Shuffling

- O processo de “shuffling” dos dados é **caro** em muitos aspectos
  - Processos iniciais, tolerância a falhas, etc.
  - Dica: Minimize o uso

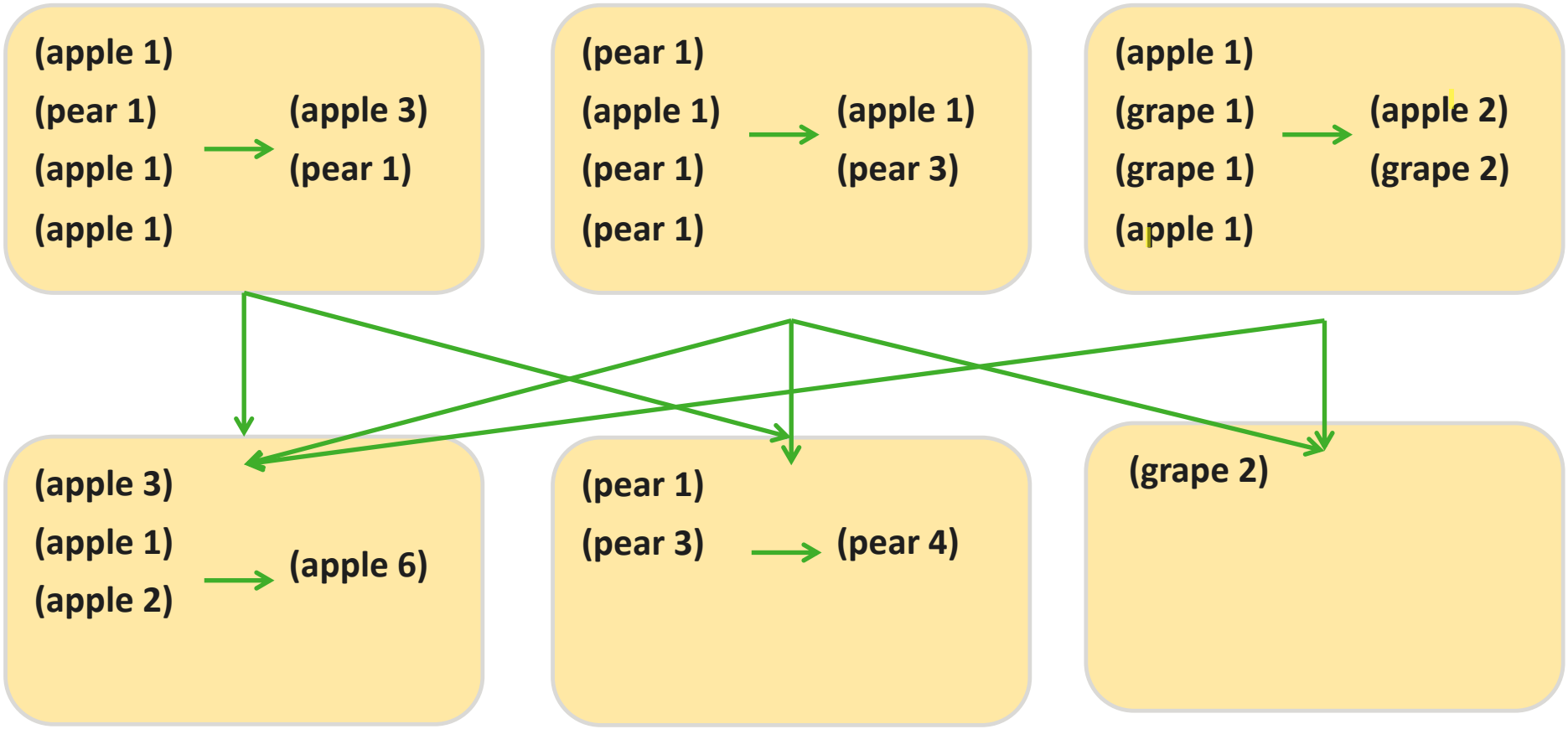
- Vamos rever alguns exemplos de um programa de contagem de palavras utilizando “shuffling”
  - Baseados em RDD

```
val line = "apple pear apple grape pear apple …"
val wordPairsRDD = sc.parallelize(line.split("\\s+")).
                                    map(word => (word,1))
val countsRDD = wordPairsRDD.reduceByKey(_ + _)
```

```
val countsRDD = wordPairsRDD.
                          .groupByKey()
                          .map(t => (t._1, t._2.sum))
```

reduceByKey()

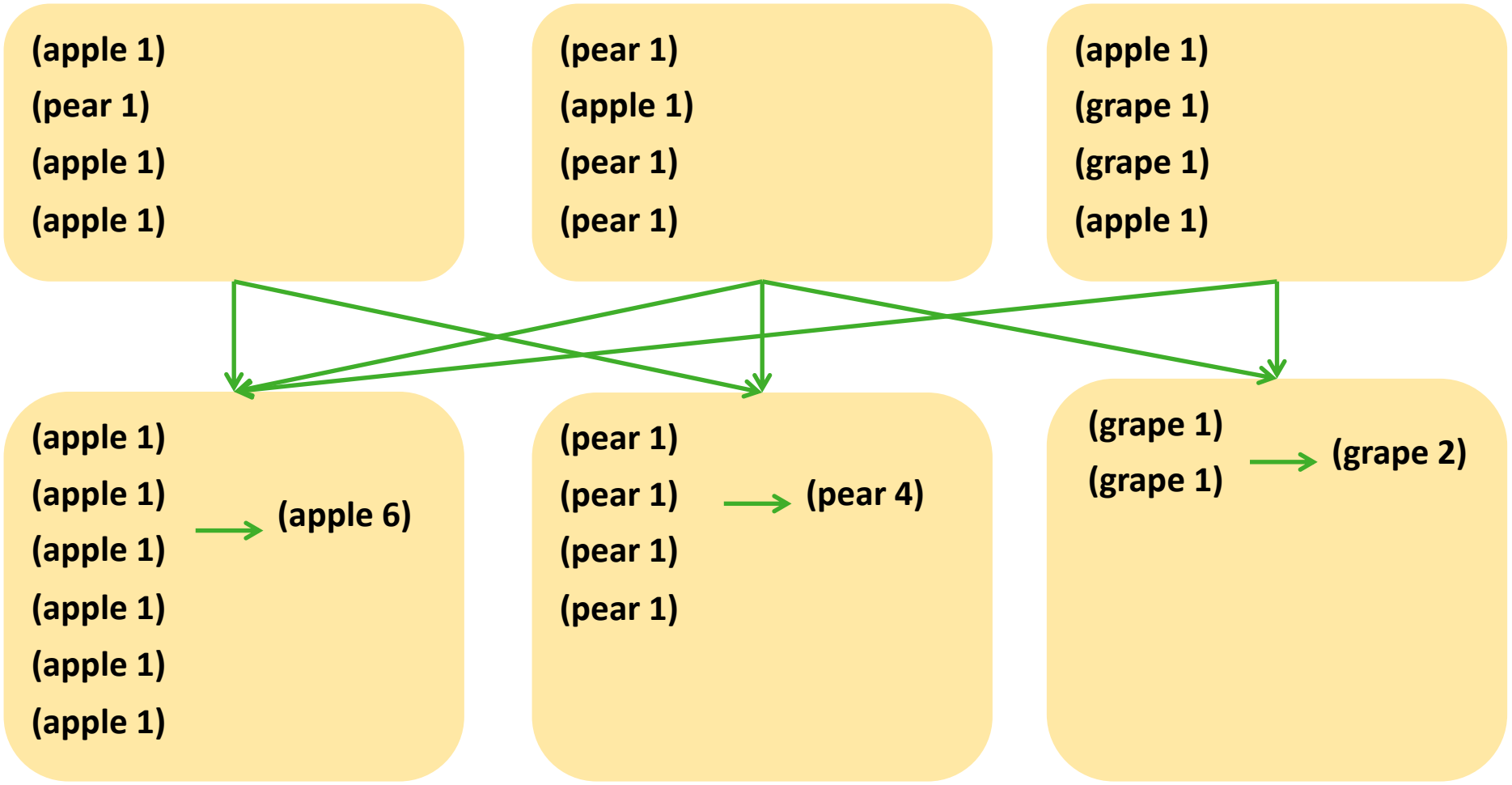
- Perceba que os pares em comum são combinados em cada partição primeiro
  - Reduz de forma significative o tamanho dos dados antes de fazer o “shuffling”
  - Os dados passam então pelo processo de “shuffling” — de forma mais rápida



Scala Systems

groupByKey()

- **Todos os dados passam pelo processo de "shuffling"** antes de serem reduzidos



Scala Systems

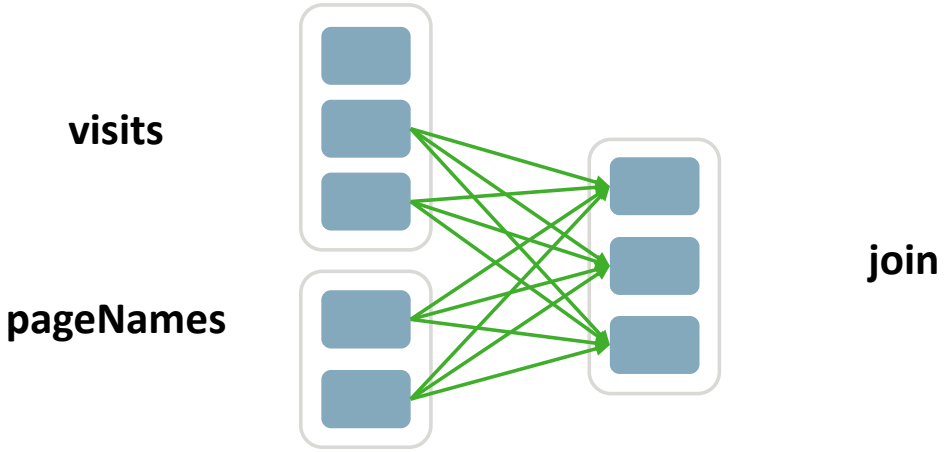
Joins e Shuffling

- Uma instrução “join” combina linhas de dois ou mais Datasets
  - Baseado em relações entra valores de colunas
  - Os valores das colunas são comparados e devem ser iguais
- Para um “join” distribuído, os dados passam por “**shuffling**“
  - Para que as linhas com os mesmos valores para a (s) coluna (s) de “join” estejam na mesma máquina
  - O “join” pode então ser realizada em cada partição e então combinado
- O tipo “join” acima pode exigir que os dados passem muito pelo processo de “**shuffling**“
  - Especialmente se você tiver um grande conjunto de dados
  - Shuffling == performance reduzida

Exemplo: Shuffling em Join

- Considere o exemplo abaixo
  - Spark realiza um “shuffle” para realizar o “join” entre os RDDs

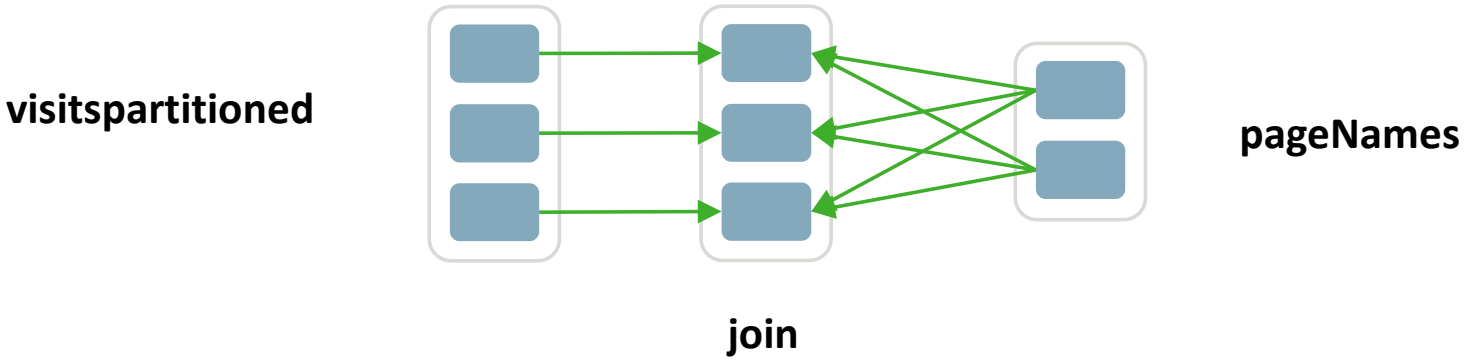
```
// RDD de (URL, visit) e.g. { ("index.html", "1.2.3.4"),
// ("about.html", "3.4.5.6"),("index.html", "1.3.3.1") }
> val visits = sc.textFile(“visits.txt”).map(...)

// RDD de (URL, name) e.g.
// { ("index.html", "Home"), ("about.html", "About") }
> val pageNames = sc.textFile(“pages.txt”).map(...)

// O join entre eles deve retornar { (about.html,(3.4.5.6,About) … }
> val joined = visits.join(pageNames)
```



Pré-particionamento para Reduzir o Shuffling

- Queremos minimizer o processo de “shuffling” nos dados
  - Se você fizer um hash-partition por “visits” antes de fazer o “join”, então não será necessário fazer o “shuffling”
    - E geralmente, os dados de “visits” é muito maior que os dado de “pages”
    - Então, isso pode reduzir considerávelmente o “shuffling”

```
val visitsPartitioned = sc.textFile(“visits.txt”).map(...)
        .partitionBy(new HashPartitioner(2))
val pageNames = sc.textFile(“pages.txt”).map(...)
val joined = visitsPartitioned.join(pageNames)
```



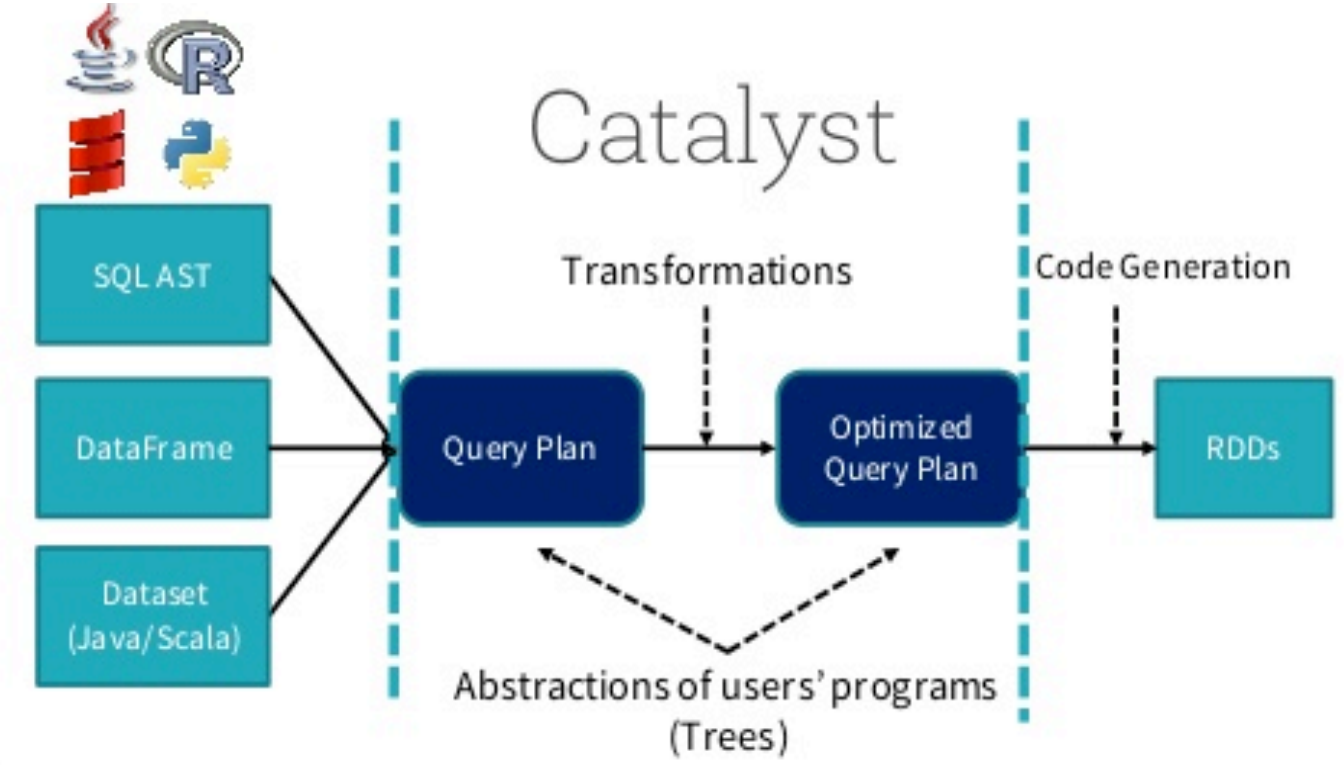
Resumo

- Ilustramos algumas preocupações em relação ao “shuffling”
  - Estes são pontos importates para se entender
  - Mas em um nível mais baixo— o **COMO** dos processos do Spark
    - Estamos pensando em como o interior do Spark se comporta
  - **NÃO** é o que queremos pensar para cada transformação
    - Tedioso, complexo e frágil
  - Queremos focar no **O QUE**
    - Não no COMO
- O **Otimizador Catalyst** te permite focar no **O QUE**
  - Ele vai otimizar o COMO para você
    - Quando trabalhando apropriadamento com DataFrames, Datasets, e SQL
    - **NÃO** usado quando trabalhamos com RDDs

Parte 5.3: O otimizador Catalyst

Visão geral sobre Catalyst

- O Catalyst **automáticamente otimiza** o seu código Spark
  - Quando é usado SQL, DataFrames, ou Datasets
  - Você foca no O QUE
  - O Catalyst automáticamente busca e realiza o melhor COMO



Scala Systems

Como o Catalyst Funciona

- Otimiza a sua transformação de ponta-a-ponta
  - Cria uma representação abstrata (uma árvore)
    - O **Plano Lógico de Análise**
  - Analisa e otimiza a representação abstrata
    - Usando muitas regras de otimização
    - Criando o **Plano Lógico Análisado** e o **Plano Lógico Otimizado**
  - Converte a representação abstrata para transformações reais
    - Cria múltiplos planos reais, aplica o modelo de custo e escolhe o melhor
    - Criando assim o **Plano Físico**

Quais otimizações são feitas pelo Catalyst?

- Provavelmente, mais do que você pode imaginar, incluindo :
  - **Predicate Pushdown**: Faça com que os filtros sejam uma das primeiras coisas a serem aplicadas
    - Elimina linhas que não atendem as pré-condições de processamento
  - **Projection Pushdown**: Faça com que as projeções de resultado sejam uma das primeiras coisas a serem aplicadas
    - Eliminando colunas que não vão ser usadas
  - **Reduzir o "shuffling"**: e.g. reduzindo antes de agrupar
    - Ou criando uma relação de uma maneira mais mais eficiente
  - **Cruzamento constante**: Cruze cálculos de constantes em literais
  - **Outras otimizações**: e.g. converter operações em Decimal para operações em Long
  - E muito mais
- Faremos alguns exemplos para ver tudo isso em ação

Exemplo: Contagem de palavras com o DSL

- Utiliza o DSL para todas as transformações
  - Será necessário realizar “shuffle” (utiliza o groupBy)
  - Filtra os resultados em dois momentos diferentes
    - Uma vez **antes** da agregação (counting) ser feita , e outra vez **depois**

```
// Preparando o dado
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF("line")

// Dividindo em palavras
> val splitWordsDF = linesDF.select(explode (split('line, "\\s+")).as("word")
).select(lower('word).as("word"))

// Filtrando e contando
> val filterThenCountDF = splitWordsDF  // Filtrando antes de contar
      .filter('word =!= "twinkle").groupBy('word).count (1)

// Contando e depois filtrando
> val countThenFilterDF = splitWordsDF.groupBy('word)
      .count.filter('word =!= "twinkle")  // Filtrando depois de contar
```

”Esclarecendo” as otimizações do Catalyst

- Utilize o método **`explain()`** em um DataFrame/Dataset para visualizer o plano de otimização do Catalyst

  `explain(): Unit` — Retorna o Plano Físico

  `explain(extended: Boolean): Unit` — Retorna os Planos Lógico e Físico

- Veremos o exemplo de filtragem **antes** de contagem, e **depois** de contagem
  - Ambos vão resultar no **mesmo plano** devido ao Catalyst

```
filterThenCountDF.explain
== Physical Plan ==
*HashAggregate(keys=[word#1168], functions=[count(1)])
+- Exchange hashpartitioning(word#1168, 200)
   +- *HashAggregate(keys=[word#1168],functions=[partial_count(1)])
```

Visão geral sobre o retorno do método explain()

- Para o resultado abaixo
  - 1-3 prepara os dados
  - 4 aplica explodes/splits nos dados
  - 5 filta os dados
  - 7 realiza um partial_count
  - 8 realiza o “shuffling” (called an exchange)
  - 9 realiza a contagem final (uma agregação)

```
countThenFilterDF.explain  // Filtrando depois da contagem
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6    +- *Project [lower(word#1165) AS word#1168]
5      +- *Filter NOT (lower(word#1165) = twinkle)
4        +- Generate explode(split(line#852, \s+)), false, false,
                                                          [word#1165]
3          +- *Project [value#850 AS line#852]
2            +- *SerializeFromObject [staticinvoke(…) AS value#850]
1              +- Scan ExternalRDDScan[obj#849]
```

Otimização: Predicate Pushdown

- O plano abaixo é sobre uma filtragem antes de uma contagem
  - O slide anterior era sobre uma filtragem depois de uma contagem
- Ambas as transformações possuem **exatamente o mesmo plano**
  - Mesmo que estejam escritas de formas diferentes
  - O Catalyst **antecipa as funções de filtro** o máximo possível

```
filterThenCountDF.explain  // Filtrando antes de contar
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6    +- *Project [lower(word#1165) AS word#1168]
5      +- *Filter NOT (lower(word#1165) = twinkle)
4        +- Generate explode(split(line#852, \s+)), false, false,
                                                          [word#1165]
3          +- *Project [value#850 AS line#852]
2            +- *SerializeFromObject [staticinvoke(…) AS value#850]
1              +- Scan ExternalRDDScan[obj#849]
```

Otimização: Reduzir o Shuffling

- O plano **minimiza o processo de shuffling**
  - Ele faz a contagem em duas etapas— em 7 (local) e em 9 (após o shuffle)
  - Isso é possível porque a contagem é comutativa
  - O Catalyst escolheu o equivalente ao nosso `reduceByKey` do RDD
  - E não o `groupByKey` por ser mais caro

```
filterThenCountDF.explain  // Filtrando antes da contagem
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7   +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6     +- *Project [lower(word#1165) AS word#1168]
5       +- *Filter NOT (lower(word#1165) = twinkle)
4         +- Generate explode(split(line#852, \s+)), false, false,
                                                              [word#1165]
3           +- *Project [value#850 AS line#852]
2             +- *SerializeFromObject [staticinvoke(…) AS value#850]
1               +- Scan ExternalRDDScan[obj#849]
```

Mais detalhes do plano

- Abaixo, mostramos como ocorre o push down dos filtros
  - Do Parsed para Optimized Logical Plan
  - Ele então gera os Planos Físicos, e seleciona o melhor

```
countThenFilterDF.explain(true)
== Parsed Logical Plan ==
'Filter NOT ('word = twinkle)
+- Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
   +- Project [lower(word#1165) AS word#1168]
   // …

== Analyzed Logical Plan ==
word: string, count: bigint
Filter NOT (word#1168 = twinkle)
// …

== Optimized Logical Plan ==
Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
+- Project [lower(word#1165) AS word#1168]
   +- Filter NOT (lower(word#1165) = twinkle)
   // …
```

Lambdas Impedem a Otimização do Catalyst

- Abaixo, criamos um filtro utilizando Lambda e depois fazemos a contagem
  - Onde a filtragem acontece no Plano Físico?
  - **Após a agregação**
  - O Catalyst **não pode otimizar Lambdas** (ou **UDFs**) — estes são opacos
  - Resultando em uma otimização menos eficiente

```
// Conte e então filtre utilizando Lambda. splitWordsDF utiliza split/explode
> val countThenFilterDF = splitWordsDF.groupBy('word)
      .count.filter(w => w.getString(0)!="twinkle").explain
== Physical Plan ==
*Filter <function1>.apply
8 *HashAggregate(keys=[word#1168], functions=[count(1)])
7 +- Exchange hashpartitioning(word#1168, 200)
6  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
5    +- *Project [lower(word#1165) AS word#1168]
4       +- Generate explode(split(line#852, \s+)), false, false,
                                                              [word#1165]
3         +- *Project [value#850 AS line#852]
2           +- *SerializeFromObject [staticinvoke(…) AS value#850]
1             +- Scan ExternalRDDScan[obj#849]
```

Mas não todas as Lambdas

- Abaixo, utilizamos o método groupByKey com um lambda, contamos, e então filtramos
  - Nosso código de filtragem está **depois** do agrupamento/agregação
  - No Plano Físico, a filtragem acontece **depois** da agregação — mesmo que o agrupamento use um lambda
  - O Catalyst entende a agregação (contagem) e continua a otimizar
  - No entanto, se o filtro usar um lambda, ele vai continuar não fazendo o push down

```
val linesDS = linesDF.as[String]  // Cria um Dataset
linesDS.flatMap(_.toLowerCase().split("\\s")).  // Divite em palavras
  .groupByKey(s => s).count                          // Agrupa e conta
  .filter('value =!= "twinkle").explain         // Aplica o filtro

== Physical Plan ==
*HashAggregate(keys=[value#1420], functions=[count(1)])
+- Exchange hashpartitioning(value#1420, 200)
  +- *HashAggregate(keys=[value#1420], functions=[partial_count(1)])
    +- *Project [value#1420]
      +- *Filter (isnotnull(value#1420) && NOT (value#1420 = twinkle))
```

Resumo

- Quando usamos DSL não devemos nos preocupar com o **COMO**
  - O Catalyst vai se preocupar com a eficiência e performance para você
  - Em geral, ele vai escolher a melhor forma de fazer

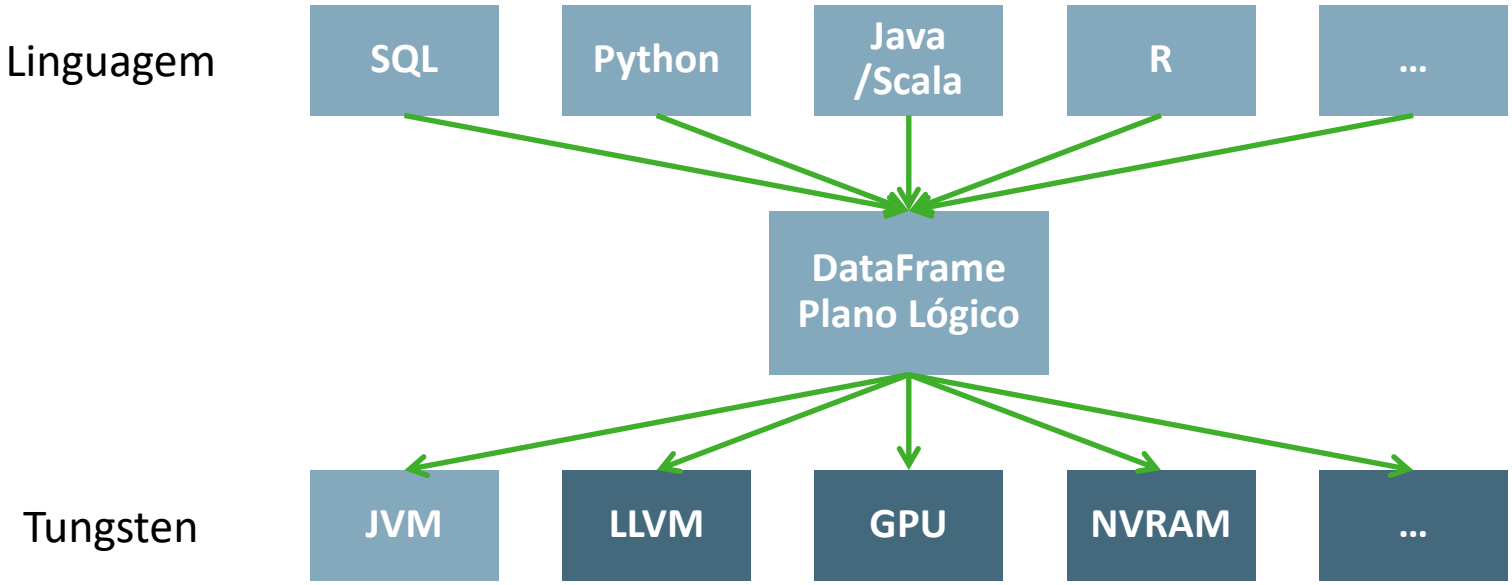
- **Lambdas** impedem o Catalyst
  - Mas nem sempre
    - Como vimos no exemplo do `groupByKey`
  - Você precisa entender como o Spark/Catalyst trabalha
    - `explain()` é seu amigo em caso de dúvida

Lab 5.2: Acompanhando o funcionamento do Catalyst

Parte 5.4: O otimizador Tungsten

Visão Geral do Tungsten

- Melhora a execução do Spark otimizando o uso de CPU / memória
  - Compreende e otimiza para arquiteturas de hardware
  - Otimizações para as características do Spark



O Formato Binário do Tungsten

- Representações binárias de objetos Java (formato de linha do Tungsten)
  - Diferente de serialização Java e Kryo
- Vantagens:
  - Tamanho menor comparado a serialização nativa do Java
  - Suporta a alocação off-heap
  - A estrutura suporta operações do Spark **sem deserialização**
    - e.g. você pode trabalhar os dados enquanto eles permanecem no formato binário
  - Evita a sobrecarga do GC
- Resultado:
  - Mais velocidade, menos uso de memória e CPU
  - Pode processar uma grande massa de dados

Computação com utilização de cache

- Hardwares modernos possuem muitos tipos de memória
  - Memória principal
  - Múltiplos níveis de cache(L1, L2, …)
  - Registros
  - Todos eles possuem características diferentes
- O Tungsten possue conhecimento dos diferentes tipos de memória e diferentes arquiteturas de hardware
  - Ele gera códigos que são otimizados para utilizer hardwares modernos
- Resultado: Muito mais performance
  - Por exemplo, ele usará um algoritmo de classificação usando cache
  - Trazendo uma melhora de 3x mais performance se comparado a uma versão sem usabilidade de cache

Geração de código em todo os estágios

- O otimizador Tungsten tem como finalidade melhorar a performance
  - Agrupa uma expressão de consulta em uma única função otimizada
- Por exemplo, suponha que estamos definindo o seguinte filtro

  ```
  filter('age>25 && 'age<50)
  ```

  - A geração de código gera dinamicamente o bytecode para este
  - Todo o código em uma única função
  - Em vez de interpretação clássica
    - Com o encaixotamento de tipos primitivos, chamadas de funções polimórficas ...
- Resultado:
  - Elimina chamadas de funções virtuais
  - Aproveita os registros da CPU para dados intermediários
  - Pode atender / exceder o desempenho da função ajustada manualmente para uma tarefa

Como usamos o Tungsten

- Sinceramente, não pense muito nisso
  - Apenas aproveite os benefícios
- Alguns problemas com **lambdas**!
  - Eles requerem objetos Java e não podem ser executados com dados no formato Tungsten
  - Isso adiciona uma camada de complexidade para trabalhar com o Tungsten
- Vamos ver isso em ação
  - Podemos ver isso em ação no Plano Físico
- Vamos observar um Plano Físico de um DataFrame e de um Dataset
  - Ilustrando o que o Tungsten está fazendo
  - Explorando alguns problemas com lambdas

Plano Físico do DataFrame

- **SerializeFromObject** converte os dados para o formato binário do Tungsten
  - A primeira coisa que é feita depois de ler os dados
  - Todas as operações depois disso são feitas neste formato altamente eficiente
  - Operações com asterisco (*) usam a geração de código em todos os estágios

```
// Preparando os dados
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF()
// Vendo o Plano Físico do DataFrames
> linesDF.select(explode (split('value, "\\s+")).as("word")  )
       .select(lower('word).as("word")).groupBy('word).count.explain
== Physical Plan ==
*HashAggregate(keys=[word#563], functions=[count(1)])
+- Exchange hashpartitioning(word#563, 200)
 +- *HashAggregate(keys=[word#563], functions=[partial_count(1)])
  +- *Project [lower(word#560) AS word#563]
   +- Generate explode(split(value#555, \s+)), false, false, [word#560]
    +- *SerializeFromObject [staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0,
java.lang.String, true], true) AS value#555]
      +- Scan ExternalRDDScan[obj#554]
```

Tungsten Melhora a Eficiência

- O exemplo anterior demonstra várias coisas
- As operações processam em cima de **dados no formato binário do Tungsten**
  - O dado é serializado para esse formato (`SerializeFromObject`) no início do pipeline
  - Todas as operações depois disso trabalham com o dado nesse formato
    - Alta eficiência de memória
- Boa parte das operações usam a **geração de código em todos os estágios**
  - Indicados por asterísco (*) antes do nome da operação
  - Alta eficiência de CPU
- Resultado: Execução significativamente mais rápida e uso de memória reduzido
  - Grande vitória para pipelines de dados

Plano Físico de Dataset

- **AppendColumnsWithObject** converte os dados para o formato Tungsten
  - **Após** o MapPartitions, que trabalha com objetos Java
  - Observe que o AppendColumnsWithObject **NÃO** utiliza a geração de código
  - Menos eficiente que o plano anterior

```
// Converte linesDF para Dataset, define a contagem de palavras e obtem o
// plano
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
s).count.explain
== Physical Plan ==
*HashAggregate(keys=[value#496], functions=[count(1)])
+- Exchange hashpartitioning(value#496, 200)
 +- *HashAggregate(keys=[value#496], functions=[partial_count(1)])
  +- *Project [value#496]
   +- AppendColumnsWithObject <function1>, […]
    +- MapPartitions <function1>, obj#492: java.lang.String
     +- Scan ExternalRDDScan[obj#483]
```

Serialização/Deserialização extra

- Fizemos uma mudança trivial ao carregar os dados
  - Nomeando nossa linha de entrada "linha" em vez do "valor" padrão
  - Observe como isso adiciona um estágio extra no nosso plano
  - Isso representa a serialização extra (para o formato do Tungsten) e após isso um passo de deserialização (para o formato Java) — a função lambda do método flatMap precisa de um objeto Java
  - Isso não é esperado, e adiciona uma sobrecarga extra

```
// Mudança trivial
val linesDF = sc.parallelize(Seq("Twinkle twinkle" … )).toDF("line")
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
s).count.explain
== Physical Plan ==  // rest of plan as before
  +- *Project [value#520]
   +- AppendColumnsWithObject <function1>, … AS value#520]
    +- MapPartitions <function1>, obj#516: java.lang.String
     +- DeserializeToObject line#509.toString, obj#515: java.lang.String
      +- *Project [value#507 AS line#509]
       +- *SerializeFromObject [… AS value#507]
        +- Scan ExternalRDDScan[obj#506]
```

Lab 5.3: Observando o trabalho do Tungsten

Parte 6: Incrementando a performance

Cache

Joins, Shuffles, Broadcasts, Accumulators

Orientações gerais

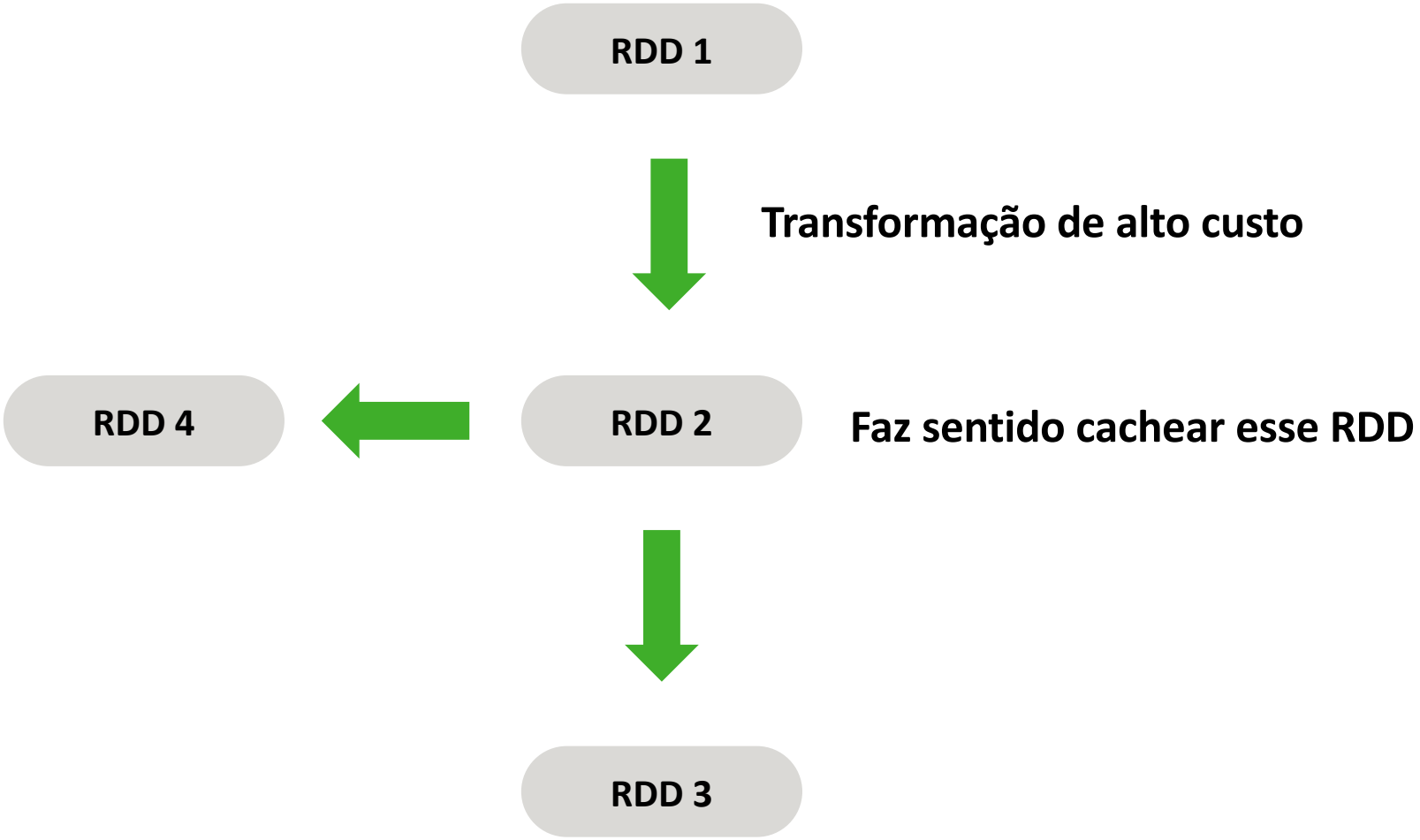
Parte 6.1: Cache

Motivos para usar o cache

- Sequencia padrão de execução de um Job do Spark:
  - Construção de um grafo de transformações
  - Após uma ação, execute o DAG, obtenha o resultado
  - **Não salva** RDDs intermediárias ou Datasets intermediários
- E isso é intencional - você pode rapidamente ficar sem memória
  - Mas as vezes você precisa persistir algum RDD
- O Spark permite persistir um RDD/Dataset entre operações
  - Você deve pedir explicitamente por isso
  - RDDs e Datasets possuem a mesma API e comportamento
- Fazemos o uso de cache para:
  - Salvar o resultado de um processamento caro
  - Processos iterativos (machine learning)

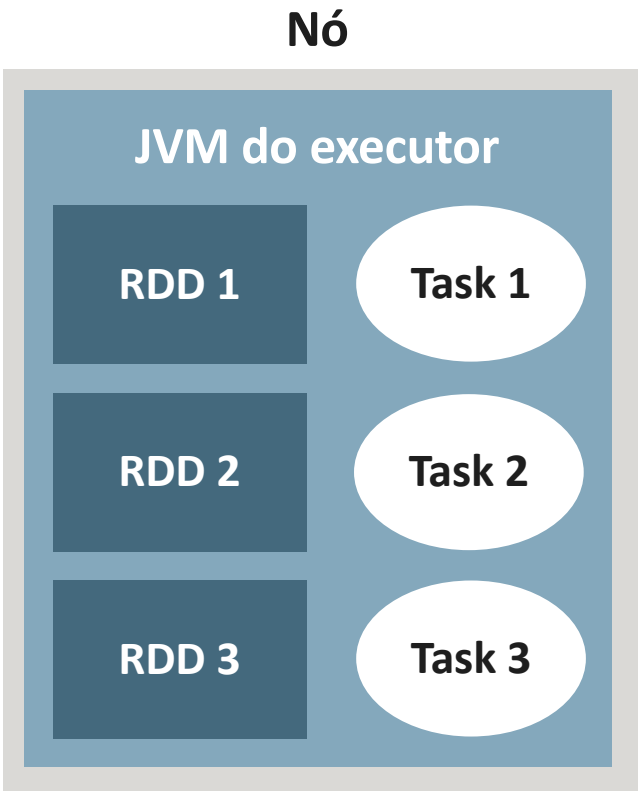
Cache

- Abaixo, o RDD 2 é usado para processar dois outros RDDs
  - Faz sentido cachear esse RDD



Mecanismos de cache

- Dados podem ser cacheados em:
  - Memória
  - HD
  - Combinação de ambos
- A ação de cachear um dado em memória é feita pelos **executores** em um **nó worker**
- Fique atento aos limites da JVM
  - Mínimo de memória da JVM: 4-8G
  - Máximo de memória da JVM: 40+G
    - Geralmente, quanto mais memória, mais tempo de GC
    - Os limites dependem do seu GC



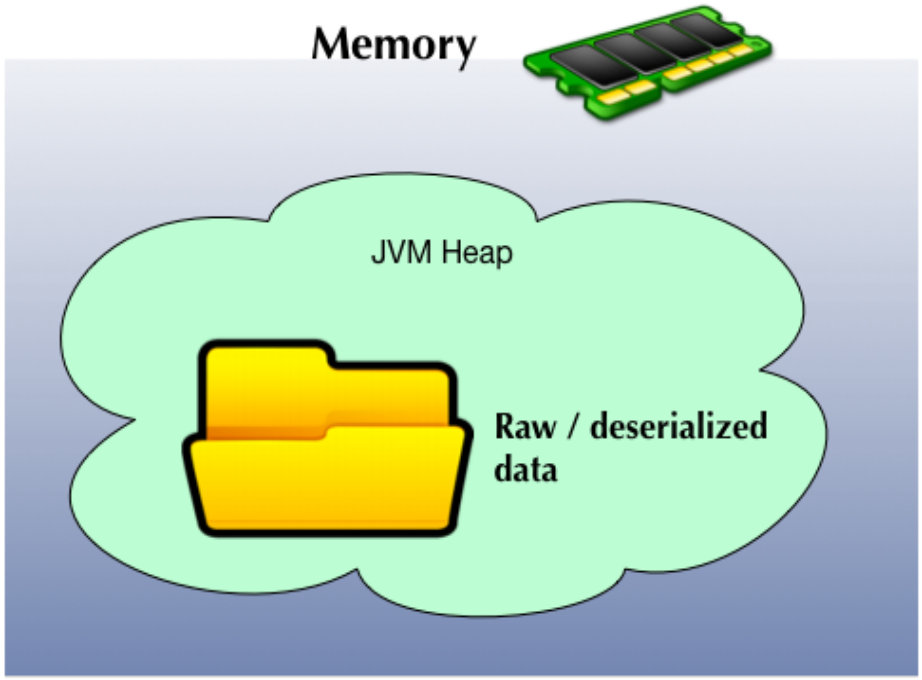
Níveis de persistência

- O Spark fornece muitos tipos de persistência
  - Os nós armazenam partições para reutilização, para que as ações futuras sejam mais rápidas

Nível de armazenamento	Comportamento
MEMORY_ONLY (nível padrão)	Armazena como objetos Java desserializados na JVM. Se o RDD não couber na memória, algumas partições não serão armazenadas em cache e vão ser reprocessadas.
MEMORY_AND_DISK	Armazena como objetos Java desserializados na JVM. Se o RDD não couber na memória, essas partições serão armazenadas no HD, e vão ser lidas quando for necessário.
MEMORY_ONLY_SER (Java e Scala)	Armazena como objetos Java serializados. Geralmente ocupa menos espaço que os objetos deserializados, porém é necessário mais uso de CPU para fazer a leitura.
MEMORY_AND_DISK_SER (Java e Scala)	Similar ao MEMORY_ONLY_SER, mas armazena no HD as partições que não couberam em memória
DISK_ONLY	Armazena as partições do RDD apenas no HD
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	O mesmo que os níveis anteriores, mas replica cada partição em dois nós do cluster
OFF_HEAP (experimental)	Armazenas os dados no formato serializado em off-heap. Reduz a sobrecarga do GC e traz outros benefícios

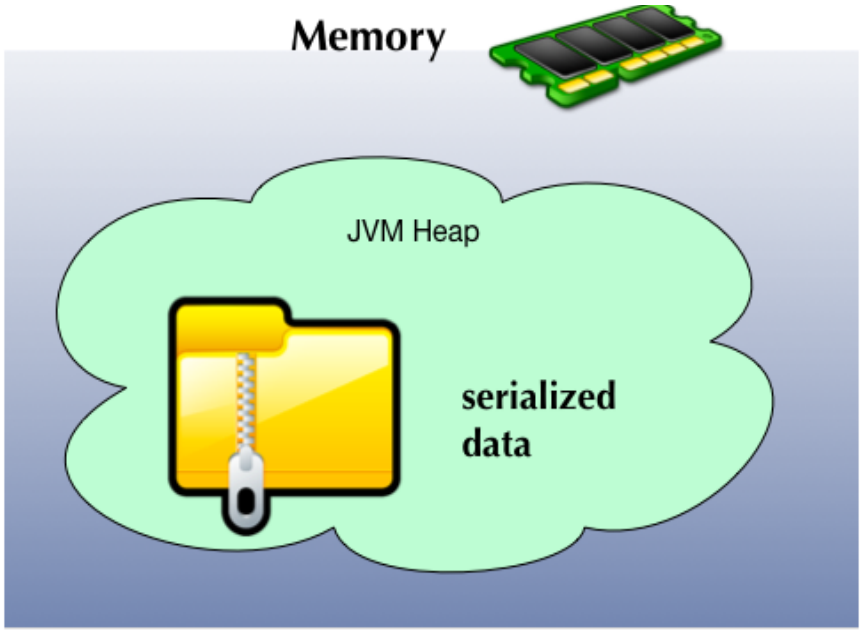
MEMORY_ONLY (Java Objects)

- `cache()` ou `persist(MEMORY_ONLY)`
  - Maior eficiência de CPU
  - Dados armazenados de forma deserialized
  - Aumenta o uso da memória (3x — 5x)
  - 1G de dados utiliza entre 3G — 5G de memória



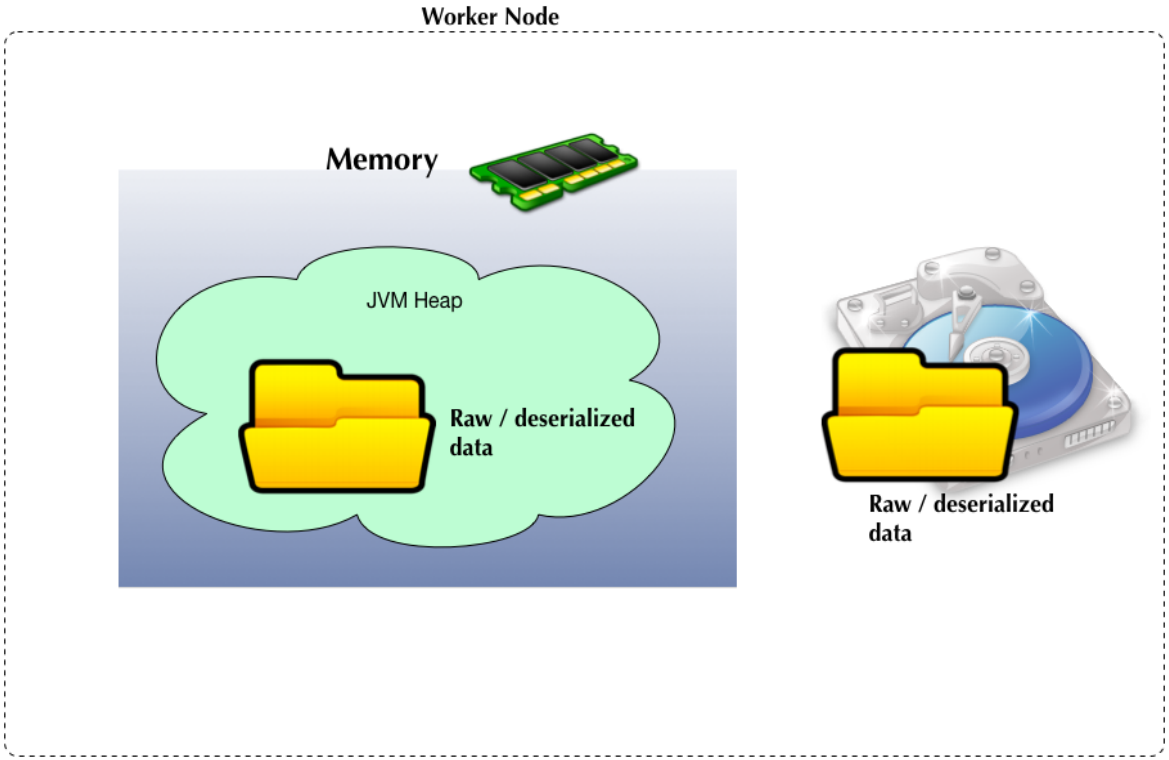
MEMORY_ONLY_SER (Serializado)

- `persist(MEMORY_ONLY_SER)`
  - Opção para uso de memória mais eficiente
  - Menor sobrecarga da memória
  - Uso intensivo de CPU (para serializar/ deserializar)
  - Por padrão utiliza o serializador Java — use o serializador ‘kryo’ para maior performance



MEMORY_AND_DISK

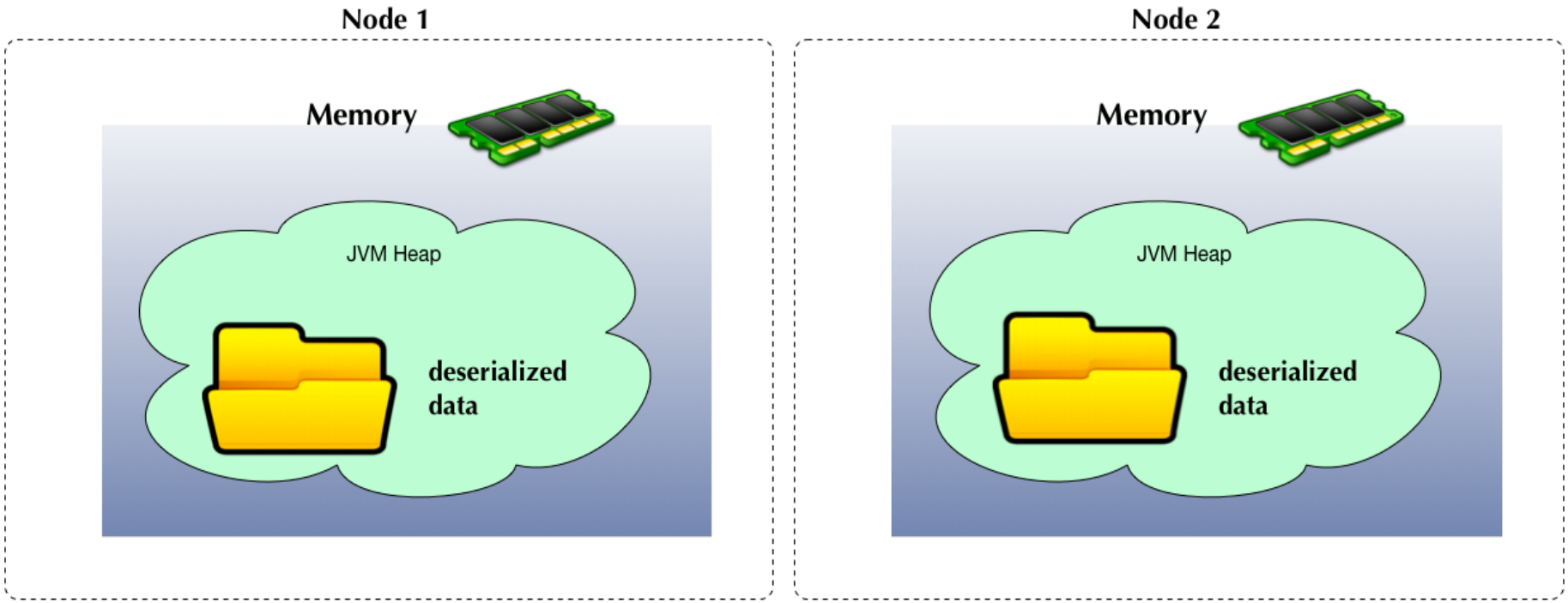
- `persist(MEMORY_AND_DISK)`
  - Tanto memória quanto HD
  - Pode sobreviver ao despejo de memória



Scala Systems

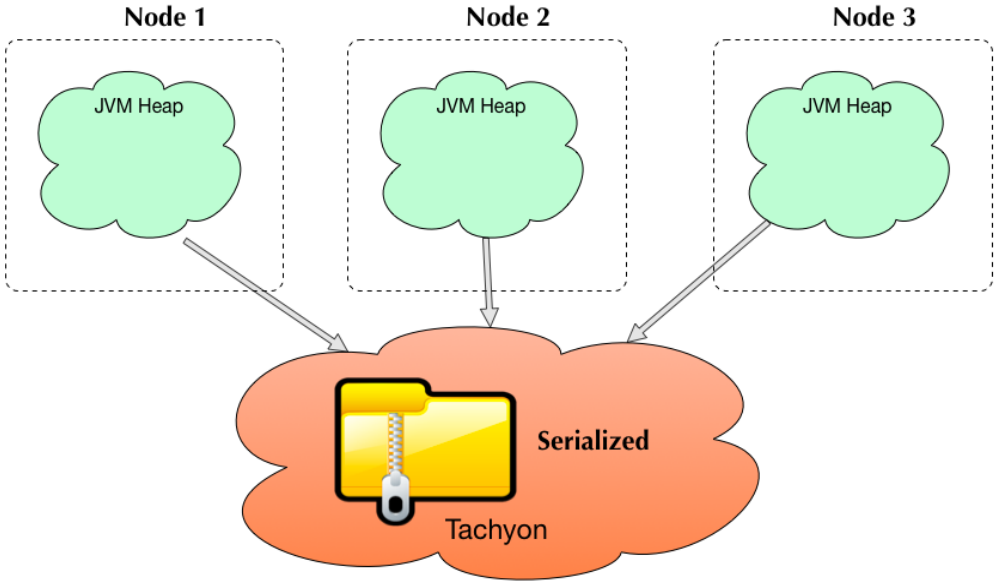
MEMORY_ONLY_2: Cache em múltiplos nós

- `persist(MEMORY_ONLY_2)`
  - Pode sobreviver a uma falha de nó



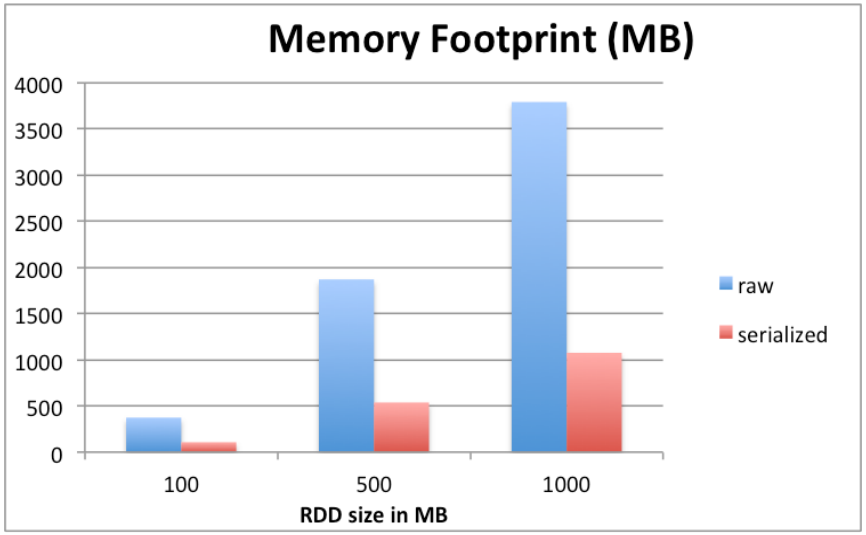
OFF_HEAP (Experimental)

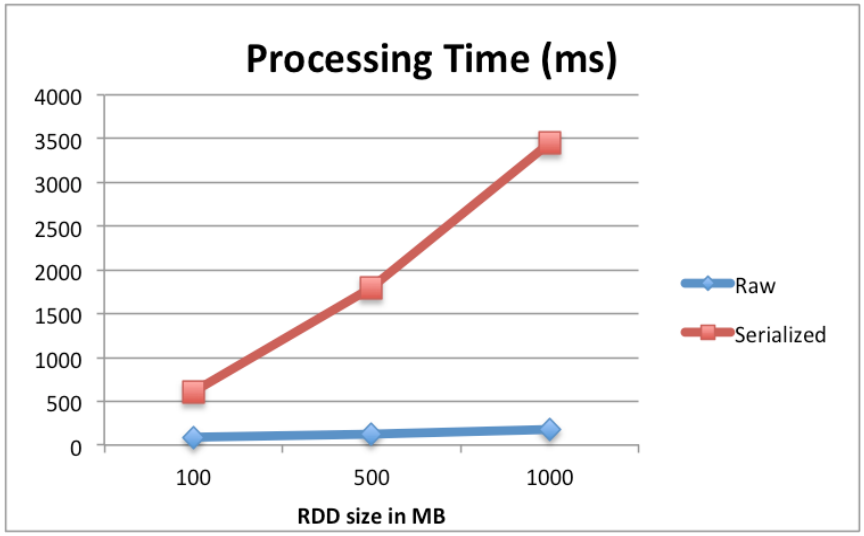
- Cache em memória (utiliza o Tachyon/Alluxio)
- Dados em cache distribuídos entre os nós
- Não utiliza o armazenamento da JVM
  - Não deve se preocupar com o Garbage Collection (GC)
  - Pode suportar uma grande massa de dados



Scala Systems

Problemas da memória cache





- O cache padrão consome mais memória (2-5x)
  - Mas é mais rápido de processar
  - MEMORY_ONLY
- O cache serializado consome menos memória
  - O tempo de processamento é maior
  - MEMORY_ONLY_SER

Dicas para a usabilidade do cache

- Apenas coloque no cache os dados que são usados com muita frequência
  - Se você cachear muitos dados, sua memória será sobrecarregada
  - Os dados menos usados recentemente serão eliminados
- Se os dados cabem na memória, utilize a opção MEMORY_ONLY (o padrão)
  - Mais eficiência de CPU
- Caso contrário, use o MEMORY_ONLY_SER
  - E escolha uma serialização mais performática
- Não use o HD a menos que seja armazenar dados que são resultados de filtros pesados e de processamento custoso
  - Caso contrário, a recomputação pode ser tão rápida quanto a leitura do disco

Dicas para a usabilidade do cache

- Use armazenamento replicado para recuperação rápida de falhas.
  - Você tem tolerância a falhas de qualquer forma, mas replicando os dados você tem menos tempo de indisponibilidade
- Considere o armazenamento OFF_HEAP (Tachyon)
  - Quando você tem muita memória
  - Quando múltiplas aplicações compartilham do mesmo dado
  - Muitas vantagens
    - Vários executores podem compartilhar o pool de memória no Tachyon
    - Reduz a ação do GC
    - Os dados armazenados em cache não são perdidos quando um executor falha
    - Aplicações externas ao Spark podem acessar os dados

API de persistência

- Os métodos a seguir são utilizados para persistência de dados
  - **`persist(newLevel: StorageLevel)`**: Defina o nível de armazenamento para newLevel
  - **`persist()`**: O mesmo que `persist(StorageLevel.MEMORY_ONLY)`
  - **`cache()`**: O mesmo que `persist(StorageLevel.MEMORY_ONLY)`

- Abaixo, montamos um exemplo
  - Observe que a chamada de cache não tem nenhum efeito imediato
  - Ela é adicionada ao DAG, e quando é criada, o Spark sabe como armazenar os dados em cache

```
val visits = sc.textFile("visits.txt").map(...)
val pageNames = sc.textFile("pages.txt").map(...)
val joined = visits.join(pageNames) // Vamos reutilizar isso
joined.cache() // Então cacheie
```

Lab 6.1: Cache