UNIVERSIDADE FEEVALE

FELIPE ALFREDO KUNZLER

AVALIAÇÃO DE DESEMPENHO ENTRE ALGORITMOS DISTRIBUÍDOS PARA MINERAÇÃO DE *ITEMSETS* FREQUENTES NO APACHE SPARK

Novo Hamburgo
2019

FELIPE ALFREDO KUNZLER

AVALIAÇÃO DE DESEMPENHO ENTRE ALGORITMOS DISTRIBUÍDOS PARA MINERAÇÃO DE *ITEMSETS* FREQUENTES NO APACHE SPARK

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Juliano Varella de Carvalho

Novo Hamburgo
2019

AGRADECIMENTOS

Agradeço a todos que de alguma forma me auxiliaram e suportaram durante o desenvolvimento deste trabalho.

Aos meus amigos, família e colegas que convivem comigo diariamente.

E agradeço especialmente ao meu orientador Juliano, por toda ajuda durante este período de um ano me acompanhando.

RESUMO

O constante aumento no volume de dados produzido todos os dias por novas tecnologias trouxe diversos desafios que se referem ao processamento destes dados em tempos aceitáveis. A etapa de geração de *itemsets* frequentes em algoritmos de mineração de regras de associação é de grande importância e ao mesmo tempo computacionalmente custosa. Isso faz com que implementações sequenciais convencionais com os algoritmos Apriori e FP-Growth não sejam viáveis à medida que o volume de dados aumenta. O modelo de programação Map Reduce, proposto pela Google, permite que programas sejam expressos através de duas funções: *map* e *reduce*, que podem ser executadas paralelamente por centenas ou milhares de computadores, viabilizando o processamento de altos volumes de dados. Entretanto, Map Reduce, quando utilizado o Apache Hadoop, só possibilita o reuso de dados entre diferentes tarefas através da escrita e leitura de dados pelo disco. Assim, diversos algoritmos de propriedade iterativa, que precisam aplicar funções repetitivamente sobre os mesmos dados, acabam por não obter tempos de execução ótimos quando implementados com Map Reduce sobre Apache Hadoop. Uma nova abordagem para a computação de alto volumes de dados é proposta pelo Apache Spark. Focando no processamento distribuído em memória, o Spark possibilita o reuso de dados entre diferentes tarefas através da memória primária ao invés do disco rígido, trazendo melhor desempenho na execução em várias classes de algoritmos. Desta forma, neste trabalho se propõe o estudo e a investigação da plataforma Apache Spark, assim como o desenvolvimento e a avaliação de desempenho de três algoritmos distribuídos YAFIM, R-Apriori e DFPS para mineração de itemsets frequentes baseados no Apriori e FP-Growth. Os resultados dos algoritmos foram analisados e comparados, variando o volume de dados e outros parâmetros como o número de nodos do *cluster*. O algoritmo DFPS demonstrou um desempenho superior na maior parte dos experimentos, seguido por R-Apriori e YAFIM, contudo, foram observadas algumas exceções dependendo de certas características do *dataset*.

Palavras-chave: Apache Spark. Computação distribuída. Mineração de *itemsets* frequentes. Big Data.

ABSTRACT

The constant increase of data that is produced every day has brought several challenges when it comes to extracting meaningful information and processing this data in acceptable times. Generating frequent itemset in association rule mining algorithms is of great importance but very CPU intensive. Because of that, conventional sequential implementation such as Apriori and FP-Growth are not feasible as the amount of data being processed increases. The Map Reduce programming model proposed by Google allows programs to be expressed by two operations: map and reduce, these are then parallelized automatically by the framework, which in turn can scale using additional machines as the amount of data grows. However, reusing data within distinct tasks is usually done writing and reading from disk in Map Reduce. Because of that, many algorithms of iterative nature, that apply a function several times on the same data, have an overhead of disk I/O when implemented in Map Reduce. A new approach to this property is proposed with Apache Spark. Focusing on distributed computing in-memory, data can be reused within distinct tasks without the need to use an external disk, bringing improvements on performance for many classes of algorithms. With that in mind, it is proposed in this paper to further investigate the Apache Spark platform, by developing and evaluating the performance of the distributed algorithms YAFIM, R-Apriori and DFPS for frequent itemset mining based on Apriori and FP-Growth. Experiments were conducted in order to analyze and compare these three distributed algorithms, varying the dataset volume and the number of nodes in the cluster. DFPS has shown a better performance overall, followed by R-Apriori and YAFIM, however, there were a few exceptions to this rule depending on a few characteristics of the *datasets.*

Keywords: Apache Spark. Distributed computing. Frequent Itemset Mining. Big Data.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
RMI	*Remote Method Invocation*
MPI	*Message Passing Interface*
HDFS	*Hadoop Distributed File System*
YARN	*Yet Another Resource Negotiator*
DAG	*Directed Acyclic Graph*
RDD	*Resilient Distributed Dataset*
JVM	*Java Virtual Machine*
I/O	*Input/Output*
EMR	*Elastic MapReduce*
EC2	*Elastic Compute Cloud*

SUMÁRIO

1 INTRODUÇÃO

Nunca antes tanta informação foi gerada e disponibilizada para consulta e processamento. A cada segundo, grandes quantidades de dados surgem e são transferidas para todos os cantos do mundo. A digitalização em massa que vem ocorrendo nos últimos anos, somada a grande disponibilidade de dispositivos conectados à internet e ligados a sensores, já são estimados por ultrapassar a população humana em quantidade. Assim, disponibilizando altos volumes de dados que podem ser levados em consideração para análise e criação de valor (DE MAURO; GRECO; GRIMALDI, 2014).

Esse conceito se tornou popular como *Big Data*, cenário onde o volume de dados, o número de transações e as diferentes fontes de informação são tão grandes e complexas, que requerem ferramentas, tecnologias e processos especializados para permitir a viabilização do seu processamento em tempos aceitáveis (SU; PATTNAIK, 2016).

Big Data é comumente associado aos três V's, sendo estes: Volume, Velocidade e Variedade. O Volume se refere ao montante de dados, indo de *gigabytes*, *terabytes* até *zettabytes*. A Velocidade é a alta taxa de transferência e atualização dos dados, criando uma corrente constante de novos dados, que acaba por deixar uma estreita janela para o processamento em tempo real. Por último, a Variedade engloba as diferentes fontes de onde os dados são gerados, assim como os diferentes formatos, podendo ser estruturados como em um banco de dados relacional; semiestruturados como arquivos XML; e não estruturados, como textos, imagens e vídeos (SU; PATTNAIK, 2016).

Mineração de regras de associação é um método popular que permite que padrões e tendências sejam encontradas a partir de um banco de dados contendo transações. Uma das aplicações mais populares para este tipo de algoritmo é na análise de compras de um supermercado, mostrando tendências entre produtos frequentemente comprados juntos, como por exemplo, a compra de queijo geralmente implica na compra de presunto. Algoritmos sequenciais convencionais para mineração de regras de associação como Apriori e FP-Growth funcionam em um número de diferentes cenários, contudo, não escalam adequadamente com o aumento do volume de dados. Dessa forma, seu processamento não pode ocorrer usando infraestruturas e ferramentas tradicionais (XIUJIN; SHAOZONG; HUI, 2017).

Map Reduce é um modelo de programação paralelo proposto pela Google em 2004. Esse modelo permite que programas sejam expressos a partir de duas funções principais: o *map*, onde os dados são inicialmente transformados em uma lista de pares contendo chaves e valores;

e o *reduce*, que é responsável por processar o conjunto de valores de uma determinada chave. Essa abstração foi inspirada em linguagens de programação funcionais como Lisp, que primam pelo uso de funções sem efeitos colaterais, assim como a imutabilidade, facilitando a paralelização de programas (DEAN; GHEMAWAT, 2004).

O modelo funcional do Map Reduce permite que cada função *map* e *reduce* seja executada isoladamente por dezenas ou centenas de computadores. Assim proporcionando a paralelização e distribuição do processamento em larga escala de forma automática. O *framework* de implementação do Map Reduce fica então responsável pelos detalhes técnicos da distribuição, como o particionamento dos dados de entrada, a divisão das tarefas entre os nodos, a tolerância a falhas, e a comunicação pela rede (DEAN; GHEMAWAT, 2004).

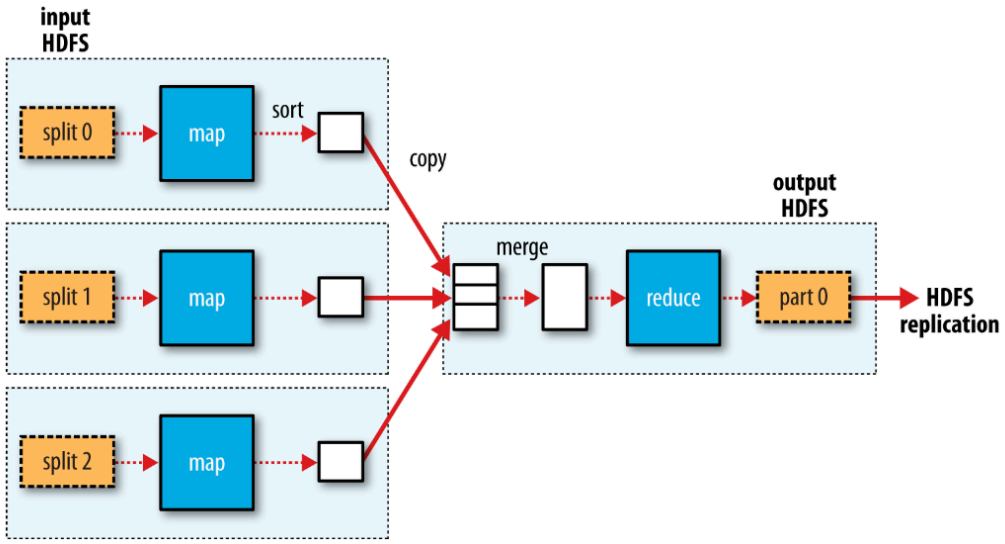


Figura 1 - Modelo de execução simples de um programa Map Reduce.
Fonte – White (2015)

A Figura 1 ilustra o fluxo de dados de um programa Map Reduce simples, que contém apenas uma tarefa de *reduce*. Cada bloco pontilhado azul representa um nodo no *cluster*, que é responsável por processar uma parte dos dados (*splits*) através da função *map.* Após este processamento, os resultados de cada nodo são ordenados e enviados para o nodo *reducer*, onde o conjunto de chaves e valores serão processados pela função *reduce* especificada (WHITE, 2015).

Apache Hadoop foi um dos primeiros e mais populares *frameworks* a prover uma implementação para o modelo de programação Map Reduce. São fornecidos módulos que permitem o processamento distribuído de grandes conjuntos de dados em *clusters* de computadores, que podem escalar de poucas dezenas até milhares de máquinas. Os principais módulos são: *Hadoop Map Reduce*, a implementação do modelo para processamento paralelo

proposto pela *Google; Hadoop YARN*, responsável pelo gerenciamento de recursos do *cluster*; *Hadoop Distributed File System* (HDFS), um sistema de arquivos distribuído de alta performance (APACHE SOFTWARE FOUNDATION, 2018a).

Fundamentalmente, Hadoop Map Reduce é um sistema de processamento em *batches*, com foco na alta taxa de transferência de dados (WHITE, 2015). Porém, essa arquitetura só possibilita o reuso de dados entre diferentes *jobs* através de escritas e leituras em disco, que pode acabar por gerar uma sobrecarga no sistema de arquivos durante a execução de algoritmos que dependem desta propriedade (SAMADI; ZBAKH; TADONKI, 2017).

Algoritmos de computação iterativa, como muitos em aprendizado de máquina e análise de grafos, necessitam aplicar funções repetitivamente sobre os mesmos dados para alcançar seus objetivos. Dessa forma, cada iteração é geralmente representada por um *job* Map Reduce, que precisa recarregar os dados processados pela iteração anterior por disco. Assim, a execução desta classe de algoritmos usando Hadoop Map Reduce acaba por não obter tempos de execução ótimos (SAMADI; ZBAKH; TADONKI, 2017).

Uma nova abordagem para o processamento deste tipo de algoritmo é proposta pelo Apache Spark. Este *framework* se baseia em duas principais abstrações: RDDs (*Resilient Distributed Dataset*), uma estrutura de dados que representa uma coleção de dados distribuída e imutável; e *parallel operations*, um conjunto de operações paralelas que podem ser aplicadas a um *dataset*, como por exemplo, *map*, *flatmap*, *reduce*, *filter* e *foreach*. Adicionalmente, operações de *caching* podem ser usadas para manter o estado da computação atual em memória, permitindo o compartilhamento de dados intermediários entre diferentes nodos (ZAHARIA et al., 2010).

O Apache Spark foi inicialmente desenvolvido na University of California, Berkeley em 2009, provendo *APIs* de baixo nível, que possibilitaram o processamento distribuído de algoritmos iterativos em memória. Conforme o crescimento e adoção do Apache Spark, novos componentes e bibliotecas foram sendo adicionados à plataforma, como *APIs* estruturadas que facilitam o desenvolvimento (*SQL, Dataframes e Datasets*); suporte para *stream processing*; componentes para *machine learning* (MLib) e grafos (GraphX) (CHAMBERS; ZAHARIA, 2018).

Desta forma, neste trabalho se propõe o estudo e a investigação da plataforma Apache Spark, assim como o desenvolvimento e a avaliação de desempenho – no contexto de *frameworks* que auxiliam neste desenvolvimento – de três algoritmos distribuídos para

mineração de *itemsets* frequentes: YAFIM, R-Apriori e DFPS, baseados no Apriori e FP-Growth. Os resultados dos algoritmos foram analisados e comparados, variando o volume de dados e outros parâmetros como o número de nodos do *cluster*.

O restante deste trabalho está organizado da seguinte forma. O capítulo 2 traz uma breve introdução à computação distribuída, seus desafios e atuais soluções. O capítulo 3 se aprofunda nas características do Apache Spark, trazendo suas principais abstrações, RDDs, Datasets e DataFrames, assim como exemplos de manipulação de dados com *transformations* e *actions*. No capítulo 4 são descritos os estudos relacionados que fazem uso do Spark na implementação de algoritmos de associação. O capítulo 5 detalha as implicações no desenvolvimento próprio dos algoritmos distribuídos implementados usando Spark. E por fim, o capítulo 6 explora os resultados dos experimentos realizados.

2 COMPUTAÇÃO DISTRIBUÍDA

Por mais de quatro décadas, a lei de Moore corretamente previu que o número de transistores em um único processador dobraria a cada 18 meses. Isso permitiu que durante este período, usuários e desenvolvedores pudessem simplesmente esperar pela nova geração de processadores para obter os ganhos de desempenho esperados nos seus sistemas. No entanto, desde 2013, essa taxa de crescimento vem desacelerando constantemente, uma vez que o aumento da performance de um único núcleo fica cada vez mais complexo devido a problemas de aquecimento e consumo de energia (CHRAIBI, 2015).

Dessa forma, fabricantes de processadores estão adotando novas arquiteturas que fazem uso de múltiplos núcleos. Ao invés de focar no avanço de processadores monolíticos, os ganhos estão sendo direcionados ao paralelismo, onde vários núcleos ou processadores são integrados em um único circuito. Como por exemplo, o microprocessador tradicional Intel Core i7, que possui quatro núcleos distintos que podem executar instruções de forma paralela (CHRAIBI, 2015; PACHECO, 2011).

Essa mudança de paradigma tem forte impacto no desenvolvimento de *software*, uma vez que o aumento no número de processadores não provê mudanças imediatas de performance na vasta maioria dos programas. Isso se dá devido à maneira em que os programas são normalmente escritos, onde em grande parte, só se considera uma única linha de execução, que por consequência, acaba por não fazer uso dos processadores ou núcleos adicionais disponíveis em um computador (PACHECO, 2011).

Para que um programa se beneficie dos ganhos de performance providos por uma arquitetura de múltiplos núcleos, se faz necessário que o mesmo seja desenvolvido a partir de modelos de programação paralela. Dessa forma, assim como ilustrado na Figura 2, é possível que a execução de uma tarefa seja quebrada em várias partes, as quais podem ser processadas ao mesmo tempo por diferentes núcleos, trazendo uma otimização no tempo de execução de um programa (DIAZ; MUÑOZ-CARO; NIÑO, 2012).

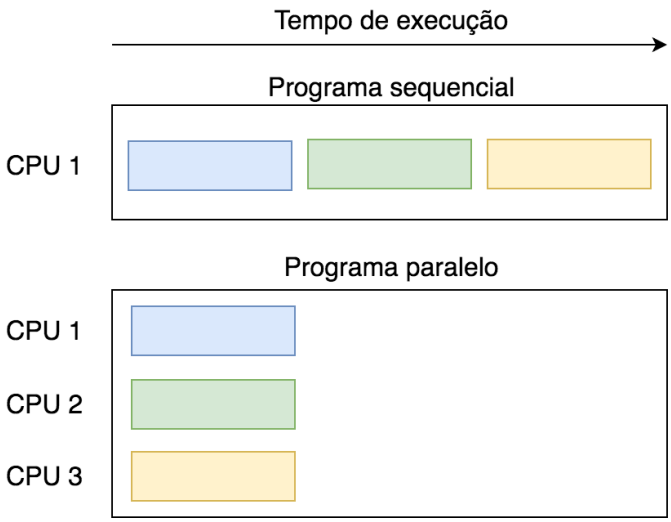


Figura 2 - Paralelização na execução de tarefas de um programa.
Fonte: elaborado pelo autor

Segundo Diaz, Muñoz e Niño (2012), a computação paralela pode ser classificada em dois principais modelos de programação: de memória compartilhada e de memória distribuída. No primeiro modelo, todos os núcleos do processador possuem acesso à mesma área de memória, podendo se comunicar diretamente através desta. Por outro lado, no modelo de memória distribuída, cada processador possui sua própria memória que é isolada dos outros processadores, a comunicação neste modelo ocorre normalmente através da rede em *clusters* de computadores.

2.1 Paralelização por memória compartilhada

A paralelização de programas pelo modelo de memória compartilhada pode ser empregada a partir do uso de *threads*, linhas de execução autônomas que compartilham o código do processo e suas variáveis. Dessa forma, cada *thread* pode ficar encarregada por processar um pedaço do programa ou algoritmo, onde a comunicação entre as diferentes linhas de execução (*threads*) se dá através do acesso das variáveis pela memória *heap* (PACHECO, 2011).

No entanto, esse modelo de comunicação entre *threads* requer uma análise cuidadosa pelo desenvolvedor, uma vez que o compartilhamento de estado pode vir a ocasionar diversos problemas no programa, como *race conditions* e *dead locks*[1]. Assim, as chamadas seções críticas, áreas do código onde o acesso de duas ou mais *threads* de forma paralela pode

[1] https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks

ocasionar inconsistência nos resultados das instruções, devem ser protegidas a partir do uso de mecanismos especiais como semáforos ou blocos de sincronização (DIAZ; MUÑOZ-CARO; NIÑO, 2012).

Uma alternativa à manipulação manual de *threads* pelo desenvolvedor é o uso de bibliotecas que abstraem parte dos detalhes de implementação necessários para o gerenciamento das *threads*. OpenMP (*Open Multi-Processing*) está entre uma das mais populares bibliotecas que suportam o modelo de programação paralela por memória compartilhada, provendo uma abstração de mais alto nível para *multithreading* em C, C++ e Fortran (TORQUATI et al., 2014).

A biblioteca OpenMP é composta por um conjunto de funções e diretivas para o compilador. Nessa biblioteca, a paralelização pode ocorrer de forma incremental, permitindo que diretivas sejam adicionadas ao código gradualmente. As áreas a serem paralelizadas não são identificadas automaticamente pela biblioteca, essas regiões precisam ser explicitamente especificadas pelo desenvolvedor. Seguindo o padrão *fork/join*, OpenMP trabalha de forma a quebrar as regiões identificadas em diferentes tarefas (*fork*), que são então executadas em paralelo por diferentes *threads*, uma vez que todas essas completam suas execuções, os resultados são combinados e a execução sequencial é resumida (*join*) (TORQUATI et al., 2014).

```
1  #include <omp.h>
2
3  ...
4
5  int array[5000];
6
7  #pragma omp parallel for
8  for (int i = 0; i < 5000; i++) {
9      array[i] = i * i;
10 }
```

Figura 3 - Uso de diretivas no OpenMP em C.
Fonte: adaptado de OpenMP Architecture Board (2016)

A Figura 3 acima demonstra o uso de uma das diretivas disponibilizadas pelo OpenMP. A diretiva *parallel for* é especificada na linha 7, dessa forma, a instrução *for* é quebrada em diferentes tarefas, onde cada *thread* alocada pelo OpenMP fica responsável por processar um segmento do *for*. Caso duas *threads* fossem alocadas neste caso, a primeira faria a execução de 0 até 2499 e a segunda de 2500 até 4999. Essa e outras cláusulas permitem que o programa seja paralelizado sem que *threads* precisem ser gerenciadas de forma explicita (OPENMP ARCHITECTURE BOARD, 2016).

2.2 Paralelização por memória distribuída

Ao contrário da paralelização por memória compartilhada, na paralelização por memória distribuída não existe uma comunicação direta entre os processadores pela memória primária. Neste modelo, cada processador possui seu próprio endereçamento de memória isolado, assim não sendo possível que diferentes computadores acessem as mesmas variáveis ou objetos de um programa. Por conta disto, a comunicação entre as diferentes linhas de execução neste modelo é comumente feita a partir da conexão pela rede (DIAZ; MUÑOZ-CARO; NIÑO, 2012).

A paralelização por memória distribuída pode ser implementada de diversas maneiras, como por exemplo:

- Programação em baixo nível usando *sockets* de rede;
- *Remote Procedure Call* (Java RMI);
- *Distributed Shared Memory* (DSM);
- *Message Passing* (MPI);

Na sua forma mais básica, a comunicação pode ser feita através do uso direto de *sockets* de rede, contudo, o desenvolvedor fica também responsável pelos detalhes de baixo nível da implementação. Dessa forma, surgiram outras abstrações como *Remote Method Invocation* (Java RMI), que permite a comunicação entre diferentes máquinas através da chamada de métodos que são expostos por interfaces. Além disto, outra possibilidade de comunicação é pelo uso de *Distributed Shared Memory* (DSM), uma arquitetura onde a memória dos diferentes processadores é virtualmente endereçada, permitindo que a comunicação seja feita como se todos processadores tivessem acesso à mesma área de memória. (DIAZ; MUÑOZ-CARO; NIÑO, 2012).

Por último, *Message Passing* está entre os modelos mais usados no desenvolvimento de aplicações paralelas de alta performance. Neste modelo, a comunicação acontece através da troca explicita de mensagens. Isso permite que programas paralelos enviem os resultados de suas computações sem compartilhar o estado de suas variáveis internas, evitando efeitos colaterais e problemas de concorrência. *Message Passing Interface* (MPI) é uma das bibliotecas que implementam o modelo de *Message Passing*. Disponível em C, C++ e Fortran, nelas são especificadas operações que possibilitam essa troca de mensagens, como por exemplo, *MPI_Send* e *MPI_Recv* (TORQUATI et al., 2014).

2.3 Modelos híbridos

A combinação dos dois modelos apresentados anteriormente é comumente empregada em programas onde o tempo de execução é propriedade crítica, possibilitando o completo uso do poder de processamento de *clusters* com nodos *multicore*. *Clusters* são conjuntos de computadores (nodos), onde cada nodo é autônomo e possui seu próprio processador e memória. Dessa forma, o modelo híbrido permite que tarefas sejam paralelizadas entre os diferentes nodos através de um modelo de memória distribuída (MPI, DSM e etc) e, além disso, cada nodo pode paralelizar sua execução interna pelo uso da memória primária que é compartilhada entre seus diferentes núcleos (*threads*, OpenMP e etc) (TORQUATI et al., 2014).

2.4 Desafios

Visto que a computação de tarefas pesadas em tempos aceitáveis não é viável através do uso de apenas um núcleo ou mesmo então de um só computador *multicore*, a computação distribuída surgiu como alternativa aos custosos supercomputadores, permitindo que o processamento dos altos volume de dados encontrados atualmente seja feito através do uso de vários computadores convencionais conectados pela rede. No entanto, essa abordagem altamente paralela que combina ambos modelos de programação acaba por ser relativamente complexa, gerando um número de pontos que precisam ser levados em conta para que uma implementação seja performática e confiável.

A execução deste tipo de programa é custosa e pode chegar a dias de processamento até sua completa conclusão. É de grande importância que caso uma falha ou erro venha a ocorrer durante essa execução – seja por conta de uma interrupção da comunicação na rede, defeitos físicos de hardware ou problemas em tarefas específicas dentro de um nodo – que o processamento da aplicação até aquele momento não seja perdido por inteiro. Logo, é mandatório que esse tipo de aplicação seja tolerante a falhas e capaz de se recuperar automaticamente caso algum problema venha a acontecer, reexecutando somente as tarefas afetadas (KATAL; WAZID; GOUDAR, 2013).

Para que a execução deste tipo de aplicação seja feita em tempos aceitáveis, *clusters* de centenas ou até milhares de computadores são empregados para realizar o processamento de forma paralela. Com isso em mente, se faz necessário que as aplicações sejam flexíveis o bastante para escalarem com o tamanho do *cluster*, permitindo que computadores sejam adicionados ou removidos dependendo da necessidade da aplicação. Nestes cenários, o compartilhamento e balanceamento dos recursos disponíveis de um *cluster* deve ser realizado

de forma ótima, buscando a maior eficiência durante a execução de cada tarefa. (KATAL; WAZID; GOUDAR, 2013).

Além disto, os dados a serem processados precisam estar armazenados em algum lugar. E quando o assunto é Big Data, muito provavelmente não será possível que estes sejam armazenados em um único computador. Uma das soluções é a realização deste armazenamento de forma distribuída, onde partições dos dados são divididas em um número de diferentes computadores. Portanto, essa abordagem adiciona um nível de complexidade adicional, onde é necessário que seja feito um controle que permita a escrita e leitura paralela por múltiplos computadores, assim como o gerenciamento das partições e a tolerância a falhas caso um nodo venha a perder seus dados (KATAL; WAZID; GOUDAR, 2013).

2.5 Frameworks

Como visto na seção anterior, aplicações distribuídas com o objetivo de processar um grande volume de dados possuem um alto nível de complexidade. O desenvolvimento deste tipo de aplicação seria extremamente árduo e demorado caso fosse feito apenas com o auxilio de uma linguagem de programação, pois todos os detalhes relacionados ao *cluster* – desde a comunicação entre os diferentes nodos, a orquestração das tarefas sendo executadas, até a tolerância a falhas – teriam que ser levados em conta e construídos do zero. Por esse motivo, diversas bibliotecas e *frameworks* relacionados à Big Data vêm sendo desenvolvidos ao longo do tempo, fornecendo um ecossistema de tecnologias que facilitam e aceleram a criação de aplicações distribuídas.

2.5.1 Hadoop MapReduce

O Apache Hadoop MapReduce é um *framework open source* que fornece uma implementação do modelo de programação MapReduce proposto pela Google em 2004, que visa suportar a computação distribuída em grandes *datasets*. Esse paradigma permite que os dados sejam divididos em partições menores e distribuídos entre vários computadores que prosseguem com o processamento destes menores pedaços de forma paralela, acelerando o tempo total de execução. O Hadoop MapReduce abstrai os detalhes de gerenciamento do *cluster* do desenvolvedor, que fica livre para concentrar seus esforços em expressar a lógica da sua aplicação através das duas funções vistas anteriormente: *map* e *reduce* (PATEL; BIRLA; NAIR, 2012).

A arquitetura do Hadoop MapReduce clássico consiste de um servidor mestre chamado *jobtracker* e de um ou mais servidores escravos, conhecidos como *tasktrackers.* O *jobtracker* é responsável por orquestrar a aplicação como um todo, dividindo e agendando as tarefas para serem executadas nos *tasktrackers*. Estes, por sua vez realizam o processamento das tarefas e comunicam o progresso ao longo do tempo para o *jobtracker*. Caso uma dessas tarefas falhe, o *jobtracker* fica encarregado por reagendar a execução com outro *tasktracker*, não deixando que a computação por inteira seja perdida (WHITE, 2015).

2.5.2 HDFS

Hadoop Distributed File System ou HDFS é um sistema de arquivos distribuído que tem o objetivo de armazenar arquivos com tamanhos na casa dos gigabytes, terabytes ou até petabytes. Surgiu como uma implementação *open source* da especificação GFS (*Google File System*) publicada pelo Google em 2003, que descreve como a arquitetura do sistema de arquivos distribuídos funcionava para escalar as bilhões de páginas indexadas em produção no Google. Na sua forma mais básica, os arquivos são divididos em vários pedaços e armazenados em diferentes computadores, permitindo que modelos de programação distribuída como o Hadoop Map Reduce façam leituras e escritas dos altos volumes de dados processando-os de forma eficiente, além de prover tolerância a falhas caso um dos nodos que contém parte dos dados venha a falhar (WHITE, 2015).

Um *cluster* HDFS segue o padrão de arquitetura mestre/escravo, contendo dois tipos de nodos, o *namenode*, que representa o mestre, e um ou vários *datanodes*, que representam os escravos. O *namenode* é responsável por controlar os metadados do sistema de arquivos, mantendo uma lista de todos os diretórios e blocos de arquivos, assim como onde estes são encontrados no *cluster*, podendo reconstruir um arquivo quando requisitado a partir dos vários blocos existentes. Por outro lado, os *datanodes* são responsáveis pelo real armazenamento dos dados, que são periodicamente sinalizados de volta para o *namenode* mantendo os metadados atualizados (WHITE, 2015).

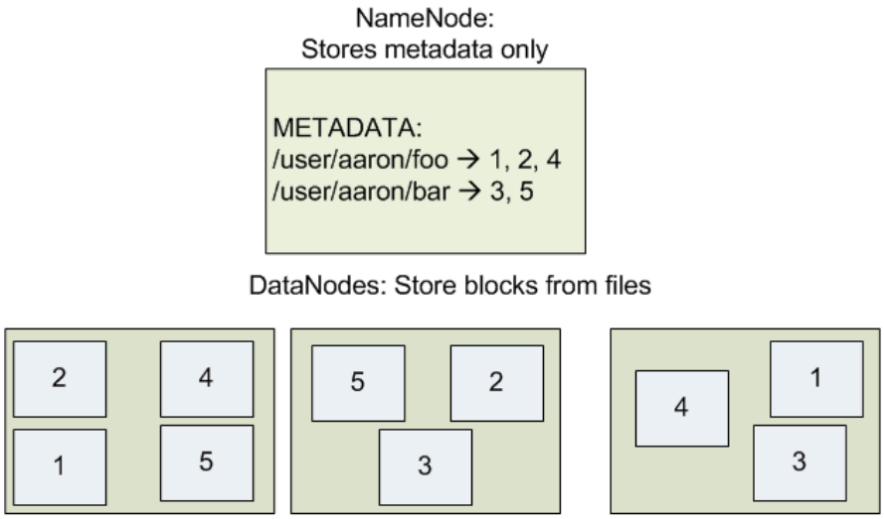


Figura 4 - Topologia de um *cluster* HDFS.
Fonte: Lai (2014)

A Figura 4 demonstra a topologia de um cluster HDFS, o *namenode* contendo a informação dos arquivos armazenados e em quais blocos estes estão repartidos, e os *datanodes* armazenando os blocos dos dados propriamente ditos. Além disso, percebe-se que cada bloco de dados é contido em mais de um *datanode*, essa propriedade é o fator de replicação, que armazena os dados de forma redundante, provendo tolerância a falhas. Essa funcionalidade é de grande importância em *clusters* HDFS, uma vez que este foi construído para ser utilizado em *clusters* de computadores convencionais (LAI, 2014).

2.5.3 YARN

Hadoop YARN (*Yet Another Resource Negotiator*) é um gerenciador de recursos de *clusters*. Sua principal função é fazer com que as unidades de trabalho (*jobs*) sejam agendadas e executadas no *cluster* da maneira mais eficiente possível. Foi introduzido inicialmente em 2012 para aprimorar o agendamento de *jobs* em relação à arquitetura clássica do Hadoop MapReduce (*jobtrackers* e *tasktrackers*), mas é uma tecnologia desacoplada do MapReduce, podendo ser usada em outros modelos de programação distribuída, assim como ilustrado na Figura 5 (WHITE, 2015).

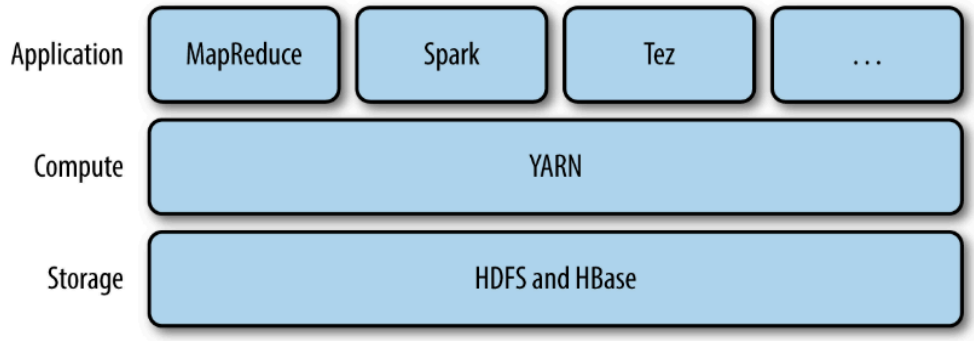


Figura 5 - Camada de atuação do YARN.
Fonte: White (2015)

Existem dois tipos de nodos em um *cluster* controlado pelo YARN: O *resource manager* que gerencia os recursos disponíveis, e os *node managers* que são responsáveis pela criação e monitoração de *containers*. Quando um *job* é agendado pelo YARN, o *resource manager* busca os nodos que possuem os recursos disponíveis suficientes para serem alocados, os *node managers* seguem com a criação dos containers, que por sua vez realizam a execução das tarefas utilizando as configurações de CPU e memória liberados pelo *resource manager* (WHITE, 2015).

Esse nível de abstração permite que *jobs* sejam executados de maneira mais eficiente. Ao invés de um nodo ser alocado por completo para a execução de uma tarefa, podendo desperdiçar recursos não totalmente utilizados, é possível que somente uma parte dos recursos de um nodo seja alocado à uma tarefa. Dessa forma, o restante fica disponível para ser utilizado em outras tarefas (WHITE, 2015).

Além disto, é possível que *jobs* sejam agendados usando três estratégias diferentes: FIFO, *Capacity scheduler* e *Fair Scheduler*.

- FIFO (*First In First Out*) é a mais simples e não requer nenhum tipo de configuração. Cada *job* é executado por ordem de agendamento e é geralmente usada em *clusters* menores. Contudo, *jobs* mais demorados podem acabar por trancar a fila por um longo período de tempo.
- Na estratégia *Capacity Scheduler* existe uma fila adicional exclusiva para *jobs* menores, que podem ser executados enquanto os *jobs* principais são processados na primeira fila. Uma parte dos recursos do *cluster* são reservados para a segunda fila, o que acaba tornando os *jobs* principais um pouco mais lentos.
- Por último, no *Fair Scheduler* os recursos do *cluster* são distribuídos dinamicamente entre todos os *jobs* agendados. Caso somente um *job* principal esteja sendo executado,

todos os recursos serão alocados para ele. Assim que um segundo *job* é agendado, metade dos recursos são realocados para o novo *job*. Quando este é finalizado, seus recursos são liberados e realocados para o *job* principal novamente. Isso permite que o *cluster* sempre esteja com uma alta utilização dos recursos, contudo, existe a desvantagem de um pequeno atraso na alocação dinâmica dos recursos entre os *jobs* (WHITE, 2015).

2.5.4 Spark

O Apache Spark é uma plataforma unificada *open source* para processamento paralelo em *clusters* de computadores. Foi desenvolvida em Scala, linguagem funcional da JVM, mas também possui bibliotecas que proveem interfaces para as linguagens mais populares como Python, Java, R e SQL. Teve seu desenvolvimento iniciado em 2009 com a publicação do artigo *Spark: Cluster Computing with Working Sets* (ZAHARIA et al., 2010) pela Universidade de Berkeley na California. Hoje, Spark é o projeto *open source* mais ativo para processamento distribuído, possuindo cerca de 2.000 contribuidores, incluindo Google, Facebook, Uber e IBM (APACHE SOFTWARE FOUNDATION, 2018b; HANDY, 2017).

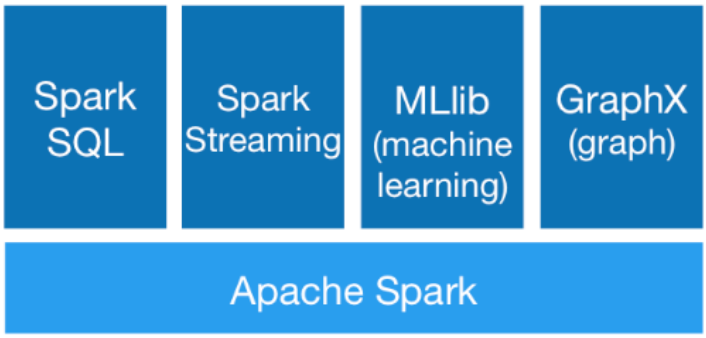


Figura 6 - Componentes do Apache Spark.
Fonte: Apache Software Foundation (2018)

Como se pode observar na Figura 6, o Spark é atualmente composto por um número de componentes que facilitam o desenvolvimento de aplicações específicas em um ambiente distribuído, como *streaming* para processamento em tempo real, MLlib para *machine learning* e GraphX para algoritmos de grafos. Estes componentes são desenvolvidos a partir da biblioteca *core* do Spark (RDDs, Datasets e DataFrames), que proveem APIs que facilitam a computação paralela em ambientes clusterizados. Essas abstrações também podem ser usadas para desenvolver aplicações genéricas que não se encaixam no nicho de nenhum dos componentes disponibilizados (CHAMBERS; ZAHARIA, 2018).

Assim como o Hadoop MapReduce, Spark é um modelo de programação distribuída, podendo utilizar outras tecnologias do ecossistema de Big Data ao seu lado, como HDFS e Amazon S3 para o armazenamento dos dados, ou então YARN e Kubernetes para o gerenciamento do *cluster* (APACHE SOFTWARE FOUNDATION, 2018b).

2.6 Considerações finais do capítulo

A viabilização do processamento de um alto volume de dados pode-se dar através da combinação dos modelos de paralelização por memória compartilhada e distribuída, os quais permitem o completo uso de todos os núcleos de um processador, assim como a distribuição de tarefas pela rede entre máquinas de um *cluster*. Adicionalmente, o conjunto de ferramentas especializadas existentes como MapReduce, HDFS, YARN e Spark, fazem parte de um ecossistema de Big Data que facilita a construção e gerenciamento de programas distribuídos.

3 APACHE SPARK

Frameworks para computação distribuída como MapReduce permitem que desenvolvedores escrevam programas paralelos sem que se tenha uma preocupação direta com detalhes de distribuição e tolerância a falhas. Em grande parte, a tolerância a falhas nestes *frameworks* é alcançada a partir da escrita dos resultados intermediários das tarefas e *jobs* em um sistema de arquivos externo. Caso uma das tarefas ou nodos venham a falhar durante execução, a computação até aquele momento pode ser reconstruída a partir do disco. No entanto, essa propriedade acaba tendo por consequência um impacto significativo na performance geral do programa devido ao grande número de escritas no disco e à replicação constante dos dados (ZAHARIA et al., 2012).

Uma classe de aplicações distribuídas específica acaba por não obter resultados de performance eficientes quando programadas neste modelo: as aplicações que fazem reuso de resultados intermediários entre múltiplas computações. Essa propriedade é comumente vista em algoritmos iterativos de *machine learning* e grafos, como PageRank e K-means. Além destes, o reuso de dados é também importante para a mineração de dados interativa, onde diferentes *ad-hoc queries* (buscas de teste) são feitas em um mesmo *subset* de dados múltiplas vezes (ZAHARIA et al., 2012).

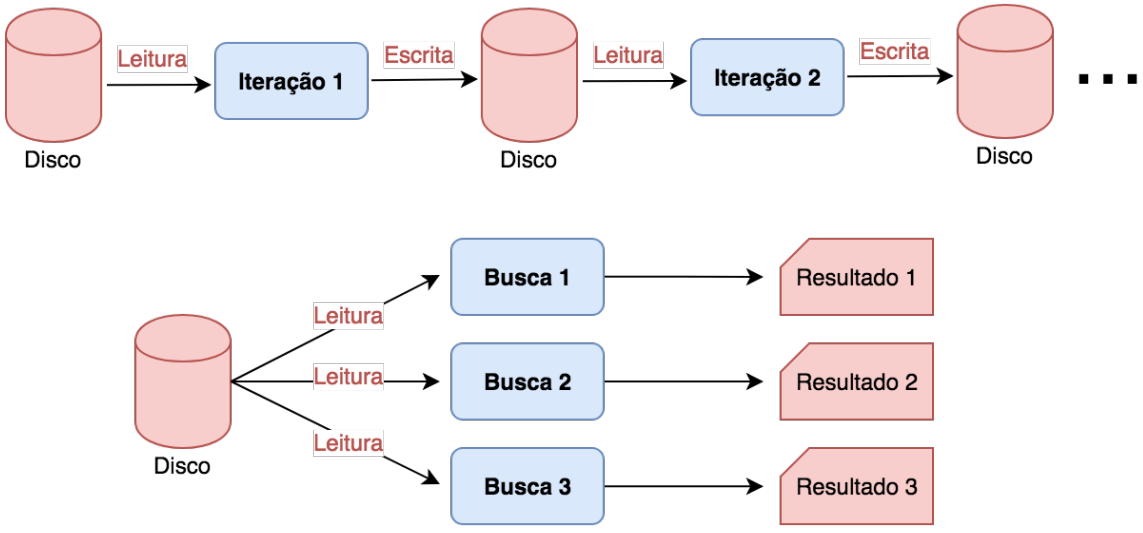


Figura 7 - Reuso de dados pelo disco.
Fonte: adaptado de Zaharia et al. (2012)

A Figura 7 ilustra os dois casos mencionados anteriormente: Algoritmos iterativos que são expressos a partir de *jobs* encadeados, compartilhando dados através de um disco rígido externo; assim como a mineração de dados interativa, que faz buscas ou processamentos nos

dados usando diferentes parâmetros múltiplas vezes, com o objetivo de encontrar padrões ou resultados de interesse, entretanto, o *input* deve ser lido do disco e processado do princípio repetidas vezes.

Segundo Zaharia (2012), um dos criadores do Spark, o *framework* Spark surgiu a partir desta necessidade, tendo como objetivo prover um modelo de programação eficiente para algoritmos de natureza iterativa e de mineração de dados. É proposto o compartilhamento de dados entre *jobs* ou tarefas pela memória primária ao invés do uso do disco, que consequentemente reduz de forma drástica o tempo de I/O. A Figura 8 ilustra o reuso de dados proposto pelo Spark nestes dois tipos de processamentos.

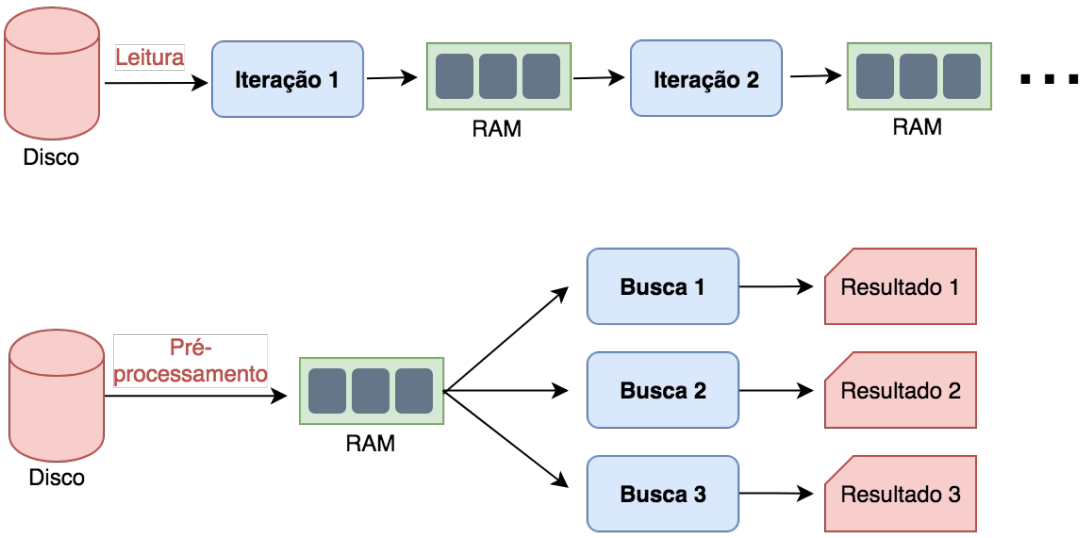


Figura 8 - Reuso de dados pela memória primária.
Fonte: adaptado de Zaharia et al. (2012)

Contudo, uma vez que o compartilhamento de dados e escrita de resultados intermediários pelo disco é removido, uma propriedade de grande importância para a computação distribuída precisa ser repensada: a tolerância a falhas. *Frameworks* especializados que já utilizam o reuso de dados pela memória primária, como Piccolo, oferecem tolerância a falhas a partir de sincronizações de granularidade fina baseado na mutação de estado, essa comunicação acontece a partir da replicação de dados ou da atualização de logs entre os nodos pela rede. Entretanto, essa abordagem acaba por gerar um *overhead* considerável no tempo de computação (ZAHARIA et al., 2012).

Spark propõe uma abordagem diferente que possibilita o reuso de dados através da memória primária, e que também possui um mecanismo eficiente para tolerância a falhas, não induzindo uma constante comunicação entre as máquinas ou replicação em um disco externo.

O restante deste capítulo apresenta o funcionamento e arquitetura base do Spark, assim como a abstração *core* que torna isso possível, *Resilient Distributed Datasets* (RDDs), demonstrando seu funcionamento e peculiaridades. Além disso, também são apresentadas as abstrações de alto nível desenvolvidas a partir de RDDs, os Datasets e DataFrames, que visam trazer funcionalidades que facilitam o desenvolvimento de aplicações distribuídas.

3.1 Arquitetura base

O modelo de execução do Spark é similar ao de outros *frameworks* para processamento distribuído. Dado um *dataset*, os dados são quebrados em pedaços menores e processados em paralelo por múltiplas *threads* e nodos. As operações nestas partições são similares à classe de arquitetura SIMD (*single instruction, multiple data*) vista na taxonomia de Flynn's, onde a mesma operação é executada em diferentes dados ao mesmo tempo, atingindo um alto nível de paralelismo, mas não de concorrência, uma vez que cada processamento é isolado e não permite o compartilhamento de estado (JINGJING; SOMASHEKAR, 2017).

Esse modelo é empregado no Spark usando uma implementação da arquitetura mestre/escravo. O mestre, ou *driver* no Spark, é responsável por coordenar a execução de uma aplicação no *cluster*, mantendo o estado das tarefas em andamento e agendando novas tarefas que serão processadas pelos escravos, os executores no Spark. Além disso, como pode ser observado na Figura 9, o *driver* se comunica diretamente com um gerenciador de *cluster*, desacoplando a alocação de recursos (CPU, memória e etc) destinados à execução de tarefas para um *framework* especializado, como Hadoop YARN, Apache Mesos ou até então Kubernetes (APACHE SOFTWARE FOUNDATION, 2018b).

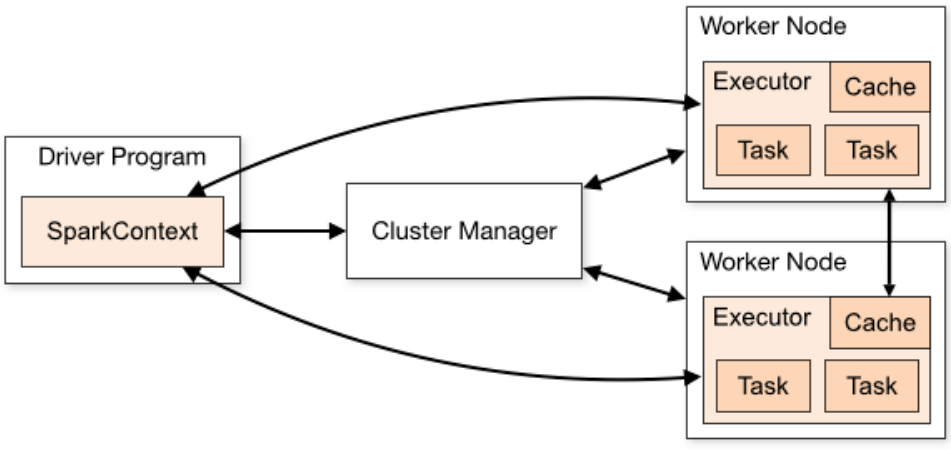


Figura 9 - Modelo de execução do Spark.
Fonte: Apache Software Foundation (2018)

3.2 *Resilient Distributed Datasets* (RDDs)

RDDs são definidos formalmente como representações imutáveis de coleções de dados particionados e distribuídos em um *cluster*. RDDs possuem três características chaves que definem seu comportamento: uma lista de partições, uma lista de dependências em outros RDDs e uma função para processar cada partição. Os dados contidos nas partições não são armazenados diretamente nos objetos RDD, ao invés disto, guardam informações de como o *dataset* deve ser computado. Cada operação aplicada em RDDs implica na construção de um DAG (*Directed Acyclic Graph*), conforme exemplo ilustrado na Figura 10. Assim, caso um *nodo* falhe, os dados perdidos até aquele ponto podem ser reconstruídos a partir da computação da lista de operações representadas pelo DAG nas partições que foram perdidas (ZAHARIA et al., 2010).

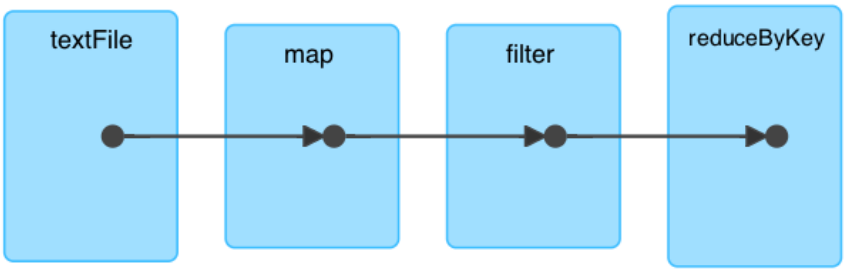


Figura 10 – Exemplo de operações em um DAG.
Fonte: elaborado pelo autor

Diversas operações paralelas podem ser realizadas em *datasets* representados por RDDs a partir de funções passadas pelo usuário. Essas operações paralelas podem ser divididas em dois tipos: *transformações* e *ações*. Transformações são funções disponibilizadas pela interface de um RDD que aplicam uma determinada lógica em todas as partições existentes, por exemplo: *map*, *filter* e *groupBy*. Quando transformações são aplicadas em RDDs, uma nova instância de RDD é gerada contendo essa nova operação, como RDDs são *lazy evaluated*, a execução da operação no *dataset* não ocorre imediatamente, mas só no momento em que uma *ação* é aplicada. Assim, quando ações são aplicadas em RDDs, como *count*, *reduce* e *collect*, as transformações encadeadas até o momento são executadas em paralelo de acordo com as partições disponíveis (ZAHARIA et al., 2010).

Além disto, RDDs permitem que o usuário controle aspectos avançados relacionados à persistência e ao particionamento do *dataset*. O método *persist* pode ser invocado em um RDD para indicar que uma vez processado, seu resultado seja mantido persistido na memória RAM, caso não exista espaço suficiente, o disco também é utilizado. Esse RDD pode ser então reusado em outras computações (inclusive de forma iterativa), não precisando que todas as

operações definidas no DAG sejam executadas uma segunda vez. Adicionalmente, operações de particionamento podem ser definidas pelo usuário, dando controle referente à alocação de registros em partições especificas, como por exemplo: todas as linhas começando com "A" devem ser armazenadas em uma única partição, permitindo que otimizações sejam feitas em transformações que venham a precisar de dados de múltiplas partições, como a função *groupBy* (ZAHARIA et al., 2010).

3.2.1 Criação de RDDs

Aplicações em Spark são criadas a partir de um objeto *SparkContext*, ponto de entrada controlado pelo nodo *driver* que dá acesso à RDDs e que também é responsável pela comunicação no *cluster*. Esse objeto pode ser obtido através da inicialização pelo construtor, onde dois principais parâmetros são aceitos: *master*, a URL do *cluster* (ex: mesos://host:port, spark://host:port ou local[4]); e *appName,* o nome da aplicação. A Tabela 1 demonstra o código em Scala usado para instanciar um contexto Spark local usando 4 núcleos chamado sparkJob em uma variável *spark*.

Tabela 1 - Obtenção do ponto de entrada do Spark.

```
val spark = new SparkContext(master = "local[4]", appName = "sparkJob")
```

Fonte: elaborado pelo autor

Com o contexto obtido, RDDs podem ser criados de duas principais maneiras, paralelizando uma coleção de dados previamente carregada em *runtime*, ou a partir de um arquivo de texto armazenado localmente ou na rede. Na Tabela 2, a função *parallelize* é usada para criar um RDD de *Strings* com duas partições contendo os dados da array palavras. O mesmo resultado também pode ser obtido lendo um arquivo com a função *textFile*, que cria um RDD representando um registro para cada linha.

Tabela 2 - Exemplos de criação de RDDs.

```
val palavras: Array[String] = "Criando RDD em Spark".split(" ")
val rdd: RDD[String] = spark.parallelize(seq = palavras, numSlices = 2)
val rddFromHDFS: RDD[String] = spark.textFile("hdfs://dir/palavras.txt")
```

Fonte: elaborado pelo autor

3.2.2 Transformações

RDDs podem ser manipulados através de transformações. Quando essas transformações são aplicadas em RDDs, nenhuma computação ocorre imediatamente no *dataset*, mas um novo RDD é criado contendo a função descrita pelo usuário, além de uma

referência ao RDD anterior, que por sua vez possui sua própria função a ser executada e uma referência a outros RDDs. Essa corrente de RDDs representa um grafo acíclico direcionado (DAG) de operações a serem aplicadas sobre o *dataset*. Essas operações são executadas somente quando uma *action* é aplicada ao RDD. Cada *action* representa um *job* em Spark, que pode conter uma ou diversas tarefas que serão processadas pelos executores (CHAMBERS; ZAHARIA, 2018).

Transformações podem ser classificadas em dois tipos: *narrow dependencies* e *wide dependencies*. Como visto anteriormente, RDDs representam dados particionados que podem residir em diferentes máquinas, com isso em mente, transformações *narrow* tem escopo isolado para cada partição, não dependendo de dados de outras partições. Isso faz com que nenhum dado tenha que ser enviado pela rede entre as máquinas, tornando-as eficientes. Por outro lado, transformações *wide* requerem dados de duas ou mais partições, estes dados são enviados pela rede para serem processados em conjunto por uma máquina. Essa movimentação de registros pela rede é conhecida como *shuffle*, e pode ter impacto significativo em uma aplicação processando um alto volume de dados. (ZAHARIA et al., 2012).

A Figura 11 ilustra exemplos de transformações *narrow*: *map*, *filter*, *union* e *join* (com registros previamente agrupados por partição). Assim como transformações *wide*: *groupByKey* e *join* (quando os registros não são previamente agrupados nas suas respectivas partições). Cada retângulo preenchido em azul representa uma partição e o conjunto de partições representa um RDD, a aplicação de uma transformação resulta em um novo RDD.

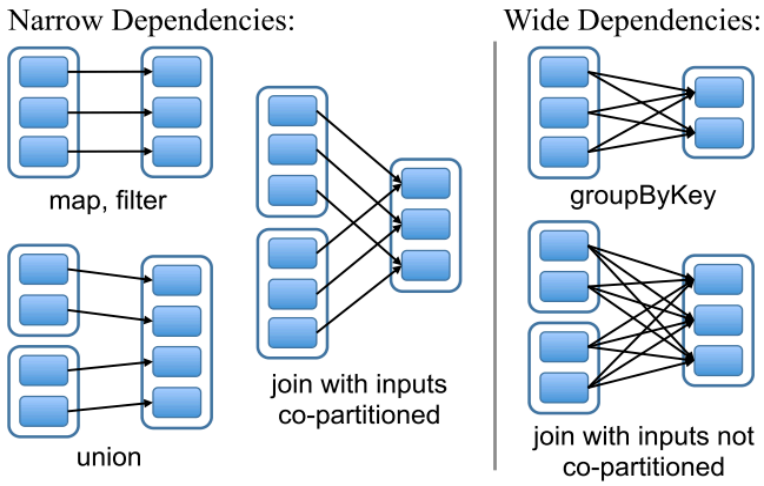


Figura 11 - Transformações *narrow* e *wide.*
Fonte: Zaharia et al. (2012)

3.2.3 *Narrow transformations*

RDDs disponibilizam um número de transformações *narrow* que permitem que operações especificadas pelo usuário através de *closures* (funções literais) sejam aplicadas em todo o *dataset*. Algumas das transformações mais utilizadas são as seguintes:

- *map(f: T => U) : RDD[T] => RDD[U]*: Map é uma operação que itera sobre todos os registros do RDD, aplicando uma função *f* passada pelo usuário que transforma um registro em outro valor. Como se pode perceber na assinatura desta operação, RDDs são tipos genéricos que operam sobre uma classe específica, como RDD[String] ou RDD[Int]. Assim, dado um RDD[T], a transformação map pode alterar o tipo genérico operado para RDD[U] através de uma função *f: T => U*.

  Como pode ser visto no código em Scala na Tabela 3, dado um RDD *nums* de inteiros com a sequência {1, 2, 3}, *nums2* representa um novo RDD que mapeia os valores multiplicando-os por 2, resultando em {2, 4, 6}. A função *num => num * 2* também pode ser expressa usando uma sintaxe do Scala chamada *placeholder syntax*, onde o sublinhado representa o parâmetro aceito pela função map. Essa sintaxe pode ser observada na expressão map de *nums3*, que é equivalente à expressão de *nums2*.

Tabela 3 – Exemplo de função *map*.

```
val nums  : RDD[Int] = spark.parallelize(Seq(1, 2, 3))
val nums2 : RDD[Int] = nums.map(num => num * 2)
val nums3 : RDD[Int] = nums.map(_ * 2)
```

Fonte: elaborado pelo autor

- *flatMap(f: T => Seq[U]) : RDD[T] => RDD[U]*: FlatMap é uma operação similar ao Map, no entanto, ao invés de transformar um registro em um único valor, um registro pode gerar 0 a N valores. A Tabela 4 demonstra um RDD operando em registros que representam linhas de um texto sendo transformado em uma coleção de palavras.

Tabela 4 - Exemplo de função *flatMap*.

```
val linhas:   RDD[String] = spark.textFile("hdfs://dir/texto.txt")
val palavras: RDD[String] = linhas.flatMap(linha => linha.split(" "))
```

Fonte: elaborado pelo autor

- *filter(f: T => Boolean) : RDD[T] => RDD[T]*: Filter é uma operação que itera sobre todos os registros e só mantem aqueles no qual o resultado de um predicado *f: T => Boolean* fornecido pelo usuário seja verdadeiro. Muito utilizado para ignorar registros que não são de interesse para a computação sendo realizada. Na Tabela 5, ocorre a filtragem de

todos os erros de um arquivo de logs. Percebe-se também que os tipos das variáveis podem ser omitidos em Scala devido à inferência de tipos.

Tabela 5 - Exemplo de função *filter*.

```
val logs   = spark.textFile("hdfs://dir/logs.txt")
val errors = logs.filter(_.contains("[ERROR]"))
```

Fonte: elaborado pelo autor

3.2.4 *Wide transformations*

Transformações *wide* são aquelas que dependem de duas ou mais partições, assim ocasionando em um *shuffle* caso o *dataset* não tenha sido pré-particionado de forma otimizada para essas operações. Algumas transformações com dependência *wide* como *groupByKey* e *reduceByKey* são especificas para RDDs de chave e valor, conhecidos como PairRDDs (CHAMBERS; ZAHARIA, 2018).

PairRDDs representam não só registros, mas também associam uma chave para cada valor, permitindo que operações sejam aplicadas em grupos de valores. PairRDDs podem ser criados usando a transformação *map*, e retornando uma tupla que contém uma chave e um valor. A Tabela 6 demonstra um RDD de palavras sendo mapeado para um PairRDD que contém o primeiro caractere como chave e sua respectiva palavra como valor.

Tabela 6 – Transformação de RDD para PairRDD.

```
// palavras: ["a1", "b1", "a2", "a3", "b2"]
val pair: RDD[(Char, String)] = palavras.map(linha => (linha.charAt(0), linha))
// pair: [('a', "a1"), ('b', "b1"), ('a', "a2"), ('a', "a3"), ('b', "b2")]
```

Fonte: elaborado pelo autor

- *groupByKey() : RDD[(K, V)] => RDD[(K, Seq[V])]*: GroupByKey é uma transformação *wide* que agrupa todas os valores de um PairRDD em suas respectivas chaves. Essa operação pode ser relativamente custosa dependendo do volume de dados sendo processado, uma vez que uma chave pode conter valores distribuídos em todas as partições, ocasionando na cópia destes pela rede para as máquinas que irão armazenar a chave específica.
  Na Tabela 7, o PairRDD *pair* visto anteriormente é agrupado por valores:

Tabela 7 - Exemplo de função groupByKey.

```
val pair: RDD[(Char, String)] = linhas.map(linha => (linha.charAt(0), linha))
val grouped: RDD[(Char, Iterable[String])] = pair.groupByKey()
// grouped: [('a', ["a1", "a2", "a3"]),
             ('b', ["b1", "b2")]
```

Fonte: elaborado pelo autor

A Figura 12 ilustra a movimentação de registros armazenados em três partições na transformação groupByKey.

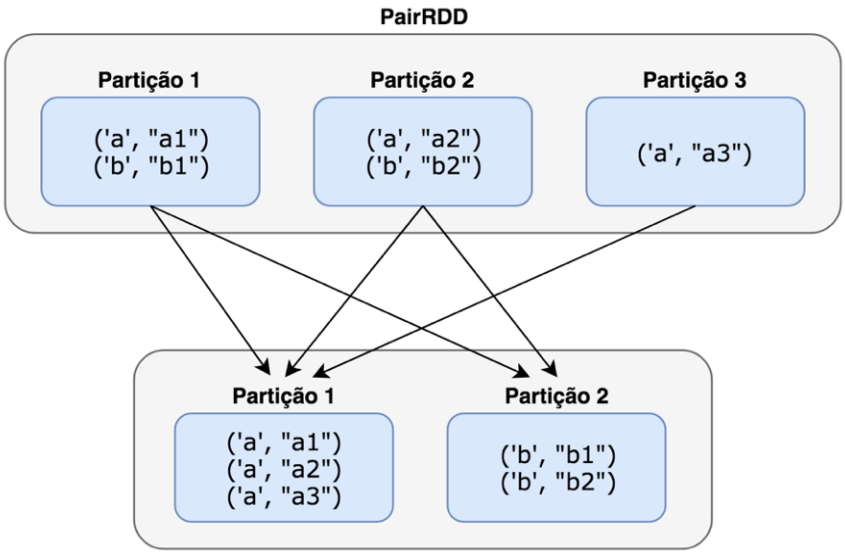


Figura 12 - Transformação *groupByKey* ocasionando em um *shuffle*.
Fonte: elaborado pelo autor

- *reduceByKey(f: (V, V) => V) : RDD[(K, V)] => RDD[(K, V)]*: ReduceByKey é uma operação similar à etapa reduce encontrada no framework Hadoop MapReduce. Dado um PairRDD, o usuário fornece uma função *f: (V, V) => V*, que representa a maneira de como dois valores de uma determinada chave devem ser combinados para gerar um terceiro valor. Essa função é chamada pelo Spark recursivamente para todos os valores de uma chave, até que somente um valor exista para cada chave. Os valores são "reduzidos" ou "combinados" por chave.

Assim como visto na transformação anterior *groupByKey*, *reduceByKey* também ocasiona em um *shuffle*, contudo, todos os valores de uma determinada chave em uma partição são pré-combinados localmente. Isso faz com que no máximo uma chave (que já tem seus valores combinados) seja enviada pela rede para outras máquinas caso necessário, não degradando performance tanto quanto a operação *groupByKey* quando o volume de dados é considerável.

Um exemplo clássico desta operação é a contagem de palavras em um texto. O trecho de código na Tabela 8 demonstra esse algoritmo, primeiramente, um RDD representando as linhas de um arquivo texto é criado; cada registro é então transformado em várias palavras usando *flatMap*; cada palavra é transformada em uma tupla, tendo a palavra como chave e

sua contagem como valor; cada chave tem então seus valores somados pela operação *reduceByKey*.

Tabela 8 - Contagem de palavras com *reduceByKey*.

```
spark.textFile("hdfs://dir/file.txt")
     .flatMap(linha => linha.split(" "))
     .map(palavra => (palavra, 1))
     .reduceByKey((v1, v2) => v1 + v2)
```

Fonte: elaborado pelo autor

Um detalhe peculiar neste código é o encadeamento de operações, prática comumente vista na programação funcional. Além disto, o fluxo deste código pode ser observado na Figura 13, onde cada bloco azul representa uma partição de dados e o conjunto de partições é representado pela instância de um RDD. Na transformação *reduceByKey*, nota-se que a combinação de valores por chave ocorre em duas etapas, localmente em cada partição, e através do *shuffle* para combinar os resultados obtidos em cada partição.

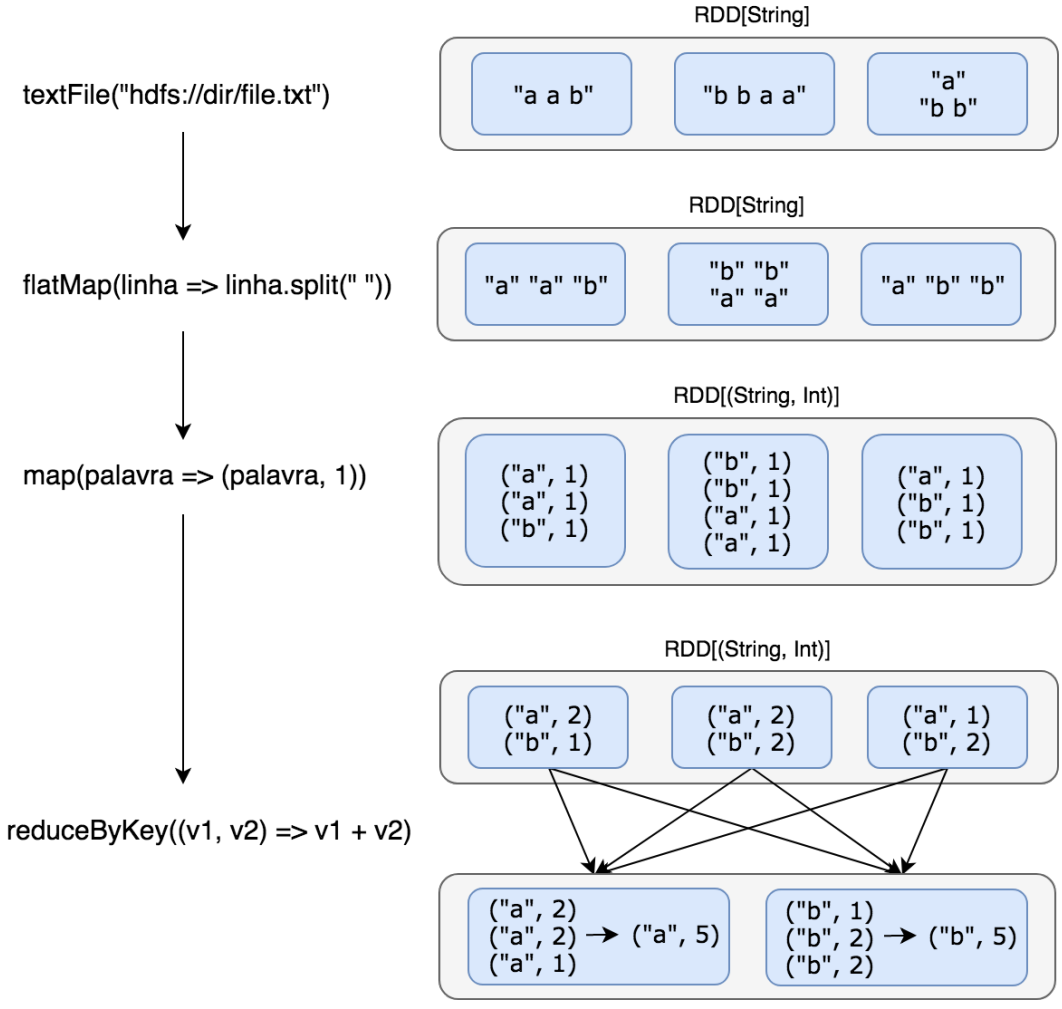


Figura 13 - Fluxo de operações em uma contagem de palavras usando *reduceByKey*.

Fonte: elaborado pelo autor

3.2.5 *Actions*

Enquanto transformações permitem a construção de um plano lógico de execução, as ações são operações que devidamente disparam a computação contida em todos os RDDs encadeados até um determinado momento. Cada ação ocasiona na criação de um *job* no Spark e implica na obtenção de um resultado. As seguintes ações estão entre as mais usadas (CHAMBERS; ZAHARIA, 2018):

- *count() : RDD[T] => Long:* retorna o número de registros em um RDD.
- *foreach(f: T => Unit) : RDD[T] => Unit:* aplica uma função sobre todos os registros de um RDD. Geralmente utilizada para armazenar informações em banco de dados, ou imprimir os dados no console durante desenvolvimento, como demonstrado na Tabela 9.

Tabela 9 - Exemplo de função *foreach.*

```
val nums = spark.parallelize(Seq(1, 2, 3))
nums.foreach(num => println("Valor: " + num))
```

Fonte: elaborado pelo autor

- *collect() : RDD[T] => Array[T]:* coleta os registros em um array. Usado quando o *dataset* já foi processado e seu tamanho cabe na memória de uma única máquina.
- *save(path: String) : RDD[T] => Unit:* salva os registros em um sistema de arquivos externo.
- *reduce(f: (T, T) => T) : RDD[T] => T:* similar à transformação *reduceByKey*, mas combina todos os registros por inteiro, não agrupando por chave. Dessa forma, somente um valor existe como resultado, e não uma lista. A Tabela 10 exemplifica a soma todos os números de um RDD:

Tabela 10 - Exemplo de função *reduce.*

```
val nums = spark.parallelize(Seq(1, 2, 3))
val soma = nums.reduce((n1, n2) => n1 + n2) // 6
```

Fonte: elaborado pelo autor

3.2.6 *Caching* e persistência

Uma das funcionalidades chave que motivaram a criação do Spark é a possibilidade que resultados intermediários de *jobs* sejam compartilhados pela memória primária de maneira eficiente, sem que ocorra a perda da tolerância a falhas. No Spark, isso é possível através do método *persist* em RDDs. Esse método faz com que RDDs tenham seus resultados intermediários colocados em cache, de forma que os resultados da computação de cada partição

até determinado momento fique na memória primária de seus respectivos nodos, impedindo que todas as operações do DAG tenham que ser computadas repetidamente para cada ação, ou escritas e lidas de um disco externo (CHAMBERS; ZAHARIA, 2018).

Assim como nas operações de transformação, o método *persist* também é *lazy*, os resultados intermediários de um RDD só serão persistidos quando a primeira ação for disparada. Um exemplo do uso de persistência é demonstrados por Zaharia et al. (2012): supõe-se que um servidor web está sofrendo com vários erros ultimamente, com o intuído de analisar a linha de crescimento destes erros, o operador decide realizar algumas buscas interativas nos terabytes de *logs* armazenados em um sistema de arquivos HDFS.

Tabela 11 - Persistência de RDDs em memória.

```
1. lines = spark.textFile("hdfs://...")
2. errors = lines.filter(_.startsWith("ERROR"))
3. errors.persist()
4. errors.filter(_.contains("MySQL")).count()
5. errors.filter(_.contains("HDFS")).count()
6. errors.filter(_.contains("HDFS")).foreach(println)
```

Fonte: adaptado de Zaharia et al. (2012)

A linha 3 na Tabela 11 indica que o estado das partições do RDD *errors* deve ser mantida em memória nos seus respectivos nodos uma vez computado. A primeira ação, neste caso a contagem de todos os erros contendo “MySQL” na linha 4, é responsável por processar todas as operações até o momento, incluindo a leitura do arquivo do disco e filtragem das linhas com erros, levando um tempo completo de execução. As operações subsequentes no RDD *errors* no entanto, fazem uso das partições contendo as linhas de erro que ficaram armazenadas na memória primária, tornando-as mais eficientes. Caso a linha 3 *persist* não existisse, cada uma das ações, linhas 4, 5 e 6 executaria as operações 1 e 2 de forma isolada, resultando no total de três leituras e três filtragens ao invés de uma cada quando em cache.

Por padrão, o método *persist* usa o nível de armazenamento *MEMORY_ONLY*, que tenta armazenar as partições de um RDD na memória RAM. Caso o espaço não seja suficiente, as partições que não foram armazenadas são recalculadas individualmente cada vez quando necessário. Outras configurações de persistência são: *MEMORY_AND_DISK*, armazena o RDD na memória RAM, caso não seja suficiente, o restante é escrito em disco; e *DISK_ONLY* que armazena o RDD somente em um disco externo, geralmente usado quando o volume de dados intermediário ainda é consideravelmente grande (APACHE SOFTWARE FOUNDATION, 2018b).

O método *cache* também é comumente visto, sendo nada mais do que um alias para o método *persist* com nível de armazenamento *MEMORY_ONLY*. Além disto, Spark automaticamente monitora o uso da persistência e cache automaticamente, removendo dados mais antigos de acordo com a política LRU (*Least Recently Used*). RDDs persistidos também podem ser removidos manualmente pelo método *unpersist* (APACHE SOFTWARE FOUNDATION, 2018b).

3.2.7 *Distributed Shared Variables*

Spark provê outra abstração que auxilia no desenvolvimento de aplicações com RDDs chamada *Distributed Shared Variables*. Existem dois tipos de variáveis compartilhadas distribuídas: *broadcast variables* e *accumulators*. Essas variáveis podem ser acessadas dentro das funções especificadas pelo usuário (*map*, *filter* e etc) e são distribuídas pelos nodos de forma eficiente.

Quando uma variável de escopo maior é referenciada dentro de um *closure*, o seu valor é serializado, enviado para cada um dos nodos pela rede e deserializado. Se essa variável é usada em diferentes funções, ela é copiada repetidamente para cada da uma das máquinas. *Broadcast variables* permitem que dados imutáveis de volume considerável sejam compartilhados pelo *cluster* de forma eficiente, sem que seus valores sejam copiados inúmeras vezes. Essas variáveis são inicialmente carregadas pelo *driver*, e então são enviadas para os executores uma única vez como ilustrado na Figura 14 (CHAMBERS; ZAHARIA, 2018).

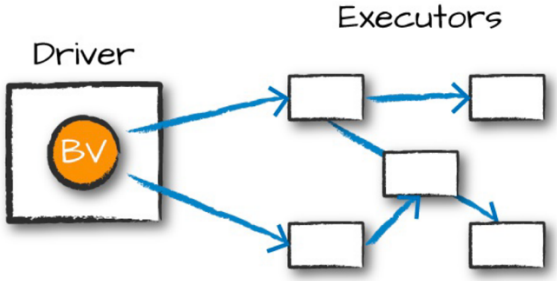


Figura 14 - *Broadcast variables*.
Fonte: Chambers e Zaharia (2018)

O trecho de código na Tabela 12 demonstra como distribuir variáveis imutáveis pelo *cluster* de forma eficiente usando o método *broadcast* de um *sparkContext*. A variável *bigMap* representa um *hashmap* com muitos registros, onde ao invés de utiliza-la diretamente na função *nums.map* do Spark, é criado um *broadcast* desta variável usando o método *spark.broadcast*. Dessa forma, caso fosse referenciada mais de uma vez em funções do Spark, o *hashmap* seria enviado uma única vez pelo *cluster*.

Tabela 12 - Uso de *broadcast variables*.

```
val bigMap = Map("a" -> 1, "b" -> 3, "c" -> 55)
val mapBroadcast = spark.broadcast(bigMap)

val nums = spark.parallelize(Seq("a", "b"))
nums.map(n => mapBroadcast.value.get(n))
```

Fonte: elaborado pelo autor

Por outro lado, *accumulators* permitem que executores atualizem o valor de variáveis dentro de transformações de forma paralela sem que problemas de concorrência ocorram. Esse tipo de variável só pode ser incrementada, e provê uma abordagem imperativa quanto à criação de agregações, como um contador. Como pode ser observado na Figura 15, a variável é inicialmente declarada pelo *driver*, e os executores incrementam o seu valor localmente durante o processamento de suas tarefas, que é então copiada para o *driver* no final de cada *job* (CHAMBERS; ZAHARIA, 2018).

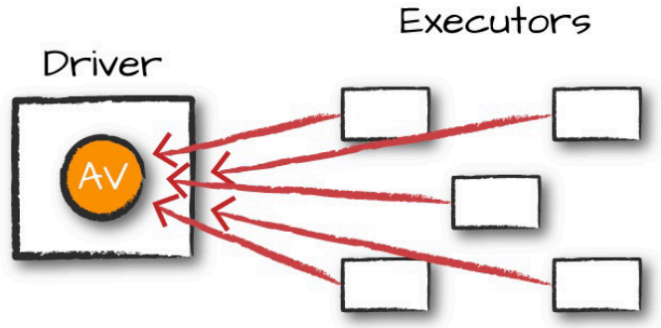


Figura 15 – *Accumulators*.
Fonte: Chambers e Zaharia (2018)

Assim como *broadcast variables*, *accumulators* podem ser instanciados a partir do contexto do Spark, o exemplo abaixo utiliza um *accumulator* para contar a quantidade de números pares em um RDD de quatro partições. Como pode ser visto na Figura 16, essas variáveis também podem ser monitoradas pela UI web do Spark, aplicação auxiliar que mantem métricas de *jobs*, tarefas, executores e outros detalhes.

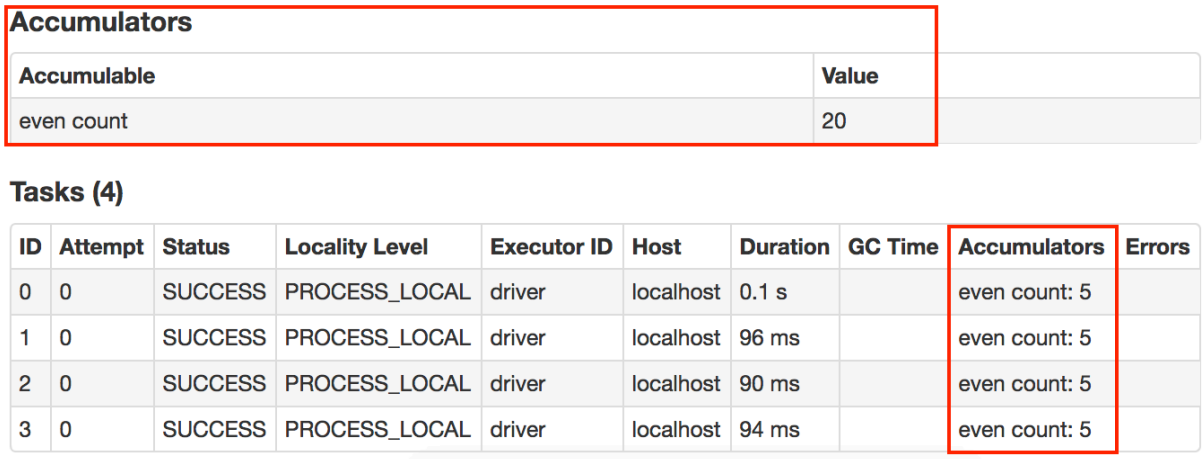
Tabela 13 - Uso de *accumulators*.

```
val accumulator = spark.longAccumulator("even count")

val nums = spark.parallelize(seq = 1 to 40, numSlices = 4)
nums.foreach(n => if (n % 2 == 0) accumulator.add(1))

println(accumulator.value) // 20
```

Fonte: elaborado pelo autor



Accumulators

Accumulable	Value
even count	20

Tasks (4)

ID	Attempt	Status	Locality Level	Executor ID	Host	Duration	GC Time	Accumulators	Errors
0	0	SUCCESS	PROCESS_LOCAL	driver	localhost	0.1 s		even count: 5	
1	0	SUCCESS	PROCESS_LOCAL	driver	localhost	96 ms		even count: 5	
2	0	SUCCESS	PROCESS_LOCAL	driver	localhost	90 ms		even count: 5	
3	0	SUCCESS	PROCESS_LOCAL	driver	localhost	94 ms		even count: 5	

Figura 16 - *Accumulators* na interface web do Spark.
Fonte: elaborado pelo autor

3.3 DataFrames

DataFrames é uma API estruturada baseada em RDDs que surgiu na versão 1.3 do Spark (2013). Assim como RDDs, DataFrames representam coleções de dados imutáveis e distribuídos. Contudo, os dados em um DataFrame são organizados em linhas e colunas, sendo conceitualmente equivalentes a uma tabela em um banco de dados relacional. DataFrames são estruturados no sentido que seus dados precisam respeitar um *schema* que é definido durante sua criação. Essa propriedade permite que Spark otimize os caminhos de execução e obtenha ganhos de performance significativos (APACHE SOFTWARE FOUNDATION, 2018b).

Em DataFrames, Spark usa internamente uma *engine* de processamento chamada Catalyst, que trabalha inteiramente com seus próprios tipos de dados (IntegerType, LongType, StringType, etc), permitindo que os dados sejam mantidos fora da memória *heap*, podendo também ser enviados pelo cluster de uma maneira mais eficiente em comparação com a serialização do Java. Como pode ser visto na Tabela 14, as transformações em DataFrames são similares a SQL, uma vez que são aplicadas através de colunas nomeadas, definidas a partir de um *schema* que contém meta-dados como o tipo do dado das colunas.

Tabela 14 - *Schema* e transformação *where* em DataFrames.

```
val dataframe = spark.read.json("people.json")
dataframe.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
dataframe.where("age > 21")
```

Fonte: elaborado pelo autor

Todo código escrito em DataFrames é convertido em um plano lógico intermediário pelo *Catalyst*, nesta fase, as expressões do usuário são validadas de acordo com o *schema* do *dataset*, o plano gerado ainda não representa fisicamente como os executores devem computar as transformações e ações. Em um segundo momento, o plano lógico intermediário é convertido em um plano físico (séries de RDDs) que será executado no *cluster*. Como pode ser observado na Figura 17, o plano físico é escolhido a partir de um modelo de custo, diversos planos são gerados e tem seus custos comparados em relação ao *dataset* em questão, levando em consideração o volume dos dados, número de partições e outras propriedades (CHAMBERS; ZAHARIA, 2018). Isso permite que DataFrames obtenham performance ótima, melhorando transformações não otimizadas pelo usuário e levando as especificidades de cada *dataset* em consideração ao criar um plano de execução.

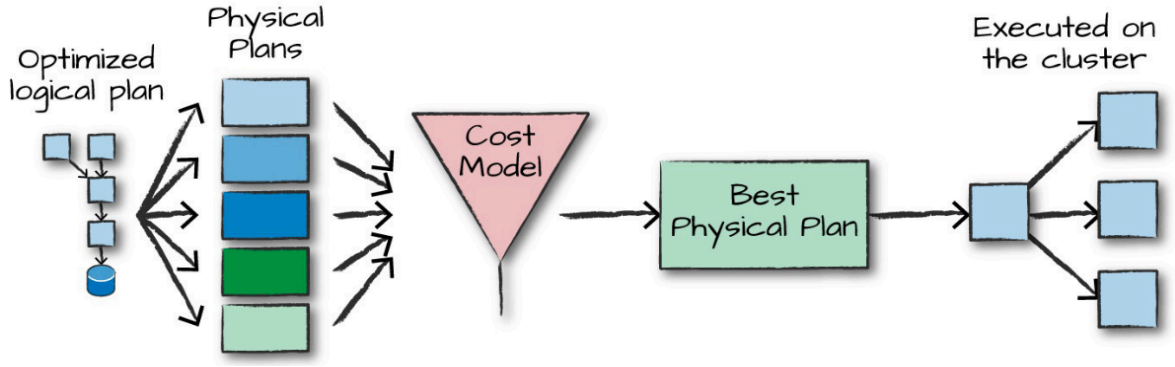


Figura 17 - Processo de escolha de um plano físico.
Fonte: Chambers e Zaharia (2018)

A Tabela 15 demonstra um exemplo de como realizar uma contagem de pessoas por idade usando DataFrames. A função *selectExpr* é similar ao *map* em RDDs e transforma a data de nascimento em idade, os registros são filtrados por idade menor que 50 usando *where*, e são então agrupados e contados. Pode-se observar que uma desvantagem desta API é a perda de *type-safety*, como os dados não operados na JVM e sim diretamente na *engine* do Spark, as manipulações são feitas com *strings*, caso uma coluna não exista ou seja de um tipo inesperado, a aplicação irá falhar em tempo de execução e não de compilação.

Tabela 15 - Contagem de pessoas por idade em DataFrames.

```
val df: DataFrame = spark.read.json("people.json")
df.show()

// +-----+------+
// |birth|  name|
// +-----+------+
// | 1990|  Andy|
// | 1985|Justin|
```

```
// | 1932|  John|
// | 1990|  Eric|
// +-----+------+

df.selectExpr("name", "2018 - birth AS age")
  .where("age < 50")
  .groupBy("age")
  .count()
  .show()

// +---+-----+
// |age|count|
// +---+-----+
// | 28|    2|
// | 33|    1|
// +---+-----+
```

Fonte: adaptado de Apache Software Foundation (2018b)

3.4 Datasets

Datasets é uma API baseada em DataFrames que surgiu na versão 1.6 do Spark em 2015. Assim como RDDs e DataFrames, Datasets também representam coleções de dados imutáveis e distribuídas. Essa nova abstração foi criada para disponibilizar *type-safety*, enquanto mantendo as otimizações fornecidas pela *catalyst engine*. As manipulações disponíveis na API de Datasets são muito similares às encontradas em RDDs, funções *lambda* ou *closures* são passadas em transformações e são aplicadas com tipos da JVM, uma vez que Datasets operam em tipos genéricos (String, Int, Person, etc) ao invés de linhas e colunas como em DataFrames (CHAMBERS; ZAHARIA, 2018).

O tipo genérico de um Dataset é inferido como seu *schema*. Isso permite que as mesmas otimizações encontradas em DataFrames sejam aplicadas também em Datasets, enquanto provendo uma abordagem funcional de programação e garantindo que o código é valido durante compilação. Contudo, como as definições das funções operam com tipos da JVM, estes precisam ser convertidos em seus respectivos tipos da *engine* do Spark, criando um certo *overhead* em relação à DataFrames (CHAMBERS; ZAHARIA, 2018).

A Tabela 16 demonstra a mesma contagem de pessoas por idade feita na Tabela 15, agora utilizando a API de Datasets. O Dataset é tipado pela classe *Person* a partir do método *as[Person]*, e em seguida, todas as transformações podem ser manipuladas usando objetos da JVM. Caso *age* não fosse um tipo numérico, o compilador acusaria erro nas expressões *2018 - p.age* e *_.age < 50*.

Tabela 16 - Contagem de pessoas por idade em Datasets.

```
case class Person(name: String, birth: Long, var age: Long)

val ds: Dataset[Person] = spark.read.json("people.json").as[Person]
ds.map(p => Person(p.name, p.birth, 2018 - p.birth))
  .filter(_.age < 50)
  .groupByKey(_.age)
  .count()
  .show()
```

Fonte: elaborado pelo autor

Cada uma das APIs possui suas vantagens e desvantagens, e por conta disso, Spark permite que aplicações possam facilmente converter entre abstrações, mesmo que no meio de uma corrente de transformações, adaptando-se à API que melhor se encaixa em cada cenário específico. O Quadro 1 mostra uma comparação geral entre as três abstrações encontradas no Spark.

Quadro 1 - Comparação APIs DataFrame, Dataset e RDD.

	DataFrame	Dataset	RDD
Erros de sintaxe	Tempo de execução	Compilação	Compilação
Otimizado para dados	Semi-estruturados e estruturados	Semi-estruturados e estruturados	Não-estruturados
Performance	Alta, serialização e plano de execução otimizados	Média, plano de execução otimizado, mas ainda opera em objetos da JVM	Média, depende do usuário e opera em objetos da JVM
Manipulação	Declarativa	Funcional	Funcional
Abstração	Alto nível	Alto nível	Baixo nível, permite manipulação de partições e tipos complexos.

Fonte: adaptado de Chambers e Zaharia (2018)

3.5 Considerações finais do capítulo

Spark traz uma nova abordagem ao processamento distribuído, através de sua abstração base, RDDs, operações de transformação são aplicadas em *datasets* e paralelizadas automaticamente em um *cluster*. Caso a execução de uma tarefa falhe, a mesma pode ser reconstruída a partir do DAG, não necessitando que os dados sejam armazenados externamente a cada *job* para garantir tolerância a falhas. Além disto, o estado de um RDD em determinado momento pode ser salvo em memória, evitando que resultados intermediários tenham que ser

escritos e lidos de um sistema de arquivos externo a cada iteração. DataFrames e Datasets proveem uma abstração de alto nível baseada em RDDs que oferece otimizações no plano físico a partir da *engine* Catalyst.

4 ANÁLISE DE ASSOCIAÇÃO

Com o grande acúmulo de dados visto atualmente, técnicas de mineração de dados permitem a descoberta de características escondidas sobre estas informações. Uma destas técnicas é a mineração de regras de associação, usada para extrair relações fortes entre itens em um conjunto de transações. Historicamente, essa técnica é ilustrada na análise de compras de um supermercado, onde os produtos que são vendidos juntos mais frequentemente são identificados. Além disto, regras de associação também podem ser vistas na análise de páginas web comumente acessadas juntas, e na detecção de ataques na área de segurança de informação (RATHEE; KAUL; KASHYAP, 2015).

O Quadro 2 ilustra um *dataset* contendo compras feitas em um supermercado, onde cada linha é uma transação identificada por um TID e representa o conjunto de itens comprados pelo cliente. As relações entre os itens são representadas na forma de regras de associação, como por exemplo, *{Fralda} => {Cerveja}*. Essa regra sugere que existe uma forte correlação que a compra de fraldas frequentemente incide na compra de cerveja (TAN; STEINBACH, 2005).

Quadro 2 - Exemplo de transações de um supermercado.

TID	Itens
1	{Pão, Leite}
2	{Pão, Fralda, Cerveja, Ovos}
3	{Leite, Fralda, Cerveja, Refrigerante}
4	{Pão, Leite, Fralda, Cerveja}
5	{Pão, Leite, Fralda, Refrigerante}

Fonte: adaptado de Tan e Steinbach (2005)

O conjunto de itens distintos encontrados no *dataset* é definido por $I = \{i^1, i^2, ..., i^n\}$ e $T = \{t^1, t^2, ..., t^n\}$ representa todas as transações existentes. Cada transação t^i é composta por um subconjunto de *I*. Um conjunto de itens é chamado de *itemset*, mais especificamente, se um itemset contém *k* itens, ele é definido como *k-itemset*. Por exemplo, o *itemset {Cerveja, Fralda, Leite}* é um *3-itemset*. Uma importante propriedade de um *itemset* é seu suporte, que representa o número de vezes em que o mesmo aparece nas transações em todo *dataset*. Dessa forma, o suporte de {Pão, Leite} é 3 ou 60%, pois aparecem juntos 3 vezes nas 5 transações (TAN; STEINBACH, 2005).

Uma regra de associação é representada por $X => Y$, ou seja, X incide em Y, onde X e Y são subconjuntos de itens de I. A força desta regra é mensurada pelo seu suporte e confiança. O suporte é definido pela frequência em que o antecedente X e consequente Y aparecem juntos nas transações. Já a confiança indica a importância de uma regra, é a probabilidade da regra ocorrer quando o antecedente acontece. Por exemplo, considerando a regra *{Leite, Fralda} => {Cerveja}*. O *itemset {Leite, Fralda, Cerveja}* ocorre 2 vezes das 5 transações, então seu suporte é *2/5 = 0,4*. A confiança desta regra é obtida dividindo o suporte do *itemset* pelo suporte do antecedente *{Leite, Fralda}*, que é 3, ou seja, a confiança dessa regra á 2/3 ou 67% (TAN; STEINBACH, 2005).

4.1 Algoritmos

Algoritmos de associação buscam encontrar as regras em um *dataset* que respeitam o suporte e confiança mínimos estipulados pelo usuário. Uma estratégia comum nestes algoritmos é a divisão do problema em duas tarefas: 1. encontrar os *itemset* frequentes, ou seja, gerar todos os conjuntos de itens possíveis no *dataset* que respeitam o suporte estipulado, também conhecido por *Frequent Itemset Mining* ou FIM; e 2. extrair as regras que atingem a confiança estipulada a partir dos *itemset* frequentes encontrados no passo anterior (TAN; STEINBACH, 2005).

A tarefa de geração dos *itemsets* frequentes tem complexidade computacional mais alta em relação à extração das regras, e é responsável por uma grande parcela no tempo total de execução. A Figura 18 mostra todas as combinações dos possíveis *itemsets* frequentes para $I = \{a, b, c, d, e\}$, também chamados de *candidate itemsets*. Um *dataset* contendo k itens distintos pode potencialmente gerar até $2^k - 1$ *itemset* frequentes. Como k pode ser muito grande em algumas aplicações, o número de *itemsets* a serem verificados é exponencialmente alto (TAN; STEINBACH, 2005).

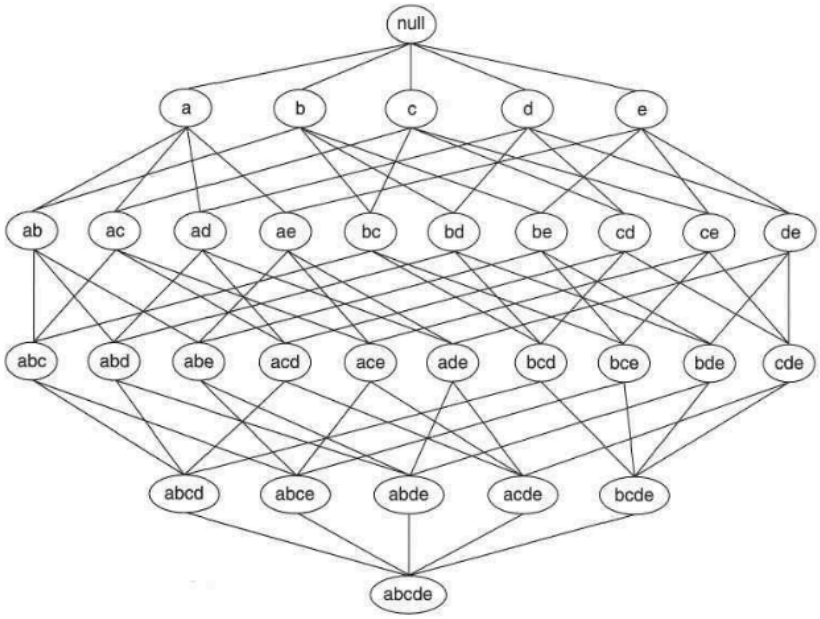


Figura 18 – Possíveis combinações de *itemsets para {a, b, c, d, e}*.
Fonte: Tan e Steinbach (2005)

A abordagem força-bruta para encontrar os *itemsets* frequentes, ou seja, aqueles que possuem um suporte mínimo, incide em gerar a completa lista de $2^k - 1$ *itemsets*, comparando cada *candidate itemset* com a lista de transações, e incrementando um contador caso o *itemset* seja um subconjunto da transação. Essa abordagem é extremamente custosa e impraticável para grandes *datasets*. Por conta disto, algoritmos mais eficientes como o Apriori e FP-Growth foram desenvolvidos (TAN; STEINBACH, 2005).

4.1.1 Apriori

Apriori é um algoritmo de mineração de regras de associação proposto por Agrawal Srikant em 1994 que reduz o número de *candidate itemsets* gerados durante a etapa de FIM, diminuindo consideravelmente o processamento necessário. Esse algoritmo é baseado no princípio de que se um *itemset* é frequente, então todos os seus *subsets* também são frequentes. Ou seja, se *{c, e}* é frequente, então *{c}* e *{e}* também são frequentes. Seguindo esta lógica, também é correto assumir que caso um *itemset* não seja frequente, todos os seus *supersets* também não serão frequentes. Essa propriedade pode ser observada na Figura 19, onde caso *{a, b}* seja verificado infrequente, todos os seus *supersets* podem ser desconsiderados (TAN; STEINBACH, 2005).

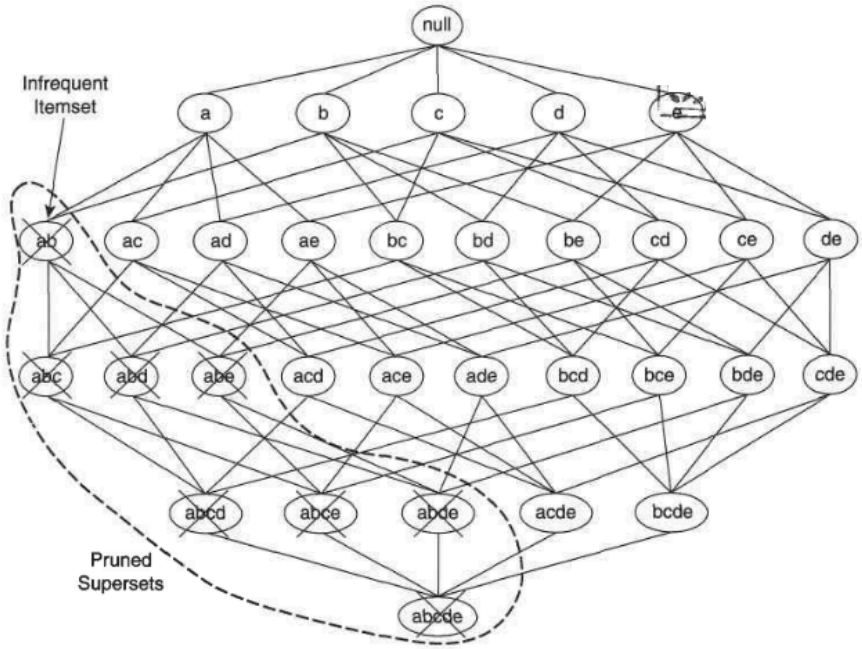


Figura 19 – Princípio usado pelo Apriori durante a etapa de FIM.
Fonte: Tan e Steinbach (2005)

A partir deste princípio, o algoritmo Apriori controla o crescimento exponencial de *itemsets* verificados. Os *itemsets* frequentes são encontrados de forma iterativa, calculando inicialmente os *itemsets* frequentes de tamanho 1, em seguida, somente estes são usados para geração dos *itemsets* candidatos de tamanho 2, e da mesma maneira, somente os *2-itemsets* frequentes encontrados são usados para geração dos *3-itemsets* candidatos, e assim por diante. O tamanho dos *itemsets* gerados cresce a cada iteração e o algoritmo é finalizado no momento em que nenhum *itemset* frequente é encontrado em uma iteração. Essa abordagem faz com que o número de *itemsets* gerados seja diminuído, melhorando drasticamente o desempenho do algoritmo (TAN; STEINBACH, 2005).

4.1.2 FP-Growth

Duas características do algoritmo Apriori fazem que ele não seja tão eficiente em certos cenários. Primeiro, o *dataset* é escaneado múltiplas vezes, cada iteração incide na varredura do *dataset* por inteiro. Segundo, mesmo com a otimização proposta, uma grande quantidade de *itemsets* candidatos é gerada e tem seus suportes calculados, muito destes podem não aparecer uma única vez em nenhuma transação. Por conta disto, FP-Growth foi proposto por Han, Pei e Yin em 2000 para resolver estes problemas (XIUJIN; SHAOZONG; HUI, 2017).

FP-Growth toma uma abordagem diferente na mineração de *itemsets* frequentes. Diferente do Apriori, ele não gera *itemsets* candidatos e em seguida verifica se estes existem e

tem um suporte mínimo. FP-Growth codifica o *dataset* e usa uma estrutura de dados compacta chamada FP-Tree, extraindo os *itemsets* frequentes diretamente desta estrutura. Ou seja, *itemsets* candidatos não são gerados, uma vez que *itemsets* são diretamente derivados das transações. Como ilustrado na Figura 20, uma FP-Tree representa todos os caminhos possíveis de *itemsets*, mantendo a frequência em que cada caminho ocorre. Nesse exemplo, considerando somente o ramo mais à esquerda da árvore, é possível identificar que o itemset {a, b, c, d} ocorre uma vez, {a, b, c} três vezes, {a, b} cinco vezes e {a} ocorre 8 vezes (TAN; STEINBACH, 2005).

TID	Items
1	{a,b}
2	{b,c,d}
3	{a,c,d,e}
4	{a,d,e}
5	{a,b,c}
6	{a,b,c,d}
7	{a}
8	{a,b,c}
9	{a,b,d}
10	{b,c,e}

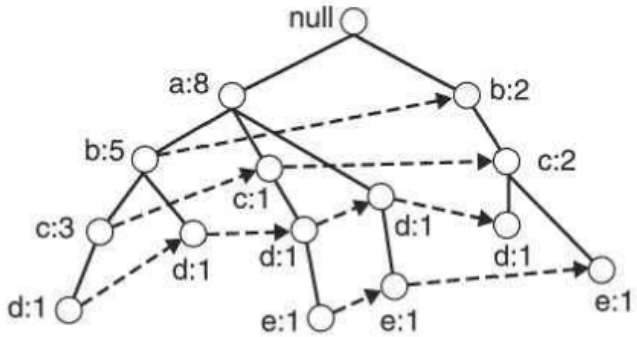


Figura 20 - Construção da FP-Tree a partir de um *dataset.*
Fonte: Tan e Steinbach (2005)

Depois de construída, a FP-Tree é então decomposta em múltiplos subproblemas, onde cada subproblema busca encontrar os *itemsets* frequentes específicos para cada item. Para cada partição da FP-Tree, uma FP-Tree condicional é gerada contendo a frequência de cada caminho que termina com determinado item, a Figura 21 ilustra a FP-tree de *{e}* e sua respectiva FP-tree condicional. *Itemset* frequentes são então encontrados de forma recursiva repetindo este procedimento de baixo para cima para todas as combinações de itens possíveis, removendo aqueles que não atingem um suporte mínimo requerido (TAN; STEINBACH, 2005).

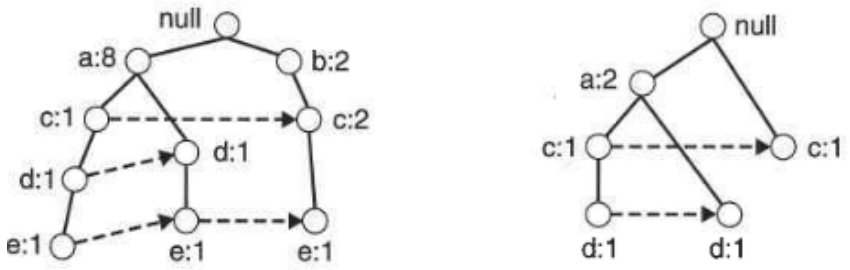


Figura 21 – FP-Tree para *{e}* na esquerda e FP-Tree condicional para *{e}* na direita considerando um suporte mínimo de 2.
Fonte: Tan e Steinbach (2005)

Em geral, FP-Growth tem um desempenho muito melhor do que o algoritmo Apriori na maioria dos casos. Contudo, a eficiência dessa abordagem depende do fator de compactação do *dataset*, se a FP-Tree conter uma imensa quantidade de folhas – casos onde a maioria das transações produzem um caminho distinto – a performance pode ser degradada rapidamente (TAN; STEINBACH, 2005).

4.2 Trabalhos relacionados

Pesquisas relacionadas ao tema escolhido foram buscadas nos motores de pesquisa IEEE e ACM. Inicialmente, uma pesquisa mais ampla foi feita para identificar as possíveis áreas de interesse relacionadas ao Spark. A *string* de busca *("Document Title": "spark" AND algorithm)* na IEEE resultou em cerca de 300 estudos, destes, uma boa parcela cobre áreas de mineração de dados, como associação e classificação. Essa área foi escolhida e usada para restringir a próxima busca: *("Document Title": "spark" AND algorithm AND ("Abstract":mining OR "Abstract":association OR "Abstract":classification)),* que obteve 77 resultados no IEEE.

Foram identificadas diversas propostas de algoritmos relacionados às regras de associação e mineração de *itemsets* frequentes. Dentre as implementações paralelas encontradas, decidiu-se focar naquelas baseadas nos algoritmos tradicionais Apriori e FP-Growth. Dessa forma, a seguinte *string* de busca foi usada como base para o estudo dos trabalhos relacionados: *("Document Title": "spark" AND ("Abstract": Apriori OR "Document Title": fp-growth))*, contando com 12 resultados na IEEE e 2 no ACM. Dentre os artigos encontrados, foram selecionados YAFIM, R-Apriori e DFPS para um estudo mais aprofundado, uma vez que estes trazem características distintas e melhorias para a mineração de *itemsets* frequentes. Além disto, também trazendo espaço para comparar o desempenho entre R-Apriori e DFPS, uma vez que essa comparação não foi abordada em nenhum dos artigos.

4.2.1 YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark

O algoritmo Apriori é amplamente usado para mineração de *itemsets* frequentes. Contudo, a natureza iterativa deste algoritmo faz com que o *dataset* tenha que ser lido diversas vezes, tornando-o computacionalmente intenso quando usado com alto volume de dados. Pesquisadores implementaram diversas versões distribuídas do algoritmo Apriori usando o *framework* MapReduce, provendo escalabilidade quando executado em grandes *datasets* (QIU et al., 2014).

Qiu et al. (2014) propõe YAFIM (*Yet Another Frequent Itemset Mining*), a primeira implementação do algoritmo Apriori no Spark baseado em RDDs. Tendo como objetivo prover uma implementação do algoritmo Apriori usando as capacidades de processamento em memória do Spark. Além disto, o desempenho do algoritmo proposto é comparado com a implementação paralela referência do Apriori no MapReduce, o MRApriori.

YAFIM consiste de duas fases. A primeira fase se encarrega por carregar o *dataset* do HDFS em RDDs, assim como gerar os *singletons*, lista de *itemsets* frequentes contendo apenas um item cada. A segunda fase é responsável por gerar a lista de *(k+1)-itemsets* frequentes usando o resultado da iteração anterior *k-itemsets* como entrada. Cada iteração resulta em uma lista de itemsets frequentes que respeitam o suporte especificado, as iterações são interrompidas quando nenhum novo *itemset* frequente é encontrado.

A Figura 22 representa o grafo acíclico de operações executadas durante a fase 1. O arquivo de entrada é lido e armazenado em um RDD *Transactions*, uma função *flatMap()* é então aplicada retornando um RDD que contém todos os itens do *dataset*. A função *map()* transforma cada item em uma tupla *<item, 1>* que são por fim acumulados usando a função *reduceByKey()*. Itens que não obtém uma frequência maior ou igual ao suporte especificado são filtrados. Como resultado, a lista de *1-temsets* frequentes que respeitam o suporte mínimo são então usados para calcular os *2-itemsets* seguintes na fase 2.

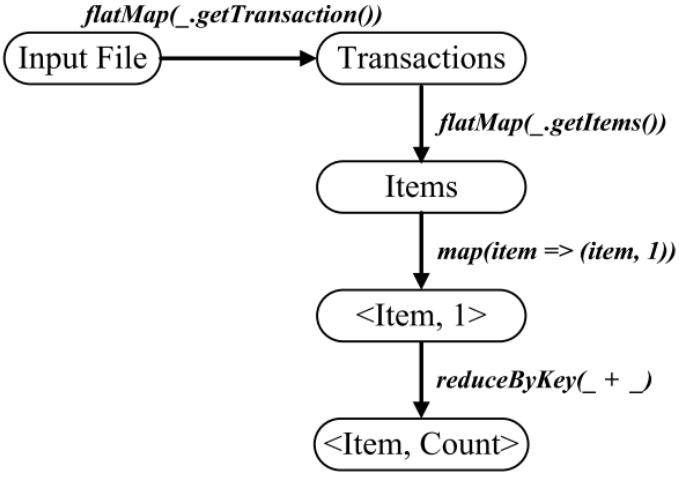


Figura 22 - Operações executadas na fase 1 do YAFIM.
Qiu et al. (2014)

Em seguida, a fase 2 representa as próximas iterações. Nessa fase, o RDD Transactions contendo o *dataset* inicialmente lido na fase 1 é mantido em memória e reaproveitado, não causando um *overhead* adicional de IO. A Figura 23 mostra o conjunto de operações executadas na fase 2.

Como ilustrado na Figura 23, o conjunto de *k-itemsets* frequentes da iteração anterior no formato de <*Itemset, Count*> é usado para gerar a lista de *candidate itemsets* C_{k+1}. A lista de *candidate itemsets* é armazenada em uma *hash tree* para aumentar a performance durante as buscas. Em seguida, o RDD Transactions é escaneado em busca de todas as ocorrências dos *candidate itemsets*, cada registro encontrado é então transformado em uma tupla <*Itemset, 1*> que é então acumulado usando a função *reduceByKey()*. Na última etapa, os candidatos que não atingiram o suporte mínimo são filtrados, resultando os *itemsets* frequentes da respectiva iteração.

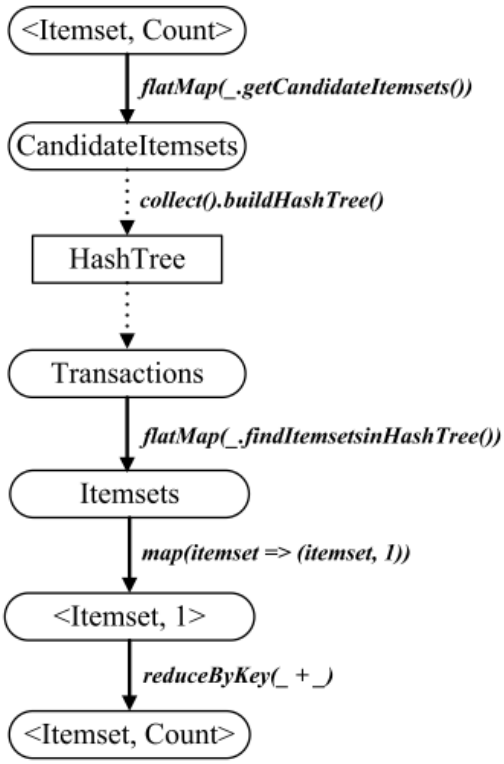


Figura 23 - Operações executadas na fase 2 do YAFIM.
Qiu et al. (2014)

Experimentos foram realizados em diversos *datasets*. Os resultados mostrados abaixo são respectivos ao *dataset* artificialmente gerado pela IBM, que contém 870 itens e 100.000 transações. O ambiente usado foi um *cluster* de 12 nodos *octa-core*. A Figura 24 mostra a performance do YAFIM em relação ao MRApriori em cada iteração, que ficou em média 10 vezes mais rápido.

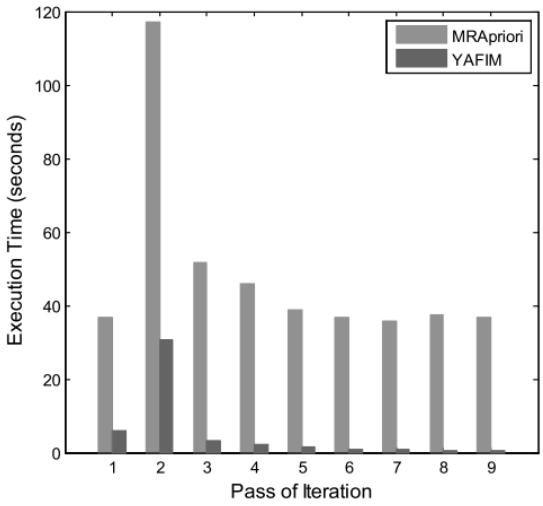


Figura 24 - Tempo de execução por iteração. YAFIM e MRApriori.
Qiu et al. (2014)

Além disso, a escalabilidade do algoritmo foi analisada replicando o *dataset* 2, 3, 4, 5 e 6 vezes, mantendo o número total de cores fixos em 48. A Figura 25 mostra o tempo de execução no eixo y e o número de vezes que o *dataset* foi replicado no eixo x. Como pode ser observado, o tempo de execução com MRApriori é aumentado drasticamente em cada replicação, por outro lado, YAFIM tem um crescimento pequeno de acordo com o aumento no volume do *dataset*.

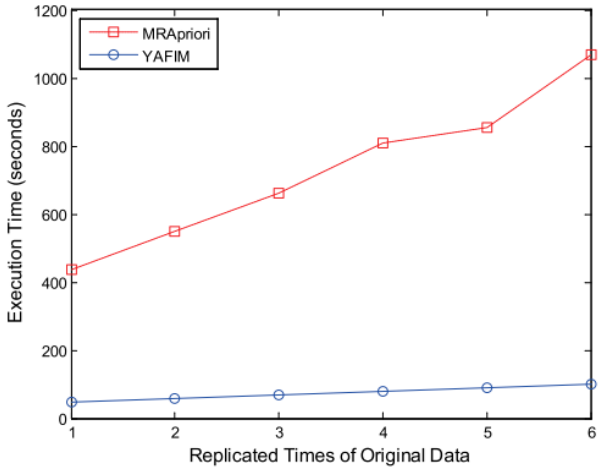


Figura 25 - Escalabilidade com diferentes volumes de dados.
Qiu et al. (2014)

Por fim, Qiu et al. (2014) conclui que o modelo de computação em memória disponibilizado no Spark permite uma drástica redução nos tempos de IO e comunicação de rede em algoritmos iterativos. Demonstrando que o algoritmo YAFIM é em média 18 vezes mais rápido do que algoritmos Apriori implementados no *framework* MapReduce sem o Spark.

4.2.2 R-Apriori: An Efficient Apriori based Algorithm on Spark

R-Apriori foi proposto por Rathee, Kaul e Kashyap (2015) e traz otimizações no algoritmo YAFIM anteriormente apresentado por Qiu et al. (2014). Foi identificado que quando YAFIM é usado para processar um grande número de itens distintos e com um suporte mínimo baixo, a etapa de geração de *candidate itemsets* na segunda iteração se torna muito custosa. Por exemplo, se a primeira iteração gerou 10.000 *singletons* frequentes, existem praticamente 1.000.000 de combinações possíveis de *candidate itemsets*, que precisam ser posteriormente validados por cada uma das transações existentes no *dataset*, o que acaba tendo um alto custo no tempo total de processamento (RATHEE; KAUL; KASHYAP, 2015).

R-Apriori é uma implementação paralela do Apriori que se baseia na implementação do YAFIM, incluindo uma fase adicional para mitigar a característica descrita acima. Assim como no YAFIM, a primeira fase do R-Apriori armazena o *dataset* em RDDs, que são mantidos em memória durante todo o processamento, e identifica quais são os *singletons* frequentes. A segunda fase, entretanto, elimina a geração de *candidate itemsets* e faz uso de *bloom filters* (RATHEE; KAUL; KASHYAP, 2015) ao invés de *hash trees,* agilizando a busca dos *singletons* durante o processamento. A terceira fase representa as próximas iterações na geração de *k-itemsets* frequentes, e como neste ponto o número de *candidate itemsets* já está reduzido, R-Apriori utiliza o mesmo mecanismo da fase 2 do algoritmo YAFIM.

A Figura 26 ilustra o fluxo de operações na execução da fase 2. Primeiramente, os *singletons* contendo *1-itemsets* são armazenados em um *bloom filter*, em seguida, o RDD contendo a lista com todas transações em memória é transformado usando *flatMap()* para somente conter itens frequentes encontrados na fase 1. A função *flatMap()* é novamente usada para transformar cada transação em uma lista de pares possíveis para tal transação. O RDD é então mapeado para o formato de tupla *(pair, 1)*, sendo posteriormente combinado usando a função *reduceByKey()*, filtrando os registros que não alcançam o suporte mínimo requerido. Como resultado, a lista de *2-itemsets* frequentes é gerada no formato *(Pair, Count)*.

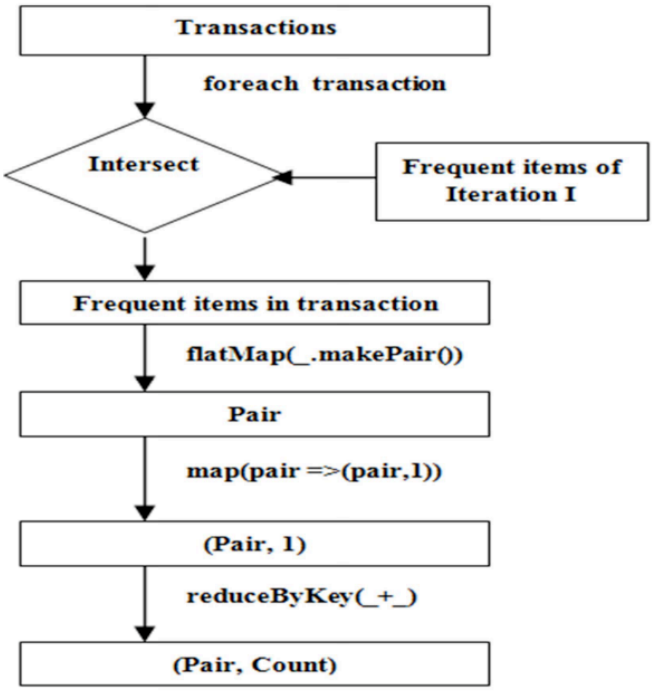


Figura 26 - Operações executadas na fase 2 do R-Apriori.
Rathee, Kaul e Kashyap (2015)

Experimentos foram realizados com diversos *datasets* e com suportes mínimos pequenos, entre 0,0% e 1%. Todos os testes foram executados em *clusters* de 48 nodos com 340GB de memória RAM no total. A Figura 27 mostra o tempo de execução para cada iteração do R-Apriori em relação ao YAFIM, usando suporte de 1% e um *dataset* contendo 990 itens e 4.900 transações. Observa-se que a segunda iteração chega a ser 4 vezes mais rápida quando comparada ao YAFIM. Como esperado, as demais iterações têm tempo de execução idênticos.

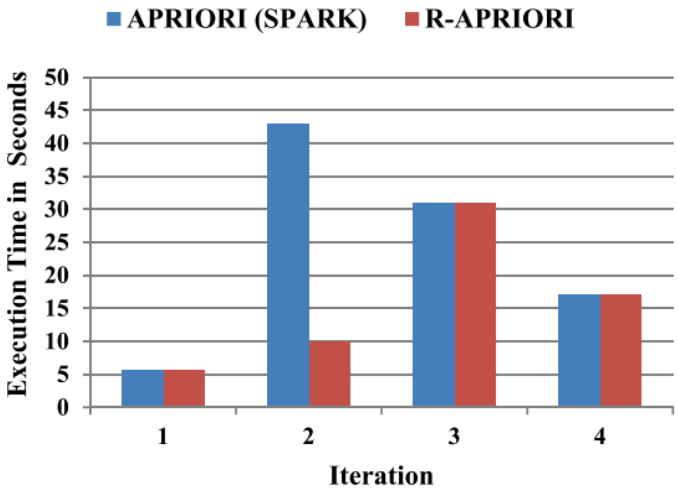


Figura 27 - Tempo de execução por iteração, suporte de 1%. R-Apriori e YAFIM.
Rathee, Kaul e Kashyap (2015)

O mesmo experimento foi também realizado em um *dataset* de varejo contendo um número maior de itens, 16.470 itens e 88.163 transações. A Figura 28 mostra que a segunda iteração do R-Apriori chaga a ser 9 vezes mais rápida com um suporte mínimo baixo de 0,15%. Novamente, as demais iterações têm tempo de execução idênticas.

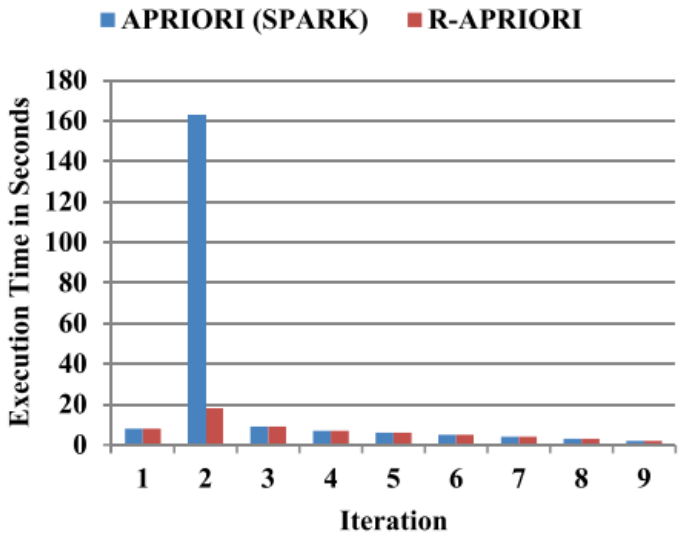


Figura 28 - Tempo de execução por iteração, suporte de 0,15%. R-Apriori e YAFIM.
Rathee, Kaul e Kashyap (2015)

Dessa forma, Rathee, Kaul e Kashyap (2015) concluem que o algoritmo R-Apriori proposto se mostra mais eficiente do que o algoritmo YAFIM para mineração de *itemsets* frequentes em grandes *datasets* com diversos atributos e suporte mínimo baixo. Visto que implementações Apriori convencionais acabam por obter altos tempos de execução na segunda iteração devido ao alto número de *candidate itemsets* gerados. Isso é mitigado no R-Apriori pela eliminação desta etapa na segunda iteração, que é substituída pela direta geração de *itemsets* frequentes diretamente a partir das transações do *dataset* e pelo uso de *bloom filters*.

4.2.3 DFPS: Distributed FP-growth Algorithm Based on Spark

Algoritmos paralelos que se baseiam no Apriori como YAFIM e R-Apriori permitem que *itemsets* frequentes sejam calculados de forma distribuída em um *cluster* sem que o *dataset* tenha que ser lido múltiplas vezes de um sistema de arquivos externo. Isso acelera o processo consideravelmente como mostrado por Qiu et al. (2014) e Rathee, Kaul e Kashyap (2015). Contudo, esse tipo de algoritmo ainda depende da geração de *candidate itemsets*, ocasionando na multiplicação do volume de dados a ser processado. Além disso, se o *dataset* é grande o suficiente, o algoritmo pode falhar, uma vez que o conjunto de *candidate itemsets* precisa ser armazenado em memória (XIUJIN; SHAOZONG; HUI, 2017).

Diferentes abordagens paralelas para mineração de *itemsets* frequentes foram propostas com base no algoritmo FP-Growth, que elimina o passo de geração de *candidate itemsets* por inteiro e reduz o volume dados processados. Li et al. (2008) traz uma implementação paralela do FP-Growth (PFP) no modelo de programação MapReduce. Entretanto, conta com alto uso de I/O de disco em cada operação. Por conta disto, Xiujin, Shaozong e Hui (2017) propuseram DFPS, uma implementação distribuída do FP-Growth com Spark, que busca eliminar o volume de dados adicional processado por YAFIM e R-Apriori, e o *overhead* de disco visto no PFP.

FP-Growth funciona em três fases. A primeira fase carrega o *dataset* do HDFS para RDDs, e gera o conjunto de *singletons* frequentes, etapa similar ao que é visto no YAFIM e R-Apriori. A segunda fase filtra cada transação para conter apenas itens frequentes, e cria uma lista de padrões condicionais base – ideia similar à geração da FP-Tree condicional – que é então particionada no *cluster* de acordo com a lista de *singletons* frequentes. Este processo é ilustrado na Figura 29.

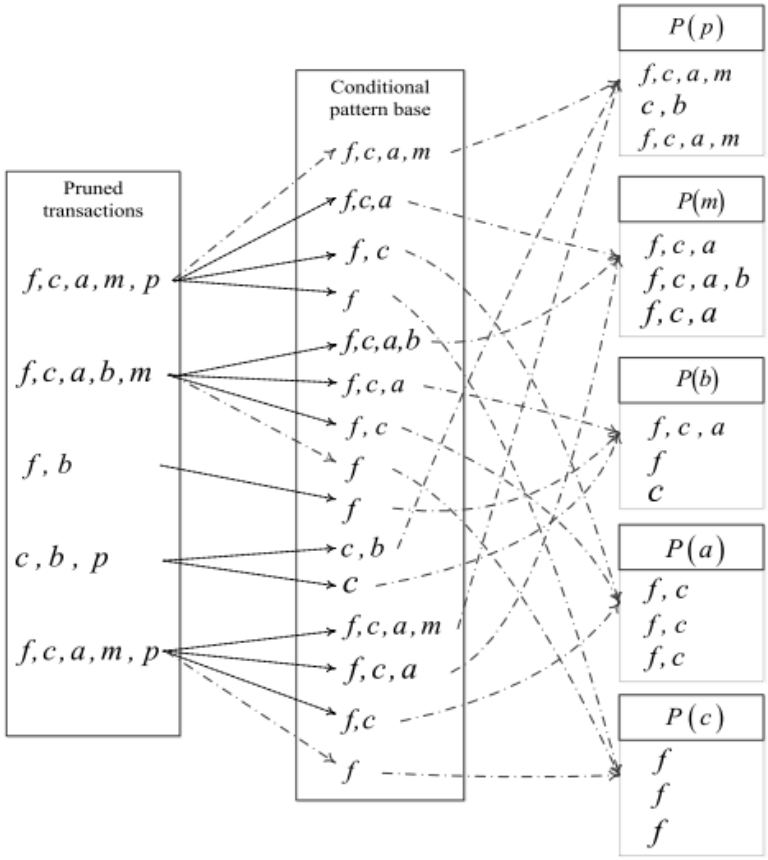


Figura 29 - Fase 2 no algoritmo DPFS.
Xiujin, Shaozong e Hui (2017)

Ao final da fase 2, a lista de padrões condicionais base estarão particionados no *cluster* de acordo com seus respectivos itens frequentes contidos em *1-itemset*. Em seguida, a fase 3 é responsável por minerar os *itemsets* frequentes de forma paralela. Isso é feito construindo uma FP-Tree convencional em cada partição assim como no FP-Growth, que gera as combinações de *itemsets* e elimina os que não atingem o suporte mínimo especificado de forma recursiva.

Experimentos foram realizados em *datasets* com diferentes características, que foram multiplicados de 2 até 9 vezes para identificar desempenho em relação ao volume de dados. Os testes comparam os algoritmos PFP, YAFIM e DFPS, sendo executados em um *cluster* de 24 nodos. A Figura 30 mostra o tempo de execução em segundos em um *dataset* que contém 119 itens e 8.124 transações. Observa-se que quando o número de *itemsets* distintos é baixo, YAFIM e DFPS são quase 5 vezes mais rápidos que PFP.

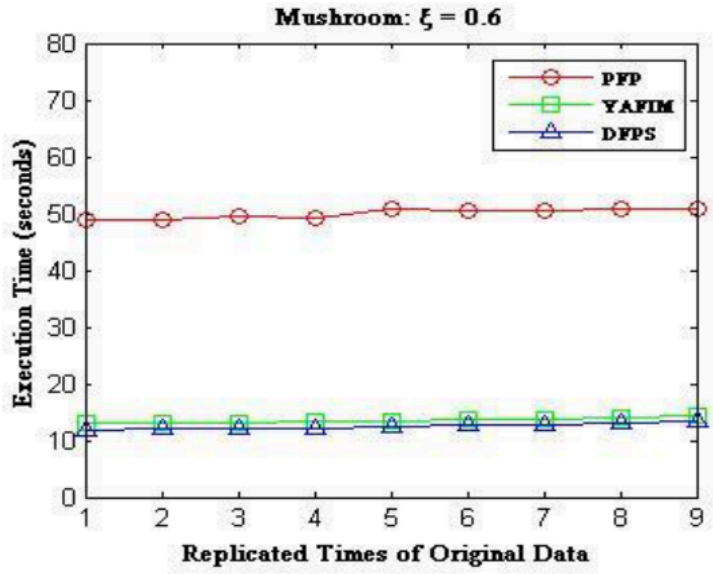


Figura 30 – Tempo de execução *dataset* Mushroom. PFP, YAFIM e DFPS.
Xiujin, Shaozong e Hui (2017)

Por outro lado, quando o mesmo experimento é feito usando um *dataset* com 2,113 itens distintos e 49.046 transações, DPFS chega a ser 18 vezes mais rápido do que YAFIM e 3,5 vezes mais rápido do que PFP. Esse comportamento é ilustrado na Figura 31. Dessa forma, Xiujin, Shaozong e Hui (2017) concluem que algoritmos baseados no Apriori como YAFIM não são otimizados para um grande número de itens distintos, e que implementações FP-Growth no MapReduce tem um overhead de I/O considerável no tempo de computação. Essas características são mitigadas com DFPS, uma implementação distribuída do FP-Growth no Spark.

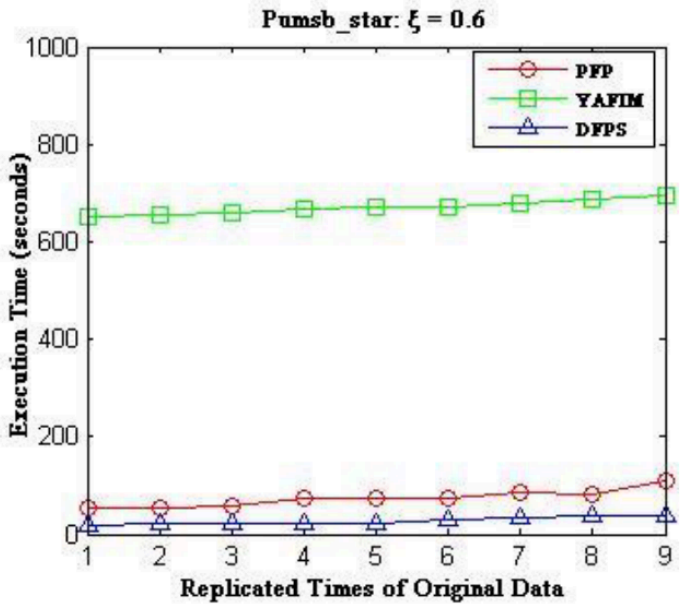


Figura 31 - Tempo de execução *dataset* Pumsb_star. PFP, YAFIM e DFPS.
Xiujin, Shaozong e Hui (2017)

4.3 Considerações finais do capítulo

A etapa de geração de *itemsets* frequentes nos métodos de mineração de regras de associação é crucial e ao mesmo tempo requer muito processamento. Os algoritmos sequenciais tradicionais não escalam com o aumento no volume de dados, e por conta disto, implementações paralelas como MRApriori e PFP foram desenvolvidas com o modelo de programação MapReduce. Com o surgimento do Spark, este tipo de algoritmo, que é iterativo por natureza, foi otimizado para ser processado inteiramente em memória, eliminando o *overhead* de disco encontrado nas implementações anteriores.

R-Apriori é baseado no algoritmo YAFIM e traz otimizações na geração de *candidate itemsets* na segunda iteração, porém, não é ideal para o processamento de um grande número de itens distintos. Mais recentemente, DFPS que é baseado no FP-Growth foi proposto, onde a etapa de geração de *candidate itemsets* foi eliminada por completo, contudo, seu artigo acaba por não trazer uma comparação direta com R-Apriori, mas somente YAFIM.

5 DESENVOLVIMENTO DOS ALGORITMOS

Uma vez que os autores dos algoritmos distribuídos apresentados acima não forneceram o código fonte dos mesmos, estes foram desenvolvidos conforme descritos em cada estudo. Esta implementação própria trouxe um melhor entendimento das estratégias de mineração de *itemset* frequentes propostas, possibilitou a avaliação de desempenho entre os algoritmos YAFIM, R-Apriori e DFPS – tendo que R-Apriori e DFPS não foram diretamente comparados em seus artigos originais –, e também permitiu a validação das conclusões tomadas em cada um dos estudos.

A linguagem de programação escolhida para o desenvolvimento dos algoritmos neste trabalho foi Scala. Scala é uma linguagem multi-paradigma da JVM que suporta programação funcional e orientada a objetos. É muito utilizada no campo de ciência de dados, principalmente quando altos volumes de dados são considerados, uma vez que traz avançado suporte à concorrência (BUGNION, 2016).

Dentre suas qualidades, a principal razão da linguagem Scala ter sido escolhida em relação às outras linguagens suportadas pelo Spark (Java, Python e R), se dá que a própria plataforma Spark foi desenvolvida em Scala. Possibilitando que a partir da obtenção de experiência em Scala, o código fonte da plataforma Spark seja mais facilmente navegado e entendido no caso de eventuais dúvidas durante a codificação. Além disso, as APIs oferecidas para as outras linguagens tendem a não serem tão completas e extensivas quanto à API nativa em Scala (APACHE SOFTWARE FOUNDATION, 2018b).

Inicialmente, uma versão força-bruta do algoritmo Apriori foi implementado com o intuito de ser usado como “fonte da verdade” para validar os resultados dos algoritmos a serem implementados. Dessa forma, testes unitários foram criados para garantir que durante mudanças nos algoritmos, como otimizações e refatorações, o resultado ainda estivesse corretamente alinhado com a “fonte da verdade”. A Figura 32 mostra alguns dos testes unitários iniciais que auxiliaram a implementação dos algoritmos.

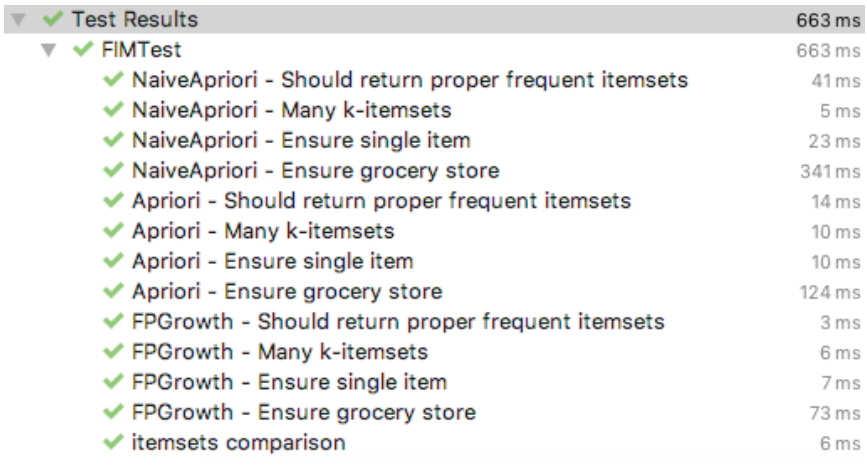


Figura 32 - Testes unitários usados para garantir os resultados corretos.
Fonte: elaborado pelo autor

Os algoritmos sequenciais Apriori e FP-Growth nos quais os três algoritmos distribuídos se baseiam foram então implementados tomando como base o livro *Introduction to Data Mining* escrito por Tan e Steinback (2005).

5.1 Apriori

Em linhas gerais, as seguintes etapas foram implementadas para a construção do algoritmo Apriori:

1. Varredura inicial do *dataset* para encontrar os *itemsets* frequentes de tamanho 1.
2. Geração da lista de *candidate itemsets*, representando as possíveis combinações de tamanho *k* a partir dos *itemsets* frequentes anteriores. Por exemplo, dado *1-itemsets = {a, b, c}* como entrada, a saída esperada é *2-itemsets = {ab, ac, bc}*.
3. Filtragem inicial – que não requer consulta ao *dataset* – durante geração dos candidatos. Por exemplo, dado os *2-itemsets* frequentes *{ab, ac, bc, bd}* e os possíveis *3-itemsets* candidatos gerados no passo anterior *{abc, bcd}*, é possível assumir que o *itemset* gerado *{bcd}* não é frequente uma vez que o seu subconjunto *{cd}* não aparece na lista de *2-itemsets* frequentes. Dessa forma, evitando a contagem do suporte deste *itemset* a partir das transações.
4. Filtragem final, para cada transação do *dataset*, verificar quais candidatos são seus subconjuntos e incrementando um contador caso positivo. Mantendo apenas os candidatos que respeitam o suporte mínimo estipulado, colocando-os assim na lista de *itemsets* frequentes.

5. Incrementar a variável *k* e voltar para o passo 2, interrompendo o ciclo caso nenhum *itemset* frequente novo seja encontrado no passo 4.

O passo 4 nos algoritmos YAFIM e R-Apriori é otimizado pelo uso de uma estrutura de dados chamada *hash tree*, que permite que um menor número de consultas no *dataset* seja feita, caso contrário, todos candidatos gerados seriam validados contra todas transações. Assim, essa otimização foi também desenvolvida e aplicada para o algoritmo sequencial Apriori. Esse novo passo é composto pelas três seguintes etapas: 1. Construir a *hash tree* a partir dos *itemsets* candidatos gerados anteriormente; 2. Para cada transação no *dataset*, percorrer a *hash tree* e encontrar os candidatos que são subconjuntos da transação em questão; 3. Agrupar todos candidatos encontrados e manter apenas os frequentes.

Como pode ser ilustrado na Figura 33 uma *hash tree* é a representação de um número de *itemset* candidatos de um mesmo tamanho *N* espalhados em vários *buckets* nas folhas de uma árvore. Essa árvore tem profundidade máxima igual ao tamanho *N* dos itemsets. Quando um candidato é inserido na árvore, o seu primeiro item (elemento de posição 0) é usado para definir qual direção o candidato irá seguir no próximo nível na árvore, seguindo uma função *hash* elaborada para direcionar cada item distinto. Com o nodo do próximo nível definido, o item na segunda posição do candidato é usado para definir a próxima direção a ser tomada, e assim por diante. Dessa forma, todos candidatos são distribuídos em seus respectivos *buckets* na *hash tree* de acordo com a função *hash* definida.

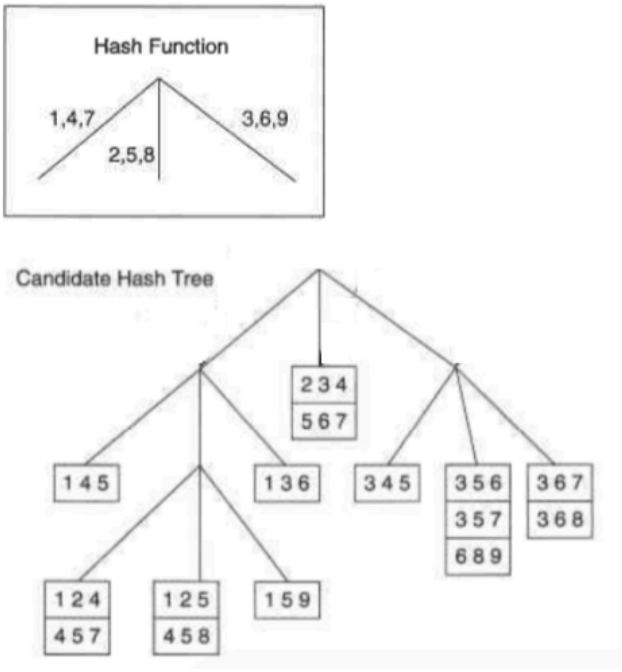


Figura 33 - Hash tree com candidatos de tamanho 3.
Fonte: adaptado de Tan e Steinbach (2005)

Na segunda etapa, a *hash tree* é então percorrida múltiplas vezes para cada transação, encontrando todos candidatos pertencentes à transação em questão. Como uma transação pode ser de tamanho maior do que os candidatos, ela é quebrada em diferentes combinações que percorrem a árvore usando a função *hash* definida para encontrar os *buckets* válidos. Quando um *bucket* é encontrado, identifica-se quais candidatos são subconjuntos da transação pai, adicionando-os na lista de candidatos válidos caso positivo. Quando todas transações terminam de percorrer a árvore, os candidatos são agrupados e contados, aqueles com suporte mínimo aceito são ditos frequentes.

Após finalização da implementação da *hash tree* e aplicação da mesma no algoritmo Apriori, não foi notado uma melhora significativa no tempo de execução. Identificando as partes lentas do código, foi percebido que no máximo 4 *buckets* eram geradas para candidatos de tamanho 2, e cerca de 11.935 candidatos eram distribuídos em apenas 4 folhas. Fazendo com que praticamente todos os candidatos tivessem que ser verificados para cada transação, acabando por não eliminar uma quantidade significativa de verificações e inclusive trazendo um *overhead* adicional de criação e busca na *hash tree*. Esse comportamento é ilustrado na Figura 34.

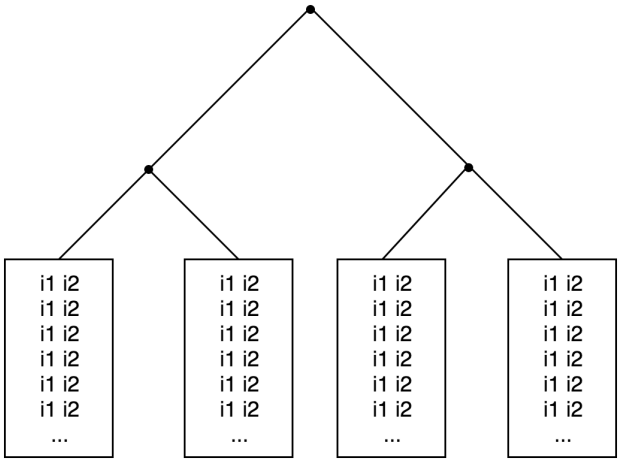


Figura 34 - *Hash tree* contendo apenas 4 *buckets*.
Fonte: elaborado pelo autor

Essa sobrecarga de candidatos se deu pelo uso da mesma função *hash* descrita por Tan e Steinback (2005) no exemplo visto na Figura 33, (*item % size)*, o resto da divisão entre o item em questão e o número de itens nos candidatos a serem armazenados. Conforme testes realizados durante desenvolvimento, foram obtidos resultados satisfatórios aumentando o número máximo de *buckets* pela relação entre o número de itens frequentes distintos da iteração anterior e o número de itens nos candidatos, *(item % (items.size / size))*.

A função *hash* usada originalmente e a nova são ilustradas na Figura 35, considerando uma lista de itens distintos com 14 elementos *{1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ,11, 12, 13, 14}* e um *size* de tamanho 2. Essa alteração possibilitou que candidatos sejam distribuídos em uma maior quantidade de *buckets*, reduzindo o tempo médio de execução do Apriori com o *dataset* T10I4D100K – que é detalhado no capítulo 6 – de 72,88 segundos para 2,99 segundos.

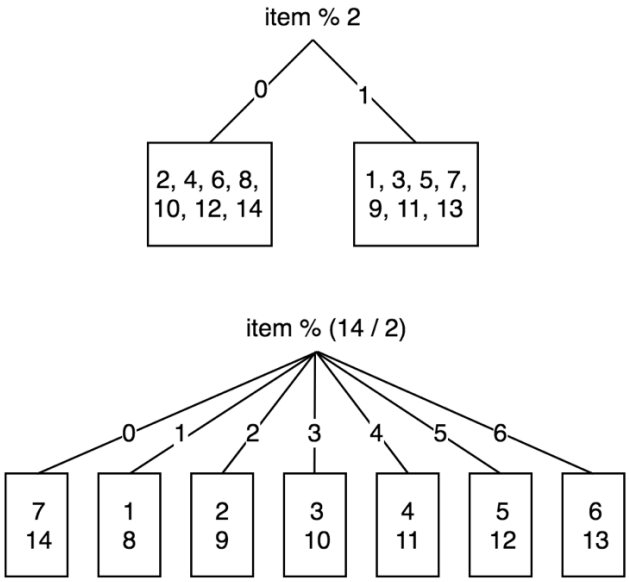


Figura 35 - Função *hash* original e alterada.
Fonte: elaborado pelo autor

5.2 FP-Growth

Para o algoritmo FP-Growth, as seguintes etapas foram desenvolvidas:

1. Construção de uma FP-Tree representando todos os caminhos possíveis de *itemsets* e seus suportes a partir do *dataset*.
2. Verificar se um prefixo *P* é frequente na FP-Tree. Adicionando-o na lista de frequentes.
3. Varrer uma FP-Tree e encontrar todos os caminhos possíveis frequentes que tem *P* como prefixo – chamados de *conditional pattern base*.
4. Construir uma FP-Tree condicional a partir das transações encontradas do passo anterior.
5. Voltar para o passo 2 até que todos os prefixos sejam verificados.

5.3 YAFIM

Até então, os algoritmos Apriori e FP-Growth foram desenvolvidos inteiramente em Scala. Em seguida, para o desenvolvimento do YAFIM, o algoritmo sequencial Apriori foi reescrito usando as APIs do Spark seguindo as recomendações descritas no artigo do YAFIM. As seguintes etapas foram implementadas:

1. Leitura e estruturação do *dataset* em um RDD. Esse RDD é então mantido em memória de forma distribuída nos servidores – cada servidor armazena partições do *dataset*.
2. Varredura inicial do *dataset* para geração dos *singletons* de forma distribuída.
3. Geração dos candidatos *k-itemsets* de forma distribuída.
4. Filtragem dos candidatos não frequentes pela leitura do *dataset* de forma distribuída. Neste passo, a *hash tree* é construída em um único nodo, sendo serializada e enviada para todos os outros nodos pelo uso de *broadcast variables*.

5.4 R-Apriori

O código implementado para o algoritmo YAFIM foi completamente reusável para o desenvolvimento do R-Apriori, que apenas sobrescreve a segunda iteração do algoritmo. Essa iteração é substituída de forma a prevenir a geração de *C(2, I)* candidatos, onde *I* é o número de *singletons* frequentes, sem a possibilidade de uma filtragem inicial. Assim, ao invés de gerar todos os candidatos possíveis, estes são diretamente derivados das transações de forma individual.

Antes que a derivação dos candidatos aconteça, é necessário que a transação tenha todos os itens não frequentes removidos. Por exemplo, dado um conjunto de *singletons* $I = \{a, c, e, f\}$, e uma transação $T' = \{a, b, c, d, f\}$, para que os pares candidatos sejam encontrados, a transação é primeiramente transformada em $\{a, c, f\}$, uma vez que *b* e *d* não são frequentes. Os pares $\{ac, af, cf\}$ são então derivados da transação filtrada. Após todas transações serem computadas, os pares idênticos são agrupados e contados, mantendo apenas os que possuem o suporte mínimo especificado, resultando nos *itemsets* frequentes de tamanho 2.

Para que os itens não frequentes possam ser removidos das transações, é preciso que a lista de *singletons* seja consultada múltiplas vezes. Essas consultas são agilizadas pelo uso de *bloom filters*, uma estrutura de dados probabilística similar a um *hash table*, que permite consultas em tempo constante. Essa estrutura garante que um elemento não está presente, ou

que um elemento provavelmente está presente, essa probabilidade de falso positivos pode ser definida na criação do *bloom filter*. Como os elementos em si não são armazenados, essa estrutura é extremamente compacta e recomendada para grandes *datasets*.

5.5 DFPS

A implementação do DFPS faz forte uso do algoritmo FP-Growth sequencial desenvolvido anteriormente. Conforme visto anteriormente na Figura 24 da seção 4.2.3, a paralelização neste algoritmo é alcançada pela geração dos padrões condicionais base a partir de cada transação, onde cada padrão condicional possui um prefixo de tamanho 1. Estes padrões são então agrupados de acordo com seus respectivos prefixos, cada grupo podendo ser processado de forma isolada em qualquer nodo do cluster. Um *dataset* com 1.000 *singletons* distintos irá gerar 1.000 tarefas que podem ser processadas sem dependências usando o algoritmo FP-Growth tradicional.

Dessa forma, as seguintes etapas foram desenvolvidas para o DFPS:

1. Leitura e estruturação do *dataset* em um RDD (similar ao YAFIM e R-Apriori).
2. Varredura inicial do *dataset* para geração dos *singletons* de forma distribuída (similar ao YAFIM e R-Apriori).
3. Construção dos padrões condicionais base a partir de cada transação.
4. Agrupamento dos padrões base e repartição do RDDs por prefixo. Nessa etapa, os grupos gerados são distribuídos pelos nodos do *cluster*.
5. Mineração dos *itemsets* frequentes de forma individual por partição usando FP-Growth.

Durante testes locais, foi identificado uma alta discrepância entre o tempo de execução apresentado por Xiujin, Shaozong e Hui (2017) e os resultados obtidos durante o desenvolvimento. Em seus experimentos, DFPS processou o *dataset* Mushroom com 60% de suporte em um *cluster* com 6 *nodos* de 4 *cores* em cerca de 11 segundos. Já com o algoritmo próprio sendo executado localmente, o tempo de execução do mesmo *dataset* e suporte usando apenas um *core* foi de 4,23 segundos. Primeiramente, a saída do algoritmo foi validada de acordo com uma ferramenta externa de mineração de dados chamada SPMF (FOURNIER-VIGER, 2019) e os resultados estavam de acordo. Em seguida foi feito um comunicado via e-mail para os autores, que confirmaram um erro no estudo original. O experimento em questão foi realizado em apenas um nodo, e não 6 como mencionado no artigo.

Outro ponto curioso encontrado neste estudo foi o suporte de 60% utilizado junto ao *dataset* T10I4D100K nos experimentos. Durante o desenvolvimento, foi percebido que somente a primeira fase do algoritmo (busca dos *singletons*) era executada neste *dataset* com este suporte, isso ocorreu uma vez que o *dataset* em questão não possui nenhum *itemset* de tamanho 1 que aparece em pelo menos em 60% das transações. Sendo assim, o algoritmo interrompe sua execução na primeira fase e tem uma lista vazia como saída. Contudo, a fase executada neste caso é idêntica para os algoritmos YAFIM e DFPS, não sendo um cenário válido para comparação de desempenho entre eles.

5.6 Considerações finais do capítulo

O desenvolvimento dos algoritmos foi o primeiro passo para que os experimentos fossem realizados. De forma geral, os algoritmos originais foram descritos nos artigos em alto nível, assim, algumas decisões em relação à detalhes de implementação, como por exemplo o número de filhos na *hash tree*, foram tomadas com base em testes realizados durante o desenvolvimento. O código fonte das implementações próprias sequenciais e distribuídas foram disponibilizadas publicamente e podem ser encontradas na plataforma GitHub[2].

Além disto, foram encontradas duas possíveis falhas no artigo do DFPS: A primeira referente ao número de nodos usado no *cluster* mencionado no estudo, que foi confirmada via e-mail pelos autores; e a segunda em relação ao suporte usado em um dos *datasets*, que acaba por não gerar nenhum *itemset* frequente, fazendo com que a comparação de desempenho das fases dos algoritmos acabem por não serem esclarecedoras.

As características gerais dos algoritmos podem ser resumidas conforme o Quadro 3.

Quadro 3 - Características dos algoritmos YAFIM, R-Apriori e DFPS.

	YAFIM	**R-Apriori**	**DFPS**
Base	Apriori	Apriori e YAFIM	FP-Growth
Tipo	Iterativo	Iterativo	Recursivo, dividir e conquistar
Detalhes	Implementação distribuída do Apriori	Otimização do YAFIM na 1ª iteração	Implementação distribuída do FP-Growth
Leituras no *dataset*	k iterações + 1	k iterações + 1	2
Proposto em	2014	2015	2017

Fonte: elaborado pelo autor

[2] Código fonte dos algoritmos: https://github.com/felipekunzler/frequent-itemset-mining-spark

6 LABORATÓRIO DE EXPERIMENTOS

O objetivo geral inicialmente proposto no Anteprojeto foi levemente alterado durante o desenvolvimento do trabalho. Tendo sido originalmente proposto avaliar e comparar o desempenho entre algoritmos sequenciais e suas implementações paralelas no Spark, variando volume dos dados e outros parâmetros. Com o estudo dos trabalhos relacionados na primeira parte do trabalho, foram encontradas diversas propostas de algoritmos distribuídos em Spark para mineração de *itemsets* frequentes, abrindo a possibilidade para uma avaliação de desempenho entre algoritmos distribuídos – tendo que a comparação de dois destes não foi abordada em nenhum dos artigos –, ao invés da comparação entre sequencial e distribuído, trazendo um maior espaço para contribuições. Dessa forma, o objetivo geral e título do trabalho foram alterados para refletir tal mudança.

Para a avaliação de desempenho dos três algoritmos distribuídos implementados, YAFIM, R-Apriori e DFPS, foram usados quatro conjuntos de dados com características distintas que são comumente usados para experimentos de mineração de *itemsets* frequentes. Estes *datasets* foram processados em *clusters* gerenciados pelo serviço Amazon EMR, variando o número de nodos e multiplicando o volume dos dados. Os tempos de execução obtidos para cada algoritmo e configuração foram analisados e comparados entre si.

6.1 Conjunto de dados

Os quatro conjuntos de dados utilizados para realização dos experimentos possuem diferentes características e foram usados originalmente pelos estudos do YAFIM e DFPS. Estes são: Mushroom, um conjunto de dados contendo cogumelos venenosos e comestíveis; T10I4D100K, um conjunto de dados artificial criado pelo gerador de dados da IBM; Chess, que descreve um número de estados de um tabuleiro de xadrez na última jogada entre rei e peão contra rei e torre; e, por fim, Pumsb_star, um censo contendo dados demográficos (DUA; GRAFF, 2017). Mushroom, Chess e Pumsb_star foram originalmente disponibilizados pelo Repositório de *Machine Learning* da UCI[3] e T10I4D100K pela IBM (GOETHALS, 2003).

[3] https://archive.ics.uci.edu/ml/index.php

Tabela 17 - Propriedades dos *datasets* utilizados.

Dataset	Número de itens em cada transação	Número de transações	Tamanho
Mushroom	119	8.124	570 KB
T10I4D100K	870	100.000	4 MB
Chess	75	3.196	342 KB
Pumsb_star	2.113	49.046	11,3 MB

Fonte: Qiu et al. (2014)

Estes *datasets* foram obtidos a partir do repositório de *datasets* para mineração de *itemsets* frequentes (GOETHALS, 2003) e suas propriedades são descritas na Tabela 17.

6.2 Laboratório e experimentos

O serviço Elastic MapReduce (EMR), disponibilizado pela Amazon, foi utilizado para execução dos experimentos. Esse serviço possibilita o processamento e análise de grandes quantidades de dados através do uso de *frameworks* como Apache Hadoop e Apache Spark sem a necessidade do uso de servidores físicos locais (AMAZON EMR, 2019), tendo como base o serviço EC2 (*Elastic Compute Cloud*), que provê instancias de servidores de vários tipos diferentes. Dessa forma, o EMR permite o rápido provisionamento de *clusters* para processamento distribuído, que podem ser criados em menos de 10 minutos a partir de *templates* prontos para Hadoop, Spark, e outros *frameworks*.

Figura 36 - Acesso remoto ao nodo *master*.

Fonte: elaborado pelo autor

Durante a criação de um *cluster*, é necessário informar o *framework* usado, número de nodos e tipo de instância (que definirá o processador e memória RAM). O serviço leva cerca de 10 minutos para finalizar o provisionamento e configuração do *cluster*, que no caso do Spark, também inclui YARN como gerenciador de recursos e HDFS como sistema de arquivos distribuído. Como pode ser visto na Figura 36, é disponibilizado acesso remoto SSH ao nodo *master* ao final do provisionamento, onde é possível interagir com o Spark através do comando *spark-shell* ou submeter *jobs* ao *cluster* usando o comando *spark-submit* pela linha de comando.

Todos experimentos foram executados em máquinas EC2 do tipo m5.xlarge, que possuem dois *cores* e 16gb de memória RAM. Os *datasets* foram armazenados no sistema de arquivos distribuído HDFS dos *clusters*, de onde cada experimento leu o *dataset* a ser processado. O tempo de execução de cada um dos três algoritmos foi avaliado para cada um dos quatro *datasets* em *clusters* contendo 1, 3, 5, 7 e 9 nodos. O volume de dados também foi variado replicando os *datasets* 1, 3, 5, 7 e 9 vezes no *cluster* de 9 nodos.

Por fim, os resultados gerados pelos algoritmos (lista de *itemsets* frequentes) para todos os quatro *datasets* usados foram validados de acordo com uma ferramenta de mineração de dados *open source*, SPMF (FOURNIER-VIGER, 2019). Assim garantindo que a saída obtida está de acordo com os resultados esperados e que os algoritmos estão funcionando propriamente.

A versão do pacote de aplicações EMR utilizada foi 5.23.0 (lançada em 30/04/2019), que contém Spark 2.4.0 (lançado em 02/11/2018) e Hadoop YARN e HDFS 2.8.5 (lançados em 15/09/2018).

6.3 Resultados e discussões

Cada combinação de configuração dos experimentos apresentados a seguir foram executadas 3 vezes para identificar possíveis *outliers* a partir da análise do desvio padrão e do coeficiente de variação. Os tempos de execução analisados nos resultados representam a média destas 3 execuções e desconsideram o tempo de *startup* inicial do Spark.

A Tabela 18 demonstra um exemplo completo da execução para uma única configuração de um dos experimento realizados, neste caso, são 9 nodos e *dataset* replicado 9 vezes. Cada linha demonstra o tempo de processamento das três execuções em segundos (*Run 1, Run 2 e Run 3*), para um algoritmo e *dataset*, a média destes tempos (*Mean*), assim como o desvio padrão (*SD*) e coeficiente de variação (*CV*) das execuções. Em sua grande maioria, as

execuções não tiveram uma variação maior do que 5% e o pior caso sendo de 12%, estes níveis foram considerados aceitáveis neste contexto, não tendo um impacto significativo na análise dos resultados.

Tabela 18 – Detalhes da execução com 9 nodos e *dataset* replicado 9 vezes.

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	10,09	10,28	9,92	10,10	0,15	1,49%
RApriori	mushroom	9,47	10,19	10,58	10,08	0,46	4,56%
DFPS	mushroom	2,75	2,57	2,74	2,69	0,08	2,97%
YAFIMHashTree	pumsb_star	9,37	9,97	10,06	9,80	0,31	3,16%
RApriori	pumsb_star	8,80	8,97	9,54	9,10	0,32	3,52%
DFPS	pumsb_star	8,64	8,77	9,35	8,92	0,31	3,48%
YAFIMHashTree	chess	13,96	13,67	12,98	13,54	0,41	3,03%
RApriori	chess	11,96	12,22	12,68	12,28	0,30	2,44%
DFPS	chess	1,73	1,84	1,82	1,80	0,05	2,78%
YAFIMHashTree	T10I4D100K	27,58	27,16	26,11	26,95	0,62	2,30%
RApriori	T10I4D100K	17,40	17,16	17,58	17,38	0,17	0,98%
DFPS	T10I4D100K	9,73	9,70	10,40	9,94	0,32	3,22%

Fonte: elaborado pelo autor

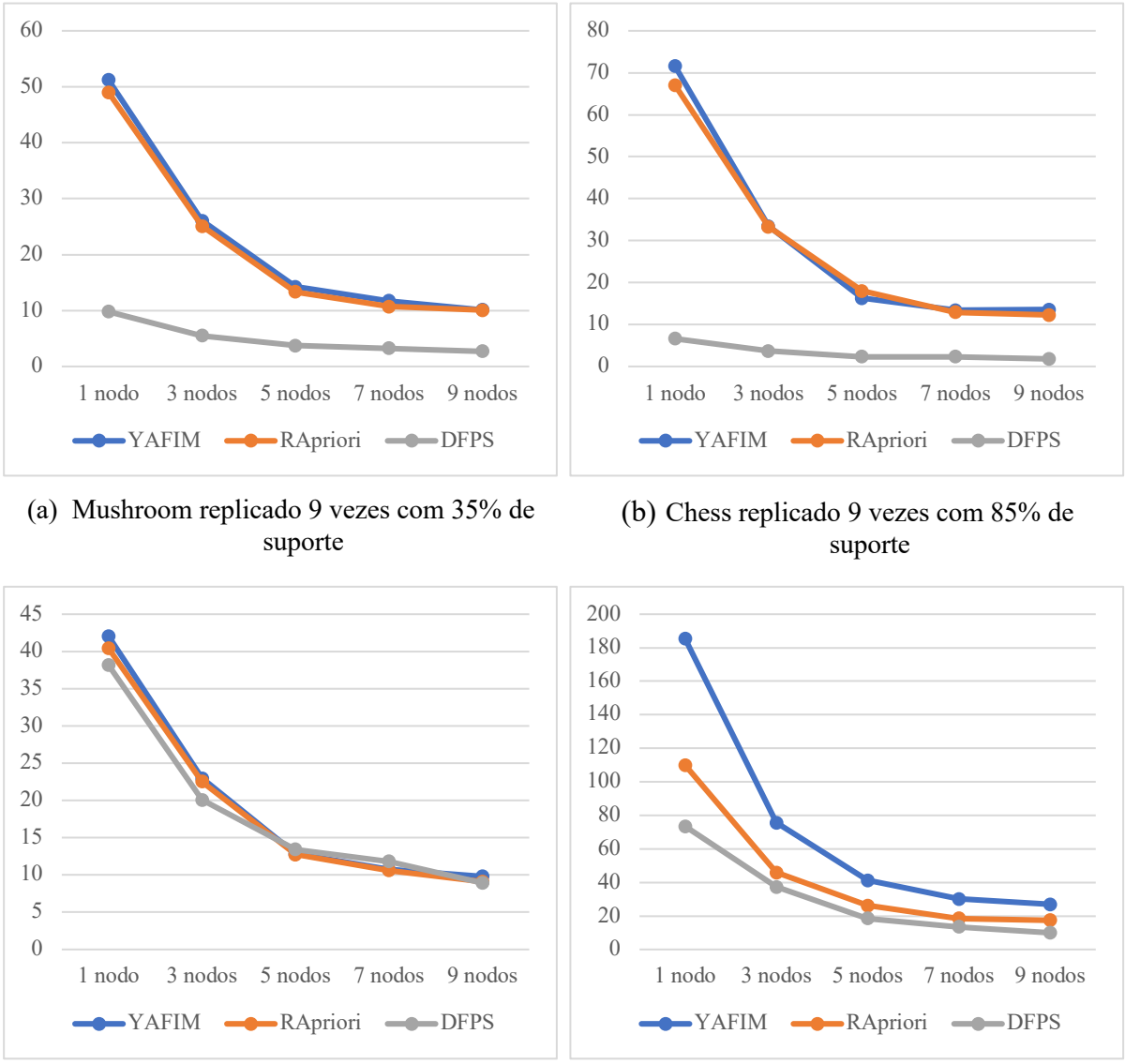
Os experimentos nos subcapítulos a seguir apresentam os tempos médios das execuções obtidas. Uma posição detalhada de todas execuções (assim como exibido na Tabela 18) está disponível no apêndice A.

6.3.1 Variação de nodos no *cluster*

Em um primeiro experimento, os *datasets* foram replicados em 9 vezes e os nodos foram variados em 1, 3, 5, 7 e 9 vezes, -- configurações similares às vistas nos artigos dos algoritmos em questão – permitindo identificar a performance de cada algoritmo e *dataset* assim como a escalabilidade quanto ao aumento no número de nodos. O suporte mínimo utilizado foi de acordo com o estudo do algoritmo YAFIM, 35%, 85%, 65% e 0,25% para os *datasets* Mushroom, Chess, Pumsb_star e T10I4D100K respectivamente. Os tempos de execução deste experimento são apresentados em segundos no eixo Y da Figura 37, e a variação de nodos no eixo X.

Como pode ser observado na Figura 37 (a), em relação ao *dataset* Mushroom, o desempenho entre YAFIM e R-Apriori são muito similares neste caso. Isso se dá devido ao baixo número de *singletons* frequentes encontrados (24), não impactando negativamente a estratégia de geração de *itemsets* do YAFIM. Além disso, DFPS é significantemente mais rápido em todas variações do *cluster*, mas acaba por não escalar de forma proporcional com o

aumento de nodos neste *dataset*. O mesmo comportamento descrito acima também é visto com o *dataset* Chess na Figura 37 (b).



(a) Mushroom replicado 9 vezes com 35% de suporte

(b) Chess replicado 9 vezes com 85% de suporte

(c) Pumsb_star replicado 9 vezes com 65% de suporte

(d) T10I4D100K replicado 9 vezes com 0,25% de suporte

Figura 37 - Gráficos contendo a variação de nodos no *cluster*. Tempo em segundos no eixo Y e variação de nodos no eixo X.

Fonte: elaborado pelo autor

Já em relação ao *dataset* Pumsb_star, encontrado na Figura 37 (c), os três algoritmos tiveram um desempenho similar em todas as diferentes configurações de *cluster*. Esse comportamento ocorreu uma vez que o número de *singletons* frequentes é baixo (12) assim como no *dataset* Mushroom, fazendo com que YAFIM e R-Apriori desempenhassem de forma parecida. Além disto, esse *dataset* possui um número relativamente grande de transações e um número de *itemsets* frequentes pequenos em todas as iterações, fazendo com que YAFIM e R-

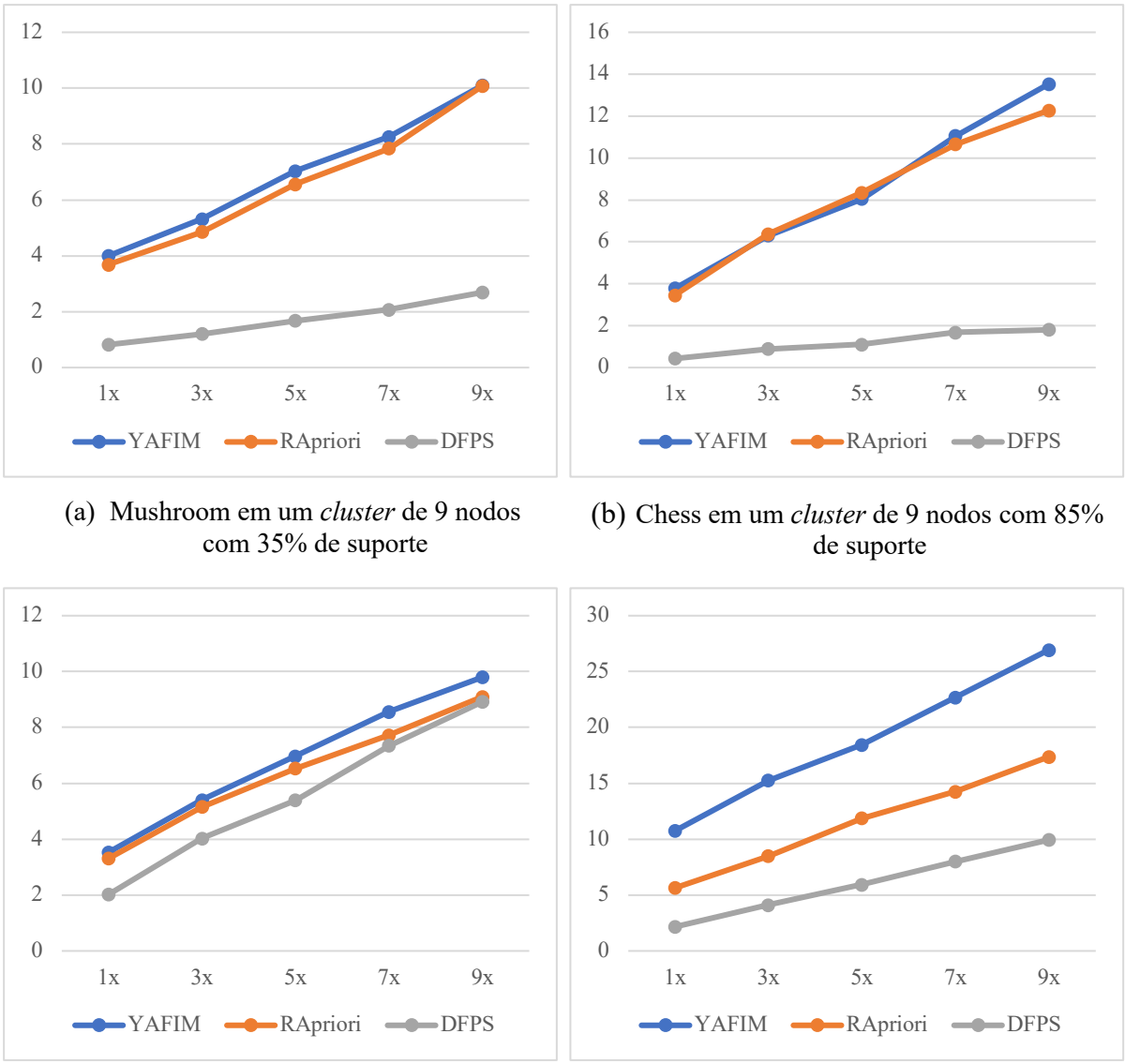
Apriori tenham tempos de execução similares ao DFPS, uma vez que geram uma pequena lista de candidatos e não tem a performance penalizada.

O quarto e último *dataset* T10I4D100K, exibido na Figura 37 (d), possui um alto número de *singletons* frequentes (717), e consequentemente reflete um melhor desempenho do R-Apriori em relação ao YAFIM devido à otimização no encontro de *itemsets* frequentes de tamanho 2. O desempenho do DFPS é melhor do que esses algoritmos neste *dataset*, que escalaram de forma proporcional com o aumento de nodos no *cluster*.

6.3.2 Replicação do *dataset*

Em um segundo experimento, o número de nodos no cluster foi fixado em 9, e a replicação dos *datasets* foi variada em 1, 3, 5, 7 e 9 vezes, permitindo identificar como os algoritmos são impactados com o aumento no volume de dados. O suportes mínimos utilizados são os mesmos do experimento anterior. O tempo de execução é apresentado em segundos no eixo Y da Figura 38, e a variação da replicação do *dataset* no eixo X.

O tempo de execução para todos os *datasets* neste experimento ficou inversamente proporcional aos tempos de execução visto no experimento anterior. O aumento no volume de dados impactou o desempenho negativamente na mesma proporção em que o aumento de nodos no *cluster* contribuiu para um melhor desempenho. Os algoritmos YAFIM e R-Apriori apresentaram tempos de execução similares nos *datasets* Mushroom, Chess e pumsb_star. Além disto, todos os algoritmos tiveram um aumento praticamente linear no tempo de execução de acordo com a replicação de dados no *dataset* T10I4D100K, onde também foi visto um melhor desempenho do R-Apriori em relação ao YAFIM com o aumento do volume de dados.



(a) Mushroom em um *cluster* de 9 nodos com 35% de suporte

(b) Chess em um *cluster* de 9 nodos com 85% de suporte

(c) Pumsb_star em um *cluster* de 9 nodos com 65% de suporte

(d) T10I4D100K em um *cluster* de 9 nodos com 0,25% de suporte

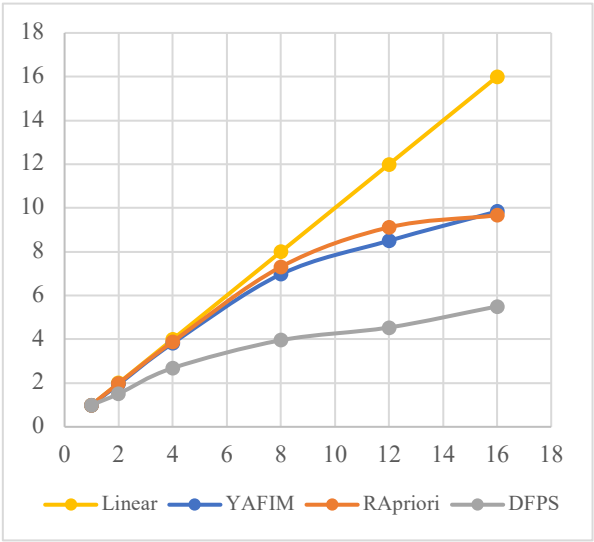
Figura 38 - Gráficos contendo a variação da replicação do *dataset*. Tempo em segundos no eixo Y e variação da replicação no eixo X.

Fonte: elaborado pelo autor

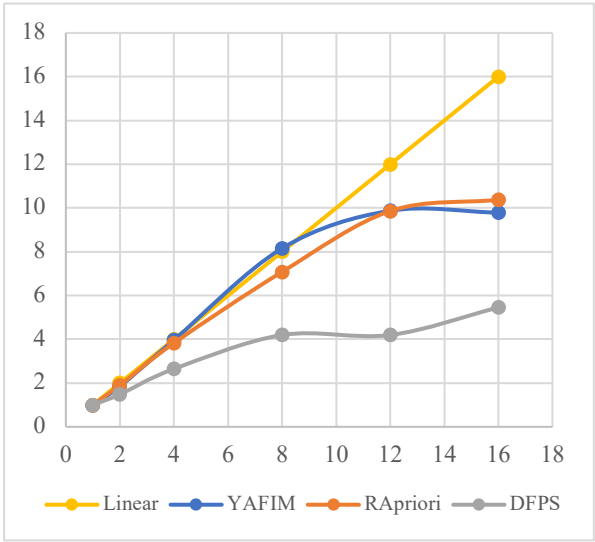
6.3.3 Fator de escalabilidade

No terceiro e último experimento, o fator de escalabilidade dos algoritmos, foi avaliado de acordo com o aumento de *cores* no *cluster* e tendo a replicação dos *datasets* fixada em 9. As mesmas execuções feitas no primeiro experimento foram reusadas para analisar a eficiência atribuída ao incremento de nodos no *cluster*. Com isto em mente, os tempos de execução foram normalizados para possibilitar a comparação da escalabilidade entre os algoritmos e *datasets*.

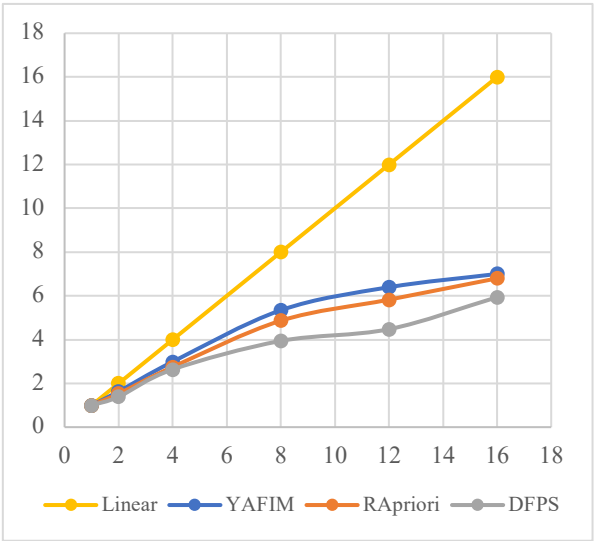
Como em *clusters* EMR um dos nodos é usado como *máster*, este por sua vez acaba por não processar as tarefas em si. Por conta disto, este nodo foi desconsiderado neste experimento. O tempo de execução dos algoritmos distribuídos em um único *core* foi tomado como base para gerar uma linha de escalabilidade linear ideal, onde caso o número de *cores* dobre, o tempo de execução é reduzido pela metade. O eixo Y na Figura 39 representa esse fator de escalabilidade, que é obtido dividindo o tempo de execução em 1 *core* pelo tempo de execução em *N cores*. E o eixo X representa o número total de *cores* executores.



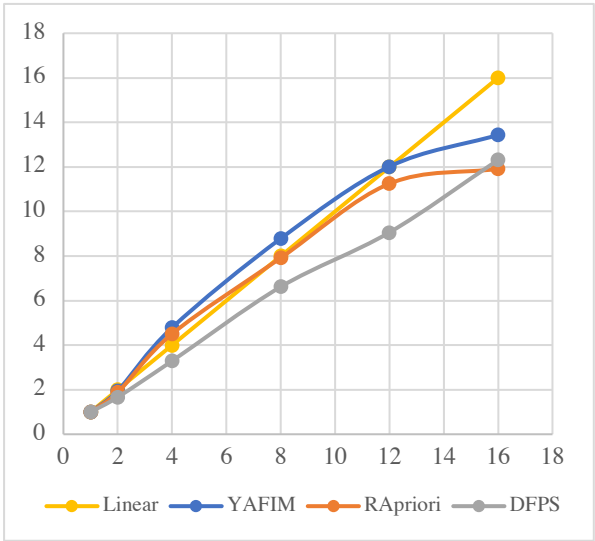
(a) Mushroom replicado 9 vezes com 35% de suporte



(b) Chess replicado 9 vezes com 85% de suporte



(c) Pumsb_star replicado 9 vezes com 65% de suporte



(d) T10I4D100K replicado 9 vezes com 0,25% de suporte

Figura 39 – Gráficos contendo o fator de escalabilidade. Fator de escalabilidade no eixo Y e número total de *cores* no cluster no eixo X.

Fonte: elaborado pelo autor

Na Figura 39 (c) pode ser observado que a escalabilidade do algoritmo DFPS com o *dataset* pumsb_star foi similar em comparação aos *datasets* Mushroom e Chess. Contudo, os algoritmos YAFIM e R-Apriori tiveram uma pior escalabilidade neste *dataset*, indo de uma melhora de 10 vezes com 16 nodos para 6,2 vezes. Por fim, T10I4D100K foi o *dataset* que obteve a melhor escalabilidade para todos os algoritmos, melhorando o desempenho proporcionalmente com o aumento de *cores* em todas as configurações testadas.

6.4 Considerações finais do capítulo

Os quatro *datasets* vistos nos estudos do YAFIM e DFPS foram usados na elaboração dos três experimentos realizados: variação de nodos no *cluster*, variação da replicação do *dataset* e fator de escalabilidade normalizada dos algoritmos. Estes experimentos foram feitos em um ambiente *cloud* Amazon EMR.

Os resultados encontrados ficaram de acordo com os resultados obtidos nos estudos originais, foi observado que o algoritmo distribuído DFPS tem melhor desempenho de forma geral em relação ao YAFIM e R-Apriori, apesar de não apresentar o melhor fator de escalabilidade e ter desempenho similar quando poucos *itemsets* frequentes são encontrados em cada uma das iterações. Por fim, R-Apriori apresentou um desempenho superior ao YAFIM quando o número de *singletons* frequentes encontrado na fase 1 é relativamente alto, uma vez que a etapa de geração de *itemsets* candidatos é eliminada. Em outros casos onde a quantidade de *singletons* não é considerável o desempenho foi similar.

CONCLUSÃO

Com o constante aumento no volume de dados disponíveis para processamento, tarefas como a mineração de *itemsets* frequentes acabam por se tornar inviáveis com algoritmos sequenciais tradicionais como Apriori e FP-Growth. Diversos estudos foram feitos na paralelização deste tipo de algoritmo usando o modelo de programação MapReduce proposto pela Google, contudo, resultados intermediários ou *datasets* precisam ser lidos do disco a cada iteração.

Mais recentemente, com o advento do *framework* Spark, algoritmos de propriedade iterativa vêm sendo desenvolvidos usando o processamento em memória oferecido pelo Apache Spark. YAFIM, R-Apriori e DFPS são implementações distribuídas para mineração de *itemsets* frequentes no Spark. DFPS é baseado no FP-Growth e pode ser considerado um dos mais performáticos atualmente. Seu estudo traz comparações com YAFIM, que é baseado no Apriori e justifica o ganho de performance pela eliminação das etapas de geração de *candidate itemsets* em todas iteração. Contudo, R-Apriori traz melhorias no processo de geração de *candidate itemsets* do YAFIM, mas acabou por não ter sido comparado diretamente com o DFPS em seu artigo.

Dessa forma, por se tratarem de pesquisas relativamente recentes, que lidam com uma tecnologia emergente, neste trabalho se deu a continuação pelo desenvolvimento destes algoritmos, YAFIM, R-Apriori e DFPS. Cada algoritmo teve seu desempenho avaliado e comparado, validando os resultados encontrados nas suas pesquisas originais.

Este desenvolvimento foi o primeiro passo para que os experimentos fossem realizados. De forma geral, os algoritmos originais foram descritos nos artigos em alto nível, assim, algumas decisões em relação à detalhes de implementação, como por exemplo o número de filhos na *hash tree*, foram tomadas com base em testes realizados durante o desenvolvimento. O código fonte das implementações próprias sequenciais e distribuídas foram disponibilizadas publicamente e podem ser encontradas na plataforma GitHub[4].

Foram encontradas duas possíveis falhas no artigo que explora o algoritmo DFPS: A primeira referente ao número de nodos usado no *cluster* mencionado no estudo, que foi confirmada via e-mail pelos autores; e a segunda em relação ao suporte usado em um dos

[4] Código fonte dos algoritmos: https://github.com/felipekunzler/frequent-itemset-mining-spark

datasets, que acaba por não gerar nenhum *itemset* frequente, fazendo que as fases distintas dos algoritmos em comparação acabem por não serem executadas.

Os quatro *datasets* vistos nos estudos do YAFIM e DFPS foram usados na elaboração dos três experimentos realizados: variação de nodos no *cluster*, variação da replicação do *dataset* e fator de escalabilidade normalizada dos algoritmos. Estes experimentos foram feitos em um ambiente *cloud* Amazon EMR.

Os resultados encontrados ficaram de acordo com os resultados obtidos nos estudos originais, foi observado que o algoritmo distribuído DFPS tem melhor desempenho de forma geral em relação ao YAFIM e R-Apriori. Apesar de não apresentar o melhor fator de escalabilidade e ter desempenho similar quando poucos *itemsets* frequentes são encontrados em cada uma das iterações. Por fim, R-Apriori apresentou um desempenho superior ao YAFIM quando o número de *singletons* frequentes encontrados na fase 1 é relativamente alto, uma vez que a etapa de geração de *itemsets* candidatos é eliminada. Em outros casos onde a quantidade de *singletons* não é considerável o desempenho foi similar.

Este trabalho pode ser explorado futuramente pelo desenvolvimento e comparação dos algoritmos estudados em diferentes plataformas como Hadoop e TensorFlow. Também levando em consideração fatores como o uso de memória RAM e latência de rede. Adicionalmente, existe espaço para um estudo mais aprofundado sobre as implicações da paralelização ou distribuição de algoritmos sequenciais, como as principais técnicas e metodologias.

REFERÊNCIAS BIBLIOGRÁFICAS

AMAZON EMR. **Amazon EMR - Elastic MapReduce**. Disponível em: <https://aws.amazon.com/emr/>. Acesso em: 14 maio. 2019.

APACHE SOFTWARE FOUNDATION. **Apache Hadoop Project**. Disponível em: <http://hadoop.apache.org/>. Acesso em: 18 ago. 2018a.

APACHE SOFTWARE FOUNDATION. **Apache Spark Project**. Disponível em: <http://spark.apache.org/>. Acesso em: 23 set. 2018b.

BUGNION, P. **Scala for Data Science**. [s.l.] Packt Publishing, 2016.

CHAMBERS, B.; ZAHARIA, M. **Spark : The Definitive Guide**. [s.l.] O'Reilly Media, 2018.

CHRAIBI, C. From Moore ' s Law to Amdahl ' s Law : The Pursuit of Computer Performance. v. 6, n. 5, p. 4412–4416, 2015.

DE MAURO, A.; GRECO, M.; GRIMALDI, M. What is Big Data? A Consensual Definition and a Review of Key Research Topics. **AIP Procedings**, v. 1644, n. SEPTEMBER, p. 97–104, 2014.

DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. **Proceedings of 6th Symposium on Operating Systems Design and Implementation**, p. 137–149, 2004.

DIAZ, J.; MUÑOZ-CARO, C.; NIÑO, A. A survey of parallel programming models and tools in the multi and many-core era. **IEEE Transactions on Parallel and Distributed Systems**, v. 23, n. 8, p. 1369–1386, 2012.

DUA, D.; GRAFF, C. **UCI Machine Learning Repository**, 2017. Disponível em: <https://archive.ics.uci.edu/ml/index.php>. Acesso em: 30 mar. 2019.

FOURNIER-VIGER, P. **SPMF - An Open-Source Data Mining Library**, 2019. Disponível em: <http://www.philippe-fournier-viger.com/spmf/index.php>

GOETHALS, B. **Frequent Itemset Mining Dataset Repository,** 2003. Disponível em: <http://fimi.uantwerpen.be/data/>. Acesso em: 30 mar. 2019.

HANDY, A. **Open Source Leaders: Matei Zaharia, Apache Spark**. Disponível em: <https://thenewstack.io/matei-zaharia-qa/>. Acesso em: 23 set. 2018.

JINGJING; SOMASHEKAR, A. M. **Programming Models for Distributed Computing**. Disponível em: <http://dist-prog-book.com/chapter/8/big-data.html#data-parallelism>. Acesso em: 7 out. 2018.

KATAL, A.; WAZID, M.; GOUDAR, R. H. Big data: Issues, challenges, tools and Good practices. **2013 6th International Conference on Contemporary Computing, IC3 2013**, p. 404–409, 2013.

LAI, J. Implementations of iterative algorithms in Hadoop and Spark. 2014.

LI, H. et al. **Pfp: Parallel Fp-growth for Query Recommendation**. Proceedings of the 2008 ACM Conference on Recommender Systems. **Anais**...: RecSys '08.New York, NY, USA: ACM, 2008

OPENMP ARCHITECTURE BOARD. **OpenMP Application Programming Interface**. Disponível em: <https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf>. Acesso em: 15 set. 2018.

PACHECO, P. S. **Introduction to Parallel Programming**. [s.l: s.n.].

PATEL, A. B.; BIRLA, M.; NAIR, U. Addressing big data problem using Hadoop and Map Reduce. **3rd Nirma University International Conference on Engineering, NUiCONE 2012**, p. 6–8, 2012.

QIU, H. et al. YAFIM: A parallel frequent itemset mining algorithm with spark. **Proceedings - IEEE 28th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2014**, p. 1664–1671, 2014.

RATHEE, S.; KAUL, M.; KASHYAP, A. R-Apriori: An Efficient Apriori based Algorithm on Spark. **In Proceedings of the 8th Workshop on Ph. D. Workshop in Information and Knowledge Management , ACM**, p. 27–34, 2015.

SAMADI, Y.; ZBAKH, M.; TADONKI, C. Comparative study between Hadoop and Spark based on Hibench benchmarks. **Proceedings of 2016 International Conference on Cloud Computing Technologies and Applications, CloudTech 2016**, p. 267–275, 2017.

SU, X.; PATTNAIK. Introduction to Big Data Analysis. **Ntnu**, p. 1–20, 2016.

TAN, I. N. G.; STEINBACH, M. **Introduction to Data Mining**. 1st. ed. [s.l.] Pearson, 2005.

TORQUATI, M. et al. Smart multicore embedded systems. **Smart Multicore Embedded Systems**, v. 9781461488, p. 1–175, 2014.

WHITE, T. **Hadoop: The Definitive Guide**. 4th. ed. [s.l.] O'Reilly Media, 2015.

XIUJIN, S.; SHAOZONG, C.; HUI, Y. DFPS : Distributed FP-growth Algorithm Based on. **IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)**, p. 1725–1731, 2017.

ZAHARIA, M. et al. Spark : Cluster Computing with Working Sets. **HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing**, p. 10, 2010.

ZAHARIA, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. **Nsdi**, p. 2–2, 2012.

APÊNDICE A. POSIÇÃO DETALHADA DOS EXPERIMENTOS

Uma posição detalhada de todas execuções utilizadas neste trabalho está disponível neste apêndice. Cada tabela representa uma configuração de *cluster* e fator de replicação de *dataset*. Cada linha representa o tempo de processamento das três execuções em segundos (Run 1, Run 2 e Run 3), para um algoritmo e *dataset*, a média destes tempos (Mean), assim como o desvio padrão (SD) e coeficiente de variação (CV) das execuções.

1 nodo (usando apenas 1 *core*) e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	99,53	99,49	99,20	99,41	0,15	0,15%
RApriori	mushroom	96,87	96,98	98,46	97,44	0,73	0,75%
DFPS	mushroom	15,60	14,54	14,25	14,80	0,58	3,92%
YAFIMHashTree	pumsb_star	68,22	69,01	68,60	68,61	0,32	0,47%
RApriori	pumsb_star	61,85	61,90	61,88	61,88	0,02	0,03%
DFPS	pumsb_star	52,96	53,74	52,22	52,97	0,62	1,17%
YAFIMHashTree	chess	130,32	131,82	135,39	132,51	2,13	1,61%
RApriori	chess	128,56	128,82	124,87	127,42	1,80	1,41%
DFPS	chess	9,87	10,04	9,55	9,82	0,20	2,04%
YAFIMHashTree	T10I4D100K	363,59	362,38	360,23	362,07	1,39	0,38%
RApriori	T10I4D100K	207,68	208,40	205,14	207,07	1,40	0,68%
DFPS	T10I4D100K	122,99	122,59	121,64	122,41	0,56	0,46%

1 nodo (usando todos 2 *cores*) e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	51,00	50,84	51,74	51,19	0,39	0,76%
RApriori	mushroom	48,55	49,48	48,91	48,98	0,38	0,78%
DFPS	mushroom	10,85	9,49	9,03	9,79	0,77	7,87%
YAFIMHashTree	pumsb_star	39,57	42,85	43,71	42,04	1,79	4,26%
RApriori	pumsb_star	38,51	41,32	41,42	40,42	1,35	3,34%
DFPS	pumsb_star	37,40	38,30	38,77	38,16	0,57	1,49%
YAFIMHashTree	chess	72,70	69,60	72,41	71,57	1,40	1,96%
RApriori	chess	67,30	66,89	66,94	67,04	0,18	0,27%
DFPS	chess	6,52	6,20	7,11	6,61	0,37	5,60%
YAFIMHashTree	T10I4D100K	188,00	189,03	178,93	185,32	4,54	2,45%
RApriori	T10I4D100K	109,14	120,31	100,30	109,92	8,19	7,45%
DFPS	T10I4D100K	86,21	67,28	66,88	73,46	9,02	12,28%

3 nodos e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	27,00	25,86	25,22	26,03	0,74	2,84%
RApriori	mushroom	25,67	24,94	24,64	25,08	0,44	1,75%
DFPS	mushroom	6,05	5,24	5,25	5,51	0,38	6,90%
YAFIMHashTree	pumsb_star	22,74	22,67	23,48	22,96	0,37	1,61%
RApriori	pumsb_star	21,22	22,45	23,86	22,51	1,08	4,80%
DFPS	pumsb_star	20,06	20,36	19,77	20,07	0,24	1,20%
YAFIMHashTree	chess	33,93	33,03	33,27	33,41	0,38	1,14%
RApriori	chess	33,32	33,51	32,99	33,27	0,21	0,63%
DFPS	chess	3,82	3,64	3,66	3,71	0,08	2,16%
YAFIMHashTree	T10I4D100K	74,74	78,03	74,18	75,65	1,70	2,25%
RApriori	T10I4D100K	46,12	45,80	45,96	45,96	0,13	0,28%
DFPS	T10I4D100K	37,63	36,59	37,37	37,20	0,44	1,18%

5 nodos e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	15,59	13,79	13,40	14,26	0,95	6,66%
RApriori	mushroom	13,80	13,24	12,94	13,33	0,36	2,70%
DFPS	mushroom	3,74	3,79	3,71	3,74	0,03	0,80%
YAFIMHashTree	pumsb_star	12,28	12,51	13,72	12,84	0,63	4,91%
RApriori	pumsb_star	11,49	12,92	13,74	12,72	0,93	7,31%
DFPS	pumsb_star	14,44	12,61	13,28	13,44	0,75	5,58%
YAFIMHashTree	chess	15,53	15,15	18,03	16,24	1,28	7,88%
RApriori	chess	18,13	17,84	17,99	17,99	0,12	0,67%
DFPS	chess	2,39	2,31	2,34	2,34	0,03	1,28%
YAFIMHashTree	T10I4D100K	44,52	38,81	40,26	41,20	2,42	5,87%
RApriori	T10I4D100K	24,66	26,43	27,40	26,16	1,13	4,32%
DFPS	T10I4D100K	19,14	17,90	18,44	18,49	0,50	2,70%

7 nodos e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	13,31	10,87	10,90	11,69	1,14	9,75%
RApriori	mushroom	11,27	10,49	10,32	10,69	0,42	3,93%
DFPS	mushroom	3,43	3,54	2,83	3,26	0,31	9,51%
YAFIMHashTree	pumsb_star	10,59	10,83	10,76	10,73	0,10	0,93%
RApriori	pumsb_star	9,93	10,93	11,00	10,62	0,49	4,61%
DFPS	pumsb_star	12,01	11,57	11,90	11,83	0,19	1,61%
YAFIMHashTree	chess	13,51	13,58	13,18	13,42	0,17	1,27%
RApriori	chess	13,18	12,83	12,74	12,92	0,19	1,47%
DFPS	chess	2,22	2,47	2,32	2,34	0,10	4,27%
YAFIMHashTree	T10I4D100K	31,15	29,77	29,47	30,13	0,73	2,42%
RApriori	T10I4D100K	18,54	18,89	17,81	18,41	0,45	2,44%
DFPS	T10I4D100K	14,07	13,49	12,98	13,51	0,44	3,26%

9 nodos e *dataset* replicado 9 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	10,09	10,28	9,92	10,10	0,15	1,49%
RApriori	mushroom	9,47	10,19	10,58	10,08	0,46	4,56%
DFPS	mushroom	2,75	2,57	2,74	2,69	0,08	2,97%
YAFIMHashTree	pumsb_star	9,37	9,97	10,06	9,80	0,31	3,16%
RApriori	pumsb_star	8,80	8,97	9,54	9,10	0,32	3,52%
DFPS	pumsb_star	8,64	8,77	9,35	8,92	0,31	3,48%
YAFIMHashTree	chess	13,96	13,67	12,98	13,54	0,41	3,03%
RApriori	chess	11,96	12,22	12,68	12,28	0,30	2,44%
DFPS	chess	1,73	1,84	1,82	1,80	0,05	2,78%
YAFIMHashTree	T10I4D100K	27,58	27,16	26,11	26,95	0,62	2,30%
RApriori	T10I4D100K	17,40	17,16	17,58	17,38	0,17	0,98%
DFPS	T10I4D100K	9,73	9,70	10,40	9,94	0,32	3,22%

9 nodos e *dataset* replicado 7 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	8,20	8,29	8,29	8,26	0,05	0,61%
RApriori	mushroom	8,05	7,44	8,03	7,84	0,28	3,57%
DFPS	mushroom	2,13	2,16	1,97	2,08	0,08	3,85%
YAFIMHashTree	pumsb_star	8,47	8,59	8,61	8,56	0,06	0,70%
RApriori	pumsb_star	7,54	8,19	7,47	7,73	0,32	4,14%
DFPS	pumsb_star	7,12	7,46	7,47	7,35	0,17	2,31%
YAFIMHashTree	chess	11,22	10,94	11,03	11,06	0,11	0,99%
RApriori	chess	10,23	10,48	11,27	10,66	0,44	4,13%
DFPS	chess	1,74	1,62	1,69	1,68	0,05	2,98%
YAFIMHashTree	T10I4D100K	23,02	22,55	22,54	22,70	0,23	1,01%
RApriori	T10I4D100K	14,49	14,61	13,73	14,28	0,39	2,73%
DFPS	T10I4D100K	7,92	7,96	8,11	8,00	0,08	1,00%

9 nodos e *dataset* replicado 5 vezes:

Class	Dataset	Run 1	Run 2	Run 3	Mean	SD	CV
YAFIMHashTree	mushroom	7,17	6,95	6,99	7,04	0,10	1,42%
RApriori	mushroom	6,75	6,18	6,77	6,56	0,27	4,12%
DFPS	mushroom	1,72	1,65	1,66	1,68	0,03	1,79%
YAFIMHashTree	pumsb_star	7,06	6,78	7,06	6,97	0,13	1,87%
RApriori	pumsb_star	6,60	6,76	6,23	6,53	0,22	3,37%
DFPS	pumsb_star	5,46	5,36	5,40	5,40	0,04	0,74%
YAFIMHashTree	chess	7,97	8,63	7,55	8,05	0,45	5,59%
RApriori	chess	8,77	7,90	8,37	8,35	0,36	4,31%
DFPS	chess	1,14	1,10	1,11	1,11	0,02	1,80%
YAFIMHashTree	T10I4D100K	19,01	18,01	18,36	18,46	0,42	2,28%
RApriori	T10I4D100K	12,04	12,26	11,34	11,88	0,39	3,28%
DFPS	T10I4D100K	5,78	6,17	5,88	5,94	0,17	2,86%

9 nodos e *dataset* replicado 3 vezes:

YAFIMHashTree	mushroom	5,26	5,43	5,28	5,32	0,08	1,50%
RApriori	mushroom	4,91	5,03	4,63	4,86	0,17	3,50%
DFPS	mushroom	1,29	1,18	1,14	1,21	0,06	4,96%
YAFIMHashTree	pumsb_star	5,47	5,35	5,42	5,41	0,05	0,92%
RApriori	pumsb_star	5,08	5,31	5,11	5,16	0,10	1,94%
DFPS	pumsb_star	3,98	4,02	4,09	4,03	0,05	1,24%
YAFIMHashTree	chess	6,74	6,10	6,06	6,30	0,31	4,92%
RApriori	chess	6,19	5,72	7,19	6,37	0,61	9,58%
DFPS	chess	0,86	0,88	0,93	0,89	0,03	3,37%
YAFIMHashTree	T10I4D100K	15,55	14,81	15,45	15,27	0,33	2,16%
RApriori	T10I4D100K	8,26	9,12	8,11	8,50	0,45	5,29%
DFPS	T10I4D100K	4,07	4,13	4,15	4,12	0,03	0,73%