Sesi 23
Spark (1)

Spark dan Resilient Distributed Dataset (RDD)

Spark (1)

- Kecepatan
  - Komputasi dalam memori proses & respon lebih cepat
  - Lebih cepat > Map Reduce proses lebih kompleks di disk
- Generalitas (Keumuman)
  - Berbagai workload dalam 1 sistem
  - Batch app (contoh MapReduce, algoritma iterasi)
  - Interactive query dan streaming data
- Kemudahan penggunaan
  - API untuk Scala, Python, dan Java
  - Library SQL, machine learning, streaming, proses graph, dll
  - Berjalan di Hadoop cluster (YARN atau Apache Mesos) & standalone

Spark (2)

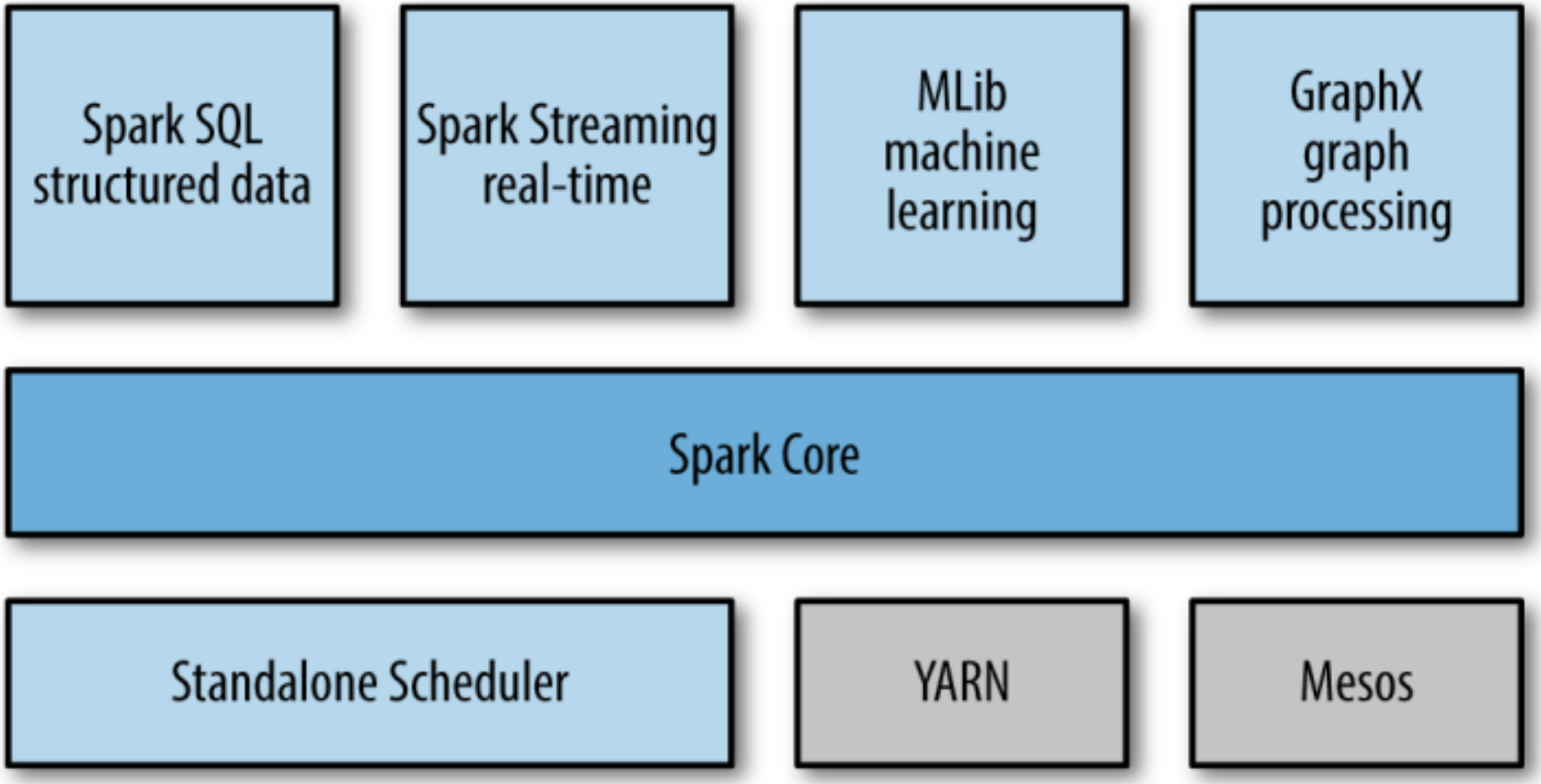
- ≅ MapReduce →support proses distribusi paralel, fault tolerance pada commodity hardware, skalabilitas, dll
- Low-latency (memory process), high level API, dan kumpulan high level tool
- Pengguna Spark: Data scientist dan Engineer

  ➢Data Scientist
  Analisis dan permodelan data → insight dengan analisi ad-hoc
  Transform data → usable format
  Statistik, machiner learning, dan SQL (Python, Matlab, R)

Spark (3)

- **Engineer**
  Develop data processing, web app , dll
  implementasi business case
  Monitor, inspect, dan tune app
  Program dengan Spark API
- **Umum**
  Mudah digunakan
  Fungsionalitas luas
  Mature dan reliable

Spark Unified Stack



Spark Unified Stack – Detail (1)

- **Spark Core** → pusat, general-purpose system → scheduling, distributing, monitoring app pada cluster-cluster
- Komponen-komponen di atas core → didesain untuk saling beroperasi, bisa dikombinasikan (seperti library pada program/project)
- Keuntungan stack → layer tinggi mewarisi (inherit) peningkatan dari layer rendah. Contoh: Optimasi Spark core → speedup library SQL, streaming, machine learning dan proses graph
- Spark core didesain scale up 1-1000 node
- Berjalan di berbagai cluster manager, Hadoop YARN dan Apache Mesos DAN
  Berjalan standalone dengan built-in scheduler
- **Spark SQL** → intermix SQL dengan Python, Scala, dan Java
- **Spark streaming** → processing live stream data

Spark Unified Stack – Detail (2)

- Spark streaming dekat dengan Spark Core API → mudah pindah antar aplikasi yang proses data tersimpan di memori dengan yang datang dalam waktu real-time
- Reliabilitas ≅ Spark Core
- Machine Learning (MLlib) ← algoritma ML → didesain juga untuk scale out antar cluster
- GraphX → API manipulasi graph dan komputasi graph-parallel

Sejarah Spark (1)

- 2010 – Spark white paper
- 2014 – Apache Spark top-level

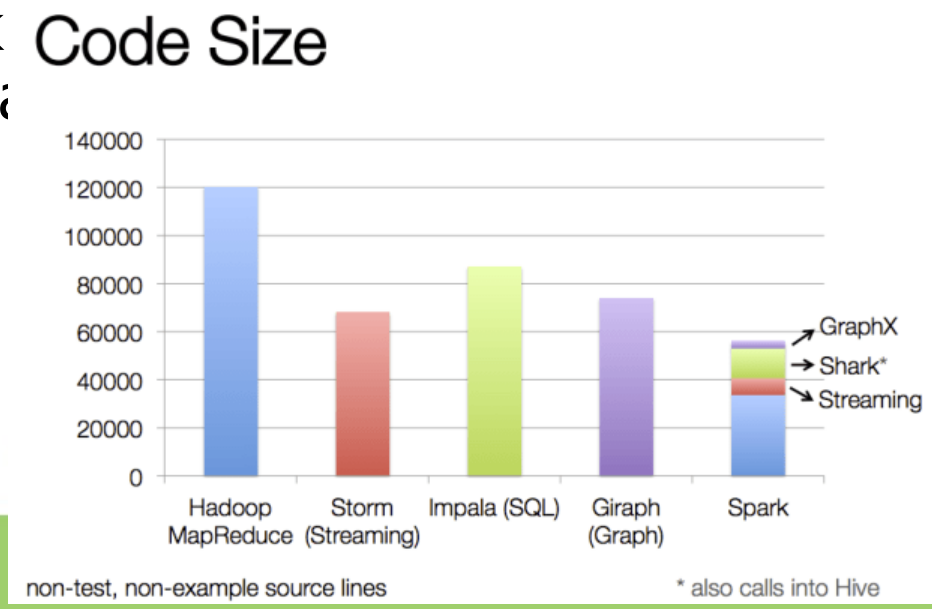
Framework Spark mirip dengan MapReduce (data processing, fault tolerance pada commodity network)

Map Reduce mulai dengan sistem general batch processing
Dua batasan:

- Kesulitan dalam pemrograman MapReduce
- Batch tidak cocok untuk semua use case case lain butuh specialized system (Storm, Impale, Graph, dll)
  MapReduce x Third party app banyak overhead

Sejarah Spark (2)

- Code size Spark + libraries (SQL, Graph, dll) < Map Reduce, Stream, SQL, Graph secara terpisah
- Libraries code hanya menambah sedikit dari total kode Spark
  Overhead Spark < overhead third party MapReduce
- Dimungk ed Stack terintegr

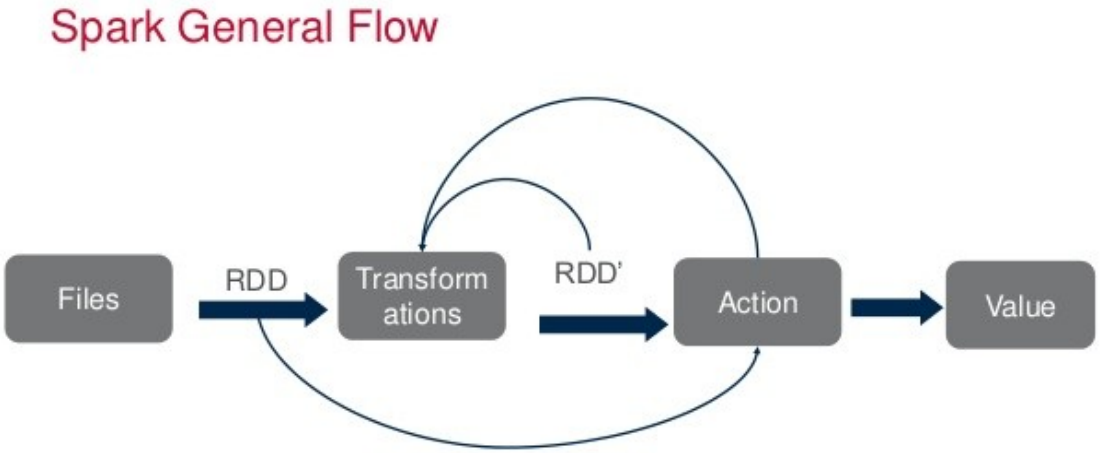


Resilient Distributed Dataset (RDD) (1)

- Spark primary core abstraction
- Kumpulan elemen terdistribusi (dataset internal dan eksternal)
- Diparalelkan ke seluruh cluster
- Dua jenis operasi RDD
  - Transformation
    tidak punya return value, hanyar return pointer ke RDD
    hanya buat dan update DAG
    tidak ada evaluasi saat definisi (hanya @runtime) lazy evaluation
  - Action
    punya return value
    transformation dievaluasi dan action panggil RDD
    Contoh: menghitung jumlah element

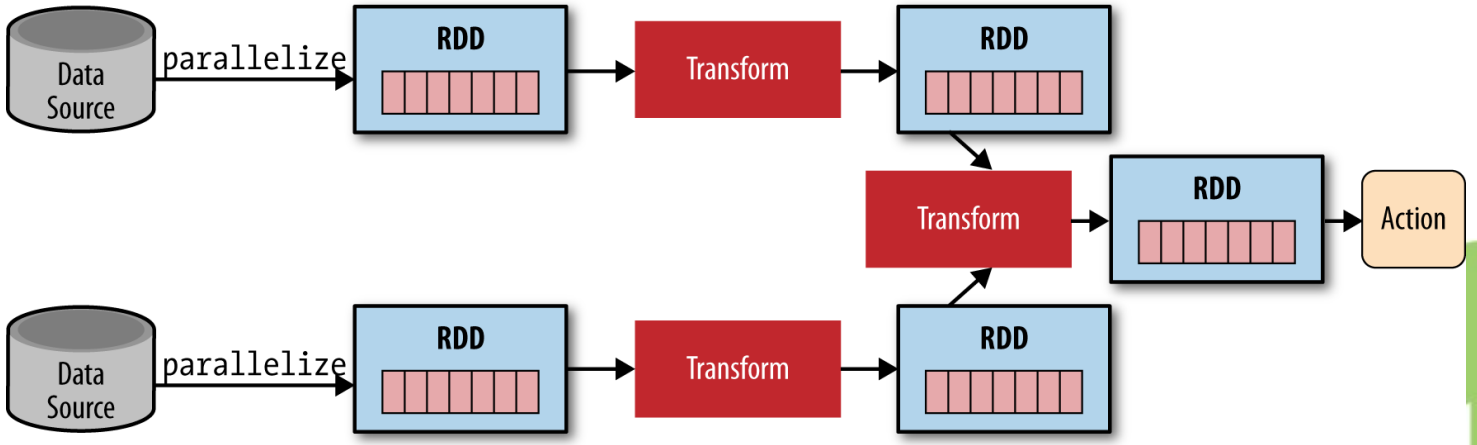
Resilient Distributed Dataset (RDD) (2)

- Fault tolerance reconstruct transformation buat lineage (track) untuk mendapat data yang hilang
- Proses: Data di-load dari Hadoop transformation (filter, map, reduce) Action (ketika dipanggil) DAG di-update setiap action



Resilient Distributed Dataset (RDD) (3)

- Proses Fault tolerance: jika 1 node offline ketika online, graph dievaluasi ulang di mana posisi terakhir
- Caching disediakan processing di memori jika memori tidak cukup pakai disk



Resilient Distributed Dataset (RDD) (4)

- Immutable (permanen)
- Tiga metode membuat RDD
  - Paralelisasi koleksi yang ada
    data yang sudah ada di Spark beroperasi secara paralel
    Contoh: data array, RDD dibuat dengan panggil metode paralelisasinya return pointer ke RDD dataset terdistribusi secara paralel di seluruh cluster
  - Referensi ke sebuah dataset (HDFS, Cassandra, Hbase, Amazon, S3, dll)
  - Transformasi dari RDD yang ada
    Hasil metode pertama filter RDD baru
- Support text, SequenceFile, Hadoop file format

Download dan Pasang Spark

- Spark berjalan di Windows dan UNIX-like system (Linux, MacOS X)
- Standalone, butuh Jawa
- Download pre-built untuk Hadoop: https://spark.apache.org/downloads.html
  Instal dan pasang di tiap node pada cluster
- Start cluster manual execute `./sbin/start-master.sh`
- Master print out `spark://HOST:PORT URL` connect worker
  default master web UI http://localhost:8080

*Informasi Lanjut
http://spark.apache.org/latest/spark-standalone.html

Spark Job dan Shell

- Spark Job Scala, Python, Java
- Spark shell Scala dan Python
- API tersedia untuk Scala, Python, Java
- Sesuaikan dengan versi Spark
- Spark native Scala (umumnya)
- Jawa 8 Lambda support gap Java dan Scala

Scala

- Semua object (termasuk primitive di Jawa, seperti int, boolean dan function)
- Angka object
  Contoh: 1+2 * 3 / 4 (1)+(((2)*(3))/(4))
- Function object
  pass function sebagai argumen
  simpan ke variabel
  return dari function lain
- Function declaration
  `def functionName (list of parameter):[return type]`

Scala –Fungsi Anonymous

- Sangat umum di Spark
- Fungsi tanpa nama yang dipanggil sekali saja
- Digunakan untuk passing ke fungsi lain
- Cukup argumen, panah kanan, dan body fungsinya setelah panah

Spark Scala dan Python Shell (1)

- Mudah mempelajari Spark API
- Tool powerful analisis data interaktif
- Scala shell Java VM menggunakan library Java yang ada
- Scala
  - Run Scala shell - `./bin/spark-shell`
  - Read text file/Create RDD dari dataset eksternal –
    ```
    scala> val textFile = sc.textFile("README.md")
    ```
  - Paralelisasi data (create RDD) – sc SparkContext
    ```
    val distData = sc.parallelize(data)
    ```
  - Additional transformation dan action –
    ```
    distData.filter(...)
    ```

Spark Scala dan Python Shell (2)

- Loading file (HDFS) –
```
val lines = sc.textFile("hdfs://data.text")
```
- Applying transformation – contoh map length tiap line
```
val lineLengths = lines.map(s => s.length)
```
- Invoke/call action – contoh reduce total length
```
val totalLengths = lineLengths.reduce((a,b) => a + b)
```
return total length ke caller

- MapReduce –
```
val wordCounts = textFile.flatMap(line => line.split (" "))
.map(word => (word, 1))
.reduceByKey((a,b) => a + b)
wordCounts.collect()
```
Word count → split file by words → map tiap word ke key-value pair → word = key, value = 1 → reduce key ≈ sum value tiap key = jumlah tiap kata → collect() return jumlah kata

Spark Scala dan Python Shell (3)

- Python
  Run Python shell - `./bin/pyspark`
  Read text file - `>>> textFile = sc.textFile("README.md")`

Aktivitas Kelas

- Mengunduh dan memasang Spark standalone
- Memasang Spark pada cluster
- Menjalankan Spark Scala dan Python shell

Direct Acyclic Graph (DAG)

- Graph business logic – bagian transformasi
- Untuk melihat DAG sebuah RDD setelah beberapa transformasi debug String – `linesLength.toDebugString`
- Contoh DAG (baca bottom up)

```
res5: String =
MappedRDD[4] at map at <console>:16 (3 partitions)
  MappedRDD[3] at map at <console>:16 (3 partitions)
    FilteredRDD[2] at filter at <console>:14 (3 partitions)
      MappedRDD[1] at textFile at <console>:12 (3 partitions)
        HadoopRDD[0] at textFile at <console>:12 (3 partitions)
```

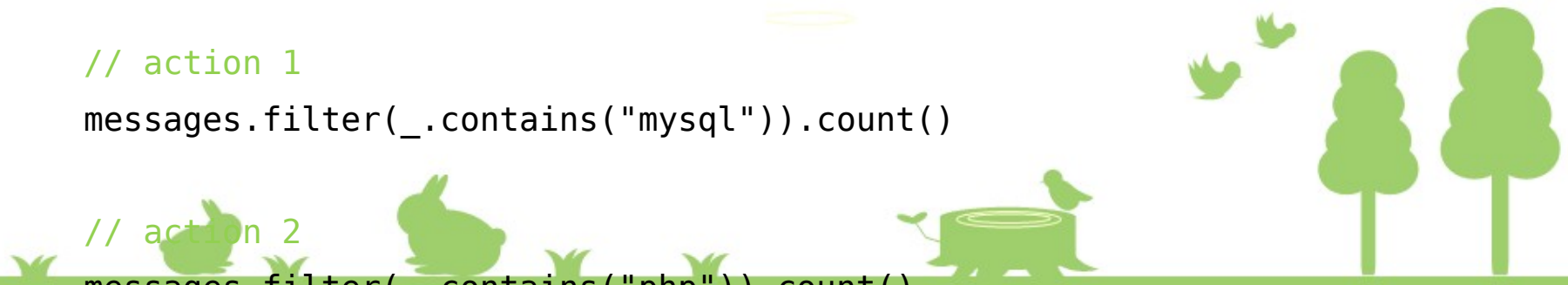
- Fault tolerance – jika 1 nodee offline saat offline ambil copy dari node terdekat dan rebuild graph terakhir sebelum offline

Action pada DAG

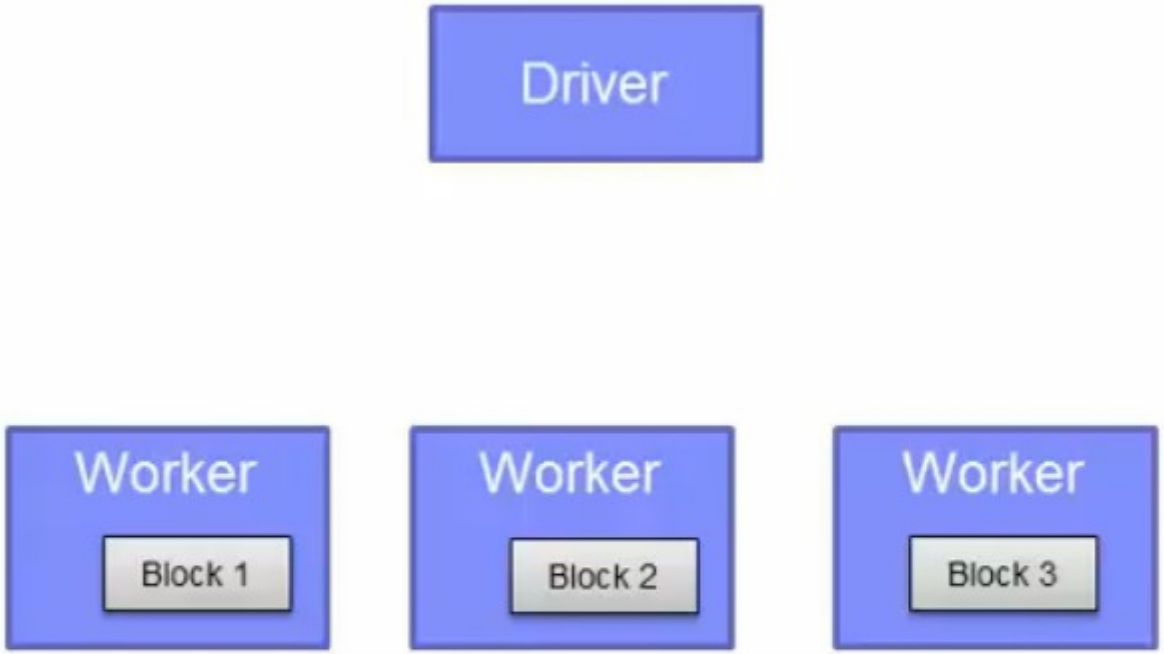
- Data dipartisi ke blocks di seluruh cluster
- Driver kirim code ke setiap block (Code berisi transformation dan action ke worker node)
- Executor di tiap worker lakukan task di setiap block
- Executor read HDFS block data diparalelkan
- Cache dibuat untuk menyimpan hasil partial di memori

Contoh

```
// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132
// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

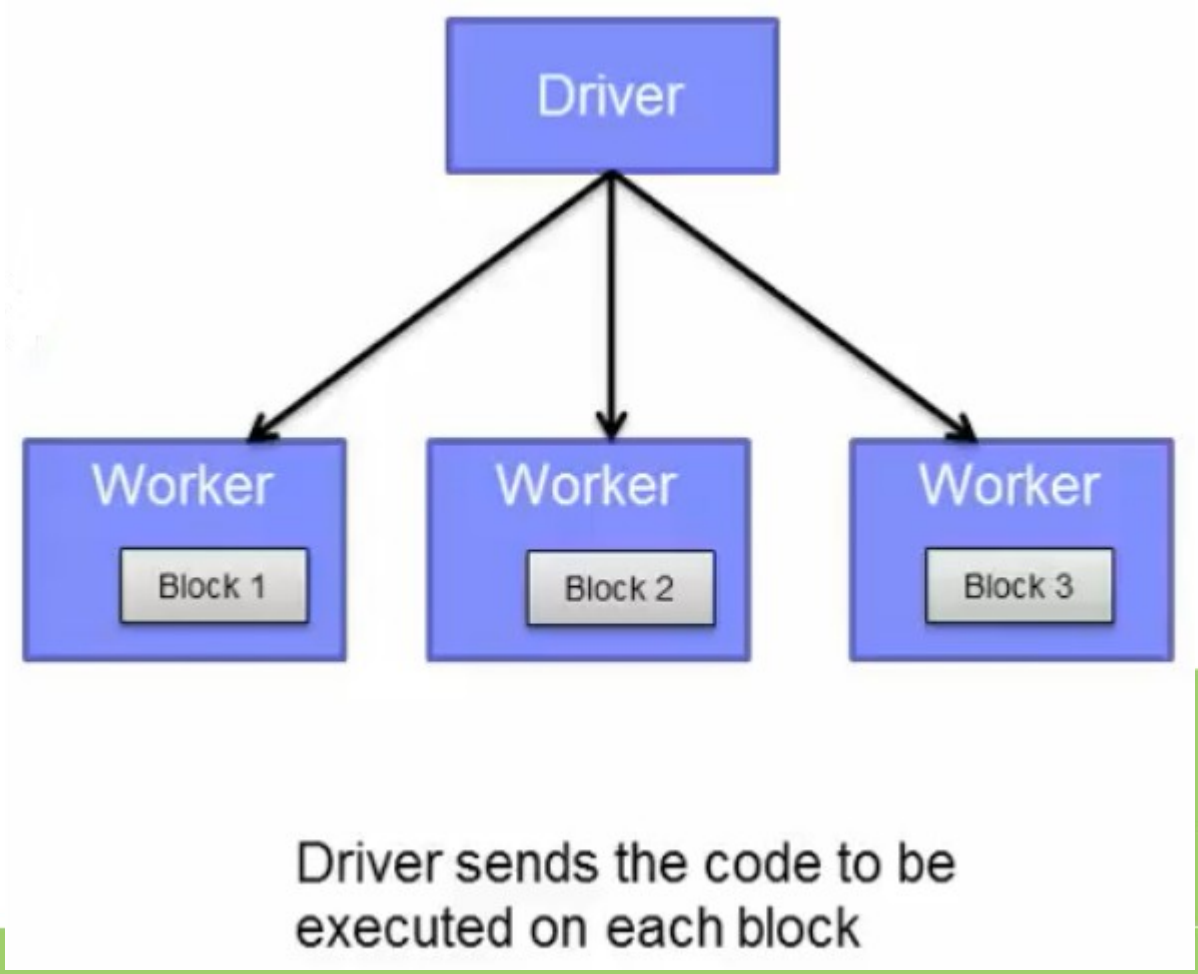


```
val lines = sc.textFile("hdfs://...")
```

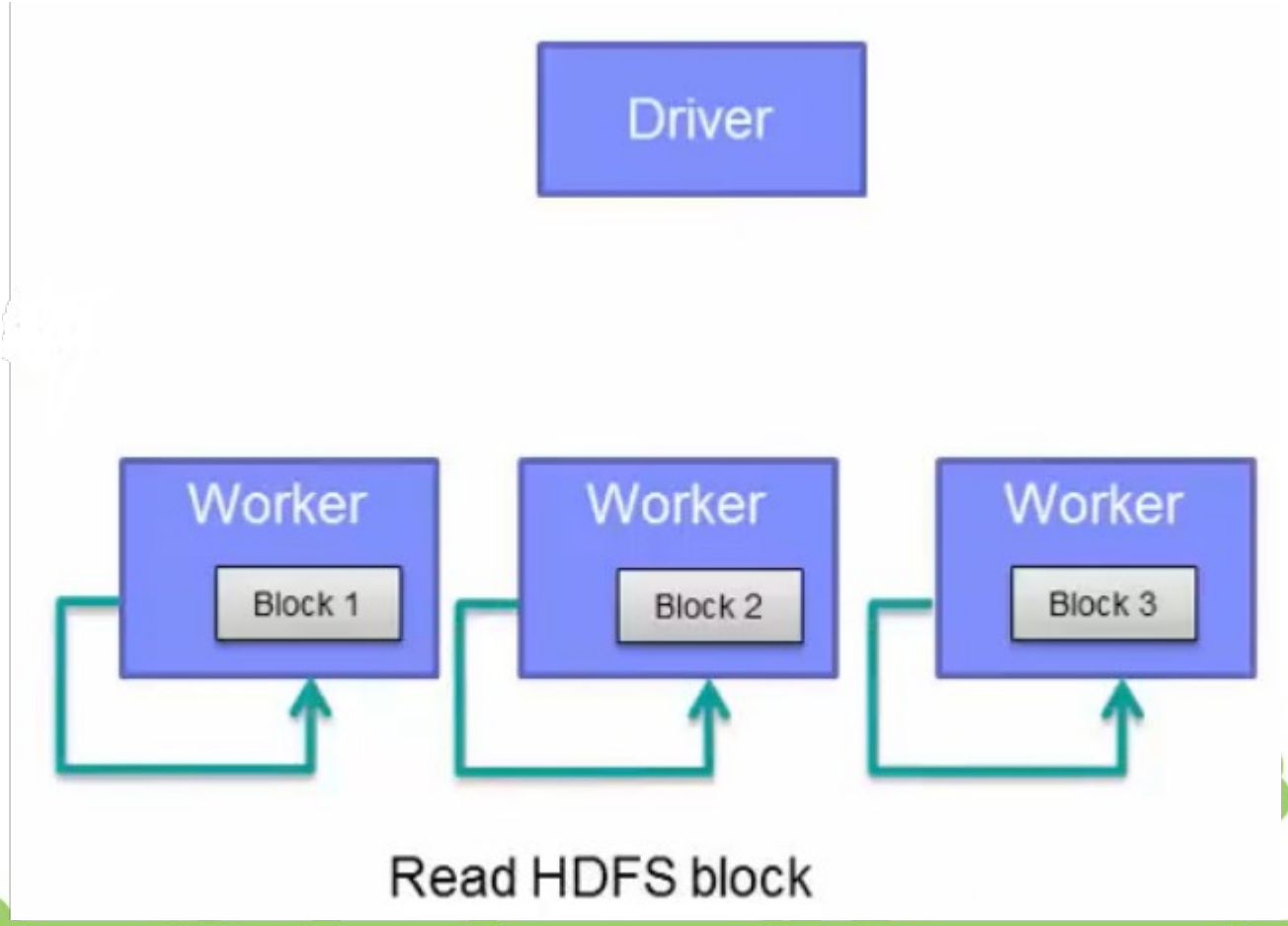


The data is partitioned into different blocks

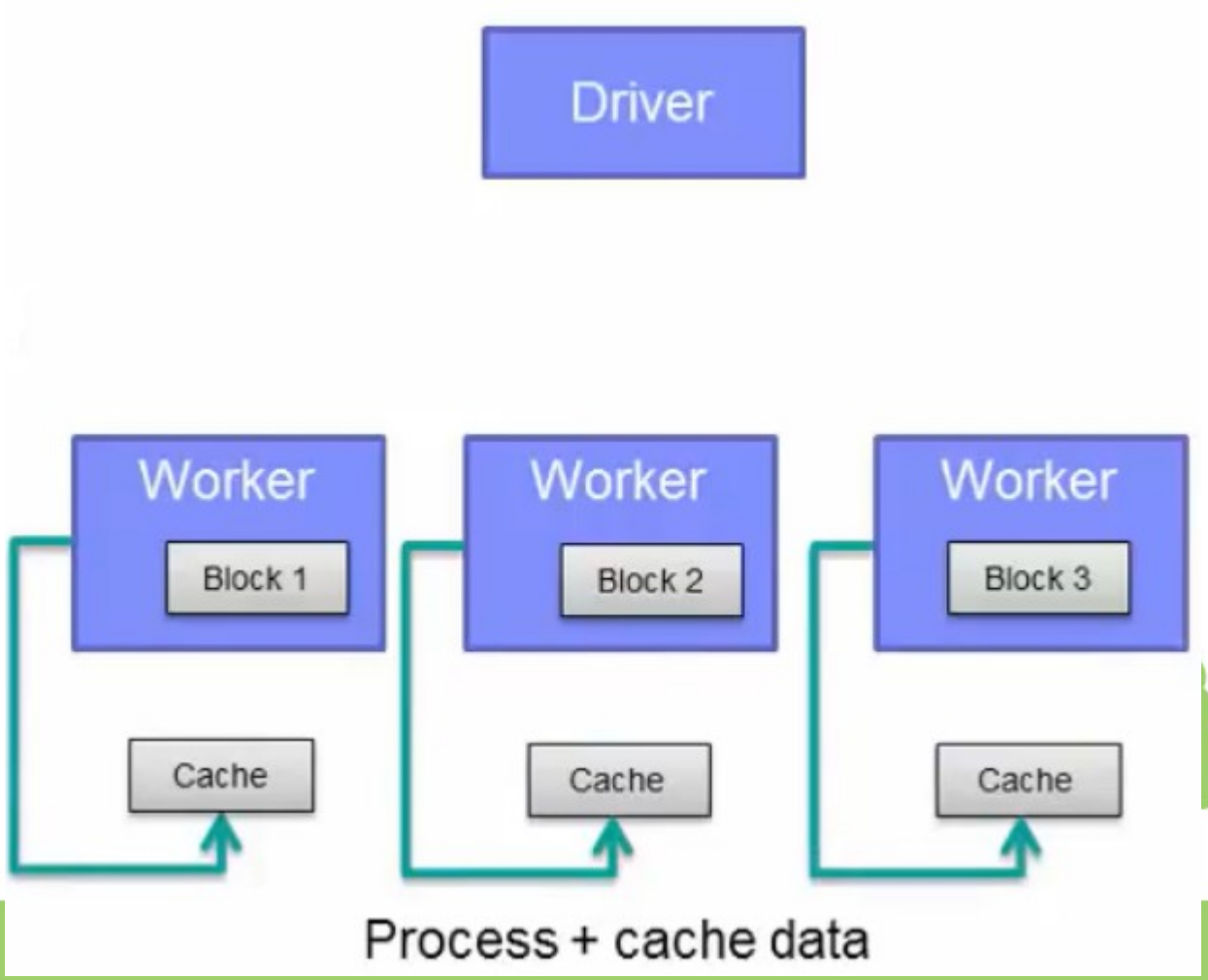
```
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
```



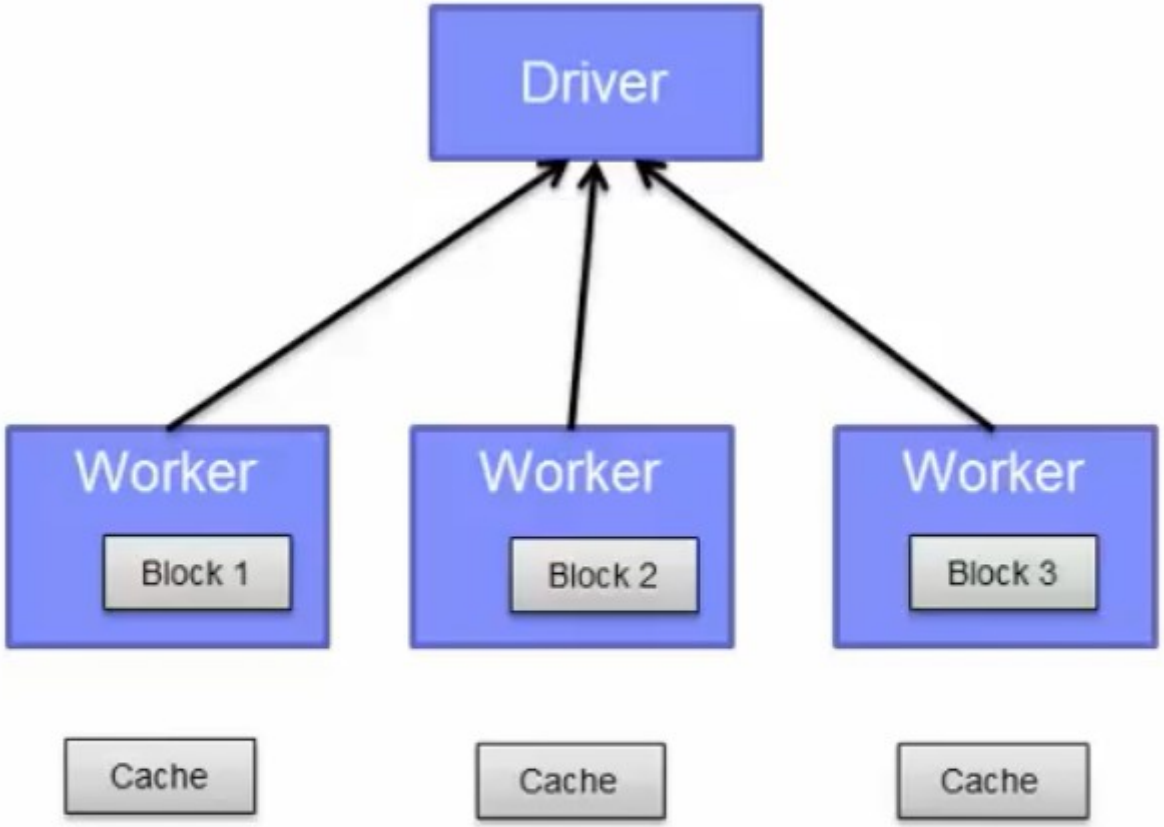
```
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
```



messages.cache()

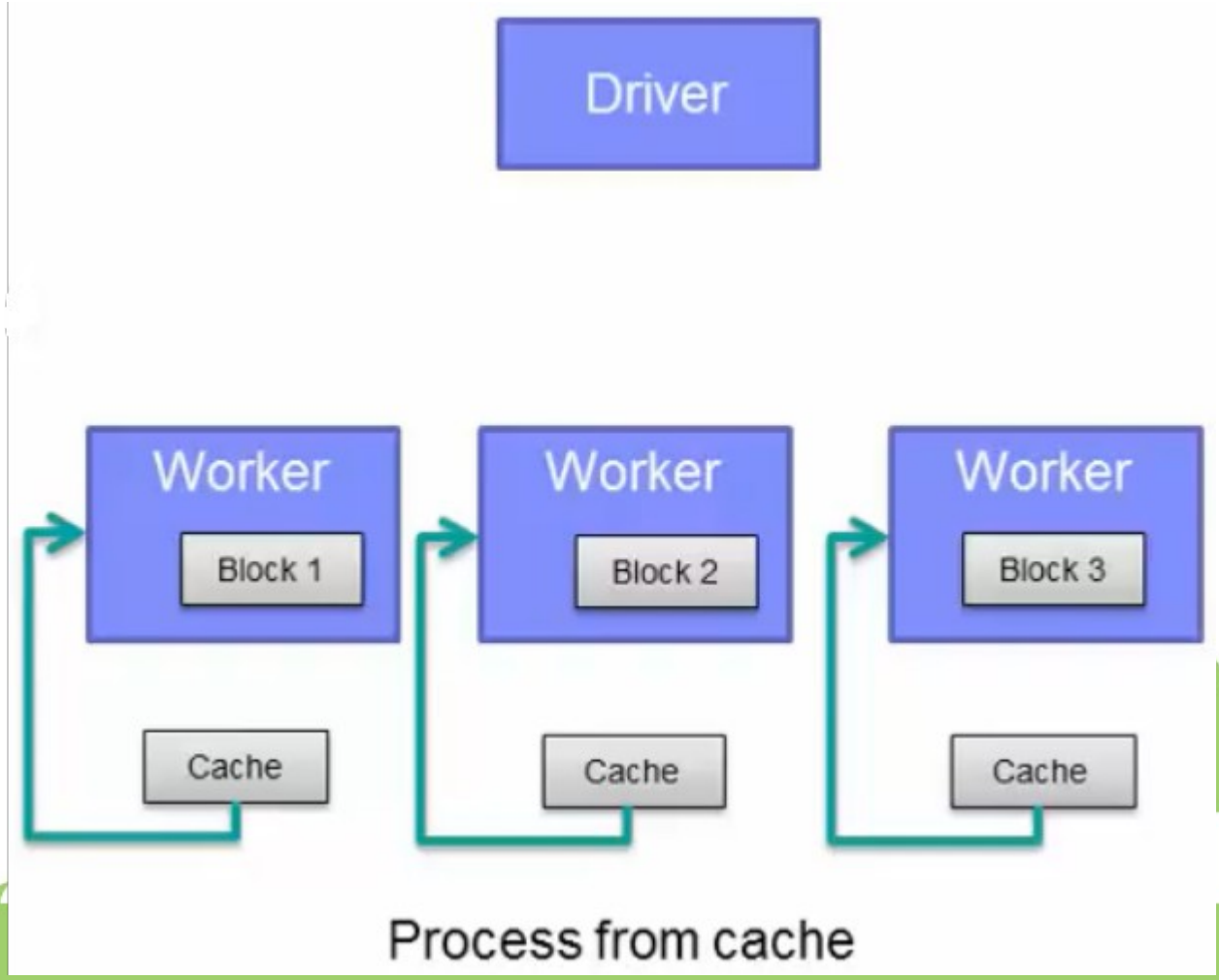


```
messages.filter(_.contains("mysql")).count()
```

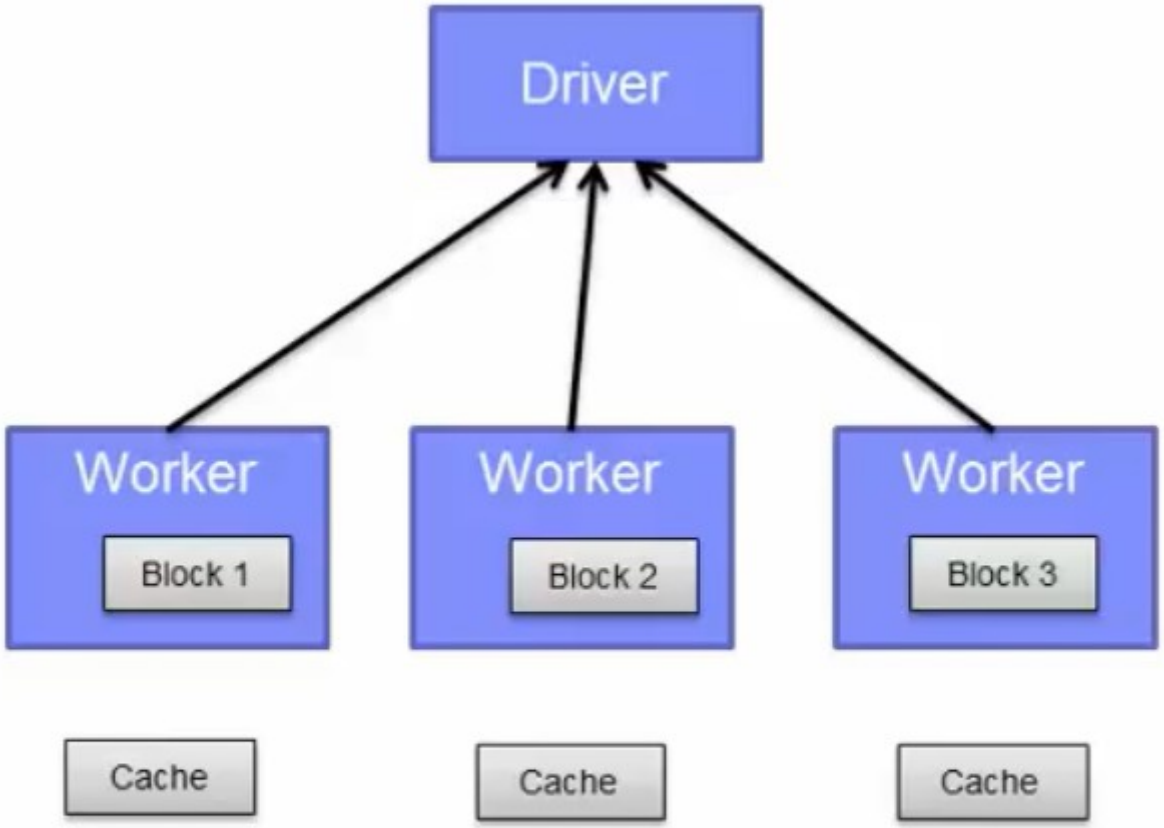


Send the data back to the driver

```
messages.filter(_.contains("php")).count()
```



Proses selesai



Send the data back
to the driver

Operasi RDD – Transformation (1)

- **`map(func)`** – return dataset baru dengan passing tiap elemen dari source melalui func
- **`filter(func)`** – return daset baru dengan memilih elements source yang func return TRUE
- **`flatMap(func)`** – mirip map, tapi tiap input bisa dimap ke 0 atau lebih output item. Func seharusnya return Seq bukan single item.
  Meratakan list dari list, untuk Map Reduce dengan text file dan setiap baris di-read, split baris dengan space individual keyword map tiap keyword ke nilai 1
- **`join(otherDataset, [numTasks])`** – ketika dipanggil di dataset tipe (K, V) dan (K, W), return kombinasi pair dataset (K, (V, W)) dengan semua pair elements untuk tiap key

Operasi RDD – Transformation (2)

- **`reduceByKey(func)`** – ketika dipanggil di dataset pair (K, V), return dataset pair (K, V) di mana value tiap key diagregasi dengan funct reduce. Cocok dengan WordCount.
- **`sortByKey([ascending], [numTasks])`** – ketika dipanggil di dataset pair (K, V) di mana K implemen. Return dataset pair (K, V) terurut dengan key ascending/descending

Operasi RDD - Action

- **`collect()`** – return semua elemen datset sebagai array driver program. Berguna setelah filter atau lainnya yang return small subset data
- **`count()`** – return jumlah elemen dataset. Cocok untuk cek dan test transformation
- **`first()`** – return elemen pertama dataset
- **`take(n)`** – return array dengan elemen n pertama dataset
- **`foreach(func)`** – run func tiap elemen dataset

RDD Persistent (1)

- Cache adalah contoh RDD persistent
- Cache MEMORY_ONLY (default) storage
- Key feature Spark speed dengan persisting dan caching
- Tiap node simpan tiap partisi cache dan compute di memori
- Ketika action di dataset sama atau turunan proses dari memori 10x lebih cepat untuk action berikutnya
- Pertama kali RDD persisted simpan di memori node
- Caching fault tolerant ketika 1 partisi hilang, di-recompute dengan transformation aslinya

RDD Persistent (2)

- Metode RDD persistent
  - **`persist()`** – spesifik storage level caching (disk atau memori (serialized object → save space))
  - **`cache()`** - MEMORY_ONLY storage (deserialized object)
    Jika partisi cache tidak cukup → recompute on the fly
- **`MEMORY_AND_DISK`** - Opsi di memori dan disk jika tidak cukup di memori
- **`MEMORY_ONLY_SER, MEMORY_AND_DISK_SER`** - Opsi serialized Java object sebelum simpan (space efficient) × butuh deserialized sebelum read → CPU workload ↑
- **`DISK_ONLY`** – opsi simpan hanya di disk
- **`MEMORY_ONLY_2, MEMORY_AND_DISK_2`**, dll - Opsi replicate tiap partisi → 2 node cluster

RDD Persistent (3)

- **`OFF_HEAP`** (experimental) - Opsi Experimental Storage Level, Tachyon simpan serialized mengurangi garbage collection overhead dan executor bisa lebih kecil dan berbagi pool memori

Pemilihan Storage Level

- Jika RDD cukup di default → biarkan default (MEMORY_ONLY) ← CPU efficient, lebih cepat
- `MEMORY_ONLY_SER` & fast serialization library → object space-efficient & tetap cukup kencang (INGAT butuh deserialization)
- Jauhi penggunaan disk, kecuali func yang compute dataset expensive/filter data ukuran besar. Lainnya, recompute partisi ≅ secepat membaca dari disk
- Gunakan replikasi ← fault recovery (Contoh web app). Semua level punya fault tolerance ← recompute lost data × Replikasi continue run task di RDD tanpa tunggu recompute partisi lost
- Environment memory atau app banyak, gunakan OFF_HEAP
  - multiple executor → share pool memori sama di Tachyon
  - mengurangi garbage collection signifikan
  - Satu executor crash → cache data tidak hilang

Shared Variable

- Ketika func di-passing dari driver ke worker copy terpisah variabel digunakan untuk tiap worker
- Dua tipe variabel:
  - Broadcast variable
    - read-only copy di tiap mesin
    - distribusi broadcast variable dengan algoritma broadcast yang efisien. Contoh: copy large dataset ke tiap node
  - Accumulator
    - untuk counter dan sum secara paralel
    - hanya bisa ditambahkan lewat operasi asosiatif
    - hanya driver yang bisa baca value-nya, bukan task hanya menambahkan
    - support numeric type, extend type baru (contoh: counter dan sum, seperti di MapReduce)

Key-Value Pair

- Support Scala, Python, dan Java seperti Spark umum
- Scala dan Jawa tidak ada zero index

Scala: key-value pairs

```
val pair = ('a', 'b')
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Python: key-value pairs

```
pair = ('a', 'b')
pair[0] # will return 'a'
pair[1] # will return 'b'
```

Java: key-value pairs

```
Tuple2 pair = new Tuple2('a', 'b');
pair._1 // will return 'a'
pair._2 // will return 'b'
```

Programming Key-Value Pair (1)

- Harus import SparkcContext/sc untuk PairRDDFunction, seperti reduceByKey
- Operasi paling umum: grouping dan aggregrating element dengan key
- RDD punya Tuple2 object, representasi key-value pair
  Dibuat dengan menulis (a, b) import **`org.apache.spark.SparkContext_`** untuk konversi implisit
- Jika ada custom object sebagai key di key-value pair
  butuh metode **`equal()`** dan **`hashCode()`** sendiri untuk perbandingan
  Contoh: PairRDDFunction – **`reduceByKey((a, b) => a + b)`**

Programming Key-Value Pair (2)

- Contoh:

```
val textFile = sc.textFile(“...”)
val readmeCount = textFile.flatMap(line => line.split(“
“)).map(word => (word, 1)).reduceByKey(_+_)
```

  - text RDD normal transforms buat PairRDD panggil reduceByKey method bagian dari PairRDDFunction API
    *reduceByKey(_+_) anonymous function, parameter tergantung output
    *semua fungsi digabung dalam 1 baris, flatMap, map buat RDD baru, panggil method reduceByKey dari RDD terakhir
    *1-1 lebih baik untuk testing fungsi