EDUARDO PETRINI SILVA CASTRO

IMPLEMENTAÇÃO DE ALGORITMOS DE REGRAS DE ASSOCIAÇÃO NOS ARCABOUÇOS HADOOP-MAPREDUCE E SPARK

LAVRAS - MG

2016

EDUARDO PETRINI SILVA CASTRO

IMPLEMENTAÇÃO DE ALGORITMOS DE REGRAS DE ASSOCIAÇÃO NOS ARCABOUÇOS HADOOP-MAPREDUCE E SPARK

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de Dados e Engenharia de Software, para a obtenção do título de Mestre.

Orientador

Dr. Denilson Alves Pereira

LAVRAS - MG

2016

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Castro, Eduardo Petrini Silva.
Implementação de Algoritmos de Regras de Associação nos Arcabouços Hadoop-MapReduce e Spark / Eduardo Petrini Silva Castro. – Lavras : UFLA, 2016.
158 p. : il.

Dissertação (mestrado acadêmico)–Universidade Federal de Lavras, 2016.
Orientador(a): Denilson Alves Pereira.
Bibliografia.

1. Regras de Associação. 2. Resolução de Entidades. 3. Hadoop. 4. MapReduce. 5. Spark. I. Universidade Federal de Lavras. II. Título.

EDUARDO PETRINI SILVA CASTRO

IMPLEMENTAÇÃO DE ALGORITMOS DE REGRAS DE ASSOCIAÇÃO NOS ARCABOUÇOS HADOOP-MAPREDUCE E SPARK

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de Dados e Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 15 de Setembro de 2016.

Dr. Ahmed A. A. Esmin UFLA

Dr. Murilo Coelho Naldi UFV

Dr. Denilson Alves Pereira

Orientador

LAVRAS - MG

2016

Dedico esta conquista a minha amada namorada e companheira Ana Paula que me apoiou integralmente durante toda esta jornada. Aos meus pais, Júlio e Adriane, que sempre acreditaram em mim e me ajudaram em mais esta conquista.

AGRADECIMENTOS

Agradeço à minha amada namorada e companheira Ana Paula por todo o seu apoio, força , conselhos e ajuda durante todo o meu percurso. Aos meus pais, Júlio e Adriane, que sempre me apoiaram e acreditam em mim. Aos meus familiares que de alguma forma contribuíram para a minha jornada.

Agradeço ao meu orientador, professor Denilson Alves Pereira, pelos seus ensinamentos, orientações, dedicação, conselhos, paciência e plena disponibilidade. Aos demais professores do DCC pelos ensinamentos compartilhados, em especial aos professores Ahmed e Marluce.

Agradeço à Universidade Federal de Lavras, ao Departamento de Ciência da Computação e ao Programa de Pós-Graduação em Ciência da Computação pela estrutura oferecida e pela oportunidade de realização do mestrado. Agradeço ao Centro de Computação Científica e Aplicada (C3A) da UFLA pelo uso do cluster.

Agradeço pelo apoio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (Capes) e da FAPEMIG, por meio do projeto CEX-APQ-01834-14.

Agradeço aos colegas de laboratório e a todos que de alguma forma fizeram parte desta minha jornada.

RESUMO

Em meio ao grande volume de dados produzidos constantemente em sistemas de informação computadorizados, há algoritmos de mineração de dados capazes de encontrar informações ocultas nesses dados. Uma das técnicas implementadas por esses algoritmos é conhecida como regras de associação, a qual visa encontrar relações entre itens de um mesmo conjunto de dados. Uma proposta recente utiliza regras de associação para tratar o problema de classificação de ofertas de produtos em lojas de vendas *online*. Porém, para grandes volumes de dados, o tempo de execução do algoritmo proposto se torna problemático, dificultando seu uso. Existem *frameworks* que possibilitam a implementação de algoritmos distribuídos em cluster de computadores, como o Hadoop e Spark. Muitos algoritmos de mineração de dados, como o algoritmo *Apriori*, que gera regras de associação, tiveram diversas propostas de implementações utilizando o modelo MapReduce. Este trabalho realizou um estudo das soluções propostas de implementações do algoritmo *Apriori* para o Hadoop-MapReduce. Os algoritmos também foram implementados no Spark e foi feito um comparativo entre as implementações de ambos *frameworks*. Os resultados mostram que as implementações no Spark superam as implementações no Hadoop-MapReduce na maioria das experimentos. Porém, não houve uma implementação única que se sobressaia em todas as situações avaliadas. Também foi implementada no Hadoop-MapReduce e Spark uma alternativa para o problema de classificação de ofertas de produtos de lojas de vendas *online* de modo a permitir o processamento de grandes volumes de dados em tempo hábil. Os resultados mostram elevada capacidade das adaptações em processar volume de dados maiores.

Palavras-chave: Regras de Associação. Resolução de Entidades. Hadoop. MapReduce. Spark. Classificação de ofertas de produtos.

ABSTRACT

In midst to the big amount of data constantly produced on computerized information systems, there are data mining algorithms able to find hidden information in this data. One of techniques implemented by this algorithms is known as association rules, which aims to find associations between items on same dataset. A recent proposal uses association rules to deal with product offer classification in online store. However, for big amount of data, the proposal algorithm runtime becomes unfeasible. There are frameworks enabling distributed algorithms implementation in computer cluster like Hadoop and Spark. Many data mining algorithms, such as *Apriori* Algorithm for association rules, has several implementation proposals using MapReduce. This work performed a study of proposed solutions of Apriori implementation on Hadoop-MapReduce. The algorithms was also adapted to Spark and a comparative was performed between frameworks. The results show that Spark implementations overcomes Hadoop-MapReduce implementations at runtime in most experiments. However, there is no single implementation that is the best in all the evaluated situations. An alternative to the product offer classification in online store problem on Hadoop-MapReduce and Spark was also carried out. The results show large capacity of adaptation to process big amount of data.

Keywords: Association rules. Entity Resolution. Hadoop. MapReduce. Spark. Classification of product offers.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em meio ao grande volume de dados produzidos constantemente em sistemas de informação computadorizados, há o interesse em extrair informações relevantes para diversos fins. Esse processo de extração muitas vezes envolve técnicas de mineração de dados, que é uma área da Ciência da Computação capaz de manipular dados de diversos modos. Mineração de dados possui diversos algoritmos para encontrar padrões previamente desconhecidos, regras de correlação entre dados, séries temporais, predição e agrupamento de dados. Essas informações podem ser úteis para, por exemplo, um processo de tomada de decisão, identificação de oportunidades de negócio, direcionamento de vendas, identificação de hábitos de usuários, detecção de problemas, dentre outros aspectos relevantes. Porém, mesmo os algoritmos mais sofisticados podem não apresentar o desempenho esperado ao processar enormes volumes de dados.

Com o advento da Internet e das tecnologias online, com a facilidade de adquirir um computador ou dispositivo capaz de se conectar à rede mundial de computadores, a quantidade de dados gerados nos sistemas *Web* chegam à casa dos *Terabytes* diários, o que torna inviável o tempo de execução de muitos dos algoritmos para descoberta de padrões ocultos. Porém, muito se tem feito para melhorar o desempenho desses algoritmos. Técnicas de paralelização e distribuição de dados e processamento estão sendo aplicadas em algoritmos de mineração de dados, visando diminuir o tempo gasto para o processamento.

1.1 Contextualização

Na última década, emergiu um modelo de programação denominado MapReduce (DEAN; GHEMAWAT, 2004). O MapReduce visa distribuir o processamento de dados para vários computadores organizados em um cluster, onde cada um processa uma porção dos dados para o qual foi atribuído. O MapReduce é constituído basicamente de duas funções: Map e Reduce. A função Map processa cada porção de dados de maneira individual e concorrente, enquanto que a função Reduce opera sobre as saídas das funções Map para produzir o resultado final. O Hadoop (APACHE..., 2015a) é um *framework* gratuito e de código aberto que implementa o MapReduce e já consolidado quando o assunto é soluções alternativas para otimizar o tempo de execução perante ao aumento da quantidade de dados de entrada. Neste documento, será usado o termo Hadoop-MapReduce para se referir à implementação do MapReduce no framework Hadoop. Em alguns casos o Hadoop-MapReduce tende a apresentar problemas de desempenho devido à necessidade de persistir em disco os dados que trafegam entre as funções Map e Reduce e entre iterações de algoritmos.

Mais recentemente, um novo framework, denominado Spark (ZAHARIA et al., 2010), foi desenvolvido para atender às exigências de maior desempenho para processamento iterativo e fluxo de dados em tempo real. É capaz de processar grandes volumes de dados em memória sem a necessidade e persisti-los em disco entre as funções Map e Reduce ou entre cada iteração dos algoritmos. O Spark também possui uma biblioteca denominada MLlib (APACHE..., 2016a), a qual implementa alguns algoritmos de mineração de dados de forma distribuída. Há na literatura muitas propostas de implementação de técnicas de mineração de dados utilizando o

Hadoop-MapReduce e Spark, as quais serão apresentadas no decorrer deste texto.

Existem diversas técnicas de mineração de dados para realizar busca de informações interessantes em conjuntos de dados, dentre as principais estão regra de associação e classificação. Regra de associação é um método usado para encontrar relações entre itens em uma base de dados (AGRAWAL; IMIELINSKI; SWAMI, 1993). Inicialmente, essa técnica foi projetada para bancos de dados de transações de supermercados, onde os itens são os produtos e as transações são um conjunto de itens adquiridos em uma mesma compra.

Já na classificação, a estratégia é atribuir rótulos a entidades desconhecidas, mas que possuem um conjunto de características que ajudam a identificar a que rótulo ou classe tal entidade pertence. Primeiro, gera-se um modelo baseando-se em dados de treinamento (dados que já possuem a classe conhecida), o qual pode ser em forma de regras, árvore de decisão, fórmulas matemáticas, redes neurais, entre outros. Depois, esse modelo é usado para predizer a classe de novos dados desconhecidos (SEBASTIANI, 2002).

As técnicas de mineração de dados podem ser utilizadas independentemente ou de forma combinada. Recentemente, um algoritmo utilizando uma combinação das técnicas de regras de associação e classificação foi proposto para o problema de classificação de ofertas de produtos em sistemas de comércio eletrônico (OLIVEIRA; PEREIRA, 2014; OLIVEIRA; PEREIRA, 2017). Tal abordagem é uma especificação do problema de resolução de entidades, o qual consiste em um processo de identificar quais registros representam a mesma entidade do mundo real (BILENKO; MOONEY, 2003). A

ideia foi desenvolver um método para encontrar um conjunto de palavras-chave nas descrições textuais de cada oferta de produtos que o diferencie dos demais. O algoritmo utiliza as descrições das ofertas de produtos para treinar um classificador, cujo modelo gerado é constituído de regras de associação. Tal modelo é utilizado na fase de testes para predizer as classes das novas instâncias de ofertas de produtos. A estratégia apresentou bons resultados, o que torna sua utilização viável para o seu contexto. Porém, quando a quantidade de dados de entrada aumenta, o tempo de execução do algoritmo também aumenta consideravelmente, o que motiva a busca por uma solução distribuída para o algoritmo.

1.2 Justificativa e proposta deste trabalho

Processar grandes volumes de dados em tempo hábil é um problema recorrente no cenário tecnológico, pois além de limpar, organizar e armazenar, também é preciso extrair informações relevantes do conjunto de dados por meio de técnicas de mineração de dados (classificação ou regras de associação, por exemplo) para serem utilizadas em seus devidos propósitos. O grande desafio é utilizar essas técnicas em um grande volume de dados e obter resultados em tempo aceitável, uma vez que muitas dessas técnicas foram projetas há anos atrás para se trabalhar com um pequeno volume de dados, executando de maneira sequencial.

Há na literatura diversas propostas que utilizam o MapReduce como paradigma para implementar algoritmos de regras de associação em busca de melhor desempenho perante a grandes volumes de dados no Hadoop (FARZANYAR; CERCONE, 2013a; FARZANYAR; CERCONE, 2013b; LI et al., 2012; LI; ZHANG, 2011; LIN; LEE; HSUEH, 2012; YAHYA; HEGAZY;

EZAT, 2012; YANG; LIU; FU, 2010; ZHOU; HUANG, 2014) e Spark (QIU et al., 2014; RATHEE; KAUL; KASHYAP, 2015). Essas implementações focam no processo de geração dos conjuntos de itens frequentes do clássico algoritmo Apriori (AGRAWAL; IMIELINSKI; SWAMI, 1993). Tal processo demanda bastante recursos computacionais, já que necessita consultar toda a base de dados e efetuar diversas combinações entre os itens.

Esses algoritmos foram projetados para propósitos específicos, possuindo diversas limitações para um contexto de maior amplitude. Para compreender melhor tais limitações, foi efetuado um estudo comparativo minucioso dessas implementações sob diversos aspectos, dentre eles, quantidade de itens distintos na base de dados, quantidade de transações, quantidade de itens por transação e quantidade de memória RAM necessária.

Comparar tais abordagens é essencial para identificar em quais circunstâncias um algoritmo é preferível a outro. Ao processar uma base de dados com o algoritmo adequado, é possível obter maiores ganhos em termos de tempo de execução, contribuindo para a eficiência de diversas aplicações que necessitam identificar relações de associatividade entre elementos de um conjunto de dados.

Neste trabalho, foram comparadas implementações do algoritmo Apriori para o Hadoop-MapReduce e proposta uma nova abordagem. Também, essas implementações foram adaptadas para o Spark. Essas abordagens são baseadas na implementação clássica do Algoritmo Apriori para a geração de *itemsets* frequentes. Embora existam diferentes propostas, nenhuma delas fez uma avaliação rigorosa de sua performance, por exemplo, comparando-a com outras implementações sob diversas características e bases de dados. Devido às diferentes estratégias de implementação adotadas por cada abor-

dagem, a hipótese é que essas implementações poderiam ter comportamentos diferentes para diferentes bases de dados e suporte mínimo requerido. Assim, deseja-se responder questões de pesquisa tais como: como esses algoritmos se comportam para diferentes valores de suporte mínimo? Como eles se comportam quando varia-se a quantidade de transações da base de dados? E quando varia-se a quantidade de itens por transação? E quando varia-se a quantidade de itens distintos? E quando varia-se a quantidade de máquinas no cluster? Esses algoritmos são escaláveis? As implementações do Hadoop-MapReduce podem ser adaptadas para o Spark? E como se comportam no Spark? A partir do estudos desses algoritmos foi possível formular um framework para recomendar a melhor implementação para cada configuração, com base nas características da base de dados e do suporte mínimo exigido.

Neste trabalho, também foi avaliado o algoritmo de resolução de entidades, o qual usa uma abordagem diferente do Apriori para gerar regras de associação. Foram propostas duas alternativas para o algoritmo de classificação de ofertas de produtos implementadas no Hadoop-MapReduce e Spark. A hipótese é que essas implementações distribuídas possam processar grandes volumes de dados em tempo hábil garantindo a qualidade da classificação. Portanto, deseja-se responder questões de pesquisas tais como: as propostas distribuídas são capazes de processar base de dados maiores de forma eficiente? Em quais situações o algoritmo sequencial é preferível em relação aos algoritmos distribuídos? Em quais situações é preferível o algoritmo no Hadoop-MapReduce do que no Spark? Em quais situações é preferível o algoritmo no Spark do que no Hadoop-MapReduce? As implementações distribuídas também foram comparadas tanto em tempo de

execução quanto na qualidade da classificação com alguns *baselines* disponibilizados pela biblioteca *MLlib* do Spark.

1.3 Objetivos gerais e específicos

Um dos objetivos foi comparar as implementações do Algoritmo Apriori para o Hadoop-MapReduce e adaptá-las para o Spark. Além disso, foi proposto e implementado um mapeamento do algoritmo de classificação de ofertas de produtos (OLIVEIRA; PEREIRA, 2014; OLIVEIRA; PEREIRA, 2017) para o Hadoop-MapReduce e para o Spark, com a finalidade de tornar seu tempo de execução viável perante a grandes volumes de dados.

Para atingir os objetivos gerais, foram definidos os seguintes objetivos específicos:

1. Configurar um *cluster* de computadores com o Hadoop e Spark para realizar os experimentos;

2. Avaliar e implementar propostas do algoritmo *Apriori* para o Hadoop-MapReduce;

3. Adaptar as implementações originais do algoritmo Apriori do Hadoop-MapReduce para o Spark;

4. Avaliar os resultados dos experimentos com as implementações do *Apriori* sob diversas situações e fazer um estudo comparativo entre as propostas;

5. Propor um framework para seleção do algoritmo mais adequado baseado nas características da base de dados e do suporte mínimo exigido;

6. Propor um mapeamento do algoritmo de classificação de ofertas de produtos para o Hadoop-MapReduce e Spark;

7. Avaliar o desempenho do algoritmo de classificação de ofertas de produtos perante a grande volumes de dados, comparando-o com a sua versão sequencial e com *baselines* disponibilizados na biblioteca *MLlib* do Spark.

1.4 Contribuições do trabalho

Uma das contribuições deste trabalho foi avaliar e comparar diversas propostas distribuídas do algoritmo de regras de associação Apriori e identificar em quais circunstâncias determinado algoritmo é preferível a outros. Com base nos resultados, foi proposto um framework para recomendação do algoritmo mais adequado de acordo com as características da base de dados, valor do suporte mínimo e quantidade de memória principal disponível. Além disso, foram propostas alternativas distribuídas para o algoritmo de resolução de entidades, as quais demonstraram ser mais eficazes e eficientes do que os classificadores da biblioteca *MLlib* do Spark.

1.5 Tipo de pesquisa

Esta pesquisa pode ser classificada como quantitativa em relação à abordagem, pois busca por meio da objetividade e variáveis mensuráveis identificar para qual situação determinado algoritmo é mais adequado. Quanto à natureza, esta pesquisa pode ser classificada como aplicada, pois gera conhecimentos para a aplicação prática e soluções para problemas específicos. Em relação aos objetivos, é uma pesquisa exploratória, pois foi realizado levantamento bibliográfico, implementação e experimentos para maior

familiarização com os problemas abordados. Quanto ao procedimento, é uma pesquisa experimental, pois, a partir das hipóteses levantadas, foram realizados diversos experimentos com diferentes variáveis a fim de identificar o comportamento dos algoritmos.

1.6 Estrutura do documento

O restante deste documento está organizado da seguinte forma: no Capítulo 2, é apresentado um referencial teórico sobre regras de associação, classificação, o paradigma MapReduce, Hadoop, Spark, resolução de entidades e apresentadas as implementações do Apriori para o Hadoop-MapReduce. No Capítulo 3, são apresentadas as adaptações dos algoritmos do Apriori para o Spark. No Capítulo 4, são descritas as adaptações do algoritmo para classificação de oferta de produtos para o Hadoop-MapReduce e Spark. No Capítulo 5, é apresentada a metodologia para avaliação experimental e os experimentos das implementações do algoritmo Apriori para o Hadoop-MapReduce e Spark, bem como, as comparações e proposta de framework para escolha da implementação mais adequada para cada situação. No Capítulo 6, é apresentada a metodologia para avaliação experimental e os experimentos com as adaptações para o Hadoop-MapReduce e Spark do algoritmo para classificação de oferta de produtos. No Capítulo 7, são apresentadas as conclusões e trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo, estão descritos os conceitos sobre Hadoop, MapReduce, Spark, mineração de dados, regras de associação, classificação de dados e trabalhos relacionados.

2.1 MapReduce

Com o constante aumento do volume de dados gerados a cada instante na rede mundial de computadores, diversos pesquisadores buscaram soluções para o problema de processamento e armazenamento de grandes volumes de dados de maneira eficiente. Nesse contexto emergiu um novo modelo de programação denominado MapReduce (DEAN; GHEMAWAT, 2004), inspirado nas primitivas *map* e *reduce* do *Lisp* e outras linguagens funcionais. Este modelo de programação foi desenvolvido para permitir ao programador implementar aplicações que possam ser executadas em uma infraestrutura de cluster. Dessa forma, o processamento de dados é distribuído para vários computadores, onde cada um processa a porção de dados que lhe foi atribuída, omitindo do usuário toda a complexidade de paralelização, comunicação e gerenciamento de carga.

O modelo trabalha com dados no formato <chave,valor>, tanto para os dados de entrada quanto para os de saída. O programador necessita desenvolver apenas duas funções específicas: Map e Reduce. A função Map, executada para cada porção de dados atribuída, recebe um conjunto de pares <chave,valor> e produz outro conjunto <chave,valor> intermediário. O arcabouço agrupa todos os valores intermediários associados com as mesmas chaves e os passa para a função Reduce.

A função Reduce recebe as chaves intermediárias e uma lista com todos os valores associados a cada chave que são processados de acordo com a implementação específica do usuário. A saída é também no formato <chave,valor> e ao finalizar, os dados são armazenados no sistema de arquivos.

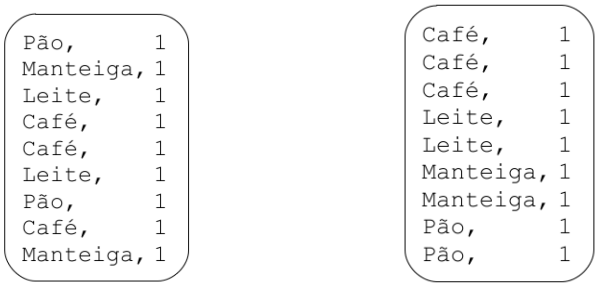
Muitas técnicas de mineração de dados também possuem implementações propostas no paradigma MapReduce, tais como, regras de associação (FARZANYAR; CERCONE, 2013a; LIN; LEE; HSUEH, 2012; YANG; LIU; FU, 2010) e classificação (WU et al., 2009; HE et al., 2010). Tais implementações alcançam melhorias consideráveis em termos de tempo de execução quando comparadas com as implementações tradicionais perante a grande volumes de dados.

A seguir, é apresentada uma visão geral do funcionamento do MapReduce implementando o algoritmo de teste chamado "WordCount" (HADOOP. . . , 2015).

O algoritmo WordCount, proposto para fins de teste, é a demonstração do funcionamento básico do MapReduce, e tem como objetivo contar a ocorrência de cada palavra em um arquivo de texto qualquer. A Figura 1 ilustra um conjunto típico de entrada para essa aplicação, que são 3 linhas, cada uma contendo 3 palavras, armazenadas em um arquivo.

Figura 1 Exemplo do arquivo de entrada.

Inicialmente a entrada <chave,valor> do algoritmo é um identificador da linha e o conteúdo são as palavras da linha em questão, em forma



(a) Saída da função Map.

(b) Chaves agrupadas lexicograficamente.

Figura 2 Etapa de ordenação.

de uma *String*. Essa é a entrada para a primeira função do MapReduce, a função Map. Nessa função, o objetivo é processar cada linha do arquivo para extrair todas as palavras do texto e, para cada palavra, adicionar um contador com o valor 1. Esse par de dados "<palavra,contador(1)>" é gerado para cada palavra do texto (Figura 2a). Terminada a função Map, um passo adicional é executado para efetuar o agrupamento das chaves por meio de uma ordenação lexicográfica. A Figura 2b apresenta o resultado da função Map com as chaves já agrupadas.

Com as chaves agrupadas, o MapReduce cria uma lista de valores para cada chave distinta, gerando um novo conjunto de <chave,valor> (Figura 3) que será a entrada para a função Reduce. Essa lista é apresentada na forma de um *Iterator* para que não sobrecarregue a memória em caso de uma quantidade muito grande de valores.

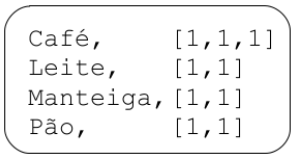


Figura 3 Chaves agrupadas e a lista de valores.

A função Reduce é, então, chamada com o objetivo de contar quantas vezes uma determinada chave apareceu. Para isso, basta contar quantos elementos a lista de valores de cada chave possui. A saída (Figura 4) será todas as ocorrências de palavras distintas encontradas no texto e a quantidade de vezes que elas apareceram.

```
Café,     3
Leite,    2
Manteiga, 2
Pão,      2
```

Figura 4 Saída da função Reduce.

A Figura 5 ilustra o fluxo de execução da aplicação WordCount no MapReduce apresentando as principais etapas, usando 3 instâncias da função Map e 4 instâncias da função Reduce. Observa-se que há 4 etapas principais para produzir o resultado final. O arquivo texto de entrada é primeiramente dividido em blocos que são espalhados pelos computadores do cluster. Em seguida, cada instância da função Map efetua o processamento do segmento de dados de acordo com a implementação feita pelo usuário. Na etapa de *Shuffling*, ocorre o agrupamento e ordenação das chaves, criando-se uma lista de valores para cada chave distinta. Os pares <chave,lista(valor)> são distribuídos para cada instância da função Reduce. Nessa última função, implementada pelo usuário, ocorre efetivamente o processo de contagem das palavras. Os dados processados são enviados para o arquivo de saída, o qual fica espalhado pelo cluster.

Existem diversas razões para a ascensão do MapReduce, dentre elas, destaca-se a facilidade de uso, os programadores não precisam ter experiên-

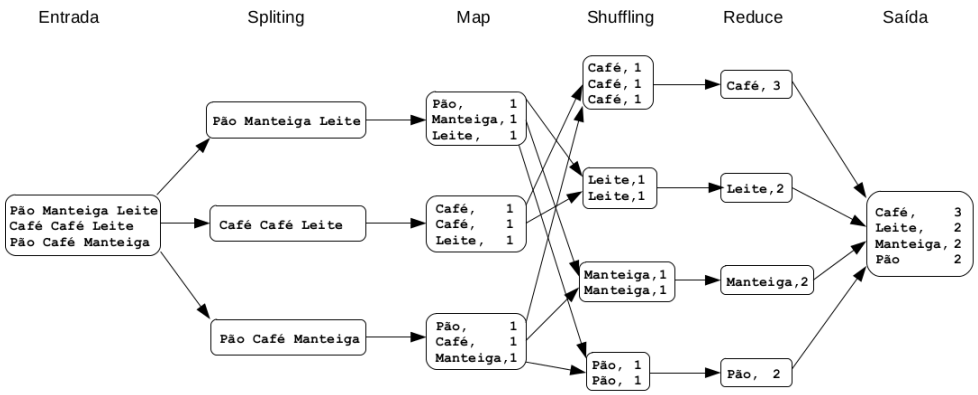


Figura 5 Etapas do MapReduce, adaptado de Groningen (2009).

cia com sistemas paralelos e distribuídos, já que toda a questão de detalhes da paralelização são invisíveis ao usuário (DEAN; GHEMAWAT, 2004).

2.2 Hadoop

O MapReduce é uma implementação de alto nível na hierarquia de uma aplicação distribuída. É necessária uma plataforma que fornece toda a estrutura necessária para sua execução, tais como, gerência de comunicação, distribuição de dados e processamento. O Hadoop (WHITE, 2012) é uma plataforma que implementa o MapReduce. Trata-se de um *framework* gratuito e de código aberto para o armazenamento e processamento distribuído de dados, que permite a execução de aplicações implementadas no modelo de programação MapReduce. Também serve como plataforma para diversos outros frameworks e ferramentas, inclusive o Spark, que usufruem de seus recursos. O Hadoop usa o Hadoop Distribuited File System (HDFS) (WHITE, 2012, p. 45), um sistema de arquivo baseado no Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003), para gerenciar os objetos de dados em um cluster de computadores. O HDFS é um sistema de arquivo

primário para armazenamento distribuído de dados usado pelas aplicações do Hadoop. É adequado para armazenamento e processamento distribuído usando computadores comuns, chamados de *commodity* hardware. As principais características que diferem o HDFS dos demais sistemas de arquivos distribuídos são sua tolerância a falhas, escalabilidade e simplicidade para expandir a capacidade de armazenamento.

O Hadoop possui uma arquitetura específica para gerenciamento e processamento de aplicações em cluster, denominada *YARN* (Yet Another Resource Negotiator) (VAVILAPALLI et al., 2013). Trata-se de um conjunto de serviços para a alocação de recursos, execução, monitoramento e relatórios de aplicações submetidas ao cluster. A Figura 6 ilustra os componentes do Hadoop de maneira hierárquica e como o framework Spark se situa nessa hierarquia.

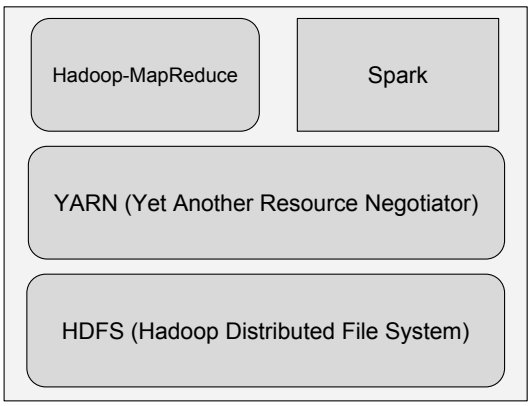


Figura 6 Hierarquia de componentes do Hadoop (MapReduce, YARN e HDFS) e Spark

2.2.1 Arquitetura

A arquitetura do Hadoop é baseada no paradigma mestre/escravo, em que uma máquina mestre é responsável por gerenciar diversas máquinas escravas. É possível ter mais de uma máquina mestre por cluster por meio de um recurso denominado HDFS Federation (HORTONWORKS, 2015; WHITE, 2012, p. 49) que fornece melhoria na escalabilidade e opção de isolamento de dados.

Para controlar todo o sistema de arquivos distribuído do Hadoop e manter o cluster em funcionamento há um serviço denominado *NameNode* que é executado na máquina mestre. Em muitos configurações típicas do Hadoop existe somente um *NameNode* (mestre) por cluster. Uma falha no mestre deixaria todo o sistema inoperante até que o *NameNode* inicie novamente (BORTHAKUR et al., 2011). Para as demais máquinas do cluster há um serviço denominado *DataNode* que gerencia o armazenamento e o controle de arquivos em cada nó. A Figura 7 apresenta o esquema da arquitetura do Hadoop.

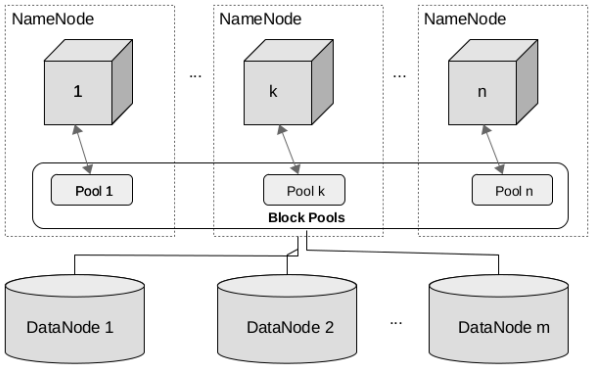


Figura 7 Arquitetura do Cluster Hadoop, adaptado de APACHE... (2015b).

Os arquivos armazenados no HDFS pelos usuários são divididos em um ou mais blocos de 128Mb (padrão) ou um tamanho definido pelo usuário, e são armazenados nos *DataNodes*. Os *DataNodes* também executam a criação, exclusão e replicação de blocos no sistema de arquivos. Os blocos no HDFS são replicados (3 vezes por padrão ou um valor especificado pelo usuário) através dos *DataNodes*. Essa replicação permite a recuperação de dados caso algum nó falhe, o que torna o Hadoop um sistema tolerante a falhas. O *Block Pool* é uma camada abstrata para facilitar o gerenciamento dos blocos de cada *NameNode* sem a necessidade de coordenação com os outros *NameNodes*. Caso algum *NameNode* venha a falhar, somente os dados o qual gerencia ficarão inacessíveis até que volte a executar.

O HDFS também suporta gerenciamento de arquivos para que o usuário possa criar, renomear e excluir arquivos e diretórios do sistema. Para isso, o *NameNode* fornece uma interface de linha de comando chamada FS Shell (WHITE, 2012, p. 51). Nela, o usuário interage com os dados no HDFS criando e removendo arquivos ou diretórios, além de ver conteúdo de arquivos de texto. Também fornece alguns comandos para administrar o cluster, por exemplo, listar e obter informações dos *DataNodes* ativos no momento.

2.2.2 Hadoop daemons

Os Hadoop *daemons* são os serviços que executam quando o Hadoop é iniciado. O *NameNode* é o *daemon* principal que controla o HDFS e dirige todos os *DataNodes*. Há ainda outros *daemons* fundamentais para sua execução. Como dito anteriormente, o *DataNode* está presente em cada máquina escrava do cluster Hadoop para executar o trabalho pesado do

sistema de arquivo - leitura e escrita no HDFS. Quando é requisitada a leitura, o *NameNode* irá informar quais *DataNodes* contêm cada bloco do arquivo. Os *DataNodes* irão processar os blocos correspondentes ao arquivo em questão. Um processo semelhante é o de escrita. Quando um arquivo é armazenado no HDFS, ele é dividido em blocos que serão replicados através dos *DataNodes*. Cada *DataNode* sabe quais blocos de quais arquivos ele está armazenando. Isso permite uma taxa alta de leitura/escrita de dados.

Outro *daemon* importante é o *Secondary NameNode* (SNN) (LAM, 2010, p. 23). Trata-se de um *daemon* assistente para monitoramento do estado do cluster HDFS, cuja principal função é criar pontos de restauração para recuperação de configuração, caso o *NameNode* venha a falhar.

Existem ainda dois outros *daemons* fundamentais na execução de aplicações para processamento de dados no HDFS: *ResourceManager* e *NodeManager*. O gerenciador de recursos, *ResourceManager*, é um *daemon* que executa na máquina mestre e sua principal função é atribuir os recursos necessários para todas as aplicações a serem executadas no sistema. Trabalha em conjunto com o *NodeManager*, o qual é executado em cada máquina escrava do cluster responsável por receber as instruções do *ResourceManager* e monitorar o uso dos recursos (cpu, memória, disco e rede) de sua respectiva máquina escrava. Há ainda outro serviço importante que trabalha junto com o *ResourceManager* e *NodeManager*, denominado *ApplicationMaster*. Esse serviço é executado para cada aplicação submetida ao cluster e responsável por negociar os recursos com o *ResourceManager*, gerenciar cada instância de uma aplicação executando no YARN, além de trabalhar com o *NodeManager* para alocar e monitorar o conjunto de recursos necessários para a aplicação.

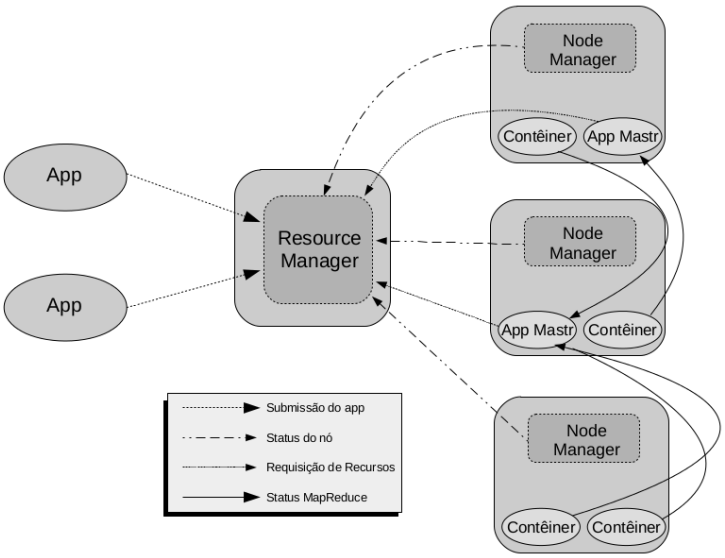


Figura 8 Arquitetura YARN, adaptado de APACHE... (2015c).

O *ResourceManager* possui dois componentes principais: um escalonador e um gerenciador de aplicação. O escalonador é responsável por alocar os recursos para todas as aplicações em execução. Os recursos são organizados abstratamente em contêineres de acordo com os requisitos da aplicação. O gerenciador de aplicação, por sua vez, é responsável por aceitar os *jobs* submetidos, negociar o contêiner de recursos com o *ApplicationMaster* e fornecer um serviço para reiniciar o *ApplicationMaster* em caso de falhas. A Figura 8 ilustra a arquitetura YARN.

É possível perceber a importância do *ResourceManager* para o cluster como um todo. O *NodeManager*, responsável por monitorar os recursos, reporta o status de cada contêiner de seus respectivos nós. Uma vez que uma aplicação é submetida pelo cliente, inicia-se o processo de negociação de recursos entre o *ApplicationMaster* e o *ResourceManager* para disponibilizar um contêiner necessário para atender os requisitos da aplicação. Enquanto

uma aplicação é executada, o *NodeManager* envia relatórios com status dos recursos utilizados. O *ApplicationMaster* gerencia a instância da aplicação que está sendo executada e, quando finaliza, ele reporta ao *ResourceManager* o fim da aplicação e termina.

A arquitetura YARN também vai além de aplicações MapReduce. É uma plataforma genérica que fornece um ambiente para outros tipos de paradigmas, tais como, processamento de grafos, processamento iterativo, machine learning e diversas computações complexas que podem ser executadas de maneira distribuída em um cluster de computadores.

2.2.3 Outros recursos do Hadoop

O Hadoop também conta com vários outros recursos que facilitam o trabalho do programador ao desenvolver seu sistema. Dentre eles, um recurso que é indispensável para muitas aplicações é a função *Combiner*, uma etapa opcional e intermediária entre o Map e o Reduce. Com essa função é possível fazer um pré-processamento dos dados antes de enviá-los para a função Reduce para diminuir a quantidade de dados que trafega pela rede do cluster (WHITE, 2012, p. 34). Em aplicações onde a função Map envia a mesma chave várias vezes para serem agrupadas na função Reduce é interessante usar a função *Combiner* para fazer esse agrupamento ainda localmente, aliviando o fluxo de dados na rede e reduzindo a demanda de processamento para a função Reduce.

Outro recurso interessante do Hadoop é o Hadoop Distributed Cache ou cache distribuído do Hadoop que fornece uma maneira de armazenar dados para serem utilizados nas aplicações MapReduce (WHITE, 2012, p. 288). Ao inserir um arquivo no cache distribuído ele será copiado lo-

calmente para cada máquina que irá executar a aplicação para economizar recursos de rede durante a execução. Além disso, é garantido que cada instância da aplicação acesse uma cópia idêntica do arquivo original. O cache distribuído não substitui a entrada padrão, apenas funciona como uma espécie de entrada secundária para que as aplicações possam acessar arquivos ou dados necessários para a sua execução.

2.3 Spark

O Spark (ZAHARIA et al., 2010) é uma plataforma distribuída de alto nível capaz de executar diversas operações sobre dados, inclusive o MapReduce. O Spark necessita de um gerenciador de cluster e pode executar integrado ao Hadoop, usufruindo de todos os recursos que ele oferece, como mostra a Figura 6. O Spark foi projetado com foco na velocidade de processamento de aplicações MapReduce para suportar de maneira eficiente diversas aplicações, tais como consultas iterativas e processamento de dados em fluxo contínuo (KARAU et al., 2015). Além disso, o Spark é mais adequado para aplicações iterativas do que o Hadoop-MapReduce, pois não necessita persistir dados em disco a cada iteração ou entre as funções Map e Reduce. Isso é possível pois os dados são manipulados em *RDDs* (Resilient Distributed Datasets) (ZAHARIA et al., 2012), uma abstração para coleções de elementos distribuídos e imutáveis. Os *RDDs* são divididos em múltiplas partições, as quais podem ser processadas, de acordo com a implementação do usuário, por máquinas diferentes em um cluster de computadores.

2.3.1 *RDD* (Resilient distributed dataset)

Os RDDs proveem às aplicações a característica de tolerância a falhas por meio da capacidade de recuperar os dados de um RDD ou alguma partição corrompida de um RDD a partir do seu histórico de operações. Ou seja, o RDD contém informações desde a sua criação que são suficientes para reconstruir uma partição, caso venha a falhar. Um RDD pode ser criado a partir de uma base de dados no HDFS ou de uma coleção de dados em memória na aplicação. É possível executar diversas operações nos dados em um RDD por meio das *actions* e *transformations*. As *transformations* (ou transformações) produzem um novo RDD a partir do RDD corrente e são executados de maneira preguiçosa, ou seja, são executados somente quando uma *action* (ou ação) posterior é executada. As ações são operações que, a partir de um RDD, retornam apenas um resultado para o programa principal ou persistem no disco. A Tabelas 1 e 2 listam algumas das principais operações para transformações e ações, respectivamente.

2.3.2 Arquitetura

Assim como o Hadoop, a arquitetura do Spark também é baseada no paradigma mestre e escravo, tendo o mestre responsável pela coordenação das execuções de aplicações no cluster. Ao submeter uma aplicação, o mestre executa o programa *driver* o qual contém uma instância do *SparkContext* e os demais códigos referentes à aplicação (RDDs, transformações e ações). O Spark converte as aplicações em unidades denominadas *tasks* e organiza a execução dessas unidades nas máquinas escravas (*executors)*. Os *executors* são responsáveis por executar as tarefas que lhe foram concebidas e reportar os resultados para o programa *driver* no mestre. A Figura 9 ilus-

Tabela 1 Operações para transformações.

OPERAÇÃO	DESCRIÇÃO
map()	Retorna um novo RDD processando cada elemento do RDD de origem por uma função.
flatMap()	Similar ao map(), porém cada elemento do RDD de origem pode ser mapeado para 0 ou mais elementos.
mapPartitions()	Similar ao map(), porém executa por partições.
mapPartitionsWithIndex()	Adiciona um identificador da partição ao mapPartitions().
groupByKey()	Retorna um novo RDD com todos os valores agrupados pela mesma chave.
reduceByKey()	Similar ao groupByKey(), porém os valores são processador por uma função.
filter()	Retorna um novo RDD com elementos que satisfazem uma condição.
sortByKey()	Retorna um novo RDD ordenado de modo ascendente ou descendente.

Fonte: Adaptado de (APACHE. . . , 2016b)

Tabela 2 Operações para ações.

OPERAÇÃO	DESCRIÇÃO
reduce()	Agrega os elementos do RDD por meio de uma função comutativa e associativa.
collect()	Retornar todos os elementos do RDD como uma única coleção.
count()	Retorna o número de elementos do RDD
saveAsTextFile()	Persiste os elementos do RDD em um ou mais arquivos de texto.
countByKey()	Retorna uma tabela hash com a chave e o contador de cada chave.
foreach()	Executa uma função sobre cada elemento do RDD.

Fonte: Adaptado de (APACHE. . . , 2016b)

tra a arquitetura do Spark. O Spark depende de um gerenciador do cluster para inicializar os *executors*, o qual pode ser Hadoop YARN, Mesos ou seu próprio gerenciador de cluster (KARAU et al., 2015, p 117).

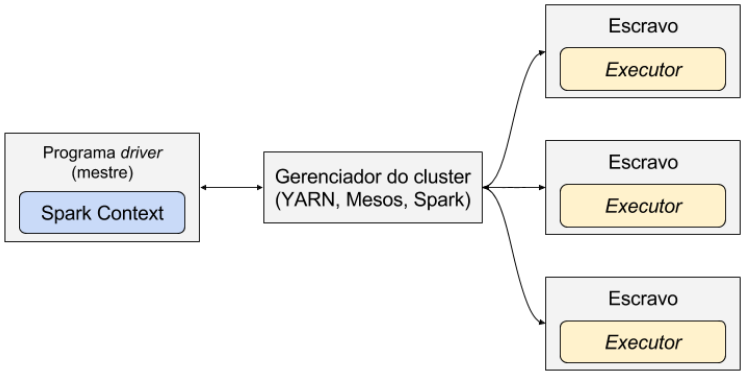


Figura 9 Arquitetura do Spark, adaptado de Karau et al. (2015).

2.3.3 Biblioteca *MLlib*

O Spark também possui uma biblioteca de algoritmos de aprendizagem de máquinas denominada *MLlib* (APACHE. . . , 2016a) para processamento de dados de maneira distribuída no cluster. É composta por algoritmos de classificação, regressão, agrupamento, filtro colaborativo, dentre outros. Todos os algoritmos foram projetados para trabalhar paralelamente utilizando os RDDs.

Os tipos dos dados de entrada suportados pela *MLlib* podem ser densos ou esparsos em formato de vetores ou matrizes. Os dados utilizados em algoritmos supervisionados são do tipo *Labeled Point*, pois cada instância deve possuir seu rótulo associado. A *MLlib* também suporta dados no formato do LIBSVM (CHANG; LIN, 2016) em que cada linha representa

uma instância com rótulo e atributos indexados. A Tabela 3 lista alguns dos principais algoritmos implementados na biblioteca *MLlib*.

Tabela 3 Alguns dos principais algoritmos na *MLlib*.

ALGORITMO	TIPO	DESCRIÇÃO
Naive Bayes	Classificação multiclasse	Baseia-se em probabilidades utilizando os valores dos atributos para predizer a classe de um determinada instância (ver Seção 2.4.2.2).
Regressão Linear	Classificação multiclasse	A classe ou rótulo é expressado como uma combinação linear dos atributos com pesos pré-determinados.
k-means	Agrupamento	As instâncias são agrupadas em k distintos grupos de acordo com os valores de seus atributos e de maneira iterativa.
Random Forest	Classificação multiclasse	Cria um conjunto de árvores de decisão construídas a partir dos valores dos atributos para predizer o rótulo de novas instâncias (ver Seção 2.4.2.1).
SVM	Classificação binária	Constroi hiperplanos para separar instâncias de classes diferentes de acordo com os valores de seus atributos.
ALS	Filtro colaborativo	Algoritmo para sistemas recomendação que se baseia em uma matriz com as dados de preferências do usuário em relação a um objeto (que pode ser um tipo de produto, por exemplo).

2.3.4 Outros recursos do Spark

O Spark contém dois recursos úteis em situações onde há a necessidade de compartilhar informações entre as máquinas do clusters: *accumulators* e *broadcast*. Os *accumulators* são utilizados como agregadores de valores (incrementadores) que são utilizados pelas máquinas escravas para incrementar um desejado valor que necessita ser recuperado na máquina mestre pelo programa *driver*. É comumente usado para contar a quantidade de ocorrências de um determinado evento para fins de depuração. São instanciados na máquina mestre (programa *driver*) e enviados como parâmetro para funções executadas pelos RDDs. As máquinas escravas podem apenas incrementar os valores dos *accumulators*.

As variáveis *broadcast* possibilitam compartilhar valores mais complexos (vetores ou tabelas) de somente leitura entre mestre e escravos. Assim como os *accumulators*, as variáveis *broadcast* são criadas no programa *driver*, onde são inseridos os valores necessários e enviadas como parâmetro para funções executadas por um RDD (map() ou reduce(), por exemplo). As máquinas escravas apenas acessam o valor da variável *broadcast* para realizar operações e demais fins.

2.4 Mineração de dados

Devido ao constante aumento do volume de dados, a necessidade de analisá-los de forma efetiva e rápida tornou-se fundamental. A mineração de dados surgiu da necessidade de busca por conhecimento oculto e padrões em diversos conjuntos de dados de forma automatizada, que podem ser utilizados para diversos fins, tais como, auxílio na tomada de decisão, compreensão de comportamentos de usuários e clientes, detecção de fraudes, dentre várias outras aplicações (WITTEN; FRANK; HALL, 2011). Com a tecnologia cada vez mais presente na sociedade, torna-se fácil armazenar dados sobre hábitos de compras, hábitos financeiros, além de outros, que podem conter algum conhecimento oculto ou algum padrão útil.

Mineração de dados (*data mining*) é uma etapa do processo *KDD* (*Knowledge Discovery in Databases* - Descoberta de Conhecimento em Banco de Dados) que visa encontrar informação importante, não trivial, implícita e previamente desconhecida em conjuntos de dados tratados e organizados em algum sistema de armazenamento como um *data warehouse* (PIATESKI; FRAWLEY, 1991).

Em mineração de dados existem diversas técnicas para a busca de padrões desconhecidos em conjunto de dados, dentre elas há a técnica de regras de associação e a de classificação. As regras de associação objetivam identificar relações de ocorrências entre itens de um mesmo conjunto de dados por meio de combinações desses itens (AGRAWAL; IMIELINSKI; SWAMI, 1993). A técnica de classificação, por sua vez, visa atribuir rótulos a entidades desconhecidas, mas que possuem um conjunto de características (*features*) capazes de definir a que rótulo ou classe tal entidade pertence (SEBASTIANI, 2002). Nas subseções seguintes são apresentados mais detalhes sobre essas técnicas.

2.4.1 Regras de associação

Regras de associação objetivam identificar relações de ocorrências entre itens de um mesmo conjunto de dados (AGRAWAL; IMIELINSKI; SWAMI, 1993). Inicialmente, essa técnica foi aplicada para banco de dados de transações de supermercados, onde os elementos ou itens são os produtos e as transações são os conjuntos de itens adquiridos em uma mesma compra no supermercado. O objetivo é gerar regras de acordo com os padrões de relacionamentos entre itens de uma determinada base de dados de transações. Essa regras são do tipo "*if then*", ou seja, "se ocorreu isso, então ocorreu aquilo", no sentido de implicabilidade. Por exemplo, em uma base de dados de supermercado descobriu-se que diversos clientes que compraram manteiga também compraram pão. A regra, então, ficaria da seguinte forma: se o cliente compra manteiga, então compra pão. Ou seja "comprar manteiga" implica "comprar pão". Em regras de associação um atributo

(comprar manteiga) pode predizer um ou mais atributos (se cliente compra manteiga, então compra pão e compra leite, por exemplo).

Uma definição mais formal (HAN; KAMBER; PEI, 2011, p. 246) pode ser apresentada da seguinte forma. Seja $I = \{I_1, I_2, \ldots, I_m\}$ um conjunto de itens. Seja D o conjunto de dados de transações, onde cada transação T é formada por um conjunto de itens, tal que $T \subseteq I$. Cada transação é associada com um identificador, chamado TID. Uma regra de associação é uma implicação da forma $A \Rightarrow B$, onde $A \subset I$, $B \subset I$, $A \neq \emptyset$, $B \neq \emptyset$ e $A \cap B = \varnothing$. A regra $A \Rightarrow B$ sustenta-se no conjunto de transações D com o suporte s, em que s é a porcentagem de transações em D que contém $A \cup B$. Isso é tomado como uma probabilidade $P(A \cup B)$, como apresentado na Equação 1. Nesse contexto, o sentido de união ($A \cup B$) se refere à probabilidade de ocorrência do conjunto A **e** B nas mesmas transações, ao invés do sentido de A **ou** B (probabilidade de ocorrer ou o conjunto A ou o conjunto B nas mesmas transações).

A regra $A \Rightarrow B$ possui confiança c no conjunto de transações D, em que c é a porcentagem de transações em D que contém A e também contém B. A confiança é tomada como uma probabilidade condicional $P(B|A)$, como apresentado na Equação 2.

$$suporte(A \Rightarrow B) = P(A \cup B) \tag{1}$$

$$confiança(A \Rightarrow B) = P(B|A) \tag{2}$$

As regras fortes, ou regras interessantes (relevantes para o usuários), são as que satisfazem um limite mínimo definido pelo usuário para o suporte e para a confiança.

Um conjunto de itens é referenciado como *itemset* e um *itemset* que contém k itens é um *k-itemset*. Por exemplo, o conjunto {pão, manteiga} é um *2-itemset*. A frequência de ocorrência de um *itemset* é conhecida como contador de suporte do *itemset*. Se o contador de suporte de um *itemset* satisfaz o suporte mínimo definido pelo usuário, então esse *itemset* é considerado frequente. O conjunto dos *k-itemsets* frequentes é comumente denotado por L_k.

A confiança de uma regra $A \Rightarrow B$ pode ser derivada dos contadores de suportes dos *itemsets* envolvidos, como mostra a Equação 3.

$$confiança(A \Rightarrow B) = P(B|A) = \frac{suporte(A \cup B)}{suporte(A)} \tag{3}$$

De maneira geral, mineração de regras de associação pode ser vista como um processo de duas etapas:

1. Encontrar todos os *itemsets* frequentes: processo que efetua combinações entre os itens da base transações D, respeitando o suporte mínimo.

2. Gerar regras de associações fortes a partir dos *itemsets* frequentes: deve-se considerar a confiança mínima para produzir regras interessantes.

Uma implementação da técnica de regras de associação é denominada *Apriori* (AGRAWAL; SRIKANT, 1994). Seu nome deriva do fato de que o algoritmo usa o conhecimento prévio das propriedades do *itemset*. Ele

emprega uma abordagem iterativa por nível, onde os *k-itemsets* são utilizados para formar os *(k+1)-itemsets*. Primeiramente, *1-itemsets* frequentes são encontrados, percorrendo a base de dados D para contar o suporte de cada item individualmente, eliminando aqueles que não atingem o valor do suporte mínimo. O resultado é denotado como L_1 e é utilizado na próxima etapa para encontrar o conjunto de itens de tamanho 2, L_2. L_2 será utilizado para construir L_3, e assim sucessivamente, até que não seja mais possível encontrar *k-itemsets* frequentes. O pseudocódigo do algoritmo *Apriori* é apresentado no Algoritmo 1.

Algoritmo 1: Algoritmo Apriori, adaptado de Agrawal e Srikant (1994)

Input: L_1, todos os 1-itemsets frequentes; D dataset de transações
Output: Todos os itemsets frequentes

```
1  for (k = 2; L_{k-1} ≠ ∅; k++) do
2  |  C_k = aprioriGen(L_{k-1});          // geração de candidatos
3  |  foreach transaction t in D do
4  |  |  C_t = subSet(C_k,t);             // verifica o candidato
5  |  |  foreach candidates c in C_t do
6  |  |  |  c.count++;                    // conta o suporte do candidato
7  |  |  end
8  |  end
9  |  L_k = {c ∈ C_k| c.count >= min_sup}
10 end
11 return U_k L_k
```

Para encontrar cada L_k, é necessário percorrer toda a base de dados. Em uma grande base de transações, essa abordagem pode não ser eficiente. Entretanto, existe uma propriedade do *Apriori* que visa melhorar o desempenho do algoritmo para iteração. Tal propriedade diz que, todo o subconjunto não vazio de um *itemset* frequente deve ser frequente também.

Ou seja, se um *itemset* I não satisfaz o limite do suporte mínimo (*min_sup*), então I não é frequente, isto é, $P(I) < min\_sup$. Se um item A é acrescentado para o *itemset* I, então o *itemset* resultante ($I \cup A$) não pode ser mais frequente do que I. Logo, $I \cup A$ não é frequente, $P(I \cup A) < min\_sup$.

O processo para encontrar L_k consiste de duas etapas: etapa de junção e etapa de poda. Na etapa de junção (função *aprioriGen()*, Algoritmo 1), é gerado um conjunto de itens candidatos a partir da junção do conjunto L_{k-1} com ele mesmo, denotado por C_k. Dado que l_1 e l_2 são *itemsets* em L_{k-1}, a notação $l_i[j]$ se refere ao j-ésimo item em l_i (e.g., $l_1[k-2]$ se refere ao penúltimo item em l_1). O algoritmo *Apriori* assume que os itens nas transações ou em *itemsets* estão ordenados lexicograficamente. Dizer que os itens estão ordenados em *(k-1)-itemset* significa que $l_i[1] < l_i[2] < \ldots < l_i[k-1]$. A junção $L_{k-1} \bowtie L_{k-1}$ é realizada, onde os *itemsets* de L_{k-1} possuem os primeiros $k-2$ itens em comum. Ou seja, os *itemsets* l_1 e l_2 de L_{k-1} são unidos se $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \ldots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$. A condição $l_1[k-1] < l_2[k-1]$ apenas evita a geração de duplicatas. O *itemset* resultante dessa união é $l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]$.

A etapa de poda consiste no processo de exclusão dos itens não frequentes do conjunto de itens candidatos, obtidos pela função *aprioriGen()*. A função *subSet()* verifica, para cada transação t, se há um subconjunto candidato em t. Esse subconjunto é descoberto percorrendo uma estrutura de *HashTree*, onde os nós folhas são os *itemsets* e cada nó interno consiste em um conjunto de códigos *hashs* de itens que formam o *itemset* do nó folha, construído na função *aprioriGen()* (AGRAWAL; SRIKANT, 1994). Para encontrar o conjunto de todos os itens candidatos, inicia-se a busca pelo nó raiz. Ao chegar no nível d da árvore, calcula-se o código *hash* do item i_d do

itemset. Esse código informa qual ramo deve-se percorrer. Ao atingir um nó folha, significa que encontrou um *itemset* candidato e este, por sua vez, é adicionado ao conjunto dos itens candidatos. O contador de suporte de cada *itemset* é calculado (linhas 5 e 6 do Algoritmo 1) para armazenar em L_k somente os *itemsets* que possuem contador de suporte maior ou igual ao *min_sup* (linha 9 do Algoritmo 1)

Uma vez que o conjunto de todos os itens frequentes da base de dados D é obtido, inicia-se o processo de geração das regras de associação entre esses itens. Para isso, é preciso utilizar a Equação 3, que calcula a taxa de confiança da regra, possibilitando a geração de regras fortes (relevantes para o usuário). As regras podem ser geradas da seguinte forma:

- Para cada *itemset* frequente l, geram-se todos os subconjuntos de l.
- Para todo subconjunto não vazio s de l gera a regra "$s \Rightarrow (l - s)$" se $\frac{suporte(l)}{suporte(s)} \geq min\_conf$, onde min_conf é o limite mínimo para o valor da confiança.

Como as regras são geradas a partir dos *itemsets* frequentes, eles satisfazem automaticamente a restrição do suporte mínimo. Dessa forma, o algoritmo produz o conjunto de todas as regras interessantes para o usuário (por meio da especificação do suporte mínimo e confiança mínima) que descrevem relações entre os itens da base de dados de transações.

Uma abordagem diferente para encontrar os *itemsets* frequentes, sem a etapa de geração e poda de itens candidatos, é implementada no algoritmo denominado *FP-Growth* (HAN; PEI; YIN, 2000). Uma estrutura de árvore, *FP-tree*, é utilizada para representar a base de dados de entrada. Ela é construída a partir dos itens das transações, ordenados pelo tamanho do

contador de suporte. Cada transação é representada como um caminho na árvore e diferentes transações podem ter itens em comum, resultando em caminhos sobrepostos. A partir dessa estrutura, é possível extrair todos os conjuntos de itens frequentes, começando pelos nós folhas da árvore e seguindo em direção à raiz.

2.4.2 Classificação

A técnica de classificação visa categorizar entidades desconhecidas, mas que possuem um conjunto de características (*features*) capazes de definir a que categoria ou classe tal entidade pertence (SEBASTIANI, 2002). Ou seja, a partir de um conjunto de dados é possível predizer, de maneira automática, a que categoria (ou classe) cada instância pertence de acordo com suas características previamente definidas.

Basicamente o processo de classificação é constituído de duas etapas: etapa de aprendizagem (treinamento) e etapa de classificação (teste). Na etapa de aprendizagem, um conjunto de dados já rotulados (que possuem suas classes conhecidas) é utilizado para a criação de um modelo que será utilizado na etapa de classificação. Essa etapa é conhecida como aprendizagem supervisionada, uma vez que o algoritmo constroi um modelo baseado em dados que possuem suas classes conhecidas.

Na etapa de classificação (ou predição), um novo conjunto de dados com seus rótulos desconhecidos é utilizado para a classificação. O processo visa predizer as classes dos novos dados por meio do modelo gerado na etapa de aprendizagem. É importante que tanto os dados de aprendizagem quanto os de classificação sejam equivalentes, ou seja, devem possuir a mesma estrutura de atributos para que o classificador gere resultados interessantes.

Exemplos de algoritmos de classificação incluem classificadores de árvore de decisão, classificadores baseados em regras, redes neurais, máquinas de vetor de suporte e classificadores Bayes simples (HAN; KAMBER; PEI, 2011). Cada técnica emprega um algoritmo de aprendizagem para identificar um modelo que seja mais apropriado para o relacionamento entre o conjunto de atributos e o rótulo da classe dos dados de entrada.

Neste trabalho, as implementações do algoritmo de classificação de ofertas de produtos no Hadoop-MapReduce e Spark foram comparadas com alguns algoritmos de classificação multiclasse, *Random Forest* e *Naive Bayes*, disponíveis na biblioteca *MLlib* do Spark. Nas próxima seções será descrito o funcionamento geral desses algoritmos.

2.4.2.1 Random forest

O *Random Forest* (HAN; KAMBER; PEI, 2011, p 382) é um algoritmo de classificação que combina diversos classificadores de Árvore de Decisão (WITTEN; FRANK; HALL, 2011, p 64) de forma que cada árvore depende dos valores de um vetor com amostras aleatórias e independentes de atributos com a mesma distribuição para todas as árvores. Cada árvore de decisão é criada a partir de uma amostra de dados do conjunto de treino. Para cada nó da árvore, um conjunto de atributos é selecionado para definir as ramificações do nó. Esse número de atributos geralmente é menor do que a quantidade total de atributos disponíveis. As árvores crescem até o tamanho máximo ou até uma profundidade definida previamente. O nó folha das árvores é representado pela classe ou rótulo. O processo de classificação consiste em aplicar dados não rotulados para as árvores construídas

no modelo. O rótulo atribuído a esses dados será aquele mais votado pelas árvores, ou seja, o mais popular encontrado pelas árvores do modelo.

2.4.2.2 Naive bayes

Naive Bayes (HAN; KAMBER; PEI, 2011, p 350) é um algoritmo de classificação que constroi modelos baseados em probabilidades para predizer uma nova instância de dados não rotulada. A probabilidade é construída de acordo com os valores dos atributos que aqui são considerados igualmente importantes e independentes. Dado uma instância X com n atributos $(x_1, x_2, ..., x_n)$ e uma classe C, a probabilidade $Pr[C|X]$ da instância X pertencer a classe C é dada pela Equação 4.

$$Pr[C|X] = \prod_{i=1}^{n} Pr(x_i|C) \tag{4}$$

Onde $Pr(x_i|C)$ é a probabilidade do valor de o atributo x_i ocorrer para a classe C. A probabilidade de uma nova instância é calculada para cada classe e a que apresentar a maior probabilidade é atribuída à instância.

2.5 Revisão de implementações do Apriori

A necessidade de otimizar o desempenho de algoritmos de mineração de dados, especificamente algoritmos de mineração de regras de associação, tem ganhado relevância nos últimos anos. O principal objetivo é reduzir o tempo gasto para a etapa de geração do conjunto de itens frequentes diante de uma quantidade elevada de dados. A ideia é utilizar uma estratégia de distribuição de processamento, onde cada parte processa uma porção dos dados e no final tem-se a junção dos resultados de cada parte.

Há estratégias que implementam a distribuição de processamento sem utilizar o MapReduce, dentre elas, destacam-se (BOLINA et al., 2013; CHEUNG; LEE; XIAO, 2002; ZAKI et al., 1996; YE; CHIANG, 2006; YU; ZHOU, 2008; ZAKI; PARTHASARATHY; LI, 1997). Essas estratégias são fundamentadas nos algoritmos *Count Distribution*, *Data Distribution* e *Candidate Distribution* (AGRAWAL; SHAFER, 1996). O *Count Distribution* distribui a etapa de geração de candidatos para os processadores, em uma abordagem de k passos. Cada processador trabalha com uma porção dos dados para produzir o conjunto de itens candidatos de tamanho k, a partir do conjunto *k-1* gerado no passo anterior. No final de cada ciclo, os dados devem ser sincronizados. O problema é que processadores diferentes podem gerar os mesmos *itemsets*. Para contornar isso, foi proposto o *Data Distribution*, onde cada processador trabalha com *itemsets* mutualmente exclusivos. Entretanto, cada processador precisa trocar dados com todos os outros, o que pode sobrecarregar o meio de comunicação. O algoritmo *Candidate Dsitribuition* particiona tanto os dados de transações, quanto os candidatos e replica-os de maneira seletiva. Dessa forma, cada processador trabalha independentemente, sem a necessidade de constantes trocas de informações entre eles.

Zaki et al. (1996) abordam o problema de geração de itens frequentes em um ambiente de memória compartilhada. O algoritmo particiona as estruturadas de dados em memória para cada processador, de modo a otimizar o balanceamento de carga. A etapa de geração de itens candidatos é realizada por meio de uma estratégia de particionamento dos itemsets em classes equivalentes. Essa estratégia é baseada nos prefixos dos itemsets, onde aqueles que tiverem prefixos comuns ficam agrupados em uma

mesma classe. A geração de novos itens candidatos é feita a partir da combinação entre os itens das mesmas classes. Zaki, Parthasarathy e Li (1997) apresentam uma estratégia, denominada *ECLAT* (*Equivalence Class Transformation*), que organiza os *itemsets* e transações relacionadas em grupos disjuntos. Também implementam a abordagem de classes equivalentes apresentada por Zaki et al. (1996) para agrupar os itemsets pelo prefixo comum. Esses grupos são distribuídos para processadores distintos, onde gera-se todos os *itemsets* em um único passo. Em seguida, utiliza-se uma estrutura de arquivo invertido para contar o suporte e produzir os *itemsets* frequentes. Essa estrutura é composta dos *itemsets* gerados e, para cada um, há uma lista de transações em que ocorreu. Essa abordagem facilita a contagem de suporte realizada por meio da interseção da lista de transações.

Em (YU; ZHOU, 2008) é apresentada uma abordagem que também utiliza arquivo invertido, além de um esquema de pesos para evitar que poucos processadores trabalhem com muitos *itemsets* enquanto outros processem menos, melhorando a distribuição de carga entre eles. Outra abordagem, denominada DTMA (*Distributed Multithread Apriori*), que utiliza arquivo invertido é apresentada em (BOLINA et al., 2013). Nela, o algoritmo distribui, de maneira circular, as operações de interseção entre as listas de transações para cada processo em execução. Cada processo cria *threads* para distribuir as operações de comparação, garantindo melhor balanceamento de carga.

Outras estratégias utilizam o MapReduce como modelo base de programação para implementar o algoritmo *Apriori* (FARZANYAR; CERCONE, 2013a; FARZANYAR; CERCONE, 2013b; LI et al., 2012; LI; ZHANG, 2011; LIN; LEE; HSUEH, 2012; YAHYA; HEGAZY; EZAT, 2012; YANG;

LIU; FU, 2010; ZHOU; HUANG, 2014). Alguns destes trabalhos apresentam estratégias parecidas, com poucas distinções, enquanto outros utilizam de uma abordagem exclusiva para a implementação. Dentre eles também há propostas de melhorias de estratégias já publicadas, visando diminuir o tempo, o custo computacional ou a quantidade de dados que trafegam pelas máquinas do cluster durante a execução do MapReduce. A seguir são apresentados maiores detalhes dessas abordagens e uma maneira de organizá-las em categorias.

2.5.1 Categorização das abordagens Hadoop-MapReduce

Geralmente a implementação de algoritmos de regras de associação para o MapReduce envolve um processo iterativo em que cada iteração consiste de uma execução MapReduce. A quantidade de iterações de uma implementação pode ser quantificada em número de "fases" MapReduce necessárias para encontrar todos os conjuntos de itens frequentes. É possível efetuar uma organização destas propostas pela quantidade de fases (iterações) necessárias para encontrar o resultado final, como já introduzido em (FARZANYAR; CERCONE, 2013a), porém não foram incluídos todos os trabalhos aqui mencionados. Foram identificadas 3 categorias principais em relação ao número de fases MapReduce: uma fase, duas fases ou k fases, onde k é o maior tamanho dos conjuntos de itens gerados.

Quanto mais fases a implementação necessitar, maior será o custo de comunicação, alocação de recursos e inicialização da aplicação MapReduce (FARZANYAR; CERCONE, 2013a; LIN; LEE; HSUEH, 2012). Em contrapartida, uma implementação que possui somente uma fase MapReduce terá maior custo computacional por máquina do cluster, elevando o tempo

gasto para encontrar todos os conjuntos de itens frequentes. Para os algoritmos que gastam duas fases MapReduce, a quantidade de dados enviados das funções Map para a Reduce pode ser bastante elevada, comprometendo o desempenho das funções Reduce. Nessa etapa, o *framework* efetua o processo de ordenação e agrupamento dos dados por meio das chaves e, muitas vezes, é necessário armazená-los no disco quando não há espaço suficiente na memória física. Processamento que envolve operações de I/O no disco geralmente tem maior custo, e a CPU fica inutilizada até que essas operações finalizem, o mesmo ocorre no ambiente Hadoop-MapReduce (MAZUR et al., 2012).

2.5.2 Algoritmo de uma fase MapReduce

Li e Zhang (2011) - The Strategy of Mining Association Rule Based on Cloud Computing

Em Li e Zhang (2011), o algoritmo encontra todos os conjuntos de itens em apenas uma execução do MapReduce. Um algoritmo genérico para essa abordagem é apresentado nos Algoritmos 2 e 3. A estratégia é gerar todos os conjuntos de itens candidatos nas funções Map, contar o suporte parcial utilizando a função Combiner e, nas funções Reduce, contar o suporte global de todos os *itemsets* candidatos para gerar o conjunto de itens frequentes.

Um exemplo genérico do fluxo MapReduce para a abordagem de uma fase é apresentado na Figura 10. A base de transações de entrada é dividida em blocos, os quais são atribuídos para as funções Map. Cada função Map lê linha por linha de seu bloco de transações correspondente (linhas 1 e 2, Algoritmo 2) e extrai todas as combinações de itens possíveis em cada

Algoritmo 2: Função Map - Algoritmo de uma fase, adaptado de Yahya, Hegazy e Ezat (2012)

```
   Input: Bloco de transações S_i, uma por linha
   Output: pares <chave,1>, chave é um item candidato
 1 foreach transaction t in S_i do
 2 |   Function Map(line offset, t)
 3 |   |   foreach itemset I in t do      /* para todos os itemsets
   |   |   possíveis em t */
 4 |   |
 5 |   |   |   Output(I,1);
 6 |   |   end
 7 |   end
 8 end
```

Algoritmo 3: Função Reduce - Algoritmo de uma fase, adaptado de Yahya, Hegazy e Ezat (2012)

```
   Input: pares <chave2, lista(valor2)>, chave2 é um itemset
          candidato e valor2 é uma lista de elementos 1; min_sup
   Output: pares <chave3,valor3>, chave3 é um itemset
           frequente e valor3 é seu contador de suporte
 1 min_sup = read min_sup do DistributedCache;
 2 Function Reduce(chave2, lista(valor2))
 3 |   soma = 0;
 4 |   while (valor2.hasNext()) do
 5 |   |   soma += valor2.next()
 6 |   end
 7 |   if (soma >= min_sup) then
 8 |   |   Output(chave2,soma);
 9 |   endif
10 end
```

transação (linha 3 da função Map). Define o contador de suporte como sendo 1 e emite a saída no formato <*itemset*,1> (linha 5 da função Map). Os dados são agrupados pelas chaves e ordenados na etapa Shuffle/Sort, executada pelo *framework*, antes de serem enviados para as funções Reduce. Observa-se na Figura 10 que, na etapa Shuffle/Sort, cria-se uma lista de valores para cada *itemset*.

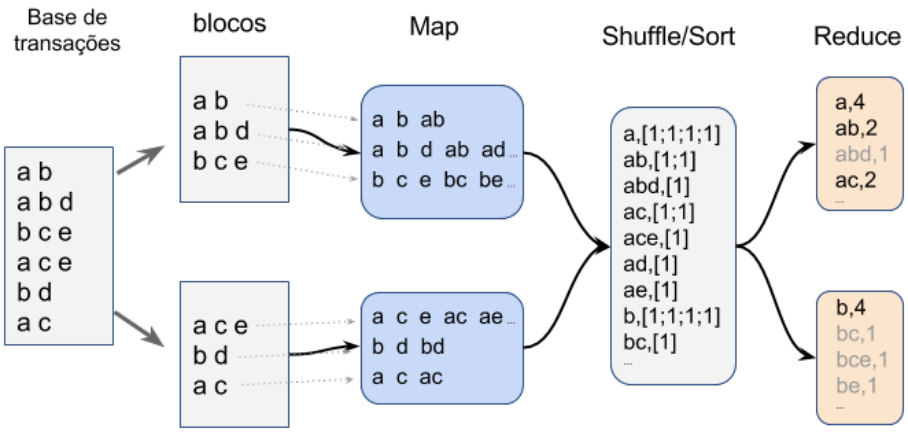


Figura 10 Exemplo genérico para um algoritmo de uma fase com suporte mínimo igual a 2 ($\approx 34\%$).

Ao receber os dados, as funções Reduce contam o suporte global de cada *itemset* (linhas 4 e 5 da função Reduce, Algoritmo 3) e efetua a poda dos *itemsets* não frequentes (linha 7 da função Reduce). Os *itemsets* não frequentes do exemplo da Figura 10 estão diferenciados com tom de cor mais claro nas funções Reduce. A saída do algoritmo (os *itemsets* frequentes) é salva no HDFS. Por fim, os resultados são mesclados para produzir todos os conjuntos de itens frequentes. Para otimizar o desempenho da implementação, os autores empregaram uma estratégia de distribuição de dados, alterando o tamanho dos blocos no HDFS de 64Mb para 16Mb. Isso possibilitou distribuir os dados do experimento de modo mais eficiente pelo

cluster. Foi observada, com essa estratégia, uma melhoria de desempenho em relação às configurações padrão do Hadoop.

Quando a base de transações de entrada é muito grande, a quantidade de dados emitidos pelas funções Map pode ser exorbitante. Para evitar a sobrecarga na rede, os autores utilizaram a função Combiner, como passo intermediário, para capturar as saídas da função Map local e contar o suporte parcial de cada item. A implementação é semelhante à função Reduce do Algoritmo 3, porém a execução é realizada localmente, em cada máquina que executa uma função Map. As funções Combiners obtêm os pares <*itemsets*,1> emitidos pelas funções Maps e realizam a contagem local do suporte, uma vez que cada função Map pode emitir o mesmo *itemset* várias vezes. A saída das funções Combiners são os pares <*itemset*,*sup*>, onde *sup* é o suporte local do *itemset*.

A fim de otimizar o desempenho da etapa Reduce, os autores reimplementaram a função de particionamento do Hadoop (LAM, 2010, p. 49). A estratégia é distribuir os pares intermediários gerados pelas funções Combiner para diferentes Reducers de maneira eficiente, evitando possível má distribuição de carga entre as funções Reduces.

O critério de avaliação utilizado foi o *SpeedUp* (HENNESSY; PATTERSON, 2012, p. 39), medindo o tempo de execução do algoritmo em relação à quantidade de máquinas no cluster para um determinado conjunto de dados e quantidade de memória. O tempo de execução com até 3 máquinas apresenta uma redução expressiva em relação a uma máquina só. A partir de 4 máquinas ainda se observa redução do tempo, porém não tão significativa. A melhoria de desempenho utilizando a estratégia de

distribuição de dados só é visível a partir da adição da quinta máquina no cluster.

2.5.3 Algoritmos de duas fases MapReduce

Na proposta apresentada em Yahya, Hegazy e Ezat (2012) são necessárias duas fases MapReduce para encontrar o resultado final. A ideia é aplicar o algoritmo Apriori (AGRAWAL; SRIKANT, 1994), para geração e poda dos *itemsets* candidatos, em cada bloco de dados da base de transações de entrada. Na Fase 1, o problema geral é subdividido em subproblemas menores, ou seja, em vez de aplicar o algoritmo Apriori em toda a base de dados de uma vez, sobrecarregando os recursos computacionais de apenas uma máquina, utilizam bem os recursos do Hadoop, pois ao inserir a base de dados no HDFS, ela é subdivida em blocos e espalhados pelas máquinas do cluster. Uma característica que diferencia essas propostas das outras é que em vez de a função Map ler linha por linha da porção de dados, ela lê todo o conjunto de transações de uma vez, por meio de alteração no núcleo de funcionamento do Hadoop-MapReduce.

As propostas apresentadas em Farzanyar e Cercone (2013a), Farzanyar e Cercone (2013b) são estratégias de melhorias aplicadas no algoritmo de Yahya, Hegazy e Ezat (2012). Em Farzanyar e Cercone (2013b) é introduzida uma solução de podas que estima o suporte global dos itemsets na função Reduce da Fase 1, somente os itemsets que possuírem o contador de suporte estimado maior ou igual ao suporte mínimo serão processados na Fase 2. Em Farzanyar e Cercone (2013a) foi aplicada uma melhoria na abordagem anterior que utiliza a estratégia de partições para que na Fase 2 os itemsets parcialmente frequentes só sejam contados nas partições (blo-

cos) em que não foram frequentes. Os detalhes dessas implementações são apresentados a seguir.

Yahya, Hegazy e Ezat (2012) - An Efficient Implementation of Apriori Algorithm Based on Hadoop-Mapreduce Model

Os Algoritmos 4 e 5 apresentam pseudocódigos genéricos para a Fase 1, baseados na abordagem de Yahya, Hegazy e Ezat (2012). Um exemplo genérico é apresentado na Figura 11. Nessa abordagem, denominada *MRA-priori* (*MapReduce Apriori*), cada bloco distinto, S_i, no cluster é processado em uma função Map, aplicando o algoritmo Apriori (linha 2 da função Map). Observa-se que é efetuada a poda, por meio do suporte mínimo, dos *itemsets* não frequentes na porção de dados. Se um *itemset* foi gerado e podado em todas as funções Map, não há chances de ele ser frequente em relação a toda a base de transações, portanto, é descartado. Como é visto na Figura 11, o algoritmo Apriori retorna apenas os *itemsets* que são frequentes em S_i, ou seja, aqueles que possuem o contador de suporte maior ou igual a 40% em S_i. As funções Map emitem os pares $<itemset,sup>$, onde *sup* é o suporte parcial do *itemset* em S_i. Os pares são agrupados e ordenados na etapa de Shuffle/Sort e enviados para as funções Reduce, que os salvam no HDFS. Se um *itemset* for frequente em pelo menos uma função Map, ele é considerado parcialmente frequente e tem chance de ser frequente em toda a base de transações.

Na Fase 2, além do bloco de entrada em cada função Map, uma entrada extra é adicionada. Essa entrada é a saída da Fase 1, que é adicionada no cache distribuído do Hadoop contendo o conjunto de itens parcialmente frequentes. Os Algoritmos 6 e 7 apresentam pseudocódigos genéricos para

Algoritmo 4: Função Map - Fase 1 do algoritmo de duas fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: Bloco de transações S_i
Output: pares <chave,valor> onde chave é um itemset frequente parcial e valor é seu contador de suporte

```
1 Function Map(block offset, S_i)
2 |   L = Apriori(S_i);   /* encontrar todos itemsets parcialmente
  |   frequentes para o bloco S_i */
3 |   foreach itemset i in L do
4 |   |   Output(i,i.parcial_sup);
5 |   end
6 end
```

Algoritmo 5: Função Reduce - Fase 1 do algoritmo de duas fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: pares <chave2,lista(valor2)>
Output: pares <chave2,1>, onde chave2 é um itemset frequente parcial

```
1 Function Reduce(chave2, lista(valor2))
2 |   Output(chave2,1)
3 end
```

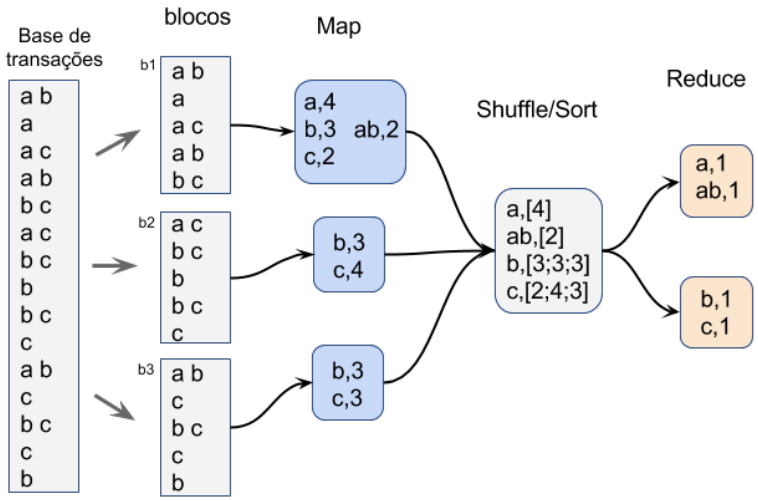


Figura 11 Exemplo genérico da Fase 1 para um algoritmo de duas fases com suporte mínimo igual a 6 em relação a toda base de transações (ou 40%).

essa fase, baseado na abordagem de Yahya, Hegazy e Ezat (2012). Um exemplo genérico é apresentado na Figura 12. O objetivo das funções Map dessa fase é efetuar a contagem local do suporte de cada *itemset* parcialmente frequente e emitir a saída para a função Reduce no formato <*itemset*,*sup*>, onde *sup* é o contador de suporte parcial (linhas 4 e 5, Algoritmo 6).

A função Reduce, por sua vez, faz a contagem global de cada item (linhas 2 e 4, Algoritmo 7) e efetua a poda de acordo com o suporte mínimo para produzir o conjunto final de itens frequentes (linhas 6 e 7, Algoritmo 7).

Para avaliar o desempenho do algoritmo, os autores compararam o tempo de execução com outras duas propostas, uma com k fases e outra com uma fase. Nos primeiros experimentos, os autores já constataram que o algoritmo de uma fase possui o tempo de execução muito mais elevado do que os demais e já o descartaram nos experimentos seguintes. Em todos

Algoritmo 6: Função Map - Fase 2 do algoritmo de duas fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: Bloco de transações S_i; L_p itemsets frequentes parciais para a partição i com suporte parcial
Output: pares <chave,valor> chave é um elemento de L_p e valor é seu contador de suporte parcial em S_i

```
1 read L_p do DistributedCache;
2 Function Map(block offset, S_i)
3 |  foreach itemset i in L_p do
4 |  |  count = countInSi(i,S_i);
5 |  |  Output(i,(count,i.partialSup)); /* enviar o suporte
  |  |  parcial e o suporte local para o Reduce */
6 |  end
7 end
```

Algoritmo 7: Função Reduce - Fase 2 do algoritmo de duas fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: pares <chave2,lista(valor2)>, chave2 é um candidato global e o valor2 são suas ocorrências em cada bloco
Output: pares <chave3,valor3>, onde chave3 é o itemset frequente global e valor3 é seu contador de suporte global

```
1 Function Reduce(chave2, lista(valor2))
2 |  soma = valor2.getPartialSup(); /* suporte parcial
  |  encontrado na Fase 1 */
3 |  while value2.hasNext() do
4 |  |  soma += value2.next().count; // suporte local para cada
  |  |  Mapper
5 |  end
6 |  if soma >= min_sup then
7 |  |  Output(chave2,soma); // itemset frequente global
8 |  endif
9 end
```

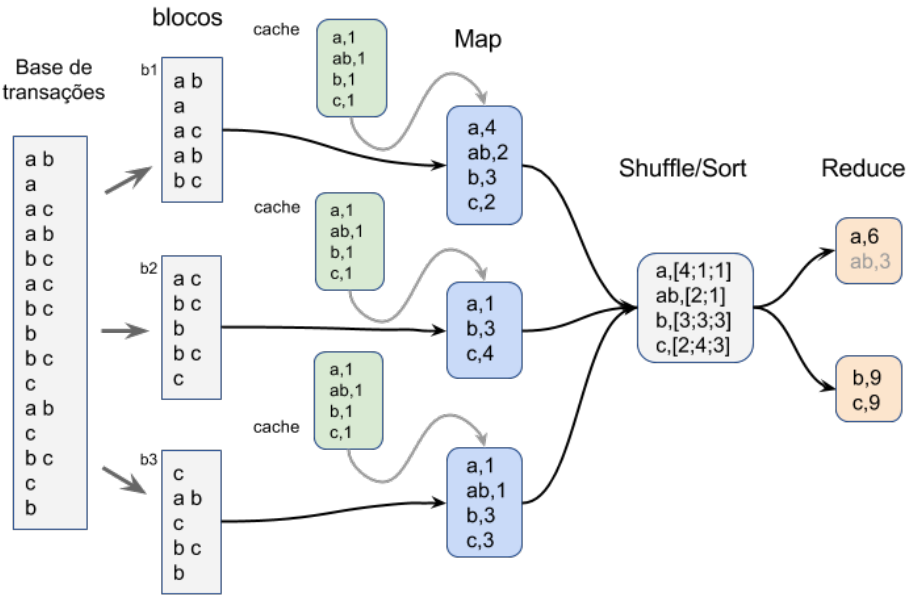


Figura 12 Exemplo genérico da Fase 2 para um algoritmo de duas fases com suporte mínimo igual a 6 em relação a toda base de transações (ou 40%).

os experimentos o algoritmo de 2 fases se sobressai diante da proposta de k fases e, quando o valor do contador de suporte é muito baixo, a discrepância ainda é maior, pois mais *itemsets* são gerados e mais iterações o algoritmo de k fases necessita para encontrar todos os conjuntos de itens frequentes.

Farzanyar e Cercone (2013b) - Efficient Mining of Frequent Itemsets in Social Network Data Based on MapReduce Framework

Farzanyar e Cercone (2013b) apresentam uma melhoria para a abordagem anterior, denominada *IMRApriori* (*Improved Map Reduce Apriori*). Os autores otimizaram o tempo de execução por meio de uma estratégia de podas, diminuindo a quantidade de *itemsets* trafegando pela rede do cluster. A ideia é identificar quais são os *itemsets* parciais não frequentes, os

INS-itemset (*Insignificant itemsets*), na função Reduce da Fase 1. Para isso foi definido que um *INS-itemset* é aquele item que foi emitido por poucas funções Map como *itemset* frequente.

De acordo com a Propriedade 1 introduzida por Farzanyar e Cercone (2013b), o contador de suporte máximo para um *itemset* não frequente oriundo do bloco S_i de tamanho D_i é igual a $(s \times D_i) - 1$, onde s é o suporte mínimo definido previamente. A partir dessa propriedade, é possível estimar o contador de suporte global de cada *itemset* X pela Equação 5.

$$X.suporteGlobal \approx (X.suporteParcial + (((s \times D_i) - 1) \times (M - N_X))) \quad (5)$$

M é a quantidade total de Maps que executaram na Fase 1. Esse valor pode ser obtido previamente por meio da contagem de blocos oriundos da base de dados de entrada que, consequentemente, define a quantidade de funções Map. N_X é a quantidade de funções Maps que emitiu o *itemset* X e pode ser obtida calculando o tamanho da lista de valores do *itemset* recebida na função Reduce. O contador de suporte parcial do item X ($X.suporteParcial$) é obtido ao incrementar a lista de contadores de suportes parciais emitida pelas funções Maps. Portanto, se o contador de suporte global de um *itemset* X for maior ou igual ao suporte mínimo (Fórmula 6), X é, então, considerado parcialmente frequente. O *itemset* é salvo no HDFS para efetuar a contagem global na Fase 2. Do contrário X é considerado um *INS-itemset* e descartado. Como exemplo, o *itemset* "ab" da Figura 11 seria descartado na função Reduce. Ao estimar seu contador de suporte global por meio da Equação 5, o resultado é 5, menor do que 6, o valor do

suporte mínimo para toda a base de transações. Portanto, “ab” não precisa ser contado na Fase 2.

$$X.suporteGlobal \geq s \times D \tag{6}$$

Os autores compararam o tempo de execução do *IMRApriori* com o *MRApriori* para diferentes valores de suporte. Em todos os experimentos, o *IMRApriori* executou mais rápido do que o *MRApriori*. Quando o valor do suporte diminui, a diferença de tempo ainda é maior, pois gera-se mais conjuntos de itens candidatos. O *IMRApriori* elimina mais *itemsets* não frequentes por meio da estratégia de podas, enquanto que o *MRApriori* processa todos os itens e os envia para a Fase 2, consumindo maior tempo para a execução.

Farzanyar e Cercone (2013a) - Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce -Based Approach

Em um outro trabalho, os próprios autores do *IMRApriori* propuseram uma extensão desse algoritmo que otimiza ainda mais seu desempenho (FARZANYAR; CERCONE, 2013a). Neste trabalho essa abordagem é denotada como *IMRAprioriAcc*. Nos experimentos, foi notado que a quantidade de *itemsets* parcialmente frequentes enviados para a Fase 2 ainda é muito grande e pode ser melhorada. A ideia é identificar de qual Map e de qual bloco um *itemset* parcialmente frequente veio, para que na Fase 2 o contador de suporte do item seja calculado somente nos blocos que não foi frequente. Dessa forma, não é necessário contar novamente o *itemset* nos blocos em que foi frequente na Fase 1. Por exemplo, na Figura 11 o conjunto de dados de entrada é dividido em 3 blocos b_1, b_2 e b_3 e cada bloco

é atribuído para uma função Map. O item "a" foi enviado pelo Map_1 para o Reduce, então a função Reduce tem o contador de suporte do item "a" no bloco b_1. Portanto, é necessário contar o suporte do item "a" apenas em b_2 e b_3.

As funções Reduce da Fase 1 enviam os *itemsets* parcialmente frequentes para M partições diferentes. O Algoritmo 8 apresenta o pseudocódigo dessa etapa. Na linha 9 do algoritmo é efetuado o cálculo para estimar o contador de suporte do *itemset*. Se maior ou igual ao suporte mínimo verifica quais Maps (blocos ou partições) o *itemset* não foi frequente (linha 10 e 11, Algoritmo 8) e o envia para um arquivo referente à partição que não foi frequente (linha 12). Na Fase 2, as funções Maps são semelhantes às apresentadas no Algoritmo 6, porém L_p contém apenas os itemsets frequentes parciais referentes à partição que lhe foi atribuída.

Dessa forma, o conjunto de todos os itens parcialmente frequentes encontrados na Fase 1 é dado pela Equação 7.

$$L_{parcial} = (L_{parcial_1}, L_{parcial_2}, ...L_{parcial_M}) \tag{7}$$

$L_{parcial_1}$ contém os *itemsets* encontrados no bloco 1; $L_{parcial_2}$ contém os *itemsets* encontrados no bloco 2 e assim sucessivamente. Cada partição gerada será atribuída para um Map específico na Fase 2 para efetuar a contagem nos blocos.

Como resultado, o número de *itemsets* enviados das funções Map para a função Reduce na Fase 2 é menor do que o *IMRApriori*. Logo, o tempo de execução do algoritmo para encontrar todos os *itemsets* frequentes também é menor. Os experimentos compararam o tempo de execução da abordagem *IMRAprioriAcc* com o *IMRApriori* em relação ao valor do su-

Algoritmo 8: Função Reduce - Fase 1 do algoritmo de duas fases IMRAprioriAcc, adaptado de Farzanyar e Cercone (2013a)

Input: pares <chave,lista(valor)> provenientes das funções Maps da Fase 1

Output: pares <chave2,sup_count>, onde chave2 pode ser um itemset frequente global ou parcial

```
1  Function Reduce(chave, lista(valor))
2      foreach value v_i in lista(valor) do
3          N_item += 1;            // o número de Maps que emitiu o itemset
4          sup_count += v_i;
5          W[i] = 1;               // W[M] é um vetor para todos os Maps
6      end
7      if N_item == M then
           // itemset é frequente em todos os Maps
8          Output_g(key,sup_count);   // envia para partição global
9      else if (sup_count + (((s*D_i) - 1) * (M - N_item)) >= s*D)
       then
10         foreach i in M do
11             if W[i]==0 then
12                 Output_i(key,sup_count);     // para a partição i
13             endif
14         end
15     endif
16 end
```

porte mínimo. Em todos os testes observou-se que a nova estratégia obtém o tempo de execução menor do que o *IMRApriori*. Ao diminuir o valor do suporte mínimo, a diferença de tempo de execução ainda é mais discrepante, pois mais *itemsets* são gerados. E enquanto o *IMRApriori* efetua as suas podas na Fase 1, a nova abordagem consegue reduzir o fluxo de dados na Fase 2, de maneira eficiente.

2.5.4 Algoritmo de k fases MapReduce

Em Li et al. (2012), Lin, Lee e Hsueh (2012), Yang, Liu e Fu (2010) e Zhou e Huang (2014), as estratégias necessitam de k fases MapReduce para encontrar todos os conjuntos de itens frequentes. A k-ésima iteração MapReduce é responsável por encontrar o conjunto de itens de tamanho k a partir do resultado da iteração anterior, que gerou o conjunto de itens de tamanho $k-1$. Essas abordagens, geralmente, necessitam de pelo menos duas aplicações MapReduce diferentes. Uma para extrair os itens individuais da base de transações de entrada, produzindo os 1-*itemsets*. E uma segunda aplicação para encontrar todos os demais *k-itemsets*, por meio de um processo iterativo, em que cada chamada gera o conjunto de itens frequentes de tamanho diferente. Isto é, na iteração 2 (ou fase 2) geram-se os *itemsets* de tamanho 2 ($k = 2$), na fase 3 geram-se os *itemsets* de tamanho 3 ($k = 3$), e assim sucessivamente até que não seja mais possível gerar *itemsets* frequentes.

Lin, Lee e Hsueh (2012) - Apriori-based Frequent Itemset Mining Algorithms on MapReduce

Os Algoritmos 9, 10 e 11 apresentam implementações genéricas baseadas na abordagem de Lin, Lee e Hsueh (2012), denominada SPC (*Single Pass Counting*). Os Algoritmos 9 e 10 encontram todos os *itemsets* de tamanho 1, que sejam frequentes. As funções Map extraem e contam o suporte local de cada 1-*itemset* candidato, gerando a saída no formato <*itemset*,1> (linhas 3 e 4, Algoritmo 9). Esses pares são agrupados pelo Hadoop por meios das chaves, gerando uma lista de valores para cada chave distinta. Nas funções Reduce, além de efetuar a contagem do suporte global para cada item (linha 5, Algoritmo 10), também é realizada a poda dos itens não frequentes, de acordo com o suporte mínimo definido (linhas 7 e 8, Algoritmo 10) . A saída é adicionada ao sistema de cache distribuído do Hadoop para ser utilizada na próxima fase. Um exemplo dessa fase é apresentado na Figura 13.

Algoritmo 9: Função Map - Fase 1 do algoritmo de k fases, adaptado de Yahya, Hegazy e Ezat (2012)

```
   Input: Bloco de transações S_i, uma por linha
   Output: pares <chave,1>, onde chave é 1-itemset
1  foreach transaction t in S_i do
2  |  Function Map(line offset, t)
3  |  |  foreach item i in t do
4  |  |  |  Output(i,1);
5  |  |  end
6  |  end
7  end
```

O Algoritmo 11 da Fase 2, por meio de um processo iterativo, encontra todos os demais *itemsets*. Cada iteração encontra os *itemsets* de

Algoritmo 10: Função Reduce - Fase 1 do algoritmo de k fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: pares <chave2,lista(valor2)>
Output: pares <chave3,valor3>, onde chave3 é um itemset frequente e valor3 é seu contador de suporte

```
1  min_sup = read min_sup do DistributedCache;
2  Function Reduce(key2, list(value2))
3  |  soma = 0;
4  |  while valor2.hasNext() do
5  |  |  soma += valor2.next();
6  |  end
7  |  if soma >= min_sup then
8  |  |  Output(chave2,soma);
9  |  endif
10 end
```

tamanho k produzidos a partir dos $(k-1)$-*itemsets* gerados na iteração anterior. É preciso obter a saída da fase anterior do cache distribuído, para gerar os *k-itemsets* (linhas 1 e 2, Algoritmo 11). Um exemplo dessa fase é apresentado na Figura 14. A função Map tem o papel de verificar se os *itemsets* em C_k ocorrem nas transações. Cada item candidato que possuir ocorrência confirmada nas transações é enviado para o Reduce com o contador de suporte igual 1 (linhas 6 e 7, Algoritmo 11). A função Reduce é semelhante à da Fase 1, responsável por contar o suporte global e efetuar a poda dos itens não frequentes. A saída dessa etapa são os *itemsets* frequentes como chave e seu contador de suporte como valor. Os novos dados são adicionados ao cache distribuído do Hadoop para serem utilizados na fase (ou iteração) seguinte.

Lin, Lee e Hsueh (2012) ainda apresentam outras duas variações do algoritmo SPC, denominadas FPC (*Fixed Passes Combined-counting*) e DPC (*Dynamic Passes Combined-counting*). Para essas variações, uma fase

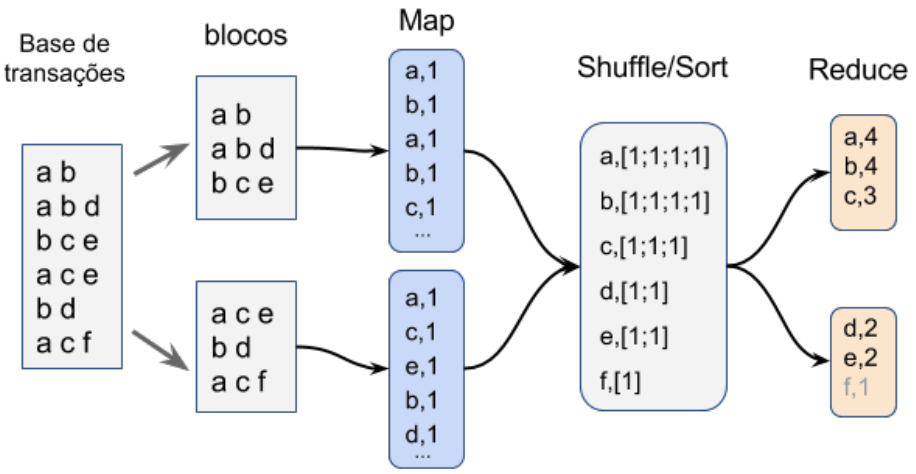


Figura 13 Exemplo genérico da Fase 1 para um algoritmo de k fases com suporte mínimo igual a 1 ($\approx 17\%$).

extra é necessária para produzir os *itemsets* de tamanho 2. Dessa forma, a Fase 1 encontra os 1-*itemsets*, a Fase 2 encontra os 2-*itemsets* e a Fase 3 encontra todos os demais *itemsets*, por meio de um processo iterativo. Essas abordagens produzem mais de um *itemset* em uma mesma iteração. A vantagem é que o tempo gasto para encontrar todos os *itemset* frequentes é menor, devido ao número de iterações reduzidas.

O FPC gera uma quantidade fixa, definida previamente, de *itemsets*, a cada iteração (no caso, 3). Ou seja, a partir do conjunto de itens recebido da Fase 2, a função Map gera os *itemsets* de tamanho k, $k+1$ e $k+2$. Em seguida verifica-se a existência, de cada *itemset* gerado, nas transações para enviá-los ao Reduce. A função Reduce, como nas demais implementações, apenas conta globalmente o suporte de cada *itemset*, poda e salva os itens frequentes no HDFS.

Já para o DPC, a quantidade de *itemsets* de tamanhos diferentes gerados em cada iteração é obtida de modo dinâmico para o melhor balan-

Algoritmo 11: Função Map - Fase 2 do algoritmo de k fases, adaptado de Yahya, Hegazy e Ezat (2012)

Input: Bloco de transações S_i, uma por linha; L_{k-1}, itemsets produzidos na iteração anterior
Output: pares <chave,1>, onde chave é um k-itemset

```
1  read L_{k-1} do DistributedCache;
2  C_k = aprioriGen(L_{k-1});            // gera itemsets candidatos
3  foreach transaction t in S_i do
4  |  Function Map(line offset, t)
5  |  |  C_t = subSet(C_k,t);             // verifica os candidatos
6  |  |  foreach candidate c in C_t do
7  |  |  |  Output(c,1);
8  |  |  end
9  |  end
10 end
```

ceamento entre a quantidade de *itemsets* gerados com a quantidade total de iterações. O Algoritmo 12 apresenta o pseudocódigo do DPC. Uma variável, denotada como *ct* (*candidate threshold*), é responsável por definir quantos conjuntos de itens de tamanhos diferentes são gerados a cada iteração do algoritmo (linha 2, Algoritmo 12). Ela é calculada utilizando a informação do tempo gasto pela fase anterior e o tamanho do conjunto de *itemsets* obtido do cache distribuído, como apresentado na Equação 8.

$$ct = \alpha \times |L_{k-1}| \tag{8}$$

A variável α é igual a 1 se o tempo de execução da fase anterior é longo (> 60s), caso contrário α é definido como 1,2. $|L_{k-1}|$ é a quantidade de *itemsets* frequentes produzidos na iteração anterior. Como apresentado nas linhas de 4 a 7 do Algoritmo 12, a abordagem produzirá itemsets candidatos até que o tamanho total do conjunto de candidatos alcance o valor de *ct*. Em

Algoritmo 12: Função Map - Fase 3 do algoritmo de k fases DPC, adaptado de Lin, Lee e Hsueh (2012)

```
   Input: Bloco de transações S_i , uma por linha; L_{k-1}
   Output: pares <chave,1>, onde chave é um k-itemset
 1 read L_{k-1} do DistributedCache;
 2 ct = α * |L_{k-1}|;
 3 C_set = C_k = aprioriGen(L_{k-1}); k = k+1;
 4 while |C_set| <= ct do
 5 |   C_set += C_k = aprioriGen(C_{k-1}); k = k+1;
 6 |   C_set += C_k = aprioriGen(C_{k-1}); k = k+1;
 7 end
 8 foreach transaction t in S_i do
 9 |   Function Map(line offset, t)
10 |   |   C_t = subSet(C_set,t));
11 |   |   foreach candidate c in C_t do
12 |   |   |   Output(c,1);
13 |   |   end
14 |   end
15 end
```

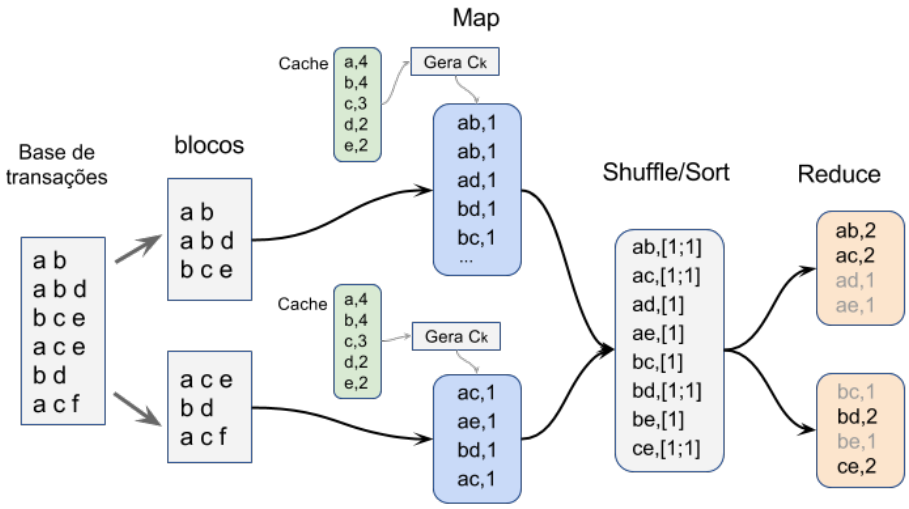


Figura 14 Exemplo genérico da Fase 2 para um algoritmo de k fases com suporte mínimo igual a 1 ($\approx 17\%$).

seguida os itemsets candidatos são verificados por meio da função *subSet()* (linha 10) e enviados para a função Reduce (linha 12).

Para avaliar o desempenho das implementações, os autores contabilizaram o tempo de execução dos algoritmos para diferentes bases de dados, variando o valor do suporte mínimo. Quando o valor do suporte é pequeno, tanto o DPC quanto o FPC apresentam menor tempo de execução do que o SPC. O DPC também leva vantagem sobre o FPC. Porém, à medida que o valor do suporte mínimo aumenta as 3 variações tendem a levar o mesmo tempo de execução, uma vez que, as fases 1 e 2 são semelhantes para as 3 variações e, quanto maior o valor do suporte, menor o número de iterações. Em alguns casos, para valores pequenos de suporte, o FPC necessitou de muito mais tempo que o SPC e o DPC para encontrar todos os *itemsets*. Os autores atribuem o motivo à quantidade de *itemsets* falsos positivos gerados de tamanho 3 a 5 pelo FPC. Nos demais casos foi possível notar a vantagem

que o DPC possui em relação aos demais no quesito tempo de execução, devido à eficiência de sua estratégia de gerar uma quantidade dinâmica de *itemsets* em uma mesma iteração.

Yang, Liu e Fu (2010) - MapReduce as a Programming Model for Association Rules Algorithm on Hadoop

Uma abordagem semelhante ao SPC (LIN; LEE; HSUEH, 2012) já tinha sido apresentada por Yang, Liu e Fu (2010). A diferença é que a combinação entre os *itemsets* é realizada previamente, antes de iniciar a próxima iteração. Em outras palavras, no momento em que uma iteração finaliza, gera-se o conjunto de itens candidato a partir dos *itemsets* frequentes produzidos por essa iteração. Esse conjunto de itens candidato será verificado e contado na próxima iteração MapReduce para produzir novos *itemsets* frequentes.

Os autores avaliaram o desempenho dessa implementação por meio do *SpeedUp*, que mede a escalabilidade do algoritmo em relação à quantidade de nós executando-o em paralelo. Foi constatado que o *SpeedUp* aumenta à medida que se acrescenta mais nós no cluster, porém não é um aumento linear ideal. Fatores como comunicação na rede, custo de iniciação das aplicações e distribuição de dados pelo cluster influenciam diretamente na escalabilidade do algoritmo.

Li et al. (2012) - Parallel Implementation of Apriori Algorithm Based on MapReduce

Outra abordagem utilizando k fases MapReduce é apresentada em (LI et al., 2012), na qual os autores denotaram como *PApriori* (*Parallel*

Apriori). Ela compartilha pontos em comum com a proposta de Yang, Liu e Fu (2010) e com o algoritmo SPC (LIN; LEE; HSUEH, 2012). Há uma fase MapReduce somente para encontrar os *itemsets* de tamanho 1, assim como no SPC. Entretanto, os *itemsets* candidatos são produzidos antes de iniciar a próxima iteração, como a abordagem de Yang, Liu e Fu (2010). Além disso, é possível definir a quantidade de iterações MapReduce a ser executada. Isso permite que o algoritmo gere apenas a quantidade de *itemsets* suficientes para o usuário. O processo iterativo se encerra quando a quantidade de iterações especificada é atingida ou quando não é mais possível gerar novos *itemsets* frequentes.

Para avaliar o desempenho do algoritmo, os autores utilizaram 3 métricas de avaliação para algoritmos paralelos e distribuídas: *ScaleUp*, *SizeUp* e *SpeedUp*. Essas três métricas são melhor discorridas na Seção 5.1.2. Ao invés de fazer uma avaliação utilizando o número de máquinas, os autores contabilizam o número de núcleos de processamento para cada métrica. Em relação ao *ScaleUp*, foi constatada a ocorrência de uma curva de queda quando a quantidade de dados e o número de núcleos aumentam. Porém, para conjunto de dados menores essa medida atinge valores de 78% a 80%. Na avaliação utilizando o *SizeUp* foi constatado que quando a quantidade de núcleos é pequena (entre 4 e 8) há pouca variação na acurácia, porém o *SizeUp* diminui consideravelmente quando a quantidade de núcleos é maior perante a variação da quantidade de dados. Isso reflete o gargalo de comunicação, distribuição de dados e distribuição de carga que ocorrem quando se aumenta o número de máquinas no cluster. Por fim, com a medida *SpeedUp* os autores constataram o que já havia sido mostrado com o *SizeUp*, quando menor a quantidade de dados e maior a quantidade de núcleos, maior é o

gargalo no cluster, prejudicando o desempenho do algoritmo. Com essas informações concluiu-se que o *PApriori* é mais eficiente para base de dados maiores.

Zhou e Huang (2014) - An Improved Parallel Association Rules Algorithm Based on MapReduce Framework for Big Data

Mais recentemente, Zhou e Huang (2014) propuseram uma abordagem denominada CPA (*Complete Parallel Apriori*) que, além de efetuar a contagem dos *itemsets*, também efetua a combinação dos itens (geração dos candidatos) utilizando o modelo MapReduce. A abordagem é semelhante à de Li et al. (2012) e ao algoritmo SPC (LIN; LEE; HSUEH, 2012). A diferença é que são executadas funções Map e Reduce para a geração dos candidatos, enquanto que nas propostas apresentadas até então, esse processo era todo feito apenas nas funções Map ou de alguma outra forma externa ao MapReduce.

A ideia central do algoritmo é empregar um processo iterativo de k fases com duas execuções MapReduce distintas em cada fase. O primeiro conjunto de funções é responsável pela geração dos candidatos e o segundo MapReduce efetua a contagem do suporte e a poda dos candidatos não frequentes. A saída de cada fase é utilizada como entrada para a fase posterior. Logo, é possível afirmar que essa abordagem, em vez de k fases, emprega um algoritmo de $2k$ fases, pois a cada iteração são necessárias duas execuções MapReduce para efetuar as combinações e a contagem dos conjuntos de itens. O processo de geração de candidatos utilizando MapReduce é ilustrado na Figura 15.

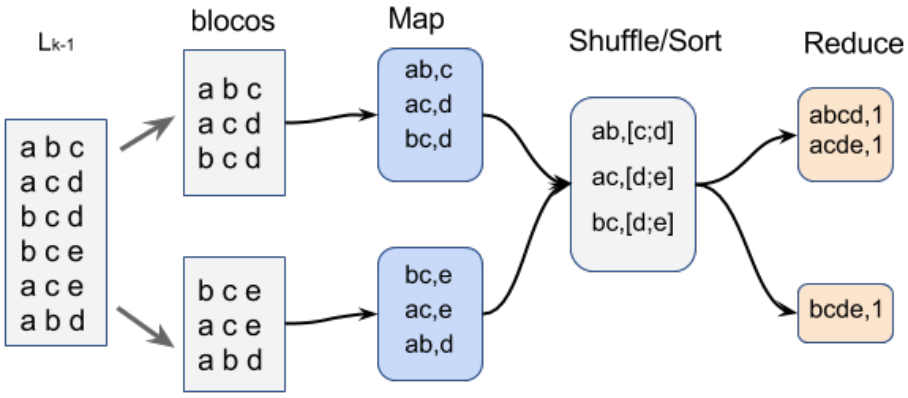


Figura 15 MapReduce para geração dos candidatos, adaptado de Zhou e Huang (2014).

Os Algoritmos 13 e 14 apresentam o pseudocódigo para o processo de geração de *itemsets* candidatos. A função Map recebe como entrada os *itemsets* frequentes gerados na iteração anterior e separa o prefixo do sufixo para cada *itemset* (linhas 3 e 4, Algoritmo 13). Por exemplo, o *itemset* "a b c" possui o prefixo "a b", que será a chave de saída, e o sufixo "c" o valor, resultando no par "<ab,c>". Na etapa de Shuffle/Sort os dados são agrupados e ordenados pela chave produzindo uma lista de valores para uma mesma chave, como ilustrado na Figura 15. A função Reduce, Algoritmo 14 é, então, responsável por efetuar as combinações desses valores com o prefixo na chave para produzir o novo conjunto de itens candidatos. Esse novo conjunto terá seu contador de suporte calculado em uma outra execução MapReduce para produzir os *itemsets* frequentes. O ciclo se repete até que todos os conjuntos de itens frequentes sejam encontrados.

Para avaliar o desempenho dessa nova abordagem, os autores a compararam com uma implementação de k fases que não utiliza o MapReduce para gerar os itens candidatos. Os resultados mostraram que as duas im-

Algoritmo 13: Função Map - geração de candidatos do algoritmo CPA de k fases, adaptado de Zhou e Huang (2014)

Input: L_{k-1}, itemsets frequentes da iteração anterior, um por linha
Output: pares <chave,valor>, onde a chave é o prefixo (os primeiros $k-2$ itens) e valor é o sufixo (os $k-1^{o}$ item)

```
1 foreach itemset i in L_{k-1} do
2 |   Function Map(line offset, i)
3 |   |   prefix = i.substring(1,k-2);
4 |   |   suffix = i.substring(k-1);
5 |   |   Output(prefix,suffix);
6 |   end
7 end
```

Algoritmo 14: Função Reduce - geração de candidatos do algoritmo CPA de k fases, adaptado de Zhou e Huang (2014)

Input: pares <chave,lista(valor)> emitidos dos Maps
Output: pares <chave2,1>, onde chave2 é um k-itemset candidato

```
1 Function Reduce(key, list(value))
2 |   C_k = combine(key,list(value));
3 |   foreach itemset c in C_k do
4 |   |   Output(c,1);
5 |   end
6 end
```

plementações apresentam desempenho semelhante para dados de entrada pequenos. Porém quando a quantidade de dados é maior e há variação no valor do suporte mínimo, a nova abordagem se sobressai. Em alguns casos onde o suporte mínimo é muito baixo, a implementação tradicional estourou o limite de tempo que uma tarefa no *ApplicationMaster* pode ficar sem reportar à máquina mestre.

2.5.5 Síntese dos trabalhos

A Tabela 4 apresenta uma síntese esquemática das principais características de cada abordagem de implementação do algoritmo *Apriori* descritas nessa seção. A coluna "Estratégia" destaca, de modo sintetizado, como os autores aplicaram o paradigma MapReduce em suas abordagens. A coluna "Avaliação" destaca os principais métodos utilizados pelos autores para medir o desempenho de seus algoritmos.

Dentre as abordagens descritas, foram extraídas três propostas principais que, ou são versões aprimoradas de outros trabalhos, ou são abordagens com maior riqueza de detalhes englobando as principais características de algumas das outras abordagens. Esses três trabalhos, Farzanyar e Cercone (2013a), Lin, Lee e Hsueh (2012) e Zhou e Huang (2014) foram utilizados no processo de comparação minuciosa para avaliar seus comportamentos sob diversas perspectivas, tais como, quantidade de itens distintos na base de dados, quantidade de transações, quantidade de itens por transações e varição do suporte mínimo. O trabalho apresentado em (LIN; LEE; HSUEH, 2012), denominado DPC, implementa um método que reduz a quantidade de iterações através de uma abordagem dinâmica, otimizando o tempo de execução perante às demais abordagens de k fases. O trabalho apresentado em

Tabela 4 Principais características das implementações

PROPOSTA	FASES	ESTRATÉGIA	AVALIAÇÃO
Li e Zhang (2011)	1	Geração dos *itemsets* nas Map; poda nos Reduce	*SpeedUp*
Yahya, Hegazy e Ezat (2012)	2	Aplica-se o *Apriori* na fase 1; contagem e poda na fase 2	*SizeUp*
Farzanyar e Cercone (2013b)	2	Melhoria do Yahya, Hegazy e Ezat (2012) com um sistema de podas na Fase 1	Tempo de execução, variando o *min_sup* e *SizeUp*
Farzanyar e Cercone (2013a)	2	Melhoria do Farzanyar e Cercone (2013b) com redução do fluxo de dados na Fase 2	Tempo de execução, variando o *min_sup* e *SizeUp*
Yang, Liu e Fu (2010)	k	Contagem e poda dos *itemsets* em cada fase	*SpeedUp*
Li et al. (2012)	k	Contagem e poda dos *itemsets* em cada fase	*ScaleUp*, *SizeUp* e *SpeedUp*
Lin, Lee e Hsueh (2012)	$< k$	Número de iterações definidas dinamicamente	Tempo de execução, variando o *min_sup*
Zhou e Huang (2014)	$2k$	Um MapReduce para geração e outro para contagem e poda em cada iteração	Tempo de execução e *SpeedUp*, variando o *min_sup*

(FARZANYAR; CERCONE, 2013a), denominado aqui de *IMRAprioriAcc* é uma melhoria de outros dois trabalhos que utilizam algoritmos de apenas 2 fases e seu desempenho, segundo os autores, se sobressai aos algoritmos de k fases convencionais. O trabalho apresentado em (ZHOU; HUANG, 2014), denominado CPA, apesar de executar duas funções MapReduce por iteração, tende a se sobressair perante as demais abordagens em situações específicas.

Essas três abordagens foram reimplementadas tanto no Hadoop, quanto no Spark. Logo, ao comparar essas implementações para as mesmas bases de dados, em um mesmo cluster, sob os diversos aspectos, foi possível extrair as vantagens e desvantagens de cada abordagem, identificando em quais situações cada qual se sobressai. Com essas informações foi possível formular um framework de recomendação do algoritmo mais adequado para cada situação.

2.6 Regras de associação para resolução de entidades

Um dos objetivos deste trabalho é implementar uma alternativa utilizando o Hadoop-MapReduce e Spark para o algoritmo de resolução de entidades aplicado para o problema de classificação de ofertas de produtos (OLIVEIRA; PEREIRA, 2014; OLIVEIRA; PEREIRA, 2017). Apesar de ser um problema de classificação, o núcleo da abordagem utiliza regras de associação para predizer a que entidade as descrições das ofertas de produtos se referem. Como já destacado nas subseções anteriores, há diversas abordagens de mineração de regras de associação utilizando o Hadoop-MapReduce e Spark para o fim de obter tempo de execução aceitável diante de uma quantidade massiva de dados. Em relação ao problema

específico de classificação de ofertas de produtos, é possível mapeá-lo para o arcabouço Hadoop-MapReduce e Spark para possibilitar a execução em tempo adequado para grandes bases de dados.

A estratégia geral para a classificação de produtos é produzir regras utilizando um conjunto de atributos (*features*) $\{f_1, f_2, ..., f_m\}$ extraídas dos elementos textuais de cada descrição das ofertas de produtos de treinamento. Em seguida, utilizar essas regras geradas para classificar novos conjuntos de produtos de teste.

Mais formalmente, esse processo pode ser descrito da seguinte forma: dado um conjunto de oferta de produtos $P = \{p_1, p_2, ..., p_{|P|}\}$, cada oferta de produto p_i possui uma lista de *features*, extraídas da suas descrições. Cada produto representa uma classe c e o conjunto de classes é dado por $C = \{c_1, c_2, ..., c_{|C|}\}$. O objetivo é produzir uma função de classificação que utiliza regras de associação para mapear cada oferta de produto, p_i, em uma das classes pré-definidas do conjunto C.

Essas regras devem possuir 100% de confiança para identificar unicamente cada classe. As regras são da seguinte forma $X \Rightarrow c_i$, onde $X \subseteq \{f_1, f_2, ..., f_m\}$ e $c_i \in C$. Por exemplo, a descrição da oferta de produto *Officejet J3680 All-in-One Printer, Fax, Scanner, Copier, HEWCB071A*, que pertence à classe c_1, pode produzir duas regras de associação $\{J3680\} \Rightarrow c_1$ e $\{HEWCB071A\} \Rightarrow c_1$, que indicam que o conjunto de *tokens* $\{J3680\}$ e o conjunto $\{HEWCB071A\}$, ambos identificam unicamente as ofertas de produtos da classe c_1.

Pode ocorrer casos em que algumas instâncias não podem ser classificadas pelas regras. Tais casos ocorrem quando nenhuma classe é atribuída à instância de teste, pois nenhum *token* ou uma combinação de *tokens* da ins-

tância correspondeu a alguma regra do modelo. Da mesma forma, quando os *tokens* das instâncias de teste correspondem a duas ou mais classes do modelo nas mesmas proporções, sendo impossível distinguir a classe majoritária, não ocorre a classificação. Para contornar, é utilizada uma função de similaridade, por exemplo Cosseno, para descobrir a classe que melhor se associa com a instância.

O Cosseno é uma função de similaridade baseada em *tokens* utilizada para calcular a semelhança entre duas *strings* utilizando o modelo vetorial (BAEZA-YATES; RIBEIRO-NETO, 2011, p 77). Cada *string* j é composta de *tokens* e cada *token* i possui um peso $w_{i,j}$ na *string*. O peso $w_{i,j}$ pode ser calculado pela frequência do termo e a frequência inversa do termo no documento, denominado TF-IDF (*term-frequency x inverse document frequency*) (BAEZA-YATES; RIBEIRO-NETO, 2011, p 72), como ilustrado na Equação 9

$$w_{i,j} = (1 + log f_{i,j}) * log \frac{|S|}{n_i} \tag{9}$$

onde $f_{i,j}$ é a frequência do *token* k_i na *string* s_j, $|S|$ é o total de *strings* e n_i é o número de *strings* em S que o *token* k_i ocorre. Uma *string* é representada como um vetor de pesos $\vec{s}_j = \{w_{1,j}, w_{2,j}, ..., w_{t,j}\}$. Cada peso $w_{i,j}$ representa a importância do *token* k_i na *string* s_j. Logo, a similaridade entre duas *strings* s_i e s_j é dada pela Equação 10.

$$sim(s_i, s_j) = \frac{\Sigma_{l=1}^{t} w_{l,i} \times w_{l,j}}{\sqrt{\Sigma_{l=1}^{t} w_{l,i}^2} \times \sqrt{\Sigma_{l=1}^{t} w_{l,j}^2}} \tag{10}$$

Para o presente trabalho foi utilizado o Hadoop-MapReduce e o Spark para adaptar o algoritmo de resolução de entidades para a computação

distribuída. Visou-se otimizar o tempo de execução para processar maiores volumes de dados, pois como relatado em Oliveira e Pereira (2014), Oliveira e Pereira (2017), para bases de dados maiores, com mais de 300.000 ofertas de produtos o algoritmo leva tempo demasiadamente alto para apresentar os resultados.

3 IMPLEMENTAÇÕES DO APRIORI NO SPARK

Uma das implementações do algoritmo Apriori para o *framework* Spark encontradas na literatura, o YAFIM (*Yet Another Frequent Itemset Mining*) (QIU et al., 2014) é uma adaptação direta de um típico algoritmo de k fases do Hadoop-MapReduce para o Spark, como o SPC para o Hadoop-MapReduce, por exemplo. O algoritmo é constituído de duas etapas distintas, uma que encontra todos os *itemsets* frequentes de tamanho 1 e outra que, por meio de um processo iterativo, encontra todos os demais *itemsets* frequentes. Devido ao fato de o Spark manter os dados em memória, por meio da utilização dos RDDs, o YAFIM tende a ser mais eficiente do que as implementações iterativas utilizando o Hadoop-MapReduce, já que evita efetuar operações de I/O no disco a cada iteração.

R-Apriori (RATHEE; KAUL; KASHYAP, 2015) é uma outra implementação do Apriori para Spark, similar ao YAFIM no sentido de possuir k iterações. A diferença básica é que na segunda iteração, o R-Apriori utiliza um esquema de Bloom Filter (BLOOM, 1970) para gerar os *2-itemsets* candidatos, combinando itens da própria transação, em vez de usar uma Hashtree.

Ambas essas abordagens são semelhantes a adaptações do algoritmo SPC (LIN; LEE; HSUEH, 2012) que, por sua vez, apresenta comportamento inferior ao algoritmo DPC em relação ao tempo de execução. Assim, optou-se por não incluí-las na comparação dos algoritmos.

Neste trabalho, foi realizada uma adaptação das abordagens IMR-AprioriAcc, DPC e CPA propostas para o Hadoop-MapReduce para o *framework* Spark, a fim de averiguar se há melhora significativa no *framework* que mantém os dados em memória. Os algoritmos seguem as mesmas estra-

tégias de suas implementações originais no Hadoop-MapReduce, exceto que os dados são mantidos em memória durante o processo iterativo, por meio dos RDDs. Essas implementações são detalhadas nas seções a seguir.

3.1 Adaptações das abordagens Hadoop-MapReduce para o Spark

Nas subseções seguintes são apresentadas as adaptações das abordagens IMRAprioriAcc, DPC e CPA para executarem no Spark.

3.1.1 IMRAprioriAcc Spark

Na implementação do IMRAprioriAcc no Spark, foi utilizada a função *mapPartitionWithIndex()*, nas duas fases, a qual permite identificar qual bloco está sendo processado no momento, tornando possível identificar o bloco de origem de um determinado *itemset*. O fluxo de execução da Fase 1 do IMRAprioriAcc no Spark é apresentado na Figura 16. Aplica-se a função *mapPartitionWithIndex()* na base de transações, com o ID e o conteúdo do bloco como parâmetros. Geram-se então todos os *itemsets* parcialmente frequentes. Em seguida, executa-se a função *mapToPair()*, com o *itemset* e sua lista de contadores de suporte como parâmetro, para separar os *itemsets* globalmente frequentes dos parcialmente frequentes, os quais ainda necessitam de contagem na Fase 2.

Na Fase 2, o suporte dos *itemsets* parcialmente frequentes são contados por meio da função *mapPartitionWithIndex()*, a qual os conta somente nas partições em que não foram parcialmente frequentes. Além do bloco com as transações, essa função recupera a lista de *itemsets* que devem ser contados na partição específica. Em seguida, aplica-se a função *mapToPair()*

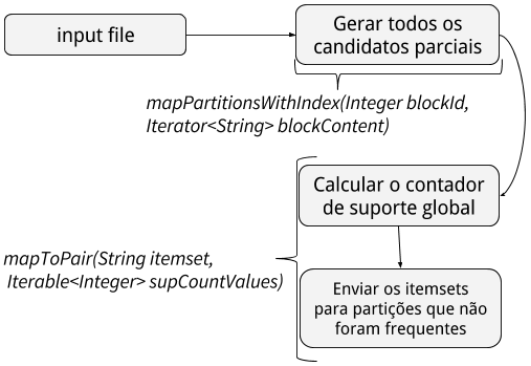


Figura 16 Fluxo da Fase 1 - IMRAprioriAcc Spark

para contar globalmente cada *itemset*. O fluxo dessa execução é ilustrado na Figura 17.

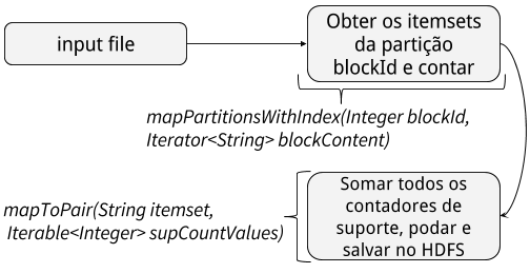


Figura 17 Fluxo da Fase 2 - IMRAprioriAcc Spark

3.1.2 DPC Spark

O fluxo de execução do DPC no Spark para a geração dos *itemsets* de tamanho 1 e 2 é apresentado na Figura 18. A partir dos *itemsets* de tamanho 3, é realizado o processo iterativo e dinâmico para produzir os demais *itemsets* frequentes, como apresentado na Figura 19. O processo dinâmico de geração de *itemsets* é semelhante ao implementado na versão Hadoop-MapReduce do algoritmo.

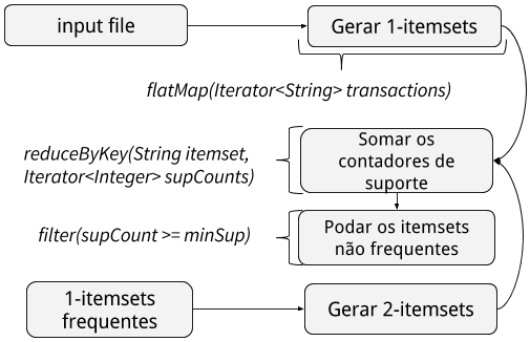


Figura 18 Fluxo das Fases 1 e 2 - DPC Spark

3.1.3 CPA Spark

O CPA foi implementado no Spark seguindo a mesma estratégia adotada no Hadoop-MapReduce. Até a geração dos *itemsets* frequentes de tamanho 2, o processo é semelhante ao das Fases 1 e 2 do DPC. A partir da geração dos *itemsets* candidatos de tamanho 3, começa o processo iterativo para geração de candidatos e contagem/poda para produção dos *itemsets* frequentes, como mostra a Figura 20.

3.2 Nova abordagem IMRAprioriAcc-CPA Spark

Neste trabalho, foi proposta e implementada uma nova abordagem de k fases, a qual combina estratégias do IMRAprioriAcc Spark e CPA Spark. É iterativa devido às vantagens do Spark em processar bases de dados de modo incremental, sem a necessidade de persistência entre as execuções, como acontece no Hadoop-MapReduce. Utiliza a estratégia de estimativa do suporte global e de partições apresentadas no IMRAprioriAcc Spark e a estratégia de geração de *itemsets* candidatos do CPA Spark.

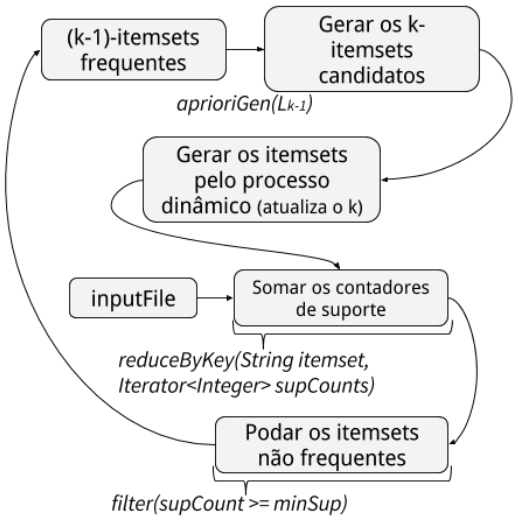


Figura 19 Fluxo da Fase dinâmica - DPC Spark

As duas primeiras iterações são para gerar os *itemsets* frequentes de tamanho 1 e 2, respectivamente. O fluxo dessas duas primeiras etapas é semelhante ao IMRAprioriAcc Spark, exceto que em vez de gerar todos os *itemsets* parcialmente frequentes, produz apenas os *itemsets* de tamanho 1 na primeira iteração e de tamanho 2 na segunda iteração. Os *itemsets* são mantidos em RDDs em vez de salvá-los no HDFS. Cada iteração estima o suporte global dos itemsets que não foram frequentes em todos os blocos e efetua a contagem dos *itemsets* parcialmente frequentes. A partir dos *itemsets* de tamanho 3, utiliza-se a estratégia do CPA Spark para geração de *itemsets* candidatos. Em seguida, aplica-se a estratégia de 2 fases do IMRAprioriAcc Spark para contar os *itemsets* em cada partição de dados, estimar o suporte global dos *itemsets* e contar os *itemsets* cujo o suporte estimado foi maior ou igual ao suporte mínimo, nas respectivas partições de dados. E, após a contagem, verifica se o contador de suporte global dos *itemsets* é maior ou igual ao suporte mínimo, a fim de produzir os *itemsets*

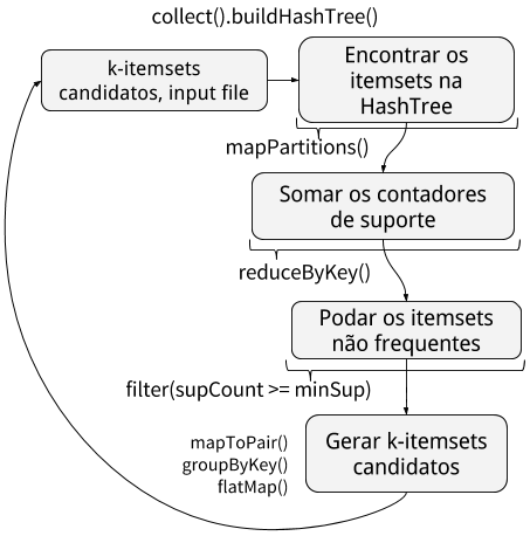


Figura 20 Fluxo CPA Spark

frequentes. O fluxo desse processo é apresentado na Figura 21. As iterações finalizam quando não é mais possível produzir *itemsets* frequentes.

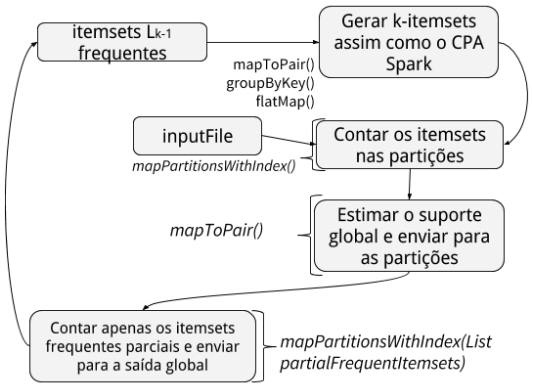


Figura 21 Fluxo IMRAprioriAcc-CPA Spark

4 ALGORITMO DE RESOLUÇÃO DE ENTIDADES NO HADOOP-MAPREDUCE E SPARK

O algoritmo de resolução de entidades aplicado para classificação de ofertas de produtos proposto em Oliveira e Pereira (2014), Oliveira e Pereira (2017) combina as técnicas de classificação e regras de associação. Utiliza-se das regras para poder predizer o rótulo de uma nova instância. Neste contexto, cada instância é composta por um identificador da entidade (classe), usado para treinamento e para cálculo das métricas de avaliação, identificador da instância (*instanciaId*) e sua descrição textual. Os termos *token* e *itemset* são tratados aqui como sinônimos, logo, 1-*token* é equivalente a 1-*itemset* representando um conjunto de itens (ou termos) de tamanho 1. O algoritmo é dividido em duas etapas principais: etapa de aprendizagem, ou treinamento, onde é gerado o modelo composto por regras a partir de dados rotulados e a etapa de classificação, ou teste, onde novas instâncias com rótulos desconhecidos são classificadas de acordo com as regas do modelo gerado.

O Algoritmo 15 apresenta o pseudocódigo da etapa de treinamento da versão sequencial, adaptado de Oliveira e Pereira (2014). A primeira parte do algoritmo (linhas 1 a 8) é destinada à construção de um arquivo invertido, contendo cada token distinto como chave e como valor uma lista contendo a classe e o *id* da instância. A partir da linha 9 é iniciado o processo de criação das regras. Os (k-1)-*tokens* de cada instância são combinados entre si para produzir os k-*tokens* candidatos à regra (linha 13 e 14). Para cada k-*token* gerado é verificado se ocorre para apenas uma classe (linha 16) a fim de garantir regras com 100% de confiança. Em seguida, se o suporte do token for maior ou igual ao suporte mínimo (*min_sup*, linha 17), então

é considerado uma regra e inserido na estrutura *R*. Assim que um *itemset* se torna uma regra ou possui suporte abaixo do mínimo ele é removido da estrutura t_k (linha 20) para evitar a criação de regras sobrepostas.

Algoritmo 15: Algoritmo sequencial - Etapa de treinamento

```
   Input: Base de treinamento D; suporte mínimo min_sup
   Output: Conjunto de regras R
 1 foreach instance d in D do
 2 |   classe = getClasse(d);
 3 |   instanceId = getInstanciaId(d);
 4 |   tokens = getTokensAndClean(d);
 5 |   foreach token t in tokens do
 6 |   |   insertInvertedIndex(t, instanceId, classe);
 7 |   end
 8 end
 9 R = ∅;
10 foreach instance d in D do
11 |   tokens = getTokensAndClean(d);
12 |   classe = getClasse(d);
13 |   for (k = 1; k < tokens.size; k++) do
14 |   |   t_k = combineTokens(k, t_{k-1});
15 |   |   foreach k-itemset it in t_k do
16 |   |   |   if (containsOneClasse (it)) then
17 |   |   |   |   if (supportRule(it,classe) ≥ min_sup ) then
18 |   |   |   |   |   insertRule(it,classe,R);
19 |   |   |   |   endif
20 |   |   |   |   removeItem(it,t_k)
21 |   |   |   endif
22 |   |   end
23 |   end
24 end
25 return R
```

Terminada a etapa de aprendizagem, inicia-se a etapa de classificação. O Algoritmo 16 ilustra esse processo, adaptado de Oliveira e Pereira (2014). Os tokens da instância de teste são combinados entre si da mesma

forma como é feito na etapa de treinamento (linhas 2 e 3). Para cada token, verifica se existe alguma regra com token e a adiciona na estrutura *countRegras* (linhas 4 a 8). Na linha 9 verifica se existe alguma classe majoritária, ou seja a classe que casou mais vezes com os tokens da instância. Caso exista, a instância foi classificada e retorna a classe predita. No final do processo, se a instância ainda não foi classificada, é utilizada uma função de similaridade para predizer sua classe (linha 13).

Algoritmo 16: Algoritmo sequencial - Etapa de classificação

```
   Input: Conjunto de regras R; base de treinamento D; instância
          de teste d
   Output: A classe predita de d
 1 tokens = getTokensAndClean(d);
 2 for k = 1; k < tokens.size; k++ do
 3 |   t_k = combineTokens(k, t_{k-1});
 4 |   foreach k-itemset it in t_k do
 5 |   |   if (r = regrasHash(it) != null) then
 6 |   |   |   countRegras.add(r);
 7 |   |   endif
 8 |   end
 9 |   if (classe = getClasseMajoritaria(countRegras) != null)
   |   then
10 |   |   return classe;                  // instância classificada
11 |   endif
12 end
13 return predictBySimilarity(D,d);        /* classificação por
   similaridade */
```

Nas seções seguintes são apresentados os detalhes das implementações do algoritmo de resolução de entidades no Hadoop-MapReduce e Spark.

4.1 Algoritmo de resolução de entidades no Hadoop-MapReduce

4.1.1 Etapa de treinamento

A etapa de treinamento é constituída de 3 conjuntos de funções MapReduce, o primeiro gera as regras candidatas de tamanho 1, o segundo conta as regras candidatas e produz as frequentes de tamanho 1 (somente função Map). O terceiro encontra as regras de tamanho k (k >1) de maneira iterativa.

A função Map da Fase 1 é apresentada no Algoritmo 17 e seu objetivo é pré-processar as instâncias, enviando os 1-*tokens* candidatos para a função Reduce. Nas linhas de 3 a 5 do Algoritmo 17 ocorre o pré-processamento da instância em que é extraída a classe, o *instanciaId* e os *tokens* da descrição. A função Map também envia para o Reduce a classe com o valor 1 (linha 9) para contagem de ocorrência de cada classe. No HDFS são persistidas as instâncias com os *tokens* pré-processados (linha 10) para serem utilizados posteriormente.

A função Reduce da Fase 1, apresentada no Algoritmo 18, verifica para cada chave se é um *token* ou classe (linha 3, Algoritmo 18). Se for um token, verifica se ele ocorre para apenas uma classe (linha 7), a fim de garantir 100% de confiança. Se sim, conta-se quantas vezes o *token* ocorreu para a classe (linha 8) e envia para a saída padrão como uma regra candidata (linha 10). Caso o *token* apareça em mais de uma classe, é enviado para uma saída separada com a lista de classes (linha 12). Para cada token é calculado e persistido em um arquivo separado no HDFS o seu IDF para ser utilizado na etapa de teste (linhas 14 e 15). Se a chave for uma classe faz a soma de seus valores e persiste separadamente no HDFS (linhas 4 e 5).

Algoritmo 17: Função Map - Fase 1 - Etapa de treinamento do algoritmo no Hadoop-MapReduce

Input: Base de treinamento D
Output: pares <token, classe:instanciaId> token candidato; pares <classe,1> para contagem de suporte das regras; pares <instanciaId, tokens> o identificador da instância com seus tokens pré-processados

```
1  foreach instance d in D do
2      Function Map(line offset, d)
3          classe = getClasse(d);
4          instanciaId = getInstanciaId(d);
5          tokens = getTokensAndClean(d);
6          foreach token t in tokens do
7              Output_1(t, classe:instanciaId);      // para a função Reduce
8          end
9          Output_2(classe, 1);                  // para a função Reduce
10         Output_3(instanciaId, tokens);            // para o HDFS
11     end
12 end
```

Algoritmo 18: Função Reduce - Fase 1 - Etapa de treinamento do algoritmo no Hadoop-MapReduce

Input: pares <chave, lista(valor)> emitidos pelas funções Maps; trainSize número total de instâncias de treino
Output: pares <token:sup, classe> token candidato à regra; pares <1-token, lista(classe)> tokens que ocorrem em mais de uma classe; pares <token, idf>

```
1  trainSize = read trainSize do DistributedCache;
2  Function Reduce(chave, lista(valor))
3      if (chave.isClasse()) then
4          soma = valor.length;
5          Output1(chave, soma);            // classe e seu suporte
6      else
7          if (containsOneClasse (valor)) then
8              sup = valor.length;
9              classe = getClasse(valor);
10             Output2(chave:sup, classe);      // regra candidata
11         else
               /* ocorre em mais de uma classe                    */
12             Output3(chave, getListaClasse(valor));
13         endif
14         idf = calcIdf(trainSize, valor.length);
15         Output4(token, idf);
16     endif
17 end
```

A Fase 2 é responsável por verificar quais regras são realmente frequentes, dentre as regras candidatas da Fase 1. Essa etapa é necessária, pois na Fase 1 ainda não se sabe o suporte de cada classe. É constituída apenas da função Map, apresentada no Algoritmo 19. É calculado o suporte da regra e verificado em relação ao suporte mínimo (linhas 5 a 8). Se for maior ou igual ao suporte mínimo, é considerada uma regra e enviada para a saída (linha 9).

Algoritmo 19: Função Map - Fase 2 - Etapa de treinamento do algoritmo no Hadoop-MapReduce

```
   Input: pares <token:sup, classe> regras candidatas emitidas
          na Fase 1; pares <classe, sup> as classes e seus
          suportes; min_sup o suporte mínimo
   Output: pares <token, classe> as regras com 1-token
 1 min_sup = read min_sup do DistributedCache;
 2 classeSup = read <classe, sup> do DistributedCache();
 3 foreach par <token:sup, classe> do
 4 |   Function Map(token:sup, classe)
 5 |   |   token = getToken(token:sup);
 6 |   |   supToken = getTokenSupport(token:sup);
 7 |   |   supClasse = classeSup.getSup(classe);
 8 |   |   if ((supToken/supClasse) >= min_sup) then
 9 |   |   |   Output(token, classe);
10 |   |   endif
11 |   end
12 end
```

Na Fase 3, o objetivo é produzir regras com k-*tokens* (k > 1) iterativamente, a partir dos *tokens* que ocorreram em mais de uma classe. O pseudocódigo da função Map dessa etapa é apresentado no Algoritmo 20. A primeira iteração dessa etapa utiliza como entrada os *tokens* pré-processados na função Map da Fase 1. A partir da segunda iteração são utilizados como entrada os (k-1)-*tokens* pré-processados que ocorreram em

mais de uma classe descobertos na iteração anterior. A função combina os (k-1)-*tokens*, que ocorrem em mais de uma classe (linha 5, Algoritmo 20), assegurando que o novo k-*token* gerado não contenha nenhum token contido em regra já gerada para evitar a sobreposição de regras. Para cada novo k-*token* gerado verifica se ocorre para somente uma classe (linha 7), caso ocorra é considerado uma regra candidata e enviado para a função Reduce (linha 8). Caso contrário o k-*token* é adicionado em uma estrutura (linha 10) e, ao final da função Map, persistido diretamente no HDFS (linha 13) para ser utilizado nas próximas iterações.

A função Reduce dessa etapa objetiva produzir as regras com k-*tokens* de maneira semelhante ao Algoritmo 19 da função Map da Fase 2.

Ao final das fases de aprendizagem todas as regras com *tokens* de tamanho k possíveis e com 100% de confiança são produzidas, onde k é um limite definido previamente. Essas regras serão utilizadas na fase de teste para classificar as novas entidades.

4.1.2 Etapa de classificação

A etapa de teste é constituída de 3 conjuntos de funções MapReduce, o primeiro classifica as entidades utilizando as regras com 1-token, o segundo, por meio de um processo iterativo, classifica as entidades que não foram classificadas na fase anterior utilizando regras com k-*tokens* ($k > 1$) e o terceiro classifica as entidades, que não foram classificadas pelas regras, utilizando a medida de similaridade do Cosseno. Nas 2 primeiras fases de teste não houve a necessidade de utilizar funções Reduces. Por tanto, as duas fases só têm funções Maps. Já a terceira fase possui ambas funções Map e Reduce.

Algoritmo 20: Função Map - Fase 3 - Etapa de treinamento do algoritmo no Hadoop-MapReduce

Input: pares <instanciaId, lista((k-1)-tokens)> na entrada padrão; pares <1-token, lista(classe)> 1-tokens que ocorrem em mais de uma classe com a lista de classe; pares <(k-1)-token, classe> regras com (k-1)-tokens

Output: pares <k-token, classe:instanciaId> as regras candidatas; <instanciaId, lista(k-tokens)> os k-tokens que ocorrem em mais de uma classe

```
1  tokensClass = read <1-token, lista(classe)> do DistributedCache();
2  regrasHash = read <(k-1)-token, classe> do DistributedCache();
3  foreach par <instanciaId, lista((k-1)-tokens)> do
4      Function Map(instanciaId, lista((k-1)-tokens))
5          newKTokens = combineTokens(lista((k-1)-tokens), tokensClass, regrasHash);   /* combina (k-1)-tokens para produzir k-tokens */
6          foreach token t in newKTokens do
7              if (containsOneClasse(t)) then
8                  Output1(t, classe:instanciaId);          /* regra candidata para o Reduce */
9              else
10                 tokens += t;
11             endif
12         end
13         Output2(instanciaId, tokens);                    // para o HDFS
14     end
15 end
```

No Algoritmo 21 é apresentado o pseudocódigo da Fase 1 de teste. Cada instância de teste é pré-processada para extrair o *instanciaId* e os *tokens* da descrição (linhas 4 e 5). Nas linhas 6 e 7, cada *token* é verificado na estrutura de regras *regrasHash* para descobrir se há alguma regra que case com o *token*. Caso uma regra case com o *token* é adicionada em uma estrutura (linha 8) para posterior verificação da classe majoritária. Após verificar todos os *tokens* da instância de teste busca-se a classe majoritária (linha 11). Se existir uma classe majoritária, a instância de teste é, então, classificada e enviada para o HDFS (linha 12). Caso contrário, a instância é enviada para a saída (linha 14) junto com a lista de *tokens* pré-processados para continuar o processo de classificação nas próximas fases.

A Fase 2 ou fase k, apresentada no Algoritmo 22, trabalha de modo iterativo onde cada iteração busca classificar as instâncias utilizando regras de tamanho k ($k > 1$). Os *tokens* são combinados entre si (linha 4) e é verificado se existem regras compostas por esses tokens (linhas 5 e 6). O processo subsequente é semelhante ao da Fase 1. A cada iteração, as instâncias que não foram classificadas com k-*tokens* serão novamente processadas na iteração seguinte, a fim de classificar utilizando as regras com (k+1)-*tokens*.

Caso o limite de iterações tenha sido atingido e, mesmo assim, permanecerem instâncias não classificadas, é iniciado o processo de classificação utilizando a função de similaridade do Cosseno. A função Map dessa etapa é apresentada no Algoritmo 23. Para cada entidade de treino, calcula-se a norma da entidade (linha 5) e, em seguida, para cada entidade de teste, calcula o Cosseno e o envia para a função Reduce (linhas 6, 7 e 8). O Hadoop-MapReduce irá agrupar, para cada instância de teste, todos os valores dos Cossenos associados a ela. A função Reduce, ilustrada no Algoritmo 24,

Algoritmo 21: Função Map - Fase 1 - Etapa de classificação do algoritmo no Hadoop-MapReduce

Input: Base de teste T; pares <1-token, classe> as regras com 1 token

Output: pares <instanciaId, classe> instâncias preditas; pares <instanciaId, lista(tokens)> as instâncias não preditas e seus tokens pré-processados

```
1  regrasHash = read <1-token, classe> do DistributedCache();
2  foreach instance d in T do
3      Function Map(line offset, d)
4          instanciaId = getInstanciaId(d);
5          tokens = getTokensAndClean(d);
6          foreach token t in tokens do
7              if (r = regrasHash(t) != null) then
8                  countRegras.add(r);
9              endif
10         end
11         if (classe = getClasseMajoritaria(countRegras) !=
           null) then
12             Output_1(instanciaId, classe);            // entidade
               classificada
13         else
14             Output_2(instanciaId, tokens);        /* entidade não
               classificada com a lista de tokens */
15         endif
16     end
17 end
```

Algoritmo 22: Função Map - Fase 2 - Etapa de classificação do algoritmo no Hadoop-MapReduce

Input: pares <instanciaId, lista(tokens)> emitidos na iteração anterior; pares <k-token, classe> as regras com k-tokens

Output: pares <instanciaId, classe> instâncias preditas; pares <instanciaId, lista(k-tokens)> as instâncias não preditas e seus k-tokens pré-processados

```
1  regrasHash = read <k-token, classe> do DistributedCache();
2  foreach par <instanciaId, lista(tokens)> do
3      Function Map(instanciaId, lista(tokens))
4          newKTokens = combineTokens(lista(tokens));
           /* combina (k-1)-tokens para produzir k-tokens */
5          foreach token t in newKTokens do
6              if (r = regrasHash(t) != null) then
7                  countRegras.add(r);
8              endif
9          end
10         if (classe = getClasseMajoritaria(countRegras) !=
           null) then
11             Output_1(instanciaId, classe);          // entidade
               classificada
12         else
13             Output_2(instanciaId, newKTokens); /* entidade não
               classificada com a lista de k-tokens */
14         endif
15     end
16 end
```

apenas verifica, dentre os elementos da lista de valores, qual classe obteve o maior valor do Cosseno. A classe que obtiver o maior valor do Cosseno é atribuída à instância de teste e enviada para o HDFS como uma instância classificada.

Algoritmo 23: Função Map - Fase 3 - Etapa de classificação pelo Cosseno no Hadoop-MapReduce

Input: Base de treino *D*; pares <instanciaId, lista(tokens)> as instâncias de teste não preditas; pares <token, idf>
Output: pares <instanciaId, classe:cosseno> instâncias com a lista de classes e os valores do Cosseno associados

```
1  insTeste = read <instanciaId, lista(tokens)> do
   DistributedCache();
2  tokenIdf = read <token, idf> do DistributedCache();
3  foreach instance d in D do
4  |  Function Map(line offset, d)
5  |  |  vecNorma = calcNormaVetor(d, tokenIdf);
6  |  |  foreach entity e in insTeste do
7  |  |  |  cosseno = calcCosseno(d, e.tokens);
8  |  |  |  Output(e.instanciaId, classe:cosseno).
9  |  |  end
10 |  end
11 end
```

Algoritmo 24: Função Reduce - Fase 3 - Etapa de classificação pelo Cosseno no Hadoop-MapReduce

Input: pares <instanciaId, lista(classe:cosseno)>
Output: pares <instanciaId, classe> instâncias classificadas

```
1  Function Reduce(instanciaId, lista(classe:cosseno)>)
2  |  classe = getClasseByMajorCosseno(lista(classe:cosseno));
3  |  Output(instanciaId, classe)
4  end
```

4.2 Algoritmo de resolução de entidades no Spark

A implementação no Spark proporcionou maior flexibilidade em determinadas situações devido aos recursos fornecidos pelos RDDs. Diferentemente do Hadoop-MapReduce, tem-se no Spark três etapas principais: pré-processamento, treinamento e classificação.

4.2.1 Etapa de pré-processamento

A partir da base de treinamento de entrada D, são criadas as estruturas em RDDs *classeSup* que contém as classes e quantidade total de suas ocorrências e a estrutura *tokensLimpos* que contém a concatenação da classe e *instanciaId* como chave e a lista de tokens pré-processados como valor. A Figura 22 ilustra os principais fluxos dessa etapa.

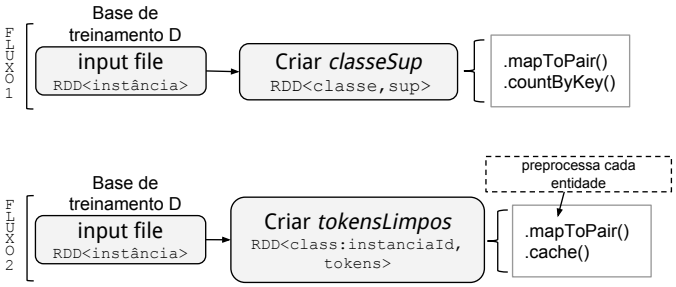


Figura 22 Fluxo do pré-processamento - Algoritmo de resolução de entidades no Spark.

A função *mapToPair()* do Fluxo 1 extrai a classe de cada instância de treino e a envia para saída como chave e o valor 1. A função *countByKey()* soma os valores associados a cada chave para emitir a saída final no RDD *classeSup*. No Fluxo 2, a função *mapToPair()* extrai a classe, *instanciaId* e pré-processa os *tokens* das entidades. Envia para a saída os pares <chave, valor> sendo a chave uma concatenação da classe com o *instanciaId* e valor

a lista de tokens processados, criando o RDD *tokensLimpos*. A partir desse RDD é iniciado o processo de treinamento.

4.2.2 Etapa de treinamento

Assim como o algoritmo Hadoop-MapReduce, o processo de treinamento no Spark é dividido em fases. A primeira Fase, ilustrada na Figura 23, objetiva criar as regras com 1-*tokens*, identificar os 1-*tokens* que ocorrem para mais de uma classe e criar a estrutura *tokenIdf*.

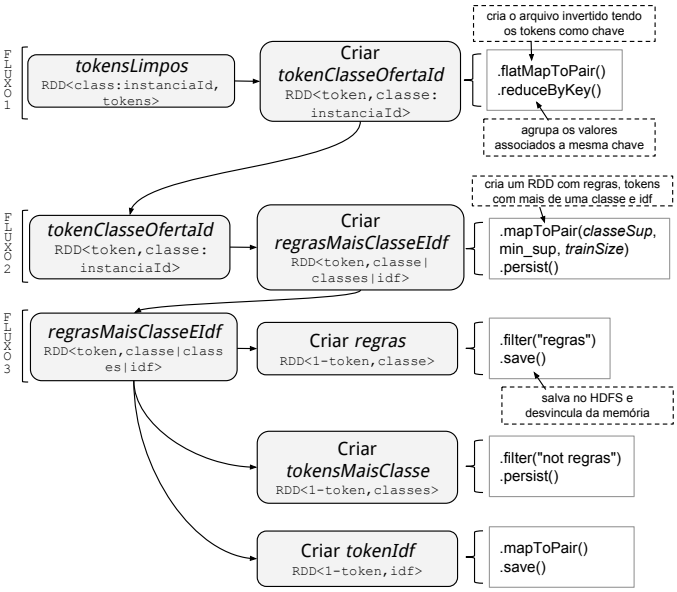


Figura 23 Fluxo do treinamento - Fase 1 - Algoritmo de resolução de entidades no Spark.

O Fluxo 1 cria o arquivo invertido definindo cada *token* distinto como chave e a lista de classes e *instanciaIds* em que ocorre como valor. O Fluxo 2 parte do arquivo invertido criado anteriormente para gerar um novo RDD contendo os tokens como chave e como valor há duas possibilidades: se o

token for uma regra, terá apenas uma classe como valor; caso contrário, se o token ocorre para mais de uma classe, terá a lista de tais classes como valor. Ambas possibilidades possuem, concatenado ao valor, o *idf* calculado na mesma função. O processo de geração de regras é semelhante ao do algoritmo no Hadoop-MapReduce descrito anteriormente. Os fluxos seguintes são para separar os tokens que são regras dos demais tokens.

A Fase 2 compreende um processo iterativo para produzir regras de tamanho k (k > 1), como ilustrado na Figura 24. No Fluxo 1 a função *flatMapToPair()* é executada para combinar somente os 1-*tokens* que ocorrem mais de uma classe para produzir os k-*tokens* candidatos (tokens que ocorrem apenas para uma classe). A função também retorna para o mesmo RDD os k-*tokens* que ocorrem em mais de uma classe identificados. No Fluxo 2 o RDD *candidatoEMaisClasse* é separado em um RDD somente de candidatos e outro RDD para os tokens que ocorrem para mais de uma classe. Este último é atribuído ao *tokensLimpos* e será utilizado na próxima iteração. A partir dos *candidatos* criam-se as regras com k-*tokens* no Fluxo 3 e as persistem no HDFS.

O processo iterativo se repete até um limite específico. Ao final do processo todas as regras com k-*tokens* foram geradas e persistidas no HDFS.

4.2.3 Etapa de classificação

A fase de testes é composta por duas fases principais: classificação por regras e classificação por similaridade. Assim como nas demais implementações do algoritmo de resolução de entidades as instâncias que não foram classificadas pelas regras no Spark serão classificadas pela função similaridade do Cosseno. Antes da etapa de classificação os dados de testes

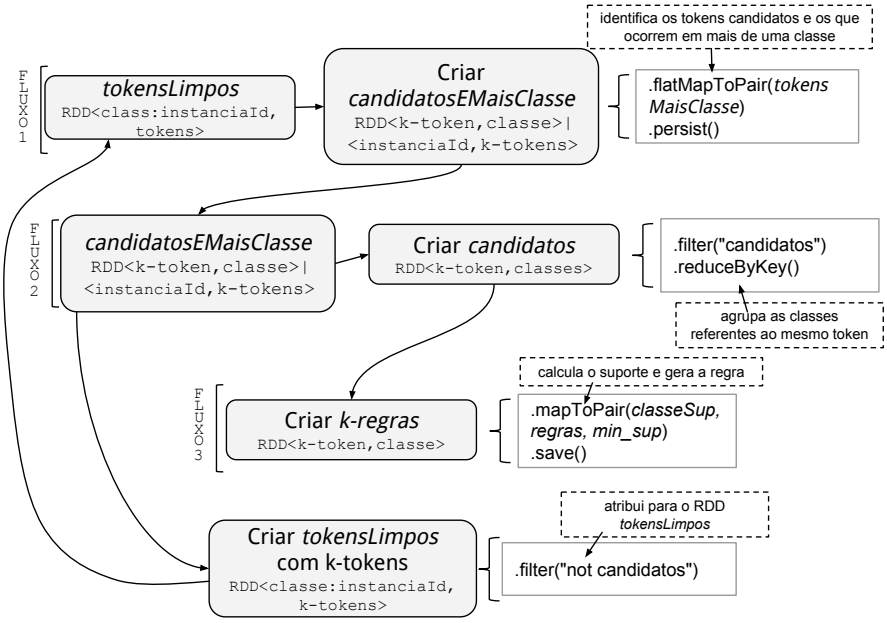


Figura 24 Fluxo do treinamento - Fase 2 - Algoritmo de resolução de entidades no Spark.

são pré-processados e os tokens extraídos de cada instância, criando o RDD *tokensLimpos*. Inicia-se um processo iterativo para classificar as instâncias utilizando regras até o valor máximo de k. A Figura 25 ilustra os principais fluxos dessa etapa.

No Fluxo 1, o processo é semelhante à etapa de teste do algoritmo Hadoop-MapReduce. A função *mapToPair()* verifica para cada token da instância se compreende alguma regra. Depois de verificar todos os tokens de uma instância, avalia a regra majoritária. Caso haja uma regra majoritária, a instância é classificada. Caso contrário a instância será classificada nas próximas iterações. Para $k > 1$, a função *mapToPair()* efetua a combinação dos tokens de cada instância para predizer a classe utilizando regras com

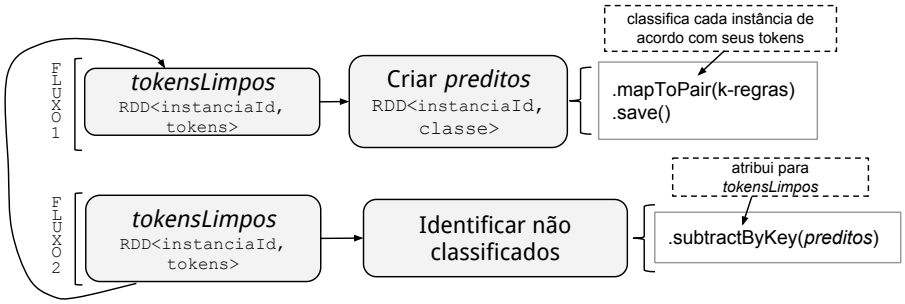


Figura 25 Fluxo da classificação - Fase 1 - Algoritmo de resolução de entidades no Spark.

k-*tokens*. O processo iterativo se repete até que todas as instâncias tenham sido classificadas ou que o limite máximo de k tenha sido atingido.

Após o término da classificação utilizando regras, se ainda houver instância não classificada, inicia-se o processo de classificação utilizando o Cosseno. O fluxo deste processo é apresentado na Figura 26.

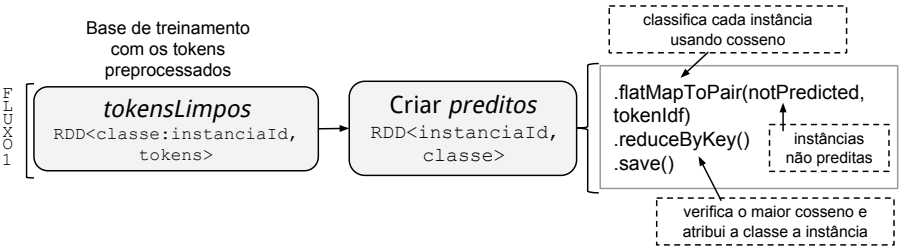


Figura 26 Fluxo da classificação - Fase 2 - Algoritmo de resolução de entidades no Spark.

A base de dados de treinamento com os tokens já pré-processados e atribuída ao RDD *tokensLimpos* executa a função *flatMapToPair()* enviando como parâmetro as entidades não preditas pelas regras e a estrutura de tokens com seus idfs. A função emite os valores associados do Cosseno para cada instância de teste em relação a cada instância de treino. A função

reduceByKey() agrupa os valores do Cosseno associados à mesma instância de teste e verifica qual possui o maior valor. Atribui a classe à instância de teste, classificando-a e a persiste no HDFS.

5 AVALIAÇÃO EXPERIMENTAL DAS IMPLEMENTAÇÕES DO ALGORITMO APRIORI NO HADOOP-MAPREDUCE E SPARK

A seguir são apresentadas as metodologias aplicadas aos experimentos para as implementações do algoritmo Apriori no Hadoop e Spark. Em seguida são apresentados os resultados dos experimentos para coleta de tempo de execução e das métricas de avaliação para computação distribuída.

5.1 Metodologia para os experimentos do algoritmo Apriori no Hadoop-MapReduce e Spark

Neste trabalho, os algoritmos IMRAprioriAcc, DPC e CPA, descritos na Seção 2.5, foram implementados de acordo com instruções apresentadas nos respectivos artigos. Esses algoritmos foram analisados por seus autores sob condições específicas e situações distintas entre as abordagens. Não houve uma comparação entre as três abordagens sob as mesmas circunstâncias. Para o IMRAprioriAcc (FARZANYAR; CERCONE, 2013a), os autores o compararam com a versão anterior do IMRApriori (FARZANYAR; CERCONE, 2013b) com apenas uma base de dados variando a quantidade de transações e o valor do suporte mínimo. Os autores do DPC (LIN; LEE; HSUEH, 2012) o compararam em relação às outras versões do algoritmo (SPC e FPC) proposta no mesmo artigo. Para o CPA (ZHOU; HUANG, 2014), os autores o compararam com uma versão iterativa, semelhante ao SPC, para apenas uma base de dados.

Também é importante avaliar como esses algoritmos se comportam quando reimplementados no *framework* Spark. Segundo a literatura, algoritmos no Spark tendem a ser até 10 vezes mais rápidos do que algoritmos

implementados no Hadoop devido à capacidade de manter os dados em memória através dos RDDs e distribuídos pelo cluster (ZAHARIA et al., 2010). Logo, foram avaliados 7 implementações distribuídas do algoritmo Apriori, sendo 3 implementações no Hadoop-MapReduce referentes aos trabalhos originais do IMRAprioriAcc, DPC e CPA e 4 implementações no Spark, 3 referentes aos algoritmos Hadoop-MapReduce mapeados para o Spark e 1 referente à nova abordagem apresentada neste trabalho, a qual combina estratégias do IMRAprioriAcc Spark e CPA Spark.

Portanto, a hipótese é que esses algoritmos possam ter comportamentos diferentes se experimentados sob as mesmas circunstâncias, ou seja, sob as mesmas bases de dados, valores de suporte mínimo e cluster. Objetiva-se avaliar se existe um relacionamento entre o desempenho dos algoritmos e diferentes variáveis do problema. Para isso, foram definidas as seguintes variáveis independentes: suporte mínimo, quantidade de transações, número de itens por transação e número de itens distintos na base de dados. Também é considerado o número de máquinas do cluster para a avaliação das métricas *SpeedUp* e *ScaleUp*, apresentadas na Seção 5.1.2. A Tabela 5 apresenta os valores assumidos por cada variável utilizada nos experimentos.

Tabela 5 Variáveis independentes utilizadas nos experimentos de tempo de execução.

VARIÁVEL INDEPENDENTE	**VALORES**
Suporte mínimo	0.5%, 0.1% e 0.01%
Número de transações	1.000.000 e 5.000.000
Número de itens distintos	5.000, 10.000 e 100.000
Número de itens por transação	5, 10, 15 e 30

A maioria dos valores assumidos pelas variáveis independentes são baseados em valores comumente utilizados na literatura, especificamente nos trabalhos aqui avaliados. Outros valores foram adicionados neste trabalho, visando avaliar o comportamento dos algoritmos em situações distintas. Para o suporte mínimo, o valor 0.01% foi inserido para compreender o comportamento dos algoritmos quando a quantidade de itemsets frequentes é muito grande em cada base de dados. O número de transações 5 milhões foi inserido para averiguar o impacto que o maior número de transações causa nos algoritmos. O número de itens distintos também são comumente utilizados na literatura, exceto o valor de 5 mil que foi inserido pois, tende-se a ter menor número de itemsets falsos candidatos. Para o número de itens por transação, buscou-se identificar o quanto a variação desse valor afeta na performance dos algoritmos.

O tempo de execução foi utilizado como variável dependente, ou seja, é desejável saber como os algoritmos se comportam em relação ao tempo de execução à medida que se altera os valores das variáveis independentes. Todos os algoritmos foram executados 3 vezes e o tempo coletado foi obtido da média das três execuções. Foram realizados testes de significância estatística do tipo t-teste independente (JAIN, 1991, p 209) para verificar se há diferença estatisticamente significante entre os tempos de execução de cada algoritmo e base de dados. Na avaliação dos experimentos, em todos os pontos em que se diz que um algoritmo é melhor do que outro, isso foi verificado estatisticamente com um intervalo de confiança de 95%. É importante ressaltar que para os experimentos de avaliação do tempo de execução, a quantidade de funções Maps e Reduces, bem como o número de máquinas, são constantes. Baseado no tamanho das base de dados e da

disponibilidade de máquinas no cluster, foram definidas 8 funções Maps e 8 funções Reduces. A Tabela 6 apresenta as variáveis controladas para os experimentos de tempo de execução.

Tabela 6 Variáveis controladas utilizadas nos experimentos de tempo de execução.

VARIÁVEL CONTROLADA	**VALOR**
Maps e Reduces	8
Máquinas no cluster	20 (1 mestre, 20 escravas)
Processador	Intel Core i5-2400, 4 núcleos a 3.10GHz
Memória	8Gb a 1333MHz
Disco rígido	500Gb
Sistema operacional	Ubuntu Desktop 14.04 LTS
Hadoop	2.7.1
Spark	1.5.2
Java	8u64

Nessas condições, a quantidade necessária de experimentos a serem executados seria 1512, valor obtido por meio da Equação 11.

$$\begin{aligned} \textit{Quantidade de experimentos} = (3\ \textit{valores de suporte mínimo}) \times \\ (2\ \textit{quantidade de transações}) \times (3\ \textit{valores para itens por transações}) \times \\ (4\ \textit{valores para itens distintos}) \times (3\ \textit{execuções}) \times (7\ \textit{algoritmos}) \end{aligned} \tag{11}$$

Porém, devido a diversos aspectos como, limitações de hardware do cluster, excessivo tempo de execução dos algoritmos, alta demanda de memória dos algoritmos, criação das bases de dados e prazo para execução dos experimentos, foram elaborados experimentos do tipo fatorial fracionado (JAIN, 1991, p 281). Em outras palavras, optou-se por reproduzir situações comumente citadas na literatura e outros casos mais extremos que exigem

alto poder de processamento, porém dentro dos limites das máquinas do cluster. Dessa forma foi possível definir os parâmetros para criação das bases de dados, como apresentado na Seção 5.1.1

5.1.1 Base de dados

As bases de dados utilizados nos experimentos são sintéticas geradas pela biblioteca "*arules*" (HAHSLER et al., 2016) do *Software R-cran*. Foram produzidas 8 bases de dados apresentadas na Tabela 7. Em relação às nomenclaturas das bases, T é a média de itens por transação, D é a quantidade de transações, N é a quantidade de itens distintos na base de dados e K é a representação de mil. As bases de dados T30D1000KN10K e T15D1000KN100K foram as que geraram mais processamento computacional devido ao alto número de itens por transação.

Tabela 7 Base de dados sintéticas utilizadas nos experimentos.

NOME	TRANSAÇÕES	ITENS DISTINTOS	ITENS POR TRANSAÇÃO
T10D1000KN5K	1.000.000	5.000	10
T10D1000KN10K	1.000.000	10.000	10
T10D1000KN50K	1.000.000	50.000	10
T5D1000KN100K	1.000.000	100.000	5
T15D1000KN100K	1.000.000	100.000	15
T30D1000KN10K	1.000.000	10.000	30
T10D5000KN10K	5.000.000	10.000	10
T10D5000KN100K	5.000.000	100.000	10

Com as bases de dados definidas, a quantidade necessária de experimentos reduziu para 504 como mostra a Equação 12.

$$\begin{aligned} \textit{Quantidade de experimentos} = (3\ \textit{valores de suporte mínimo}) \times \\ (8\ \textit{base de dados}) \times (3\ \textit{execuções}) \times (7 \textit{algoritmos}) \end{aligned} \tag{12}$$

5.1.2 Métricas de avaliação

Além do tempo de execução, também foram calculadas outras métricas de avaliação específicas para computação distribuída: *SpeedUp*, *SizeUp* e *ScaleUp*.

SpeedUp avalia o comportamento do algoritmo para uma mesma base de dados, porém variando o número de máquinas que a processa como ilustrado na Equação 13, onde T_1 é o tempo de execução do algoritmo com uma máquina e T_p é o tempo de execução do algoritmo com p máquinas. A situação ideal é quando o tempo de execução de um algoritmo como p máquinas seja p vezes menor do que o tempo de execução com apenas uma máquina, criando uma linha linear no gráfico.

$$SpeedUp = \frac{T_1}{T_p} \tag{13}$$

O *SizeUp* avalia o comportamento do algoritmo quando apenas a base de dados aumenta, mas o número de máquinas processando-as mantém-se constante como apresentado na Equação 14, onde T_m é o tempo de execução com $m * data$ e T_1 é o tempo de execução para processar $data$. A situação ideal é quando o tempo de execução de um algoritmo para processar $m * data$

seja m vezes maior do que o tempo para processar $data$, criando uma linha linear no gráfico.

$$SizeUp(data, m) = \frac{T_m}{T_1} \tag{14}$$

E o $ScaleUp$ avalia o comportamento do algoritmo quando, tanto a base de dados a ser processada aumenta, quanto, proporcionalmente, cresce o número de máquinas processando esses dados. A Equação 15 apresenta o cálculo do $ScaleUp$ onde T_1 é o tempo de execução para processar $data$ em uma máquina e T_{mm} é o tempo de execução para processar $m * data$ em m máquinas. A situação ideal é quando o tempo de execução de um algoritmo para processar $m * data$ em m máquinas seja equivalente ao tempo gasto para processar $data$ em apenas uma máquina, ou seja, o valor ideal do $ScaleUp$ deve sempre manter-se 1.

$$ScaleUp(data, m) = \frac{T_1}{T_{mm}} \tag{15}$$

Para os experimentos com as implementações do algoritmo Apriori no Hadoop-MapReduce e Spark a fim de avaliar essas 3 métricas, foi utilizada a base de dados T10D1000KN10K com características intermediárias, variando apenas a quantidade de transações, para garantir a execução de todos os algoritmos sem exceder os recursos do hardware. Para o $SpeedUp$ foi utilizado o valor 0.01% como suporte mínimo a fim de observar o quão escalável são os algoritmos perante à alta demanda de processamento. Para as métricas $SizeUp$ e $ScaleUp$ a quantidade de transações variou entre 250K, 500K, 1000K, 2000K e 4000K. Para o $ScaleUp$ também variou-se, de maneira proporcional, a quantidade de máquinas (1, 2, 4, 8 e 16). Para essas duas

últimas métricas, o valor do suporte mínimo foi definido como sendo 0.1% para garantir a execução de todos os algoritmos sem ultrapassar os limites de memória ao processar as bases de dados maiores. Para cada métrica, foram realizadas 105 execuções obtidas pela Equação 16, onde *variações* é a quantidade de variações dos experimentos, isto é, a métrica *SpeedUp* varia 5 vezes em relação à quantidade de máquinas, a métrica *SizeUp* varia 5 vezes em relação ao tamanho da base de dados e a métrica *ScaleUp* também varia 5 vezes.

$$\begin{aligned} \textit{Quantidade de experimentos} = (7\ \textit{algoritmos}) \times \\ (5\ \textit{variações}) \times (3\ \textit{execuções}) \end{aligned} \tag{16}$$

Para avaliar as hipóteses em relação ao comportamento dos algoritmos distribuídos do Apriori diversos experimentos foram executados, como apresentado na Seção 5.1. Nas seções seguintes são apresentados os resultados dos experimentos, bem como, as métricas de avaliação para computação distribuída.

5.2 Algoritmos no Hadoop-MapReduce

5.2.1 Tempo de execução

Para o suporte mínimo de 0,5% (0,005), os três algoritmos apresentaram um padrão de comportamento semelhante para todas as bases de dados, sendo o IMRAprioriAcc o mais rápido, seguido pelo DPC e o CPA. A Figura 27 apresenta a média dos tempos das três execuções para os três algoritmos para as bases de dados avaliadas. Com esse suporte, a quantidade de *itemsets* frequentes produzida é relativamente pequena, ≈16 mil

itemsets frequentes ao todo para a base de dados que produziu mais *itemsets* (T30D1000KN10K) e ≈120 para a base de dados que produziu menos *itemsets* (T5D1000KN100K).

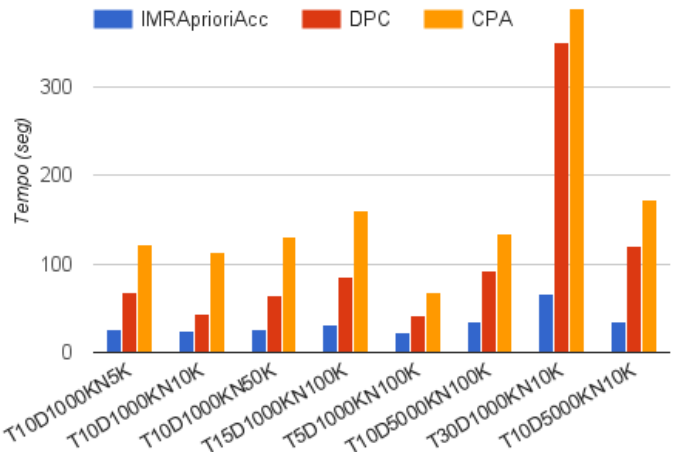


Figura 27 Tempo de execução dos algoritmos para suporte mínimo 0,5%, 8 Maps e 8 Reduces

Para a base de dados que produziu menos *itemsets*, o IMRAprioriAcc foi 83% mais rápido do que do DPC e 203% mais rápido do que o CPA, enquanto que o DPC foi 65% mais rápido do que o CPA. Para a base de dados que produziu mais *itemsets*, o IMRAprioriAcc foi 430% mais rápido do que do DPC e 488% mais rápido do que o CPA, enquanto que o DPC foi 11% mais rápido do que o CPA.

Essa grande diferença de comportamento dos algoritmos de k fases em relação ao algoritmo de duas fases se deve principalmente ao maior número de execuções MapReduce necessárias para encontrar todos os *itemsets* frequentes. Para a menor base de dados, o DPC gastou 3 execuções MapReduce enquanto que o CPA gastou 7. Para a maior base de dados, o DPC gastou 5 execuções e o CPA, 18. Apesar de o CPA ter gastado muito

mais execuções MapReduce do que o DPC, ele foi apenas 11% mais lento para a maior base de dados em relação à menor. O motivo é que, nas fases dinâmicas do DPC, os *k-itemsets* candidatos são produzidos a partir dos *(k-1)-itemsets* também candidatos (e não com os frequentes como ocorre no CPA). Isso resulta em uma grande quantidade de *itemsets* falsos candidatos sendo processados, prejudicando o tempo de execução. Por exemplo, na Fase 4 do DPC, foram gerados os *itemsets* de tamanho 4, 5 e 6. Os *4-itemsets* candidatos foram gerados a partir dos *3-itemsets* frequentes da fase anterior, mas *5-itemsets* candidatos foram gerados a partir desses *4-itemsets* candidatos. O mesmo ocorre para os *6-itemsets* candidatos, que foram gerados a partir dos *5-itemsets* candidatos. Só após o término da geração dos candidatos é que ocorre a contagem e poda para produzir os 4, 5 e *6-itemsets* frequentes.

Para o suporte mínimo 0,1% (0,001), os tempos de execução dos algoritmos tendem a apresentar o mesmo comportamento em relação ao suporte mínimo 0,5%, exceto para a maior base de dados, T30D1000KN10K, onde o CPA passa a ter o melhor desempenho. A Figura 28 apresenta a média dos tempos de execução para essa configuração. O CPA foi 25% mais rápido do que o IMRAprioriAcc e 41% mais rápido do que o DPC. O IMRAprioriAcc foi 13% mais rápido do que o DPC. Para essa base de dados, foram gerados ao todo 88.295 *itemsets* frequentes, com tamanhos $k = 1$ a $k = 10$. Vale ressaltar que IMRAprioriAcc gera todos os *itemsets* candidatos e frequentes em apenas duas fases, com ênfase na Fase 1 onde são geradas todas as combinações de itens. Na Fase 1, o consumo de tempo foi 99,89% do tempo total de execução do IMRAprioriAcc. É um processo

que demanda bastante tempo, pois, nessa fase, foram produzidos *itemsets* de 10 tamanhos distintos em cada função Map.

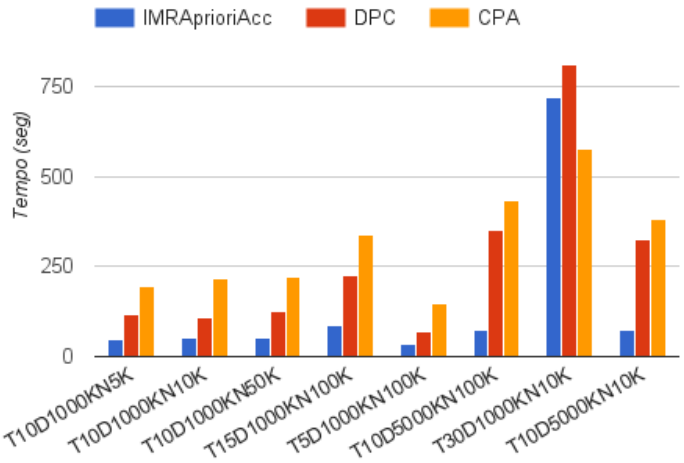


Figura 28 Tempo de execução dos algoritmos para suporte mínimo 0,1%, 8 Maps e 8 Reduces

Para o suporte mínimo de 0,01% (0,0001) houve problemas de estouro de memória para a maior base de dados, pois o volume de *itemsets* processados é bastante elevado. Para as demais bases o CPA foi o mais rápido em quatro delas, como mostra a Figura 29. Além disso, o DPC também foi melhor do que o IMRAprioriAcc em algumas bases de dados. Por exemplo, na base de dados T15D1000KN100K, o tempo de execução do CPA foi 541% mais rápido do que o DPC e 7.987% mais rápido do que o IMRAprioriAcc. Para essa base, foram gerados 390.769 *itemsets* frequentes, com tamanhos k=1 a k=13, resultando em 25 execuções MapReduce para o CPA e 6 execuções para o DPC. Apesar da quantidade de execuções MapReduce, o CPA processa relativamente menos *itemsets* por execução do que o DPC, isso torna todo o tempo gasto em inicialização e comunicação do Hadoop-MapReduce desprezível. Observa-se que conforme a quantidade de

itemsets frequentes produzidos aumenta os algoritmos com menos execuções MapReduce tendem a demandar mais tempo de execução.

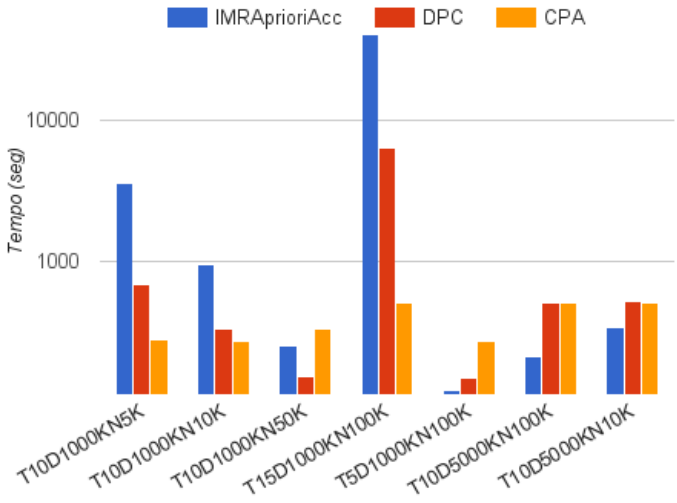


Figura 29 Tempo de execução dos algoritmos para suporte mínimo 0,01%, 8 Maps e 8 Reduces

5.2.2 SpeedUp

O gráfico dos experimentos de SpeedUp é apresentado na Figura 30. Foi utilizada a base de dados T10D1000KN10K com suporte mínimo de 0.01%. Nota-se que o DPC foi o que apresentou maior SpeedUp para até 4 máquinas. A partir de então, ele tende a se estabilizar. O CPA teve um crescimento mais constante, com diminuição no crescimento depois de 8 máquinas. O IMRAprioriAcc, por outro lado, apresentou piora no desempenho à medida que a quantidade de máquinas aumentou. O motivo se deve ao controle que o algoritmo deve manter das partições de dados espalhadas pelo cluster para o seu processo de contagem e poda. Quanto maior

a quantidade de partições, mais operações de I/O em disco são necessárias na execução do algoritmo.

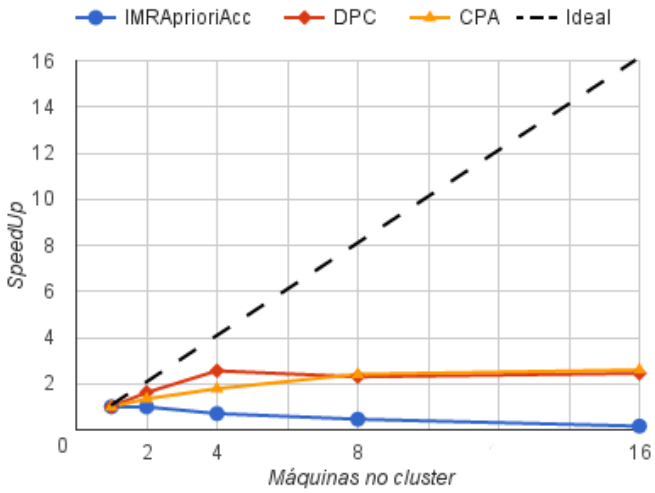


Figura 30 SpeedUp para os algoritmos Hadoop-MapReduce

5.2.3 SizeUp

O gráfico dos experimentos de SizeUp é apresentado na Figura 31. Foram mantidas 8 máquinas escravas no cluster, enquanto que a base de dados era aumentada em número de transações utilizando o suporte mínimo de 0.1%. Observa-se que os algoritmos tendem a perder desempenho à medida que a base de dados aumenta de 2 milhões para 4 milhões, especialmente o DPC que teve o pior desempenho no SizeUp dentre os 3 algoritmos.

As Fases 2 e 4 do DPC foram as que mais demandaram tempo e as que mais processaram *itemsets*, especialmente a Fase 2 que demandou mais tempo para encontrar todos o *2-itemsets* frequentes, seguido pela Fase 4, a qual produziu mais *itemsets*. Isso implica que, para alguns casos, a abordagem dinâmica não é tão eficiente, pois ocorrem má distribuição de

carga entre uma fase e outra causando *overheads* paralelos o que impacta no tempo de execução final.

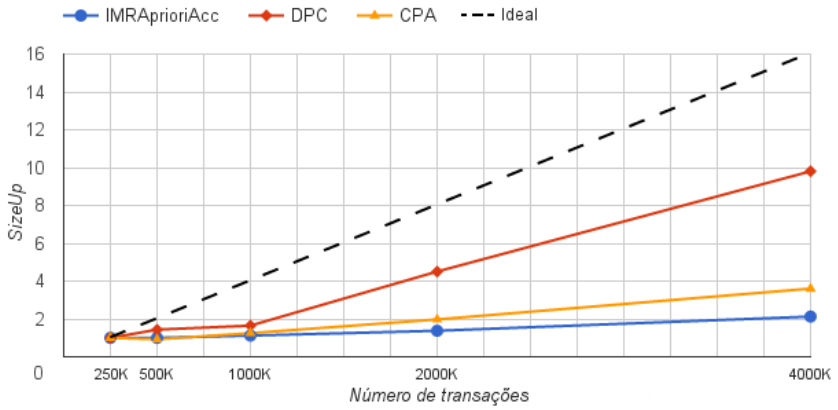


Figura 31 SizeUp para os algoritmos Hadoop-MapReduce

5.2.4 ScaleUp

O gráfico dos experimentos de ScaleUp é apresentado na Figura 32, também utilizando o suporte mínimo de 0.1%. É possível notar que o IMRAprioriAcc e o DPC apresentam baixa escalabilidade em relação ao CPA. A partir de 8 máquinas, o DPC apresenta piora na escalabilidade ficando abaixo dos 60%, enquanto que o CPA e IMRAprioriAcc ficam acima dos 70%.

5.3 Algoritmos Spark

5.3.1 Tempo de execução

A Figura 33 apresenta as médias dos tempos de execução dos algoritmos, utilizando o suporte mínimo de 0,5% (0,005). O algoritmo IMRAprioriAcc Spark foi o mais rápido em 4 bases de dados (T10D1000KN5K,

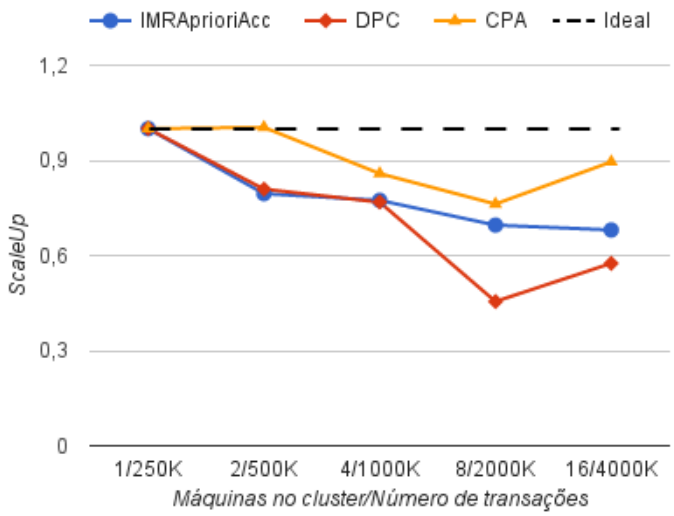


Figura 32 ScaleUp para os algoritmos Hadoop-MapReduce

T10D1000KN10K, T10D1000KN50K, T30D1000KN10K). O algoritmo DPC Spark foi o mais rápido nas outras 4 bases (T15D1000KN100K, T5D1000KN 100K, T10D5000KN100K, T10D5000KN10K). Observa-se que o IMRApriori-Acc Spark foi mais rápido nas bases de dados que possuem menos *itemsets* distintos, enquanto que o DPC Spark foi mais rápido naquelas que possuem mais transações e mais *itemsets* distintos. Porém, a diferença de tempo entre esses dois algoritmos é bem pequena. Em média, para as bases de dados que o IMRAprioriAcc Spark se sobressaiu, ele foi 9,5% mais rápido do que o DPC Spark. Estatisticamente, não há diferença significativa nesse caso. Enquanto que para aquelas em que o DPC Spark se sobressaiu, ele foi 24,2% mais rápido que o IMRAprioriAcc Spark.

O IMRAprioriAcc-CPA Spark foi o segundo algoritmo mais rápido, atrás somente do IMRAprioriAcc Spark, para a base de dados T10D1000KN 5K e o terceiro mais rápido, na frente do CPA Spark, para a base de dados T10D5000KN10K. Nos demais casos, o IMRAprioriAcc-CPA Spark foi o

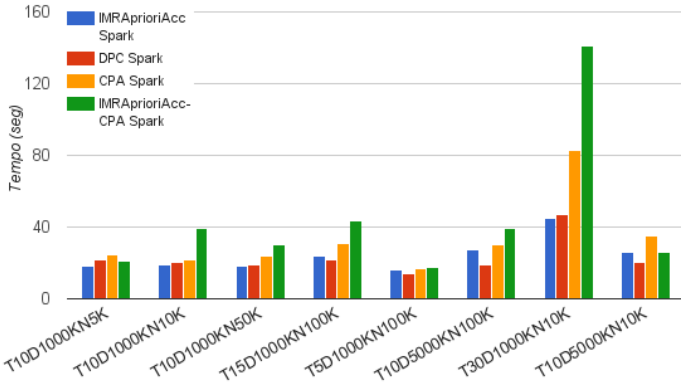


Figura 33 Tempo de execução dos algoritmos Spark para suporte mínimo 0,5% e 8 blocos

mais lento. Esses resultados mostram que a nova abordagem não é adequada para experimentos com esse nível de suporte mínimo, pois apesar de ser um algoritmo de k fases ele executa muitas operações por fase, mesclando a estratégia de poda do IMRAprioriAcc-Spark e a de geração de candidatos do CPA-Spark, logo para situações com poucos itemsets produzidos, essas operações podem não ser eficientes.

Para o suporte mínimo de 0,1% (0,001), o IMRAprioriAcc Spark foi o algoritmo mais rápido para 7 bases de dados, seguido pelo DPC Spark, CPA Spark e IMRAprioriAcc-CPA Spark, como mostra a Figura 34. Somente para a maior base de dados (T30D1000KN10K) houve variação em relação ao algoritmo mais rápido. Para essa base de dados, o CPA Spark foi o algoritmo mais rápido e o IMRAprioriAcc Spark, o mais lento, sendo 1.231% mais lento. Observa-se um comportamento semelhante ao dos algoritmos Hadoop-MapReduce conforme o suporte mínimo vai diminuindo. O IMRAprioriAcc-CPA Spark foi o segundo mais rápido para essa base de dados. Não houve diferença estatisticamente significante entre os algorit-

mos IMRAprioriAcc Spark e DPC Spark para as bases T5D1000KN100k e T10D1000KN5K.

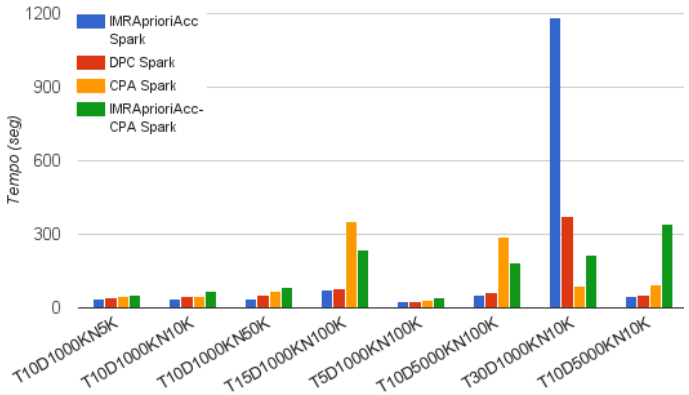


Figura 34 Tempo de execução dos algoritmos Spark para suporte mínimo 0,1% e 8 blocos

Para o suporte mínimo de 0,01% (0.0001), a quantidade de *itemsets* produzida para cada base de dados é muito grande, o que resultou em problemas de execução dos algoritmos Spark para algumas bases de dados, devido ao limite de memória de cada máquina. Portanto, não foi possível obter resultados para algumas bases de dados devido à alta demanda por memória. Para as bases processadas, observou-se que o CPA Spark foi o algoritmo mais rápido para 4 delas e o DPC Spark, para outras duas bases de dados, como apresentado na Figura 35. O IMRAprioriAcc Spark, por executar somente duas iterações, da mesma forma que em sua implementação no Hadoop-MapReduce, foi o mais lento dos 4 algoritmos, chegando a ser 6.038% mais lento que o CPA Spark para a base de dados T10D1000KN5K. O DPC Spark e o IMRAprioriAcc-CPA Spark foram 921% e 78% mais lentos que o CPA Spark, respectivamente, para a mesma base de dados. Como observado nos algoritmos Hadoop-MapReduce, conforme a quantidade de

itemsets frequentes produzida aumenta, os algoritmos Spark com menos iterações tendem a demandar mais tempo de execução.

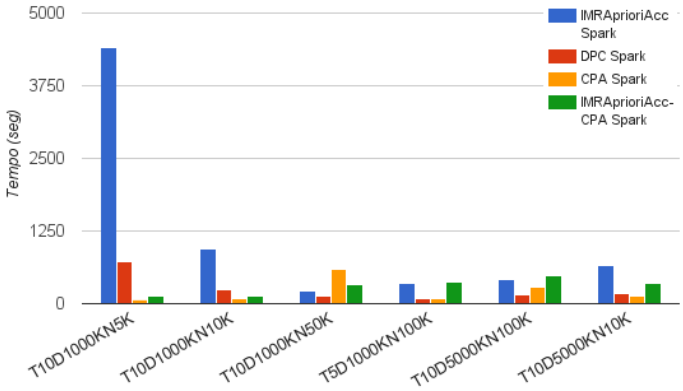


Figura 35 Tempo de execução dos algoritmos Spark para suporte mínimo 0,01% e 8 blocos

5.3.2 SpeedUp

O gráfico dos experimentos de SpeedUp é apresentado na Figura 36. Assim como nos algoritmos no Hadoop, foi utilizada a base de dados T10D1000KN10K com suporte mínimo de 0,01%. Observa-se que com duas máquinas a nova abordagem é a que apresenta o melhor SpeedUp. Com 4 máquinas, o IMRAprioriAcc-CPA Spark e o CPA Spark apresentam queda de SpeedUp. Ambos utilizam o mesmo método para geração de *itemsets* candidatos, o que implica mais dados trafegando pela rede. Com 8 máquinas, a quantidade de dados trafegando entre cada máquina é reduzida, aliviando a sobrecarga da rede e elevando a taxa de SpeedUp. Com 16 máquinas, o CPA Spark foi o que obteve o melhor SpeedUp enquanto que o IMRAprioriAcc Spark teve um comportamento semelhante a sua implementação no Hadoop. No geral, os algoritmos no Hadoop apresentaram

melhor SpeedUp, exceto o IMRAprioriAcc que teve desempenho inferior a sua versão Spark.

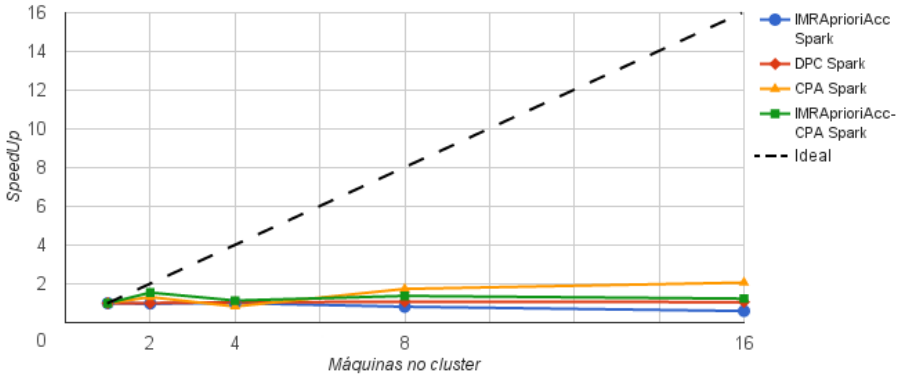


Figura 36 SpeedUp para os algoritmos Spark

5.3.3 SizeUp

O gráfico dos experimentos de SizeUp, utilizando o suporte mínimo de 0,1%, é apresentado na Figura 37. Nota-se que o IMRAprioriAcc-CPA Spark apresenta o melhor SizeUp dentre os 4 algoritmos, ou seja, o algoritmo é mais eficiente do que os demais conforme a base de dados aumenta. Para esse caso, o CPA Spark apresenta baixo desempenho para as bases de dados com 2 milhões de transações, enquanto o IMRAprioriAcc Spark foi o pior para a base de dados com 4 milhões de transações. Apesar de o IMRAprioriAcc-CPA Spark gastar a mesma quantidades de iterações que o CPA Spark, seu método de contagem e poda dos *itemsets* é mais eficiente, o que contribui para o seu melhor desempenho. As abordagens Spark superam os algoritmos no Hadoop em relação ao SizeUp.

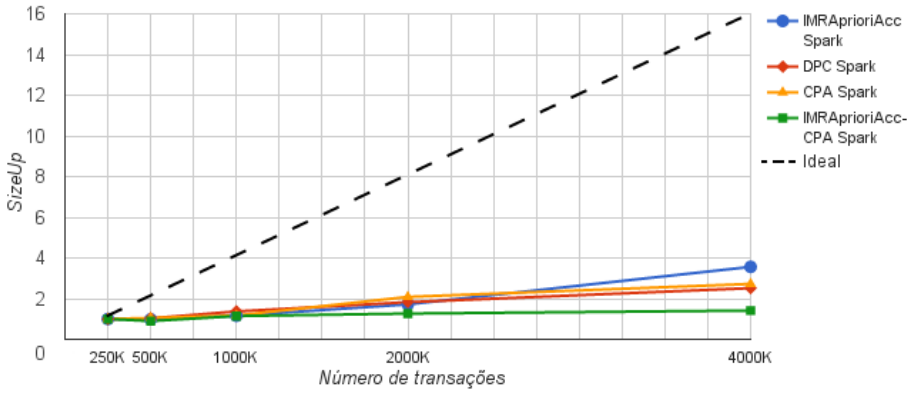


Figura 37 SizeUp para os algoritmos Spark

5.3.4 ScaleUp

O gráfico dos experimentos de ScaleUp, utilizando o suporte mínimo de 0,1%, é apresentado na Figura 38. O IMRAprioriAcc-CPA Spark apresenta um comportamento interessante, pois se mantém melhor do que as demais abordagens para quase todas as situações. Somente para o caso de 4 máquinas e base de dados com 1 milhão de transações que o DPC Spark foi ligeiramente melhor. Dentre os algoritmos, o que apresentou o pior desempenho para o ScaleUp foi o IMRAprioriAcc Spark mantendo a escalabilidade em torno de 40% ou menos para 4, 8 e 16 máquinas.

Para o ScaleUp, as abordagens implementadas no Hadoop-MapReduce apresentam melhor comportamento em relação aos algoritmos Spark, principalmente quando o número de máquinas e tamanho das base de dados são maiores.

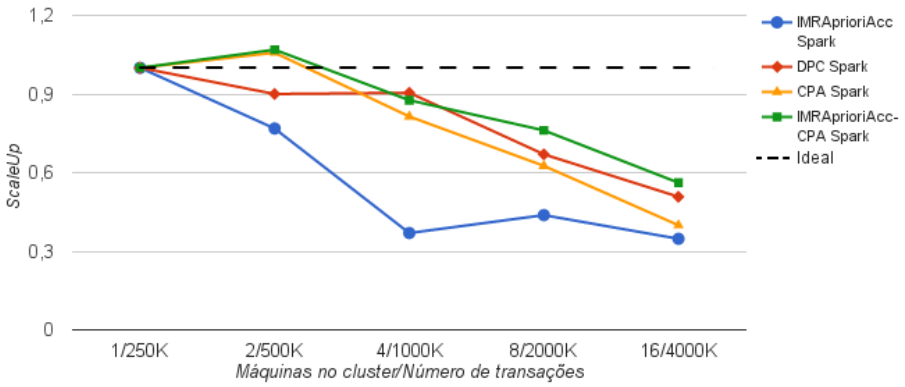


Figura 38 ScaleUp para os algoritmos Spark

5.4 Hadoop-MapReduce vs Spark

Para os algoritmos no Hadoop-MapReduce, o IMRAprioriAcc foi o algoritmo mais rápido para suportes mais altos e bases de dados que não ultrapassaram a marca de 63 mil *itemsets* frequentes produzidos. Enquanto que nas demais situações, o CPA foi a melhor opção. No Spark, observa-se que ambos, IMRAprioriAcc Spark e DPC Spark, foram boas opções para suporte mínimo mais alto, com preferência para o DPC Spark. Conforme a quantidade de *itemsets* frequentes produzida aumenta (entre 10 mil e 50 mil), o IMRAprioriAcc Spark torna-se a melhor alternativa. Porém, para valores de suporte mínimo baixo, o CPA Spark é o que apresenta menor tempo de execução, seguido pelo IMRAprioriAcc-CPA Spark.

Em todos os experimentos realizados, pelo menos um algoritmo Spark foi mais rápido do que todos os algoritmos no Hadoop-MapReduce. Entretanto, as implementações no Spark têm mais limitações devido ao tamanho da memória disponível. Para os casos em que foram gerados muitos *itemsets* frequentes (≈200 mil), o CPA no Hadoop-MapReduce obteve o melhor desempenho. Quando a quantidade de *itemsets* frequentes produzida

foi entre 50 mil e 200 mil, o CPA Spark foi o mais eficiente. Entre 10 mil e 50 mil, o IMRAprioriAcc Spark foi a melhor opção. Abaixo disso, o algoritmo que apresentou melhor desempenho foi o DPC Spark. A Figura 39 apresenta uma síntese dos resultados, com o algoritmo que obteve o melhor desempenho, de acordo com a quantidade de *itemsets* frequentes produzida. Nota-se que das quatro situações (*baixa, média-baixa, média-alta e alta quantidade de itemsets*), três dos algoritmos com melhor desempenho são Spark, enquanto que o CPA no Hadoop obteve melhor resultado para *alta quantidade de itemsets*.

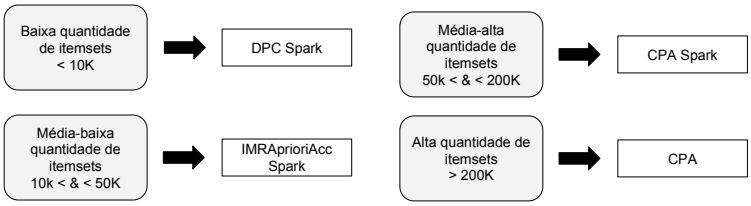


Figura 39 Síntese dos casos em cada algoritmo obteve o melhor desempenho, baseado na quantidade de *itemsets* frequentes.

5.5 Framework para recomendação de algoritmos

De acordo com os resultados dos experimentos, é apresentado no Algoritmo 25 um *framework* para recomendar a melhor implementação para cada execução, com base nas características da base de dados e no suporte mínimo exigido. A recomendação está baseada no potencial para geração de *itemsets* frequentes, generalizando os dados dos nossos experimentos.

A recomendação é dividida em três níveis de suporte mínimo. Para o valores maiores (maior que 0.5%), é recomendado usar o IMRAprioriAcc Spark. Para valores intermediários (entre 0.05% e 0.5%), se a base de dados tiver poucos itens por transação é recomendado o IMRAprioriAcc Spark.

Algoritmo 25: Framework para recomendação de algoritmos

```
   Input: Suporte mínimo (ms);
          número de itens por transação
          (it)
   Output: Algoritmo recomendado
 1 if (ms >= 0.5) then
 2 |  return "IMRAprioriAcc Spark";
 3 else
 4 |  if (ms >= 0.05) then
 5 |  |  if (it < 20) then
 6 |  |  |  return "IMRAprioriAcc
   |  |  |  Spark";
 7 |  |  else
 8 |  |  |  if (memory > 8GB)
   |  |  |  then
 9 |  |  |  |  return "CPA
   |  |  |  |  Spark";
10 |  |  |  else
11 |  |  |  |  return "CPA";
12 |  |  |  endif
13 |  |  endif
14 |  else
15 |  |  if (it < 10 or memory >
   |  |  8GB) then
16 |  |  |  return "CPA Spark";
17 |  |  else
18 |  |  |  return "CPA";
19 |  |  endif
20 |  endif
21 endif
```

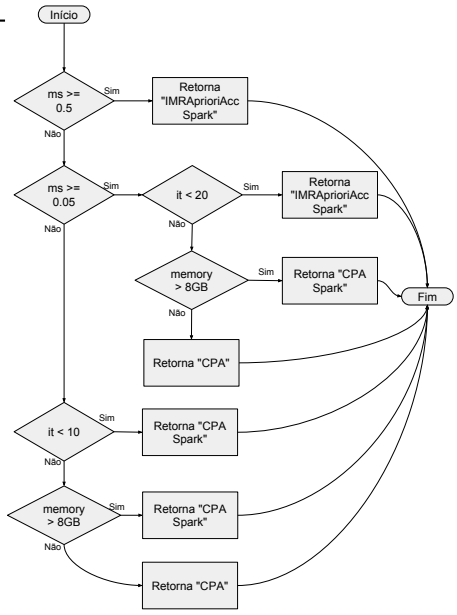


Figura 40 Fluxograma para o framework de recomendação.

Para mais itens por transação é recomendado o CPA Spark se houver ao menos 8Gb de memória principal disponível por máquina, caso contrário, é recomendado o CPA no Hadoop-MapReduce. Para valores de suporte mínimo mais baixo (menor que 0.05%), se houver memória suficiente ou poucos itens por transação é recomendado o CPA Spark, caso contrário, a melhor opção é o CPA no Hadoop-MapReduce.

Um fluxograma do framework de recomendação é apresentado na Figura 40.

6 AVALIAÇÃO EXPERIMENTAL DOS ALGORITMOS DE RESOLUÇÃO DE ENTIDADES

Nesta seção, é apresentada a metodologia para a avaliação experimental dos algoritmos de resolução de entidade aplicados para o problema de classificação de ofertas de produtos, bem como, os resultados dos experimentos para as adaptações no Hadoop-MapReduce e Spark. Também são apresentadas as comparações com o algoritmo sequencial e os algoritmos da biblioteca *MLlib*.

6.1 Metodologia para os experimentos do algoritmo de resolução de entidades no Hadoop e Spark

O algoritmo de resolução de entidades aplicado para ofertas de produtos em comércio eletrônico (OLIVEIRA; PEREIRA, 2014; OLIVEIRA; PEREIRA, 2017) apresenta resultados interessantes para as base de dados experimentadas em relação aos *baselines*. Porém, os autores observaram que para bases de dados maiores, o algoritmo tende a demandar muito tempo para apresentar o resultado final, tornando-o inviável para essas situações. Por exemplo, para a base de dados *UOL-books* (ver Seção 6.1.1), o algoritmo levou mais de 33 horas somente na etapa de treinamento e mais 4 horas na etapa de teste considerando somente a classificação por regras.

Portanto, a hipótese é que as implementações do algoritmo utilizando Hadoop-MapReduce e Spark possam processar base de dados maiores em tempo hábil, sem comprometer a qualidade dos resultados. As variáveis independentes para este contexto foram definidas como sendo as diferentes bases de dados utilizadas e apresentadas na Seção 6.1.1. Além do tempo de execução e as métricas SpeedUp, SizeUp e ScaleUp (Seção 5.1.2), como va-

riáveis dependentes, também foram medidos os valores das métricas microF_1 e macroF_1 (BAEZA-YATES; RIBEIRO-NETO, 2011, p 328), apresentadas na Seção 6.1.3.

Os experimentos com os algoritmos de resolução de entidades foram realizados em um cluster diferente do utilizado para os experimentos das implementações distribuídas do algoritmo Apriori. Logo, as variáveis controladas para este experimento são diferentes, como apresentado na Tabela 8. Oliveira e Pereira (2014) observaram que o tamanho máximo ideal para os itemsets é 3, pois para valores maiores os resultados se mantêm estáveis e para valores menores a qualidade da classificação é reduzida. O mesmo ocorre para o valor do suporte mínimo, conforme aumenta, a qualidade da classificação aumenta, porém a quantidade de entidades classificadas por regras diminui. Com o valor do suporte mínimo menor, mais regras são produzidas e a qualidade da classificação diminui. Por tanto, o valor ideal para o suporte mínimo encontrado pelos autores é de 30%.

Tabela 8 Variáveis controladas utilizadas nos experimentos dos algoritmos de resolução de entidades.

VARIÁVEL CONTROLADA	**VALOR**
Suporte mínimo	30%
Tamanho máximo do itemset	3
Maps e Reduces	8
Máquinas no cluster	20 (1 mestre, 20 escravas)
Processador	Intel Core i7-4770, 4 núcleos a 3.40GHz
Memória	8Gb (16Gb mestre) a 1333MHz
Disco rígido	500Gb
Sistema operacional	Ubuntu Desktop 14.04 LTS
Hadoop	2.7.2
Spark	1.6.1
Java	8u91

Além da comparação com o algoritmo sequencial, os algoritmos distribuídos também foram comparados com alguns *baselines* disponíveis na biblioteca *MLlib* do Spark (Seção 2.3.3). Foram utilizados os algoritmos RandomForest (Seção 2.4.2.1) e *Naive Bayes* (Seção 2.4.2.2). Logo, o total de algoritmos experimentos é 5, sendo o algoritmo sequencial, as duas adaptações para o Hadoop-MapReduce e Spark e os 2 algoritmos da *MLlib*.

6.1.1 Base de dados

As bases de dados utilizadas nesses experimentos são as mesmas utilizadas em Oliveira e Pereira (2014), apresentadas na Tabela 9, exceto a base *Uol-books-2* que foi coletada neste trabalho, contendo mais instâncias e tokens por instância do que a base *Uol-books*. Os dados foram obtidos por meio de *Web Crawling*. Cada instância é constituída por um identificador da entidade (tratado aqui como classe), pelo identificador da instância (*instanciaId*) e por sua descrição textual.

Tabela 9 Bases de dados para os experimentos dos algoritmos de resolução de entidades.

NOME	Nº DE INSTÂNCIAS	Nº DE CLASSES	MÉDIA DE *TOKENS*
Printers	2.167	157	7,5
UOL-eletronics	9.552	2.218	12,6
UOL-non-eletronics	26.640	5.299	6,42
UOL-books	385.797	93.886	5,72
UOL-books-2	604.922	99.591	7,33
UOL-books-2-rep5	3.024.610	99.591	7,33

Para o experimento de coleta do tempo de execução, cada uma das bases de dados foram executadas 3 vezes e calculada a média do tempo das três execuções como resultado final. Logo, a quantidade de experimentos

para esse caso foi de 75, definida pela Equação 17. Assim como os algoritmos distribuídos do Apriori, também foi realizado teste de significância estatística t-teste (JAIN, 1991, p 209) com algoritmos de resolução de entidades.

$$\begin{aligned} \textit{Quantidade de experimentos} = (5\ \textit{base de dados}) \times \\ (5\ \textit{algoritmos}) \times (3\ \textit{execuções}) \end{aligned} \tag{17}$$

Entretanto, como é visto na Seção 6.2 somente dois algoritmos conseguiram processar as bases de dados *Uol-books* e *Uol-books-2* reduzindo a quantidade total de experimentos para a coleta de tempo de execução para 57.

6.1.2 Baselines

As implementações na biblioteca *MLlib* permitem a utilização de alguns parâmetros para otimizar os resultados de acordo com as bases de dados. Antes da execução dos experimentos foram feitas buscas pelos melhores parâmetros de cada algoritmo, um processo conhecido como *grid search*. Para cada algoritmo e base de dados foi realizada a validação cruzada (*cross-validation*) para cada variação do valor dos parâmetros. Variou-se um parâmetro por vez enquanto os demais permaneceram com valores padrão. Conforme eram encontrados os valores que produziam os melhores resultados para determinado parâmetro, definia-se o parâmetro com o dado valor e iniciava-se a busca dos valores para o próximo parâmetro. Para cada novo valor que um parâmetro assumia é executada a *cross-validation* com a base de dados para verificar a qualidade dos resultados.

Nesse processo, foi observado que alguns parâmetros não influenciam de maneira significativa a qualidade dos resultados, porém impactam no tempo de execução do algoritmo. O número de árvores do algoritmo *Random Tree*, por exemplo, é um parâmetro que impacta no tempo de execução do algoritmo caso assuma valores altos sem refletir de maneira significativa na qualidade dos resultados para as bases de dados experimentadas. Valores muito pequenos para esse parâmetro tendem a degradar a qualidade dos resultados, portanto foi definido um valor que visa manter um equilíbrio entre tempo de execução e qualidade dos resultados. As Tabelas 10 e 11 listam os melhores parâmetros para o *Naive Bayes* e *Random Forest* referentes a cada base de dados avaliada. Não foi possível executar o *grid search* para as bases de dados *Uol-books* e *Uol-books-2*, pois ocorreu estouro de memória devido às limitações de *hardware* do cluster.

Tabela 10 Parâmetros para o *Naive Bayes* da biblioteca *MLlib*.

BASE DE DADOS	LAMBDA	MODAL TYPE
Printers	0.0088	multinomial
UOL-eletronics	0.001	multinomial
UOL-non-eletronics	0.0001	multinomial

Tabela 11 Parâmetros para o *Random Forest* da biblioteca *MLlib*.

BASE DE DADOS	NUM TREES	IMPURUTY	DEPTH	BINS
Printers	50	gini	30	12
UOL-eletronics	50	gini	30	82
UOL-non-eletronics	50	gini	30	2

Para os algoritmos da *MLlib*, os dados precisam ser organizados em atributos (*features*) no formato *Labeled Point* ou LIBSVM. As *features* são representadas por cada *token* distinto do conjunto de treinamento e os valores correspondem aos *idfs* associados a cada token. Portanto, é necessária

uma etapa de pré-processamento para transformar os dados brutos no formato adequado. O tempo gasto nessa etapa é incorporado ao tempo total do algoritmo.

6.1.3 Métricas de avaliação

Todas as bases de dados foram divididas em partes para treino e teste de acordo com método *cross-validation*, dividindo cada arquivo em 10 partes, sendo uma para teste e as noves restantes para treino. É um processo cíclico em que cada execução do algoritmo testa com uma das 10 partes, efetuando o processo de aprendizagem com as 9 partes restantes, até que todas as partes tenham sido testadas. A *cross-validation* foi utilizada para avaliar a qualidade dos resultados mensurada pelas métricas microF_1 e macroF_1 (BAEZA-YATES; RIBEIRO-NETO, 2011, p 328). Ambas são baseadas na medida F_1 que representa uma média harmônica entre a Precisão e Revocação (BAEZA-YATES; RIBEIRO-NETO, 2011, p 135). A Precisão, apresentada na Equação 18, é a fração de todas as instâncias atribuídas à classe c pelo algoritmo que realmente pertence a esta classe, de acordo com o conjunto de teste.

$$P_{(c)} = \frac{n_{f,t}}{n_f} \tag{18}$$

O numerador $n_{f,t}$ é a quantidade de instâncias que ambos, o conjunto de teste e o algoritmo atribuíram à classe c, ou seja, é o número de instâncias que o algoritmo acertou. E o denominador n_f é a quantidade de instâncias associadas à classe c pelo algoritmo. Revocação é a fração de todas as instâncias que pertencem à classe c de acordo com o teste que foi

corretamente atribuído à classe c pelo algoritmo, apresentada na Equação 19.

$$R_{(c)} = \frac{n_{f,t}}{n_t} \tag{19}$$

O denominador n_t representa a quantidade de instâncias que realmente pertencem à classe c.

A medida F_1 equilibra a importância relativa da precisão e revocação, apresentada na Equação 20.

$$F_{1(c)} = \frac{2P_{(c)}R_{(c)}}{P_{(c)} + R_{(c)}} \tag{20}$$

A macroF_1 corresponde à média simples de F_1 sobre todas as classes $|C|$, apresentada na Equação 21.

$$macroF_1 = \frac{\sum_{i=1}^{|C|} F_{1(c_i)}}{|C|} \tag{21}$$

A microF_1 corresponde ao valor global de F_1 obtida pela precisão e revocação calculadas sobre todas as classes, como apresentada na Equação 22.

$$P = \frac{\sum_{c \in C} n_{f,t}}{\sum_{c \in C} n_f} \qquad R = \frac{\sum_{c \in C} n_{f,t}}{\sum_{c \in C} n_t} \tag{22}$$

Com a média geral da precisão e revocação em relação a todas as classes, é possível obter a métrica microF$_1$, como apresentada na Equação 23

$$microF_1 = \frac{2PR}{P+R} \tag{23}$$

Para obter os resultados finais das métricas microF$_1$ e macroF$_1$, foram calculadas as médias para as 10 execuções do *cross-validation* para cada base de dados. Também foi realizado teste de significância estatística t-teste para essas métricas.

Para a avaliação dos algoritmos utilizando as métricas de computação distribuída, foi utilizada a base de dados *UOL-books-2* variando apenas a quantidade de instâncias entre 25k, 50k, 100k, 200k e 400k. Para o *SpeedUp* a variação de máquinas foi semelhante ao experimento com as implementações do Apriori, variando entre 1, 2, 4, 8 e 16 máquinas. Para o *ScaleUp* a quantidade de instâncias variou proporcionalmente à quantidade de máquinas, de modo a medir a escalabilidade dos algoritmos.

Como discorrido na Seção 6.3, os experimentos para as métricas *SpeedUp*, *SizeUp* e *ScaleUp* foram realizados somente para as implementações do algoritmo de resolução de entidades no Hadoop-MapReduce e Spark, logo, o total de experimentos para cada métrica foi de 30. Para as métricas microF$_1$ e macroF$_1$, foram realizados 190 experimentos ao todo considerando que somente 2 algoritmos executaram para as bases *Uol-books* e *Uol-books-2*.

6.2 Tempo de execução

As médias dos tempos de execução dos algoritmos para a base de dados *Printers* são apresentados na Figura 41. Nota-se que devido ao pe-

queno tamanho do conjunto de dados, o algoritmo sequencial foi estatisticamente o mais rápido dentre os demais. O *Naive Bayes* da *MLlib* e a implementação no Spark foram o segundo e o terceiro mais rápidos, respectivamente. O *Random Forest* foi o algoritmo que mais demandou tempo dentre as implementações que utilizam o Spark, pois a quantidade de árvores e a profundidade necessárias para produzir os melhores resultados exigem maior tempo para processamento. Isso se reflete no tempo gasto na etapa de treinamento que somou mais de 90% do tempo total do algoritmo. O Hadoop-MapReduce, por necessitar ler e persistir arquivos a cada fase e entre cada função Map e Reduce, demandou bastante tempo para leitura e escrita do que para processar a base efetivamente.

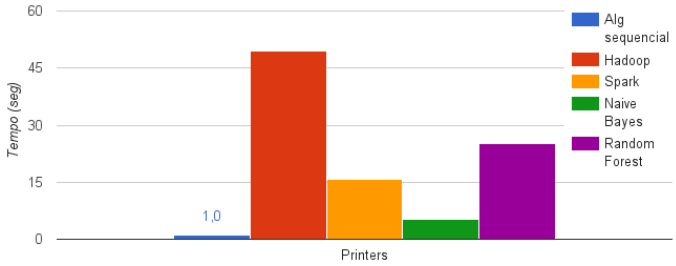


Figura 41 Tempo de execução dos algoritmos para a base de dados *Printers*.

Para a base de dados *Uol-eletronics* o algoritmo sequencial começa a apresentar maiores demandas de tempo para processar devido a uma maior quantidade de instâncias. O gráfico com as médias dos tempos de execução são apresentados na Figura 42. Nota-se que a partir dessa base de dados, os algoritmos distribuídos tendem a ser mais eficientes em relação ao algoritmo sequencial. O *Naive Bayes* foi o algoritmo que apresentou estatisticamente o melhor tempo de execução, sendo 385% mais rápido do que a implementação no Spark e 1100% mais rápido do que a implementação no Hadoop-

MapReduce. Porém, como apresentado na Seção 6.4, a qualidade da classificação é inferior aos algoritmos mapeados para o Hadoop-MapReduce e Spark. O algoritmo sequencial foi o quarto mais lento, seguido pelo *Random Forest* que, por sua vez, demandou mais de 400 segundos para processar os dados, sendo a maior parte do tempo gasta na etapa de treinamento.

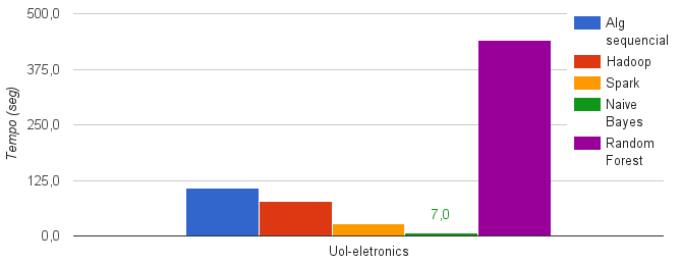


Figura 42 Tempo de execução dos algoritmos para a base de dados *Uol-eletronics*.

O comportamento dos algoritmos na base *Uol-non-eletronics*, ilustrado na Figura 43, demonstra maior eficiência por parte das implementações distribuídas. A média do tempo gasto pelo algoritmo sequencial para processar os dados foi significantemente maior do que nas bases de dados anteriores, tornando-o o algoritmo mais lento. O *Naive Bayes* foi o mais rápido, sendo 220% mais rápido que a implementação no Spark e 725% mais rápido do que a implementação no Hadoop-MapReduce. O comportamento do *Random Forest* segue como o algoritmo mais lento dentre as implementações distribuídas.

Para as bases de dados *Uol-books* e *Uol-books-2*, o algoritmo sequencial levou mais de dois dias para processar uma das partições dos dados, inviabilizando a coleta do tempo de execução. Os algoritmos da biblioteca *MLlib*, *Naive Bayes* e *Random Forest* não terminaram a execução devido ao consumo de memória que extrapolou os limites de *hardware* do cluster. Ape-

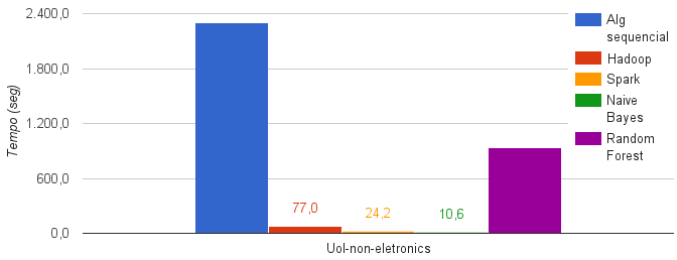


Figura 43 Tempo de execução dos algoritmos para a base de dados *Uol-non-eletronics*.

nas as implementações no Hadoop-MapReduce e Spark conseguiram processar essas duas bases de dados, como apresentado na Figura 44. Observa-se que para ambas as bases a adaptação no Spark apresentou menor tempo de execução. Para a base *Uol-books-2* a diferença de tempo é ainda maior, sendo a implementação no Hadoop-MapReduce 303% mais lenta do que a implementação no Spark.

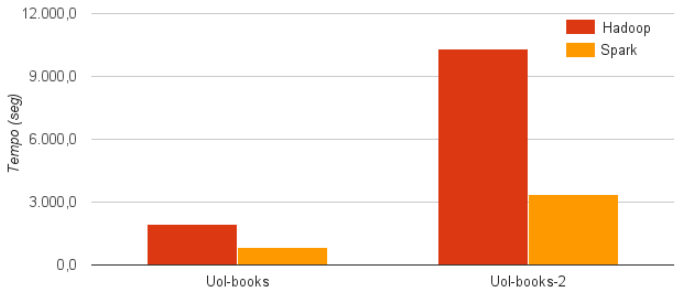


Figura 44 Tempo de execução dos algoritmos implementados no Hadoop-MapReduce e Spark para as bases de dados *Uol-books* e *Uol-books-2*.

6.3 Métricas de avaliação para algoritmos distribuídos

Para a avaliação das métricas *SpeedUp*, *SizeUp* e *ScaleUp*, foram utilizadas somente as implementações do algoritmo de resolução de entidades

no Hadoop-MapReduce e no Spark. Para os algoritmos *Naive Bayes* e *Random Forest* ocorreu estouro de memória conforme aumentava o tamanho das bases de dados ou diminuía a quantidade de máquinas escravas executando. Uma solução seria diminuir o tamanho das bases de dados utilizadas nesses experimentos para possibilitar a execução de todos os algoritmos. Porém, para bases de dados pequenas não foram observadas variações significativas no tempo de execução para diferentes quantidades de máquinas ou variações do tamanho da base de dados. Portanto, para esses experimentos foram utilizadas bases de dados maiores e avaliados somente os algoritmos mapeados para o Hadoop-MapReduce e Spark.

6.3.1 SpeedUp

O gráfico dos experimentos de SpeedUp é apresentado na Figura 45. Para a base de dados experimentada, a implementação no Hadoop-MapReduce apresentou escalabilidade crescente até 8 máquinas. Para 16 máquinas no cluster, houve queda de desempenho, isso significa que o tempo gasto para processar a mesma base de dados com 16 máquinas foi maior do que processar com 8 máquinas. A implementação no Spark apresentou escalabilidade inferior ao Hadoop-MapReduce, visto que para 8 e 16 máquinas houve queda de desempenho.

6.3.2 SizeUp

O gráfico dos experimentos de SizeUp é apresentado na Figura 46. Ambas as implementações apresentaram comportamentos estáveis conforme aumenta a base de dados. Com até 200 mil instâncias o algoritmo no Spark apresenta SizeUp ligeiramente abaixo do Hadoop-MapReduce. Entretanto,

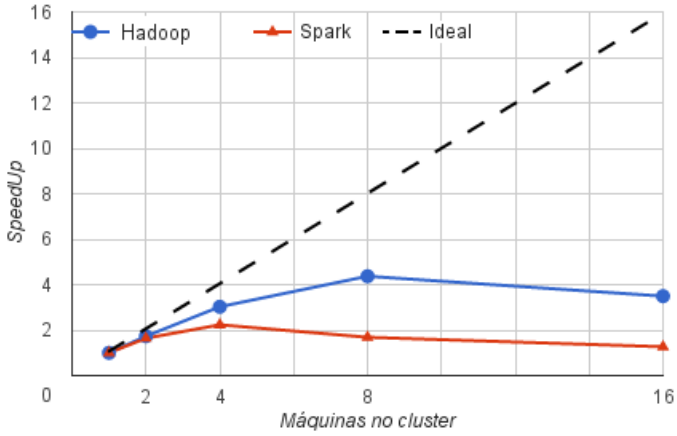


Figura 45 SpeedUp para os algoritmos no Hadoop-MapReduce e Spark.

com 400 mil instâncias o Spark apresenta maior SizeUp, mostrando que tende a sofrer mais impacto do que o Hadoop-MapReduce para maiores bases de dados com poucas máquinas no cluster devido à demanda maior de memória por máquina.

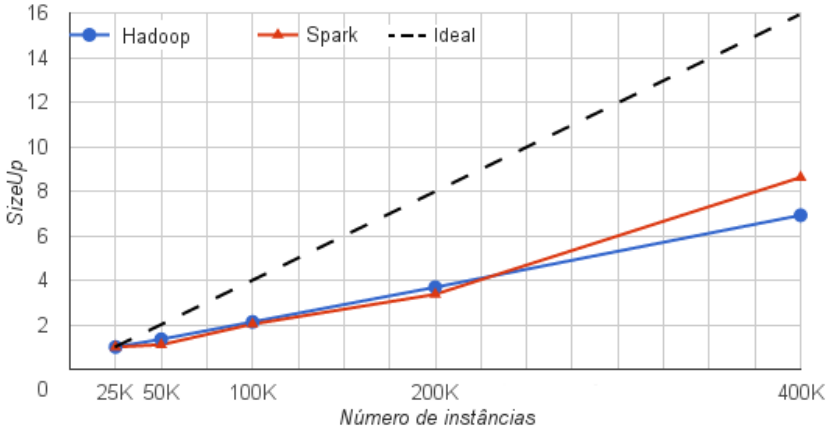


Figura 46 SizeUp para os algoritmos no Hadoop-MapReduce e Spark.

6.3.3 ScaleUp

Em relação à métrica ScaleUp, apresentada na Figura 47, nota-se que apesar de o algoritmo no Spark ser mais rápido do que a implementação no Hadoop-MapReduce, ele é menos escalável. A implementação no Hadoop-MapReduce apresenta escalabilidade estável para até 8 máquinas e 200 mil instâncias. A queda significativa de performance ocorreu com 16 máquinas e 400 mil instâncias, pois, apesar de várias máquinas processarem de maneira independente cada porção de dados, é necessário construir estruturas, proporcionalmente grandes à quantidade de instâncias, em cada máquina. Isso implica que é necessário adequar a quantidade de máquinas escravas em relação à base de dados a ser processada com o propósito de minimizar a degradação de performance. Ou seja, nem sempre utilizar muitas máquinas é sinônimo de aumento de performance, pois o custo para distribuição dos dados, inicialização das aplicações, comunicação em rede, criação e manutenção de estruturas em memória é maior à medida que se aumenta o número de máquinas no cluster.

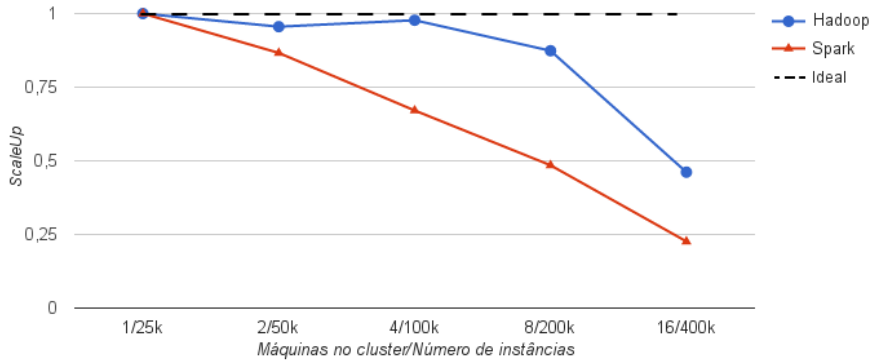


Figura 47 ScaleUp para os algoritmos no Hadoop-MapReduce e Spark.

6.4 Métricas MicroF$_1$ e MacroF$_1$

A Tabela 12 apresenta os valores das métricas microF$_1$ e macroF$_1$ para todos os algoritmos e as bases de dados *Printers, Uol-eletronics* e *Uol-non-eletronics*. Observa-se que tanto o algoritmo sequencial quanto suas adaptações para o Hadoop-MapReduce e Spark apresentam valores equivalentes para as métricas nas 3 bases de dados. O algoritmo sequencial e suas adaptações obtiveram melhores resultados que os *baselines*. Esses resultados foram avaliados estatisticamente com um intervalo de confiança de 95%.

Tabela 12 Métricas microF$_1$ e macroF$_1$ para todos os algoritmos.

ALGORITMO	**Printers**		**Uol-eletronics**		**Uol-non-eletronics**	
	MacroF$_1$	**MicroF$_1$**	**MacroF$_1$**	**MicroF$_1$**	**MacroF$_1$**	**MicroF$_1$**
Sequencial Hadoop-Map-Reduce Spark	97.14%	97.85%	87.14%	91.77%	80.84%	86.73%
Naive Bayes	92.25%	92.34%	71.50%	74,63%	67.89%	70.82%
Random Forest	76.77%	76.74%	10.51%	12.06%	2.85%	5.63%

Como já mencionado, para as base *Uol-books* e *Uol-books-2*, o algoritmo sequencial gastou mais de dois dias para processá-las enquanto que os algoritmos da *MLlib* extrapolaram os limites de memória. Portanto, na Tabela 13 são apresentados os resultados das métricas microF$_1$ e macroF$_1$ para o Hadoop-MapReduce e Spark.

Tabela 13 Métricas microF$_1$ e macroF$_1$ para as implementações no Hadoop-MapReduce e Spark.

ALGORITMO	**Uol-books**		**Uol-books-2**	
	MacroF$_1$	**MicroF$_1$**	**MacroF$_1$**	**MicroF$_1$**
Hadoop-Map-Reduce Spark	87.02%	91.68%	97,46%	98,31%

6.5 Comparação entre as implementações

Para as bases de dados relativamente pequenas, como a *Printers*, o algoritmo sequencial é suficiente para produzir bons resultados em tempo hábil. Para as demais bases de dados a adaptação no Spark se sobressaiu perante às demais em relação ao tempo e qualidade dos resultados. Portanto, a implementação no Spark é a mais adequada dentre os algoritmos distribuídos desde que tenha memória principal disponível.

Conforme aumenta a base de dados, mais memória é requisitada para executar o algoritmo no Spark. O Hadoop-MapReduce, por trabalhar com arquivos e não necessitar de memória tanto quanto o Spark, pode se tornar uma opção viável em situações que houver limitações de memória.

7 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho, foi realizado um estudo detalhado de diversas implementações do algoritmo Apriori para o *framework* Hadoop-MapReduce, bem como de suas adaptações para o *framework* Spark. Foram comparadas implementações que produzem os *itemsets* frequentes em duas ou k fases, onde cada fase é uma iteração MapReduce. As implementações foram comparadas usando-se valores distintos para o suporte mínimo e diferentes bases de dados, variando-se a quantidade de transações, a quantidade de itens distintos e a quantidade de itens por transação.

Para o Hadoop-MapReduce, observou-se que quando a quantidade de *itemsets* frequentes produzida é baixa, o IMRAprioriAcc, que é um algoritmo de duas fases, obtém melhor desempenho do que as outras abordagens de k fases. Por outro lado, o algoritmo CPA se sobressai quando a quantidade de *itemsets* produzida é grande. O CPA também é o algoritmo com melhor escalabilidade.

Para o Spark, os algoritmos adaptados apresentaram um comportamento semelhante ao do Hadoop-MapReduce, exceto que o DPC também apresentou um bom desempenho quando a quantidade de *itemsets* frequentes foi baixa. O CPA, que obteve o melhor desempenho para grandes quantidades de *itemsets* frequentes, teve escalabilidade instável.

Comparando o desempenho dos algoritmos entre o Hadoop-MapReduce e o Spark, as implementações do algoritmo Apriori no Spark obtiveram um melhor desempenho. Entretanto, o Spark utiliza RDDs, que mantém os dados em memória, e assim, se a quantidade de memória nas máquinas do cluster não for suficiente para processar grandes bases de dados, é recomendável utilizar a implementação do CPA no Hadoop-MapReduce.

Como resultado da avaliação, foi produzido um *framework* para recomendar aos usuários o algoritmo a ser aplicado, de acordo com as características da base de dados e do valor do suporte mínimo desejado.

Também neste trabalho, foi realizado um estudo e adaptação para o Hadoop-MapReduce e Spark do algoritmo de resolução de entidades aplicada para classificação de ofertas de produtos. Os algoritmos foram comparados em termos de tempo de execução e qualidade da classificação para diversas bases de dados. Também foram utilizados os algoritmos *Naive Bayes* e *Random Forest* da biblioteca *MLlib* do Spark como *baselines* para comparação e avaliação das implementações. Observou-se que não é viável utilizar os algoritmos distribuídos para base de dados muito pequenas pois o custo da distribuição de processamento, comunicação e sincronização das máquinas do clusters se sobrepõe ao custo necessário para processar a base de dados efetivamente. Conforme a base de dados cresce, a implementação no Spark é preferível em relação às demais por manter um bom equilíbrio entre tempo de execução e qualidade da classificação.

Como trabalhos futuros, é possível elaborar uma estratégia que aproveite o bom desempenho do Spark, mas que também possa armazenar dados em disco quando o processamento em memória for um limite. Especificamente, para os algoritmos de resolução de entidades, é interessante trabalhar em uma estratégia que seja mais escalável do que o algoritmo Spark para que seu comportamento seja mais estável conforme varia a quantidade de máquinas e o tamanho da base de dados.

Além disso, é interessante avaliar o desempenho do algoritmo de resolução de entidades para outros tipos de bases de dados, além de oferta de produtos, de forma a avaliar sua generalidade. Também, é importante

torná-lo incremental, de forma a reconhecer novas classes para as quais ele não tenha sido treinado.

REFERÊNCIAS

AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. Mining association rules between sets of items in large databases. **ACM SIGMOD Record**, New York, v. 22, n. 2, p. 207–216, June 1993.

AGRAWAL, R.; SHAFER, J. Parallel mining of association rules. **IEEE Transactions on Knowledge and Data Engineering**, New York, v. 8, n. 6, p. 962–969, Dec. 1996.

AGRAWAL, R.; SRIKANT, R. Fast algorithms for mining association rules in large databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 20., 1994, USA. **Proceedings...** USA: Morgan Kaufmann Publishers, 1994. v. 1215, p. 487–499.

APACHE Hadoop. HDFS Federation. **Apache Hadoop**, United States, 2015b. Disponível em: <http://hadoop.apache.org/docs/stable2/hadoop-project-dist/hadoop-hdfs/Federation.html>. Acesso em: 15 fev. 2015.

APACHE Hadoop. What is apache hadoop. **Apache Hadoop**, United States, p. 8, 2015a. Disponível em: <http://hadoop.apache.org/\#What+Is+Apache+Hadoop>. Acesso em: 15 fev. 2015.

APACHE Spark. Spark MLlib: RDD-based API. **Apache Spark**, New York, 2016a. Disponível em: <http://spark.apache.org/docs/latest/mllib-guide.html>. Acesso em: 15 jul. 2016.

APACHE Spark. Spark programming guide. **Apache Spark**, New York, 2016b. Disponível em: <spark.apache.org/docs/latest/programming-guide.html>. Acesso em: 15 jul. 2016.

APACHE Yarn. Apache Hadoop NextGen MapReduce (YARN). **Apache Hadoop**, United States, 2015c. Disponível em: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarnsite/YARN.html>. Acesso em: 15 fev. 2015.

BAEZA-YATES, R.; RIBEIRO-NETO, B. **Modern information retrieval**: the concepts and cechnology behind search. 2. ed. Harlow: Pearson Education, 2011. 913 p.

BILENKO, M.; MOONEY, R. J. Adaptive duplicate detection using learnable string similarity measures. In: ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, 9., 2003, New York. **Proceedings...** USA: ACM, 2003. p. 39–48.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. **Communications of the ACM**, New York, v. 13, n. 7, p. 422–426, July 1970.

BOLINA, A. C. et al. Uma nova proposta de paralelismo e balanceamento de carga para o algoritmo apriori. **Revista de Sistemas de Informação da FSMA**, Rio de Janeiro, n. 11, p. 33–41, 2013.

BORTHAKUR, D. et al. Apache hadoop goes realtime at facebook. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2011, New York. **Proceedings...** USA: ACM, 2011. p. 1071–1080.

CHANG, C.-C.; LIN, C.-J. **LIBSVM**: A library for support vector machines. [S.l.: s.n], p. 38, 2016. Disponível em: <http://www.csie.ntu.edu.tw/cjlin/libsvm/>. Acesso em: 15 jul. 2016.

CHEUNG, D.; LEE, S.; XIAO, Y. Effect of data skewness and workload balance in parallel data mining. **IEEE Transactions on Knowledge and Data Engineering**, New York, v. 14, n. 3, p. 498–514, May 2002.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In: CONFERENCE ON SYMPOSIUM ON OPEARTING SYSTEMS DESIGN & IMPLEMENTATION, 6., 2004, Berkeley. **Proceedings...** USA: USENIX Association, 2004. p. 395–408.

FARZANYAR, Z.; CERCONE, N. Accelerating frequent itemsets mining on the cloud: a mapreduce-based approach. In: INTERNATIONAL CONFERENCE ON DATA MINING WORKSHOPS, 13., 2013, Dallas. **Proceedings...** Dallas: IEEE, 2013a. p. 592–598.

FARZANYAR, Z.; CERCONE, N. Efficient mining of frequent itemsets in social network data based on mapreduce framework. In: INTERNATIONAL CONFERENCE ON ADVANCES IN SOCIAL NETWORKS ANALYSIS AND MINING, 2013, New York. **Proceedings...** USA: ACM, 2013b. p. 1183–1188.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: ACM SIGOPS OPERATING SYSTEMS REVIEW, 3., 2003, New York. **Proceedings...** USA: ACM, 2003. p. 29–43.

GRONINGEN, M. V. Introduction to hadoop. **Search Workings**, Washington, 2009. Disponível em: <http://www.searchworkings.org/blog/-/blogs/introduction-to-hadoop/>. Acesso em: 15 fev. 2015.

HADOOP Wiki. WordCount example. **Hadoop Wiki**, New York, 2015. Disponível em: <http://wiki.apache.org/hadoop/WordCount>. Acesso em: 15 fev. 2015.

HAHSLER, M. et al. **Introduction to arules**: a computational environment for mining association rules and frequente itemsets. [S.l.: s.n.], p. 37, 2016. Disponível em: <https://cran.rproject.org/web/packages/arules/vignettes/arules.pdf>. Acesso em: 20 jan. 2016.

HAN, J.; KAMBER, M.; PEI, J. **Data Mining**: concepts and techniques. 3. ed. San Francisco: Morgan Kaufmann Publishers, 2011. 744 p.

HAN, J.; PEI, J.; YIN, Y. Mining frequent patterns without candidate generation. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2000, New York. **Proceedings...** USA: ACM, 2000. p. 1–12.

HE, Q. et al. Parallel implementation of classification algorithms based on mapreduce. In: WANG, G. et al. (Ed.). **Rough Set and Knowledge Technology**. Berlin: Springer Berlin Heidelberg, 2010. p. 655–662.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture**: a quantitative approach. 4th. ed. San Francisco: Elsevier, 2012. 856 p.

HORTONWORKS. An introduction to HDFS Federation. **Hortonworks**, USA, 2015. Disponível em: <http://hortonworks.com/blog/an-introduction-to-hdfs-federation/>. Acesso em: 15 fev. 2015.

JAIN, R. **The art of computer systems performance analysis**: techniques for experimental design, measurement, simulation, and modeling. USA: John Wiley & Sons, 1991. 720 p.

KARAU, H. et al. **Learning spark**: lightning-fast big data analytics. USA: O'Reilly Media, 2015. 274 p.

LAM, C. **Hadoop in action**. Stamford: Manning Publications, 2010. 325 p.

LI, L.; ZHANG, M. The strategy of mining association rule based on cloud computing. In: INTERNATIONAL CONFERENCE ON BUSINESS COMPUTING AND GLOBAL INFORMATIZATION, 2011, Washington. **Proceedings...** USA: IEEE Computer Society, 2011. p. 475–478.

LI, N. et al. Parallel implementation of apriori algorithm based on mapreduce. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ARTIFICIAL INTELLIGENCE, NETWORKING AND PARALLEL DISTRIBUTED COMPUTING, 13., 2012, Kyoto. **Proceedings...** Kyoto: IEEE, 2012. p. 236–241.

LIN, M.-Y.; LEE, P.-Y.; HSUEH, S.-C. Apriori-based frequent itemset mining algorithms on mapreduce. In: INTERNATIONAL CONFERENCE ON UBIQUITOUS INFORMATION MANAGEMENT AND COMMUNICATION, 6., 2012, New York. **Proceedings...** New York: ACM, 2012. p. 1–8.

MAZUR, E. et al. Scalla: A platform for scalable one-pass analytics using mapreduce. **ACM Transactions on Database Systems**, New York, v. 37, n. 4, p. 27:1–27:43, Dec. 2012.

OLIVEIRA, C. M.; PEREIRA, D. A. **Um método baseado em regras de associação para classificação de ofertas de produtos em lojas de comércio eletrônico**. 2014. 79 p. Dissertação (Mestrado) — Universidade Federal de Lavras, Lavras, 2014.

OLIVEIRA, C. M.; PEREIRA, D. A. An association rules based method for classifying product offers from e-shopping. **Intelligent Data Analysis**, Oxford, v. 21, n. 3, 2017.

PIATESKI, G.; FRAWLEY, W. **Knowledge Discovery in Databases**. Cambridge: MIT Press, 1991. 625 p.

QIU, H. et al. Yafim: A parallel frequent itemset mining algorithm with Spark. In: INTERNATIONAL DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS, 28., 2014, Phoenix. **Proceedings...** Phoenix: [s.n.], 2014. p. 1664–1671.

RATHEE, S.; KAUL, M.; KASHYAP, A. R-apriori: An efficient Apriori based algorithm on Spark. In: WORKSHOP IN INFORMATION AND

KNOWLEDGE MANAGEMENT, 8., 2015, Melbourne. **Proceedings...** Melbourne: ACM, 2015. p. 27–34.

SEBASTIANI, F. Machine learning in automated text categorization. **ACM Computing Surveys**, New York, v. 34, n. 1, p. 1–47, Mar 2002.

VAVILAPALLI, V. K. et al. Apache hadoop yarn: Yet another resource negotiator. In: ANNUAL SYMPOSIUM ON CLOUD COMPUTING, 4., 2013, New York. **Proceedings...** New York: ACM, 2013. p. 1–16.

WHITE, T. **Hadoop**: The definitive guide. 3. ed. USA: O'Reilly Media, 2012. 688 p.

WITTEN, I. H.; FRANK, E.; HALL, M. A. **Data mining**: pratical machine learning tools and techniques. USA: Morgan Kaufmann, 2011. 664 p.

WU, G. et al. Mrec4.5: C4.5 ensemble classification with mapreduce. In: CHINAGRID ANNUAL CONFERENCE, 2009, Yantai. **Proceedings...** Yantai: IEEE, 2009. p. 249–255.

YAHYA, O.; HEGAZY, O.; EZAT, E. An efficient implementation of apriori algorithm based on hadoop-mapreduce model. **International Journal of Reviews in Computing**, Oxford, v. 12, n. 7, p. 59–67, Dec. 2012.

YANG, X. Y.; LIU, Z.; FU, Y. Mapreduce as a programming model for association rules algorithm on hadoop. In: INTERNATIONAL CONFERENCE ON INFORMATION SCIENCES AND INTERACTION SCIENCES, 3., 2010, Chengdu. **Proceedings...** Chengdu: IEEE, 2010. p. 99–102.

YE, Y.; CHIANG, C.-C. A parallel apriori algorithm for frequent itemsets mining. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS, 4., 2006, Seattle. **Proceedings...** Seattle: IEEE, 2006. p. 87–94.

YU, K.-M.; ZHOU, J.-L. A weighted load-balancing parallel apriori algorithm for association rule mining. In: IEEE INTERNATIONAL CONFERENCE ON GRANULAR COMPUTING, 2008, Hangzhou. **Proceedings...** Hangzhou: IEEE, 2008. p. 756–761.

ZAHARIA, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 9., 2012, Berkeley. **Proceedings...** Berkeley: USENIX Association, 2012. p. 2.

ZAHARIA, M. et al. Spark: cluster computing with working sets. In: USENIX CONFERENCE ON HOT TOPICS IN CLOUD COMPUTING, 2., 2010, Berkeley. **Proceedings...** Berkeley: USENIX Association, 2010.

ZAKI, M. J. et al. Parallel data mining for association rules on shared-memory multi-processors. In: CONFERENCE ON SUPERCOMPUTING, 1996, Washington. **Proceedings...** Washington: IEEE Computer Society, 1996. p. 1–33.

ZAKI, M. J.; PARTHASARATHY, S.; LI, W. A localized algorithm for parallel association mining. In: ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 9., 1997, New York. **Proceedings...** New York: ACM, 1997. p. 321–330.

ZHOU, X.; HUANG, Y. An improved parallel association rules algorithm based on mapreduce framework for big data. In: INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS AND KNOWLEDGE DISCOVERY, 11., 2014, Xiamen. **Proceedings...** Xiamen: IEEE, 2014. p. 284–288.