PUC Minas
Virtual

Ciência de dados e Big Data

Soluções para Processamento Paralelo e Distribuído de Dados

SÍNTESE DA DISCIPLINA

Cláudio Lúcio do Val Lopes

Sumário

1. Introdução

1.1 Motivação

Sistemas de computadores seqüenciais estão cada vez mais velozes, principalmente considerando a velocidade de processador, memória e comunicação com o mundo externo. No entanto neste mesmo cenário temos o crescimentos exponencial dos dados sendo gerados pela atual sociedade. As aplicações exigem computadores cada vez mais rápidos e estas aplicações requerem um grande poder de computação ou requerem o processamento de grandes quantidades de dados.

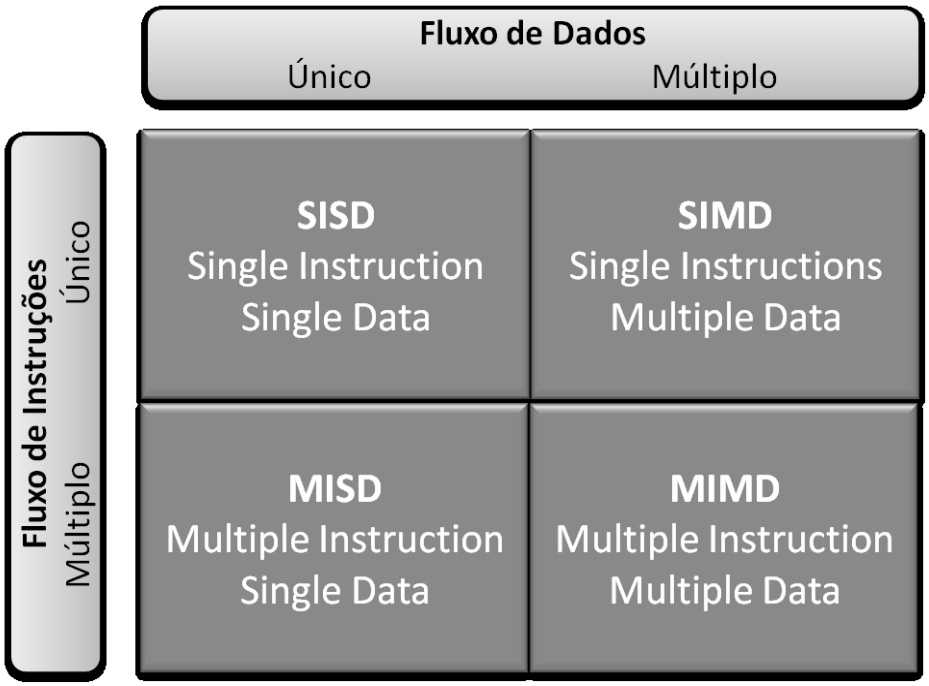
Precisamos de computação paralela e distribuída pois, a estrutura de arquivos convencional não é capaz de lidar com dados massivos, precisa-se de um paradigma que comporte escalabilidade elástica; além do mais bancos de dados relacionais utilizam conceitos, que às vezes, não são um requisito para dados massivos. Uma infra estrutura tolerante a falhas e que permita computação paralela é necessária.

1.2 Arquitetura de soluções distribuídas

Algumas características interessantes de soluções paralelas:

- Conectividade ⇒ rede de interconexão
- Heterogeneidade ⇒ hardware e software distintos
- Compartilhamento ⇒ utilização de recursos (memória, disco, CPU)
- Imagem do sistema ⇒ como usuário o percebe
- Escalabilidade ⇒ mais estações de trabalho levam a melhor desempenho/eficiência

Uma das metodologias mais conhecidas e utilizadas para classificar uma arquitetura de computadores ou conjunto de computadores é a taxonomia de *Flynn* (1966). Duas dimensões: instruções e dados; Cada dimensão assume dois valores distintos: *single* ou *multiple*;



SISD – *Single Instruction Single Data*

- Arquitetura dos computadores com um único processador;
- Apenas uma instrução é processada a cada momento.
- Apenas um fluxo de dados é processado a cada momento.
- Exemplos: PCs, workstations e servidores com um único processador.

SIMD – *Single Instruction Multiple Data*

- Tipo de arquitetura paralela desenhada para problemas específicos (alto padrão de regularidade nos dados, ex.:processamento de imagem);
- Todas as unidades de processamento executam a mesma instrução a cada momento;
- Cada unidade de processamento pode operar sobre um fluxo de dados diferente;
- Exemplos: Computadores com unidades de processadores gráficos (GPUs);

MISD – *Multiple Instruction Single Data*

- Arquitetura paralela desenhada para problemas caracterizados por um alto padrão de regularidade funcional (ex.: processamento de sinal);
- Constituída por uma pipeline de unidades de processamento independentes que operam sobre um mesmo fluxo de dados enviando os resultados de uma unidade para a próxima;
- Cada unidade de processamento executa instruções diferentes a cada momento;
- Exemplos:Não existem exemplos práticos

MIMD – *Multiple Instruction Multiple Data*

- Arquitetura paralela predominante;
- Cada unidade de processamento executa instruções diferentes a cada momento;
- Cada unidade de processamento pode operar sobre um fluxo de dados diferente;
- Exemplos: alguns supercomputadores, clusters de computadores paralelos em rede, PCs multi-core;

Plataformas MIMD podem ser agrupadas em quatro grupos: SMPs (*Symmetric MultiProcessors*), MPPs *(Massively Parallel Processors*), Cluster ou NOWs (*Network Of Worstations*) e Grades Computacionais

Os Clusters possuem espaço de endereçamento não compartilhado, mecanismo de comunicação com troca de mensagens, unidades de processamento podem ser conjunto de estações de trabalho ou Pcs (Nós: elementos de processamento = processador + memória). Além disto ainda mantém características como: sistema homogêneo ou heterogêneo, a interconexão pode ser por redes locais e tendem a ser mais lentas que no MPP, não possuem um escalonador centralizado, cada nó tem seu próprio escalonador loca e ainda contam com a possibilidade de compor um sistema de alto desempenho e um baixo custo (principalmente quando comparados com MPP).

1.3 Paradigmas de processamento paralelo

Para uma aplicação ser paralela existem fatores que limitam o desempenho, tais como: código sequencial (existem partes do código que são inerentemente seqüenciais), concorrência (o número de tarefas pode ser escasso e/ou de difícil definição), granularidade (o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo - custos de criação, comunicação e sincronização) e balanceamento de Carga (ter os processadores o máximo ocupados é decisivo para o desempenho global do sistema.

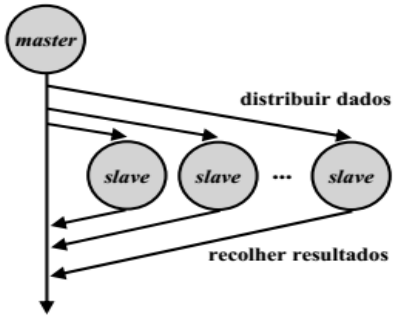
Os principais paradigmas de programação paralela abrangem diversos problemas aplicáveis. O desenvolvimento de algoritmos paralelos pode ser classificado em diferentes paradigmas.Cada paradigma representa uma classe de algoritmos similares.

Master/Slave

Divide a computação em duas entidades distintas:

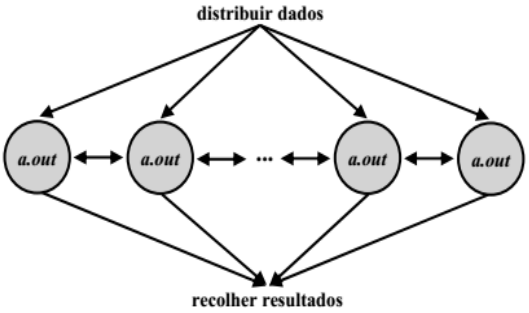
- *Master*: é o responsável por decompor o problema em tarefas, distribuir as tarefas pelos slaves e recolher os resultados parciais dos slaves de modo a calcular o resultado final

- *Slaves*: responsabilidade trivais e simples: obter uma tarefa do master, processar a tarefa e enviar o resultado de volta para o master.



Single Program Multiple Data (SPMD)

-Consiste em processos que executam o mesmo programa (executável) mas sobre diferentes partes dos dados.

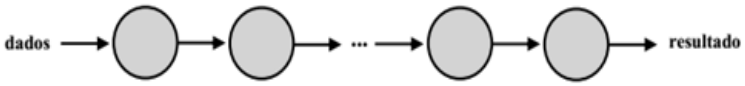


Data Pipelining

- Utiliza uma decomposição funcional do problema em que cada processo executa apenas uma parte do algoritmo completo e total;

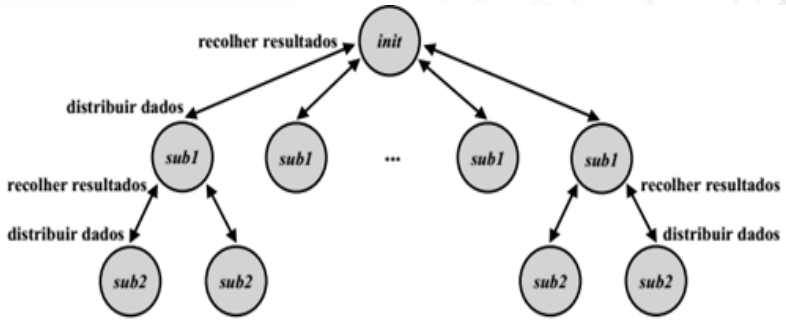
- O padrão de comunicação é definido e simples: processos são organizados em sequência (pipeline) e cada processo só troca informação com o processo seguinte;

- Dependente muito da capacidade de balancear a carga entre as diferentes etapas da pipeline;



Divide and Conquer

- Utiliza uma divisão recursiva do problema inicial em subproblemas independentes instâncias mais pequenas do problema inicial) cujos resultados são depois combinados para obter o resultado fina



1.4 Exemplos de produtos e ferramentas

Há muitos exemplos de produtos, incluindo produtos específicos por fornecedores. No entanto será tratado aqui produtos mais gerais, tais como : *Hadoop* e *Spark*. Outros exemplos são o *Storm* e *Flink*

2. Hadoop

Plataforma para acesso a dados estruturados, semi estruturados(logs, tweets, sensor data) e não estruturados. Permea todo o ciclo de vida dos dados: obter, reter e analisar grandes volumes de dados; É *Open Source* e mantida pela fundação Apache. Algumas das suas características são:

– Escalabilidade;
– Replicação de dados distribuída;
– Utilização de '*commodity hardware*' (NOW - Number Of Workstation);

Possui os seguintes componentes essenciais:

HDFS: Sistema de arquivos distribuídos para dados estruturados, semi-estruturados e não estruturados. Arquivos são divididos em bloco e armazenados com redundância no cluster;

Map Reduce:Framework para execução de processamento paralelos em múltiplos nós de trabalho para posteriormente combinar os resultados;

YARN(Hadoop 2.0): Gerenciamento da execução das aplicações no cluster;

Demais componentes (exemplos):

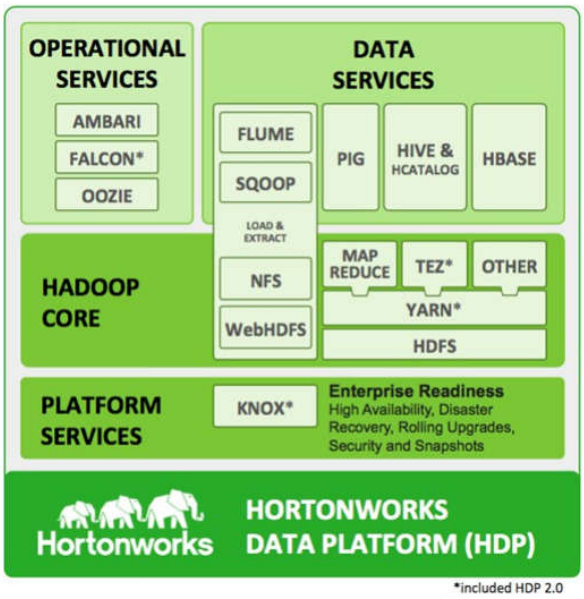
Flume: Serviço distribuído para coleta, agregação e movimentação de streams de logs de dados para o HDFS;

Sqoop: Movimentação de dados para dentro do Hadoop a partir de bancos de dados relacionais e vice versa;

PIG: Possibilitar a escrita de programas de transformação e movimentação de dados utilizando uma linguagem simples de script;

Hive: Interface SQL para o Hadoop que possibilita sumarização de dados, consultas ad-hoc e análise de grandes volumes de dados;

HBASE: Banco de dados NoSQL(colunar -'big table clone') para utilização de dados de forma interativa (não batch) ;



2.1 Sistema de arquivo distribuído

Características DFS(sistema de arquivo distribuído) são:

– Arquivos devem ser 'grandes', gigabytes, pelo menos; Arquivos menores não fazem sentido no DFS;

– Arquivos no DFS são raramente atualizados (write-once read-many). Adicionalmente dados são adicionados para os arquivos (periodicidade, processamento batch);

– Arquivos são divididos em partes ('chunks' ou blocos), normalmente 64 megabytes e replicados para, pelo menos, 3 nós (em racks diferentes);

– As informações dos blocos e replicas é controlado utilizando metadados e com um figura central no cluster: 'name node' ou 'master node'; O Name node: Gerencia o sistema de arquivos(réplicas, blocos, nós e racks):abrir, fechar, renomear arquivos; Também gerencia o acesso dos clientes ao arquivos;

– Os outros nós do cluster são chamados de 'data node'ou 'slave node'; Executam as operações enviados pelo 'Name node': criação, exclusão e replicação de blocos;

2.2 Map Reduce

A implementação do algoritmo é utilizada para realizar computação no DFS para arquivos 'grandes' e com execução tolerante a falha; É necessário escrever as duas funções: Map e reduce; O sistema lida com os demais detalhes: Execução paralela; Coordenação de tarefas (Map e reduce); Lidar com a tolerância a falhas;

Seguintes passos de execução:

– Um arquivo é armazenado no DFS com vários blocos em vários nós e racks;

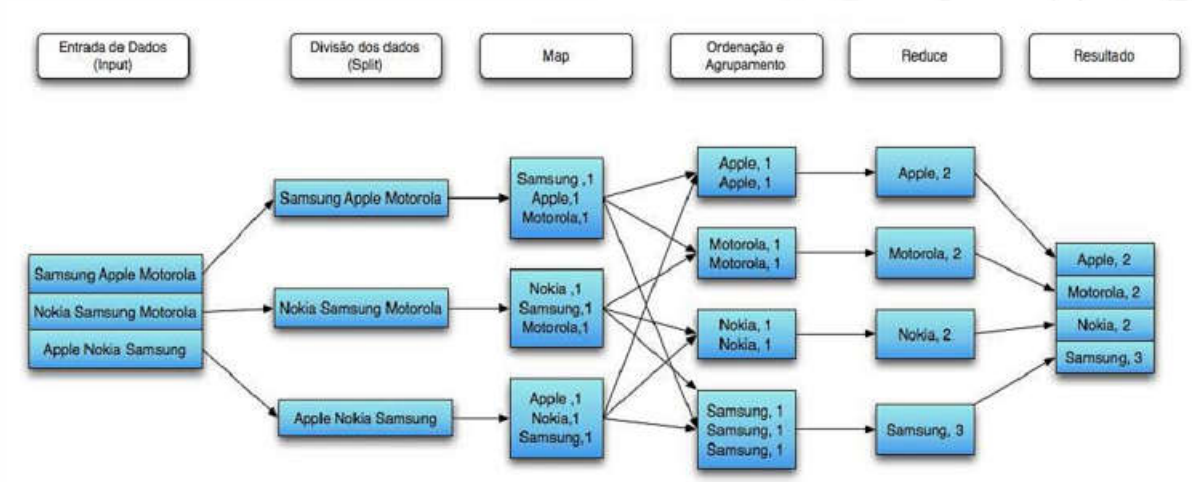
– Um conjunto de tarefas do tipo Map é criado, para cada Map existe um ou mais blocos que serão processados; As tarefas Map vão transformar o dado em um estrutura chave valor ou tuplas;

– As estruturas chave valor são coletadas pelo controlador master e ordenadas pelas suas chaves;

– As chaves serão agrupadas e divididas para as tarefas do tipo Reduce (uma chave, com vários valores será processado por uma e só uma tarefa Reduce);

– As tarefas do tipo Reduce vão então agrupar os dados pelas chave,uma por vez.

Exemplo de funcionamento map reduce:



2.3 YARN

O Yarn é gestor de recursos que foi criado pela comunidade e é usado no Hadoop 2.0 para melhorar o processo de gestão de recursos do Hadoop. Possui os seguintes componentes: Resource Manage, Application Manager e Node Manager.

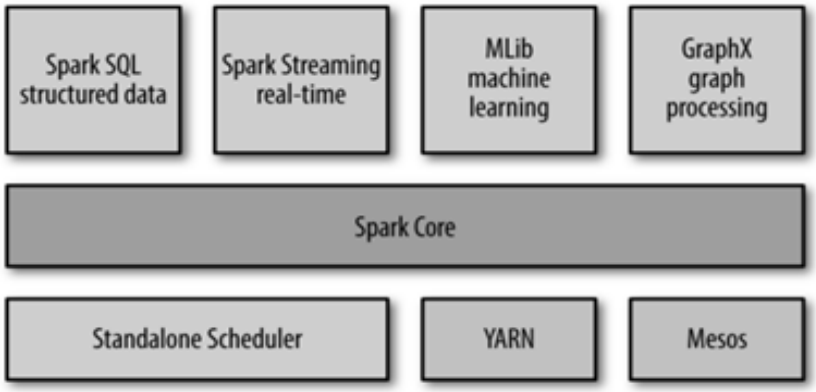
O Node Manager é mais flexível e eficiente que o TaskTracker(sua versão anterior – no hadoop 1.0). Executa qualquer tipo de computação que faz sentido no contexto do Application Manager: não somente map e reduce. Possui o conceito de containers que são capazes de lidar com recursos variáveis (RAM, CPU, IO) e não exigem número de funções map´s e reduce pré definidas;

3. Spark

3.1 Conceitos

É uma plataforma (em cluster) para processamento rápido, paralelo e de propósito geral que extende o algoritmo padrão: *MapReduce*. Muitos cálculos são feitos em memória (podem ser executados no disco também). Possui *API´s* para *Java*, *Scala*, *Python*, *SQL* e *R*.

Tem o propósito de ser um ambiente único para execução de vários tipos de processamentos: Batch, processamentos interativas e *streamings*. Possui 4 módulos: aprendizado de máquina, Grafos, processamento de consultas e *streaming*;

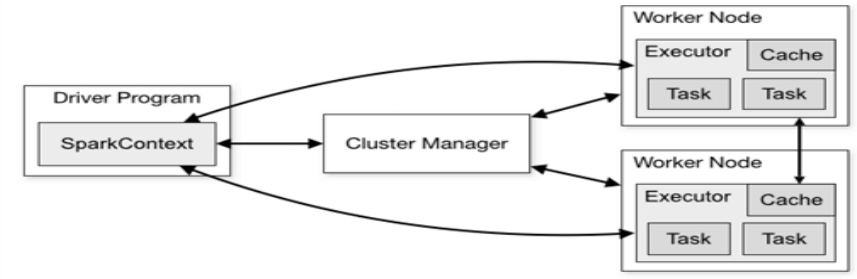


Conceitos importantes em uma aplicação Spark:

- *Driver Program:* Qualquer aplicação *Spark* é um *'Driver program'*. Esta aplicação é capaz de fazer operações paralelas no cluster *Spark*. É como se fosse uma função principal (*main*) e define/cria dados distribuídos no cluster e aplica operações paralelas aos dados distribuídos no cluster. As operações não são apenas *Map* e *Reduce*;

- Contexto *Spark:* O '*Driver program*' acessa o *spark* através de um objeto contexto que na verdade, representa a conexão da aplicação com o Cluster. Possui alguns parâmetros de configuração.

- *Executors*: Para realizar as operações do *'Driver Program*', o mesmo faz uso de vários nós que possuem os executores

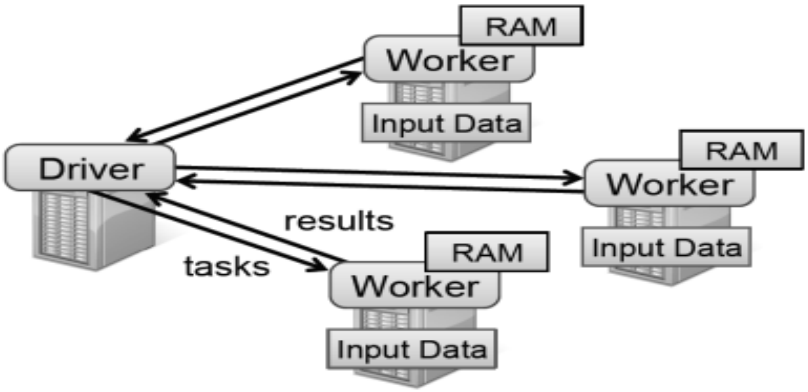


3.2 RDD´s

É uma coleção distribuída de elementos. A lógica de trabalho no Spark é criar RDD´s, transformar RDD´s existentes e executar operações nos RDD´s para gerar resultados. De forma transparente o *Spark* distribui os dados dos RDD´s no *cluster* e paraleliza as operações executadas nos RDD´s.

RDD´s representam um conjunto distribuídos de objetos 'imutáveis' (não volatéis).Isto facilita o processo de distribuição, replicação e compartilhamento dos dados. Tais dados imutáveis podem permanecer em disco ou memória de maneira indistinta.

RDD´s são divididos em várias *'partitions*' que podem residir em vários nós do Cluster. A criação de RDD´s é feito de duas maneiras: carregando dados de forma externa, por exemplo, ou distribuindo uma coleção de objetos no programa Driver. O *Driver program* é quem coordena as atividades e os *Workers* armazenam e manipulam as partições dos RDD´s



Uma vez criado os RDD´s, duas operações são possíveis. As *transformations* criam um RDD a partir de um anterior (exemplo é o *transformation filter*).

Já as *actions* calculam resultados a partir de um RDD. Este resultado é então enviado para o *Driver program* ou pode ser salvo, por exemplo, no HDFS. Retornam algum outro tipo de dados, com exceção de RDD. Os valores de retorno são enviados para o *Driver program* ou persistidos em disco.Por padrão todas as vezes que um action é acionada os RDD´s envolvidos são recalculados.

As transformações (*transformations*) são sempre postergadas até encontrar uma ação (action). O conceito de avaliação tardia('*Lazy evaluation*') em conjunto com os DAG´s fazem com as otimizações de código no Spark sejam muito boas.

Para reusar o RDD em várias '*actions*' o comando '*persist*' deve ser explicitado Há algumas opções para persistência dos dados. Uma vez que o '*persist*' é utilizado, não haverá mais recálculos quando houver outras '*actions*'. Pragmaticamente, é necessário persistir os dados em memória uma vez e fazer várias consultas (*'actions'*) nestes dados em memória;

StorageLevel	Espaço usado	CPU	Em memória	Em Disco
MEMORY_ONLY	Alto	Baixo	S	N
MEMORY_ONLY_SER	Baixo	Alto	S	N
MEMORY_AND_DISK	Alto	Médio	Parcial	Parcial
MEMORY_AND_DISK_SER	Baixo	Alto	Parcial	Parcial
DISK_ONLY	Baixo	Alto	N	S

3.3 Transformações e Ações

Uma vez criado os RDD´s, duas operações são possíveis. As *transformations* criam um RDD a partir de um anterior (exemplo é o *transformation filter*). As *actions* calculam resultados a partir de um RDD.

Exemplos de *transformations:*

- *Map*() : aplica uma função a cada elemento do RDD, e os elementos gerados pela função formam o novo RDD
- *Filter*(): aplica uma função e retorno o conjunto de elementos filtrados com um novo RDD
- *FlatMap*() : Similar ao Map com a diferença que para cada item de entrada pode ser gerada um mais itens de saída;
- *Distinct*() = para gerar elementos distintos de um RDD;
- *Union*() = faz união dos dados dos conjuntos. As repetições são mantidas;
- *Intersection*() = retorna os elementos existente em ambos os RDD´s (elimina os duplicados = atua como o distinct(), mantendo sua obsevação quanto ao desempenho);
- *Subtract*() = retorna os valores existentes apenas no primeiro RDD e que não existem no segundo RDD (tome cuidado com o desempenho);
- *Cartesian*() = realiza um produto cartesiano entre os dois RDD´s envolvidos.

Exemplos de *actions:*

- *reduce*(): Aplica uma função em dois elementos (tipo de dados) do RDD gerando um resultado do mesmo tipo;
- *fold*(): Faz a mesma coisa que o Reduce, mas atribui um valor inicial para cada elemento na soma;
- *collect*(): Forma mais simples de obter todo conteúdo de um RDD;
- *take*(n): Retorna os n elementos do RDD;
- *takeSample* (withReplacement, num, seed): Faz uma amostragem com ou sem reposição (assume uma distruibuição uniforme dos dados);
- *foreach*(): executa uma função em cada elemento do RDD, mas não retorna nada para o Driver ;

- *count*(): retorna a quantidade de elementos do RDD;

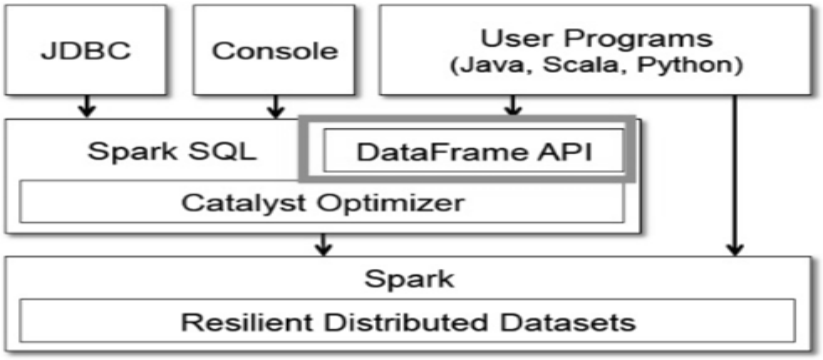
3.4 Spark SQL

É a proposta Spark para lidar com dados estruturados e semi-estruturados (dados com algum tipo de esquema). É capaz de carregar vários tipos de dados(Json, Hive, Parquet). As consultas são realizadas utilizando SQL (através de um programa spark ou através conexões externas: JDBC/ODBC). Ainda possui forte integração para ser utilizada com as linguagens *spark*: *Java*, *Scala* e *Python*. Permitir mesclar um RDD com dados do SQL Spark.

Os dados carregados com SQL ainda são RDD´s. Pode possuir forte integração com o Hive. São similares a tabelas em bancos de dados relacionais e os resultados das operações de carga dos dados ou SQL retornam *SchemaRDD´*s. Internamente os *SchemaRDD´*s são um conjunto de objetos do tipo *Row* e cada objeto do tipo *Row* é uma espécie de vetor com campos definidos pelo *Schema*. Como o *SchemaRDD* é ainda um RDD, todos os métodos utilizados para o RDD podem ser usados no *SchemaRDD.*

3.5 DataFrame

É um tipo de objeto com vários métodos para manipular estruturas de dados com linhas e colunas (tabelas). Pode ser usado por todas as linguagens que acessam o Spark:



Algumas operações comuns:

- Tipo de dados das colunas: `DF.printSchema()`

- Obtenção das primeiras linhas do data frame: `DF.head(5)`

- Número de linhas: `DF.count()`

- Número de colunas: `len(DF.columns)`

- Estatísticas descritivas para colunas numéricas: `DF.describe()`

- Mostrando somente algumas colunas: `DF.select('Col1','Col2').show(5)`

- Quantidade de elementos distintos em uma coluna: `DF.select('Col1').distinct().count()`

- Eliminado linhas duplicadas em um data frame: `DF.select('Col1','Col2').dropDuplicates().show()`

- Excluindo linhas com valores nulos: `DF.dropna()`

- Preenchendo valores nulos com outros valores: `DF.fillna(-1).show(2)`

- Fitlrar linhas: `DF.filter(DF.Col1 > 15000)`

- Geração de tabelas cruzadas(crosstab): `DF.crosstab('Col1', 'Col2').show()`