Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Apresentamos *Resilient Distributed Datasets* (RDDs), uma abstração de memória distribuída que permite que programadores realizem processamentos em memória de grandes *clusters* de modo a ter uma tolerância a falhas. Os RDDS foram motivados por dois tipos de aplicações que a os frameworks de computação atuais lidam de forma ineficiente: algoritmos iterativos e ferramentas de mineração de dados interativos. Em ambos os casos, manter os dados na memória pode melhorar o desempenho por uma ordem de magnitude. Para alcançar a tolerância a falhas de forma eficiente, RDDS fornece uma forma restrita de memória compartilhada, baseada nas transformações de *coarse-grined* ao invés de atualizações de *fine-grained* para uma condição compartilhada. No entanto, mostramos que RDDS são expressivos o suficiente para capturar uma ampla classe de cálculos, incluindo recentes modelos de programação especializados para trabalhos iterativos, como Pregel, e novas aplicações que esses modelos não capturam. Temos implementado RDDS em um sistema chamado Spark, que nós avaliamos por meio de uma variedade de aplicações de usuários e benchmarks.

1 Introdução

Frameworks de computação em cluster como MapReduce [10] e Dryad [19] têm sido amplamente adotados para análise de dados em larga escala. Estes sistemas permitem aos usuários escrevam computações paralelas usando um conjunto de operadores de alto nível, sem ter que se preocupar com a distribuição do trabalho e a tolerância a falhas.

Embora os frameworks atuais forneçam inúmeras abstrações para acessar recursos computacionais de um cluster, falta-lhes abstrações para um processamento em memória distribuída. Isto os torna ineficiente para uma importante classe de aplicações emergentes: aqueles que reutilizam resultados intermediários em vários cálculos. Reutilização de dados é comum em aprendizado de máquina e algoritmos gráficos iterativos, incluindo *PageRank*, cluster *K-means*, e regressão logística. Outro caso de uso convincente é a mineração de dados interativa, onde um usuário executa várias consultas ad-hoc sobre o mesmo subconjunto de dados. Infelizmente, na maioria das estruturas atuais, a única maneira de reutilizar dados entre computações (por exemplo, entre dois MapReduce *jobs*) é escrevê-lo a um sistema externo como uma *storage*, por exemplo, um sistema de arquivos distribuído. Isto implica em substanciais *overheads* devido à replicação de dados, I/O de disco, e serialização que podem afetar os tempos de processamentos da aplicação. Reconhecendo esse problema, os pesquisadores desenvolveram frameworks especializados para algumas aplicações que requerem reutilização de dados. Por exemplo, Pregel [22] é um sistema para cálculos de gráfico iterativos que mantém os dados intermediários em memória, enquanto HaLoop [7] oferece uma interface MapReduce iterativo. No entanto, estes frameworks só suportam padrões de computação específicos (por exemplo, uma séria de etapas de MapReduce), e realiza a partilha de dados implicitamente para esses padrões. Eles não fornecem abstrações para uma reutilização mais genérica, por exemplo, para permitir que um usuário carregue vários conjuntos de dados na memória e executem consultas ad-hoc através deles.

Neste trabalho, nós propomos uma nova abstração chamada *resilient distributed datasets* (RDDS) que permite a reutilização de dados eficiente numa ampla gama de aplicações. RDDS são estruturas de dados paralelas tolerantes a falhas que permitem aos usuários persistirem explicitamente os resultados intermediários na memória, controlando o seu particionamento para otimizar o posicionamento de dados, e então manipula-los usando um vasto conjunto de operadores.

O principal desafio na concepção dos RDDs é definir uma interface de programação que pode fornecer tolerância a falhas de forma eficiente. Abstrações existentes para armazenamento em memória em clusters, como memória distribuída compartilhada [24], key-value stores [25], bancos de dados e Piccolo [27], oferecem uma interface baseada em atualizações fine-grained para um estado mutável (por exemplo, células de uma tabela). Com esta interface, as únicas formas de fornecer tolerância a falhas são replicar os dados entre máquinas ou atualizações de log em máquinas. Ambas as abordagens são dispendiosas para cargas de trabalho intensivas de dados, uma vez que requerem copiar grandes quantidades de dados através da rede do cluster, cuja largura de banda é muito menor do que a RAM, e incorrem em custos de armazenamento substanciais.

Em contraste a estes sistemas, RDDs fornecem uma interface baseado em transformações *coarse-grained* (por exemplo, *map*, *filter* e *join*) que aplica a mesma operação para muitos dados. Isto permite que eles forneçam eficientemente um serviço de tolerância a falhas, registrando as transformações usadas para construir um conjunto de dados (*lineage*), ao invés de dados reais. Caso uma partição de uma RDD for perdida, o RDD tem informação suficiente sobre como ela foi derivada de outros RDDs para recalcular

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4

apenas essa partição. Assim, os dados perdidos podem ser recuperados, muitas vezes rapidamente, sem a exigir replicação dispendiosa.

Embora uma interface baseada em transformação coarse-grained pode parecer à primeira vista limitado, RDDs são uma boa opção para muitas aplicações paralelas, porque essas aplicações naturalmente aplicam a mesma operação para vários dados. De fato, nós mostramos que RDDs podem expressar eficientemente muitos modelos de programação em cluster que até agora têm sido propostas como sistemas separados, incluindo MapReduce, DryadLINQ, SQL, Pregel e HaLoop, bem como novas aplicações que estes sistemas não entendem, como a mineração de dados interativa. A capacidade dos RDDs de acomodar as necessidades de computação que anteriormente eram satisfeitas apenas pela introdução de novos frameworks é, acreditamos, a evidência mais confiável do poder da abstração RDD.

Temos implementado RDDs em um sistema chamado Spark, que está sendo usado para pesquisa e produção Aplicações da UC Berkeley e em várias empresas. Spark oferece uma interface de programação integrada com uma linguagem, semelhante ao DryadLINQ [31] na linguagem de programação Scala [2]. Além disso, o Spark pode ser usado interativamente para consultar grandes conjuntos de dados a partir do Scala. Acreditamos que o Spark é o primeiro sistema que permite uma linguagem de programação de uso geral para ser usado em interativas velocidades em memória para mineração de dados em clusters.

Nós avaliamos RDDs e o Spark através de *micro benchmarks* e medições de aplicações de usuário. Comprovamos que o Spark é até 20 mais rápido do que o Hadoop para aplicações interativas, processa um relatório de dados do mundo real em 40x, e pode ser usado de forma interativa para digitalizar um conjunto de dados de 1 TB com latência 5-7s. Fundamentalmente, para ilustrar a generalidade dos RDDs, temos implementado os modelos de programação Pregel e HaLoop no Spark, incluindo as otimizações de posicionamento que eles empregam, como bibliotecas relativamente pequenas (200 linhas de código cada).

Este artigo começa com uma visão geral dos RDDs (§2) e Spark (§3). Discutiremos a representação interna dos RDDs (§4), nossa implementação (§5), e os resultados experimentais (§6). Finalmente, discutiremos como os RDDs capturaram vários modelos de programação em cluster existente (§7), realizam trabalhos relacionado (§8), e concluem.

2 Resilient Distributed Datasets (RDDs)

Esta seção fornece uma visão geral de RDDs. Primeiro definimos RDDs (§2.1) e introduzimos sua interface de programação em Spark (§2.1). Em seguida, comparamos RDDs com abstrações de memória compartilhada de grão fino (§2.3). Finalmente, discutimos as limitações do modelo RDD (§2.4).

2.1 Abstração RDD

Formalmente, um RDD é um read-only, coleção particionada de registos. RDDs somente podem ser criados por meio de operações deterministicas em (1) dados em armazenamento estável ou (2) outros RDDs. Chamamos essas transformações de operações para diferenciá-los de outras operações sobre RDDs. Exemplos de transformações incluem *map*, *filter* e join.

RDDs não precisam ser materializados em todos os momentos. Em vez disto, um RDD tem informação suficiente sobre como foi derivado a partir de outros conjuntos de dados (*lineage*) para calcular as suas partições de dados em armazenamento estável. Esta é uma propriedade poderosa: em essência, um programa não pode fazer referência a um RDD que não pode reconstruir após uma falha.

Finalmente, os usuários podem controlar outros dois aspectos de RDDs: persistência e particionamento. Os usuários podem indicar qual RDDs serão reutilizados e escolher uma estratégia de armazenamento para eles (por exemplo, o armazenamento em memória). Eles também podem pedir que os elementos de um RDD sejam particionados entre máquinas com base em uma key em cada registro. Isto é útil para otimizações de particionamento, como garantir que dois conjuntos de dados que serão unidos em um conjunto sejam *hash-partitioned* da mesma maneira.

2.2 Spark Programming Interface

O Spark expõe os RDDs através de uma API integrada na línguagem semelhante ã DryadLINQ [31] e FlumeJava [8], onde cada conjunto de dados está representado como um objeto e as transformações são invocados usando métodos sobre nestes objetos.

Os programadores começam definindo um ou mais RDDs através de transformações em dados em armazenamento estável (por exemplo, *map* e *filter*). Eles podem usar esses RDDS em ações, que são operações que retornam um valor para a aplicação ou exportam dados para um sistema de armazenamento. Exemplos de ações incluem *count* (que devolve o número de elementos no conjunto de dados), collect (que retorna os próprios elementos), e *save* (fornece o dataset para um sistema de armazenamento). Como DryadLINQ, Spark computa RDDs *lazily* a primeira vez que eles são usados em uma ação, para que ele possa fazer a transformação em Pipeline.

Além disso, os programadores podem chamar o método *persist* para indicar quais RDDs querem reutilizar em operações no futuro. O Spark mantém persistentes RDDs na memória por padrão, mas pode fazer um *flush* para o disco se não houver memória RAM suficiente. Os usuários também podem solicitar outras estratégias de persistência, como armazenar o RDD apenas em disco ou replicá-lo através das máquinas, através de *flags* para persistir. Finalmente, os usuários podem definir uma prioridade de persistência em cada RDD para especificar quais dados na memória deve se espalhar para o disco primeiro.

2.2.1 Example: Console Log Mining

Suponha-se que um serviço web está tendo erros e um operador deseja pesquisar terabytes de logs no sistema de arquivos Hadoop (HDFS) para encontrar a causa. Usando Spark, o operador pode carregar apenas as mensagens de erro a partir dos logs para a RAM através de um conjunto de nós e consultá-los de forma interativa. Ela iria primeiro digite o seguinte código Scala:

[2]Although individual RDDs are immutable, it is possible to imple-ment mutable state by having multiple RDDs to represent multiple ver-sions of a dataset. We made RDDs immutable to make it easier to de-scribe lineage graphs, but it would have been equivalent to have our abstraction be versioned datasets and track versions in lineage graphs.

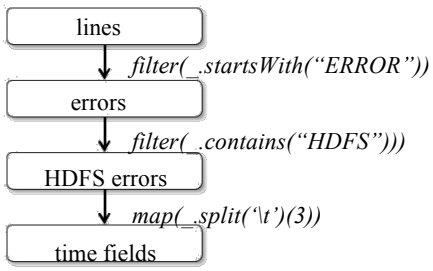


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

Linha 1 define uma RDD suportado por um arquivo HDFS (como uma coleção de linhas de texto), enquanto que a linha 2 deriva de um RDD filtrado. Linha 3, em seguida, pede *errors* sejam persistidos na memória para que eles posaam ser compartilhado entre consultas. Observe que o argumento de filter é Scala sintaxe para um fechamento.

Neste ponto, nenhum trabalho foi realizado no cluster. No entanto, agora, o usuário pode usar o RDD em ações, por exemplo, para contar o número de mensagens:

```
errors.count()
```

O usuário pode também executar outras transformações no RDD e usar os seus resultados, como nas linhas seguintes:

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Após a primeira ação envolvendo *errors*, o Spark armazenará as partições de *errors* na memória, deixando os próximos processamentos mais rápidos. Observe que a base RDD, linhas, não é carregado na memória RAM. Isso é desejável porque as mensagens de *errors* pode ser apenas uma pequena fração dos dados (pequenas o suficiente para caber na memória).

Finalmente, para ilustrar a forma como o nosso modelo obtém tolerância a falha, mostramos o gráfico *lineage* dos RDDs da terceira consulta da Figura 1. Nesta consulta, começamos com *errors*, o resultado de um filtro em linhas, e aplicado um filtro adicional e *map* antes de executar a *collect*. O Spark irá encaminhará as duas últimas transformações e enviará um conjunto de tarefas para computa-los para os nós que mantem as partições em cache dos *errors*. Além disso, se uma partição de erros é perdida, Spark reconstrói aplicando um filtro somente na partição correspondente das linhas.

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

2.3 Advantages of the RDD Model

Para entender os benefícios dos RDDs como uma abstração de memória distribuída, comparamo-los com memória compartilhada distribuída (DSM) na Tabela 1. Nos sistemas DSM, as aplicações leem e escrevem em locais arbitrários em um espaço de endereço global. Nota-se que, segundo esta definição, que incluem não só os sistemas tradicionais de memória partilhada [24], mas também outros sistemas onde as aplicações realizam gravações *fine-grained* no estado compartilhado, incluindo Piccolo [27], que fornece uma DHT compartilhada e bancos de dados distribuídos. DSM é uma abstração muito geral, mas esta generalidade torna mais difícil de implementar de forma eficiente e tolerante a falhas em clusters de commodities.

A principal diferença entre RDDs e DSM é que os RDDs só podem ser criados ("written"), através de transformações de coarse-grained, enquanto DSM permite que ler e escrever em cada local de memória. Isso restringe RDDs para aplicações que executam as gravações em massa, mas permite a tolerância a falhas mais eficiente. Além disso, RDDs não precisam ser recalculados em caso de falha, apenas as partições perdidas, uma vez que pode ser recuperado usando o *lineage*. In addition, os RDDs podem ser recalculadas em paralelo em diferentes nós, sem ter que reverter todo o programa.

Um segundo benefício dos RDDs é que a sua natureza imutável permite que um sistema de mitigue os nós lentos (*stragglers*) executando cópias de backup referente a tarefas lentas como no MapReduce [10]. As tarefas de backup seriam difícil de implementar com DSM, como as duas cópias de uma tarefa seria acessar as mesmas posições de memória e interferir nas atualizações de cada um.

Finalmente, os RDDs fornecem duas outras vantagens sobre DSM. Primeiro, em operações em massa em RDDs, um runtime pode agendar tarefas com base na localidade

[3]Note that reads on RDDs can still be fine-grained. For example, an application can treat an RDD as a large read-only lookup table.

[4]In some applications, it can still help to checkpoint RDDs with long lineage chains, as we discuss in Section 5.4. However, this can be done in the background because RDDs are immutable, and there is no need to take a snapshot of the whole application as in DSM.

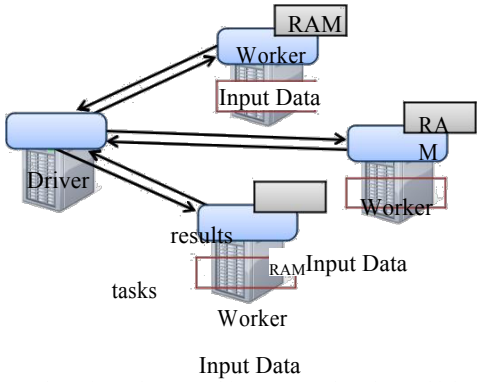


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

de dados para melhorar o desempenho. Segundo, RDDs fornecem duas outras vantagens sobre DSM. Primeiro, em operações em massa em RDDs, um *runtime* pode agendar tarefas com base na localidade. Em segundo lugar, os RDDs degradam normalmente quando não há memória suficiente para armazená-los, contanto que eles estejam sendo usados apenas em operações baseado em varredura. As partições que não cabem na memória RAM podem ser armazenadas no disco e fornecerão um desempenho semelhante aos sistemas de dados em paralelo atuais.

2.4 Applications Not Suitable for RDDs

Como discutido na introdução, RDDs são mais adequados para aplicações batch que se aplicam a mesma operação para todos os elementos de um conjunto de dados. Nestes casos, RDDs pode lembra-se de forma mais eficientemente de cada transformação como uma etapa de um gráfico de *lineage* e podem recuperar partições perdidas sem ter que registrar grandes quantidades de dados. Os RDDs seriam menos adequadas para aplicações que fazem atualizações de assíncronas fine-grained para o estado compartilhado, como um sistema de armazenamento para uma aplicação web ou um web *crawler* incremental. Para estas aplicações, é mais eficiente utilizar sistemas que executam o log de atualização tradicional e a verificação de dados, como bases de dados, RAMCloud [25], coador [26] e Piccolo [27]. Nosso objetivo é fornecer um modelo de programação eficiente para análise em batch e deixar essas aplições assíncronos para sistemas especializados.

3 Spark Programming Interface

Spark fornece uma abstração de RDD através de um API integrada em línguagem semelhante ao DryadLINQ [31] no Scala [2], uma linguagem de programação funcional de estaticamente tipada para a Java VM. Nós escolhemos o Scala devido à sua combinação de concisão (que é conveniente para o uso interativo) e eficiência (devido a tipagem estática). No entanto, nada sobre a abstração RDD requer uma linguagem funcional.

Para usar o Spark, os desenvolvedores escrevem um programa *driver* que se conecta a um cluster de *wokers*, como mostrado na Figura 2. O *driver* define um ou mais RDDs e invoca ações sobre eles. O código Spark no Spark também segue o *lineage* dos RDDs. Os *workers* são processos de longa duração que podem armazenar partições RDDs na memória RAM em todas as operações.

Como mostramos no exemplo de mineração de logs na Seção 2.2.1, os usuários fornecem argumentos para o RDD, como operações *map* passando fechamentos (literais funções). Scala representa cada encerramento como um objeto de Java, e estes objetos podem ser serializados e carregado em outro nó para passar o fechamento através da rede. Scala salva todas as variáveis ligadas ao encerramento de campos em objeto Java. Por exemplo, pode-se escrever código como var x = 5; rdd.map (_ + x) para adicionar 5 para cada elemento de um RDD.

Os próprios RDDs são objetos tipados estaticamente parametrizados por um tipo de elemento. Por exemplo, RDD [Int] é um RDD de inteiros. Entretanto, a maioria dos nossos exemplos omite tipos desde que Scala suporte a inferência de tipos.

Embora o nosso método de expor RDDs em Scala é conceitualmente simples, tivemos que contornar problemas com objetos de fechamento do Scala usando reflexão [33]. Também precisávamos de mais trabalho para fazer o Spark ser utilizável a partir da linguagem intérprete Scala, como discutiremos na Seção 5.2. No entanto, não precisamos modificar o compilador Scala.

3.1 RDD Operations in Spark

A Tabela 2 lista as principais transformações RDDs e ações disponíveis no Spark. Nós damos a assinatura de cada operação, mostrando parâmetros de tipo entre colchetes. Recall são operações de transformações *lazy* que definem um novo RDD, enquanto as ações lançam um cálculo para retornar um valor para o programa ou gravar dados em uma *storage*.

Observe que as operações, como *join*, só estão disponíveis em RDDs de pares key-value. Além disso, nossos nomes de função são escolhidos para coincidir com outras APIs em Scala e outras linguagens funcionais; por exemplo, o map é um mapeamento um-para-um, enquanto flatMap mapeia cada valor de entrada para uma ou mais saídas (semelhante ao map em MapReduce).

Além desses operadores, os usuários podem solicitar a persistência de um RDD. Além disso, os usuários podem obter a ordem de partição de um RDD, que é representado por uma classe *Partitioner* e particionar um outro conjunto de dados de acordo com o seu. Operações tais como *groupByKey*, *reduceByKey* e *sort* automaticament resultam em um hash ou um range de RDD partitioned.

3.2 Example Applications

Complementamos o exemplo de mineração de dados na seção 2.2.1 com duas aplicações iterativas: *Logist Regression* e *PageRank*. O último mostra como o controle de particionamento dos RDDs pode melhorar o desempenho.

3.2.1 Logistic Regression

Muitos algoritmos de aprendizado de máquina são iterativos por natureza porque eles executam procedimentos de otimização iterativos, como gradiente descendente, para maximizar uma função. Eles podem, portanto, executar muito mais rápido, mantendo seus dados na memória.

Como exemplo, o seguinte programa implementa regressão logística [14], um algoritmo de classificação comum

[5]We save each closure at the time it is created, so that the map in this example will always add 5 even if ***x*** changes.

Transformations	map(f : T) U) : RDD[T]) RDD[U] filter(f : T) Bool) : RDD[T]) RDD[T] flatMap(f : T) Seq[U]) : RDD[T]) RDD[U] sample(fraction : Float) : RDD[T]) RDD[T] (Deterministic sampling) groupByKey() : RDD[(K, V)]) RDD[(K, Seq[V])] reduceByKey(f : (V; V)) V) : RDD[(K, V)]) RDD[(K, V)] union() : (RDD[T]; RDD[T])) RDD[T] join() : (RDD[(K, V)]; RDD[(K, W)])) RDD[(K, (V, W))] cogroup() : (RDD[(K, V)]; RDD[(K, W)])) RDD[(K, (Seq[V], Seq[W]))] crossProduct() : (RDD[T]; RDD[U])) RDD[(T, U)] mapValues(f : V) W) : RDD[(K, V)]) RDD[(K, W)] (Preserves partitioning) sort(c : Comparator[K]) : RDD[(K, V)]) RDD[(K, V)] partitionBy(p : Partitioner[K]) : RDD[(K, V)]) RDD[(K, V)]
Actions	count() : RDD[T]) Long collect() : RDD[T]) Seq[T] reduce(f : (T; T)) T) : RDD[T]) T lookup(k : K) : RDD[(K, V)]) Seq[V] (On hash/range partitioned RDDs) save(path : String) : Outputs RDD to a storage system, e.g., HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

que procura por um *hyperplane* w que melhor separe dois conjuntos de pontos (por exemplo, spam e e-mails que não são spam). O algoritmo utiliza descida gradiente: ele começa W com um valor aleatório, e em cada iteração, ele soma uma função de w sobre os dados para mover w numa direção de melhora.

```
val points = spark.textFile(...)
                  .map(parsePoint).persist() var w
= // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Começamos definindo um RDD persistente chamado *points* como o resultado de um de uma transformação *map* em um arquivo de texto que analisa cada linha de texto em um objeto *Point*. Em seguida, executamos um *map* e *reduce* nos pontos para calcular o gradiente em cada etapa somando uma função de w. Manter *points* na memória através de iterações pode render 20x *speedup*, como mostramos na Seção 6.1.

3.2.2 PageRank

Um padrão mais complexo de compartilhamento de dados ocorre em PageRank [6]. O algoritmo de forma iterativa atualiza uma classificação para cada documento, adicionando-se contribuições de documentos que ligam a ela. Em cada iteração, cada documento envia uma contribuição de r para os seus vizinhos, onde r é a sua posição e n é o número de vizinhos. Em seguida, ele atualiza a sua classificação para $\alpha/N + (1 - \alpha)\sum ci$, em que a soma é sobre as contribuições que recebeu, e N é o número total de documentos. Podemos escrever PageRank em Spark da seguinte forma:

```
// Load graph as an RDD of (URL, outlinks) pairs
```

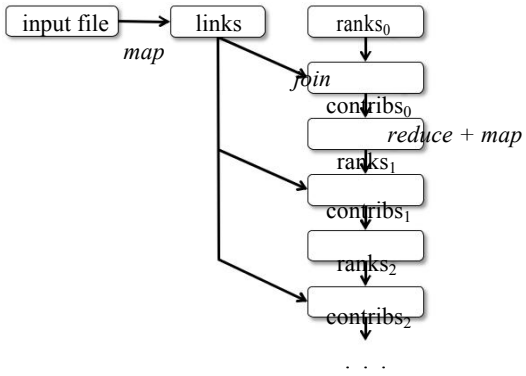


Figure 3: Lineage graph for datasets in PageRank.

```
val links = spark.textFile(...).map(...).persist() var ranks = //
RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  //  Build an RDD of (targetURL, float) pairs
  //  with the contributions sent by each page val
  contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  //  Sum contributions by URL and get new ranks ranks
  = contribs.reduceByKey((x,y) => x+y)
              .mapValues(sum => a/N + (1-a)*sum)
}
```

Este programa leva para o gráfico de *lineage* RDD na Figure-3. Em cada iteração, criamos um novo *ranks* de dados de baseados nas *contribs* e *ranks* a partir da iteração anterior e estáticos *links* de dados. Uma característica interessante deste gráfico é que ele cresce mais com o número

[6]Note that although RDDs are immutable, the variables ***ranks*** and ***contribs*** in the program point to different RDDs on each iteration.

de iterações. Assim, em um trabalho com muitas iterações, pode ser necessário replicar de forma confiável algumas das versões de *ranks* para reduzir o tempo de recuperação de falhas [20]. O usuário pode chamar *persist* com uma *RELIABLE* flag para fazer isso. No entanto, note que o conjunto de dados de ligações não precisa ser replicado, porque as partições dele podem ser reconstruídas de forma eficiente pela reruning de um *map* em blocos de arquivos de entrada. Este conjunto de dados será tipicamente muito maior do que os *ranks*, porque cada documento tem muitos *links*, mas apenas um número como seu *rank*, então recuperá-lo usando *lineage* economiza tempo em relação aos sistemas que que controlam o estado inteiro de memória de um programa.

Finalmente, podemos otimizar a comunicação no PageRank, controlando o particionamento nos RDDs. Se especificarmos um particionamento para *links* (por exemplo, *hash-partition* o link listas de URL através de nós), podemos particionar *ranks* da mesma maneira e garantir que a operação de *join* entre os *ranks* e *links* e não requeiram comunicação (como URL's *rank* será na mesma máquina como a sua lista de *link*). Também podemos escrever uma classe *Partitioner* personalizada para agrupar páginas que ligam-se entre sí (por exemplo, particionar URLs por domínio). Ambas as otimizações podem ser expressadas chamando um partitionBy quando definimos *links*:

```
links = spark.textFile(...).map(...)
              .partitionBy(myPartFunc).persist()
```

Após esta chamada inicial, a operação *join* entre *links* e *ranks* agregará automaticamente as contribuições para cada URL para a máquina de suas listas de *links* ligadas, calcula a sua nova classificação, e *join* com os seus links. Este tipo de particionamento consistente em iterações e é uma das principais otimizações em estruturas especializadas como Pregel. RDDs permitem que o usuário expresse este objetivo diretamente.

4 Representing RDDs

Um dos desafios na apresentação de RDDs como uma abstração é escolher uma representação para eles que possa acompanhar o *lineage* através de uma ampla gama de transformações. Idealmente, um sistema implementando RDDs deve fornecer um conjunto de operadores de transformação tão rico quanto possível (por exemplo, os da Tabela 2), e permitir que os usuários compõem de forma arbitrária. Propomos uma representação simples baseada em gráfico para RDDs que facilita esses objetivos. Utilizamos esta representação no Spark para suportar uma ampla gama de transformações sem adicionar lógica especial para o programador para cada um, o que simplificou bastante o design do sistema.

Em resumo, propomos representar cada RDD através de uma interface comum que expõe cinco informações: um conjunto de partições, que são partes atômicas do conjunto de dados; um conjunto de dependências de RDDS pais; uma função para calcular o conjunto de dados com base em seus pais; e metadados sobre seu esquema de particionamento e posicionamento dos dados. Por exemplo, um RDD que representa um arquivo HDFS tem uma partição para cada bloco do arquivo e sabe em quais máquinas cada bloco está ligado. Enquanto isso, o resultado

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

de um *map* neste RDD tem as mesmas partições, mas aplica-se a função de *map* aos dados do pai ao calcular seus elementos. Resumimos essa interface na Tabela 3.

A questão mais interessante na concepção desta interface é como representar dependências entre RDDs. Descobrimos forma o suficiente e útil para classificar dependências em dois tipos: *narrow* dependências, onde cada partição do RDD pai é usada por no máximo, uma partição do RDD filho, *wide* dependências, onde várias partições filhas podem depender dele. Por exemplo, o *map* leva a uma dependência *narrow*, enquanto *join* leva a uma dependência *wide* (a menos que os pais são *hash-partitioned*). A Figura 4 mostra outros exemplos.

Esta distinção é útil por duas razões. Primeiro, dependências *narrow* permitem a execução em pipeline em um nó do cluster, que pode calcular todas as partições pai. Por exemplo, pode-se aplicar um *map* seguido por um *filter*, numa base de elemento por elemento. Em contraste, as dependências *wide* necessitam de dados de todas as partições pai para estar disponível e ser *shuffled* entre os nós usando uma operação MapReduce. Em segundo lugar, a recuperação após uma falha de um nó é mais eficiente com uma dependência *narrow*, uma vez que apenas as partições pai perdidos precisam ser recalculados, e eles podem ser recalculadas em paralelo em diferentes nós. Em contraste, em um gráfico de *lineage* com dependências *wide*, a falha de um único nó pode causar a perda da partição de todos os antepassados de um RDD, exigindo uma nova execução completa.

Essa interface comum para os RDDs tornou possível a implementação da maioria das transformações em Spark em menos de 20 linhas de código. De fato, mesmo os novos usuários do Spark implementam transformações (por exemplo, amostragem e vários tipos de *joins*) sem conhecer os detalhes do *scheduler*. Esboçamos algumas implementações do RDD abaixo.

Arquivos HDFS: os RDDs de entrada em nossas amostras foram arquivos do HDFS. Para esses RDDs, as partições retornam uma partição para cada bloco do arquivo (com o deslocamento do bloco armazenado em cada objeto *Partition*), preferredLocations fornece onde os nós do bloco está ligado e o *iterator* lê o bloco.

Mapa: Chamando *map* em qualquer RDD retorna um objeto MappedRDD. Este objeto tem as mesmas partições e locais preferenciais como seu pai, mas se aplica a função passada para

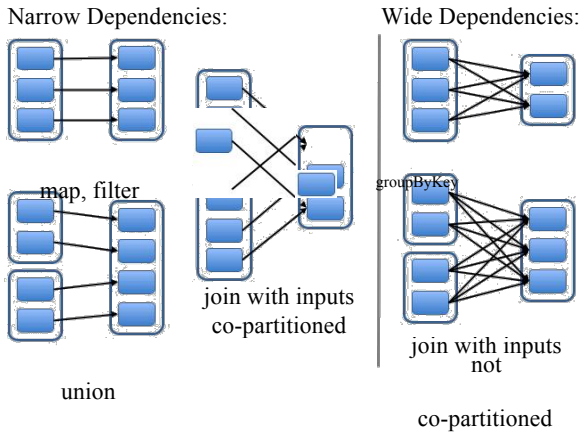


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

map os registros do pai em seu método *iterator*.

union: chamando união em dois RDDs retorna um RDD cujas partições são a união das dos pais. Cada partição filha é calculado através de uma depência *narrow* sobre o pai correspondente.

sample: a amostragem é semelhante para mapeamento, exceto que o RDD armazena um gerador de número aleatórios para cada partição para determinar deterministamente os registros do pai.

join: a junção de dois RDDs pode levar a qualquer uma das duas dependências *narrow* (se eles são ambos *hash/range partitioned* particionado com o mesmo *partitioner*), duas dependências *wide*, ou uma mistura (se um dos pais tiver um particionador e outro não). Em ambos os casos, a saída RDD tem um *partitioner* (qualquer um herdado dos pais ou por um *partitioner* de hash padrão).

5 Implementation

Implementamos o Spark em cerca de 14.000 linhas de Scala. O sistema funciona sobre o gerenciador de cluster Mesos [17], permitindo que ele compartilhe recursos com Hadoop, MPI e outras aplicações. Cada programa Spark é executado como uma aplicação Mesos separado, com seu próprio *driver* (*master*) e os *workers*, e o compartilhamento de recursos entre estas aplicações é gerenciado pelos Mesos.

O Spark pode ler dados a partir de qualquer fonte de entrada Hadoop (por exemplo, HDFS ou HBase) usando APIs de plug-ins de entrada existentes do Hadoop e executa em uma versão não modificada do Scala.

Esboçamos agora algumas das partes tecnicamente interessantes do sistema: o *job scheduler* (§5.1), o interpretador Spark que permite o uso interativo (§5.2), gerenciador de memória (§5.3), e suporte para *checkpoints* (§5.4).

5.1 Job Scheduling

Spark *scheduler* usa nossa representação de RDDs, descrita na seção 4.

Em geral, nosso scheduler é similar ao Dryad's [19], mas leva em conta quais partições de RDDs persistentes estão disponíveis em memória.

[7]Note that our union operation does not drop duplicate values.

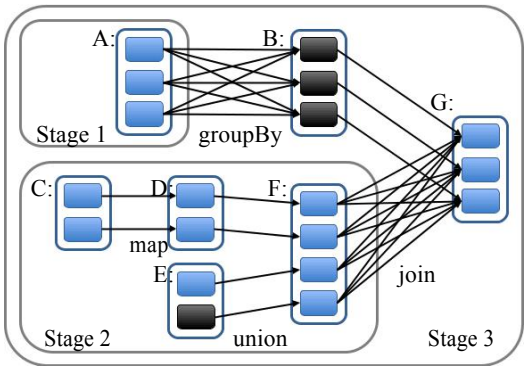


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline nar-row transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Sempre que um usuário executa uma ação (por exemplo, contar ou salvar) em um RDD, o scheduler examina o gráfico *lineage* desse RDD para construir um DAG de fases a ser executado, conforme ilustrado na Figura 5. Cada fase contém várias transformações em pipeline com dependências *narrow* quanto possível. Os limites das fases são as operações de *shuffle* necessários para *wide* dependências, ou quaisquer partições já computados que podem ter short-circuit de computação de um RDD pai. Então, o *scheduler* inicia tarefas para computar partições ausentes de cada fase até que tenha calculado o RDD destino.

Nosso *scheduler* atribui tarefas a máquinas com base na localidade dos dados usando *delay scheduling* [32]. Se uma tarefa precisa processar uma partição que está disponível na memória em um nó, enviamos para esse nó. Caso contrário, se uma tarefa processa uma partição para o qual o *containing* RDD fornece localizações preferidas (por exemplo, um arquivo HDFS), enviamos para eles.

Para dependências *wide* (isto é, dependências *Shuffle*), atualmente materializamos que possuem partições pai para simplificar a recuperação de falhas, assim como MapReduce materializa saídas de *map*.

Se uma tarefa falhar, nós reexecutamos em outro nó, desde que os pais de sua fase ainda estejam disponíveis. Se algumas fases se tornaram indisponíveis (por exemplo, porque uma saída do "*map side*" de um *Shuffle* foi perdido), nós reenviamos as tarefas para computar as partições ausentes em paralelo. Ainda não toleraramos falhas do *sheduler*, embora a resplicação do gráfico *lineage* RDD seria simples.

Finalmente, embora todos as computações no Spark atualmente serem executados em resposta a ações chamadas dentro do *driver program*, estamos também experimentando deixar tarefas no cluster (por exemplo, *maps*) e chamar a operação de pesquisa, que fornece acesso aleatório aos elementos de *hash-partitioned* RDDs por chave. Neste caso, as tarefas precisarão dizer ao *scheduler* para calcular a partição necessária se ele estiver ausente.

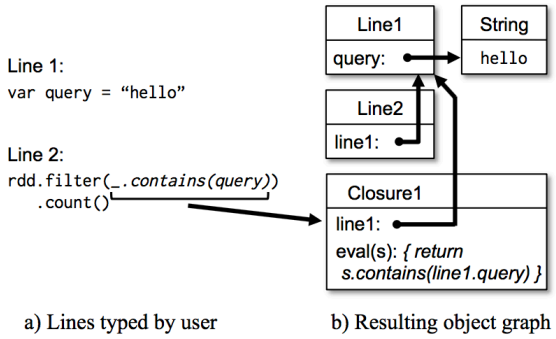


Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

5.2 Interpreter Integration

Scala inclui um *shell* interativo semelhantes ao de Ruby e Python. Dadas as latências baixas obtidas com dados na memória, queríamos permitir que os usuários executassem o Spark interativamente a partir do interpretador para consultar grandes conjuntos de dados.

O interpretador Scala normalmente opera uma classe para cada linha digitada pelo usuário, carregando-a na JVM, e invocando uma função nele. Esta classe inclui um objeto *singleton* que contém as variáveis ou funções nessa linha e executa o código da linha em um método inicialização. Por exemplo, se o usuário digitar var x = 5 seguido por println (x), o interpretador define uma classe chamada Linha1 contendo X e faz com que a segunda linha compile ***println(Line1.getInstance().x)***.

Fizemos duas alterações no interpretador Spark:

1. **Class shipping:** para permitir que os nós worker busquem o bytecode para as classes criadas em cada linha, fizemos com que o interpretador servisse as classes via HTTP.
2. **Modified code generation:** normalmente, o objeto criado para cada linha de código é acessado através de um método estático em sua classe correspondente permitir que os nós worker. Isto significa que quando serializamos uma clousure referenciando uma váriavel definida emu ma linha anterior, como ***Line1.x*** no exemplo acima, o Java não rastreará atrav[es do gráfico do objeto para enviar a instância ***Line1*** envolvento x. Portanto, os nós worker não receberão x. Modificamos a lógica de geração de código para fazer referência a instância de cada objeto de linha diretamente.

A Figura 6 mostra como o interpretador traduz um conjunto de linhas digitadas pelo usuário para objetos Java após nossas alterações.

Descobrimos que o interpretador Spark é útil no processamento de grandes traces obtidos como parte de nossa pesquisa e explorando conjuntos de dados armazenados no HDFS. Nós também planejamos usar linguagens de consulta de alto nível de forma interativa, por exemplo, SQL.

5.3 Memory Management

O Spark fornece três opções para o armazenamento de RDDs persistentes: armazenamento na memória como objetos Java desserializados, armazenamento em memória como dados serializados e armazenamento em disco. A primeira opção oferece o desempenho mais rápido, porque o Java VM pode acessar cada elemento RDD nativamente. A segunda opção permite que os usuários escolham uma representação mais eficiente em termos de memória de gráficos de objetos Java quando o espaço é limitado, ao custo de menor desempenho. A terceira opção é útil para RDDs que são muito grandes para se manter em RAM, mas caro para recalcular em cada uso.

Para gerenciar a memória limitada disponível, usamos uma política de *flush* LRU no nível da RDDs. Quando uma nova partição RDD é computada, mas não há espaço suficiente para armazená-lo, despejamos uma partição do RDD menos recentemente acessado, a menos que este é o mesmo RDD que aquele com a nova partição. Neste caso, mantemos a partição antiga na memória para impedir que partições de ciclismo do mesmo RDD dentro e para fora. Isto é importante porque a maioria das operações irá executar tarefas ao longo de todo o RDD, por isso, é provável que a partição que está na memória seja necessária no futuro. Até agora, está abordagem padrão funcionava bem em todas as aplicações, mas também fornecemos aos usuários controle adicional por meio "prioridade persistência" para cada RDD.

Finalmente, cada instância do Spark em um cluster tem seu próprio espaço de memória separado. Em trabalhos futuros, pretendemos investigar partilha de RDDs através de instâncias do Spark através de um gerenciador de memória unificado.

5.4 Support for Checkpointing

Embora o lineage sempre pode ser usado para recuperar RDDs após uma falha, esta recuperação pode ser demorada para RDDs com cadeias de lineage longos. Assim, pode ser útil verificar alguns RDDs para armazenamento estável.

Em geral, checkpointing é útil para RDDs com gráficos de lineage contendo dependências wide, tais como os conjuntos de dados *rank* em nosso exemplo PageRank (§3.2.2). Nestes casos, uma falha do nó no cluster pode resultar na perda de algum slice de dados a partir de cada um dos pais RDD, exigindo uma recomputação completa [20]. Em contraste, para RDDs com dependências *narrow* em dados de armazenamento estável, como os pontos no nosso exemplo de regressão logística (§3.2.1) e as listas de *links* em PageRank, checkpointing nunca pode valer a pena. Se um nó falhar, partições perdidas destes RDDs podem ser recalculadas em paralelo em outros nós, em uma fração do custo de replicar o RDD inteiro.

Spark atualmente fornece uma API para checkpointing (a flag REPLICATE para persistir), mas deixa a decisão de quais dados para o checkpoint para o usuário. No entanto, estamos investigando como executar checkpointing de forma automática. Como o nosso programador sabe o tamanho de cada conjunto de dados, bem como o tempo que levou a primeira computação, ele deve ser capaz de selecionar um conjunto ideal de RDDs para o checkpoint minimizar o tempo de recuperação do sistema [30].

Finalmente, nota-se que RDDs são somente leitura, e

[8]The cost depends on how much computation the application does per byte of data, but can be up to 2 for lightweight processing.

por conta disto torna-os mais simples o ponto de verificação do que a memoria compartilhada geral. Como a consistência não é uma preocupação, os RDDs podem ser escrito em segundo plano sem a necessidade de pausas de programas ou esquemas de snapshot distribuídos.

6 Evaluation

Avaliamos o Spark e RDDs através de uma série de experimentos no Amazon EC2, bem como benchmarks de aplicativos do usuário. Em geral, nossos resultados mostram o seguinte:

- Spark supera Hadoop em até 20X em aplicações de aprendizado de máquina e gráficas. O speedup/aceleraçãoo evita custos de I/O e desserialização armazenando dados na memória como objetos Java.
- Aplicações escritas por nossos usuários executam e escalam bem. Em particular, usamos o Spark para processar um relatório analítico que estava sendo executado no Hadoop em 40x.
- Quando nós falharem, o Spark pode recuperar-se rapidamente por resconstruindo apenas as partições RDD perdidas.
- O Spark pode ser usado para consultar um conjunto de dados de 1 TB interativamente com latências de 5-7 segundos.

Apresentamos os benchmarks para aplicações interativas aprendizagem de máquina (§6.1) e PageRank (§6.2) contra o Hadoop. Em seguida, avaliamos a recuperação de falhas do Spark (§6.3) e o comportamento quando um conjunto de dados não se encaixa em memoria (§6.4). Finalmente, discutimos os resultados de aplicações do usuário (§6.5) e mineração de dados interativa (§6.6).

A menos que assinalado de outro modo, os nossos testes utilizaram nós m1.xlarge EC2 com 4 cores e 15 GB RAM. Usamos HDFS para armazenamento, com blocos de 256 MB. Antes de cada teste, limpamos os caches de buffer do OS para medir os custos de IO com precisão.

6.1 Iterative Machine Learning Applications

Implementamos duas aplicações iterativas de aprendizagem de máquina, regressão logística e k-means, para comparar o desempenho dos seguintes sistemas:

- Hadoop: A versão estável do Hadoop 0.20.2.
- HadoopBinMem: implementação Hadoop que converte os dados de entrada em um formato binário de baixa sobrecarga na primeira iteração para eliminar análise de texto nos mais recentes, e armazena uma instância HDFS em memória.
- Spark: Nossa implementação de RDDs.

Executamos ambos os algoritmos com 10 iterações em um conjunto de dados de 100 GB usando 25-100 máquinas. A diferença chave entre as duas aplicações é a quantidade de computação que realizam por byte de dados. O tempo de iteração do k-means é tomado pela computação, enquanto regressão logística tem uma computação menos intensiva, portanto, mais sensitiva para o tempo gasto em deserialização e de I/O.

Como os algoritmos de aprendizagem típicos precisam de dezenas de iterações para convergir, relatamos o tempo da primeira iteração e das iterações subseqüentes separadamente. Descobrimos que os compartilhar dados via RDDs melhora muito as iterações de processamento futuras.

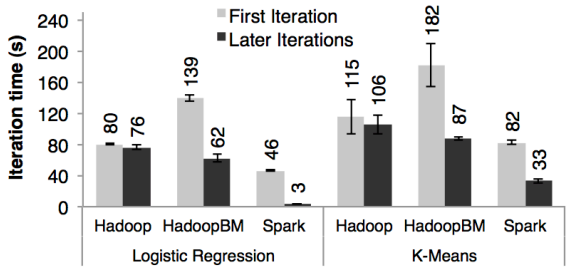


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

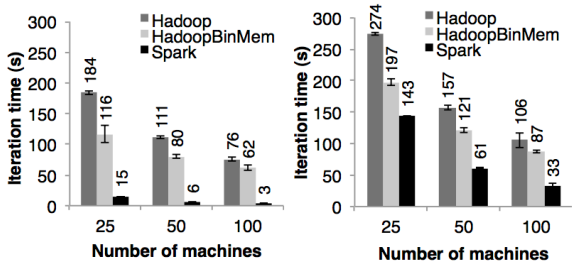


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Primeiras iterações Todos os três sistemas de leem a entrada de texto do HDFS em suas primeiras iterações. Como mostrado nas barras claras na Figura 7, o Spark foi moderadamente mais rápido do que o Hadoop nos experimentos. Esta diferença foi devida os overheads no protocolo heartbeat do Hadoop entre seu *master* e *slave*. HadoopBinMem foi o mais lento porque ele executou um MapReduce extra para converter os dados para binário, ele teve que escrever esses dados através da rede para uma instância HDFS em memória replicada.

Subsequentes iterações Figura 7 mostra os tempos médios de execução das iterações subsequentes, enquanto que a Figura 8 mostra como estes estão dimensionado com o tamanho do cluster. Para a regressão logistica, o Spark executou 25.3X e 20.7X mais rápido do que o Hadoop HadoopBinMem respectivamente em 100 máquinas. Para umaaplicação k-means, o Spark alcançou um processamento maior de 1.9X a 3.2X.

Compreender o Speedup Ficamos surpresos ao descobrir que o Spark superou até mesmo o Hadoop com armazenamento em memória dos dados binários (HadoopBinMem) por uma margem de 20x. Em HadoopBinMem, usamos o formato padrão binário do Hadoop (SequenceFile) e um tamanho de bloco de 256 MB, e foi forçado o diretório de dados do HDFS estar em um sistema de arquivos em memória. No entanto, o Hadoop ainda executou mais lento devido a alguns fatores:

1. Overhead minima do Hadoop software stack,
2. Overhead do HDFS ao servir os dados, e

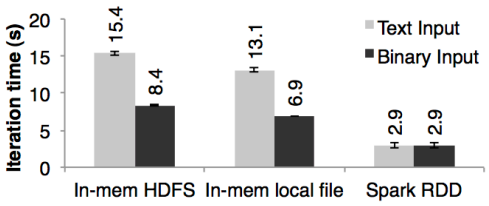


Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

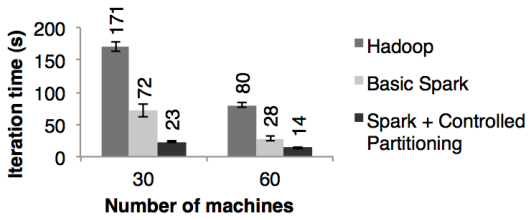


Figure 10: Performance of PageRank on Hadoop and Spark.

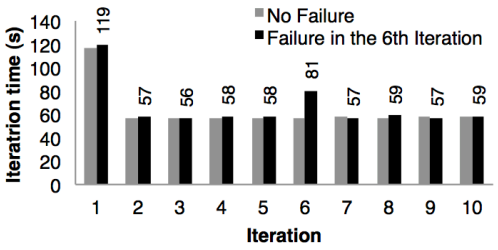


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

3. Deserialization cost to convert binary records to usable in-memory Java objects.

Investigamos cada um desses fatores, por sua vez. Para medir (1), executamos no-op Jobs no Hadoop, e viu-se que estes processaram em 25s para completar os requisitos mínimos de configuração do job de setup, iniciar as tarefas, e fazer a limpeza. Em relação a (2), descobrimos que HDFS realizou várias cópias de memória e uma soma de verificação para atender cada bloco.

Finalmente, para medir (3), executamos microbenchmarks em uma única máquina para executar a computação de regressão logística em 256 MB em vários formatos. Em particular, comparamos o tempo para processar texto e entradas digitais de ambos HDFS (onde overheads do *stack* do HDFS se manifestar) e um arquivo na memória local (onde o kernel passa transmitir eficientemente dados para o programa).

Mostramos os resultados destes testes na Figura 9. As diferenças entre o HDFS na memória e o arquivo local mostram que a leitura através HDFS introduziu 2 segundos de overhead, mesmo quando os dados estavam na memória da maquina local. As diferenças entre o texto e a entrada binária indicam que o overhead de análise foi de 7 segundos. Finalmente, mesmo quando se lê um arquivo na memória, a conversão dos dados binários pré-analisados em objetos Java levou 3 segundos, que ainda é quase trabalhoso quanto a própria regressão logística. Ao armazenar elementos RDD diretamente como objetos Java na memória, o Spark evita todas estes overheada.

6.2 PageRank

Comparamos o desempenho do Spark com o Hadoop para PageRank usando um total de 54 GB de dados do Wikipedia. Percorremos 10 iterações do algoritmo PageRank para processar um Link gráfico de aproximadamente 4 milhões de artigos. A figura 10
demonstra que o armazenamento na memória sozinho, proporcionou ao uma aceleração de 2.4x mais eficiente do que 30 nós Hadoop. Além disso, o controle do particionamento dos RDDs para torná-lo consistente em iterações, conforme discutido na Seção 3.2.2, melhorou a velocidade de processamento para 7.4X. Esses resultados também escalaram quase linearmente para 60 nós.

Também avaliamos uma versão do PageRank usando, usando nossa implementação do Pregel sobre Spark, que descrevemos na Seção 7.1. Os tempos de iteração foram semelhantes para os da Figura 10, mas mais lentos, cerca 4 segundos, porque Pregel executa uma operação extra em cada iteração para deixar os vértices "vote" se deve terminar o job.

6.3 Fault Recovery

Nós avaliamos o custo de reconstruir partições RDD usando *lineage* após a falha de nó na aplicação de k-means. A Figura 11 compara os tempos de execução de 10 iterações de k-means em um cluster de 75 nós no cenário normal de operação, onde um nó falha no início da sexta iteração. Sem qualquer falha, cada iteração consistiu em 400 *tasks* executando em 100 GB de dados.

Até o final da 5º iteração, os tempos de iteração foram cerca de 58 segundos. Na 6º iteração, uma das máquinas foi parada, resultando na perda das tasks em execução na máquina e as partições RDD armazenados lá. O Spark reexecutou estas tasks em paralelo em outras maquinas, onde eles releram os dados de entrada correspondente e reconstituíram RDDs através do lineage, que aumentou o tempo de iteração para 80s. Uma vez que as partições RDD perdidas foram reconstruídas, o tempo de iteração voltou para 58s.

Observe que, com um mecanismo de recuperação de falhas baseado em ponto de verificação, a recuperação provavelmente exigiria a reinicialização de pelo menos algumas iterações, dependendo da frequência de pontos de verificação. Além disso, o sistema precisaria replicar o conjunto de trabalho de 100 GB da aplicação (os dados de entrada de texto convertido em binário) através da rede, e consumiria o dobro da memória do Spark para replicá-lo na memória RAM, ou teria que esperar para escrever 100 GB no disco. Em contraste, os gráficos de *lineage* para os RDDs em nossos exemplos foram todos menos de 10 KB.

6.4 Behavior with Insufficient Memory

Até agora, asseguramos que todas as máquinas do cluster tinham memória suficiente para armazenar todos os RDDs através das iterações

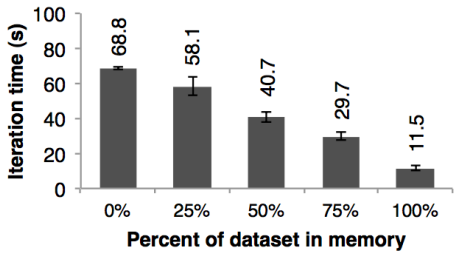


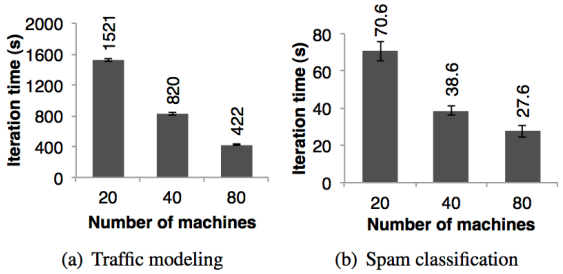
Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

A pergunta natural é como o Spark executado se não houver memória suficiente para armazenar os dados de um *job*. Neste experimento, configuramos o Spark para não usar mais do que um determinado percentual de memória para armazenar RDDs em cada máquina. São apresentados resultados para quantidades diferentes de espaço da *storage* para a regressão logística na Figura 12. Verificou-se que o desempenho diminui com menos espaço.

6.5 User Applications Built with Spark

In-Memory Analytics Conviva Inc, uma empresa distribuição de vídeo, usou o Spark para processar uma série de relatórios de dados de análise que anteriormente executavam no Hadoop. Por exemplo, um relatório foi executado como uma série de consultas Hive [1] que computavam várias estatísticas para um cliente. Essas consultas funcionaram no mesmo subconjunto de dados (registros que correspondem a um filtro fornecido pelo cliente), mas realizram agregações (médias, percentis e *count distinct*) em diferentes campos de agrupamento, exigindo processamentos separados de MapReduce. Ao implementar as consultas no Spark e carregar o subconjunto de dados compartilhados entre eles, uma vez em um RDD, a empresa foi capaz de processar o relatório por 40X mais rápido. Um relatório de 200 GB de dados compactados demorou 20 horas para processar em um cluster Hadoop, já no Spark no levou 30 minutos utilizando apenas duas máquinas. Além disso, o programa Spark precisava apenas de 96 GB de memória RAM, pois armazenava apenas as linhas e colunas correspondentes ao filtro do cliente em um RDD, não todo o arquivo descompactado.

Tráfego Modeling pesquisadores do projeto móvel Millennium em Berkeley [18] paralelizaram um algoritmo de aprendizagem para inferir o congestionamento do tráfego rodoviário a partir de medições de GPS de carros. Os dados de origem foram uma rede de 10.000 ligação rodoviária para uma área metropolitana, bem como 600.000 amostras de tempos de viagem ponto-a-ponto para automóveis equipados com GPS (tempos de viagem para cada caminho pode incluir várias ligações rodoviárias). Usando um modelo de tráfego, o sistema pode estimar o tempo que leva para percorrer as ligações rodoviárias individuais. Os pesquisadores treinaram este modelo usando um algoritmo de maximização expectativa (EM) que repete iterativamente duas etapas map e reduceByKey. A aplicação escalas quase linearmente de 20 a 80 nós com 4 núcleos cada, como mostrado na Figura 13 (a).



(a) Traffic modeling (b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

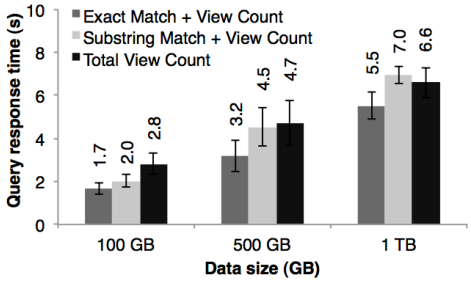


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Twitter Spam Classification no projeto Monarch, em Berkeley [29], foi usado o Spark para identificar link spam em mensagens do Twitter. Implementaram um classificador de regressão logística no Spark, semelhante ao exemplo na Seção 6.1, mas usaram um *reduceByKey* distribuído para somar os vetores gradientes em paralelo. Na Figura 13 (b) mostramos os resultados de escalonamento para treinar um classificador em um subconjunto de 50 GB de dados: 250.000 URLs e 107 características/dimensões relacionadas as propriedades da rede e conteúdo das páginas em cada URL. A escala não é tão linear devido a um custo de comunicação fixo maior por iteração.

6.6 Interactive Data Mining

Para demonstrar a capacidade do Spark de consultar interativamente grandes conjuntos de dados, foi utilizado para analisar 1TB de registros do Wikipedia (2 anos de dados). Para esta experiência, usamos 100 m2.4xlarge EC2 com 8 núcleos e 68 GB de memória RAM cada. Executamos consultas para encontrar uma visão total de (1) todas as páginas, (2) páginas com títulos que corresponde exatamente a uma determinada palavra, e (3) páginas com títulos parcialmente combinando uma palavra. Cada consulta escaneava os dados de entrada inteiros.

A Figura 14 mostra os tempos de resposta das consultas no conjunto de dados completo, metade e um décimo dos dados. Mesmo em 1 TB de dados, consultas no Spark levaram 5-7 segundos. Isto era mais do que uma ordem de magnitude rápida, do que trabalhar com dados em disco; por exemplo, consultando um arquivo de 1 TB de disco levou 170s. Isso mostra que RDDs fazem do Spark uma ferramenta poderosa para a mineração de dados interativa.

7 Discussion

Embora os RDDs parecerem oferecer uma interface de programação limitada devido à sua natureza imutável e suas transformações *coarse-grained*, os classificamos adequados para uma ampla classe de aplicações. Em particular, os RDDs podem expressar um número surpreendente de modelos de programação de clusters que até agora foram propostas como frameworks separados, permitindo aos usuários compor esses modelos em um programa (por exemplo, executar uma operação MapReduce para construir um gráfico e, em seguida, executar Pregel sobre ele) e compartilhar dados entre eles. Nesta seção, discutimos quais os modelos de programação que os RDDs podem expressar e por que são tão amplamente aplicáveis (§7.1). Além disso, discutimos outro benefício da informação de *lineage* nos RDDs que estamos buscando, que é facilitar a depuração em todos esses modelos (§7.2).

7.1 Expressing Existing Programming Models

Os RDDs podem expressar de forma eficiente uma série de modelos que até agora foram propostos de forma independente. 'Eficientemente', queremos dizer que não só podem ser usando RDDs para produzir a mesma saída, como programas escritos nestes modelos, mas que os RDDs também podem otimizar a execução dos frameworks, como manter dados específicos na memória, particioná-los para minimizar comunicação e recuperação de falhas de forma eficiente. Os modelos expressíveis que usam RDDs incluem:

MapReduce: Este modelo pode ser expresso usando as operações FlatMap e groupByKey no Spark, ou reduceByKey se houver um combinador.

DryadLINQ: O sistema DryadLINQ fornece uma gama mais ampla de operadores do que o MapReduce, como o tempo de execução geral do Dryad, mas estes são todos os operadores que correspondem diretamente às transformações RDD disponíveis no Spark (*map*, *groupByKey*, *join*, etc).

SQL: Como as expressões DryadLINQ, as consultas SQL executam operações paralelas de dados em um conjunto de registros.

Pregel: o Google's Pregel [22] é um modelo especializado para aplicações gráficas iterativas que, a princípio, parecem bastante diferentes dos modelos de programação orientados a conjuntos em outros sistemas. Em Pregel, um programa é executado como uma série de "*supersteps*", coordenados. Em cada "*supersteps*", cada vértice do gráfico executa uma função de usuário que pode atualizar o estado associado ao vértice, alterar a topologia do gráfico e enviar mensagens para outros vértices para uso no próximo "*superstep*". Este modelo pode expressar muitos algoritmos gráficos, incluindo caminhos mais curtos, correspondência bipartida e PageRank.

A observação chave que nos permite implementar este modelo com RDDs é que Pregel aplica a mesma função do usuário para todos os vértices em cada iteração. Assim, podemos armazenar os estados do vértice para cada iteração de um RDD e executar uma transformação em massa (*flatMap*) para aplicar esta função e gerar um RDD de mensagens. Podemos juntar este RDD com os estados do vértice para executar a troca de mensagens.

Além disso, os RDDs nos permitem manter os vértices em memória como Pregel, para minimizar a comunicação, controlando sua partição e recuperação parcial de falhas. Implementamos Pregel como uma biblioteca de 200 linhas no Spark, para mais detalhes consulte [33].

Iterative MapReduce: Vários sistemas propostos recentemente, incluindo HaLoop [7] e Twister [11], fornecem um modelo iterativo de MapReduce onde o usuário fornece ao sistema uma série de jobs MapReduce para fazer um loop. Os sistemas mantem os dados particionados consistentemente em iterações, e o Twister também pode mantê-los na memória. Ambas as otimizações são simples de expressar com RDDs, fomos capazes de implementar HaLoop como uma biblioteca de 200 linhas usando Spark.

Batched Stream Processing: recentemente pesquisadores propuseram vários sistemas de processamento incremental para aplicações que atualizam periodicamente um resultado com novos dados [21, 15, 4]. Por exemplo, um aplicativo que atualiza estatísticas sobre cliques de anúncios a cada 15 minutos, deveria ser capaz de combinar o estado intermediário da janela anterior de 15-minutos, com dados dos novos registros. Estes sistemas são capazes de realizar operações em massa, semelhante ao Dryad, mas armazenam o estado da aplicação no *filesystem* distribuído. Colocar os RDDs em um estado intermediário, aceleraria o seu processamento.

Explicando a Expressividade dos RDDs Por que os RDDs são capazes de expressar esses diversos modelos de programação? A razão é que as restrições aos RDDs têm pouco impacto, em muitas aplicações paralelas. Em particular, apesar que os RDDs só possam ser criados através de transformações em massa, muitos programas paralelos naturalmente aplicam a mesma operação para muitos registros, tornando-os fáceis de expressar. Da mesma forma, a imutabilidade dos RDDs não é um obstáculo, porque pode-se criar vários RDDs para representar versões do mesmo conjunto de dados. De fato, muitas aplicações de MapReduce atuais são executados em *filesystem* que não permitem atualizações de arquivos, como HDFS.

Uma última questão é por que as estruturas anteriores não ofereceram o mesmo nível de generalidade. Acreditamos que isso ocorre porque esses sistemas exploraram problemas específicos que o MapReduce e Dryad não manipulam bem, como a iteração, sem observar que a causa comum desses problemas era a falta de abstrações de compartilhamento de dados.

7.2 Leveraging RDDs for Debugging

Embora inicialmente concebido RDDS para ser deterministicamente recomputável para tolerância a falhas, esta propriedade também facilita depuração. Em particular, ao registrar um *lineage* de RDDs criados durante um *job*, é possível (1) reconstruir esses RDDs mais tarde e permitir que o usuário interaja de forma interativa e (2) executar novamente qualquer tarefa do *job* em um depurador de processo único, recuputando as partições RDD de que depende. Ao contrário dos depuradores de repetições tradicionais para sistemas distribuídos em geral [13], que deve capturar ou inferir a ordem dos eventos em vários nós, essa abordagem acrescenta praticamente quase zero de sobrecarga na gravação porque somente o gráfico de *lineage* RDD

precisa ser registado. Atualmente estamos desenvolvendo um depurador Spark baseodo nessas ideias [33].

8 Related Work

Modelos de programação em Cluster: Trabalhos relacionados em modelos de programação em cluster é dividido em várias classes. Em primeiro lugar, modelos de fluxo de dados como MapReduce [10], Dríade [19] e Ciel [23] suportam um rico conjunto de operadores para processamento de dados, mas compartilhar através de sistemas de armazenamento estáveis. Os RDDs representam uma abstração de compartilhamento de dados mais eficiente do que o armazenamento estável porque evitam o custo de replicação de dados, I/O e a serialização.

Em segundo lugar, várias interfaces de programação de alto nível para sistemas de fluxo de dados, incluindo DryadLINQ [31] e FlumeJava [8], fornecem APIs de integração com linguagens onde o usuário pode manipular "coleções paralelas" através operadores como *map* e *join*. No entanto, nestes sistemas, as coleções paralelas representam arquivos em disco ou conjuntos de dados efêmeros usados para expressar um plano de consulta. Embora os sistemas canalizem dados de entre operadores na mesma consulta (por exemplo, um *map* seguido por outro *map*), eles não podem compartilhar dados de forma eficiente entre consultas. Baseamos API do Spark no modelo de coleção paralela devido à sua conveniência e não reivindicamos novidade para a interface integração com linguagens, mas fornecendo RDDs como abstração de armazenamento por trás dessa interface, que lhe permitem apoiar uma classe muito mais ampla de aplicações.

Uma terceira classe de sistemas fornece interfaces de alto nível para classes específicas de aplicações que requerem compartilhamento de dados. Por exemplo, Pregel [22] suporta aplicações gráficas iterativas, enquanto Twister [11] e HaLoop [7] MapReduce iterativos em tempos de execução. No entanto, esses frameworks realizam o compartilhamento de dados implicitamente para o padrão de computação que elas suportam, e não fornecem uma abstração geral que o usuário pode empregar para compartilhar dados de sua escolha entre as operações de sua escolha. Por exemplo, um usuário não pode usar Pregel ou Twister para carregar um conjunto de dados na memória e, em seguida, decidir qual consulta executar. Os RDDs fornecem uma abstração de armazenamento distribuída de forma explícita e pode, assim, oferecer suporte a aplicações que esses sistemas especializados não entendem, como mineração de dados interativa.

Finalmente, alguns sistemas de expõem o estado mutável compartilhado para permitir que o usuário realize a computação na memória. Por exemplo, Piccolo [27] permite que os usuários executem funções paralelas que leem e atualizem células em uma tabela *hash* distribuída. Sistemas de memória compartilhada distribuída (DSM) [24] e *key-value stores* como RAMCloud [25] oferecem um modelo similhante. Os RDDs diferem destes sistemas de duas maneiras. Primeiro, RDDS fornecem uma interface de programação de alto nível com base em operadores como *map*, *sort* e *join*, enquanto que a interface em Piccolo e DSM é apenas leitura e atualizações das células da tabela. Em segundo lugar, os sistemas Piccolo e DSM implementam a recuperação através de *checkpoints* e *roll-back*, que é mais custoso do que a estratégia baseada em *lineage* de RDDs em muitas aplicações. Finalmente, como discutido na Seção 2.3, RDDs fornecem outras vantagens sobre o DSM, como a mitigação do retardador.

Sistemas de armazenamento em cache: Néctar [12] pode reutilizar resultados intermediários através de DryadLINQ identificando subexpressões comuns com análise de programa [16]. Esta capacidade seria convincente para adicionar a um sistema baseado em RDD. No entanto, Nectar não fornece cache em memória (ele coloca os dados em um sistema de arquivos distribuídos), nem permite que os usuários controlem explicitamente quais conjuntos de dados devem persistir e como particionar-los. Ciel [23] e FlumeJava [8] também podem armazenar em cache os resultados das tarefas, mas não fornecem cache em memória ou controle explícito sobre quais os dados são armazenados em cache.

Ananthanarayanan et al. propuseram a adição de um cache em memória para sistemas de arquivos distribuídos para explorar a localidade temporal e espacial de acesso aos dados [3]. Enquanto esta solução permite o acesso mais rápido aos dados que já está no sistema de arquivos, não é um meio mais eficiente de resultados intermediários dentro de uma aplicação como RDDs, porque ele ainda exigiria aplicações para escrever estes resultados no sistema de arquivos entre os estágios.

Lineage: Captura de informação no estágio *lineage* ou proveniência para dados tem sido um tema de pesquisa em computação científica e bancos de dados, para aplicações de como demonstrar os resultados, permitindo que possam ser reproduzidas por outros, e recalculando os dados se um bug é encontrado em um *workflow* ou se um conjunto de dados é perdido. Nós nos referimos o leitor a [5] e [9] para pesquisas deste trabalho. Os RDDs fornecem um modelo de programação paralela, onde o *fine-grained lineage* não é custoso para capturar, de modo que ele possa ser usado para a recuperação de falhas.

Nosso mecanismo de recuperação baseada em *lineage* também é semelhante ao mecanismo de recuperação usado em uma computação (job) em MapReduce e Dryad, que rastreia dependências entre um DAG de tarefas. No entanto, nestes sistemas, as informações de *lineage* são perdidas após a conclusão de um job, exigindo o uso de um sistema de armazenamento replicado para compartilhar dados entre computações. Em contraste, os RDDs aplicama o lineage para persistir os dados na memória de forma eficiente nas computações, sem o custo de replicação I/O de disco.

Bancos de dados relacionais: Os RDDs são conceitualmente semelhantes ao ponto de vista de um banco de dados, e os RDDs persistentes assemelham a visões materializadas [28]. No entanto, como os sistemas DSM, bancos de dados o acesso de leitura e gravação *fine-grained* a todos os registros, exigindo o registro de operações e dados para tolerância a falhas e um *overhead* adicional para manter a consistência.

[9]Unlike these systems, an RDD-based debugger will not replay non-deterministic behavior in the user's functions (e.g., a nondeterministic map), but it can at least report it by checksumming data.

[10]Note that running MapReduce/Dryad over an in-memory data store like RAMCloud [25] would still require data replication and serializa-tion, which can be costly for some applications, as shown in x6.1.

Estes overheads não são necessários com o modelo de transformação *coarse-grained* dos RDDs.

9 Conclusion

Apresentamos conjuntos de dados distribuídos resilientes (RDDs), uma abstração eficiente, de proposito geral e tolerante a falhas para compartilhamento de dados em aplicações em cluster. Os RDDs podem expressar uma ampla gama de aplicações paralelas, incluindo muitos modelos de programação especializados que foram propostos para computação iterativa e novas aplicações que outros modelos não suportam. Ao contrário das abstrações de armazenamento existentes para clusters, que exigem replicação de dados para tolerância a falhas, os RDDs oferecem uma API baseada em *coarse-grained* transformações que lhe permite recuperar dados de forma eficiente usando *lineage*. Implementamos RDDs em um sistema chamado Spark que supera o Hadoop até 20x em aplicações iterativas e podem ser usadas interativamente para consultar centenas de gigabytes de dados.

Abrimos Spark em spark-project.org como um meio para análise de dados escalável, e um centro de pesquisa de sistemas.

Acknowledgements

Agradecemos aos primeiros usuários do Spark, incluindo Tim Hunter,Lester Mackey, Dilip Joseph e Jibin Zhan, por testarem nosso sistema em suas aplicações reais, fornecendo boas sugestões e identificando alguns desafios de pesquisa pelo caminho. Agradecemos também ao nosso pastor, Ed Nightingale, e nossos revisores por seus comentários. Esta pesquisa foi apoiada em parte pela Berkeley AMP Lab, patrocinadores Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp e VMWare, pela DARPA (contrato # FA8650-11-C-7136), e pelo Conselho de Pesquisas de Ciências Naturais e Engenharia do Canadá.

References

[1] Apache Hive. ***http://hadoop.apache.org/hive***.

[2] Scala. ***http://www.scala-lang.org***.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In HotOS '11, 2011.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In ACM SOCC '11, 2011.

[5] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. ACM Computing Surveys, 37:1–28, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In WWW, 1998.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. Proc. VLDB Endow., 3:285–296, September 2010.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In PLDI '10. ACM, 2010.

[9] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. Foundations and Trends in Databases, 1(4):379–474, 2009.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In OSDI, 2004.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In HPDC '10, 2010.

[12] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In OSDI '10, 2010.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. OSDI'08, 2008.

[14] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Publishing Company, New York, NY, 2009.

[15] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In SoCC '10.

[16] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In ACM SIGPLAN Notices, pages 311–320, 2000.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In NSDI '11.

[18] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen. Scaling the Mobile Millennium system in the cloud. In SOCC '11, 2011.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In EuroSys '07, 2007.

[20] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In HotOS '09, 2009.

[21] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. SoCC '10.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In SIGMOD, 2010.

[23] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In NSDI, 2011.

[24] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. Computer, 24(8):52 –60, Aug 1991.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazieres,` S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. SIGOPS Op. Sys. Rev., 43:92–105, Jan 2010.

[26] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In OSDI 2010.

[27] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In Proc. OSDI 2010, 2010.

[28] R. Ramakrishnan and J. Gehrke. Database Management Systems. McGraw-Hill, Inc., 3 edition, 2003.

[29] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In IEEE Symposium on Security and Privacy, 2011.

[30] J. W. Young. A first order approximation to the optimum checkpoint interval. Commun. ACM, 17:530–531, Sept 1974.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In OSDI '08, 2008.

[32] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In EuroSys '10, 2010.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, UC Berkeley, 2011.