Pertemuan 9 — Dari Hadoop ke Spark

Ketika satu mesin tidak lagi cukup: MapReduce, Spark dari dalam, dan *job* pertama di *cluster*

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari UTS ke UAS: kapan satu mesin mentok
2. Ekosistem Hadoop dan sisanya di industri
3. MapReduce: model, *data locality*, tahan gagal
4. Spark dari dalam: *driver*, *executor*, RDD
5. *Narrow* vs *wide*, DAG, *stage*, *lineage*
6. DataFrame dan Spark SQL dasar
7. *Shuffle* sebagai komunikasi *all-to-all*
8. Spark UI: membaca DAG dan *stage*
9. Scala vs PySpark, dan Pandas UDF
10. Praktikum: *cluster* Spark dan *job* silver → gold

Sasaran pertemuan ini

Anda dapat menjelaskan apa yang terjadi di dalam *cluster* ketika sebuah *job* Spark dijalankan, dari *driver* sampai *task*, dan menulis *job* yang sama dalam PySpark dan Scala untuk memindahkan data dari *layer* silver ke gold.

Dari UTS ke UAS: kapan satu mesin mentok

Pipeline UTS sudah jalan. Lalu apa?

Di UTS Anda membangun *pipeline* bronze → silver → gold di **satu mesin** dengan Polars, DuckDB, atau dbt (P5–P6). Untuk puluhan juta baris, itu cepat dan murah.

UAS menaikkan taruhannya

- Data **ratusan juta sampai miliaran** baris: 12 bulan transaksi semua kanal
- Banyak *job* berjalan bersamaan dan harus selesai sebelum pagi
- Mesin boleh mati di tengah jalan tanpa mengulang dari nol
- Hasil akhirnya disajikan sampai *dashboard* atau aplikasi (P13, P16)

Tanda satu mesin mulai mentok

- Data yang dibaca satu *query* melebihi RAM dan *spill* ke disk terus-menerus
- Waktu *job* naik lebih cepat daripada ukuran data
- Satu mesin besar lebih mahal (atau tidak tersedia) dibanding beberapa mesin sedang
- Satu mesin mati = seluruh *pipeline* berhenti

Tetap ukur dulu

Cluster punya *cost*: jaringan, koordinasi, operasi. Pindah ke *cluster* harus dibuktikan dengan angka, itu inti P10.

Tiga gagasan yang membuat *cluster* bekerja

1. Pecah data

File dipecah menjadi blok atau *partition* dan disebar ke banyak mesin (HDFS dan *object storage*, P3).

2. Kirim program ke data

Program (kilobyte) dikirim ke mesin tempat data (gigabyte) berada: *data locality*.

3. Anggap mesin pasti gagal

Kerja dirancang agar bisa diulang otomatis: *re-execution* di MapReduce, *lineage* di Spark.

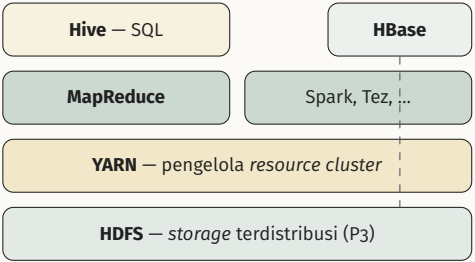
Hitungan kasar (ilustrasi): membaca 1 TB dari satu disk $\approx$ 200 MB/s butuh $\approx$ 83 menit; dari 100 disk sekaligus $\approx$ 50 detik, asal pekerjaannya bisa dipecah dan hasil sebagiannya bisa digabung.

Ringkasan

Hari ini kita membedah **bagaimana** Hadoop dan Spark mewujudkan ketiga gagasan itu; pertemuan depan kita membuatnya **cepat dan terukur**.

Ekosistem Hadoop

Hadoop: satu *stack*, beberapa *layer*



- **HDFS**: *file* dipecah menjadi blok besar dan direplikasi di banyak mesin.
- **YARN**: membagi CPU dan memori *cluster* untuk banyak aplikasi.
- **MapReduce**: model pemrograman *batch* pertama di Hadoop.
- **Hive**: menerjemahkan SQL menjadi *job* di atas data HDFS; menyimpan *schema* tabel di ***Hive Metastore***.
- **HBase**: *wide-column store* di atas HDFS, meniru Bigtable (P7).

Sumber: White, *Hadoop: The Definitive Guide*; Dean dan Ghemawat (2004); Chang dkk., Bigtable (2006).

Apa yang tersisa dari Hadoop di industri hari ini

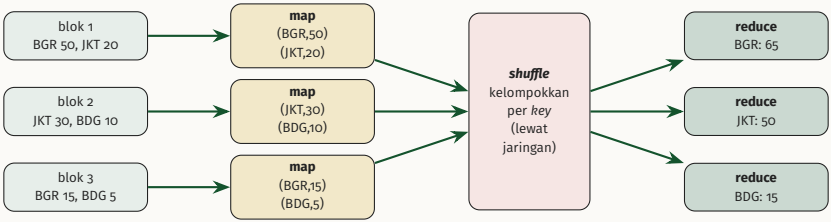
Komponen	Keadaan sekarang	Penggantinya yang umum
HDFS	Masih banyak di *cluster on-premise* besar	*Object storage* (S3, MinIO, Ceph) di *cloud* dan *on-premise* baru
YARN	Menyusut	Kubernetes, atau layanan terkelola (EMR, Dataproc, Databricks)
MapReduce	Hampir tidak dipakai untuk *job* baru	Spark, Flink, mesin SQL
Hive (*query engine*)	Menyusut	Spark SQL, Trino
Hive Metastore	**Masih hidup** sebagai katalog tabel	Katalog REST Iceberg, Nessie, Unity Catalog (P4)
HBase	Bertahan di sistem lama	Cassandra, Bigtable, ScyllaDB (P7)

Ringkasan

Hadoop sebagai produk menyusut, tetapi **gagasannya** (pecah data, pindahkan komputasi ke data, anggap mesin pasti gagal, simpan *schema* di katalog) menjadi dasar semua yang kita pakai hari ini.

MapReduce: model dan eksekusinya

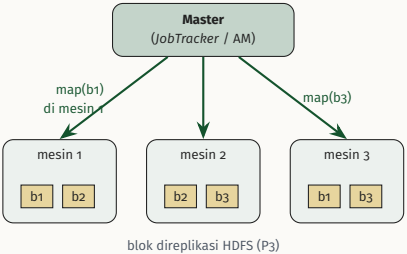
MapReduce dalam satu gambar: total transaksi per cabang



Map membaca blok di mesin tempat blok itu berada dan mengeluarkan pasangan (*key*, nilai). ***Reduce*** menerima semua nilai untuk satu *key*.

Shuffle memindahkan pasangan antar-mesin agar *key* yang sama bertemu. Ini langkah termahal, dan tetap termahal di Spark.

Eksekusi MapReduce: pindahkan program ke data



Sumber: Dean dan Ghemawat, MapReduce (OSDI 2004), bagian 3.

- Master membagi *input* menjadi *split* (biasanya satu blok HDFS) dan membuat satu *map task* per *split*.
- ***Data locality***: master menjadwalkan *map task* di mesin yang **menyimpan** bloknya; bila sibuk, di mesin satu rak; baru terakhir di rak lain.
- Program (kilobyte) yang dikirim, bukan data (gigabyte). Jaringan hanya dipakai berat saat *shuffle*.
- *Output map* ditulis ke disk lokal, dipartisi per *reducer* dengan `hash(kunci) mod R`.

Tahan gagal: ulangi saja, dan jangan menunggu yang lambat

Mesin mati di tengah *job*

- Master memantau *worker* lewat *heartbeat*.
- *Map task* di mesin yang mati **dijalankan ulang** di mesin lain, termasuk yang sudah selesai, karena keluarannya ada di disk lokal mesin itu.
- *Reduce task* yang sudah selesai tidak diulang; keluarannya sudah di HDFS.
- Ini aman karena fungsi *map* dan *reduce* **deterministik** dan tanpa efek samping.

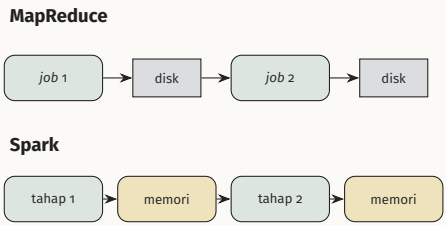
Straggler dan *speculative execution*

- ***Straggler***: *task* yang jauh lebih lambat karena disk rusak, mesin penuh, atau data tidak merata.
- Menjelang akhir *job*, master menjalankan **salinan cadangan** *task* yang tersisa di mesin lain; mana yang selesai lebih dulu dipakai.
- Spark punya *feature* serupa (`spark.speculation`), *default* mati (cek versi).

Pelajaran untuk *pipeline*

Eksekusi ulang otomatis hanya benar bila *task* *idempotent*. Menulis ke *database* eksternal dari dalam *task* bisa menggandakan data.

Kelemahan MapReduce, dan mengapa Spark menggantikannya

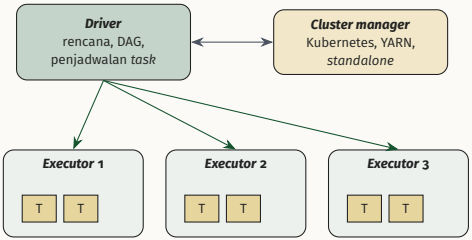


Sumber: Zaharia dkk., RDD (NSDI 2012).

- Setiap *job* MapReduce menulis hasil antara ke HDFS (disk, direplikasi). Kerja bertahap dan iteratif (*machine learning*, *graph*) menjadi sangat lambat.
- Hanya dua operasi: *map* dan *reduce*. *Join* dan *aggregation* bertingkat harus dirangkai manual.
- Spark merangkai banyak tahap dalam satu DAG, menyimpan hasil antara di memori bila muat, dan menyediakan API tingkat tinggi (DataFrame, SQL).

Apache Spark: arsitektur dan cara kerjanya

Arsitektur aplikasi Spark



T = *task*, satu per *partition* data

- ***Driver***: menjalankan program Anda, mengubahnya menjadi rencana, membaginya menjadi *task*.
- ***Executor***: proses JVM di tiap mesin yang menjalankan *task* dan menyimpan data di memori.
- ***Partition***: potongan data; satu *partition* diolah satu *task*. Jumlah *partition* menentukan paralelisme.
- ***Cluster manager***: meminjamkan *resource*. Di industri modern sering Kubernetes.

Transformation, *action*, dan DAG

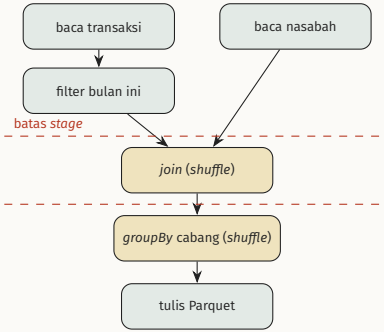
Transformation bersifat *lazy*

`filter`, `select`, `join`, `groupBy`: tidak ada yang dijalankan, Spark hanya mencatat rencananya.

Action memicu kerja

`count`, `show`, `collect`, `write`: Spark mengoptimalkan seluruh rencana, lalu menjalankannya.

Keuntungan *lazy*: Spark melihat seluruh rangkaian sebelum bekerja, sehingga bisa membuang kolom dan baris yang tidak dibutuhkan sejak awal.



Setiap *shuffle* memotong DAG menjadi ***stage***; tiap *stage* terdiri dari banyak ***task***.

RDD: fondasi di bawah DataFrame

RDD (*Resilient Distributed Dataset*): kumpulan data yang

- **terpartisi** ke banyak *executor*,
- **tidak dapat diubah** (*immutable*): setiap *transformation* menghasilkan RDD baru,
- **mengingat asal-usulnya**: dari RDD mana dan dengan fungsi apa ia dibuat.

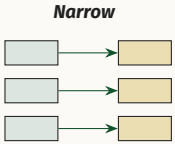
DataFrame dan Spark SQL dikompilasi menjadi operasi atas RDD. Anda jarang menulis RDD langsung, tetapi konsepnya menjelaskan *stage*, kegagalan, dan *cache*.

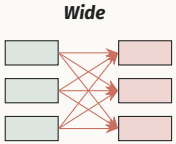
Lima hal yang mendefinisikan RDD

1. Daftar *partition*
2. Fungsi untuk menghitung tiap *partition*
3. Daftar *dependency* ke RDD induk
4. (Opsional) *partitioner*, misalnya *hash*
5. (Opsional) lokasi yang disukai, *data locality* seperti MapReduce

Sumber: Zaharia dkk., NSDI 2012, bagian 4.

Narrow vs wide dependency



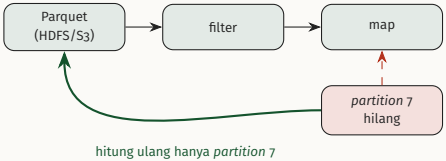


- ***Narrow***: tiap *partition* anak bergantung pada **satu** *partition* induk. Spark menggabungkannya menjadi satu *task* berurutan (*pipelining*), tanpa jaringan.
- ***Wide***: tiap *partition* anak bergantung pada **banyak** *partition* induk. Butuh *shuffle*, dan menjadi **batas *stage***.
- Satu *stage* = rangkaian *narrow* terpanjang. Satu *task* = satu *stage* untuk satu *partition*.

Ringkasan

Menghitung *wide dependency* di kode Anda sama dengan menghitung *shuffle*, dan biasanya sama dengan menghitung di mana waktu habis.

Lineage: pulih tanpa menyalin data



- Bila *executor* mati, *partition* di memorinya hilang.
- Spark tidak menyimpan salinan; ia **menghitung ulang** *partition* yang hilang dari induknya, mengikuti ***lineage*** sampai ke data sumber yang tahan lama.
- Untuk *narrow dependency*, cukup *partition* induk yang bersangkutan. Untuk *wide*, Spark memakai *file shuffle* yang tersimpan di disk bila masih ada; bila tidak, *stage* sebelumnya diulang.
- *Lineage* sangat panjang (iterasi ratusan kali) membuat pemulihan mahal: putuskan dengan `checkpoint()` ke *storage* tahan lama.

Sumber: Bandingkan dengan *replication* (Sistem Terdistribusi P7): *replication* membayar di depan, *lineage* membayar saat gagal.

`cache` dan `persist`: kapan menyimpan hasil antara

```
from pyspark import StorageLevel
bersih = (trx.filter(F.col("status") == "SUKSES")
          .join(F.broadcast(cabang), "id_cabang"))
bersih.persist(StorageLevel.MEMORY_AND_DISK)

h = bersih.groupBy("tanggal").agg(F.sum("nominal"))
c = bersih.groupBy("cabang").agg(F.count("*"))
h.write.parquet("s3a://gold/harian/") # hitung+simpan
c.write.parquet("s3a://gold/cabang/") # pakai simpanan
bersih.unpersist()
```

- Tanpa `persist`, `bersih` dihitung **dua kali**: sekali untuk tiap *action*, karena Spark *lazy*.
- `cache()` = `persist()` dengan tingkat *default*.
- `MEMORY_AND_DISK`: *partition* yang tidak muat di memori ditumpahkan ke disk.
- *Cache* hanya berguna bila hasil antara **dipakai lebih dari sekali**; selebihnya hanya memakan memori.

Jangan lupa

`unpersist()` setelah selesai, dan periksa tab *Storage* di Spark UI.

DataFrame dan Spark SQL

DataFrame dan Spark SQL: dua wajah, satu mesin

```python
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

spark = SparkSession.builder.getOrCreate()
trx = spark.read.parquet("s3a://bronze/trx/")
nsb = spark.read.parquet("s3a://bronze/nasabah/")

harian = (trx
  .filter(F.col("tanggal") >= "2026-10-01")
  .join(nsb, "id_nasabah")
  .groupBy("tanggal", "cabang")
  .agg(F.sum("nominal").alias("total"),
       F.count("*").alias("jumlah")))

(harian.write.mode("overwrite")
   .partitionBy("tanggal")
   .parquet("s3a://gold/harian/"))
```

```sql
SELECT t.tanggal, n.cabang,
       SUM(t.nominal) AS total,
       COUNT(*)       AS jumlah
FROM   trx t
JOIN   nasabah n USING (id_nasabah)
WHERE  t.tanggal >= DATE '2026-10-01'
GROUP  BY t.tanggal, n.cabang;
```

Ringkasan

Kode Python dan SQL di atas menghasilkan ***physical plan yang sama***. Keduanya hanya deskripsi; yang menjalankan adalah *optimizer* Catalyst (P10) dan *executor* JVM.

Shuffle: komunikasi *all-to-all*

Shuffle: *cost* terbesar dalam *job batch*

Shuffle terjadi ketika baris dengan *key* yang sama harus berkumpul di *partition* yang sama: `groupBy`, `join`, `distinct`, `orderBy`, `repartition`.

Mengapa mahal

- Data ditulis ke disk lokal, lalu dikirim lewat jaringan
- *Serialization* dan *deserialization* setiap baris
- Tahap berikutnya menunggu semua *task* tahap sebelumnya selesai

Cara menekan *shuffle*

- Filter dan pilih kolom **sebelum** *join*
- *Aggregation* sebagian sebelum *shuffle* (Spark melakukannya untuk `sum`, `count`)
- *Broadcast* tabel kecil alih-alih mengocok tabel besar
- Atur `spark.sql.shuffle.partitions` (*default* 200) atau biarkan AQE menyesuaikan

Shuffle adalah komunikasi *all-to-all*: setiap *task* pengirim bicara dengan setiap *task* penerima. Bandingkan dengan *collective* MPI di Sistem Terdistribusi P11.

Spark UI

Spark UI: tempat mencari bukti

Tab	Yang dicari
Jobs	*Action* mana yang lama
Stages	Durasi *task* (median vs maks), *shuffle read/write*
SQL / DataFrame	Rencana dalam bentuk grafik, baris per operator
Storage	Data yang di-*cache*
Executors	Memori, waktu GC, *task* gagal

Tiga pertanyaan diagnosis

1. Di *stage* mana waktu habis?
2. Apakah ada *task* yang jauh lebih lama dari median (*skew*)?
3. Berapa byte yang dikocok, dan apakah bisa dikurangi?

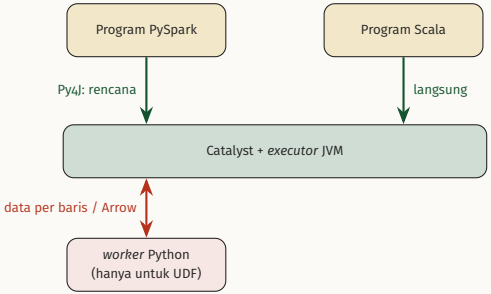
Spark UI berjalan di `http://localhost:4040` selama aplikasi hidup; pasang *history server* untuk melihat *job* yang sudah selesai.

Ringkasan

Jangan menebak mengapa *job* lambat. *Physical plan* dan Spark UI hampir selalu sudah menunjukkan jawabannya.

Scala vs PySpark

Satu mesin JVM, dua bahasa pemanggil



- Dengan DataFrame dan fungsi bawaan (`F.sum`, `F.when`), PySpark **hanya mengirim rencana**; kerja berat tetap di JVM. Kecepatannya praktis sama dengan Scala.
- Perbedaan muncul saat memakai **UDF Python**: data harus keluar dari JVM ke proses Python dan kembali.
- Scala memberi ***Dataset*** bertipe: kesalahan nama kolom tertangkap saat kompilasi, bukan saat *job* sudah berjalan satu jam.

UDF biasa vs Pandas UDF berbasis Arrow

```python
# UDF biasa: satu baris per panggilan,
# setiap nilai diserialisasi (pickle)
@F.udf("double")
def zakat_udf(saldo):
    return saldo * 0.025 if saldo >= NISAB else 0.0

# Pandas UDF: satu batch per panggilan,
# dipindah dalam format kolumnar Arrow
import pandas as pd
@F.pandas_udf("double")
def zakat_vec(saldo: pd.Series) -> pd.Series:
    return (saldo * 0.025).where(saldo >= NISAB, 0.0)
```

Urutan pilihan dalam praktik

1. **Fungsi bawaan Spark** (`F.when`, `F.expr`): dioptimalkan Catalyst, tanpa keluar JVM
2. **Pandas UDF** bila logika butuh *library* Python
3. **UDF Scala** bila butuh logika khusus dan kinerja
4. UDF Python biasa: hindari pada data besar

Contoh zakat 2,5% dengan nisab hanya ilustrasi pemrograman, bukan aturan perhitungan zakat yang lengkap.

Job yang sama dalam Scala

```scala
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

case class Trx(idNasabah: Long, tanggal: java.sql.Date,
               nominal: BigDecimal)

object Harian {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder.getOrCreate()
    import spark.implicits._
    val trx = spark.read
      .parquet("s3a://bronze/trx/").as[Trx]
    val nsb = spark.read.parquet("s3a://bronze/nasabah/")
    trx.filter($"tanggal" >= "2026-10-01")
       .join(broadcast(nsb), "idNasabah")
       .groupBy("tanggal", "cabang")
       .agg(sum("nominal").as("total"))
       .write.mode("overwrite")
       .parquet("s3a://gold/harian/")
  }
}
```

- `case class Trx` memberi *schema* bertipe: salah ketik `nominl` gagal saat kompilasi.
- Dikemas dengan `sbt package`, dijalankan dengan `spark-submit`.
- Rencana fisiknya sama dengan versi Python; bandingkan dengan `explain()`.

Kapan memilih Scala

Library atau *connector* internal ditulis untuk JVM, UDF berkinerja tinggi, tim platform yang memelihara *job* kritis bertahun-tahun.

Praktikum

Praktikum: *cluster* Spark kecil dengan Docker Compose

```
services:
  minio:
    image: minio/minio
    command: server /data --console-address ":9001"
    ports: ["9000:9000", "9001:9001"]
  spark-master:
    image: apache/spark:3.5.1-python3
    command: /opt/spark/bin/spark-class
      org.apache.spark.deploy.master.Master
    ports: ["7077:7077", "8080:8080"]
  spark-worker:
    image: apache/spark:3.5.1-python3
    command: /opt/spark/bin/spark-class
      org.apache.spark.deploy.worker.Worker
      spark://spark-master:7077
    environment:
      SPARK_WORKER_CORES: "2"
      SPARK_WORKER_MEMORY: "2g"
    deploy: { replicas: 2 }
```

- Satu *master* (*cluster manager* mode *standalone*) dan dua *worker*; MinIO menyimpan *layer* bronze, silver, gold dari UTS.
- UI *master* di `localhost:8080`, Spark UI aplikasi di `localhost:4040`.
- Akses MinIO dari Spark lewat konektor `s3a` (paket `hadoop-aws`); *configuration* lengkap ada di repositori kelas.

Tag *image* dan nama perintah dapat berubah antar versi; ikuti *file* yang dibagikan di kelas.

Praktikum: *job* silver → gold

```
# gold_harian.py -- dijalankan dengan spark-submit
spark = (SparkSession.builder
         .appName("gold-harian").getOrCreate())
trx = spark.read.parquet("s3a://silver/transaksi/")
cab = spark.read.parquet("s3a://silver/cabang/")

gold = (trx.filter(F.col("status") == "SUKSES")
          .join(F.broadcast(cab), "id_cabang")
          .groupBy("tanggal", "wilayah", "cabang")
          .agg(F.sum("nominal").alias("total"),
               F.countDistinct("id_nasabah")
                .alias("nasabah_aktif")))
(gold.write.mode("overwrite")
     .partitionBy("tanggal")
     .parquet("s3a://gold/harian_cabang/"))
```

```
spark-submit --master spark://spark-master:7077 \
  --total-executor-cores 4 gold_harian.py
```

1. Jalankan *job* di atas dengan data sintetis 10 juta baris (naikkan bila laptop sanggup).
2. Tulis ulang dalam Scala (`sbt package`, lalu `spark-submit --class`); bandingkan `explain()` kedua versi.
3. Matikan satu *worker* (`docker kill`) saat *job* berjalan. Amati di Spark UI: *task* mana yang diulang?

Anotasi DAG

Tangkap layar tab SQL dan Stages. Tandai: setiap *stage*, setiap `Exchange` (*shuffle*), dan apakah *join* memakai *broadcast*.

Kaitan dengan mata kuliah Sistem Terdistribusi

Hari ini	Sistem Terdistribusi	Sudut pandangnya
Lineage sebagai cara pulih	P7 (14 Nov): *replication*	Membayar di depan dengan salinan vs membayar saat gagal dengan hitung ulang
Shuffle mahal	P11 (Sabtu 12 Des): *collective* MPI	*All-to-all* dan model *communication cost*
Executor JVM, GC	P10 (Sabtu 5 Des): manajemen memori	Mengapa jeda GC terasa di *task* panjang

Ringkasan

Hadoop, MapReduce, dan Spark, dari pemakaian sampai cara kerja di dalam, sepenuhnya dibahas di mata kuliah ini. Sistem Terdistribusi hanya menyebutnya sebagai salah satu pola paralel.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: DAG dan *stage* yang dianotasi, perbandingan `explain()` PySpark vs Scala, dan hasil uji mematikan *worker*.
2. Hitung berapa *wide dependency* di *job* Anda dan cocokkan dengan jumlah `Exchange` di *physical plan*.
3. Siapkan generator data sintetis yang bisa diskalakan (misalnya 50 juta, 200 juta, 500 juta baris), dipakai untuk eksperimen *scaling* pertemuan depan.
4. Baca Chambers dan Zaharia, *Spark: The Definitive Guide*, Bab 15–19.

Pertemuan depan

Spark skala besar: Catalyst dan AQE, strategi *join* dan *skew*, *partition* dan ukuran *file*, menulis ke Iceberg/Delta, *tuning*, lalu membuktikan *strong* dan *weak scaling* dengan pengukuran, *HPC data* untuk UAS.

Terima kasih

Pertanyaan?