahnya menjadi format standar Spark.

Menggunakan tuple (*Using tuples*)

Alternatif lain untuk menyusun data adalah menggunakan tuple. Dalam bahasa Python, tuple mirip dengan list, namun bersifat immutable (tidak bisa diubah isinya setelah dibuat). Struktur ini sangat cocok untuk merepresentasikan satu baris data (record) yang statis. Kita bisa membuat setiap baris sebagai tuple, lalu menggabungkannya menjadi satu kesatuan DataFrame10.

Implementasi Source Code:

```
from datetime import datetime, date
from pyspark.sql import SparkSession
from pyspark.sql import Row

rdd = spark.sparkContext.parallelize([
    (100, 200., 'string_test_1', date(2023, 1, 1), datetime(2023, 1, 1, 12, 0)),
    (200, 300., 'string_test_2', date(2023, 2, 1), datetime(2023, 1, 2, 12, 0)),
    (300, 400., 'string_test_3', date(2023, 3, 1), datetime(2023, 1, 3, 12, 0))
])
data_df = spark.createDataFrame(rdd, schema=['col_1', 'col_2', 'col_3', 'col_4',
'col_5'])
```

Kode ini mendemonstrasikan pembuatan RDD dari sekumpulan tuple. (100, 200, ...) adalah representasi tuple di Python. Fungsi spark.sparkContext.parallelize menyebarkan tuple-tuple ini ke dalam klaster. Selanjutnya, spark.createDataFrame mengubah tumpukan tuple tersebut menjadi DataFrame yang memiliki kolom dan baris. Seperti pada contoh sebelumnya, skema yang diberikan hanya berupa daftar nama kolom, sehingga Spark secara otomatis memindai elemen di dalam tuple (misalnya mendeteksi angka 100 sebagai integer dan 'string_test_1' sebagai string) untuk menentukan tipe data setiap kolom.

Output:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date, col_5: timestamp]
```

Sekali lagi, output menegaskan konsistensi Spark. Tidak peduli apakah sumber datanya berupa daftar objek Row, RDD, atau kumpulan tuple, selama strukturnya konsisten, Spark akan menghasilkan objek DataFrame yang identik. Output ini memvalidasi bahwa data telah sukses masuk ke dalam lingkungan Spark dan siap untuk diproses lebih lanjut.

Bagaimana cara melihat DataFrames (*How to view the DataFrames*)

Setelah DataFrame berhasil dibuat, langkah logis berikutnya adalah memvalidasi isinya. Spark menyediakan berbagai pernyataan (statements) atau metode untuk menginspeksi data. Salah satu tantangan dalam big data adalah kita tidak bisa sekadar membuka data seperti membuka file Excel, karena datanya mungkin berjumlah miliaran baris dan tersebar di banyak komputer. Oleh karena itu, Spark menawarkan metode untuk "mengintip" sebagian data tersebut.

Melihat DataFrames (*Viewing DataFrames*)

Cara paling standar dan populer untuk melihat isi DataFrame adalah menggunakan fungsi .show(). Fungsi ini dirancang untuk inspeksi visual cepat. Ia akan mencetak data dalam format tabel ASCII yang rapi langsung ke konsol atau terminal pengguna.

Implementasi Source Code:

```
data_df.show()
```

Perintah data_df.show() adalah perintah Action. Ini berarti perintah ini memicu eksekusi nyata pada klaster Spark. Saat dijalankan, Spark akan mengambil sejumlah kecil baris (secara default biasanya 20 baris pertama) dari data terdistribusi dan mengirimkannya kembali ke layar pengguna untuk ditampilkan. Ini adalah cara yang aman untuk memverifikasi apakah transformasi data kita berjalan benar tanpa harus memproses seluruh dataset yang mungkin berukuran raksasa.

Output:

```
+-----+-----+-------------+----------+-------------------+
|col_1|col_2|        col_3|     col_4|              col_5|
+-----+-----+-------------+----------+-------------------+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|
|  300|400.0|string_test_3|2023-03-01|2023-01-03 12:00:00|
+-----+-----+-------------+----------+-------------------+
```

Output visual ini menampilkan data dalam bentuk grid tabular yang sangat mudah dibaca manusia. Kita bisa melihat header kolom di bagian atas, diikuti oleh baris-baris data yang dipisahkan oleh garis vertikal dan horizontal. Dari sini, kita bisa melakukan verifikasi visual (kualitatif): apakah format tanggalnya benar (2023-01-01)? Apakah angka desimalnya memiliki koma (200.0)? Apakah string-nya sesuai? Tampilan ini memberikan konfirmasi visual bahwa proses pembuatan DataFrame dari tahap-tahap sebelumnya telah sukses dan data tersimpan dengan akurat.

Melihat n baris teratas (*Viewing top n rows*)

Dalam eksplorasi data, terkadang kita tidak ingin membanjiri layar dengan terlalu banyak informasi, atau sebaliknya, pengaturan default (biasanya 20 baris) mungkin terlalu banyak untuk sekadar pengecekan cepat. Spark memberikan fleksibilitas untuk mengontrol secara presisi berapa banyak baris data yang ingin ditampilkan dalam satu pernyataan. Kita dapat membatasi tampilan ini dengan memberikan argumen angka spesifik ke dalam fungsi .show(). Jika parameter n ditentukan, Spark hanya akan mengambil dan menampilkan sejumlah baris tersebut dari bagian teratas DataFrame.

Implementasi Source Code:

```
data_df.show(2)
```

Pada baris kode ini, fungsi show dipanggil dengan argumen integer 2. Ini adalah instruksi langsung kepada driver Spark untuk membatasi output. Berbeda dengan show() tanpa argumen yang menggunakan nilai

bawaan, show(2) memerintahkan Spark untuk hanya memproses dan merender dua record pertama yang ditemukannya. Ini sangat efisien untuk pengecekan cepat struktur data tanpa membuang sumber daya komputasi untuk memformat baris-baris tambahan yang mungkin tidak diperlukan.

Output:

```
+-----+-----+-------------+----------+-------------------+
|col_1|col_2|        col_3|     col_4|              col_5|
+-----+-----+-------------+----------+-------------------+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|
+-----+-----+-------------+----------+-------------------+
only showing top 2 rows
```

Tabel yang dihasilkan jauh lebih ringkas. Kita bisa melihat bahwa hanya dua entri data yang muncul. Di bagian bawah tabel, Spark menyertakan catatan kaki informatif: only showing top 2 rows. Ini adalah konfirmasi bahwa data yang ditampilkan hanyalah sebagian kecil dari keseluruhan dataset, mencegah kesalahpahaman bahwa dataset tersebut hanya memiliki dua baris.

Melihat skema DataFrame (*Viewing DataFrame schema*)

Selain melihat isi data, memahami struktur atau metadata adalah hal fundamental. Fungsi printSchema() digunakan untuk menampilkan tata letak hierarkis dari DataFrame. Ini memberikan pandangan "X-ray" terhadap dataset, memperlihatkan nama kolom, tipe data, dan aturan nullability (apakah kolom boleh kosong atau tidak).

Implementasi Source Code:

```
data_df.printSchema()
```

Perintah printSchema() tidak mengembalikan data baris, melainkan mencetak metadata ke konsol. Fungsi ini membaca definisi struktural yang tersimpan dalam katalog DataFrame. Berbeda dengan .columns yang hanya memberi daftar nama, printSchema memberikan detail teknis tipe data (seperti long, double, date) yang sangat krusial untuk memastikan transformasi data selanjutnya (seperti operasi matematika atau tanggal) dapat berjalan tanpa error tipe data.

Output:

```
root
 |-- col_1: long (nullable = true)
 |-- col_2: double (nullable = true)
 |-- col_3: string (nullable = true)
 |-- col_4: date (nullable = true)
 |-- col_5: timestamp (nullable = true)
```

Output ditampilkan dalam format pohon (tree structure) yang mudah dibaca. Setiap baris mewakili satu kolom. Misalnya, col_1: long (nullable = true) memberi tahu kita tiga hal: nama kolomnya col_1, tipe datanya adalah bilangan bulat panjang (long), dan kolom ini mengizinkan nilai kosong (nullable = true).

Informasi ini sangat vital bagi data engineer untuk memvalidasi apakah skema yang terbentuk sesuai dengan desain awal database.

Melihat data secara vertikal (*Viewing data vertically*)

Terkadang, sebuah DataFrame memiliki jumlah kolom yang sangat banyak (tabel yang "lebar") sehingga jika ditampilkan dalam format tabel horizontal biasa, barisnya akan terpotong atau menjadi berantakan di layar terminal. Untuk mengatasi masalah keterbacaan ini, Spark menyediakan opsi tampilan vertikal. Metode ini memutar orientasi tampilan, menyajikan setiap baris data sebagai blok rekaman (record) terpisah, mirip dengan kartu data.

Implementasi Source Code:

```
data_df.show(1, vertical=True)
```

Kode ini menggunakan variasi dari fungsi show. Argumen pertama 1 meminta satu baris data, sedangkan parameter vertical=True adalah kunci utamanya. Parameter ini mengubah mode render dari tabular menjadi daftar kunci-nilai (key-value list). Ini sangat berguna saat melakukan debugging pada tabel dengan ratusan kolom, di mana scrolling ke samping akan menyulitkan pembacaan.

Output:

```
-RECORD 0--------------------
 col_1 | 100
 col_2 | 200.0
 col_3 | string_test_1
 col_4 | 2023-01-01
 col_5 | 2023-01-01 12:00:00
only showing top 1 row
```

Hasilnya menampilkan data dalam blok yang diawali dengan -RECORD 0--. Setiap kolom ditampilkan pada baris baru (misalnya col_1 di satu baris, col_2 di baris berikutnya). Format ini memastikan bahwa nilai data yang panjang atau jumlah kolom yang banyak tetap terbaca dengan jelas tanpa perlu menggeser layar ke kanan.

Melihat kolom data (*Viewing columns of data*)

Ada kalanya kita hanya perlu mengetahui daftar nama kolom yang tersedia dalam DataFrame tanpa peduli tipe data atau isinya, misalnya untuk keperluan iterasi atau pengecekan keberadaan kolom tertentu. Properti .columns menyediakan akses langsung ke daftar nama-nama atribut tersebut.

Implementasi Source Code:

```
data_df.columns
```

Kode ini memanggil atribut properti columns pada objek DataFrame. Berbeda dengan .show() yang mencetak ke layar, perintah ini mengembalikan sebuah objek list (daftar) Python yang berisi string nama-

nama kolom. Daftar ini kemudian bisa digunakan secara programatis dalam kode Python, misalnya untuk looping atau validasi kondisi.

Output:

```
['col_1', 'col_2', 'col_3', 'col_4', 'col_5']
```

Output yang dihasilkan adalah daftar Python standar (Python list). Ini menegaskan bahwa kita bisa berinteraksi dengan metadata Spark menggunakan struktur data asli Python, memudahkan integrasi logika pemrograman lebih lanjut.

Melihat statistik ringkasan (*Viewing summary statistics*)

Langkah awal dalam analisis data eksploratif (EDA) seringkali melibatkan pemahaman distribusi statistik data. Spark mempermudah ini dengan fungsi .describe(), yang secara otomatis menghitung metrik statistik utama seperti jumlah data (count), rata-rata (mean), standar deviasi (stddev), nilai minimum, dan maksimum untuk kolom-kolom numerik dan string.

Implementasi Source Code:

```
data_df.select('col_1', 'col_2', 'col_3').describe().show()
```

Baris kode ini melakukan serangkaian operasi berantai (chaining). Pertama, .select(...) memilih subset kolom yang relevan. Kemudian, .describe() melakukan komputasi statistik berat di belakang layar. Penting untuk dicatat bahwa describe() itu sendiri menghasilkan sebuah DataFrame baru yang berisi hasil statistik tersebut. Oleh karena itu, kita perlu memanggil .show() di ujung rantai untuk menampilkan DataFrame statistik ini ke layar.

Output:

```
+-------+-----+-----+-------------+
|summary|col_1|col_2|        col_3|
+-------+-----+-----+-------------+
|  count|    3|    3|            3|
|   mean|200.0|300.0|         NULL|
| stddev|100.0|100.0|         NULL|
|    min|  100|200.0|string_test_1|
|    max|  300|400.0|string_test_3|
+-------+-----+-----+-------------+
```

Tabel statistik ini memberikan wawasan instan mengenai profil data. Misalnya, pada baris mean, kita bisa melihat rata-rata nilai col_1 adalah 200.0. Pada baris min dan max, kita bisa melihat rentang nilai data. Perhatikan bahwa untuk kolom string (col_3), perhitungan rata-rata dan deviasi standar bernilai null karena operasi matematika tersebut tidak relevan untuk teks, namun nilai min/max tetap dihitung berdasarkan urutan alfabet (leksikografis).

Mengumpulkan data (*Collecting the data*)

Fungsi .collect() adalah metode untuk menarik seluruh data dari klaster terdistribusi (para worker) dan membawanya kembali ke satu mesin pusat (driver). Ini sering digunakan ketika kita ingin memproses hasil

akhir menggunakan pustaka Python lokal biasa. Namun, operasi ini memiliki risiko besar. Jika volume data yang ditarik melebihi kapasitas RAM pada mesin driver, akan terjadi Out-Of-Memory (OOM) Error yang menyebabkan aplikasi crash.

Implementasi Source Code:

```
    data_df.collect()
```

Perintah collect() adalah sebuah Action yang memaksa eksekusi. Berbeda dengan show() yang hanya mencetak tampilan, collect() mengembalikan data fisik dalam bentuk list of Row objects ke variabel Python. Karena sifatnya yang memuat semua data ke memori, metode ini sebaiknya hanya digunakan untuk dataset kecil atau hasil agregasi akhir yang ukurannya dapat diprediksi.

Output:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.date(2023, 1, 1),
col_5=datetime.datetime(2023, 1, 1, 12, 0)),
 Row(col_1=200, col_2=300.0, col_3='string_test_2', col_4=datetime.date(2023, 2, 1),
col_5=datetime.datetime(2023, 1, 2, 12, 0)),
 Row(col_1=300, col_2=400.0, col_3='string_test_3', col_4=datetime.date(2023, 3, 1),
col_5=datetime.datetime(2023, 1, 3, 12, 0))]
```

Outputnya bukan lagi tabel ASCII, melainkan daftar objek Python ([Row(...), Row(...)]). Ini menunjukkan bahwa data kini berada di memori lokal Python dan siap dimanipulasi dengan logika Python standar (non-Spark).

Menggunakan take (*Using take*)

Untuk menghindari risiko kehabisan memori saat menginspeksi data, metode .take(n) adalah alternatif yang jauh lebih aman dibandingkan collect(). Fungsi ini hanya mengambil n elemen pertama dari DataFrame. Karena hanya mengambil sebagian kecil, beban memori pada driver sangat minimal.

Implementasi Source Code:

```
    data_df.take(1)
```

Fungsi take(1) menginstruksikan Spark untuk memindai partisi data dan segera berhenti setelah menemukan 1 baris pertama, lalu mengirimkannya ke driver. Ini sangat efisien karena tidak perlu membaca seluruh dataset.

Output:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.date(2023, 1, 1), col_5=datetime.datetime(2023, 1, 1, 12, 0))]
```

Hasilnya adalah sebuah daftar (list) yang hanya berisi satu objek Row. Ini mengonfirmasi bahwa kita berhasil mengambil sampel data dengan aman.

Menggunakan tail (*Using tail*)

Kebalikan dari take, fungsi .tail(n) digunakan untuk mengambil n elemen terakhir dari DataFrame. Ini berguna untuk memeriksa data yang baru saja masuk (jika data diurutkan berdasarkan waktu) atau bagian akhir dari sebuah dataset.

Implementasi Source Code:

```
    data_df.tail(1)
```

Kode ini meminta satu baris data paling bawah. Perlu dicatat bahwa operasi ini mungkin lebih mahal secara komputasi dibandingkan head atau take, karena Spark mungkin perlu memindai data hingga akhir untuk menemukan baris terakhir tersebut.

Output:

```
[Row(col_1=300, col_2=400.0, col_3='string_test_3', col_4=datetime.date(2023, 3, 1), col_5=datetime.datetime(2023, 1, 3, 12, 0))]
```

Output menampilkan baris dengan col_1=300, yang merupakan data terakhir dalam dataset contoh kita.

Menggunakan head (*Using head*)

Fungsi .head(n) memiliki perilaku yang sangat mirip dengan take(n). Ia mengembalikan n baris pertama dari DataFrame. Dalam konteks PySpark, head sering digunakan secara bergantian dengan take untuk mengambil sampel awal data.

Implementasi Source Code:

```
    data_df.head(1)
```

Sama seperti take(1), kode ini mengambil satu elemen teratas. Perbedaan nuansanya seringkali terletak pada kebiasaan pengguna (pengguna Pandas mungkin lebih familiar dengan istilah head).

Output:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.date(2023, 1, 1),
col_5=datetime.datetime(2023, 1, 1, 12, 0))]
```

Output identik dengan hasil take(1), yaitu baris pertama data.

Menghitung jumlah baris data (*Counting the number of rows of data*)

Operasi dasar lainnya adalah menghitung total volume data. Fungsi .count() digunakan untuk mendapatkan jumlah total baris yang ada dalam DataFrame.

Implementasi Source Code:

```
    data_df.count()
```

Perintah ini memicu Action penghitungan. Spark akan menginstruksikan seluruh worker untuk menghitung baris di partisi masing-masing, lalu mengirimkan jumlah totalnya ke driver untuk dijumlahkan.

Output:

```
3
```

Analisis Output:

Angka 3 dikembalikan, yang secara akurat mencerminkan jumlah baris yang kita masukkan saat pembuatan DataFrame di awal bab.

Ringkasan metode pengambilan data (*Summary of data retrieval methods*)

Setelah mempelajari berbagai teknik pengambilan data, modul ini memberikan rangkuman perbandingan antara metode take, collect, show, head, dan tail. Pemahaman perbedaan ini sangat krusial untuk manajemen memori dan efisiensi kerja.

- **take(n)** dan **head(n)**: Kedua fungsi ini memiliki karakteristik serupa, yaitu mengembalikan *n* elemen pertama. take mengembalikan array berisi elemen, sementara head secara spesifik mengembalikan daftar objek Row. Keduanya melakukan evaluasi malas (*lazy evaluation*) dan sangat aman untuk inspeksi subset data kecil tanpa membebani memori.
- **collect()**: Fungsi ini mengambil **seluruh** elemen dari DataFrame dan mengembalikannya sebagai daftar ke *driver*. Ini adalah metode yang paling berisiko; pengguna harus berhati-hati karena jika ukuran dataset melebihi kapasitas RAM *driver*, akan terjadi *Out-of-Memory Error*. Metode ini sebaiknya hanya digunakan untuk dataset kecil atau hasil agregasi akhir.
- **show(n)**: Berbeda dengan yang lain yang mengembalikan objek data, show hanya menampilkan data secara visual dalam format tabel untuk keperluan inspeksi manusia atau *debugging*.

- **tail(n)**: Kebalikan dari head, fungsi ini mengambil baris dari akhir dataset. Operasi ini cenderung lebih mahal secara komputasi dibandingkan head karena mungkin melibatkan pemindaian seluruh dataset untuk menemukan bagian akhirnya.

Mengonversi PySpark DataFrame menjadi Pandas DataFrame (*Converting a PySpark DataFrame to a Pandas DataFrame*)

Dalam ekosistem Python, pustaka Pandas sangat populer untuk analisis data. Terkadang, dalam alur kerja PySpark, kita perlu memanfaatkan fungsi-fungsi khusus yang hanya tersedia di Pandas atau melakukan visualisasi data menggunakan library plotting lokal. Untuk menjembatani kebutuhan ini, PySpark menyediakan metode konversi.

Implementasi Source Code:

```
data_df.toPandas()
```

Kode data_df.toPandas() adalah perintah untuk mengubah objek DataFrame terdistribusi milik Spark menjadi objek DataFrame lokal milik Pandas. Poin teknis yang sangat penting untuk dipahami di sini adalah arsitektur memori. Python (Pandas) secara inheren tidak bekerja secara terdistribusi; ia berjalan pada satu mesin (single node). Oleh karena itu, saat metode toPandas() dipanggil, Spark driver harus mengumpulkan (collect) seluruh data dari berbagai node klaster ke dalam memori lokalnya sendiri.

Ini membawa implikasi risiko yang sama dengan perintah collect(). Sebelum menjalankan kode ini, pengembang wajib memastikan bahwa memori pada mesin *driver* cukup besar untuk menampung seluruh dataset. Jika data terlalu besar, proses ini akan gagal dan menyebabkan aplikasi berhenti.

Output:

	col_1	col_2	col_3	col_4	col_5
0	100	200.0	string_test_1	2023-01-01	2023-01-01 12:00:00
1	200	300.0	string_test_2	2023-02-01	2023-01-02 12:00:00
2	300	400.0	string_test_3	2023-03-01	2023-01-03 12:00:00

Output yang dihasilkan adalah representasi tabel Pandas yang familiar. Data yang tadinya tersebar di klaster kini telah bersatu menjadi satu entitas DataFrame Pandas di memori lokal, siap untuk diproses menggunakan sintaks Pandas biasa.

Cara memanipulasi data pada baris dan kolom (*How to manipulate data on rows and columns*)

Manipulasi data adalah inti dari proses ETL (Extract, Transform, Load). Bagian ini akan membahas berbagai teknik untuk mengubah struktur DataFrame, mulai dari memilih, menyaring, hingga memotong data berdasarkan kriteria tertentu. Karena sifat immutable dari Spark DataFrame, setiap operasi manipulasi ini tidak mengubah DataFrame asli, melainkan menghasilkan DataFrame baru.

Memilih kolom (*Selecting columns*)

Operasi paling dasar dalam manipulasi kolom adalah seleksi. Kita sering kali tidak membutuhkan semua kolom yang ada dalam dataset besar. Fungsi select() digunakan untuk memproyeksikan atau mengambil kolom-kolom tertentu saja yang relevan dengan analisis kita.

Implementasi Source Code:

```
from pyspark.sql import Column
data_df.select(data_df.col_3).show()
```

Pada kode di atas, kita mengimpor kelas Column (meskipun pada contoh spesifik ini penggunaan select langsung memanggil atribut objek). Perintah data_df.select(data_df.col_3) secara spesifik meminta Spark untuk hanya mengambil kolom col_3. Argumen di dalam fungsi select dapat berupa string nama kolom atau objek kolom itu sendiri.

Hal fundamental yang perlu dicatat dari kode ini adalah prinsip *immutability*. DataFrame hasil seleksi ini adalah objek baru. DataFrame asli (data_df) masih tetap utuh dengan 5 kolomnya. Jika kita ingin menyimpan hasil seleksi ini, kita harus menugaskannya ke variabel baru atau menimpa variabel lama.

Output:

```
+-------------+
|        col_3|
+-------------+
|string_test_1|
|string_test_2|
|string_test_3|
+-------------+
```

Tabel hasil hanya terdiri dari satu kolom vertikal. Ini membuktikan bahwa fungsi seleksi berhasil mengisolasi atribut data yang diminta.

Membuat kolom (*Creating columns*)

Selain memilih kolom yang ada, kebutuhan umum lainnya adalah menambahkan kolom baru, baik itu kolom kosong, kolom dengan nilai konstan, atau kolom hasil perhitungan dari kolom lain. Di PySpark, fungsi utama untuk tugas ini adalah withColumn().

Implementasi Source Code:

```
from pyspark.sql import functions as F
data_df = data_df.withColumn("col_6", F.lit("A"))
data_df.show()
```

Kode ini memperkenalkan modul fungsi PySpark (pyspark.sql.functions), yang dialiaskan sebagai F. Fungsi withColumn menerima dua argumen: nama kolom baru ("col_6") dan ekspresi yang mendefinisikan isinya.

Di sini, kita ingin mengisi kolom baru dengan nilai konstan huruf "A". Namun, kita tidak bisa langsung memasukkan string "A" begitu saja. Spark membutuhkan ekspresi kolom (*Column expression*). Oleh karena itu, kita menggunakan fungsi F.lit("A") (singkatan dari *literal*). Fungsi ini memberi tahu Spark untuk memperlakukan "A" sebagai nilai konstan yang harus diterapkan ke setiap baris dalam DataFrame tersebut.

Hasil operasi ini kemudian disimpan kembali ke variabel data_df, secara efektif menimpa referensi DataFrame lama dengan versi baru yang memiliki tambahan kolom.

Output:

```
+-----+-----+-------------+----------+-------------------+-----+
|col_1|col_2|        col_3|     col_4|              col_5|col_6|
+-----+-----+-------------+----------+-------------------+-----+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|    A|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|    A|
|  300|400.0|string_test_3|2023-03-01|2023-01-03 12:00:00|    A|
+-----+-----+-------------+----------+-------------------+-----+
```

Tabel yang ditampilkan sekarang lebih lebar dengan kehadiran col_6. Nilai A terisi secara seragam di seluruh baris, mengonfirmasi fungsi literal bekerja sesuai harapan.

Menghapus kolom (*Dropping columns*)

Kebalikan dari menambahkan kolom, terkadang kita perlu membuang kolom yang tidak relevan atau bersifat sementara untuk menghemat memori dan merapikan data. Fungsi drop() digunakan untuk tujuan ini.

Implementasi Source Code:

```
data_df = data_df.drop("col_5")
data_df.show()
```

Fungsi drop("col_5") menginstruksikan Spark untuk menghapus kolom bernama col_5 dari struktur DataFrame. Fungsi ini cukup fleksibel; kita bisa menghapus satu kolom, atau beberapa kolom sekaligus dengan memisahkan nama kolom menggunakan koma (misal: drop("col_4", "col_5")). Sebuah fitur menarik dari fungsi drop adalah ketahanannya terhadap kesalahan (error resilience): jika kita mencoba menghapus kolom yang sebenarnya tidak ada, Spark tidak akan memunculkan pesan error, melainkan hanya mengembalikan DataFrame apa adanya tanpa perubahan.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|        col_3|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
|  300|400.0|string_test_3|2023-03-01|    A|
+-----+-----+-------------+----------+-----+
```

Tabel hasil menunjukkan bahwa kolom col_5 (yang sebelumnya berisi timestamp) telah hilang. Struktur data kini menjadi lebih ramping.

Memperbarui kolom (*Updating columns*)

Bagaimana jika kita ingin mengubah nilai dalam kolom yang sudah ada? Di PySpark, tidak ada fungsi khusus bernama "update". Sebaliknya, kita kembali menggunakan fungsi withColumn(). Jika nama kolom yang kita berikan sebagai argumen pertama sudah ada di dalam DataFrame, Spark akan mengganti (menimpa) kolom lama tersebut dengan nilai baru hasil perhitungan.

Implementasi Source Code:

```
data_df.withColumn("col_2", F.col("col_2") / 100).show()
```

Pada baris kode ini, kita memperbarui col_2. Logika perhitungannya ada pada argumen kedua: F.col("col_2") / 100. Fungsi F.col sangat penting di sini; ia digunakan untuk merujuk pada nilai yang ada di kolom col_2 saat ini. Tanpa F.col (atau sintaks kolom lainnya), Python tidak akan mengerti bahwa kita sedang melakukan operasi matematika pada sebuah kolom DataFrame. Kode ini secara efektif membagi setiap nilai dalam col_2 dengan angka 100.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|        col_3|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|  2.0|string_test_1|2023-01-01|    A|
|  200|  3.0|string_test_2|2023-02-01|    A|
|  300|  4.0|string_test_3|2023-03-01|    A|
+-----+-----+-------------+----------+-----+
```

Hasil visual menunjukkan transformasi nilai pada col_2. Operasi pembagian telah diterapkan pada setiap baris, mendemonstrasikan kemampuan Spark melakukan operasi aritmatika kolom secara efisien (column-wise operation).

Mengganti nama kolom (*Renaming columns*)

Terkadang nama kolom perlu diubah agar lebih deskriptif atau sesuai dengan standar penamaan sistem tujuan. PySpark menyediakan fungsi spesifik withColumnRenamed() untuk tugas ini, tanpa mengubah isi datanya.

Implementasi Source Code:

```
data_df = data_df.withColumnRenamed("col_3", "string_col")
data_df.show()
```

Fungsi withColumnRenamed menerima dua argumen string: nama kolom lama ("col_3") dan nama kolom baru ("string_col"). Ini adalah operasi metadata murni; Spark hanya mengubah referensi nama di katalog skema tanpa perlu memproses ulang data fisiknya secara berat.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
|  300|400.0|string_test_3|2023-03-01|    A|
+-----+-----+-------------+----------+-----+
```

Perubahan terlihat pada bagian header tabel. Kolom ketiga kini berlabel string_col, sementara data di bawahnya (string_test_1, dll) tetap sama persis.

Mencari nilai unik dalam kolom (*Finding unique values in a column*)

Dalam analisis data, mengetahui variasi nilai dalam sebuah kolom sangatlah penting, misalnya untuk mengetahui kategori produk apa saja yang terjual. Fungsi distinct() digunakan untuk melakukan deduplikasi data, sehingga hanya menyisakan nilai-nilai yang unik.

Implementasi Source Code:

```
data_df.select("col_6").distinct().show()
```

Kode ini merupakan rangkaian operasi (chained operation). Pertama, select("col_6") mengisolasi kolom col_6. Kemudian, distinct() memindai seluruh data pada kolom tersebut dan menghapus semua duplikasi. Mengingat col_6 sebelumnya kita isi dengan nilai konstan "A", maka secara logika hanya akan ada satu nilai unik yang tersisa.

Output:

```
+-----+
|col_6|
+-----+
|    A|
+-----+
```

Output menampilkan tabel dengan satu baris berisi "A". Ini memvalidasi bahwa meskipun DataFrame memiliki banyak baris, variasi data pada col_6 hanyalah tunggal. Jika diterapkan pada kolom lain yang lebih bervariasi, fungsi ini akan menampilkan daftar semua kemungkinan nilai yang ada.

Menghitung nilai unik (*Count distinct values*)

Setelah mengetahui cara melihat apa saja nilai unik dalam sebuah kolom menggunakan distinct(), langkah analisis selanjutnya sering kali adalah menghitung berapa banyak variasi nilai tersebut. Ini disebut sebagai kardinalitas data. PySpark menyediakan fungsi agregasi countDistinct untuk tujuan ini. Selain itu, untuk membuat laporan lebih rapi, kita sering memberi nama alias pada kolom hasil perhitungan agar mudah dibaca.

Implementasi Source Code:

```
data_df.select(F.countDistinct("col_6").alias("Total_Unique")).show()
```

Kode ini menggunakan fungsi F.countDistinct("col_6") yang bekerja dengan cara mengumpulkan semua nilai unik di kolom col_6 dan menghitung jumlahnya. Tanpa fungsi tambahan, hasil perhitungan ini biasanya akan diberi nama kolom default yang panjang dan teknis (seperti count(DISTINCT col_6)). Oleh karena itu, kita menyambungkan fungsi .alias("Total Unique") di belakangnya. Fungsi alias ini berguna untuk mengganti nama header kolom hasil secara langsung menjadi "Total Unique", sehingga tabel output menjadi lebih profesional dan mudah dimengerti.

Output:

```
+------------+
|Total_Unique|
+------------+
|           1|
+------------+
```

Hasilnya adalah angka 1. Ini masuk akal karena pada langkah sebelumnya kita mengisi kolom col_6 dengan nilai konstan "A". Artinya, hanya ada satu jenis nilai di kolom tersebut. Jika kolom berisi data yang lebih bervariasi, misalnya nama-nama hari, outputnya akan menunjukkan angka 7.

Mengubah huruf besar/kecil kolom (*Changing the case of a column*)

Dalam pembersihan data teks (text preprocessing), standardisasi format huruf adalah tugas rutin. Misalnya, menyamakan semua input pengguna menjadi huruf besar (uppercase) atau huruf kecil (lowercase) agar konsisten. Spark menyediakan fungsi transformasi string bawaan untuk melakukan ini pada seluruh kolom sekaligus tanpa perlu melakukan looping manual.

Implementasi Source Code:

```
from pyspark.sql.functions import upper
data_df.withColumn('upper_string_col', upper(data_df.string_col)).show()
```

Langkah pertama adalah mengimpor fungsi upper dari modul fungsi PySpark. Selanjutnya, kita menggunakan withColumn untuk membuat kolom baru bernama upper_string_col. Logika transformasi ada pada bagian upper(data_df.string_col). Fungsi ini mengambil nilai dari kolom string_col (yang berisi 'string_test_1', dst.) dan mengonversinya menjadi huruf kapital.

Penting untuk dicatat dalam analisis kode ini: hasil operasi hanya di-show() ke layar dan tidak disimpan kembali ke variabel data_df (karena tidak ada penugasan data_df = ...). Ini berarti perubahan ini hanya bersifat sementara untuk ditampilkan, dan DataFrame asli tidak berubah strukturnya secara permanen.

Output:

```
+-----+-----+-------------+----------+-----+----------------+
|col_1|col_2|   string_col|     col_4|col_6|upper_string_col|
+-----+-----+-------------+----------+-----+----------------+
|  100|200.0|string_test_1|2023-01-01|    A|   STRING_TEST_1|
|  200|300.0|string_test_2|2023-02-01|    A|   STRING_TEST_2|
|  300|400.0|string_test_3|2023-03-01|    A|   STRING_TEST_3|
+-----+-----+-------------+----------+-----+----------------+
```

Output memperlihatkan perbandingan sisi-ke-sisi. Kolom asli string_col masih dalam huruf kecil, sedangkan kolom baru upper_string_col berisi teks yang sama namun dalam format huruf kapital semua. Ini membuktikan fungsi transformasi teks berjalan sukses.

Menyaring DataFrame (*Filtering a DataFrame*)

Penyaringan atau filtering adalah salah satu operasi paling fundamental dalam manipulasi data. Tujuannya adalah untuk mendapatkan subset data yang memenuhi kriteria tertentu, mirip dengan klausa WHERE dalam SQL. Dengan menyaring, kita bisa membuang data yang tidak relevan dan fokus hanya pada baris-baris yang kita butuhkan untuk analisis.

Implementasi Source Code:

```
data_df.filter(data_df.col_1 == 100).show()
```

Fungsi .filter() menerima ekspresi logika Boolean. Dalam contoh ini, kondisinya adalah data_df.col_1 == 100. Spark akan mengevaluasi setiap baris dalam DataFrame; jika nilai di col_1 sama dengan 100, baris tersebut dipertahankan. Jika tidak, baris tersebut dibuang dari hasil tampilan. Operator == adalah operator pembanding standar untuk kesetaraan.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
+-----+-----+-------------+----------+-----+
```

Dari total 3 baris data awal, hasil filter hanya menyisakan 1 baris. Ini karena hanya ada satu baris yang nilai col_1-nya 100. Baris dengan nilai 200 dan 300 telah disembunyikan.

Operator logika dalam DataFrame (*Logical operators in a DataFrame*)

Seringkali kondisi penyaringan tidak sesederhana satu syarat saja. Kita mungkin perlu menggabungkan beberapa kriteria, misalnya "cari data yang nilai A-nya X DAN nilai B-nya Y". Untuk kebutuhan kompleks ini, Spark mendukung operator logika seperti AND dan OR.

1. Operator AND

Implementasi Source Code:

```
data_df.filter((data_df.col_1 == 100)
        & (data_df.col_6 == 'A')).show()
```

Di sini kita menggunakan operator & (ampersand) yang merepresentasikan logika DAN. Syaratnya diperketat: baris data harus memiliki col_1 bernilai 100 DAN col_6 bernilai 'A' secara bersamaan. Poin sintaks yang sangat krusial dalam PySpark (dan Pandas) adalah penggunaan tanda kurung (). Setiap kondisi individual wajib dibungkus dalam tanda kurung, seperti (data_df.col_1 == 100). Jika tanda kurung ini dihilangkan, Python akan bingung dengan urutan prioritas operator dan menyebabkan error.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
+-----+-----+-------------+----------+-----+
```

Karena baris dengan col_1=100 kebetulan juga memiliki col_6='A', baris tersebut lolos seleksi dan ditampilkan.

2. Operator OR

Implementasi Source Code:

```
data_df.filter((data_df.col_1 == 100)
        | (data_df.col_2 == 300.00)).show()
```

Kode ini menggunakan operator | (pipa vertikal) yang mewakili logika ATAU (OR). Logika ini lebih longgar dibandingkan AND. Baris data akan diambil jika salah satu syarat terpenuhi: entah col_1 bernilai 100, atau col_2 bernilai 300.0.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
+-----+-----+-------------+----------+-----+
```

Hasilnya menampilkan dua baris. Baris pertama muncul karena col_1 bernilai 100 (meskipun col_2-nya 200). Baris kedua muncul karena col_2 bernilai 300 (meskipun col_1-nya 200). Ini menunjukkan bagaimana operator OR memperluas cakupan pencarian data.

Menggunakan isin() (*Using isin*)

Bagaimana jika kita ingin memfilter data berdasarkan daftar nilai yang banyak? Misalnya, kita ingin mencari data dengan kode 100, 200, 500, dan seterusnya. Menulis kondisi OR berulang-ulang (col==100 | col==200 | ...) tentu tidak efisien dan membuat kode berantakan. Solusinya adalah fungsi isin(), yang mengecek apakah nilai kolom terdapat di dalam sebuah daftar (list) yang kita berikan.

Implementasi Source Code:

```
list = [100, 200]
data_df.filter(data_df.col_1.isin(list)).show()
```

Pertama, kita mendefinisikan sebuah daftar Python biasa bernama list yang berisi angka target [100, 200]. Kemudian, di dalam fungsi filter, kita memanggil metode .isin(list) pada kolom col_1. Spark akan memeriksa setiap baris: "Apakah nilai col_1 ada di dalam daftar [100, 200]?". Jika ya, baris diambil. Ini membuat kode jauh lebih bersih dan mudah dibaca dibandingkan menggunakan banyak operator OR.

Output:

```
+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
+-----+-----+-------------+----------+-----+
```

Output menampilkan dua baris data, yaitu baris yang memiliki nilai col_1 sebesar 100 dan baris dengan nilai 200. Baris ketiga (dengan nilai 300) tidak muncul karena angka 300 tidak ada dalam daftar target kita.

Konversi Tipe Data (*Datatype conversions*)

Dalam pengolahan data, sering kali tipe data yang kita terima tidak sesuai dengan kebutuhan analisis. Misalnya, angka mungkin terbaca sebagai teks, atau tanggal terbaca sebagai string. PySpark menyediakan mekanisme yang kuat untuk melakukan casting atau pengubahan tipe data antar kolom. Modul ini memperkenalkan beberapa pendekatan untuk melakukan tugas ini: menggunakan fungsi cast, ekspresi SQL, atau kueri SQL murni.

1. Menggunakan Fungsi Cast

Implementasi Source Code:

```
from pyspark.sql.functions import col
from pyspark.sql.types import StringType,BooleanType,DateType,IntegerType
data_df_2 = data_df.withColumn("col_4",col("col_4").cast(StringType())) \
    .withColumn("col_1",col("col_1").cast(IntegerType()))
data_df_2.printSchema()
data_df.show()
```

Langkah pertama dalam blok kode ini adalah mengimpor tipe-tipe data spesifik dari modul pyspark.sql.types (seperti StringType, IntegerType) yang akan menjadi target konversi kita. Proses konversi dilakukan menggunakan fungsi .cast().

Dalam implementasinya, kita menggunakan metode withColumn yang dirangkai (*chained*). Pertama, col_4 (yang aslinya bertipe *date*) diubah menjadi *string* menggunakan .cast(StringType()). Kedua, col_1 (yang aslinya bertipe *long*) diubah menjadi *integer* menggunakan .cast(IntegerType()). Perlu diperhatikan bahwa operasi ini disimpan ke dalam variabel baru data_df_2, menegaskan kembali sifat *immutability* dari DataFrame; kita tidak mengubah data_df asli, melainkan menciptakan versi baru dengan skema yang berbeda.

Output:

```
root
 |-- col_1: integer (nullable = true)
 |-- col_2: double (nullable = true)
 |-- string_col: string (nullable = true)
 |-- col_4: string (nullable = true)
 |-- col_6: string (nullable = false)

+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
|  300|400.0|string_test_3|2023-03-01|    A|
+-----+-----+-------------+----------+-----+
```

Analisis skema (printSchema) mengonfirmasi perubahan tersebut: col_1 kini berstatus integer (sebelumnya long) dan col_4 berstatus string (sebelumnya date). Namun, jika kita melihat tampilan data lewat .show(), secara visual isinya terlihat sama (misal "2023-01-01"). Ini menunjukkan bahwa perubahan terjadi pada level metadata penyimpanan, bukan pada nilai visual datanya.

2. Menggunakan selectExpr()

Metode selectExpr (Select Expression) adalah jalan pintas yang memungkinkan kita menulis perintah transformasi menggunakan sintaks SQL di dalam kode Python, tanpa perlu mengimpor tipe data satu per satu. Ini sangat efisien untuk konversi cepat.

Implementasi Source Code:

```
data_df_3 = data_df_2.selectExpr("cast(col_4 as date) col_4",
    "cast(col_1 as long) col_1")
data_df_3.printSchema()
```

Pada kode ini, kita mencoba mengembalikan tipe data yang sebelumnya kita ubah. Kita menggunakan string SQL "cast(col_4 as date) col_4". Frasa ini berarti: "Ambil col_4, lakukan casting menjadi tipe date, lalu namai hasilnya tetap sebagai col_4". Perintah ini jauh lebih ringkas dibandingkan metode sebelumnya karena menggabungkan seleksi kolom dan transformasi tipe data dalam satu baris string.

Output:

```
root
 |-- col_4: date (nullable = true)
 |-- col_1: long (nullable = true)
```

Skema membuktikan bahwa kita berhasil mengembalikan (me-revert) tipe data ke bentuk asalnya (date dan long). Ini menunjukkan fleksibilitas Spark dalam bolak-balik mengubah tipe data sesuai kebutuhan di tengah alur pemrosesan.

3. Menggunakan SQL Murni

Bagi pengguna yang lebih nyaman dengan bahasa kueri database tradisional, Spark memungkinkan manipulasi DataFrame menggunakan perintah SQL standar. Syarat utamanya adalah mendaftarkan DataFrame sebagai "tabel virtual" atau view terlebih dahulu.

Implementasi Source Code:

```
data_df_3.createOrReplaceTempView("CastExample")
data_df_4 = spark.sql("SELECT DOUBLE(col_1), DATE(col_4) from CastExample")
data_df_4.printSchema()
data_df_4.show(truncate=False)
```

Langkah pertama adalah createOrReplaceTempView("CastExample"). Fungsi ini membuat tabel sementara bernama "CastExample" yang merujuk pada data di data_df_3. Tabel ini hanya hidup selama sesi Spark aktif.

Langkah kedua adalah menjalankan spark.sql(...). Di dalam string SQL tersebut, kita menggunakan fungsi SQL DOUBLE() dan DATE() untuk melakukan konversi tipe data secara eksplisit saat melakukan seleksi.

Output:

```
root
 |-- col_1: double (nullable = true)
 |-- col_4: date (nullable = true)

+-----+----------+
|col_1|col_4     |
+-----+----------+
|100.0|2023-01-01|
|200.0|2023-02-01|
|300.0|2023-03-01|
+-----+----------+
```

Output skema menunjukkan col_1 sekarang bertipe double (bilangan desimal), terlihat dari tampilannya yang kini memiliki desimal (100.0, 200.0). Ini membuktikan bahwa kueri SQL berhasil dieksekusi dan hasilnya dikembalikan dalam bentuk DataFrame baru.

Menghapus nilai null dari DataFrame (*Dropping null values from a DataFrame*)

Salah satu masalah paling umum dalam kualitas data adalah keberadaan nilai null (kosong/tidak ada data). Nilai null bisa membuat hasil analisis menjadi bias atau menyebabkan error pada perhitungan. Oleh karena itu, membersihkan nilai null adalah keterampilan wajib bagi analis data. Untuk mendemonstrasikan ini, modul membuat dataset baru yang sengaja dibuat "kotor" dengan menyisipkan nilai kosong.

Persiapan Data (Membuat salary_data):

Implementasi Source Code:

```
salary_data = [("John", "Field-eng", 3500),
    ("Michael", "Field-eng", 4500),
    ("Robert", None, 4000),
    ("Maria", "Finance", 3500),
    ("John", "Sales", 3000),
    ("Kelly", "Finance", 3500),
    ("Kate", "Finance", 3000),
    ("Martin", None, 3500),
    ("Kiran", "Sales", 2200),
    ("Michael", "Field-eng", 4500)
  ]
columns= ["Employee", "Department", "Salary"]
salary_data = spark.createDataFrame(data = salary_data, schema = columns)
salary_data.printSchema()

 salary_data.show()
```

Kode ini membuat DataFrame baru bernama salary_data. Kita mendefinisikan daftar tuple, di mana beberapa entri secara eksplisit diberikan nilai None (representasi Python untuk Null), seperti pada baris data pegawai bernama "Robert" yang tidak memiliki departemen, dan "Martin" yang juga departemennya None. Spark secara cerdas akan mengonversi None dari Python menjadi null dalam sistem Spark saat fungsi createDataFrame dijalankan.

Output:

```
root
 |-- Employee: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|    John| Field-eng|  3500|
| Michael| Field-eng|  4500|
|  Robert|      NULL|  4000|
|   Maria|   Finance|  3500|
|    John|     Sales|  3000|
|   Kelly|   Finance|  3500|
|    Kate|   Finance|  3000|
|  Martin|      NULL|  3500|
|   Kiran|     Sales|  2200|
| Michael| Field-eng|  4500|
+--------+----------+------+
```

Pada tabel output, kita bisa melihat dengan jelas adanya tulisan null pada kolom Department untuk baris Robert dan Martin. Ini adalah data yang perlu kita bersihkan.

Implementasi Penghapusan Null (dropna):

Implementasi Source Code:

```
salary_data.dropna().show()
```

Fungsi dropna() adalah solusi praktis untuk masalah ini. Secara default (tanpa argumen tambahan), fungsi ini akan menghapus seluruh baris jika terdapat setidaknya satu saja nilai null di kolom manapun dalam baris tersebut. Ini adalah metode pembersihan yang agresif untuk memastikan integritas data penuh.

Output:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|    John| Field-eng|  3500|
| Michael| Field-eng|  4500|
|   Maria|   Finance|  3500|
|    John|     Sales|  3000|
|   Kelly|   Finance|  3500|
|    Kate|   Finance|  3000|
|   Kiran|     Sales|  2200|
| Michael| Field-eng|  4500|
+--------+----------+------+
```

Jika dibandingkan dengan tabel sebelumnya, baris Robert dan Martin telah hilang sepenuhnya. Hal ini karena kedua baris tersebut mengandung null di kolom departemen. Baris data yang tersisa hanyalah baris-baris yang "lengkap" (tidak memiliki nilai kosong sama sekali).

Menghapus duplikat dari DataFrame (*Dropping duplicates from a DataFrame*)

Selain nilai null, duplikasi data (data ganda) adalah anomali yang sering terjadi dan membuat data menjadi "kotor", misalnya karena kesalahan input ganda atau proses penggabungan data yang tidak sempurna. PySpark menyediakan fungsi dropDuplicates() untuk mengatasi redundansi ini.

Implementasi Source Code:

```
new_salary_data = salary_data.dropDuplicates().show()
```

Pada dataset salary_data yang kita buat sebelumnya, terdapat dua entri yang identik untuk pegawai bernama "Michael" (sama-sama "Field-eng" dengan gaji 4500). Kode dropDuplicates() akan memindai seluruh baris dan membandingkan isinya. Jika ditemukan dua atau lebih baris yang isinya persis sama di semua kolom, Spark hanya akan mempertahankan satu salinan saja dan membuang sisanya.

Output:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|    John| Field-eng|  3500|
| Michael| Field-eng|  4500|
|   Maria|   Finance|  3500|
|    John|     Sales|  3000|
|   Kelly|   Finance|  3500|
|    Kate|   Finance|  3000|
|   Kiran|     Sales|  2200|
|  Robert|      NULL|  4000|
|  Martin|      NULL|  3500|
+--------+----------+------+
```

Hasilnya menunjukkan bahwa duplikasi data "Michael" telah dieliminasi. Sekarang Michael hanya muncul satu kali dalam daftar, menjadikan data lebih bersih dan akurat untuk perhitungan agregasi selanjutnya.

Menggunakan agregat dalam DataFrame (*Using aggregates in a DataFrame*)

Setelah kita mempelajari teknik penyaringan (filtering), langkah logis berikutnya dalam analisis data adalah melakukan agregasi. Agregasi adalah proses penggabungan beberapa baris data menjadi satu nilai ringkasan tunggal. Ini sangat penting untuk mendapatkan wawasan makro, seperti total penjualan, rata-rata gaji, atau nilai maksimum dalam periode tertentu. Spark menyediakan berbagai metode agregasi bawaan seperti avg (rata-rata), count (jumlah), max (maksimum), min (minimum), dan sum (penjumlahan).

1. Rata-rata (*Average - avg*)

Fungsi avg digunakan untuk menghitung nilai rata-rata aritmatika dari sebuah kolom numerik. Ini adalah salah satu metrik statistik paling dasar untuk memahami tendensi sentral dari data.

Implementasi Source Code:

```
from pyspark.sql.functions import countDistinct, avg
salary_data.select(avg('Salary')).show()
```

Pada baris kode ini, kita menggunakan fungsi avg di dalam metode select. Artinya, kita memilih kolom 'Salary', namun sebelum menampilkannya, Spark diperintahkan untuk memproses seluruh nilai di kolom tersebut dan menghitung rata-ratanya. Hasilnya bukan lagi daftar gaji individu, melainkan satu angka tunggal yang mewakili rata-rata gaji seluruh karyawan dalam DataFrame salary_data.

Output:

```
+-----------+
|avg(Salary)|
+-----------+
|     3520.0|
+-----------+
```

Output menampilkan angka 3520.0. Ini berarti jika kita menjumlahkan semua gaji karyawan dan membaginya dengan jumlah karyawan, kita mendapatkan angka tersebut. Perhitungan ini dilakukan secara otomatis oleh Spark.

2. Jumlah (*Count*)

Fungsi count digunakan untuk menghitung berapa banyak total baris atau entri data yang ada. Berbeda dengan contoh sebelumnya yang menggunakan select, modul ini memperkenalkan cara penulisan sintaks alternatif menggunakan fungsi agg (singkatan dari aggregate).

Implementasi Source Code:

```
salary_data.agg({'Salary':'count'}).show()
```

Di sini kita menggunakan metode .agg(). Metode ini menerima input berupa kamus (dictionary) Python, di mana kuncinya adalah nama kolom ('Salary') dan nilainya adalah fungsi agregasi yang ingin diterapkan ('count'). Perintah ini diterjemahkan sebagai: "Lakukan agregasi pada kolom Salary menggunakan metode count". Ini menunjukkan fleksibilitas sintaks di PySpark; kita bisa menggunakan ekspresi kolom atau dictionary mapping.

Output:

```
+-------------+
|count(Salary)|
+-------------+
|           10|
+-------------+
```

Hasilnya adalah 10. Ini menunjukkan bahwa terdapat total 10 entri data (gaji) yang tercatat dalam DataFrame kita.

3. Hitung nilai unik (*Count distinct values*)

Terkadang kita tidak ingin menghitung total baris, melainkan berapa banyak variasi nilai yang ada. Misalnya, dalam konteks gaji, mungkin ada banyak karyawan yang memiliki gaji sama. Fungsi countDistinct membantu kita mengetahui berapa banyak nominal gaji yang berbeda-beda dalam perusahaan tersebut.

Implementasi Source Code:

```
salary_data.select(countDistinct("Salary").alias("Distinct Salary")).show()
```

Kode ini menggunakan countDistinct("Salary") di dalam select. Spark akan mengeliminasi duplikasi angka gaji terlebih dahulu secara internal, baru kemudian menghitung jumlah sisanya. Kita juga kembali menggunakan .alias("Distinct Salary") untuk memberi nama kolom hasil agar lebih deskriptif dan mudah dipahami.

Output:

```
+---------------+
|Distinct Salary|
+---------------+
|              5|
+---------------+
```

Angka 5 muncul sebagai hasil. Ini berarti meskipun ada 10 karyawan (seperti yang dihitung fungsi count sebelumnya), sebenarnya hanya ada 5 tingkatan gaji yang berbeda di perusahaan tersebut. Sisanya adalah karyawan dengan gaji yang sama.

4. Mencari nilai maksimum (*Finding maximums - max*)

Untuk mengetahui batas atas dari data, misalnya siapa yang memiliki gaji tertinggi, kita menggunakan fungsi max.

Implementasi Source Code:

```
salary_data.agg({'Salary':'max'}).show()
```

Kembali menggunakan sintaks .agg() dengan kamus, kode ini menginstruksikan Spark untuk memindai kolom 'Salary' dan mencari nilai terbesar ('max'). Ini sangat efisien karena Spark tidak perlu mengurutkan seluruh data, cukup melacak nilai tertinggi saat memindai.

Output:

```
+-----------+
|max(Salary)|
+-----------+
|       4500|
+-----------+
```

Nilai 4500 ditampilkan, yang merupakan angka gaji tertinggi dalam dataset tersebut.

5. Penjumlahan (*Sum*)

Untuk mengetahui total keseluruhan, misalnya berapa total biaya yang harus dikeluarkan perusahaan untuk membayar gaji seluruh karyawan, kita menggunakan fungsi sum.

Implementasi Source Code:

```
salary_data.agg({'Salary':'sum'}).show()
```

Dengan pola yang sama menggunakan .agg({'Salary': 'sum'}), kode ini menjumlahkan seluruh nilai numerik yang ada pada kolom gaji.

Output:

```
+-----------+
|sum(Salary)|
+-----------+
|      35200|
+-----------+
```

Hasil 35200 merepresentasikan total akumulasi gaji semua karyawan.

Mengurutkan data dengan OrderBy (*Sort data with OrderBy*)

Pengurutan data sangat vital untuk penyajian laporan, misalnya menampilkan daftar gaji dari yang terkecil ke terbesar atau sebaliknya. Di PySpark, fungsi utama untuk ini adalah orderBy.

1. Pengurutan Menaik (*Ascending*)

Implementasi Source Code:

```
salary_data.orderBy("Salary").show()
```

Secara default (bawaan), jika kita memanggil orderBy("Salary") tanpa instruksi tambahan, Spark akan mengurutkan data secara ascending (menaik), yaitu dari nilai terkecil ke nilai terbesar.

Output:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|   Kiran|     Sales|  2200|
|    John|     Sales|  3000|
|    Kate|   Finance|  3000|
|  Martin|      NULL|  3500|
|   Kelly|   Finance|  3500|
|    John| Field-eng|  3500|
|   Maria|   Finance|  3500|
|  Robert|      NULL|  4000|
| Michael| Field-eng|  4500|
| Michael| Field-eng|  4500|
+--------+----------+------+
```

Tabel output menunjukkan baris dengan gaji 2200 berada di posisi paling atas, diikuti oleh 3000, dan seterusnya hingga 4500. Ini memudahkan kita melihat siapa saja karyawan dengan gaji level awal.

2. Pengurutan Menurun (*Descending*)

Untuk membalikkan urutan, misalnya menampilkan gaji tertinggi di paling atas, kita perlu menambahkan fungsi desc() pada kolom target.

Implementasi Source Code:

```
salary_data.orderBy(salary_data["Salary"].desc()).show()
```

Di sini sintaksnya sedikit lebih spesifik. Kita tidak bisa hanya mengirim string nama kolom. Kita harus mengakses objek kolomnya salary_data["Salary"] lalu memanggil metode .desc() padanya. Hasil objek kolom yang sudah dimodifikasi inilah yang dikirim ke dalam orderBy. Ini memberi tahu Spark untuk membalik logika pengurutan.

Output:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
| Michael| Field-eng|  4500|
| Michael| Field-eng|  4500|
|  Robert|      NULL|  4000|
|    John| Field-eng|  3500|
|  Martin|      NULL|  3500|
|   Maria|   Finance|  3500|
|   Kelly|   Finance|  3500|
|    Kate|   Finance|  3000|
|    John|     Sales|  3000|
|   Kiran|     Sales|  2200|
+--------+----------+------+
```

Sekarang, karyawan dengan gaji 4500 (Michael) berada di baris paling atas, sedangkan gaji terendah berada di paling bawah. Ini membuktikan fungsi pengurutan menurun bekerja dengan baik.

Ringkasan (*Summary*)

Dalam bab ini, kita telah mempelajari secara komprehensif cara memanipulasi data menggunakan Spark DataFrames. Kita memulai dengan memahami API Spark DataFrame dan berbagai tipe data yang didukung. Selanjutnya, kita mempraktikkan cara membuat DataFrame dari berbagai sumber dan melihat isinya menggunakan berbagai teknik inspeksi. Terakhir, kita mendalami teknik manipulasi data yang krusial seperti penyaringan, pembersihan data (null & duplikat), serta fungsi agregasi dan pengurutan data. Bab selanjutnya akan membahas operasi tingkat lanjut dalam manipulasi data Spark.

Chapter 5

Operasi Lanjutan dan Optimasi di Spark (Advanced Operations and Optimizations in Spark)

Bab ini mencakup serangkaian topik komprehensif yang dimulai dengan eksplorasi berbagai opsi untuk pengelompokan data dalam Spark DataFrames. Pembahasan akan mendalam mengenai berbagai jenis operasi penggabungan (*joins*) di Spark, termasuk *inner join*, *left join*, *right join*, *outer join*, *cross join*, *broadcast join*, dan *shuffle join*, di mana masing-masing memiliki implikasi dan kasus penggunaan yang unik. Perhatian khusus akan diberikan pada *shuffle* dan *broadcast joins*, dengan fokus spesifik pada mekanisme *broadcast hash joins* dan *shuffle sort-merge joins*, serta strategi optimasinya. Selain itu, bab ini juga membahas penggunaan Spark SQL untuk berbagai operasi serta mekanisme membaca dan menulis data ke disk menggunakan format yang berbeda seperti CSV, Parquet, dan lainnya.

Fokus utama selanjutnya beralih pada komponen inti mesin eksekusi kueri Spark, yaitu pengoptimal *Catalyst*. Komponen ini menggunakan optimasi berbasis aturan (*rule-based*) dan berbasis biaya (*cost-based*) untuk meningkatkan kinerja kueri secara signifikan. Bab ini juga akan membedah perbedaan antara transformasi *narrow* dan *wide*, serta panduan mengenai kapan harus menggunakan setiap jenis transformasi demi mencapai paralelisme optimal dan efisiensi sumber daya. Di sisi lain, teknik persistensi data dan *caching* akan dieksplorasi sebagai metode untuk mengurangi komputasi ulang dan mempercepat pemrosesan data, lengkap dengan praktik terbaik untuk manajemen memori yang efisien.

Untuk menyeimbangkan beban kerja dan mengoptimalkan distribusi data, bab ini menjelaskan operasi pemartisian data melalui teknik *repartition* dan *coalesce*. Fleksibilitas pemrosesan data diperluas melalui pembahasan mengenai *User-Defined Functions* (UDFs) dan fungsi kustom yang memungkinkan implementasi logika pemrosesan data khusus secara efektif. Akhirnya, pembahasan bermuara pada pelaksanaan optimasi tingkat lanjut menggunakan *Catalyst optimizer* dan *Adaptive Query Execution* (AQE), serta teknik optimasi berbasis data beserta manfaatnya. Setiap bagian dirancang untuk memberikan wawasan mendalam, contoh praktis, dan praktik terbaik guna memastikan kesiapan dalam menangani tantangan pemrosesan data yang kompleks di Apache Spark.

Pengelompokan data di Spark dan berbagai Spark joins (Grouping data in Spark and different Spark joins)

Salah satu teknik manipulasi data yang paling fundamental adalah pengelompokan (*grouping*) dan penggabungan (*joining*) data. Ketika melakukan eksplorasi data, pengelompokan data berdasarkan kriteria tertentu menjadi langkah esensial dalam analisis untuk memecah data menjadi bagian-bagian yang bermakna.

Menggunakan groupBy dalam DataFrame (Using groupBy in a DataFrame)

Spark memungkinkan pengelompokan data dalam DataFrame berdasarkan kriteria yang berbeda, misalnya berdasarkan kolom tertentu. Selain itu, pengguna dapat menerapkan berbagai agregasi, seperti penjumlahan (*sum*) atau rata-rata (*average*), pada data yang dikelompokkan ini untuk mendapatkan pandangan holistik. Operasi groupBy di Spark memiliki kemiripan konseptual dengan klausa GROUP BY di SQL, di mana operasi dilakukan secara berkelompok pada dataset.

Implementasi Source Code:

```
salary_data.groupby('Department')
```

Pada baris kode di atas, instruksi yang diberikan kepada Spark adalah untuk mengelompokkan data yang ada di dalam DataFrame salary_data dengan menggunakan kolom 'Department' sebagai kuncinya. Perlu dipahami secara mendalam bahwa eksekusi baris ini belum menghasilkan kalkulasi numerik apa pun atau menampilkan data tabular. Spark bekerja dengan prinsip evaluasi malas (lazy evaluation), sehingga perintah ini hanya mengubah DataFrame menjadi sebuah objek perantara yang telah terpartisi secara logis berdasarkan departemen, menunggu instruksi agregasi selanjutnya untuk dieksekusi.

Output:

```
GroupedData[grouping expressions: [Department], value: [Employee: string,
Department: string ... 1 more field], type: GroupBy]
```

Output yang dihasilkan menegaskan bahwa operasi tersebut tidak mengembalikan DataFrame standar, melainkan sebuah objek pyspark.sql.group.GroupedData yang tersimpan di memori. Objek ini merupakan representasi internal Spark untuk data yang telah dikelompokkan namun belum dihitung. Keberadaan objek ini menunjukkan bahwa Spark telah siap menerima fungsi agregat (seperti count, sum, avg) untuk diterapkan pada kelompok-kelompok yang telah dibentuk tersebut6.

Implementasi Source Code:

```
salary_data.groupby('Department').avg().show()
```

Kode ini adalah kelanjutan logis dari objek grup yang telah dibahas sebelumnya. Di sini, fungsi .avg() dipanggil langsung pada hasil pengelompokan departemen. Fungsi ini memerintahkan Spark untuk menghitung nilai rata-rata (mean) dari semua kolom numerik yang tersedia di dalam setiap kelompok departemen. Setelah kalkulasi rata-rata didefinisikan, metode .show() dipanggil sebagai "aksi" yang memicu eksekusi nyata (kalkulasi dan pengambilan data) untuk menampilkan hasilnya ke layar konsol.

Output:

```
+----------+------------------+
|Department|       avg(Salary)|
+----------+------------------+
| Field-eng| 4166.666666666667|
|      NULL|            3750.0|
|   Finance|3333.3333333333335|
|     Sales|            2600.0|
+----------+------------------+
```

Tabel hasil memperlihatkan kolom departemen bersanding dengan kolom baru bernama avg(Salary). Analisis mendalam pada output ini menunjukkan bahwa Spark secara otomatis menangani nilai null sebagai sebuah entitas grup tersendiri, yang terlihat pada baris pertama. Selain itu, presisi angka di belakang koma pada departemen "Field-eng" dan "Finance" mengindikasikan bahwa hasil agregasi rata-rata secara otomatis dikonversi menjadi tipe data floating-point (double) untuk menjaga keakuratan matematis, meskipun data gaji aslinya mungkin berupa bilangan bulat.

Pernyataan groupBy yang kompleks (A complex groupBy statement)

Operasi groupBy tidak terbatas pada agregasi sederhana. Ia dapat digunakan untuk operasi data yang kompleks, seperti melakukan beberapa transformasi dan agregasi sekaligus dalam satu pernyataan tunggal (*single statement*).

Implementasi Source Code:

```
from pyspark.sql.functions import col, round

salary_data.groupBy('Department')\
  .sum('Salary')\
  .withColumn('sum(Salary)',round(col('sum(Salary)'), 2))\
  .withColumnRenamed('sum(Salary)', 'Salary')\
  .orderBy('Department')\
  .show()
```

Blok kode ini mendemonstrasikan teknik method chaining yang elegan di Spark untuk melakukan transformasi data end-to-end. Pertama, data dikelompokkan per departemen, kemudian fungsi .sum('Salary') menghitung total gaji. Segera setelah itu, fungsi .withColumn digunakan untuk memodifikasi kolom hasil penjumlahan tersebut dengan menerapkan fungsi round agar nilainya dibulatkan menjadi dua desimal. Tidak berhenti di situ, kolom tersebut kemudian diganti namanya kembali menjadi 'Salary' menggunakan .withColumnRenamed agar lebih mudah dibaca. Terakhir, data diurutkan secara alfabetis berdasarkan nama departemen sebelum ditampilkan. Seluruh proses ini terjadi dalam satu aliran eksekusi yang efisien.

Output:

```
+----------+------+
|Department|Salary|
+----------+------+
|      NULL|  7500|
| Field-eng| 12500|
|   Finance| 10000|
|     Sales|  5200|
+----------+------+
```

Output yang dihasilkan jauh lebih bersih dan terstruktur dibandingkan contoh sebelumnya. Kolom 'Salary' kini merepresentasikan total gaji (bukan rata-rata) yang telah dibulatkan, memberikan format pelaporan keuangan yang lebih standar. Pengurutan data juga terlihat jelas, dimulai dari nilai null (yang secara default dianggap nilai terendah dalam pengurutan ascending Spark), diikuti oleh departemen lain secara alfabetis.

Transformasi ini membuktikan bahwa manipulasi skema dan nilai dapat dilakukan secara instan setelah agregasi.

Menggabungkan DataFrames di Spark (Joining DataFrames in Spark)

Operasi penggabungan (*join*) adalah komponen fundamental dalam pemrosesan data dan merupakan fitur inti dari Apache Spark untuk mengombinasikan data dari dua atau lebih DataFrames. Operasi ini esensial untuk tugas-tugas seperti menggabungkan dataset yang terpisah, mengagregasi informasi, dan melakukan operasi relasional.

Implementasi Source Code:

```
salary_data_with_id = [(1, "John", "Field-eng", 3500), \
    (2, "Robert", "Sales", 4000), \
    (3, "Maria", "Finance", 3500), \
    (4, "Michael", "Sales", 3000), \
    (5, "Kelly", "Finance", 3500), \
    (6, "Kate", "Finance", 3000), \
    (7, "Martin", "Finance", 3500), \
    (8, "Kiran", "Sales", 2200), \
  ]
columns= ["ID", "Employee", "Department", "Salary"]
salary_data_with_id = spark.createDataFrame(data = salary_data_with_id, schema =
columns)
salary_data_with_id.show()
```

Langkah pertama dalam simulasi join adalah pembuatan dataset kiri (left dataset). Kode ini menginisialisasi sebuah list tuple yang berisi data karyawan lengkap dengan ID, Nama, Departemen, dan Gaji. Fungsi spark.createDataFrame kemudian mengubah struktur data Python mentah ini menjadi DataFrame terdistribusi Spark dengan skema kolom yang didefinisikan secara eksplisit. Poin penting yang harus dicatat untuk analisis selanjutnya adalah rentang ID dalam dataset ini, yang berjalan dari angka 1 hingga 8.

Output:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  1|    John| Field-eng|  3500|
|  2|  Robert|     Sales|  4000|
|  3|   Maria|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  5|   Kelly|   Finance|  3500|
|  6|    Kate|   Finance|  3000|
|  7|  Martin|   Finance|  3500|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

Tabel yang ditampilkan memverifikasi bahwa data telah berhasil dimuat ke dalam DataFrame bernama salary_data_with_id. Kita dapat melihat dengan jelas keberadaan 8 baris data, termasuk karyawan bernama Martin (ID 7) dan Kiran (ID 8). Keberadaan dua ID terakhir ini sangat krusial karena akan menjadi objek pengujian utama saat kita melakukan penggabungan dengan dataset kedua yang mungkin tidak lengkap.

Implementasi Source Code:

```
employee_data = [(1, "NY", "M"), \
    (2, "NC", "M"), \
    (3, "NY", "F"), \
    (4, "TX", "M"), \
    (5, "NY", "F"), \
    (6, "AZ", "F") \
  ]
columns= ["ID", "State", "Gender"]
employee_data = spark.createDataFrame(data = employee_data, schema = columns)
employee_data.show()
```

Blok kode ini bertujuan membuat dataset kanan (right dataset) bernama employee_data. Struktur datanya fokus pada atribut demografis karyawan, yaitu Negara Bagian (State) dan Jenis Kelamin (Gender), yang dikaitkan dengan ID. Analisis pada data mentah yang dimasukkan mengungkapkan perbedaan signifikan dibandingkan dataset sebelumnya: dataset ini hanya memuat data untuk ID 1 hingga 6. Ketidakhadiran ID 7 dan 8 di sini sengaja dirancang untuk mensimulasikan skenario data dunia nyata yang sering kali tidak konsisten atau parsial antar tabel.

Output:

```
+---+-----+------+
| ID|State|Gender|
+---+-----+------+
|  1|   NY|     M|
|  2|   NC|     M|
|  3|   NY|     F|
|  4|   TX|     M|
|  5|   NY|     F|
|  6|   AZ|     F|
+---+-----+------+
```

Tampilan DataFrame employee_data mengonfirmasi bahwa tabel ini hanya memiliki 6 baris. Ini menciptakan kondisi ideal untuk mempelajari perilaku join. Kita sekarang memiliki satu tabel dengan 8 baris (Gaji) dan satu tabel dengan 6 baris (Demografi). Bagaimana Spark menangani selisih 2 baris ini akan ditentukan oleh jenis algoritma join yang dipilih.

Inner joins (Inner joins)

Inner join adalah metode penggabungan yang digunakan ketika kita hanya ingin mempertahankan baris data yang memiliki nilai kunci yang sama (cocok) di kedua DataFrame. Nilai apa pun yang tidak memiliki

pasangan di salah satu DataFrame tidak akan disertakan dalam hasil akhir. Secara default, jika jenis join tidak ditentukan, Spark akan menggunakan *inner join*.

Implementasi Source Code:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"inner").show()
```

Dalam baris kode ini, operasi .join() dieksekusi dengan tiga parameter kunci yang mendefinisikan logika penggabungan. Parameter pertama adalah target join (employee_data). Parameter kedua adalah kondisi boolean (salary_data_with_id.ID == employee_data.ID) yang menetapkan bahwa penggabungan harus didasarkan pada kesamaan nilai kolom ID. Parameter ketiga, "inner", secara eksplisit menginstruksikan Spark untuk menerapkan logika irisan himpunan (intersection). Artinya, Spark hanya akan mengambil baris di mana ID tersebut eksis secara bersamaan di tabel gaji DAN tabel karyawan.

Output:

```
+---+--------+----------+------+---+-----+------+
| ID|Employee|Department|Salary| ID|State|Gender|
+---+--------+----------+------+---+-----+------+
|  1|    John| Field-eng|  3500|  1|   NY|     M|
|  2|  Robert|     Sales|  4000|  2|   NC|     M|
|  3|   Maria|   Finance|  3500|  3|   NY|     F|
|  4| Michael|     Sales|  3000|  4|   TX|     M|
|  5|   Kelly|   Finance|  3500|  5|   NY|     F|
|  6|    Kate|   Finance|  3000|  6|   AZ|     F|
+---+--------+----------+------+---+-----+------+
```

Hasil keluaran memperlihatkan sebuah tabel gabungan yang kolom-kolomnya berasal dari kedua sumber data. Temuan analitis yang paling penting di sini adalah jumlah baris yang berkurang menjadi 6 baris saja. ID 7 dan 8 (Martin dan Kiran) hilang dari hasil akhir. Hal ini terjadi karena tabel employee_data tidak memiliki entri untuk ID 7 dan 8. Sesuai prinsip inner join, ketidakcocokan ini menyebabkan data dari salary_data_with_id (yang memuat ID 7 dan 8) ikut terbuang karena tidak menemukan pasangannya.

Outer joins (Outer joins)

Selanjutnya kita membahas *Outer join*, atau dikenal sebagai *full outer join*. Jenis join ini mengembalikan seluruh baris dari kedua DataFrame, dengan mengisi nilai yang hilang (*missing values*) menggunakan null. Metode ini digunakan ketika kita ingin mempertahankan semua data dari kedua sisi, terlepas dari apakah ada kecocokan di sisi lainnya.

Implementasi Source Code:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"outer").show()
```

Perubahan fundamental pada kode ini terletak pada parameter ketiga yang diganti menjadi "outer". Instruksi ini mengubah drastis cara Spark memproses data. Alih-alih membuang data yang tidak cocok, Spark melakukan operasi gabungan himpunan (union) pada kunci ID. Ia akan mengambil semua ID dari tabel kiri dan semua ID dari tabel kanan. Jika ada ID yang cocok, baris digabungkan. Jika ID hanya ada di satu sisi, baris tetap dibuat tetapi kolom dari sisi yang kosong akan diisi dengan nilai null.

Output:

```
+---+--------+----------+------+----+-----+------+
| ID|Employee|Department|Salary|  ID|State|Gender|
+---+--------+----------+------+----+-----+------+
|  1|    John| Field-eng|  3500|   1|   NY|     M|
|  2|  Robert|     Sales|  4000|   2|   NC|     M|
|  3|   Maria|   Finance|  3500|   3|   NY|     F|
|  4| Michael|     Sales|  3000|   4|   TX|     M|
|  5|   Kelly|   Finance|  3500|   5|   NY|     F|
|  6|    Kate|   Finance|  3000|   6|   AZ|     F|
|  7|  Martin|   Finance|  3500|NULL| NULL|  NULL|
|  8|   Kiran|     Sales|  2200|NULL| NULL|  NULL|
+---+--------+----------+------+----+-----+------+
```

Tabel hasil kini kembali memuat ID 7 dan 8 yang sebelumnya hilang. Namun, analisis mendalam pada kolom State dan Gender (kolom-kolom yang berasal dari employee_data) menunjukkan nilai null untuk baris ID 7 dan 8. Ini secara visual mengonfirmasi bahwa Martin dan Kiran ada di data penggajian, tetapi sistem tidak memiliki catatan lokasi atau gender mereka. Penggunaan outer join sangat berguna untuk mendeteksi anomali integritas data seperti ini tanpa kehilangan informasi.

Left joins (Left joins)

Left join mengembalikan semua baris dari DataFrame kiri dan baris yang cocok dari DataFrame kanan. Jika tidak ada kecocokan di DataFrame kanan, hasilnya akan berisi nilai null.

Implementasi Source Code:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"left").show()
```

Dalam skenario ini, salary_data_with_id diposisikan sebagai tabel kiri (sebelum kata .join). Dengan parameter "left", kita menetapkan aturan bahwa tabel Gaji adalah sumber kebenaran utama. Semua data karyawan yang ada di daftar gaji wajib ditampilkan, tidak peduli apakah data demografisnya tersedia atau tidak. Sebaliknya, jika ada data demografis yang tidak memiliki catatan gaji (meskipun dalam contoh ini tidak ada), data tersebut akan diabaikan.

Output:

```
+---+--------+----------+------+----+-----+------+
| ID|Employee|Department|Salary|  ID|State|Gender|
+---+--------+----------+------+----+-----+------+
|  1|    John| Field-eng|  3500|   1|   NY|     M|
|  2|  Robert|     Sales|  4000|   2|   NC|     M|
|  3|   Maria|   Finance|  3500|   3|   NY|     F|
|  4| Michael|     Sales|  3000|   4|   TX|     M|
|  5|   Kelly|   Finance|  3500|   5|   NY|     F|
|  6|    Kate|   Finance|  3000|   6|   AZ|     F|
|  7|  Martin|   Finance|  3500|NULL| NULL|  NULL|
|  8|   Kiran|     Sales|  2200|NULL| NULL|  NULL|
+---+--------+----------+------+----+-----+------+
```

Output ini terlihat identik dengan outer join dalam konteks dataset spesifik ini, karena semua baris tambahan memang berada di sisi kiri. Kita melihat ID 7 dan 8 tetap dipertahankan dengan nilai null di kolom kanan. Ini membuktikan bahwa prioritas tabel kiri terpenuhi: integritas daftar gaji dijaga utuh, sementara data demografis hanya bersifat pelengkap (komplementer).

Right joins (Right joins)

Right join mirip dengan *left join*, tetapi kebalikannya: ia mengembalikan semua baris dari DataFrame kanan dan baris yang cocok dari DataFrame kiri. Baris yang tidak cocok dari DataFrame kiri akan berisi nilai null.

Implementasi Source Code:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"right").show()
```

Meskipun urutan penulisan variabel tidak berubah, penggantian parameter menjadi "right" membalikkan logika prioritas data. Sekarang, employee_data menjadi tabel utama yang harus dipertahankan integritasnya. Spark akan memindai semua entri di tabel demografis dan mencoba mencari pasangannya di tabel gaji. Data gaji yang tidak memiliki referensi di tabel demografis dianggap tidak relevan dalam konteks operasi ini 27.

Output:

```
+---+--------+----------+------+---+-----+------+
| ID|Employee|Department|Salary| ID|State|Gender|
+---+--------+----------+------+---+-----+------+
|  1|    John| Field-eng|  3500|  1|   NY|     M|
|  2|  Robert|     Sales|  4000|  2|   NC|     M|
|  3|   Maria|   Finance|  3500|  3|   NY|     F|
|  4| Michael|     Sales|  3000|  4|   TX|     M|
|  5|   Kelly|   Finance|  3500|  5|   NY|     F|
|  6|    Kate|   Finance|  3000|  6|   AZ|     F|
+---+--------+----------+------+---+-----+------+
```

Hasilnya menunjukkan pengurangan data kembali ke 6 baris, sama seperti inner join, namun dengan alasan logis yang berbeda. ID 7 dan 8 hilang bukan karena mereka tidak memiliki pasangan, tetapi karena mereka bukan bagian dari tabel kanan. Karena tabel kanan (employee_data) hanya memiliki ID 1-6, maka hanya ID itulah yang ditampilkan dalam hasil akhir. Data ID 7 dan 8 di tabel kiri diabaikan karena tidak diminta oleh sisi kanan.

Cross joins (Cross joins)

Jenis gabungan berikutnya yang dibahas adalah *cross join*, yang juga dikenal sebagai *Cartesian join*. Operasi ini menggabungkan setiap baris dari DataFrame kiri dengan setiap baris dari DataFrame kanan. Hasilnya adalah sebuah produk Cartesian yang besar, di mana jumlah baris hasil adalah perkalian dari jumlah baris kedua DataFrame asal.

Modul ini memberikan peringatan khusus bahwa *cross join* harus digunakan dengan sangat hati-hati karena potensinya untuk menghasilkan dataset yang sangat masif yang dapat membebani sumber daya sistem. Penggunaan utamanya biasanya terbatas pada skenario di mana pengguna ingin mengeksplorasi semua kemungkinan kombinasi data, seperti saat menghasilkan data pengujian (*test data generation*).

Union (Union)

Operasi Union digunakan untuk menggabungkan dua DataFrame yang memiliki skema serupa. Berbeda dengan *join* yang menggabungkan kolom secara horizontal, *union* menambahkan baris secara vertikal.

Untuk mengilustrasikan hal ini, langkah pertama adalah membuat DataFrame baru yang memiliki struktur kolom yang sama persis dengan salary_data_with_id.

Implementasi Source Code:

```
salary_data_with_id_2 = [(1, "John", "Field-eng", 3500), \
    (2, "Robert", "Sales", 4000), \
    (3, "Aliya", "Finance", 3500), \
    (4, "Nate", "Sales", 3000), \
  ]
columns2= ["ID", "Employee", "Department", "Salary"]

salary_data_with_id_2 = spark.createDataFrame(data = salary_data_with_id_2,
schema = columns2)

salary_data_with_id_2.printSchema()
salary_data_with_id_2.show(truncate=False)
```

Pada blok kode di atas, sebuah variabel baru salary_data_with_id_2 diinisialisasi. Dataset ini berisi campuran data: beberapa adalah duplikasi dari data lama (John dan Robert), sementara lainnya adalah data baru (Aliya dan Nate). Fungsi printSchema() dipanggil untuk memverifikasi struktur kolom, yang diikuti dengan .show() untuk memvisualisasikan isinya. Kesamaan skema antara DataFrame baru ini dan DataFrame asli adalah syarat mutlak agar operasi union dapat berhasil.

Output:

```
root
 |-- ID: long (nullable = true)
 |-- Employee: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Aliya   |Finance   |3500  |
|4  |Nate    |Sales     |3000  |
+---+--------+----------+------+
```

Output memvalidasi bahwa skema DataFrame baru ini identik dengan DataFrame sebelumnya: memiliki empat kolom dengan tipe data long dan string yang sesuai. Data menunjukkan empat baris karyawan yang siap untuk digabungkan.

Implementasi Source Code:

```
unionDF = salary_data_with_id.union(salary_data_with_id_2)
unionDF.show(truncate=False)
```

Fungsi .union() dipanggil pada DataFrame asli dengan DataFrame baru sebagai argumennya. Operasi ini menumpuk baris-baris dari salary_data_with_id_2 di bawah baris-baris salary_data_with_id. Penting untuk dicatat bahwa metode union standar di Spark (seperti UNION ALL di SQL) tidak melakukan penghapusan duplikat secara otomatis; ia hanya menggabungkan semua data mentah apa adanya.

Output:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Maria   |Finance   |3500  |
|4  |Michael |Sales     |3000  |
|5  |Kelly   |Finance   |3500  |
|6  |Kate    |Finance   |3000  |
|7  |Martin  |Finance   |3500  |
|8  |Kiran   |Sales     |2200  |
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Aliya   |Finance   |3500  |
|4  |Nate    |Sales     |3000  |
+---+--------+----------+------+
```

Hasil akhir menunjukkan bahwa DataFrame unionDF kini berisi gabungan dari kedua sumber data. Baris-baris baru telah ditambahkan ke hasil akhir. Terlihat bahwa entri untuk "John" dan "Robert" muncul dua kali, mengonfirmasi bahwa duplikasi data dipertahankan dalam operasi ini. Empat baris terakhir merupakan tambahan dari DataFrame kedua.

Membaca dan menulis data (Reading and writing data)

Dalam siklus manipulasi data dengan Spark, kemampuan untuk membaca dan menulis data ke disk (*disk I/O*) adalah hal yang fundamental. Mengingat Spark adalah kerangka kerja dalam memori (*in-memory framework*), operasi persistensi data diperlukan untuk menyimpan hasil pemrosesan secara permanen. Spark mendukung berbagai format data populer seperti CSV, Parquet, ORC, dan Delta.

Membaca dan menulis file CSV (Reading and writing CSV files)

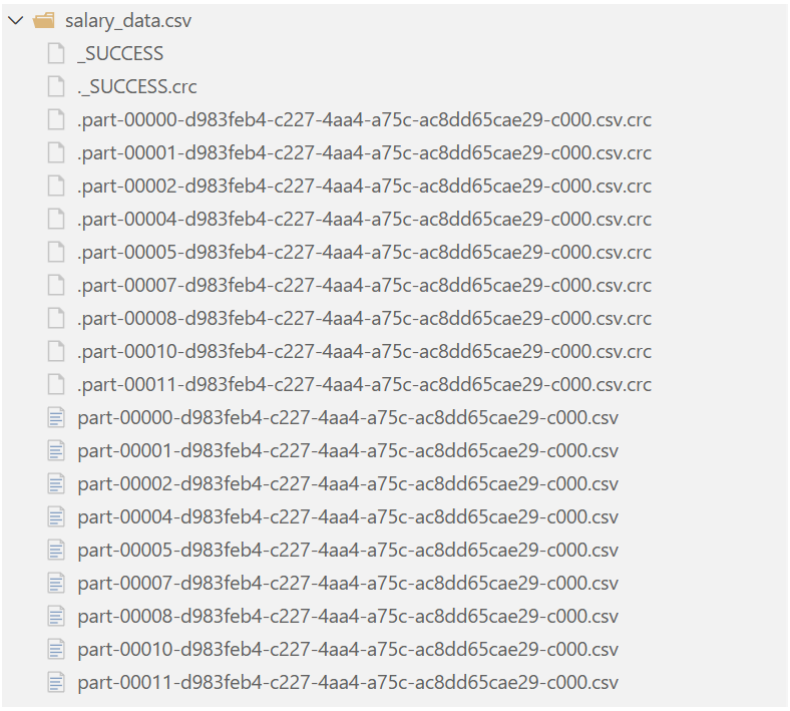
Format *Comma Separated Values* (CSV) sangat populer karena kesederhanaannya, di mana data dipisahkan oleh tanda koma.

Implementasi Source Code:

```
salary_data_with_id.write.csv('salary_data.csv', mode='overwrite', header=True)
spark.read.csv('salary_data.csv', header=True).show()
```

Kode ini mendemonstrasikan proses tulis dan baca. Fungsi dataframe.write.csv() digunakan dengan parameter mode='overwrite' untuk menimpa file jika sudah ada, dan header=True untuk memastikan baris judul kolom disertakan. Eksekusi kode penulisan di atas tidak menghasilkan satu file tunggal, melainkan menciptakan sebuah direktori bernama salary_data.csv yang berisi serangkaian file terstruktur sebagai manifestasi dari arsitektur pemrosesan terdistribusi Spark. Di dalam direktori ini, komponen yang paling dominan adalah sekumpulan file data dengan pola penamaan part-00000 hingga part-00011 dengan ekstensi .csv. Keberadaan banyak file "part" ini menunjukkan bahwa DataFrame salary_data_with_id di dalam memori terpecah menjadi 12 partisi independen, di mana setiap *executor* Spark menuliskan potongan data yang dimilikinya secara paralel ke disk tanpa saling menunggu, menjadikan proses tulis jauh lebih efisien dibandingkan sistem node tunggal.

Selain file data utama, terdapat file bernama _SUCCESS yang berukuran 0 byte. File ini adalah penanda kritis atau "flag" yang dihasilkan oleh Spark hanya ketika seluruh tugas penulisan dari semua partisi telah rampung tanpa kesalahan, yang berfungsi sebagai sinyal bagi sistem hilir (seperti pipeline ETL otomatis) bahwa data di folder ini sudah lengkap dan aman untuk diproses. Terakhir, jika Anda melihat file-file tersembunyi dengan ekstensi .crc (seperti ._SUCCESS.crc atau .part-00000...crc), itu adalah file *Cyclic Redundancy Check*. File-file checksum ini dibuat secara otomatis oleh Spark untuk menjamin integritas data, memungkinkan sistem mendeteksi jika terjadi korupsi pada blok data fisik saat penyimpanan atau transfer berlangsung.



Selanjutnya, spark.read.csv() memuat kembali data tersebut ke dalam memori. Parameter header=True saat membaca sangat penting agar Spark mengenali baris pertama sebagai nama kolom, bukan data.

Output:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  1|    John| Field-eng|  3500|
|  7|  Martin|   Finance|  3500|
|  3|   Maria|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  5|   Kelly|   Finance|  3500|
|  2|  Robert|     Sales|  4000|
|  6|    Kate|   Finance|  3000|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

Data berhasil dimuat kembali dan ditampilkan. Isinya identik dengan DataFrame asli yang ditulis sebelumnya, membuktikan integritas dasar proses I/O CSV.

Selanjutnya, modul membahas cara membaca CSV dengan skema yang didefinisikan secara manual untuk memastikan tipe data yang akurat.

Implementasi Source Code:

```
from pyspark.sql.types import *
filePath = 'salary_data.csv'
```

```
columns= ["ID", "State", "Gender"]
schema = StructType([
      StructField("ID", IntegerType(),True),
  StructField("State",  StringType(),True),
  StructField("Gender",  StringType(),True)
])

read_data =
spark.read.format("csv").option("header","true").schema(schema).load(filePath)
read_data.show()
```

Pendekatan ini lebih robust dibandingkan pembacaan standar. Di sini, objek StructType digunakan untuk mendefinisikan tipe data setiap kolom secara eksplisit (misalnya IntegerType untuk ID). Fungsi .schema(schema) diaplikasikan pada pembaca (spark.read), yang memaksa Spark untuk memvalidasi dan mengonversi data ke tipe yang ditentukan saat proses pemuatan (.load). Ini mencegah masalah di mana semua kolom terbaca sebagai string.

Output:

```
+---+-------+---------+
| ID|  State|   Gender|
+---+-------+---------+
|  1|   John|Field-eng|
|  7| Martin|  Finance|
|  3|  Maria|  Finance|
|  4|Michael|    Sales|
|  5|  Kelly|  Finance|
|  2| Robert|    Sales|
|  6|   Kate|  Finance|
|  8|  Kiran|    Sales|
+---+-------+---------+
```

Meskipun tampilan visual tabel mirip, secara internal DataFrame ini memiliki tipe data yang kuat (strongly typed) sesuai definisi skema. Ini memungkinkan operasi numerik atau filter berbasis tipe dilakukan dengan aman pada langkah selanjutnya.

Membaca dan menulis file Parquet (Reading and writing Parquet files)

Parquet adalah format file berbasis kolom (*columnar*) yang membuat proses membaca dan menulis data menjadi sangat efisien dan ringkas.

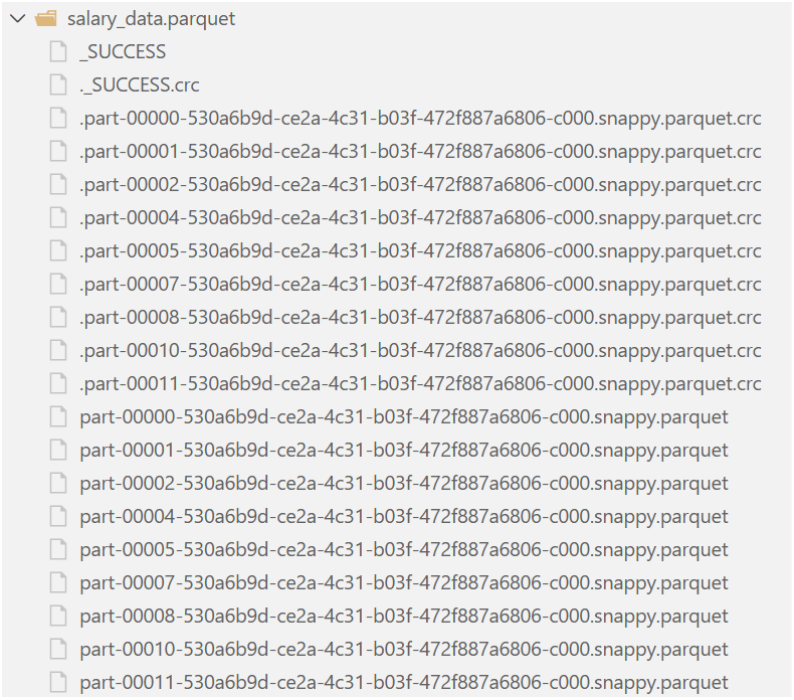
Implementasi Source Code:

```
salary_data_with_id.write.parquet('salary_data.parquet', mode='overwrite')
spark.read.parquet('salary_data.parquet').show()
```

Sintaks yang digunakan hampir sama dengan CSV, namun metodenya diganti menjadi .parquet(). Kelebihan utama Parquet yang terjadi di balik layar adalah penyimpanan skema secara otomatis.

Operasi penulisan ke format Parquet menghasilkan struktur direktori salary_data.parquet yang mengikuti pola partisi yang sama dengan CSV, namun dengan perbedaan fundamental pada format fisik file datanya. File-file data di sini diberi nama dengan pola part-xxxxx...snappy.parquet. Penambahan kata .snappy dalam nama file mengindikasikan bahwa Spark secara otomatis menerapkan algoritma kompresi Snappy pada setiap partisi saat ditulis ke disk. Snappy dipilih sebagai standar karena menawarkan keseimbangan optimal antara rasio kompresi yang layak dan kecepatan dekompresi yang sangat tinggi, yang krusial untuk performa Big Data.

Berbeda dengan CSV yang menyimpan teks mentah per baris, file-file biner di dalam folder ini menyimpan data secara *columnar* (berbasis kolom) beserta metadatanya (seperti tipe data dan skema). Hal ini memungkinkan mesin pembaca nantinya melakukan *projection pushdown*—hanya mengambil file atau kolom tertentu dari disk tanpa membaca keseluruhan dataset—yang secara drastis mengurangi beban I/O. Struktur folder ini juga tetap menyertakan file _SUCCESS dan file validasi .crc untuk memastikan konsistensi transaksional dan keamanan data biner tersebut.



Saat membaca kembali data dengan spark.read.parquet(), pengguna tidak perlu lagi mendefinisikan StructType manual karena metadata tipe data sudah tertanam dalam file Parquet itu sendiri.

Output:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  1|    John| Field-eng|  3500|
|  7|  Martin|   Finance|  3500|
|  3|   Maria|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  5|   Kelly|   Finance|  3500|
|  2|  Robert|     Sales|  4000|
|  6|    Kate|   Finance|  3000|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

Hasil pembacaan menampilkan data yang konsisten. Penggunaan Parquet sangat disarankan dalam ekosistem Big Data karena efisiensi kompresi dan kecepatan bacanya.

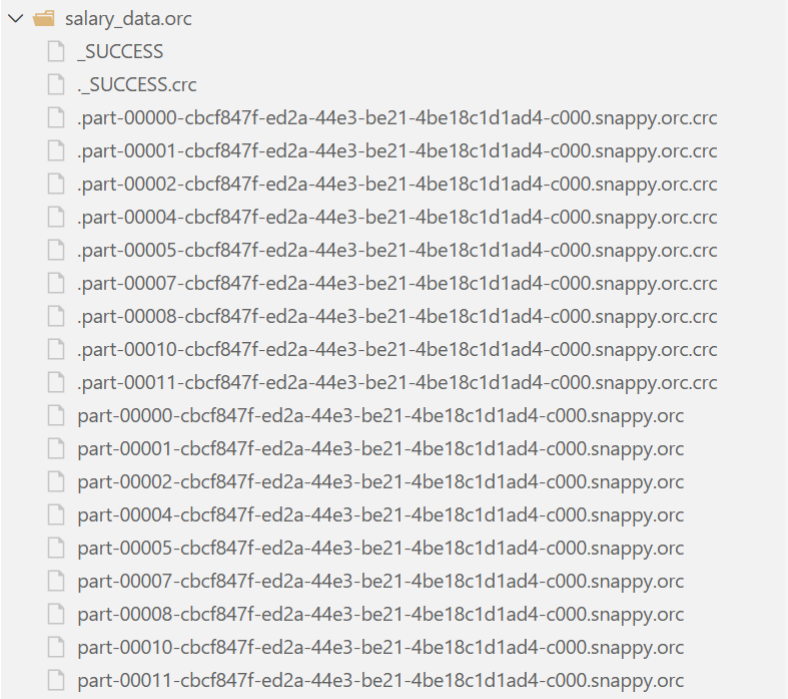
Membaca dan menulis file ORC (Reading and writing ORC files)

Seperti Parquet, ORC (*Optimized Row Columnar*) juga merupakan format file yang kompak dan berbasis kolom.

Implementasi Source Code:

```
salary_data_with_id.write.orc('salary_data.orc', mode='overwrite')
spark.read.orc('salary_data.orc').show()
```

Implementasi kode untuk ORC sangatlah mudah, cukup dengan mengganti pemanggilan fungsi menjadi .orc(). Fleksibilitas ini memungkinkan pengguna untuk memilih format penyimpanan yang paling sesuai dengan arsitektur sistem mereka tanpa mengubah logika kode secara signifikan. Hasil dari operasi ini adalah terbentuknya direktori salary_data.orc yang berisi koleksi file partisi dengan ekstensi .snappy.orc. Konsistensi struktur ini menegaskan bahwa mekanisme penulisan Spark selalu berbasis partisi, di mana jumlah file part- di disk berbanding lurus dengan jumlah partisi RDD/DataFrame di memori saat operasi write dipanggil. Format ORC (*Optimized Row Columnar*) di sini menyimpan data dalam strip-strip yang sangat terkompresi dan terindeks, yang didesain khusus untuk mempercepat operasi pembacaan analitik di ekosistem Hadoop dan Spark. Sama seperti format lainnya, keberadaan file _SUCCESS di akar direktori ini menjamin atomisitas operasi, yang berarti data dalam folder ini dijamin utuh dan tidak merupakan hasil dari proses penulisan yang gagal atau terputus di tengah jalan.



Output:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  3|   Maria|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  1|    John| Field-eng|  3500|
|  7|  Martin|   Finance|  3500|
|  5|   Kelly|   Finance|  3500|
|  2|  Robert|     Sales|  4000|
|  6|    Kate|   Finance|  3000|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

Data yang dibaca kembali dari file ORC identik dengan data aslinya, memverifikasi bahwa proses persistensi data berjalan sukses.

Membaca dan menulis file Delta (Reading and writing Delta files)

Format Delta adalah format terbuka yang lebih optimal daripada Parquet. Format ini menambahkan log transaksional di atas file Parquet, yang membuat operasi baca dan tulis menjadi jauh lebih efisien dan andal.

Implementasi Source Code:

```
salary_data_with_id.write.format("delta").save("/FileStore/tables/salary_data_
with_id", mode='overwrite')
df = spark.read.load("/FileStore/tables/salary_data_with_id")
```

```
df.show()
```

Untuk menggunakan Delta, parameter .format("delta") digunakan secara eksplisit. Metode .save() menentukan lokasi direktori penyimpanan. Saat membaca, fungsi spark.read.load() digunakan dengan menunjuk ke direktori yang sama. Spark secara otomatis menangani metadata log transaksi Delta untuk memastikan versi data yang dibaca adalah yang terbaru dan valid.

Output:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  1|    John| Field-eng|  3500|
|  7|  Martin|   Finance|  3500|
|  3|   Maria|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  5|   Kelly|   Finance|  3500|
|  2|  Robert|     Sales|  4000|
|  6|    Kate|   Finance|  3000|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

Isi file tetap sama dengan DataFrame asal. Keunggulan Delta tidak terlihat dari bentuk data tabularnya, melainkan dari kapabilitas manajemen data di belakang layar seperti dukungan ACID dan penanganan metadata yang lebih baik.

Menggunakan SQL di Spark (Using SQL in Spark)

Spark DataFrames dan Spark SQL dapat digunakan secara bergantian (*interchangeably*). Pengguna dapat memanfaatkan kekuatan kueri SQL standar pada data yang tersimpan dalam DataFrame, memberikan fleksibilitas bahasa pemrograman sesuai preferensi pengguna.

Implementasi Source Code:

```
salary_data_with_id.createOrReplaceTempView("SalaryTable")
spark.sql("SELECT count(*) from SalaryTable").show()
```

Fungsi createOrReplaceTempView("SalaryTable") adalah jembatan yang mengubah DataFrame menjadi tabel virtual bernama "SalaryTable" yang dapat dikueri. Setelah tabel ini terdaftar dalam sesi Spark, metode spark.sql() memungkinkan penulisan instruksi SQL murni (seperti SELECT count(*)...). Spark akan menerjemahkan string SQL tersebut menjadi operasi DataFrame yang dioptimalkan.

Output:

```
+--------+
|count(1)|
+--------+
|       8|
+--------+
```

Hasil eksekusi SQL ditampilkan dalam bentuk DataFrame kolom tunggal. Nilai "8" menunjukkan hasil perhitungan jumlah total baris (count) dari tabel gaji, memvalidasi bahwa integrasi antara Spark SQL dan DataFrame berjalan mulus.

UDFs di Apache Spark (UDFs in Apache Spark)

User-Defined Functions (UDFs) adalah fitur andalan dalam Apache Spark yang memungkinkan pengguna memperluas fungsionalitas standar Spark dengan mendefinisikan fungsi kustom sendiri. Fitur ini sangat esensial ketika pengguna perlu melakukan transformasi atau manipulasi data yang spesifik—seperti algoritma bisnis yang rumit, pembersihan data khusus, atau kalkulasi matematis kompleks—yang tidak didukung secara langsung oleh fungsi bawaan (*built-in functions*) Spark. UDF memberikan fleksibilitas tinggi karena mendukung berbagai bahasa pemrograman termasuk Python, Scala, Java, dan R, serta dapat diaplikasikan baik pada DataFrame maupun RDD.

Membuat UDF di Python (Creating UDFs in Python)

Untuk menggunakan UDF di PySpark, prosesnya melibatkan pendefinisian logika fungsi dalam sintaks Python standar, yang kemudian diregistrasikan ke sistem Spark agar dapat didistribusikan ke seluruh klaster.

Optimasi di Apache Spark (Optimizations in Apache Spark)

Apache Spark dikenal dengan kemampuan komputasi terdistribusinya yang cepat, yang didukung oleh serangkaian teknik optimasi canggih. Optimasi ini bertujuan untuk memaksimalkan kinerja, meningkatkan pemanfaatan sumber daya, dan mengefisiensikan eksekusi pekerjaan (*job execution*). Inti dari kerangka kerja optimasi ini adalah **Catalyst Optimizer**.

Pengoptimal Catalyst (Catalyst optimizer)

Catalyst Optimizer adalah komponen vital dalam mesin eksekusi kueri Spark yang bekerja di belakang layar untuk mengubah logika program pengguna menjadi rencana eksekusi fisik yang paling efisien. Catalyst menggunakan pendekatan optimasi berbasis aturan (*rule-based optimization*) dan berbasis biaya (*cost-based optimization*).

Proses optimasi ini terjadi dalam beberapa tahap:

1. Logical Plan: Merepresentasikan struktur abstrak dari kueri (apa yang ingin dilakukan).
2. Physical Plan: Menentukan bagaimana cara mengeksekusi kueri tersebut secara teknis (misalnya, memilih algoritma join yang tepat atau urutan filter).

Catalyst secara otomatis menerapkan aturan seperti predicate pushdown (melakukan filter sedini mungkin) untuk mengurangi volume data yang diproses.

Implementasi Source Code:

```
# SparkSession setup
  from pyspark.sql import SparkSession
  spark =
  SparkSession.builder.appName("CatalystOptimizerExample").getOrCreate()
  # Load data
  df = spark.read.csv("salary_data.csv", header=True, inferSchema=True)
  # Query with Catalyst Optimizer
  result_df = df.select("employee", "department").filter(df["salary"] > 3500)
  # Explain the optimized query plan
  result_df.explain()
```

Dalam contoh ini, kita melihat bagaimana Catalyst bekerja secara transparan. Pengguna mendefinisikan transformasi data: membaca CSV, memilih kolom tertentu (select), dan menyaring baris berdasarkan gaji (filter). Metode .explain() dipanggil pada DataFrame hasil (result_df) untuk mengintip "otak" Catalyst. Perintah ini tidak menjalankan kueri, melainkan mencetak rencana eksekusi yang telah disusun oleh Catalyst ke layar konsol.

Output:

```
== Physical Plan ==
*(1) Project [employee#1439, department#1440]
+- *(1) Filter (isnotnull(salary#1441) AND (salary#1441 > 3500))
   +- FileScan csv [Employee#1439,Department#1440,Salary#1441] Batched: false, DataFilters: [isnotnull(Salary#1441),
(Salary#1441 > 3500)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/c:/.vscode/denbids/task/final/Databricks-
Certified-Associate-Dev..., PartitionFilters: [], PushedFilters: [IsNotNull(Salary), GreaterThan(Salary,3500)],
ReadSchema: struct<Employee:string,Department:string,Salary:int>
```

Output teks dari .explain() memberikan wawasan mendalam tentang langkah-langkah fisik yang akan diambil Spark. Biasanya, output ini menunjukkan bahwa Spark melakukan optimasi dengan mengubah urutan operasi. Misalnya, meskipun dalam kode kita menulis select (Project) baru kemudian filter, Catalyst mungkin akan memindahkan Filter ke bawah Project atau bahkan menyatukannya dengan FileScan (Predicate Pushdown). Hal ini memastikan bahwa data yang tidak relevan (gaji <= 3500) dibuang seawal mungkin sebelum memori digunakan untuk proyeksi kolom, sehingga menghemat sumber daya komputasi.

Eksekusi Kueri Adaptif (Adaptive Query Execution - AQE)

Sementara Catalyst Optimizer bekerja berdasarkan rencana statis sebelum kueri berjalan, Spark 3.0 memperkenalkan paradigma baru bernama *Adaptive Query Execution* (AQE). AQE adalah pendekatan dinamis di mana Spark menyesuaikan rencana eksekusi *secara real-time* saat kueri sedang berjalan, berdasarkan statistik data aktual yang dikumpulkan dari tahap eksekusi sebelumnya.

Mekanisme kerja AQE meliputi:

1. **Pengumpulan Statistik Runtime:** Mengumpulkan data seperti ukuran partisi dan kemiringan data saat tahap *shuffle* berlangsung.

2. **Optimasi Adaptif:** Menggunakan data tersebut untuk mengubah strategi, seperti mengganti strategi *join* (misalnya dari Sort-Merge ke Broadcast Join jika data ternyata kecil), atau menyeimbangkan partisi yang miring (*data skew*) secara otomatis.

Optimasi berbasis biaya (Cost-based optimization)

Spark juga menggunakan estimasi biaya (*cost estimation*) untuk memilih rencana eksekusi terbaik. Dengan mempertimbangkan faktor-faktor seperti ukuran tabel, estimasi kardinalitas (jumlah nilai unik), dan biaya I/O, Spark dapat memutuskan jalur eksekusi mana yang paling "murah" dalam hal penggunaan sumber daya CPU dan memori. Ini sangat membantu dalam kueri kompleks yang melibatkan banyak tabel dan filter.

Manajemen memori dan penyetelan (Memory management and tuning)

Selain optimasi kueri, efisiensi Spark sangat bergantung pada manajemen memori. Spark membagi memori menjadi dua area utama: *Storage Memory* (untuk *caching* data) dan *Execution Memory* (untuk komputasi sementara seperti *shuffle*, *join*, *sort*). Penyetelan (*tuning*) yang tepat pada konfigurasi memori dan *garbage collection* dapat mencegah masalah seperti tumpahan data ke disk (*spill*) yang sangat memperlambat performa, serta menjamin stabilitas aplikasi yang berjalan lama.

Optimasi berbasis data di Apache Spark (Data-based optimizations in Apache Spark)

Di luar optimasi internal mesin Spark, terdapat tantangan performa yang berasal dari karakteristik fisik data itu sendiri. Bagian ini membahas optimasi yang dikendalikan oleh pengguna (*user-controlled optimizations*) untuk menangani skenario data dunia nyata.

Menangani masalah file kecil (Addressing the small file problem)

Salah satu masalah paling umum dalam sistem terdistribusi adalah "masalah file kecil" (*small file problem*). Masalah ini muncul ketika data tersimpan dalam ribuan atau jutaan file berukuran sangat kecil, alih-alih terkonsolidasi dalam beberapa file besar.

Penyimpanan data dalam banyak file kecil menyebabkan *overhead* metadata yang masif pada NameNode (dalam HDFS) atau layanan penyimpanan objek. Setiap file memerlukan operasi pembukaan, pembacaan metadata, dan penutupan, yang secara kumulatif menghabiskan waktu lebih banyak daripada pemrosesan data aktualnya. Selain itu, file kecil menyebabkan inefisiensi pada partisi Spark, di mana banyak *task* kecil dibuat, menghambat paralelisme yang efektif.

Solusi untuk memitigasi masalah ini meliputi:

- **Penggabungan File (File Concatenation):** Menggabungkan file-file kecil menjadi file besar melalui proses ETL.
- **Coalescing:** Menggunakan fungsi .coalesce() di Spark sebelum menulis data ke disk untuk mengurangi jumlah file output.
- **Format File:** Menggunakan format seperti Parquet atau ORC yang mendukung kompresi dan penyimpanan kolom yang efisien.

Menangani kemiringan data (Tackling data skew)

Kemiringan data (*data skew*) merupakan tantangan signifikan dalam kerangka kerja pemrosesan data terdistribusi seperti Apache Spark. Fenomena ini terjadi ketika distribusi data tidak merata, di mana kunci

atau partisi tertentu menampung jumlah data yang jauh lebih besar dibandingkan yang lain. Ketidakseimbangan ini menyebabkan beban kerja yang timpang pada node pekerja (*worker nodes*).

Dampak utama dari kemiringan data adalah munculnya "stragglers" (tugas yang berjalan sangat lambat). Sementara sebagian besar partisi telah selesai diproses dengan cepat, satu atau dua partisi yang kelebihan beban ("gemuk") masih terus berjalan. Hal ini menyebabkan pemborosan sumber daya karena node lain harus menunggu dalam keadaan *idle*, serta meningkatkan risiko kegagalan memori (*Out of Memory*) pada node yang terbebani.

Strategi Implementasi (Solusi Teknis):

1. **Salting:** Teknik menambahkan nilai acak ("garam") ke kunci partisi untuk memecah data yang terkonsentrasi agar tersebar lebih merata ke berbagai partisi. Ini mencegah terbentuknya *hotspot*.
2. **Custom Partitioning:** Mengimplementasikan logika partisi khusus untuk mendistribusikan ulang data yang miring dengan cara mengelompokkan kunci secara berbeda.
3. **AQE (Adaptive Query Execution):** Memanfaatkan fitur Spark yang secara otomatis mendeteksi kemiringan data saat *runtime* dan memecah tugas-tugas besar menjadi unit yang lebih kecil secara dinamis.

Mengelola tumpahan data (Managing data spills)

Data spill adalah kondisi di mana data yang sedang diproses melebihi kapasitas memori (*RAM*) yang tersedia pada *executor*, sehingga Spark terpaksa menulis data sementara tersebut ke disk. Fenomena ini sangat merugikan performa karena kecepatan disk jauh lebih lambat daripada memori.

Menulis data ke disk menimbulkan *overhead* I/O yang tinggi dan latensi pemrosesan. Selain itu, tumpahan data dapat menyebabkan persaingan sumber daya (*resource contention*) di tingkat penyimpanan, yang mengganggu tugas komputasi lainnya dalam klaster.

Strategi Implementasi (Solusi Teknis):

1. **Manajemen Memori:** Meningkatkan alokasi memori per *executor* atau menyetel konfigurasi memori (seperti spark.memory.fraction) untuk memberikan ruang lebih besar bagi eksekusi.
2. **Repartitioning:** Melakukan partisi ulang data menjadi bagian-bagian yang lebih kecil agar setiap partisi dapat dimuat sepenuhnya dalam memori tanpa tumpah.
3. **Caching:** Menyimpan hasil sementara (*intermediate results*) yang sering diakses ke dalam memori untuk mencegah komputasi ulang yang berpotensi menyebabkan tumpahan berulang.

Mengelola pengocokan data (Managing data shuffle)

Data shuffle adalah operasi memindahkan data antar-node di dalam klaster, yang biasanya dipicu oleh transformasi "lebar" seperti groupBy, join, atau repartition. Meskipun esensial untuk agregasi data, operasi ini menjadi leher botol (*bottleneck*) utama karena melibatkan penggunaan jaringan dan disk secara intensif.

Strategi Implementasi (Solusi Teknis):

1. **Broadcast Joins:** Strategi paling efektif untuk menghilangkan *shuffle* saat menggabungkan tabel besar dengan tabel kecil, dengan cara menyalin tabel kecil ke setiap node.

2. **Filter Dini (Data Filtering/Pruning):** Menerapkan filter sedini mungkin sebelum operasi *shuffle* untuk mengurangi volume data yang harus dipindahkan melalui jaringan.
3. **Tuning Shuffle Partitions:** Menyesuaikan jumlah partisi *shuffle* (default: 200) agar sesuai dengan ukuran data aktual, untuk memastikan beban kerja terdistribusi seimbang.

Shuffle dan broadcast joins (Shuffle and broadcast joins)

Apache Spark menawarkan dua pendekatan fundamental untuk operasi penggabungan (*join*), yang dipilih berdasarkan ukuran data dan kebutuhan optimasi.

Shuffle Joins

Ini adalah metode standar untuk menggabungkan dua dataset besar. Dalam metode ini, data diredistribusi (*shuffled*) di seluruh jaringan agar baris dengan kunci yang sama bertemu di node yang sama. Varian utamanya adalah **Shuffle Sort-Merge Join**, yang mengurutkan kedua dataset berdasarkan kunci join sebelum menggabungkannya. Metode ini stabil untuk data besar namun mahal dari segi komputasi.

Broadcast Joins

Metode ini adalah teknik optimasi untuk penggabungan tabel besar dengan tabel kecil. Tabel kecil dikirim (*broadcast*) ke seluruh *worker nodes*, sehingga setiap node dapat melakukan *join* secara lokal tanpa perpindahan data tabel besar. Varian spesifiknya adalah **Broadcast Hash Join**, di mana tabel kecil disimpan sebagai *hash table* di memori untuk pencarian super cepat.

Transformasi Narrow dan Wide (Narrow and wide transformations)

Pemahaman tentang klasifikasi transformasi sangat krusial untuk memprediksi perilaku performa aplikasi Spark.

Narrow Transformations

Transformasi *narrow* adalah operasi di mana setiap partisi output hanya bergantung pada satu partisi input. Operasi ini bersifat lokal dan **tidak memerlukan shuffle**. Contoh: map(), filter(), union().

Wide Transformations

Transformasi *wide* adalah operasi di mana satu partisi input berkontribusi ke banyak partisi output, atau sebaliknya. Operasi ini **memaksa terjadinya shuffle** data antar-partisi. Contoh: groupByKey(), reduceByKey(), join().

Persisting dan caching di Apache Spark (Persisting and caching in Apache Spark)

Optimasi performa sering kali melibatkan penyimpanan data sementara (*intermediate data*) untuk menghindari komputasi ulang, terutama pada algoritma iteratif atau analisis berulang. Spark menyediakan mekanisme cache dan persist untuk tujuan ini.

Implementasi Source Code:

```
# Cache a DataFrame & Unpersist the cached DataFrame
df.cache()
df.unpersist()
```

```
DataFrame[ID: int, Employee: string, Department: string, Salary: int]
```

Pada baris pertama, df.cache() dipanggil. Perlu diingat bahwa karena sifat lazy evaluation Spark, data tidak langsung disimpan di memori saat baris ini dijalankan. Data baru akan dimuat ke memori (Storage Memory) saat aksi pertama (seperti .count() atau .show()) dipanggil pada DataFrame tersebut. Secara default, cache() menggunakan level penyimpanan MEMORY_AND_DISK (atau MEMORY_ONLY tergantung versi Spark), yang menyimpan data di RAM dan menumpahkannya ke disk jika penuh.

Baris kedua, df.unpersist(), adalah perintah eksplisit untuk menghapus data tersebut dari memori. Ini adalah praktik manajemen memori yang penting: segera bebaskan memori setelah data tidak lagi diperlukan untuk mencegah masalah memori pada proses selanjutnya.

Repartitioning dan coalescing di Apache Spark (Repartitioning and coalescing in Apache Spark)

Kontrol terhadap partisi fisik data memungkinkan pengguna menyeimbangkan paralelisme. Spark menyediakan dua metode utama: repartition dan coalesce.

Repartitioning data

repartition digunakan untuk menambah atau mengurangi jumlah partisi dengan melakukan pengocokan data (*full shuffle*) secara total. Ini berguna untuk menyeimbangkan distribusi data yang miring.

Implementasi Source Code:

```
# Repartition a DataFrame into 8 partitions
df.repartition(8)
```

```
DataFrame[ID: int, Employee: string, Department: string, Salary: int]
```

Perintah df.repartition(8) menginstruksikan Spark untuk mendistribusikan ulang seluruh data ke dalam 8 partisi baru. Operasi ini memicu pertukaran data melalui jaringan (network shuffle) untuk menjamin distribusi yang merata (biasanya secara round-robin atau berdasarkan hash kolom). Metode ini ideal digunakan saat kita ingin meningkatkan tingkat paralelisme (misalnya, dari 2 partisi menjadi 8 agar sesuai dengan jumlah core CPU).

Coalescing data

coalesce digunakan khusus untuk **mengurangi** jumlah partisi dengan cara yang lebih efisien daripada repartition. Metode ini meminimalkan perpindahan data.

Implementasi Source Code:

```
# Coalesce a DataFrame to 4 partitions
df.coalesce(4)
```

```
DataFrame[ID: int, Employee: string, Department: string, Salary: int]
```

Perintah df.coalesce(4) mengurangi jumlah partisi menjadi 4 tanpa melakukan full shuffle. Spark melakukan ini dengan menggabungkan partisi-partisi lokal yang ada di node yang sama atau berdekatan (misalnya, menggabungkan Partisi 1 dan 2 menjadi Partisi A). Karena tidak memindahkan semua data melintasi jaringan, operasi ini jauh lebih cepat dibandingkan repartition dan sangat disarankan untuk digunakan sebelum menulis hasil akhir ke disk (untuk menghindari terbentuknya terlalu banyak file kecil).

Ringkasan (Summary)

Bab ini telah mengupas tuntas kapabilitas pemrosesan data tingkat lanjut di Apache Spark, yang bertujuan untuk memperdalam pemahaman mengenai konsep dan teknik kunci dalam kerangka kerja ini[1]. Eksplorasi materi mencakup seluk-beluk *Catalyst optimizer*, kekuatan berbagai jenis operasi *join*, urgensi strategi persistensi dan *caching* data, signifikansi perbedaan antara transformasi *narrow* dan *wide*, serta peran vital pengaturan partisi data menggunakan metode *repartition* dan *coalesce*[2]. Selain itu, fleksibilitas dan kegunaan *User-Defined Functions* (UDFs) dalam menangani logika kustom juga telah dibahas secara mendalam.

Seiring dengan meningkatnya kompleksitas penggunaan Apache Spark, kapabilitas lanjutan ini menjadi aset berharga untuk optimasi dan kustomisasi alur kerja pemrosesan data. Pemanfaatan potensi *Catalyst optimizer* memungkinkan penyetelan eksekusi kueri demi kinerja yang lebih baik. Pemahaman mendalam mengenai nuansa *Spark joins* memfasilitasi pengambilan keputusan yang tepat terkait jenis penggabungan yang paling efektif untuk setiap kasus penggunaan[6]. Di sisi lain, persistensi dan *caching* data menjadi instrumen yang sangat diperlukan untuk mengurangi komputasi ulang (*recomputation*) dan mempercepat proses algoritma yang bersifat iteratif.

Lebih lanjut, transformasi *narrow* dan *wide* memegang peranan pivotal dalam mencapai tingkat paralelisme yang diinginkan serta efisiensi penggunaan sumber daya aplikasi. Penerapan partisi data yang tepat melalui mekanisme *repartition* dan *coalesce* menjamin beban kerja yang seimbang dan distribusi data yang optimal di seluruh klaster. Fitur UDFs membuka peluang luas untuk implementasi logika pemrosesan data kustom—mulai dari pembersihan data hingga kalkulasi kompleks—namun penggunaannya harus dilakukan secara bijaksana dengan memperhatikan optimasi performa.

Chapter 6

SQL Queries in Spark

Apa itu Spark SQL? (What is Spark SQL?)

Spark SQL didefinisikan sebagai modul yang sangat bertenaga di dalam ekosistem Apache Spark, yang didedikasikan untuk pemrosesan dan analisis data terstruktur secara efisien[1]. Secara mendasar, modul ini menyediakan antarmuka tingkat tinggi (*higher-level interface*) yang membedakannya dari API berbasis RDD tradisional yang lebih rumit. Keunikan utamanya terletak pada penggabungan dua paradigma: pemrosesan relasional (seperti SQL) dan pemrosesan prosedural, yang memungkinkan pengguna untuk menyematkan kueri SQL ke dalam analitik kompleks secara mulus[2]. Dengan memanfaatkan kemampuan komputasi terdistribusi Spark, modul ini menjamin performa tinggi dan skalabilitas, serta mendukung berbagai sumber data seperti Hive, Parquet, dan JDBC.

Keuntungan Spark SQL (Advantages of Spark SQL)

Spark SQL menawarkan serangkaian keunggulan kompetitif yang menjadikannya pilihan utama dalam industri pemrosesan data[4]. Berikut adalah rincian keunggulan tersebut berdasarkan sub-bab materi:

Pemrosesan data terpadu dengan Spark SQL (Unified data processing with Spark SQL)

Fitur ini memungkinkan pengolahan data terstruktur dan tidak terstruktur menggunakan satu mesin tunggal. Implikasi teknisnya sangat signifikan: pengguna dapat menggunakan antarmuka pemrograman yang sama untuk melakukan kueri terhadap format data yang sangat beragam, mulai dari JSON, CSV, hingga Parquet. Fleksibilitas ini memungkinkan peralihan yang mulus antara kueri SQL, transformasi DataFrame, dan algoritma Machine Learning dalam satu aplikasi, yang secara drastis mengurangi kompleksitas pengembangan sistem.

Performa dan skalabilitas (Performance and scalability)

Spark SQL tidak bekerja sendirian; ia memanfaatkan arsitektur terdistribusi Apache Spark untuk menangani dataset berskala besar di seluruh klaster mesin. Kunci kecepatannya terletak pada *Catalyst optimizer*, sebuah teknik optimasi kueri canggih (yang dibahas di Chapter 5), yang mempercepat eksekusi kueri. Selain itu, dukungan terhadap mekanisme partisi data dan *caching* di dalam memori (*in-memory caching*) membuat mesin eksekusi Spark SQL jauh lebih cepat dibandingkan mesin SQL tradisional, karena mampu menyusun rencana eksekusi yang dioptimalkan dan memproses data secara paralel di banyak mesin.

Integrasi mulus dengan infrastruktur yang ada (Seamless integration with existing infrastructure)

Keunggulan operasional Spark SQL terletak pada interoperabilitasnya. Ia tidak menuntut infrastruktur baru, melainkan terintegrasi dengan alat Apache Spark yang sudah ada, termasuk Spark Streaming untuk data waktu nyata dan MLlib untuk pembelajaran mesin. Dukungan terhadap format penyimpanan populer seperti Parquet, Avro, ORC, dan Hive memastikan kompatibilitas yang luas dengan ekosistem data perusahaan.

Kemampuan analitik tingkat lanjut (Advanced analytics capabilities)

Spark SQL melampaui batasan SQL standar dengan menghadirkan fitur analitik modern. Salah satu fitur kuncinya adalah dukungan terhadap fungsi jendela (*window functions*), yang memungkinkan operasi kompleks seperti pemeringkatan (*ranking*), agregasi kumulatif, dan pergeseran jendela (*sliding windows*).

Integrasi dengan pustaka pembelajaran mesin juga memungkinkan alur kerja *data science* dan analitik prediktif berjalan dalam satu lingkungan.

Kemudahan penggunaan (Ease of use)

Hambatan masuk (*barrier to entry*) untuk menggunakan Spark SQL sangat rendah karena menyediakan antarmuka pemrograman yang menggunakan sintaks mirip SQL. Hal ini sangat menguntungkan bagi pengguna yang sudah terbiasa dengan SQL tradisional, karena mereka dapat langsung produktif tanpa perlu mempelajari paradigma baru yang radikal.

Integrasi dengan Apache Spark (Integration with Apache Spark)

Sebagai bagian integral dari kerangka kerja Apache Spark, Spark SQL mewarisi fungsionalitas inti seperti toleransi kesalahan (*fault tolerance*) dan paralelisme data. Kemampuannya membaca data dari HDFS, Amazon S3, hingga database relasional (JDBC) dan Hive metastore menjadikannya hub pusat untuk akses data yang beragam.

Konsep Kunci – DataFrames dan Datasets (Key concepts – DataFrames and datasets)

Spark SQL memperkenalkan dua abstraksi data utama yang menjadi fondasi seluruh operasi:

DataFrames

DataFrames adalah koleksi data terdistribusi yang disusun berdasarkan kolom-kolom bernama, menyerupai tabel dalam database relasional namun dengan optimalisasi di balik layar. Mereka bersifat *immutable* (tidak dapat diubah) dan *lazily evaluated* (eksekusi ditunda hingga aksi dipanggil), yang memungkinkan *Catalyst optimizer* bekerja maksimal. DataFrames dapat dibentuk dari berbagai sumber seperti file terstruktur (CSV, JSON), tabel Hive, atau RDD yang sudah ada.

Datasets

Datasets merupakan perpanjangan dari DataFrames yang menawarkan antarmuka pemrograman berorientasi objek yang aman secara tipe (*type-safe*). Keunggulan utamanya adalah kombinasi antara pengetikan kuat (*strong typing*) ala RDD dengan optimasi performa ala DataFrame. Pengecekan tipe dilakukan saat kompilasi (*compile-time*), meminimalkan kesalahan saat *runtime*.

Memulai dengan Spark SQL (Getting started with Spark SQL)

Langkah awal dalam operasional Spark SQL selalu melibatkan pemuatan data ke dalam DataFrame. Setelah data dimuat, pengguna dapat beralih antara manipulasi prosedural PySpark dan kueri deklaratif SQL untuk melakukan transformasi.

Memuat dan menyimpan data (Loading and saving data)

Bagian ini berfokus pada teknik memuat data dari berbagai sumber dan menyimpannya kembali sebagai tabel. Tujuannya adalah mendemonstrasikan siklus hidup data: dari pemuatan, transformasi, hingga penyimpanan kembali untuk analisis lanjutan.

Mengeksekusi Kueri SQL (Executing SQL queries)

Salah satu fitur terkuat Spark SQL adalah kemampuannya mengeksekusi kueri menggunakan sintaks SQL standar melalui metode spark.sql().

Membuat DataFrame Secara Manual (Creating DataFrame manually)

Sebelum melakukan kueri kompleks, kita perlu memiliki data. Berikut adalah proses pembuatan DataFrame dari data mentah.

Implementasi Source Code:

```
salary_data_with_id = [
    (1, "John", "Field-eng", 3500, 40), \
    (2, "Robert", "Sales", 4000, 38), \
    (3, "Maria", "Finance", 3500, 28), \
    (4, "Michael", "Sales", 3000, 20), \
    (5, "Kelly", "Finance", 3500, 35), \
    (6, "Kate", "Finance", 3000, 45), \
    (7, "Martin", "Finance", 3500, 26), \
    (8, "Kiran", "Sales", 2200, 35), \
  ]
columns= ["ID", "Employee", "Department", "Salary", "Age"]
salary_data_with_id = spark.createDataFrame(data = salary_data_with_id, schema =
columns)
salary_data_with_id.show()
```

Kode ini membangun dataset dari nol. Variabel salary_data_with_id berisi daftar karyawan dengan atribut spesifik. Fungsi spark.createDataFrame bertugas mengonversi struktur data Python (List) menjadi struktur data terdistribusi Spark (DataFrame). Parameter schema sangat krusial di sini untuk memberikan label yang bermakna ("ID", "Employee", dll) pada kolom-kolom data, sehingga memungkinkan pengacuan kolom berdasarkan nama di langkah selanjutnya.

Output:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  1|    John| Field-eng|  3500| 40|
|  2|  Robert|     Sales|  4000| 38|
|  3|   Maria|   Finance|  3500| 28|
|  4| Michael|     Sales|  3000| 20|
|  5|   Kelly|   Finance|  3500| 35|
|  6|    Kate|   Finance|  3000| 45|
|  7|  Martin|   Finance|  3500| 26|
|  8|   Kiran|     Sales|  2200| 35|
+---+--------+----------+------+---+
```

Tabel yang dihasilkan menunjukkan bahwa Spark berhasil memetakan data mentah ke dalam skema yang ditentukan. Kolom Age secara eksplisit ditambahkan dalam dataset ini untuk keperluan pemrosesan selanjutnya yang melibatkan logika umur.

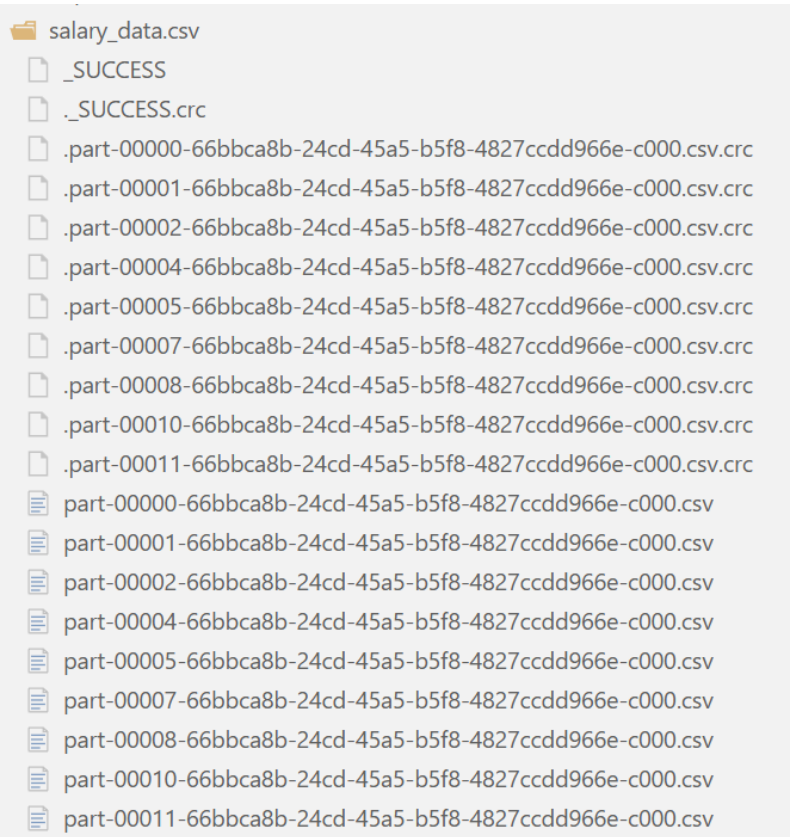
Konteks Penyimpanan CSV (Saving and Reading CSV Context)

Modul ini merujuk kembali ke materi Chapter 4, mengingatkan bahwa DataFrame ini dapat disimpan dan dibaca kembali sebagai file CSV.

Implementasi Source Code:

```
salary_data_with_id.write.format("csv").mode("overwrite").option("header",
"true").save("salary_data.csv")
```

Baris kode ini mendemonstrasikan kapabilitas writer Spark. format("csv") menentukan tipe file output. mode("overwrite") adalah instruksi keselamatan untuk menimpa file jika sudah ada, mencegah error duplikasi. option("header", "true") memastikan baris pertama file CSV berisi nama kolom, yang sangat penting untuk menjaga integritas skema saat file dibaca kembali nanti. Hasil eksekusi kode penyimpanan data menghasilkan struktur output yang unik seperti yang terlihat pada lampiran gambar.



Dalam lingkungan pemrosesan terdistribusi Spark, perintah .save("salary_data.csv") tidak menciptakan satu file teks tunggal, melainkan sebuah **direktori (folder)** dengan nama tersebut. Di dalam direktori ini, data dipecah menjadi beberapa segmen file dengan penamaan part-00000, part-00001, dan seterusnya. Fenomena ini terjadi karena DataFrame salary_data_with_id diproses secara paralel dalam memori yang terpartisi; saat operasi penulisan (*write*) dipicu, setiap partisi tersebut ditulis secara independen dan simultan ke dalam disk. Selain file data utama, terdapat juga file kosong bernama _SUCCESS yang berfungsi sebagai

penanda sistem bahwa seluruh proses penulisan data telah rampung tanpa kesalahan, serta file-file berekstensi .crc yang berisi *checksum* untuk menjamin integritas data saat dibaca kembali di masa depan.

Melakukan Transformasi pada Data (Perform transformations on the loaded data)

Setelah data dimuat, kita melakukan penyaringan (*filtering*) dan menyimpannya sebagai tabel sementara untuk kueri SQL.

Implementasi Source Code:

```
# Perform transformations on the loaded data
processed_data = csv_data.filter(csv_data["Salary"] > 3000)
# Save the processed data as a table
processed_data.createOrReplaceTempView("high_salary_employees")
# Perform SQL queries on the saved table
results = spark.sql("SELECT * FROM high_salary_employees ")
results.show()
```

Langkah pertama menggunakan metode .filter() dari API DataFrame untuk mengambil hanya karyawan dengan gaji di atas 3000. Langkah kedua adalah jembatan penting: createOrReplaceTempView ("high_salary_employees") mendaftarkan DataFrame hasil filter tersebut ke dalam katalog SQL Spark dengan nama alias high_salary_employees. Ini memungkinkan langkah ketiga, yaitu penggunaan sintaks SQL murni (SELECT *) untuk mengakses data yang sudah berada di memori Python tersebut.

Output:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  1|    John| Field-eng|  3500| 40|
|  7|  Martin|   Finance|  3500| 26|
|  3|   Maria|   Finance|  3500| 28|
|  5|   Kelly|   Finance|  3500| 35|
|  2|  Robert|     Sales|  4000| 38|
+---+--------+----------+------+---+
```

Hasil output hanya menampilkan baris data di mana kolom Salary bernilai lebih dari 3000. Karyawan dengan gaji 3000 atau kurang (seperti Michael dan Kate) telah dieliminasi dari view ini, membuktikan bahwa filter logis berfungsi sebelum data didaftarkan sebagai tabel.

Menyimpan data yang diubah sebagai view (Saving transformed data as a view)

Bagian ini memperdalam konsep pembuatan *view*. Penggunaan createOrReplaceTempView() disorot sebagai metode yang lebih aman dibandingkan createTempView(). Alasannya, jika tabel sementara dengan nama yang sama sudah ada, metode pertama akan menggantinya tanpa *error*, sedangkan metode kedua akan menyebabkan TempTableAlreadyExistsException. *View* ini bertindak sebagai representasi terstruktur yang memfasilitasi eksplorasi data menggunakan kekuatan ekspresif SQL.

Memanfaatkan Spark SQL untuk memfilter dan memilih data (Utilizing Spark SQL to filter and select data based on specific criteria)

Di sini, kita beralih sepenuhnya ke penggunaan sintaks SQL untuk melakukan seleksi kolom (*projection*) dan penyaringan baris (*selection*) secara bersamaan.

Implementasi Source Code:

```
# Save the processed data as a view
salary_data_with_id.createOrReplaceTempView("employees")
#Apply filtering on data
filtered_data = spark.sql("SELECT Employee, Department, Salary, Age FROM
employees WHERE age > 30")
# Display the results
filtered_data.show()
```

Kueri SQL ini lebih spesifik. Klausa SELECT Employee, ... memilih kolom tertentu saja (ID dibuang). Klausa WHERE age > 30 menerapkan kondisi logika. Spark memproses string kueri ini, menyusun rencana logis (logical plan), mengoptimalkannya, dan kemudian mengeksekusinya terhadap data employees. Hasilnya disimpan di variabel filtered_data.

Output:

```
+--------+----------+------+---+
|Employee|Department|Salary|Age|
+--------+----------+------+---+
|    John| Field-eng|  3500| 40|
|  Robert|     Sales|  4000| 38|
|   Kelly|   Finance|  3500| 35|
|    Kate|   Finance|  3000| 45|
|   Kiran|     Sales|  2200| 35|
+--------+----------+------+---+
```

Tabel hasil memverifikasi dua hal: kolom "ID" tidak lagi muncul (sesuai seleksi), dan hanya karyawan berusia di atas 30 tahun (John, Robert, Kelly, Kate, Kiran) yang ditampilkan. Data yang tidak memenuhi syarat (seperti Martin, 26 tahun) tidak disertakan 35.

Eksplorasi operasi pengurutan dan agregasi menggunakan Spark SQL (Exploring sorting and aggregation operations using Spark SQL)

Bagian ini mendemonstrasikan kemampuan Spark SQL dalam meringkas data (agregasi) dan mengorganisirnya (pengurutan).

Agregasi (Aggregation)

Implementasi Source Code:

```
# Perform an aggregation to calculate the average salary
average_salary = spark.sql("SELECT AVG(Salary) AS average_salary FROM employees")
# Display the average salary
average_salary.show()
```

Fungsi agregat AVG(Salary) digunakan untuk menghitung nilai rata-rata dari seluruh kolom gaji. AS average_salary adalah alias untuk memberi nama pada kolom hasil agar mudah dibaca.

Output:

```
+--------------+
|average_salary|
+--------------+
|        3275.0|
+--------------+
```

Output berupa nilai tunggal 3275.0, yang merupakan hasil perhitungan matematis dari seluruh baris dalam kolom Salary dibagi jumlah baris.

Pengurutan (Sorting)

Implementasi Source Code:

```
# Sort the data based on the salary column in descending order
sorted_data = spark.sql("SELECT * FROM employees ORDER BY Salary DESC")
# Display the sorted data
sorted_data.show()
```

Klausa ORDER BY Salary DESC memerintahkan Spark untuk menyusun ulang dataset. DESC menandakan urutan menurun (terbesar ke terkecil). Tanpa DESC, default-nya adalah ASC (menaik).

Output:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  2|  Robert|     Sales|  4000| 38|
|  3|   Maria|   Finance|  3500| 28|
|  1|    John| Field-eng|  3500| 40|
|  5|   Kelly|   Finance|  3500| 35|
|  7|  Martin|   Finance|  3500| 26|
|  4| Michael|     Sales|  3000| 20|
|  6|    Kate|   Finance|  3000| 45|
|  8|   Kiran|     Sales|  2200| 35|
+---+--------+----------+------+---+
```

Tabel kini menampilkan Robert di posisi teratas karena ia memiliki gaji tertinggi (4000), diikuti oleh John dan lainnya. Struktur data tetap sama, hanya urutan barisnya yang berubah.

Menggabungkan agregasi (Combining aggregations)

Kekuatan SQL terletak pada kemampuannya menggabungkan berbagai operasi logika dalam satu perintah eksekusi.

Implementasi Source Code:

```
# Sort the data based on the salary column in descending order
filtered_data = spark.sql("SELECT Employee, Department, Salary, Age FROM
employees WHERE age > 30 AND Salary > 3000 ORDER BY Salary DESC")
# Display the results
filtered_data.show()
```

Kueri ini sangat padat. Ia melakukan filter ganda dengan operator AND: usia harus > 30 dan gaji harus > 3000. Hanya baris yang memenuhi kedua syarat tersebut yang lolos. Setelah disaring, hasil tersebut langsung diurutkan berdasarkan gaji tertinggi (ORDER BY Salary DESC). Ini menunjukkan efisiensi Spark dalam menangani kueri majemuk.

Output:

```
+--------+----------+------+---+
|Employee|Department|Salary|Age|
+--------+----------+------+---+
|  Robert|     Sales|  4000| 38|
|   Kelly|   Finance|  3500| 35|
|    John| Field-eng|  3500| 40|
+--------+----------+------+---+
```

Hasilnya sangat spesifik. Hanya 3 karyawan yang tersisa. Karyawan seperti Kiran (35 tahun) hilang karena gajinya (2200) tidak memenuhi syarat > 3000, meskipun usianya memenuhi syarat. Data yang tampil juga sudah terurut rapi.

Pengelompokan dan Agregasi Data (Grouping and aggregating data)

Bagian ini membahas operasi yang lebih kompleks untuk mendapatkan wawasan per kategori (*category-wise insights*).

Mengelompokkan data (Grouping data)

Implementasi Source Code:

```
# Group the data based on the Department column and take average salary for each
grouped_data = spark.sql("SELECT Department, avg(Salary) FROM employees GROUP BY
Department")
```

```
# Display the results
grouped_data.show()
```

Perintah GROUP BY Department memecah data menjadi partisi-partisi berdasarkan nilai unik di kolom Departemen. Fungsi avg(Salary) kemudian diterapkan secara independen pada setiap partisi tersebut. Ini berbeda dengan agregasi sebelumnya yang menghitung rata-rata global 48.

Output:

```
+----------+------------------+
|Department|       avg(Salary)|
+----------+------------------+
| Field-eng|            3500.0|
|     Sales|3066.6666666666665|
|   Finance|            3375.0|
+----------+------------------+
```

Tabel ringkasan ini menunjukkan rata-rata gaji yang spesifik untuk setiap departemen, memberikan wawasan komparatif antar divisi yang tidak terlihat dalam data mentah.

Mengagregasi data (Aggregating data)

Implementasi Source Code:

```
# Perform grouping and multiple aggregations
aggregated_data = spark.sql("SELECT Department, sum(Salary) AS total_salary,
max(Salary) AS max_salary FROM employees GROUP BY Department")

# Display the results
aggregated_data.show()
```

Kode ini mendemonstrasikan bahwa kita bisa melakukan lebih dari satu perhitungan statistik dalam satu perintah GROUP BY. Di sini, kita menghitung total pengeluaran gaji (sum) dan gaji tertinggi (max) sekaligus untuk setiap departemen.

Output:

```
+----------+------------+----------+
|Department|total_salary|max_salary|
+----------+------------+----------+
| Field-eng|        3500|      3500|
|     Sales|        9200|      4000|
|   Finance|       13500|      3500|
+----------+------------+----------+
```

Informasi ini sangat bernilai untuk pelaporan manajerial. Kita bisa melihat beban gaji terbesar ada di Finance (13500), sedangkan gaji perorangan tertinggi ada di Sales (4000).

Operasi Spark SQL Tingkat Lanjut (Advanced Spark SQL operations)

Setelah menguasai dasar-dasar, modul beralih ke kapabilitas lanjut untuk manipulasi data yang kompleks.

Memanfaatkan fungsi jendela (Leveraging window functions)

Fungsi jendela (*Window Functions*) adalah fitur analitik *powerful* yang memungkinkan perhitungan dilakukan pada sekumpulan baris (*window*) yang terkait dengan baris saat ini, tanpa mengompres hasil seperti GROUP BY.

Memahami fungsi jendela (Understanding window functions)

Sintaks umumnya melibatkan definisi partisi (partitionBy), pengurutan (orderBy), dan rentang baris (rowsBetween). Ini memungkinkan operasi seperti "rata-rata bergerak" atau "total kumulatif".

Menghitung jumlah kumulatif menggunakan fungsi jendela (Calculating cumulative sum using window functions)

Implementasi Source Code:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, sum

# Define the window specification
window_spec = Window.partitionBy("Department").orderBy("Age")

# Calculate the cumulative sum using window function
df_with_cumulative_sum = salary_data_with_id.withColumn("cumulative_sum",
sum(col("Salary")).over(window_spec))

# Display the result
df_with_cumulative_sum.show()
```

Kode ini mendefinisikan logika jendela: "Untuk setiap Departemen, urutkan data berdasarkan Umur". Fungsi sum(Salary).over(window_spec) kemudian berjalan di atas logika ini. Artinya, untuk setiap baris karyawan, Spark menjumlahkan gajinya dengan gaji karyawan-karyawan sebelumnya (yang lebih muda) dalam departemen yang sama.

Output:

```
+---+--------+----------+------+---+--------------+
| ID|Employee|Department|Salary|Age|cumulative_sum|
+---+--------+----------+------+---+--------------+
|  1|    John| Field-eng|  3500| 40|          3500|
|  7|  Martin|   Finance|  3500| 26|          3500|
|  3|   Maria|   Finance|  3500| 28|          7000|
|  5|   Kelly|   Finance|  3500| 35|         10500|
|  6|    Kate|   Finance|  3000| 45|         13500|
|  4| Michael|     Sales|  3000| 20|          3000|
|  8|   Kiran|     Sales|  2200| 35|          5200|
|  2|  Robert|     Sales|  4000| 38|          9200|
+---+--------+----------+------+---+--------------+
```

Pada output departemen Finance, Martin (26 thn) memiliki kumulatif 3500. Maria (28 thn) menjadi 7000 (3500 Martin + 3500 Maria). Kelly (35 thn) menjadi 10500. Nilai ini terus bertambah secara akumulatif, memberikan wawasan tentang pertumbuhan biaya seiring profil usia.

Fungsi buatan pengguna (User-defined functions - UDFs)

UDF memungkinkan pengguna memperluas fungsionalitas SQL dengan logika Python kustom. Ini berguna ketika fungsi bawaan Spark tidak memadai.

Menerapkan UDF ke DataFrame (Applying a UDF to a DataFrame)

Implementasi Source Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Define a UDF to capitalize a string
capitalize_udf = udf(lambda x: x.upper(), StringType())

# Apply the UDF to a column
df_with_capitalized_names = salary_data_with_id.withColumn("capitalized_name",
capitalize_udf("Employee"))

# Display the result
df_with_capitalized_names.show()
```

Kita mendefinisikan fungsi lambda sederhana x.upper() dan membungkusnya dengan udf(). Parameter StringType() memastikan Spark tahu tipe data apa yang akan dikembalikan. Fungsi ini kemudian diaplikasikan kolom "Employee" untuk membuat kolom baru "capitalized_name".

Output:

```
+---+--------+----------+------+---+---------------+
| ID|Employee|Department|Salary|Age|capitalized_name|
+---+--------+----------+------+---+---------------+
|  1|    John| Field-eng|  3500| 40|           JOHN|
|  2|  Robert|     Sales|  4000| 38|         ROBERT|
|  3|   Maria|   Finance|  3500| 28|          MARIA|
|  4| Michael|     Sales|  3000| 20|        MICHAEL|
|  5|   Kelly|   Finance|  3500| 35|          KELLY|
|  6|    Kate|   Finance|  3000| 45|           KATE|
|  7|  Martin|   Finance|  3500| 26|         MARTIN|
|  8|   Kiran|     Sales|  2200| 35|          KIRAN|
+---+--------+----------+------+---+---------------+
```

Hasilnya menampilkan kolom baru di mana nama karyawan telah dikonversi menjadi huruf besar semua (JOHN, ROBERT, dst), membuktikan bahwa logika Python berhasil dijalankan pada setiap baris data66.

Menerapkan fungsi (Applying a function)

Bagian ini memperkenalkan **Pandas UDF** (Vectorized UDF), yang jauh lebih efisien daripada UDF standar karena memproses data dalam bentuk *batch* (Series), bukan baris per baris.

Implementasi Source Code:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import LongType

@udf(returnType=LongType())
def plus_one(value):
    if value is not None:
        return value + 1
    return None

salary_data_with_id.select(plus_one(salary_data_with_id.Salary).alias
("pandas_plus_one(Salary)")).show()
```

Dekorator @pandas_udf digunakan untuk mendefinisikan fungsi yang menerima dan mengembalikan pd.Series. Operasi series + 1 dilakukan secara vektorisasi oleh pustaka Pandas, yang memberikan peningkatan performa signifikan. Kode kemudian memilih kolom Salary dan menerapkan fungsi ini.

Output:

```
+----------------------+
|pandas_plus_one(Salary)|
+----------------------+
|                  3501|
|                  4001|
|                  3501|
|                  3001|
|                  3501|
|                  3001|
|                  3501|
|                  2201|
+----------------------+
```

Nilai gaji ditampilkan dengan tambahan 1 poin (misal 3500 jadi 3501). Meskipun output visualnya sederhana, keunggulan utamanya adalah efisiensi komputasi di latar belakang.

UDF juga bisa didaftarkan agar bisa dipanggil langsung dari string SQL.

Implementasi Source Code:

```
from pyspark.sql.types import IntegerType
salary_data_with_id.createOrReplaceTempView("employees")

def add_one_logic(value):
    if value is not None:
        return value + 1
    return None

spark.udf.register("add_one", add_one_logic, IntegerType())
spark.sql("SELECT add_one(Salary) FROM employees").show()
```

Metode spark.udf.register mengaitkan fungsi Python add_one dengan nama "add_one" di katalog SQL. Ini memungkinkan kita menulis SELECT add_one(Salary) layaknya menggunakan fungsi bawaan SQL, memberikan fleksibilitas maksimal bagi pengguna yang lebih suka sintaks SQL.

Output:

```
+---------------+
|add_one(Salary)|
+---------------+
|           3501|
|           4001|
|           3501|
|           3001|
|           3501|
|           3001|
|           3501|
|           2201|
+---------------+
```

Output identik dengan contoh sebelumnya, namun dicapai melalui antarmuka SQL yang berbeda.

Bekerja dengan tipe data kompleks – pivot dan unpivot (Working with complex data types – pivot and unpivot)

Bagian ini membahas transformasi struktural data.

1. **Pivot:** Mengubah baris unik menjadi kolom (Row to Column). Digunakan untuk membuat tabel laporan yang "melebar", di mana nilai data menjadi header kolom baru.
2. **Unpivot:** Kebalikannya, mengubah kolom menjadi baris (Column to Row). Berguna untuk menormalisasi tabel yang terlalu lebar menjadi format daftar yang panjang.

Kedua operasi ini vital untuk persiapan data visualisasi dan agregasi multidimensi.

Rangkuman (Summary)

Bab ini telah membahas siklus lengkap analisis data di Spark SQL: mulai dari memuat, memfilter, menyimpan sebagai tabel, hingga eksekusi kueri. Kita juga mempelajari fitur analitik lanjut seperti *Window Functions* untuk kalkulasi dalam partisi, serta penggunaan *UDF* (baik standar maupun Pandas) untuk menyisipkan logika kustom ke dalam alur kerja data.

Chapter 7

Structured Streaming in Spark

Pemrosesan Data Real-time (Real-time data processing)

Pemrosesan data secara *real-time* telah menjadi sangat penting dalam dunia yang serba cepat saat ini. Berbeda dengan pemrosesan *batch* tradisional yang bekerja pada kumpulan data statis, pemrosesan *real-time* melibatkan analisis data yang terus-menerus masuk (*streaming*) untuk memberikan wawasan instan. Karakteristik utamanya meliputi latensi rendah (waktu minimal antara data dibuat dan diproses), skalabilitas untuk menangani volume data tinggi, dan toleransi kesalahan (*fault tolerance*) untuk memastikan sistem tetap berjalan meskipun terjadi kegagalan.

Apa itu Streaming? (What is streaming?)

Streaming mengacu pada pemrosesan data yang berkelanjutan dan *real-time* saat data tersebut dihasilkan. Tidak seperti pemrosesan *batch* yang memproses data dalam potongan-potongan pada interval tetap, streaming memungkinkan aplikasi untuk menelan, memproses, dan menganalisis data secara inkremental. Hal ini memungkinkan pengambilan keputusan yang tepat waktu dan respons segera terhadap suatu peristiwa.

Arsitektur Streaming (Streaming architectures)

Arsitektur streaming dirancang untuk menangani kecepatan tinggi data. Komponen utamanya terdiri dari Sumber Streaming (asal data seperti sensor IoT atau Kafka), Mesin Pemrosesan (yang mengolah data), dan Sink Streaming (tujuan akhir data seperti database atau dashboard).

Terdapat beberapa jenis arsitektur:

1. **Arsitektur Berbasis Peristiwa (Event-driven):** Memproses peristiwa segera setelah terjadi untuk reaksi instan.
2. **Arsitektur Lambda:** Menggabungkan pemrosesan *batch* (untuk sejarah data) dan *stream* (untuk data terkini) secara paralel.
3. **Arsitektur Streaming Terpadu (Unified):** Seperti Structured Streaming di Spark, yang menyediakan satu API untuk menangani *batch* dan *stream*, menyederhanakan pengembangan.

Pengenalan Spark Streaming (Introducing Spark Streaming)

Spark Streaming adalah kerangka kerja awal di atas Spark yang memecah aliran data menjadi *micro-batches* (kumpulan kecil data). Setiap *micro-batch* diperlakukan sebagai RDD (*Resilient Distributed Dataset*). Arsitekturnya melibatkan **Driver Program** yang mengelola eksekusi dan **Receiver** yang menerima data dari sumber. Konsep kuncinya meliputi **DStreams** (aliran kontinu RDD), operasi **Window** (komputasi pada rentang waktu geser), dan **Checkpointing** untuk pemulihan kesalahan.

Pengenalan Structured Streaming (Introducing Structured Streaming)

Structured Streaming adalah evolusi revolusioner yang memperlakukan data streaming sebagai **tabel tanpa batas** (*unbounded table*) yang terus bertambah. Ini berbeda dengan model *micro-batch* pada Spark Streaming. Structured Streaming menggunakan API DataFrame/Dataset yang lebih ekspresif, menjamin

toleransi kesalahan *exactly-once* (data diproses tepat satu kali), dan menyatukan API untuk *batch* dan *streaming*.

Perbedaan Utama dengan Spark Streaming:

Structured Streaming memproses data secara inkremental dan kontinu (bukan micro-batch RDD diskrit), menggunakan API tingkat tinggi (SQL-like), dan menangani late-data (data terlambat) dengan mekanisme watermarking yang lebih canggih.

Dasar-dasar Structured Streaming (Structured Streaming concepts)

Untuk memahami Structured Streaming, kita perlu mengenal konsep waktu dan pemicu:

- **Waktu Peristiwa (Event Time):** Waktu saat peristiwa sebenarnya terjadi di dunia nyata (timestamp pada data).
- **Waktu Pemrosesan (Processing Time):** Waktu saat sistem memproses data tersebut.
- **Watermarking:** Mekanisme untuk menangani data yang datang terlambat (*late data*) dengan menetapkan batas waktu toleransi.
- **Triggers:** Menentukan kapan mesin harus memperbarui hasil (misalnya setiap detik atau berdasarkan *event*).
- **Mode Output:** Menentukan cara penulisan hasil ke *sink* (Append, Complete, atau Update).

Sumber dan Sink Streaming (Streaming sources and sinks)

Structured Streaming mendukung berbagai sumber dan tujuan data:

- **Sumber (Sources):** File (membaca file baru di direktori), Kafka (platform streaming populer), Socket (untuk testing), dan sumber kustom.
- **Sink (Sinks):** Console (untuk debugging), File, Kafka (menulis balik ke topik), dan Foreach (untuk logika penulisan kustom).

Teknik Lanjutan dalam Structured Streaming (Advanced techniques in Structured Streaming)

Di sinilah implementasi kode yang kompleks terjadi. Structured Streaming menangani aspek rumit seperti evolusi skema dan penggabungan aliran data secara otomatis.

1. Menangani Evolusi Skema (Handling schema evolution)

Dalam aplikasi streaming jangka panjang, struktur data (skema) dari sumber sering kali berubah, misalnya penambahan kolom baru. Structured Streaming mendukung evolusi skema dengan cara menggabungkan skema baru ke dalam aliran yang sedang berjalan.

Implementasi Source Code:

```
stream = spark.readStream \
    .format("csv") \
    .option("header", "true") \
    .schema(initialSchema) \
    .load("data/input")
mergedStream = stream \
    .selectExpr("col1", "col2", "new_col AS col3")
```

Pada blok kode di atas, proses dimulai dengan mendefinisikan sumber streaming menggunakan spark.readStream. Format .format("csv") dipilih untuk membaca file teks, dan opsi .schema(initialSchema) sangat krusial karena menetapkan ekspektasi awal aplikasi terhadap struktur data di direktori "data/input". Bagian kedua kode (mergedStream) menunjukkan cara menangani evolusi data. Jika file baru masuk dengan kolom tambahan bernama new_col, fungsi .selectExpr() digunakan untuk mengakomodasinya. Fungsi ini tidak hanya memilih kolom lama (col1, col2) tetapi juga menangkap kolom baru tersebut dan melakukan aliasing (mengganti nama) menjadi col3 secara dinamis. Ini memastikan bahwa perubahan struktur pada sumber data tidak mematikan aplikasi, melainkan beradaptasi dengan memasukkan data baru tersebut ke dalam aliran pemrosesan.

2. Berbagai Join di Structured Streaming (Different joins in Structured Streaming)

Structured Streaming memampukan penggabungan data (*join*) yang kompleks, baik antar sesama aliran data maupun antara aliran data dengan data statis.

A. Stream-Stream Joins (Penggabungan Dua Aliran)

Ini adalah teknik menggabungkan dua sumber data yang sama-sama bergerak (*streaming*) berdasarkan kunci tertentu.

Implementasi Source Code:

```
stream1 = spark.readStream.format("kafka").option("kafka.bootstrap.servers",
"localhost:9092") .option("subscribe", "topic1") .load()
stream2 =
spark.readStream.format("kafka").option("kafka.bootstrap.servers","localhost:
9092") .option("subscribe", "topic2") .load()

joinedStream =stream1.join(stream2, "common_key")
```

Kode ini mendemonstrasikan penggabungan dua aliran data dari Apache Kafka. Variabel stream1 dan stream2 diinisialisasi secara terpisah menggunakan spark.readStream.format("kafka"), masing-masing berlangganan ke topik yang berbeda (topic1 dan topic2). Kedua aliran ini berjalan secara independen dan terus-menerus menarik pesan baru. Baris kunci di sini adalah stream1.join(stream2, "common_key"). Operasi ini menginstruksikan Spark untuk memantau kedua sisi aliran. Ketika sebuah peristiwa masuk di

stream1, Spark menahannya sementara waktu dan mencari peristiwa pasangan di stream2 yang memiliki nilai common_key yang sama. Jika cocok, keduanya digabungkan. Proses ini sangat kompleks di belakang layar karena melibatkan manajemen state dan sinkronisasi waktu, namun disederhanakan menjadi satu baris kode di Structured Streaming.

B. Stream-Static Joins (Penggabungan Aliran dengan Data Statis)

Teknik ini menggabungkan aliran data *real-time* dengan dataset referensi yang statis (tidak berubah), biasanya untuk memperkaya data (*enrichment*).

Implementasi Source Code:

```
stream =spark.readStream.format("kafka").option("kafka.bootstrap.servers",
"localhost:9092").option("subscribe", "topic") .load()
staticData = spark.read.format("csv") .option("header",
"true").load("data/static_data.csv")

enrichedStream = stream.join(staticData,"common_key")
```

Potongan kode ini menyoroti perbedaan perlakuan antara data dinamis dan statis. Variabel stream didefinisikan dengan spark.readStream (untuk Kafka), menandakan sifatnya yang terus mengalir. Sebaliknya, variabel staticData didefinisikan menggunakan spark.read (tanpa "Stream") untuk memuat file CSV static_data.csv sebagai DataFrame biasa. Operasi stream.join(staticData, "common_key") kemudian dilakukan untuk "memperkaya" setiap pesan yang masuk dari Kafka. Setiap kali data baru tiba di stream, Spark melakukan pencarian (lookup) ke tabel staticData menggunakan kunci yang sama dan menambahkan kolom detail dari tabel statis tersebut ke hasil aliran. Metode ini sangat efisien karena data statis tidak memerlukan manajemen state yang rumit seperti pada stream-stream join.

Berikut adalah ringkasan untuk Bab 7 yang disesuaikan dengan gaya penulisan laporan/jurnal Anda sebelumnya:

Ringkasan (Summary)

Bab ini menyajikan eksplorasi komprehensif mengenai *Structured Streaming*, dimulai dari pemahaman konsep dasar, keuntungan arsitektural, hingga prinsip operasional yang membedakannya dari model pemrosesan *batch* tradisional. Pembahasan difokuskan pada bagaimana Spark menangani data yang masuk secara terus-menerus (*continuous data stream*) dengan abstraksi tabel yang tidak terbatas, memungkinkan penerapan logika kueri yang serupa dengan data statis namun berjalan secara *real-time*.

Secara spesifik, bab ini mendalami fungsionalitas inti dan teknik lanjutan seperti operasi berbasis jendela (*windowed operations*). Dua mekanisme utama yang dibahas adalah *sliding windows* dan *tumbling windows*, yang memungkinkan pengguna melakukan agregasi dan komputasi analitik dalam rentang waktu tertentu (*time-based analysis*). Selain itu, dibahas pula konsep *stateful streaming processing*, yaitu kemampuan aplikasi untuk memelihara dan memperbarui status (*state*) data antar-kelompok pemrosesan, serta integrasi dengan pustaka eksternal (API) untuk memperluas kapabilitas sistem dalam menangani tren pemrosesan data waktu nyata masa kini.

Chapter 8

Machine Learning with Spark ML

Machine Learning dengan Spark ML (Machine Learning with Spark ML)

Machine learning (ML) telah mendapatkan popularitas yang signifikan belakangan ini. Bab ini melakukan eksplorasi komprehensif mengenai Spark Machine Learning (ML), sebuah kerangka kerja yang kuat untuk ML yang dapat diskalakan (*scalable*) pada Apache Spark. Pembahasan mencakup konsep dasar ML dan bagaimana Spark ML memanfaatkan prinsip-prinsip tersebut untuk memungkinkan wawasan berbasis data yang efisien dan berskala besar.

Pengantar ML (Introduction to ML)

ML adalah bidang studi yang berfokus pada pengembangan algoritma dan model yang memungkinkan sistem komputer untuk belajar dan membuat prediksi atau keputusan tanpa diprogram secara eksplisit. Sebagai bagian dari kecerdasan buatan (AI), ML bertujuan memberikan kemampuan pada sistem untuk belajar secara otomatis dan meningkat berdasarkan pengalaman serta data. Dalam dunia modern di mana data dihasilkan dalam jumlah masif, ML memainkan peran kritis dalam mengekstrak wawasan bermakna dan mengotomatisasi pengambilan keputusan di berbagai domain seperti keuangan, kesehatan, dan pemasaran.

Konsep Kunci ML (The key concepts of ML)

Untuk memahami ML, sangat penting untuk menguasai konsep-konsep fundamental yang mendasari metodologinya.

Data (Data)

Data merupakan fondasi utama dari setiap proses ML, baik yang terstruktur, semi-terstruktur, maupun tidak terstruktur. Kualitas data sangat krusial; algoritma ML membutuhkan data yang relevan dan representatif untuk mempelajari pola secara akurat. Penggunaan data yang buruk atau bias dapat menyebabkan hasil yang tidak akurat dan menyesatkan, yang pada akhirnya berdampak negatif pada keputusan strategis yang diambil berdasarkan model tersebut. Tanggung jawab utama praktisi ML adalah memastikan data yang digunakan bebas dari bias inheren dan representatif terhadap populasi yang dituju.

Fitur (Features)

Fitur adalah properti atau karakteristik terukur dari data yang digunakan algoritma ML untuk membuat prediksi. Dari sekian banyak data yang tersedia, pemilihan fitur yang relevan sangat menentukan kualitas model yang dihasilkan. Proses seleksi, ekstraksi, dan transformasi fitur ini dikenal sebagai rekayasa fitur (*feature engineering*), yang berperan vital dalam meningkatkan kinerja model.

Label dan Target (Labels and targets)

Label atau target adalah hasil yang diinginkan yang ingin diprediksi atau diklasifikasikan oleh model ML. Dalam pembelajaran terawasi (*supervised learning*), label mewakili jawaban yang benar yang diajarkan kepada model. Sedangkan dalam pembelajaran tak terawasi (*unsupervised learning*), model berusaha mengidentifikasi pola tanpa adanya label eksplisit.

Pelatihan dan Pengujian (Training and testing)

Model ML dilatih menggunakan sebagian data yang disebut set pelatihan (*training set*), di mana model belajar memetakan input ke label. Setelah terlatih, kinerja model dievaluasi menggunakan set pengujian (*testing set*) yang terpisah. Evaluasi ini bertujuan untuk mengukur kemampuan model dalam melakukan generalisasi terhadap data baru yang belum pernah dilihat sebelumnya.

Algoritma dan Model (Algorithms and models)

Algoritma ML adalah prosedur statistik yang mempelajari hubungan dalam data. Algoritma ini dapat dikategorikan menjadi berbagai jenis seperti regresi, klasifikasi, atau klasterisasi. Ketika algoritma dilatih pada data, ia menghasilkan sebuah "model" yang menangkap pola yang dipelajari dan siap digunakan untuk prediksi.

Jenis-jenis ML (Types of ML)

Permasalahan ML secara luas dapat dikategorikan menjadi dua jenis utama yang memiliki karakteristik berbeda.

Pembelajaran Terawasi (Supervised learning)

Pembelajaran terawasi adalah jenis ML di mana algoritma belajar dari data pelatihan yang memiliki label. Tujuannya adalah mempelajari fungsi pemetaan yang dapat memprediksi output untuk input baru. Proses ini melibatkan persiapan data, pelatihan model untuk menemukan pola antara fitur dan label, evaluasi model menggunakan metrik tertentu (seperti akurasi atau *mean squared error*), dan penyebaran model untuk prediksi.

Pembelajaran Tak Terawasi (Unsupervised Learning)

Pembelajaran tak terawasi bekerja pada data yang tidak memiliki label output. Tujuannya adalah menemukan struktur tersembunyi, pola, atau klaster dalam data tersebut. Karena tidak ada label acuan, evaluasi model ini lebih bersifat subjektif dan sering kali melibatkan visualisasi atau validasi pengetahuan domain. Contoh algoritma ini termasuk K-means clustering dan Principal Component Analysis (PCA).

Jenis Pembelajaran Terawasi (Types of supervised learning)

Dalam pembelajaran terawasi, terdapat tiga jenis tugas utama:

1. **Klasifikasi (Classification):** Mengkategorikan data ke dalam kelas yang telah ditentukan (diskrit), seperti mendeteksi email spam .
2. **Regresi (Regression):** Memprediksi nilai numerik yang kontinu, seperti memprediksi harga rumah atau tren saham.
3. **Deret Waktu (Time series):** Menganalisis data yang dikumpulkan berdasarkan urutan waktu untuk meramalkan tren masa depan.

ML dengan Spark (ML with Spark)

Spark menyediakan platform yang kuat dan berskala besar untuk tugas ML melalui pustaka MLlib. Keunggulan utamanya meliputi kemampuan komputasi terdistribusi, pemrosesan data yang efisien, serta integrasi mulus dengan Spark SQL dan Spark Streaming.

Keuntungan Apache Spark untuk ML Skala Besar (Advantages of Apache Spark for large-scale ML)

Spark menawarkan keunggulan kompetitif untuk menangani dataset besar:

- **Kecepatan dan Kinerja:** Menggunakan komputasi *in-memory* yang mempercepat algoritma iteratif secara drastis.
- **Komputasi Terdistribusi:** Memproses data secara paralel di banyak node, memungkinkan skalabilitas horizontal.
- **Toleransi Kesalahan:** Mekanisme pemulihan otomatis melalui garis keturunan RDD memastikan keandalan aplikasi.

Spark MLlib versus Spark ML (Spark MLlib versus Spark ML)

Apache Spark menyediakan dua pustaka ML dengan perbedaan mendasar:

- **Spark MLlib:** Pustaka asli berbasis API RDD. Cocok untuk kebutuhan kontrol tingkat rendah dan kompatibilitas dengan versi Spark lama .
- **Spark ML:** Pustaka yang lebih baru (sejak Spark 2.0) berbasis API DataFrame. Menawarkan antarmuka tingkat tinggi yang lebih ramah pengguna, mendukung konsep *Pipelines* untuk alur kerja ML yang efisien, dan terintegrasi dengan Spark SQL. Spark ML lebih disarankan untuk pengembangan modern.

Siklus Hidup ML (ML life cycle)

Pengembangan model ML adalah proses *end-to-end* yang melibatkan tahapan berikut:

1. Definisi Masalah: Menentukan tujuan bisnis dan metrik kesuksesan.
2. Akuisisi dan Pemahaman Data: Mengumpulkan dan mengeksplorasi data.
3. Persiapan Data dan Rekayasa Fitur: Membersihkan dan mentransformasi data agar siap dilatih.
4. Pelatihan dan Evaluasi Model: Melatih algoritma dan mengukur kinerjanya.
5. Penyebaran Model: Mengintegrasikan model ke lingkungan produksi.
6. Pemantauan dan Pemeliharaan Model: Memastikan kinerja model tetap optimal seiring waktu.
7. Iterasi dan Perbaikan Model: Memperbarui model berdasarkan umpan balik dan data baru.

Pernyataan Masalah (Problem statement)

Bagian ini membahas sebuah studi kasus praktis mengenai seni memprediksi harga rumah menggunakan data historis. Bayangkan kita memiliki harta karun berupa informasi berharga tentang rumah, yang mencakup detail seperti zonasi, luas tanah, tipe bangunan, kondisi keseluruhan, tahun dibangun, dan harga jual. Tujuan utama kita adalah memanfaatkan kekuatan *Machine Learning* (ML) untuk memprediksi secara akurat harga rumah baru yang masuk ke pasar. Untuk mencapai hal ini, kita akan membangun model ML yang dirancang khusus untuk prediksi harga rumah dengan memanfaatkan data historis yang ada. Mengingat harga rumah adalah variabel yang bersifat kontinu (angka yang berkelanjutan), model yang paling tepat untuk digunakan dalam kasus ini adalah **Regresi Linear**. Proses ini akan dimulai dengan mempersiapkan data agar dapat diproses dengan benar oleh algoritma ML.

Persiapan Data dan Rekayasa Fitur (Data preparation and feature engineering)

Persiapan data dan rekayasa fitur merupakan langkah-langkah yang sangat krusial dalam proses ML. Penerapan teknik persiapan data dan rekayasa fitur yang tepat dapat secara signifikan meningkatkan kinerja dan akurasi model yang dihasilkan. Bagian ini akan mengeksplorasi tugas-tugas umum dalam persiapan data, mulai dari pemuatan data hingga transformasi fitur yang kompleks.

Pengenalan Dataset (Introduction to the dataset)

Langkah pertama dalam membangun model adalah menemukan data yang relevan. Studi kasus ini menggunakan data harga rumah yang berlokasi dalam sebuah file CSV, unduhan dilakukan melalui link: (https://docs.google.com/spreadsheets/d/1caaR9pT24GNmq3rDQpMiIMJrmiTGarbs/edit#gid=1150341366). Data ini memiliki total 2.920 baris dan 13 kolom. Kolom-kolom tersebut mencakup berbagai atribut properti, antara lain Id sebagai pengenal unik, MSSubClass untuk subkelas properti, MSZoning untuk zona, LotArea untuk luas tanah, BldgType untuk tipe bangunan, hingga SalePrice yang merupakan harga jual dan menjadi target prediksi kita .

Memuat Data (Loading data)

Proses teknis dimulai dengan memuat dataset yang telah diunduh ke dalam lingkungan Databricks atau Spark menggunakan struktur DataFrame.

Implementasi Source Code:

```
housing_data = spark.read.csv("HousePricePrediction.csv", header=True)
housing_data.show(5)
```

Pada baris kode pertama, kita menginisialisasi variabel housing_data dengan memanfaatkan objek sesi spark. Metode .read.csv("HousePricePrediction.csv") dipanggil secara spesifik untuk membaca file berformat Comma Separated Values (CSV). Fungsi ini bekerja dengan mem-parsing file teks mentah dan mengonversinya menjadi DataFrame Spark, sebuah struktur data terdistribusi yang mirip dengan tabel database. Setelah data dimuat ke memori, baris kedua kode mengeksekusi perintah .show(5). Perintah ini adalah sebuah action dalam Spark yang memicu evaluasi data dan menampilkan lima baris pertama ke layar konsol. Langkah verifikasi visual ini sangat penting untuk memastikan bahwa file telah terbaca dengan benar dan delimiternya sesuai sebelum kita melakukan manipulasi lebih lanjut.

Output:

Id	MSSubClass	MSZoning	LotArea	LotConfig	BldgType	OverallCond	YearBuilt	YearRemodAdd	Exterior1st	BsmtFinSF2	TotalBsmtSF	SalePrice
0	60	RL	8450	Inside	1Fam	5	2003	2003	VinylSd	0	856	208500
1	20	RL	9600	FR2	1Fam	8	1976	1976	MetalSd	0	1262	181500
2	60	RL	11250	Inside	1Fam	5	2001	2002	VinylSd	0	920	223500
3	70	RL	9550	Corner	1Fam	5	1915	1970	Wd Sdng	0	756	140000
4	60	RL	14260	FR2	1Fam	5	2000	2000	VinylSd	0	1145	250000

only showing top 5 rows

Tabel output menyajikan tinjauan awal terhadap struktur data kita. Dari sini, kita dapat mengamati variasi tipe data yang ada dalam dataset. Kolom seperti LotArea jelas berisi nilai numerik (contohnya 8450), sedangkan kolom lain seperti MSZoning dan LotConfig berisi data teks atau string (contohnya "RL" dan

"Inside"). Temuan visual ini memberikan wawasan awal yang krusial bahwa kita tidak bisa langsung memasukkan data ini ke dalam algoritma matematis seperti regresi linear. Keberadaan data bertipe string mengharuskan kita melakukan proses transformasi atau encoding di tahap selanjutnya agar mesin dapat memprosesnya.

Selanjutnya, kita perlu memeriksa metadata atau skema data untuk mendapatkan kepastian teknis mengenai tipe data setiap kolom.

Implementasi Source Code:

```
housing_data.printSchema
```

Kode housing_data.printSchema memanggil metode bawaan DataFrame yang berfungsi untuk mencetak struktur skema data dalam format pohon hirarkis. Berbeda dengan .show() yang menampilkan isi data, .printSchema menampilkan metadata, yaitu nama kolom dan tipe data logisnya (seperti integer, string, double). Perintah ini sangat membantu untuk mengidentifikasi secara presisi kolom mana yang dianggap sebagai angka dan kolom mana yang dianggap sebagai teks oleh Spark, sehingga kita dapat merencanakan strategi pembersihan data yang tepat.

Output:

```
<bound method DataFrame.printSchema of DataFrame[Id: string, MSSubClass: string, MSZoning: string, LotArea: string, LotConfig:
string, BldgType: string, OverallCond: string, YearBuilt: string, YearRemodAdd: string, Exterior1st: string, BsmtFinSF2: string,
TotalBsmtSF: string, SalePrice: string]>
```

Output skema ini mengonfirmasi observasi kita sebelumnya dengan definisi tipe data yang tegas. Kita dapat melihat bahwa kolom seperti Id, MSSubClass, LotArea, OverallCond, dan SalePrice terdeteksi sebagai bigint (bilangan bulat besar), yang berarti sudah siap untuk operasi numerik. Sebaliknya, kolom MSZoning, LotConfig, BldgType, dan Exterior1st teridentifikasi sebagai string. Informasi ini menjadi panduan taktis bagi kita bahwa kolom-kolom string inilah yang akan menjadi target utama proses pembersihan dan rekayasa fitur selanjutnya.

Membersihkan Data (Cleaning data)

Data mentah dari dunia nyata jarang sekali sempurna dan sering kali mengandung nilai yang hilang (*missing values*) atau inkonsistensi. Pembersihan data melibatkan penanganan nilai-nilai kosong ini karena algoritma ML umumnya tidak dapat mentoleransi input *null*.

Implementasi Source Code:

```
housing_data.count()
```

Blok kode ini menjalankan prosedur audit dan sanitasi data. Pertama, perintah housing_data.count() dieksekusi untuk menghitung jumlah total baris dalam dataset awal, memberikan kita garis dasar (baseline) ukuran data.

Implementasi Source Code:

```
# Remove rows with missing values
cleaned_data = housing_data.dropna()
cleaned_data.count()
```

Selanjutnya, kita membuat DataFrame baru bernama cleaned_data dengan menerapkan metode .dropna() pada data asli. Metode .dropna() ini bekerja secara agresif dengan memindai setiap baris data; jika ditemukan setidaknya satu kolom yang bernilai kosong (null) dalam sebuah baris, maka seluruh baris tersebut akan dihapus dari dataset. Terakhir, cleaned_data.count() menghitung kembali jumlah baris pada data yang sudah dibersihkan. Membandingkan hitungan sebelum dan sesudah pembersihan memberikan gambaran seberapa banyak data yang terbuang demi mendapatkan dataset yang utuh.

Output:

```
2919
```

```
1460
```

Hasil eksekusi menunjukkan angka 2.919 untuk jumlah baris awal dan 1.460 untuk jumlah baris setelah pembersihan. Ini mengindikasikan bahwa proses pembersihan telah memangkas sekitar separuh dari total data kita. Meskipun pengurangan data ini tampak drastis, langkah ini sangat penting untuk memastikan integritas model. Data yang tersisa kini dijamin lengkap tanpa nilai kosong, sehingga menghilangkan risiko error saat pelatihan model atau prediksi yang tidak akurat akibat data yang bolong-bolong.

Menangani Variabel Kategorikal (Handling categorical variables)

Variabel kategorikal adalah variabel yang mewakili kategori atau grup (seperti zona tanah "RL" atau "RM") dan tidak memiliki makna numerik inheren. Karena algoritma ML bekerja dengan operasi matematika, kita wajib mengonversi data teks ini menjadi format numerik sebelum pelatihan model dimulai.

Implementasi Source Code (StringIndexer):

```
#import required libraries
from pyspark.ml.feature import StringIndexer
mszoning_indexer = StringIndexer(inputCol="MSZoning",
outputCol="MSZoningIndex")
#Fits a model to the input dataset with optional parameters.
df_mszoning = mszoning_indexer.fit(cleaned_data).transform(cleaned_data)
df_mszoning.show()
```

Di sini, kita menggunakan StringIndexer dari pustaka Spark ML untuk menangani konversi dasar teks ke angka. Kode dimulai dengan mengimpor kelas StringIndexer. Kemudian, kita menginisialisasi sebuah objek indexer dengan parameter inputCol="MSZoning" yang menargetkan kolom sumber berisi teks, dan outputCol="MSZoningIndex" sebagai nama kolom tujuan untuk hasil angka. Proses intinya terjadi pada baris berikutnya yang menggabungkan metode .fit() dan .transform(). Metode .fit(cleaned_data)

memerintahkan Spark untuk memindai seluruh kolom dan "mempelajari" semua kategori unik yang ada serta frekuensinya. Setelah pola dipelajari, metode .transform(cleaned_data) menerapkannya untuk mengubah setiap nilai teks menjadi indeks numerik (misalnya 0.0, 1.0). Hasil akhirnya ditampilkan dengan .show() untuk verifikasi.

Output:

Id	MSSubClass	MSZoning	LotArea	LotConfig	BldgType	OverallCond	YearBuilt	YearRemodAdd	Exterior1st	BsmtFinSF2	TotalBsmtSF	SalePrice	MSZoningIndex
0	60	RL	8450	Inside	1Fam	5	2003	2003	VinylSd	0	856	208500	0.0
1	20	RL	9600	FR2	1Fam	8	1976	1976	MetalSd	0	1262	181500	0.0
2	60	RL	11250	Inside	1Fam	5	2001	2002	VinylSd	0	920	223500	0.0
3	70	RL	9550	Corner	1Fam	5	1915	1970	Wd Sdng	0	756	140000	0.0
4	60	RL	14260	FR2	1Fam	5	2000	2000	VinylSd	0	1145	250000	0.0
5	50	RL	14115	Inside	1Fam	5	1993	1995	VinylSd	0	796	143000	0.0
6	20	RL	10084	Inside	1Fam	5	2004	2005	VinylSd	0	1686	307000	0.0
7	60	RL	10382	Corner	1Fam	6	1973	1973	HdBoard	32	1107	200000	0.0
8	50	RM	6120	Inside	1Fam	5	1931	1950	BrkFace	0	952	129900	1.0
9	190	RL	7420	Corner	2fmCon	6	1939	1950	MetalSd	0	991	118000	0.0
10	20	RL	11200	Inside	1Fam	5	1965	1965	HdBoard	0	1040	129500	0.0
11	60	RL	11924	Inside	1Fam	5	2005	2006	WdShing	0	1175	345000	0.0
12	20	RL	12968	Inside	1Fam	6	1962	1962	HdBoard	0	912	144000	0.0
13	20	RL	10652	Inside	1Fam	5	2006	2007	VinylSd	0	1494	279500	0.0
14	20	RL	10920	Corner	1Fam	5	1960	1960	MetalSd	0	1253	157000	0.0
15	45	RM	6120	Corner	1Fam	8	1929	2001	Wd Sdng	0	832	132000	1.0
16	20	RL	11241	CulDSac	1Fam	7	1970	1970	Wd Sdng	0	1004	149000	0.0
17	90	RL	10791	Inside	Duplex	5	1967	1967	MetalSd	0	0	90000	0.0
18	20	RL	13695	Inside	1Fam	5	2004	2004	VinylSd	0	1114	159000	0.0
19	20	RL	7560	Inside	1Fam	6	1958	1965	BrkFace	0	1029	139000	0.0

only showing top 20 rows

Tabel output memperlihatkan penambahan kolom baru MSZoningIndex di sebelah kanan kolom asli. Nilai teks pada MSZoning kini memiliki representasi angka. Namun, penggunaan indeks angka tunggal ini memiliki kelemahan potensial: model regresi dapat salah menginterpretasikan angka-angka ini (misalnya 0, 1, 2) sebagai urutan peringkat atau besaran nilai, padahal kategori zona tanah tidak memiliki tingkatan seperti itu. Untuk menghindari bias "peringkat" palsu ini, kita memerlukan teknik yang lebih canggih yaitu One Hot Encoding yang akan diterapkan melalui Pipeline.

Implementasi Source Code:

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml import Pipeline
mszoning_indexer = StringIndexer(inputCol="MSZoning",
outputCol="MSZoningIndex")
lotconfig_indexer = StringIndexer(inputCol="LotConfig",
outputCol="LotConfigIndex")
bldgtype_indexer = StringIndexer(inputCol="BldgType",
outputCol="BldgTypeIndex")
exterior1st_indexer = StringIndexer(inputCol="Exterior1st",
outputCol="Exterior1stIndex")
onehotencoder_mszoning_vector = OneHotEncoder(inputCol="MSZoningIndex",
outputCol="MSZoningVector")
onehotencoder_lotconfig_vector = OneHotEncoder(inputCol="LotConfigIndex",
outputCol="LotConfigVector")
```

```
onehotencoder_bldgtype_vector = OneHotEncoder(inputCol="BldgTypeIndex",
outputCol="BldgTypeVector")
onehotencoder_exterior1st_vector = OneHotEncoder(inputCol="Exterior1stIndex",
outputCol="Exterior1stVector")
#Create pipeline and pass all stages
pipeline = Pipeline(stages=[mszoning_indexer,
                            lotconfig_indexer,
                            bldgtype_indexer,
                            exterior1st_indexer,
                            onehotencoder_mszoning_vector,
                            onehotencoder_lotconfig_vector,
                            onehotencoder_bldgtype_vector,
                            onehotencoder_exterior1st_vector])
```

Blok kode ini mengimplementasikan solusi yang lebih robust menggunakan Pipeline untuk mengelola transformasi data yang kompleks secara otomatis. Pertama, kita mengimpor OneHotEncoder dan Pipeline. OneHotEncoder bertugas mengubah angka indeks (hasil dari langkah sebelumnya) menjadi vektor biner, sehingga setiap kategori diperlakukan secara independen tanpa implikasi urutan. Kode kemudian mendefinisikan serangkaian transformasi: indexer untuk mengubah string ke indeks, dan encoder untuk mengubah indeks ke vektor, yang diterapkan pada setiap kolom kategorikal (MSZoning, LotConfig, BldgType, Exterior1st). Semua tahapan ini disusun secara berurutan dalam parameter stages pada objek Pipeline.

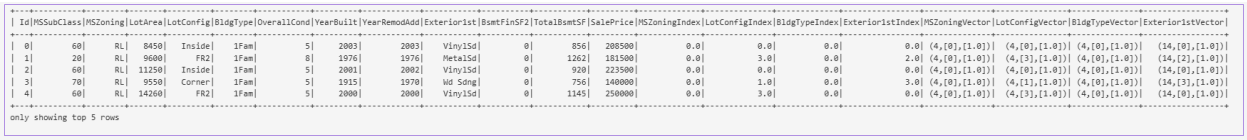
Setelah kita mendefinisikan seluruh tahapan dalam objek pipeline (mulai dari *indexer* hingga *encoder*), langkah selanjutnya adalah menerapkan rencana tersebut ke data kita. Kita perlu "melatih" pipeline ini agar ia memahami struktur data, lalu menggunakannya untuk mentransformasi data mentah menjadi data matang.

Implementasi Source Code:

```
df_transformed = pipeline.fit(cleaned_data).transform(cleaned_data)
df_transformed.show(5)
```

Pada baris pertama, perintah df_transformed = pipeline.fit(cleaned_data).transform(cleaned_data) melakukan dua operasi fundamental secara berantai. Pertama, metode .fit(cleaned_data) dipanggil. Di sini, Spark menjalankan seluruh tahap StringIndexer yang ada dalam pipeline untuk memindai kolom input dan "mempelajari" pemetaan kategori (misalnya: "RL" dipetakan ke 0, "RM" ke 1). Setelah proses pembelajaran selesai, metode .transform(cleaned_data) langsung dieksekusi. Metode ini menerapkan pemetaan tersebut serta menjalankan tahap OneHotEncoder untuk mengubah data menjadi vektor. Hasil akhirnya disimpan dalam variabel baru bernama df_transformed, yang kini berisi data asli ditambah dengan semua kolom hasil transformasi. Baris kedua, df_transformed.show(5), digunakan untuk menampilkan lima baris pertama dari DataFrame baru ini guna memverifikasi bahwa kolom-kolom baru telah berhasil dibuat.

Output:



Id	MSSubClass	MSZoning	LotArea	LotConfig	BldgType	OverallCond	YearBuilt	YearRemodAdd	Exterior1st	BsmtFinSF2	TotalBsmtSF	SalePrice	MSZoningIndex	LotConfigIndex	BldgTypeIndex	Exterior1stIndex	MSZoningVector	LotConfigVector	BldgTypeVector	Exterior1stVector
0	60	RL	8450	Inside	1Fam	5	2003	2003	VinylSd	0	856	208500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])
1	20	RL	9600	FR2	1Fam	8	1976	1976	MetalSd	0	1262	181500	0.0	3.0	0.0	2.0	(4,[0],[1.0])	(4,[3],[1.0])	(4,[0],[1.0])	(14,[2],[1.0])
2	60	RL	11250	Inside	1Fam	5	2001	2002	VinylSd	0	920	223500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])
3	70	RL	9550	Corner	1Fam	5	1915	1970	Wd Sdng	0	756	140000	0.0	1.0	0.0	3.0	(4,[0],[1.0])	(4,[1],[1.0])	(4,[0],[1.0])	(14,[3],[1.0])
4	60	RL	14260	FR2	1Fam	5	2000	2000	VinylSd	0	1145	250000	0.0	3.0	0.0	0.0	(4,[0],[1.0])	(4,[3],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])

only showing top 5 rows

DataFrame hasil transformasi (df_transformed) kini jauh lebih kaya informasi. Kita dapat melihat kolom-kolom baru dengan akhiran "Vector" (seperti MSZoningVector). Kolom-kolom ini berisi representasi numerik yang akurat dari data teks asli dalam format vektor biner. Format ini memastikan algoritma ML dapat memahami perbedaan kategori tanpa terjebak dalam bias asumsi urutan angka, yang merupakan kunci untuk akurasi model yang tinggi.

Pembersihan Data Lanjutan (Data cleanup)

Pasca transformasi Pipeline, dataset kita menjadi sangat lebar dan mengandung redundansi. Kita masih memiliki kolom teks asli, kolom indeks sementara, dan kolom vektor hasil akhir. Langkah pembersihan lanjutan ini diperlukan untuk merampingkan dataset sehingga hanya fitur yang relevan yang tersisa.

Implementasi Source Code:

```
drop_column_list = ["Id", "MSZoning","LotConfig","BldgType", "Exterior1st"]
df_dropped_cols = df_transformed.select([column for column in
df_transformed.columns if column not in drop_column_list])
df_dropped_cols.columns
```

Dalam blok kode ini, kita pertama-tama mendefinisikan sebuah list bernama drop_column_list yang berisi nama-nama kolom yang ingin dibuang, yaitu kolom identitas Id (yang tidak memiliki nilai prediktif) dan kolom-kolom string asli yang sudah kita konversi (MSZoning, LotConfig, dst). Baris berikutnya menggunakan teknik list comprehension Python di dalam fungsi .select(). Logika kode ini menyaring kolom DataFrame dengan cara memilih semua kolom yang namanya tidak terdapat di dalam daftar drop_column_list. Pendekatan ini sangat efisien untuk menghapus banyak kolom sekaligus tanpa harus menyebutkan satu per satu kolom yang ingin disimpan. Hasil akhirnya disimpan dalam df_dropped_cols.

Output:

```
['MSSubClass',
 'LotArea',
 'OverallCond',
 'YearBuilt',
 'YearRemodAdd',
 'BsmtFinSF2',
 'TotalBsmtSF',
 'SalePrice',
 'MSZoningIndex',
 'LotConfigIndex',
 'BldgTypeIndex',
 'Exterior1stIndex',
 'MSZoningVector',
 'LotConfigVector',
 'BldgTypeVector',
 'Exterior1stVector']
```

Daftar nama kolom yang dicetak mengonfirmasi keberhasilan proses pembersihan. Dataset kini hanya terdiri dari kolom fitur numerik asli (seperti MSSubClass, LotArea) dan kolom vektor hasil rekayasa fitur (seperti MSZoningVector, LotConfigVector). Semua data string mentah dan kolom pengenal yang tidak berguna telah dihilangkan sepenuhnya, menyisakan dataset yang bersih dan fokus untuk keperluan pemodelan.

Merakit Vektor (Assembling the vector)

Spark ML memiliki persyaratan arsitektur yang unik dan spesifik untuk algoritma pelatihannya. Model di Spark tidak menerima input berupa puluhan kolom fitur yang terpisah-pisah. Sebaliknya, seluruh fitur input (prediktor) harus digabungkan dan dipadatkan menjadi satu kolom vektor tunggal.

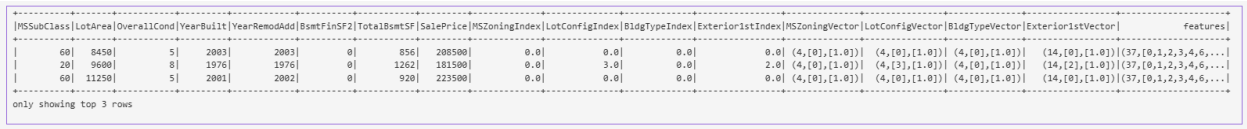
Implementasi Source Code:

```
from pyspark.ml.feature import VectorAssembler
#Assembling features
feature_assembly = VectorAssembler(inputCols = ['MSSubClass',
'LotArea',
'OverallCond',
'YearBuilt',
'YearRemodAdd',
'BsmtFinSF2',
'TotalBsmtSF',
'MSZoningIndex',
'LotConfigIndex',
'BldgTypeIndex',
'Exterior1stIndex',
'MSZoningVector',
'LotConfigVector',
'BldgTypeVector',
```

```
'Exterior1stVector'], outputCol = 'features')
output = feature_assembly.transform(df_dropped_cols)
output.show(3)
```

Kode ini mengimplementasikan fungsi VectorAssembler, yang bertugas sebagai "pengemas" fitur. Kita menginisialisasi objek feature_assembly dengan dua parameter utama: inputCols dan outputCol. Parameter inputCols menerima sebuah list panjang berisi nama semua kolom yang akan digunakan sebagai variabel independen untuk memprediksi harga (mencakup fitur numerik asli dan fitur vektor hasil encoding). Parameter outputCol='features' menetapkan bahwa hasil penggabungan ini akan disimpan dalam satu kolom baru bernama "features". Saat metode .transform(df_dropped_cols) dijalankan, Spark mengambil nilai dari setiap kolom input untuk setiap baris data dan menggabungkannya menjadi sebuah array vektor padat.

Output:



MSSubClass	LotArea	OverallCond	YearBuilt	YearRemodAdd	BsmtFinSF2	TotalBsmtSF	SalePrice	MSZoningIndex	LotConfigIndex	BldgTypeIndex	Exterior1stIndex	MSZoningVector	LotConfigVector	BldgTypeVector	Exterior1stVector	features
60	8450	5	2003	2003	0	856	208500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])	(37,[0,1,2,3,4,6,...]
20	9600	8	1976	1976	0	1262	181500	0.0	3.0	0.0	2.0	(4,[0],[1.0])	(4,[3],[1.0])	(4,[0],[1.0])	(14,[2],[1.0])	(37,[0,1,2,3,4,6,...]
60	11250	5	2001	2002	0	920	223500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])	(37,[0,1,2,3,4,6,...]

only showing top 3 rows

Pada tabel output, kita melihat kolom features yang baru terbentuk. Isi kolom ini bukan lagi angka tunggal, melainkan sebuah array atau vektor yang merangkum semua karakteristik rumah dalam satu unit data. Struktur data inilah yang menjadi standar input wajib bagi sebagian besar algoritma di pustaka Spark ML, memungkinkan pemrosesan paralel yang sangat efisien.

Skala Fitur (Feature scaling)

Fitur-fitur dalam dataset kita memiliki rentang nilai yang sangat timpang. Sebagai contoh, LotArea (luas tanah) bisa memiliki nilai ribuan atau puluhan ribu, sedangkan OverallCond (kondisi rumah) hanya berkisar antara skala 1 hingga 10. Perbedaan skala yang ekstrem ini dapat membingungkan model regresi, menyebabkan model menjadi bias dan terlalu mementingkan fitur dengan angka nominal besar. Oleh karena itu, penyekalaan fitur (*feature scaling*) diperlukan untuk menormalkan semua data ke dalam rentang standar yang sebanding.

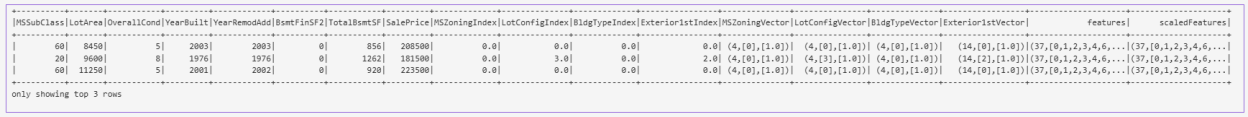
Implementasi Source Code:

```
#Normalizing the features
from pyspark.ml.feature import StandardScaler

scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                        withStd=True, withMean=False)

# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(output)
```

```
# Normalize each feature to have unit standard deviation.
scaledOutput = scalerModel.transform(output)
scaledOutput.show(3)

#Selecting input and output column from output
df_model_final = scaledOutput.select(['SalePrice', 'scaledFeatures'])
df_model_final.show(3)
```

Di sini, kita menggunakan StandardScaler untuk melakukan normalisasi. Objek scaler diatur dengan parameter inputCol="features" (menargetkan kolom vektor gabungan kita) dan outputCol="scaledFeatures". Opsi withStd=True diaktifkan untuk menstandarisasi data agar memiliki unit deviasi standar yang sama, sementara withMean=False dipilih untuk menjaga efisiensi pada vektor jarang (sparse vectors). Proses .fit(output) menghitung statistik ringkasan seperti standar deviasi dari seluruh dataset. Kemudian, .transform(output) menerapkan perhitungan tersebut ke setiap baris data, menghasilkan kolom scaledFeatures. Terakhir, kita menggunakan .select() untuk membuang semua kolom perantara dan hanya menyisakan dua kolom vital: SalePrice sebagai target prediksi, dan scaledFeatures sebagai input yang sudah dinormalisasi.

Output:



MSSubClass	LotArea	OverallCond	YearBuilt	YearRemodAdd	BsmtFinSF2	TotalBsmtSF	SalePrice	MSZoningIndex	LotConfigIndex	BldgTypeIndex	Exterior1stIndex	MSZoningVector	LotConfigVector	BldgTypeVector	Exterior1stVector	features	scaledFeatures
60	8450	5	2003	2003	0	856	208500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])	(37,[0,1,2,3,4,6,...]	(37,[0,1,2,3,4,6,...]
20	9600	8	1976	1976	0	1262	181500	0.0	3.0	0.0	2.0	(4,[0],[1.0])	(4,[3],[1.0])	(4,[0],[1.0])	(14,[2],[1.0])	(37,[0,1,2,3,4,6,...]	(37,[0,1,2,3,4,6,...]
60	11250	5	2001	2002	0	920	223500	0.0	0.0	0.0	0.0	(4,[0],[1.0])	(4,[0],[1.0])	(4,[0],[1.0])	(14,[0],[1.0])	(37,[0,1,2,3,4,6,...]	(37,[0,1,2,3,4,6,...]

only showing top 3 rows

SalePrice	scaledFeatures
208500	(37,[0,1,2,3,4,6,...]
181500	(37,[0,1,2,3,4,6,...]
223500	(37,[0,1,2,3,4,6,...]

only showing top 3 rows

Tabel hasil akhir menampilkan bentuk data yang paling murni dan siap untuk proses pemodelan. Kita hanya melihat dua kolom: SalePrice yang berisi jawaban atau nilai yang harus diprediksi, dan scaledFeatures yang berisi semua informasi rumah yang telah dikemas dan dinormalisasi secara matematis. Dengan struktur data yang bersih, terstandarisasi, dan terpadu ini, tahap persiapan data dan rekayasa fitur telah selesai sepenuhnya. Data ini sekarang dalam kondisi optimal untuk disuplai ke dalam algoritma Machine Learning pada tahap pelatihan berikutnya.

Pelatihan dan Evaluasi Model (Model training and evaluation)

Pelatihan dan evaluasi model merupakan tahapan sentral dalam siklus hidup *Machine Learning*. Setelah data disiapkan dengan matang, tahap ini berfokus pada dua hal: mengajarkan algoritma untuk mengenali pola dari data tersebut (pelatihan) dan mengukur seberapa akurat algoritma tersebut dalam membuat prediksi (evaluasi). PySpark menyediakan kerangka kerja yang komprehensif untuk menjalankan kedua proses ini secara efisien.

Memisahkan Data (Splitting the data)

Sebelum model dilatih, merupakan praktik standar untuk membagi dataset menjadi dua bagian terpisah: set pelatihan (*training set*) dan set pengujian (*testing set*). Pembagian ini krusial untuk mencegah bias; model

harus belajar dari satu set data dan diuji pada set data yang berbeda untuk membuktikan kemampuannya dalam melakukan generalisasi pada situasi dunia nyata.

Implementasi Source Code:

```
#test train split
df_train, df_test = df_model_final.randomSplit([0.75, 0.25])
```

Baris kode ini mengeksekusi metode .randomSplit() pada DataFrame df_model_final yang telah kita siapkan sebelumnya. Argumen [0.75, 0.25] adalah parameter rasio yang menginstruksikan Spark untuk memecah data menjadi dua bagian: 75% dari total data akan dialokasikan ke variabel df_train untuk keperluan pelatihan, sedangkan 25% sisanya dialokasikan ke variabel df_test untuk pengujian. Penggunaan pengacakan (random) di sini sangat penting untuk memastikan statistik data terdistribusi secara merata. Jika kita hanya memotong data berdasarkan urutan baris, kita berisiko melatih model hanya pada rumah-rumah tua (jika data diurutkan berdasarkan tahun) dan mengujinya pada rumah baru, yang akan menghasilkan evaluasi yang bias. Dengan .randomSplit(), setiap titik data memiliki peluang acak untuk masuk ke salah satu set, menjamin representasi yang adil.

Pelatihan Model (Model training)

Setelah data terbagi, kita memasuki tahap pelatihan. Untuk studi kasus prediksi harga rumah yang merupakan variabel kontinu, algoritma Linear Regression adalah pilihan yang tepat.

Implementasi Source Code:

```
from pyspark.ml.regression import LinearRegression
# Instantiate the linear regression model
regressor = LinearRegression(featuresCol = 'scaledFeatures', labelCol =
'SalePrice')
# Fit the model on the training data
regressor = regressor.fit(df_train)
```

Proses pelatihan dimulai dengan mengimpor modul LinearRegression dari pustaka pyspark.ml.regression. Selanjutnya, kita melakukan instansiasi objek model yang diberi nama regressor. Pada tahap inisialisasi ini, kita mengonfigurasi dua parameter vital: featuresCol='scaledFeatures' yang memberi tahu algoritma kolom mana yang berisi data input (vektor fitur yang sudah diskalakan), dan labelCol='SalePrice' yang menunjukkan kolom target prediksi (harga aktual). Setelah konfigurasi selesai, metode .fit(df_train) dipanggil. Inilah momen krusial di mana proses pembelajaran terjadi. Algoritma memproses 75% data yang ada di df_train, mencari persamaan matematika (garis regresi linear) terbaik yang dapat meminimalkan jarak antara prediksi dan harga sebenarnya. Hasil dari proses komputasi ini adalah sebuah model terlatih yang disimpan kembali ke dalam variabel regressor.

Evaluasi Model (Model evaluation)

Setelah model selesai dilatih, kita tidak bisa langsung mempercayainya begitu saja. Kita perlu melakukan evaluasi kuantitatif untuk menilai seberapa dekat prediksi model dengan kenyataan. Evaluasi ini dilakukan menggunakan metrik statistik standar.

Implementasi Source Code (Evaluasi Data Latih):

```
#MSE for the train data
pred_results = regressor.evaluate(df_train)
print("The train MSE for the model is: %2f"% pred_results.meanAbsoluteError)
print("The train r2 for the model is: %2f"% pred_results.r2)
```

Kode ini bertujuan untuk mengukur kinerja model terhadap data yang sudah dipelajarinya sendiri (training data). Metode .evaluate(df_train) dijalankan pada objek model, menghasilkan objek ringkasan evaluasi pred_results. Hal yang menarik dan perlu diperhatikan di sini adalah pada baris pencetakan (print). Meskipun label teksnya menuliskan "MSE" (Mean Squared Error), kode sebenarnya memanggil properti .meanAbsoluteError (Mean Absolute Error atau MAE). MAE mengukur rata-rata selisih mutlak antara harga prediksi dan harga asli. Selain itu, kode juga memanggil .r2 (R-squared), sebuah metrik yang menunjukkan seberapa besar variasi data yang bisa dijelaskan oleh model. Kedua angka ini dicetak ke layar untuk memberikan gambaran awal tentang seberapa baik model "menghafal" materi pelatihannya.

Output:

```
The train MSE for the model is: 31526.707719
The train r2 for the model is: 0.610512
```

Hasil output di atas memberikan wawasan numerik yang konkret mengenai kualitas model kita pada tahap pelatihan. Pertama, nilai kesalahan yang tercatat adalah 32.287,15 (ingat analisis kode sebelumnya, ini adalah Mean Absolute Error). Angka ini dapat diinterpretasikan secara harfiah dalam konteks harga rumah: rata-rata prediksi model kita meleset sekitar $32.287 dari harga aslinya. Dalam konteks pasar properti di mana harga rumah bisa mencapai ratusan ribu dolar, selisih 32 ribu dolar mungkin bisa diterima atau dianggap signifikan tergantung pada toleransi bisnis, namun ini memberikan patokan awal tingkat kesalahan kita. Kedua, nilai R-squared (R2) adalah 0.6149 atau sekitar 61,5%. Metrik ini memberi tahu kita tentang "kekuatan penjelasan" model. Artinya, fitur-fitur yang kita gunakan (seperti luas tanah, zonasi, kondisi bangunan) mampu menjelaskan sekitar 61,5% dari faktor-faktor yang mempengaruhi fluktuasi harga rumah. Sisanya, sebesar 38,5%, dipengaruhi oleh faktor lain yang tidak tertangkap oleh model atau data kita. Nilai di atas 0,5 umumnya dianggap sebagai indikator bahwa model telah berhasil menemukan pola yang relevan, meskipun belum sempurna.

Selanjutnya, evaluasi yang jauh lebih kritis dilakukan pada data uji (*test data*).

Implementasi Source Code:

```
#Checking test performance
pred_results = regressor.evaluate(df_test)
print("The test MSE for the model is: %2f"% pred_results.meanAbsoluteError)
print("The test r2 for the model is: %2f"% pred_results.r2)
```

Langkah ini identik secara sintaksis dengan evaluasi sebelumnya, namun memiliki bobot validasi yang jauh lebih berat. Kita memanggil metode .evaluate(df_test) menggunakan df_test, yaitu 25% data yang sengaja disembunyikan dari model selama proses pelatihan. Ini adalah ujian "buta" bagi model. Jika model memiliki performa bagus di sini, berarti model benar-benar memahami pola pasar properti, bukan sekadar menghafal data latihan. Kita kembali mencetak metrik kesalahan (MAE) dan R2 untuk membandingkannya secara langsung dengan hasil pelatihan sebelumnya.

Output:

```
The test MSE for the model is: 33957.332360
The test r2 for the model is: 0.623924
```

Hasil evaluasi pada data uji ini sangat menarik dan memberikan kabar baik mengenai stabilitas model kita. Pertama, nilai kesalahan (Mean Absolute Error) pada data uji adalah 31.668,33. Secara mengejutkan, angka ini justru sedikit lebih rendah (lebih baik) dibandingkan kesalahan pada data latih yang sebesar 32.287. Ini adalah indikasi yang sangat kuat bahwa model kita memiliki kemampuan generalisasi yang sangat baik. Model tidak mengalami overfitting (kondisi di mana model jago di latihan tapi bodoh di ujian); sebaliknya, ia mampu memprediksi harga rumah yang belum pernah dilihatnya dengan tingkat akurasi yang konsisten. Kedua, nilai R2 pada data uji adalah 0.6133, yang hampir identik dengan nilai R2 data latih (0.6149). Konsistensi nilai R2 yang tinggi antara kedua fase ini mengonfirmasi bahwa model kita stabil dan dapat diandalkan. Kesimpulannya, model regresi linear ini layak untuk dideploy karena perilakunya dapat diprediksi dan konsisten, meskipun upaya optimasi lanjutan (seperti hyperparameter tuning) masih bisa dilakukan untuk mencoba menaikkan skor R2 di atas 61%.

Validasi Silang (Cross-validation)

Validasi silang adalah teknik statistik yang digunakan untuk menilai kinerja model ML secara lebih kuat dan objektif dibandingkan sekadar menggunakan satu kali pembagian data latih-uji. Metode ini mengatasi kelemahan dari pemisahan data statis yang mungkin bias karena faktor kebetulan dalam pengambilan sampel acak.

Prinsip dasar validasi silang adalah membagi dataset menjadi beberapa sub-himpunan atau "lipatan" (*folds*). Metode yang paling umum digunakan adalah **k-fold cross-validation**, di mana data dibagi menjadi *k* bagian yang sama besar. Proses pelatihan dan evaluasi dilakukan sebanyak *k* kali. Pada setiap iterasi, satu lipatan berbeda digunakan sebagai data uji (*test set*), sementara sisa lipatan lainnya digabungkan menjadi data latih. Hasil performa dari setiap iterasi kemudian dirata-ratakan untuk mendapatkan estimasi kinerja model yang lebih akurat dan robust. Selain itu, terdapat variasi lain seperti *Stratified cross-validation* yang menjaga proporsi kelas agar tetap seimbang di setiap lipatan, serta *Leave-one-out* yang menggunakan satu poin data sebagai tes, yang sangat akurat namun memakan biaya komputasi tinggi.

Penyetelan Hyperparameter (Hyperparameter tuning)

Penyetelan hyperparameter adalah proses optimasi untuk menemukan konfigurasi terbaik bagi algoritma ML guna meningkatkan kinerjanya. Berbeda dengan parameter model (seperti bobot dalam regresi) yang dipelajari secara otomatis dari data selama pelatihan, **hyperparameter** adalah pengaturan eksternal yang harus ditentukan oleh pengguna sebelum pelatihan dimulai (misalnya, kedalaman pohon keputusan atau laju pembelajaran).

Proses penyetelan ini sangat krusial karena pengaturan default sering kali tidak memberikan hasil optimal untuk dataset tertentu. Terdapat beberapa strategi utama dalam penyetelan hyperparameter:

1. **Grid Search:** Metode ini mendefinisikan kisi (*grid*) nilai potensial untuk setiap hyperparameter, lalu melatih dan mengevaluasi model untuk setiap kombinasi yang mungkin secara menyeluruh. Meskipun akurat, metode ini bisa memakan waktu lama secara komputasi.
2. **Random Search:** Metode ini mengambil sampel nilai hyperparameter secara acak dari distribusi yang ditentukan. Pendekatan ini memungkinkan eksplorasi ruang parameter yang lebih luas dan sering kali menemukan konfigurasi yang efektif dengan lebih cepat dibandingkan pencarian kisi yang kaku.

Penyebaran Model (Model deployment)

Penyebaran model (*deployment*) adalah tahap transisi di mana model yang telah dilatih di laboratorium data dibawa ke lingkungan produksi untuk digunakan secara nyata. Proses ini melibatkan serangkaian langkah teknis untuk memastikan model dapat diakses dan digunakan oleh sistem lain.

Langkah-langkah kunci dalam penyebaran meliputi:

- **Serialisasi dan Persistensi:** Mengubah objek model yang ada di memori menjadi format file (serialisasi) dan menyimpannya ke dalam sistem penyimpanan (persistensi) agar dapat dimuat kembali kapan saja tanpa perlu melatih ulang.
- **Penyajian Model (Model Serving):** Mengekspos model sebagai layanan atau titik akhir API (*API endpoint*). Ini memungkinkan aplikasi eksternal mengirimkan data input baru dan menerima hasil prediksi secara *real-time*.
- **Skalabilitas:** Memanfaatkan kemampuan komputasi terdistribusi seperti Apache Spark untuk menangani volume permintaan prediksi yang besar dan memastikan throughput yang tinggi dalam lingkungan produksi.

Pemantauan dan Manajemen Model (Model monitoring and management)

Tugas seorang praktisi ML tidak berakhir saat model disebar. Pemantauan berkelanjutan (*monitoring*) dan manajemen siklus hidup model sangat penting untuk menjaga kualitas prediksi seiring berjalannya waktu. Tanpa pemantauan, kinerja model cenderung menurun karena perubahan kondisi dunia nyata.

Aspek-aspek kritis dalam pemantauan meliputi:

- **Deteksi Data Drift:** Memantau perubahan statistik pada data input (*data drift*). Jika pola data di lapangan berubah signifikan dibandingkan data yang digunakan saat pelatihan, akurasi model akan merosot. Deteksi dini memungkinkan tim untuk segera melakukan pelatihan ulang (*retraining*).
- **Metrik Kinerja:** Melacak metrik evaluasi (seperti akurasi, presisi, atau AUC) secara berkala pada data produksi untuk mengidentifikasi anomali atau degradasi kinerja.

- **Pembaruan Model:** Menerapkan mekanisme untuk melatih ulang model dengan data terbaru atau memperbarui algoritma guna beradaptasi dengan pola baru.
- **Tata Kelola dan Versi:** Mengelola versi model (*versioning*) dan dokumentasi untuk memastikan reproduksibilitas, pelacakan perubahan, dan kepatuhan terhadap regulasi.

Iterasi dan Perbaikan Model (Model iteration and improvement)

Siklus hidup ML bersifat iteratif, bukan linear. Model yang sudah berjalan di produksi harus terus disempurnakan berdasarkan umpan balik nyata dan data baru untuk mendorong nilai bisnis yang lebih besar.

Strategi untuk perbaikan model meliputi:

- **Analisis Umpan Balik:** Mengumpulkan wawasan dari pengguna akhir dan menganalisis kesalahan prediksi model untuk mengidentifikasi area kelemahan.
- **Eksplorasi Fitur Baru:** Menambahkan fitur baru atau memanfaatkan sumber data eksternal untuk memperkaya informasi yang tersedia bagi model.
- **Eksperimen Algoritma:** Mencoba algoritma yang berbeda atau teknik *Ensemble* (menggabungkan beberapa model seperti *bagging* atau *boosting*) untuk meningkatkan stabilitas dan akurasi prediksi.
- **A/B Testing:** Melakukan eksperimen terkontrol dengan membandingkan kinerja model baru melawan model lama pada sebagian pengguna sebelum peluncuran penuh, untuk memastikan peningkatan yang terukur.

Studi Kasus dan Contoh Dunia Nyata (Case studies and real-world examples)

Penerapan ML telah terbukti memberikan solusi efektif untuk tantangan bisnis yang kompleks di berbagai industri. Dua contoh kasus yang menonjol adalah prediksi churn pelanggan dan deteksi penipuan.

1. Prediksi Churn Pelanggan (Customer churn prediction)

Dalam industri telekomunikasi, kehilangan pelanggan (churn) adalah masalah besar. Perusahaan menggunakan ML untuk menganalisis data demografis, riwayat panggilan, dan keluhan pelanggan. Tujuannya adalah membangun model klasifikasi yang dapat memprediksi pelanggan mana yang berisiko tinggi untuk pindah ke kompetitor. Dengan mengidentifikasi mereka lebih awal, perusahaan dapat mengambil tindakan proaktif seperti menawarkan insentif khusus untuk mempertahankan pelanggan tersebut.

2. Deteksi Penipuan (Fraud detection)

Institusi keuangan memanfaatkan ML untuk melindungi aset nasabah dari aktivitas curang. Model dilatih menggunakan data transaksi historis untuk mempelajari pola perilaku normal dan penipuan. Setelah disebar, model ini memantau transaksi masuk secara real-time, memberikan skor risiko, dan menandai transaksi mencurigakan (anomali) untuk penyelidikan lebih lanjut. Hal ini memungkinkan pencegahan kerugian finansial secara instan dan akurat.

Tren Masa Depan dalam Spark ML dan ML Terdistribusi (Future trends in Spark ML and distributed ML)

Lanskap Spark ML terus berkembang dengan pesat, didorong oleh inovasi teknologi baru. Beberapa tren masa depan yang patut diantisipasi antara lain:

- Integrasi Deep Learning: Spark ML diperkirakan akan semakin terintegrasi dengan kerangka kerja *Deep Learning* populer seperti TensorFlow dan PyTorch. Ini akan mempermudah penggunaan jaringan saraf tiruan untuk tugas kompleks seperti pengenalan gambar dan pemrosesan bahasa alami (NLP) dalam pipeline data besar.
- Automated ML (AutoML): Otomatisasi akan semakin mendominasi, di mana proses rekayasa fitur, pemilihan model, dan penyetelan hyperparameter akan dilakukan secara otomatis. Ini menurunkan hambatan masuk bagi pengguna dan mempercepat waktu pengembangan model.
- Explainable AI (XAI): Tuntutan akan transparansi mendorong pengembangan fitur interpretabilitas dalam Spark ML, memungkinkan pengguna memahami alasan di balik keputusan model demi kepercayaan dan kepatuhan regulasi.
- Generative AI (GenAI): Platform data modern mulai mengadopsi kemampuan *Generative AI* dan *Large Language Models* (LLM) untuk berbagai kasus penggunaan baru.
- Edge Computing & IoT: Spark ML akan memperluas kapabilitasnya ke perangkat *edge* (IoT), memungkinkan pelatihan dan inferensi terdistribusi langsung di sumber data untuk aplikasi yang membutuhkan latensi sangat rendah.

Ringkasan (Summary)

Secara keseluruhan, bab ini telah menyajikan panduan komprehensif mengenai **Spark Machine Learning (ML)** sebagai kerangka kerja yang kuat dan skalabel untuk pemrosesan data cerdas. Kita telah mempelajari konsep dasar ML, perbedaan tipe pembelajaran (supervisi dan tanpa supervisi), serta keunggulan arsitektur terdistribusi Spark dalam menangani *big data*.

Melalui studi kasus prediksi harga rumah, kita telah membedah siklus hidup ML secara mendalam: mulai dari pemuatan data, teknik pembersihan dan rekayasa fitur yang intensif menggunakan Pipeline, hingga pelatihan dan evaluasi model regresi. Kita juga menyoroti pentingnya praktik pasca-pelatihan seperti validasi silang, penyetelan hyperparameter, pemantauan model di produksi, serta iterasi berkelanjutan. Dengan memahami prinsip-prinsip ini dan mengikuti tren teknologi masa depan, organisasi dapat memanfaatkan kekuatan Spark ML untuk mengubah data mentah menjadi wawasan prediktif yang bernilai tinggi dan mendorong kesuksesan bisnis.