PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DESCOBERTA DE CONJUNTOS DE ITENS FREQUENTES COM O MODELO DE PROGRAMAÇÃO MAPREDUCE SOBRE CONTEXTOS DE INCERTEZA

JULIANO VARELLA DE CARVALHO

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Duncan Dubugras Alcoba Ruiz

Porto Alegre
2015

FICHA CATALOGRÁFICA EMITIDA PELA BIBLIOTECA

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO
EMITIDA E ASSINADA PELA FACULDADE

Dedico este trabalho à minha mulher, Vera, ao nosso encontro, amor e parceria.

"... Todo mundo ama um dia todo mundo chora,
Um dia a gente chega, no outro vai embora
Cada um de nós compõe a sua história
Cada ser em si carrega o dom de ser capaz
E ser feliz..."
Almir Sater e Renato Teixeira

AGRADECIMENTOS

Preciso agradecer primeiramente ao companheirismo, paciência e amor da minha mulher, a minha Vera, obrigado pelo teu incentivo diário, pelo teu bom humor, pelas nossas conversas, pelas nossas Basti e Zoe. Sem a tua presença, este trabalho não seria finalizado.

O agradecimento mais do que especial vai para os meus pais, Edite e Tabajara, e aos meus dois irmãos, André Luiz e Rodrigo. Pessoas maravilhosas que moldaram meu caráter, me ensinaram com muita clareza os significados de carinho, respeito e amor. Sinto saudades de vocês todos os dias ao acordar.

Agradeço a toda família: tios, primos, sogros, cunhados e todos aqueles que estiveram próximos a mim nestes quatro anos de muitas renúncias e bastante dedicação.

Agradeço aos meus amigos que estiveram próximos durante o doutorado, peço desculpas pelas ausências e muito obrigado pelo incentivo. Em especial, ao meu amigo Eurico, com quem aprendo a cada explanação divertida e didática dada por ele, sobre qualquer assunto.

Gostaria de agradecer ao meu orientador, Prof. Dr. Duncan Dubugras Ruiz, pela oportunidade dada a mim de concluir mais esta etapa pessoal e profissional. Também preciso agradecer a sua orientação, com dicas e caminhos a seguir, que contribuíram sobremaneira para a melhoria desta pesquisa.

Agradeço ao grupo de pesquisa ao qual estive vinculado durante este período, GPIN. Agradeço a todos aqueles com quem tive contato e que foram de fundamental importância para o meu aperfeiçoamento e amadurecimento. Gostaria de citar meus amigos de GPIN: Christian, Holisson, Renata e Sílvia. Agradeço também aos meus amigos e ex-colegas de PUCRS: Ana Winck, Daiane Hemerich, Felipe Lammel, Juliano Silveira, Leandro Bogoni, Luciano Blomberg e Tiago Silva, obrigado pelas conversas e conselhos.

Agradeço ao Programa de Pós-Graduação em Ciência da Computação, PPGCC, pela oportunidade de cursar o meu doutorado em um dos melhores programas de pós-graduação do Brasil. Agradeço a todos os professores do PPGCC, principalmente àqueles com que tive aulas e que contribuíram para a minha pesquisa. Agradeço ao Prof. Dr. Rodrigo Coelho Barros pela disponibilidade em conversar a respeito da minha pesquisa e suas sugestões. Aproveito para agradecer aos secretários do PPGCC, Diego e Régis, pela disponibilidade e celeridade com que sempre responderam as minhas dúvidas.

Agradeço também ao Prof. Dr. Aad Van Moorsel e sua equipe da Universidade de Newcastle. Obrigado pela oportunidade de passar uma curta temporada em seu laboratório de pesquisa. Aproveito para agradecer aos amigos que fiz em Newcastle, em especial, Francisco Rocha e Winai Wongthai. Agradeço também a Comunidade Europeia, em função do *Seventh Framework Programme* pela ajuda de custo concedida neste período.

Gostaria também de agradecer à Universidade Feevale, por ceder seu espaço para que eu fizesse meus experimentos. Também agradeço aos meus amigos e alunos desta Universidade, os quais me ajudam, diariamente, para que eu me torne um professor melhor. Agradeço a Profa. Dra. Marta Bez, pela sua amizade e presteza. Obrigado àqueles que participaram direta e indiretamente na conclusão desta etapa de minha formação, em especial, agradeço aos professores Daniel Dalalana, Gabriel Simões, Luis André Werlang, Rodrigo Goulart e Sandra Miorelli.

Preciso também agradecer a todos os professores que tive, desde a minha infância até este momento. Sem dúvida, muitos destes mestres influenciaram a minha escolha em seguir esta carreira profissional, da qual tenho muito orgulho e creio ser das mais importantes de nossa sociedade.

DESCOBERTA DE CONJUNTOS DE ITENS FREQUENTES COM O MODELO DE PROGRAMAÇÂO MAPREDUCE, SOBRE CONTEXTOS DE INCERTEZA

RESUMO

Frequent Itemsets Mining (FIM) é uma tarefa de mineração de dados utilizada para encontrar relações entre os itens de um *dataset*. O Apriori é um tradicional algoritmo da classe *Generate-and-Test* para descobrir tais relações. Estudos recentes mostram que este e outros algoritmos desta tarefa não estão aptos para executar em contextos onde haja incerteza associada, pois eles não estão preparados para lidar com as probabilidades existentes nos itens do *dataset*. A incerteza nos dados ocorre em diversas aplicações como, por exemplo, dados coletados de sensores, informações sobre a presença de objetos em imagens de satélite e dados provenientes da aplicação de métodos estatísticos. Dada a grande quantidade de dados com incertezas associadas, novos algoritmos têm sido desenvolvidos para trabalharem neste contexto: UApriori, UF-Growth e UH-Mine. O UApriori, em especial, é um algoritmo baseado em suporte esperado, abordado frequentemente pela comunidade acadêmica. Quando este algoritmo é aplicado sobre grandes *datasets*, em um contexto com probabilidades associadas aos itens do *dataset*, ele não apresenta boa escalabilidade. Por outro lado, alguns trabalhos têm adaptado o algoritmo Apriori para trabalhar com o modelo de programação MapReduce, a fim de prover uma melhor escalabilidade. Utilizando este modelo, é possível descobrir itens frequentes de modo paralelo e distribuído. No entanto, tais trabalhos focam seus esforços na descoberta de itens frequentes sobre *datasets* determinísticos. Esta tese apresenta o desenvolvimento, implementação e os experimentos realizados, a partir da aplicação e discussão de três algoritmos: UAprioriMR, UAprioriMRByT e UAprioriMRJoin. Os três algoritmos citados evoluem o algoritmo tradicional Apriori para que possam executar com o modelo de programação MapReduce sobre contextos com incerteza associada. O algoritmo UAprioriMRJoin é um algoritmo híbrido com base nos algoritmos UAprioriMR e UAprioriMRByT. Os experimentos revelam o bom desempenho do algoritmo UAprioriMRJoin quando aplicado sobre grandes *datasets*, com muitos atributos e um número médio pequeno de itens por transação, em um *cluster* de nodos.

Palavras-chave: Mineração de itens frequentes; Padrões frequentes; Incerteza; Apriori; UApriori; MapReduce; Apache Hadoop.

DISCOVERING FREQUENT ITEMSETS WITH THE MAPREDUCE PROGRAMMING MODEL ON UNCERTAIN CONTEXT

ABSTRACT

Frequent Itemsets Mining (FIM) is a data mining task used to find relations between dataset items. Apriori is the traditional algorithm of the *Generate-and-Test* class to discover these relations. Recent studies show that this algorithm and others of this task are not adapted to execute in contexts with uncertainty because these algorithms are not prepared to handle with the probabilities associated to items of the dataset. Nowadays, data with uncertainty occur in many applications, for example, data collected from sensors, information about the presence of objects in satellite images and data from application of statistical methods. Due to big datasets with associated uncertainty, new algorithms have been developed to work in this context: UApriori, UF-Growth and UH-Mine. UApriori, specially, is an algorithm based in expected support, often addressed by scientific community. On the one hand, when this algorithm is applied to big datasets, in a context with associated probabilities to dataset items, it does not present good scalability. On the other hand, some works have evolved the Apriori algorithm joining with the model of programming MapReduce, in order to get a better scalability. With this model, it is possible to discover frequent itemsets using parallel and distributed computation. However, these works focus their efforts on discovering frequent itemsets on deterministic datasets. This thesis present the development, implementation and experiments applied to three algorithms: UAprioriMR, UAprioriMRByT and UAprioriMRJoin. The three cited algorithms evolve the traditional algorithm Apriori, integrating the model of programming MapReduce, on contexts with uncertainty. The algorithm UAprioriMRJoin is a hybrid algorithm based on the UAprioriMR and UAprioriMRByT algorithms. The experiments expose the good performance of the UAprioriMRJoin algorithm, when applied on big datasets, with many distinct items and a small average number of items per transaction in a cluster of nodes.

Keywords: Frequent Itemsets Mining; Frequent Patterns; Uncertain Data; Apriori; UApriori; MapReduce; Apache Hadoop.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

Amazon EC2	*Amazon Elastic Compute Cloud*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
DPC	*Dynamic Passes Combined-counting*
EMR	*Elastic Map Reduce*
FIM	*Frequent Itemsets Mining*
FIMI	*Frequent Itemset Mining Dataset Repository*
FPC	*Fixed Passes Combined-counting*
GFS	*Google File System*
GID	*Group Identification*
GPS	*Global Positioning System*
HDFS	*Hadoop Distributed Filesystem*
IDC	*International Data Corporation*
JVM	*Java Virtual Machine*
KDD	*Knowledge Discovery on Databases*
MPI	*Message Passing Interface*
OpenMP	*Open Multi-Processing*
NID	Número de Itens distintos no Dataset
RFID	*Radio-Frequency IDentification*
RPC	*Remote Procedure Call*
SGBDR	Sistemas Gerenciadores de Bancos de Dados Relacionais
SPC	*Single Pass Counting*
TMT	Tamanho Médio de uma Transação
UCI	*University of California Irvine*

SUMÁRIO

1. INTRODUÇÃO

Uma das tarefas descritivas mais conhecidas e utilizadas da área de mineração de dados é a Extração de Regras de Associação (*Association Rules Mining*) [AGR93], também denominada por Han et al. [HAN11] de Mineração de Padrões Frequentes. Esta tarefa é responsável por descobrir relacionamentos interessantes escondidos em grandes conjuntos de dados. Ela consiste basicamente em varrer uma grande quantidade de dados e descobrir correlações entre os itens existentes nos dados. Segundo Agrawal et al. [AGR93], esta tarefa varre uma grande coleção de transações com o objetivo de encontrar regras de associação entre conjuntos de itens (atributos).

Para Tan et al. [TAN09], esta tarefa é denominada Análise de Associação e é composta por duas etapas: *Frequent Itemsets Mining* (FIM) e Extração das Regras de Associação. FIM é a primeira etapa dos algoritmos de Análise de Associação e também o gargalo desta tarefa de mineração de dados. Esta etapa tem como característica o grande consumo de memória e de tempo de processamento, principalmente quando os *datasets* envolvidos são grandes e os limiares utilizados nos algoritmos têm valores baixos.

Diversos algoritmos de Análise de Associação têm sido propostos a fim de minimizar os custos de memória e de processamento da etapa FIM. Ao longo das décadas estes algoritmos concentraram esforços sobre *datasets* determinísticos. No entanto, nos últimos anos observa-se um crescimento dos trabalhos que desenvolvem algoritmos de Análise de Associação para trabalhar em contextos onde existam incertezas associadas [CAR13].

1.1 Motivação

Nos dias atuais muitos *datasets* têm armazenado incertezas relacionadas aos dados. *Datasets* estes que guardam dados de sensores, informações sobre a presença de objetos em imagens de satélite, dados provenientes da aplicação de métodos estatísticos, etc. Os algoritmos tradicionais de FIM não são adequados para lidar com a incerteza em um *dataset*. A fim de permitir a geração de conjuntos de itens frequentes em *datasets* com incerteza, novos algoritmos têm sido propostos.

De acordo com Liu [LIU12], os algoritmos para minerar itens frequentes de dados incertos podem ser classificados como *Apriori-based*, *FP-growth-based* e *H-mine-based*. Já para Aggarwal [AGG09a], os algoritmos dessa área estão subdivididos somente em duas classes: *Generate-and-Test* e *Pattern Growth*.

Observando-se as duas classificações pode-se dizer que os algoritmos da classe *Generate-and-Test* são aqueles baseados no algoritmo Apriori [AGR94], ou seja, *Apriori-based*. Estes algoritmos caracterizam-se por criar, a cada iteração, candidatos que serão avaliados se devem participar do conjunto de itens frequentes ou não. Por outro lado, a classe *Pattern Growth* é constituída das subclasses *FP-growth-based* e *H-mine-based*. Estas duas subclasses, ainda segundo Aggarwal [AGG09a], têm como principal diferença a estrutura utilizada para representação dos dados. Algoritmos *FP-growth-based* adotam uma estrutura de representação baseada em árvore [HAN00], enquanto algoritmos *H-mine-based* utilizam uma estrutura baseada em um *array* denominado *hyper-linked* [PEI01].

O primeiro algoritmo desenvolvido da tarefa de Análise de Associação, com o objetivo de lidar com incerteza, foi o UApriori, proposto por Chui et al. [CHU07]. Neste algoritmo os autores redefinem o suporte de um itemset, denominando-o de *suporte esperado* e o aplicam em um contexto de incerteza. Este algoritmo é da classe *Generate-and-Test*, de acordo com a classificação de Aggarwal [AGG09a] e da classe *Apriori-based*, de acordo com Liu [LIU12].

A partir da concepção de suporte esperado e do UApriori, outros algoritmos foram construídos

nos últimos anos [CHU07], [CHU08], [GAO11] [AGG09a], [BHA11]. Estes algoritmos evoluem o UApriori procurando melhorar o seu desempenho em relação ao consumo de memória e tempo de processamento, já que o desempenho do algoritmo UApriori é prejudicado quando, além de executar sobre *datasets* com baixas probabilidades associadas aos dados, ele é executado sobre grandes *datasets*.

Atualmente, tecnologias tais como as redes sociais, diversos instrumentos que registram logs, dispositivos móveis e a disseminação de vídeos e imagens na internet, contribuem para o armazenamento de uma enorme quantidade de dados, dos mais diferentes formatos. Desta forma, grandes *datasets* tornam-se alvo da aplicação da tarefa de Análise de Associação e, portanto, alternativas são necessárias para propiciar uma melhor escalabilidade de algoritmos como o UApriori, quando executados sobre essa grande massa de dados.

Dean e Sanjay publicaram em 2008 [DEA08] o modelo de programação MapReduce. Este modelo trabalha de forma paralela sobre grandes *datasets*, em um ambiente com nodos distribuídos que formam *clusters*. O *dataset* é dividido em partes, onde cada pedaço é direcionado a um nodo do *cluster*. O programador, por sua vez, precisa implementar duas funções, denominadas de *Map* e *Reduce*, que são distribuídas aos nodos do *cluster* para que executem sobre uma porção do *dataset*.

Alguns trabalhos têm surgido, nos últimos anos, utilizando o modelo de programação MapReduce para desenvolver algoritmos de FIM sobre dados determinísticos [LI12], [LIN12b]. Estes trabalhos transformam os algoritmos tradicionais em funções *Map* e *Reduce*, para que possam tirar proveito de um ambiente distribuído e paralelizar sua execução. Os mesmos relatam bons resultados em relação à escalabilidade, ao utilizar a implementação do modelo de programação *MapReduce* denominada *Apache Hadoop* [APA14].

No entanto, poucos trabalhos têm dado enfoque na utilização de MapReduce e FIM sobre contextos com incerteza associada. Apenas o artigo de Leung e Hayduk [LEU13] aborda a integração destes três temas. Neste trabalho, foi desenvolvido o algoritmo MR-growth, que está baseado no algoritmo UF-Growth da classe *FP-growth-based*. O MR-growth descobre os padrões frequentes gerando árvores (UF-trees) que são processadas nos nodos do *cluster*. São evidenciados melhores desempenhos do MR-growth, quando executado sobre um *cluster*, do que o seu correspondente algoritmo sequencial: UF-growth.

Alguns trabalhos mostram que algoritmos da classe *FP-growth-based* são mais eficientes do que os algoritmos da classe *Apriori-based* quando aplicados sobre *datasets* determinísticos. No entanto, quando aplicados sobre *datasets* com incerteza associada, este comportamento se altera. O trabalho de Tong et al. [TON12] demonstra que algoritmos da classe *Apriori-based*, tais como o UApriori, têm desempenho superior aos algoritmos da classe *FP-growth-based*, tais como o UF-Growth, quando utilizados *datasets* com incerteza associada.

Esta tese constrói o algoritmo UApriori [CHU07], integrando-o com o modelo de programação MapReduce, baseando-se no trabalho de Lin et al. [LIN12b], mas evoluindo-o para trabalhar em contextos de incerteza e denominando-o de UAprioriMR. Com base nesta motivação e na implementação do algoritmo UAprioriMR, elabora-se a questão de pesquisa desta tese: **é possível desenvolver um algoritmo, utilizando o modelo de programação MapReduce, que apresente uma média de tempo de execução inferior à média de tempo de execução do algoritmo UAprioriMR, sobre *datasets* com incerteza?**

Para responder esta questão de pesquisa foi desenvolvido e implementado o algoritmo híbrido UAprioriMRJoin. Posteriormente, foram realizados experimentos com os algoritmos UAprioriMR e UAprioriMRJoin em um laboratório experimental. Assim, foi possível variar o número de nodos no *cluster*, os *datasets* envolvidos e os mínimos suportes esperados. Foram elaboradas as hipóteses nula e alternativa para, a partir dos resultados dos experimentos e de testes estatísticos de *Wilcoxon*, avaliar se houve diferenças estatisticamente relevantes das médias dos tempos de execu-

ção dos algoritmos UAprioriMR e UAprioriMRJoin, permitindo, desta forma, responder a questão de pesquisa.

1.2 Objetivo

Assumindo este contexto e partindo desta questão de pesquisa, esta tese apresenta três algoritmos: *UAprioriMR, UAprioriMRByT e UAprioriMRJoin* para lidar com incerteza associada dentro dos *datasets*, a fim de descobrir conjuntos de itens frequentes. Estes três algoritmos são desenvolvidos e executados em um ambiente distribuído, de modo paralelo, utilizando o modelo de programação MapReduce e a implementação Apache Hadoop. O desenvolvimento e testes dos dois primeiros algoritmos servem de base para a implementação do algoritmo UAPrioriMRJoin.

O ambiente distribuído, sobre o qual foram realizados os experimentos, demonstra bons resultados dos dois algoritmos: UAprioriMR e UAprioriMRByT, à medida que o *cluster* aumenta o número de nodos. De acordo com as características dos *datasets*, em determinados passos do algoritmo, ora o algoritmo UAprioriMR tem melhor desempenho, ora o algoritmo UAprioriMRByT é melhor. Desta forma, o algoritmo UAprioriMRJoin é implementado para tirar proveito de situações onde cada um dos outros dois algoritmos tem seu melhor desempenho. Assim, o desempenho do algoritmo UAprioririMRJoin mostra-se superior aos algoritmos UAprioriMR e UAprioriMRByT nos *datasets* testados.

O principal objetivo desta tese é desenvolver o algoritmo híbrido *UAprioriMRJoin*, que ora cria itemsets candidatos baseados no conjunto frequente de passo anterior, ora cria itemsets candidatos por transação, utilizando o modelo de programação MapReduce, evidenciando uma média de tempo de execução mais baixa e uma métrica de *speedup* superior deste algoritmo em relação ao algoritmo UAprioriMR, quando utilizado *datasets* com incerteza.

1.3 Organização

Esta tese está organizada em sete capítulos, além desta introdução. O segundo capítulo aborda o Modelo de Programação MapReduce, que permite a execução de algoritmos de modo paralelo e distribuído. O capítulo três faz uma revisão de literatura sobre a tarefa de descoberta de conjuntos de itens frequentes e suas características ao lidar com incerteza nos dados. O quinto capítulo detalha os algoritmos desenvolvidos para descobrir conjuntos de itens frequentes sobre *datasets* com incerteza associada, usando o modelo de programação *MapReduce*. No capítulo seis são apresentados os experimentos realizados aplicando-se os algoritmos implementados. Por fim, o último capítulo é dedicado às conclusões, considerações finais e perspectivas de trabalhos futuros.

2. MODELO DE PROGRAMAÇÃO MAPREDUCE

Big data é o nome dado para conceituar a grande quantidade de dados dos dias atuais, que cresce de modo muito rápido [KOL13]. Para Edd Dumbill, em [ORE12], *big data* são os dados que excedem a capacidade de processamento dos sistemas de banco de dados convencionais, pois são muitos grandes, crescem rapidamente ou ainda não se adaptam às arquiteturas destes sistemas de banco de dados.

Este volume e transferência de dados têm sido possível por causa de uma série de fatores: o barateamento do hardware responsável por armazenar dados, o aumento da largura de banda das redes, processadores mais velozes e a disseminação massiva de aplicativos para dispositivos como *smartphones* e *tablets*. *Big data* é produzido por redes sociais, logs de servidores web, fluxo de dados de sensores, imagens de satélites, *streaming* de áudio e vídeo, transações bancárias, documentos governamentais, rotas de GPS (*Global Positioning System*), leitores de RFID (*Radio-Frequency IDentification*), mercado financeiro, dentre outros. A Figura 2.1 exibe algumas destas fontes de dados.

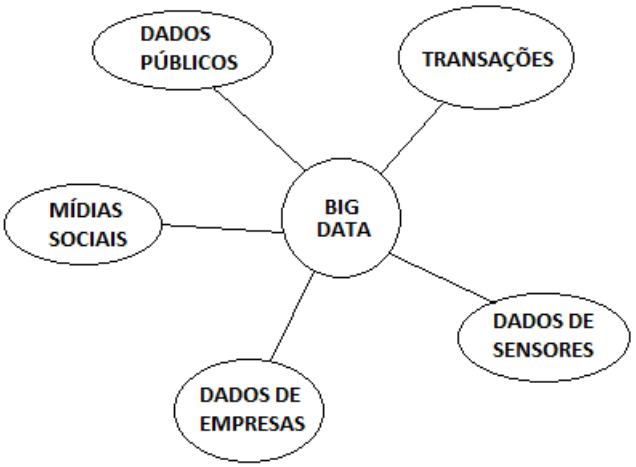

Figura 2.1: Algumas fontes de dados responsáveis por *big data*.

Gantz e Reinsel [GAN11], por meio de uma pesquisa do IDC (*International Data Corporation*), estimam o tamanho do "Universo Digital": "... a informação mundial dobra a cada 2 anos e até 2020 o mundo gerará 50 vezes a quantidade de informação que havia em 2011". Tom White [WHI12] comenta sobre a grande massa de dados gerada pela humanidade. Ele citou, em 2012, o site de genealogia Ancestry.com que armazenava ao redor de 2.5 petabytes ($2.5 \cdot 10^{15}$) de dados e o *facebook* que guardava aproximadamente cerca de 10 bilhões de fotos, ocupando 1 petabyte de disco. A IBM estima que 2.5 quintilhões ($2.5 \cdot 10^{18}$) de bytes são criados a cada dia [SIE13].

Big data caracteriza-se por 3Vs: Volume, Variedade e Velocidade [ORE12]. O Volume diz respeito à quantidade de dados. A Variedade denota que o termo *big data* não está associado somente aos dados estruturados, pois também inclui àqueles não estruturados, tais como figuras, vídeos, textos e áudios. A Velocidade está relacionada à rapidez com que os dados mudam. Para Sam Siewert [SIE13], há ainda um quarto V: a Veracidade. Esta característica está relacionada à incerteza dos dados, pois com este Volume, Variedade e Velocidade dos dados, para que seja realizada uma tomada de decisão assertiva, é crucial saber em qual instante de tempo determinada informação é verdadeira.

Uma organização pode necessitar e processar este volume de dados para uso analítico. Para Edd

Dumbill [ORE12] analisar *big data* pode revelar conhecimento oculto existente nos dados tais como influências sobre clientes, a partir de análises de transações em cestas de compras, que mantenham dados sociais e geográficos dos mesmos. Gautam Shroff [SHR13] conceitua o processo de descoberta de fatos e conhecimentos aprendidos com *big data* de Inteligência Preditiva ou *Big Data Analytics*.

Gautam Shroff [SHR13] cita diversas aplicações que unem *big data* e inteligência artificial: prever as intenções dos clientes quando navegam na web; aferir o sentimento do consumidor, a partir de suas conversas no *twitter* e *facebook*; predizer o comportamento dos consumidores; usar *tweets* para identificar eventos como enchentes, incêndios, acidentes, desastres, e rapidamente prever seu impacto e reagir; compartilhar o genoma para compreender os ancestrais humanos e a potencial probabilidade de contrair determinadas doenças genéticas, além de aplicar medicamentos mais adequados aos perfis genéticos descobertos; consumir e distribuir de modo mais inteligente a energia, água e outros recursos, usando sensores inteligentes e análises aprofundadas sobre o grande volume de dados coletados.

Para Sam Madden [MAD12] diversas ferramentas suportam análises sofisticadas de dados (SAS, R, Matlab), mas elas não estão preparadas para lidar com *datasets* no contexto de *big data*. Tom White [WHI12] afirma ainda que Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDR) apresentam dificuldade para gerenciar, consultar, analisar e visualizar esta escala massiva de dados. Estes sistemas, normalmente por meio da estrutura de dados *B-Tree*, têm bom desempenho quando trabalham sobre porções de registros na ordem de terabytes. No entanto, ao lidar com um grande volume de registros (na ordem de petabytes), esta estrutura torna-se menos eficiente. Para auxiliar *Big Data Analytics*, outras tecnologias têm sido propostas, tais como o modelo de programação MapReduce.

2.1 MapReduce

Processar uma grande quantidade de documentos, logs de requisições web, grandes volumes de imagens, dentre outros *datasets*, têm sido uma necessidade dos dias atuais. Uma das maneiras encontradas para lidar com este problema é distribuir o processamento através de centenas ou milhares de máquinas, a fim de obter um tempo de execução razoável. No entanto, paralelizar o processamento, distribuir os dados e solucionar falhas, são questões complexas a serem resolvidas, inerentes à programação distribuída. Para Dean e Sanjay [DEA08] o modelo de programação MapReduce provê o paralelismo automático e distribuição do processamento em larga escala, a partir do uso de grandes *clusters* de computadores, ou ainda, sobre múltiplos *cores* da mesma máquina.

De acordo com Tom White [WHI12] MapReduce é um modelo de programação para processamento de grandes *datasets*. Este modelo executa o processamento de um grande conjunto de dados, utilizando algoritmos paralelos e distribuídos, sobre um *cluster* de nodos. O programador especifica a computação utilizando basicamente duas funções: Map e Reduce. A função Map é responsável por implementar a extração, filtro e ordenação dos dados, enquanto a função Reduce se responsabiliza por sintetizar os dados e exibi-los.

De acordo com Dean e Sanjay [DEA08] MapReduce foi desenvolvido originalmente pela Google e suas funções Map e Reduce inspiram-se na programação funcional. A semelhança está em dividir um problema dentro de um conjunto de funções que capturam um conjunto de entradas com pares chave/valor e produzem um conjunto de saídas, também com pares chave/valor.

De modo resumido pode-se afirmar que a função Map toma um par chave/valor e produz um conjunto de pares intermediários chave/valor. Todos os valores intermediários com a mesma chave são agrupados e enviados para a função Reduce. Esta última recebe cada chave e seu conjunto de valores, os combina para compor um conjunto ainda menor de valores e, tipicamente, produz como saída nenhum ou apenas um valor por chave. A Figura 2.2 resume o modelo de programação

MapReduce.

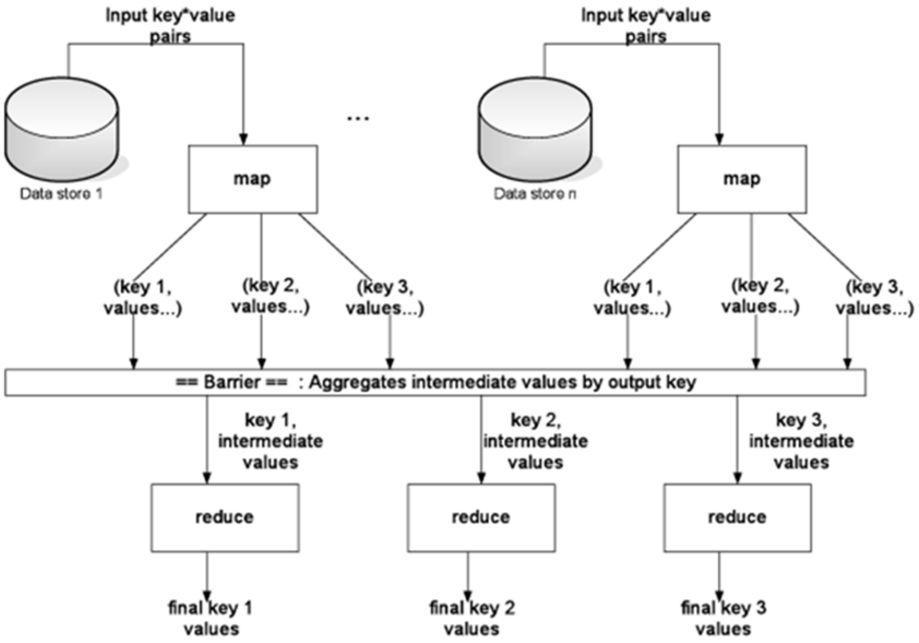

Figura 2.2: Fluxo de dados do modelo de programação MapReduce [MIL13].

Uma implementação do modelo de programação MapReduce muito utilizada atualmente é a Apache Hadoop [APA14]. Ela foi criada por Doug Cutting, criador do Apache Lucene, em meados de 2006, a partir das ideias do sistema de arquivos distribuídos da Google, o *Google File System* (GFS) e do artigo que difundiu a ideia de MapReduce, escrito por Dean e Sanjay [DEA08]. Esta implementação coordena e controla os nodos distribuídos. Ela executa as diversas tarefas em paralelo, gerenciando todas as comunicações e transferências de dados entre as várias partes do sistema, além de garantir a tolerância a falhas, balanceamento de carga e gerenciar o processo como um todo.

Portanto, a implementação Apache Hadoop permite o desenvolvimento das funções Map e Reduce, que podem assumir diversas formas, dependendo do trabalho a ser feito. Além disso, proporciona escalabilidade e tolerância a falhas para uma gama enorme de aplicações, otimizando suas execuções. A arquitetura da versão 1.x desta implementação é constituída basicamente por um sistema de arquivos distribuído, denominado HDFS (*Hadoop Distributed File System*) e pelo modelo de programação distribuído MapReduce. Ela conta também com diversos projetos associados, tais como Hive, Pig, Hbase, Sqoop, dentre outros [MIL13] [WHI12].

2.2 HDFS

Quando um *dataset* cresce além da capacidade de armazenamento em uma única máquina física, é preciso particioná-lo entre computadores diversos. A fim de gerenciar o armazenamento através da rede de máquinas, é necessário que haja um sistema de arquivos distribuídos. Além de efetuar este gerenciamento, ele terá também como desafio a coordenação das falhas dos nodos da rede, garantindo que os dados não sejam perdidos.

HDFS é o sistema de arquivos distribuídos da implementação Apache Hadoop. Ele trabalha bem com arquivos muito grandes, cujos tamanhos vão de centenas de megabytes até o armazenamento de petabytes de dados [WHI12]. O HDFS está baseado na ideia de escrita do *dataset* uma única

vez e na permissão de múltiplas leituras sobre ele. Este sistema de arquivos distribuídos não requer hardware de alto custo, pois foi projetado para executar sobre *commodity hardware* [WHI12].

HDFS trabalha, por padrão, com um bloco de dados de tamanho igual a 64MB. Logo, os arquivos no HDFS são quebrados em *splits* (*chunks*) que possuem o tamanho de um bloco e estes são armazenados como unidades independentes. Esta abstração do sistema de arquivos distribuídos em blocos permite: a manipulação de arquivos maiores do que qualquer disco na rede; que os blocos de um arquivo fiquem em discos distintos; o gerenciamento de quantos blocos podem ser armazenados em um determinado disco; a facilidade de replicar os dados, garantindo a tolerância a falhas e disponibilidade. A fim de garantir que blocos corrompidos sejam recuperados por causa de falhas em discos e/ou máquinas, cada bloco é replicado para um pequeno número de nodos distintos (3 por padrão).

Um *cluster* HDFS tem dois tipos de nodos sendo executados em um padrão *master-worker*. O nodo *master* é denominado de *namenode* e há um determinado número de *workers*, chamados de *datanodes*. O *namenode* é responsável por manter a árvore do sistema de arquivos e guardar os metadados dos arquivos e diretórios desta árvore. Tais informações são persistidas no disco local. O *namenode* também está encarregado de saber onde estão os blocos de cada arquivo nos *datanodes*. Os *datanodes*, no HDFS, mantêm e recuperam os blocos quando estes são solicitados. Além disso, reportam ao *namenode*, periodicamente, uma lista dos blocos que eles armazenam.

A Figura 2.3 ilustra o padrão *master-worker*, usado pelo HDFS. Nela notam-se diversas características da arquitetura HDFS: a interação entre os clientes de leitura e escrita; a organização dos *datanodes* em *racks*; os blocos armazenados em seus respectivos *datanodes*; o processo de replicação entre *datanodes* distintos, inclusive entre *racks* diferentes; e a importância dos metadados para que o *namenode* saiba em quais *datanodes* procurar os blocos requisitados. A Figura 2.3 também deixa claro que os nodos não compartilham memória, nem disco.

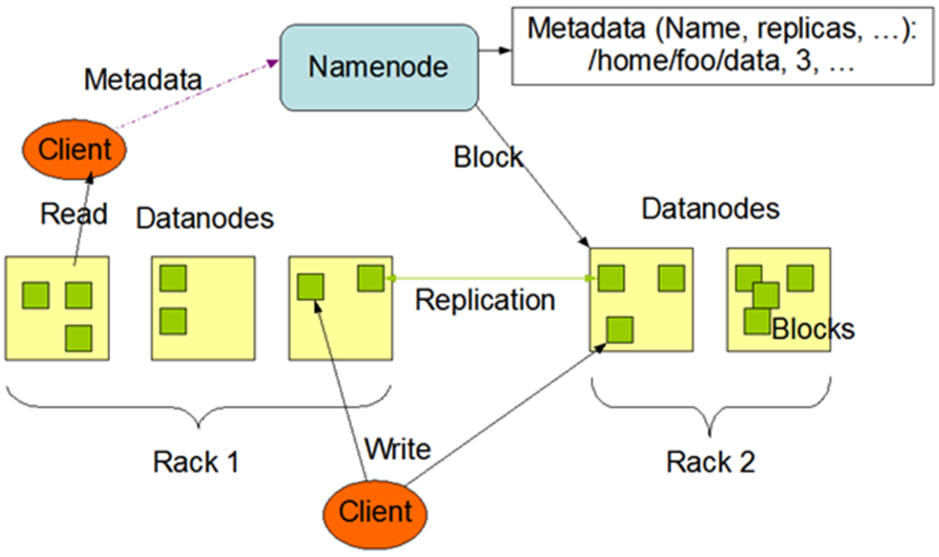

Figura 2.3: Arquitetura HDFS (adaptado de [APA14]).

As interações com o HDFS, leituras e escritas dos arquivos, podem ser realizadas através da API Java disponível. A fim de compreender o fluxo de dados e eventos que ocorrem no HDFS, a Figura 2.4 mostra como é realizada a leitura de um arquivo, solicitado por um programa cliente, desenvolvido em Java. O cliente inicialmente abre o arquivo que deseja ler, usando o método open() do objeto DistributedFileSystem (1). Este objeto chama o *namenode*, usando *Remote Procedure Call* (RPC), a fim de descobrir os locais onde estão os primeiros blocos do arquivo a ser lido (2).

Para cada bloco, o *namenode* retorna os endereços dos *datanodes* onde existem cópias dele.

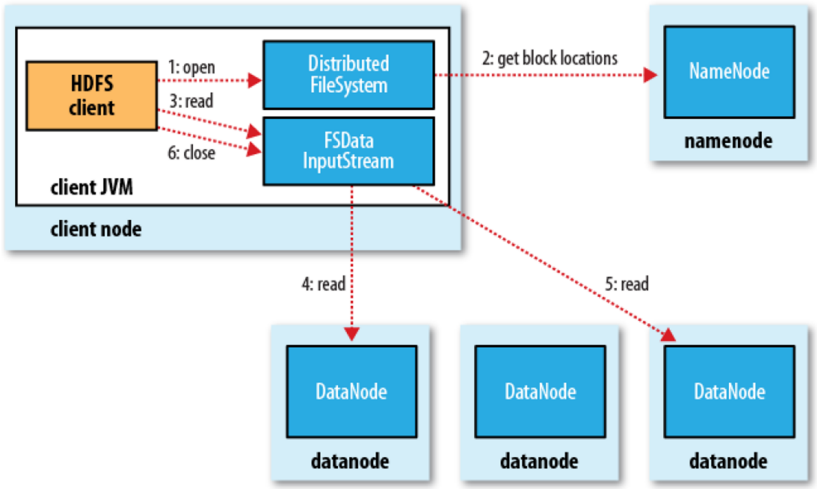

Figura 2.4: Cliente executando a tarefa de leitura dos dados sobre HDFS e usando a API Java [WHI12].

Posteriormente o objeto DistributedFileSystem retorna um objeto FSDataInputStream para o cliente. Este objeto é responsável por gerenciar a entrada/saída (I/O) entre o *namenode* e *datanodes*. O cliente então chama o método read() sobre o objeto FSDataInputStream, que conecta-se ao *datanode* mais próximo (3), a fim de ler os primeiros blocos do arquivo. Os dados então são transferidos repetidamente do *datanode* até o cliente (4). Quando o final do bloco é atingido, o objeto FSDataInputStream fecha a conexão entre o cliente e o *datanode*, além de buscar o *datanode* para o próximo bloco (5). Os blocos são lidos com a abertura de novas conexões entre *datanodes* e o cliente. O *namenode* é acessado sempre que necessário recuperar o próximo lote de blocos. Quando o cliente finalizar a leitura, ele chama o método close() sobre o objeto FSDataInputStream e o objeto DistributedFileSystem finaliza a operação (7), informando ao *namenode*.

2.3 Usando MapReduce sobre o HDFS

A implementação Apache Hadoop faz a coordenação dos trabalhos (*jobs*) executados sobre as máquinas do *cluster*, de modo transparente ao usuário, a partir do modelo de programação MapReduce. Nesta implementação, o modelo MapReduce consiste de um único *job* denominado *Master JobTracker*, onde cada nodo do *cluster* tem um *job* escravo chamado *TaskTracker*. O *Master JobTracker* é responsável por coordenar as tarefas dos escravos *TaskTrackers*, monitorando-as e reexecutando aquelas que apresentarem falhas. Desta forma, como cada *datanode* no *cluster* tem um *TaskTracker*, normalmente existem múltiplos processos *TaskTrackers*, dependendo do número de nodos no *cluster*.

O *dataset* de entrada é salvo sobre o HDFS. Logo em seguida, o *namenode* particiona o *dataset* de entrada em *splits*, que são distribuídos aos *datanodes*, antes de executar a função Map. Esta função lê uma partição do arquivo de entrada (um ou mais blocos de um arquivo) e cria uma série de pares chaves/valores. A função Map não requer ordenação nos dados de entrada. Ela processa cada par, gerando zero ou mais pares de saídas chaves/valores, os quais são escritos em um arquivo (conjunto de dados temporário). A função Reduce é executada, capturando as respostas geradas pela função Map e combinando-as, de forma a solucionar o problema original.

2.3.1 Exemplo de Aplicação usando MapReduce: Contando Palavras

Dado um documento qualquer, deseja-se contar a quantidade de ocorrências das palavras existentes nele. O modelo de programação MapReduce divide o documento em diversos blocos de arquivo, nos diferentes *datanodes* presentes. A função Map executa sobre cada *datanode*, que lê as linhas de texto dos blocos do documento. Ela varre as linhas procurando pelas palavras, tais como tokens separados por um espaço em branco ou outra pontuação. Para cada palavra encontrada a função gera um par chave/valor. Neste caso a chave é a palavra e o valor é o número de ocorrências da palavra, por exemplo, 1.

Depois que a função Map processa toda a entrada, a função Reduce atuará. A entrada na função Reduce é a saída de cada função Map, com a união dos dados, de modo ordenado, baseada na sua chave. Desta forma, antes da aplicação da função Reduce, o modelo MapReduce particiona e agrupa os registros que tenham o mesmo valor de chave. Tais tarefas de Partition (partição) e Combine (agrupamento) são realizadas após o trabalho principal da função Map estar completo. A tarefa de partição é sempre desempenhada se existe mais do que um nodo para executar a tarefa Reduce. A função Combine é opcional e seu uso depende da natureza dos dados e o processamento a ser feito.

A função Reduce é executada uma única vez para cada chave única ordenada. Ela então itera através dos valores associados com aquela chave, e exibe zero ou mais valores. A função Reduce trabalha sobre um conjunto de pares chave/valor, independentemente de qualquer outro conjunto de pares chave/valor pois, assim, múltiplas funções Reduce podem ser processadas em paralelo. A saída de uma função Reduce é escrita para o HDFS e é portanto replicada, tipicamente sobre três *datanodes* distintos.

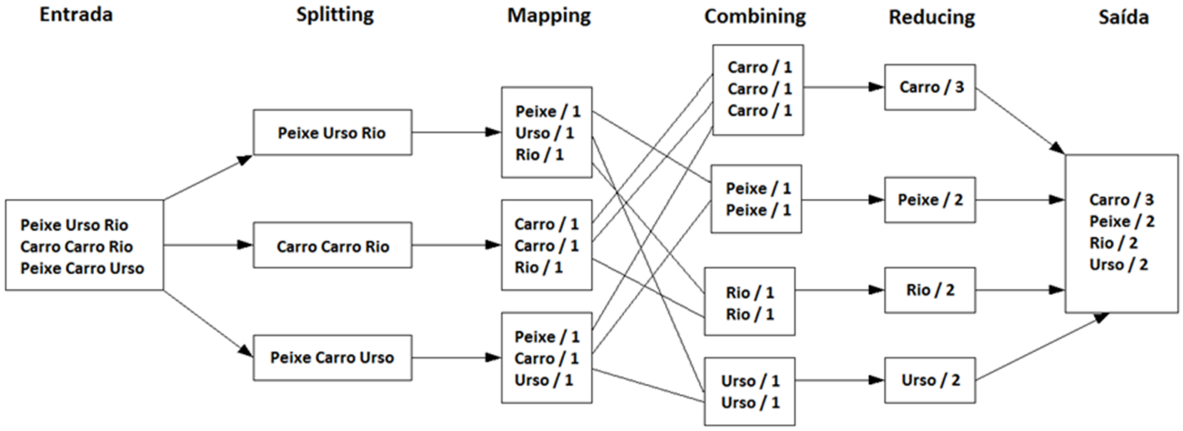

Figura 2.5: Uso dos processos Mapper, Combiner e Reducer para contar as palavras de um arquivo, utilizando MapReduce (baseado em [APA14]).

Observando-se o exemplo de contar palavras, os pares chave/valor, gerados pelas funções Map, foram ordenados sobre valores das chaves (as palavras) e todos os registros de mesmo valor chave foram enviados para a mesma função Reduce. Esta, por sua vez, itera através de todos os registros com os mesmos valores chave (mesma palavra) e calcula o número de ocorrências. A função Reduce então emite um outro registro que contém a palavra e o total do número de ocorrências daquela palavra.

A função Map, potencialmente, tem como saída um grande número de pares chave/valor. A fim de que a função Reduce possa ser executada, todos os pares chave/valor, para um particular valor de chave, devem ser enviados para o mesmo *datanode*. Utiliza-se a função Partition para que o sistema saiba para qual *datanode* uma particular chave/valor deve ser enviada. Esta função tem

uma chave e um número de *datanodes* onde executam as funções Reduce [WHI12].

Todos os registros emitidos através das funções Map são transferidos para os *datanodes* que processam a função Reduce. Logo, como o Apache Hadoop processa tipicamente trabalhos com grande quantidade de dados, haverá muito tráfego de rede entre os nodos no *cluster*. Isto tem um efeito negativo sobre o desempenho do *job*. O tempo de processamento de um *job* dentro de um *cluster* depende do acesso ao disco sobre um *datanode* (ler os *splits*) e da transmissão dos dados entre os *datanodes*. A fim de reduzir o tráfego de rede do *job*, é possível utilizar a função Combine, antes que a função Reduce seja executada. A Figura 2.5 ilustra os passos necessários para a contagem de palavras, utilizando a ideia da função Combine no modelo de programação MapReduce.

No exemplo de contagem de palavras, cada *datanode* Map processa uma grande quantidade de dados. A probabilidade é grande de uma particular função Map varrer a mesma palavra diversas vezes. Portanto, transferir muitos registros (chave/valor) da mesma palavra para um *datanode* Reducer, sendo que cada um deles está com o valor 1, não é a melhor estratégia. Utilizando-se a função Combine é possível reduzir o tráfego de rede, pois, ao invés de enviar 10 mil registros da palavra "mas", cada um deles com o valor 1, pode-se transmitir um único registro da palavra "mas" com valor igual a 10000. A Figura 2.6 exibe o fluxo de dados entre as funções Map (processos Mapper), Partition (processos Partitioner), Combine (processos que fazem Agrupamento) e Reduce (processos Reducer).

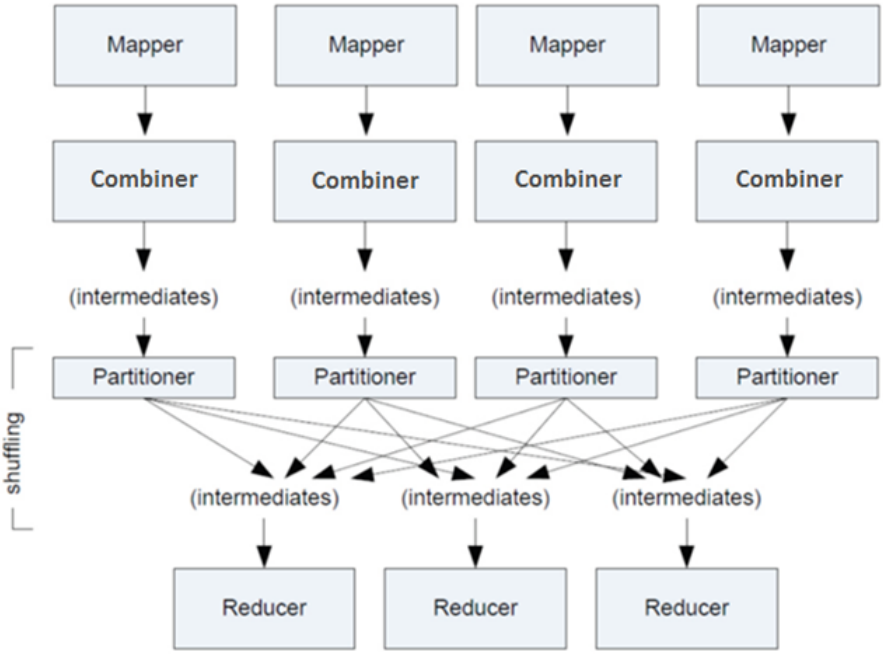

Figura 2.6: Fluxo de dados entre os processos Mapper, Combiner, Partitioner e Reducer [MIL13].

Para implementação das funções Map, Partition, Combine e Reduce, as aplicações MapReduce, usando a implementação Apache Hadoop, normalmente utilizam a API Java. No entanto, outras linguagens podem ser escolhidas, tais como Perl, Python ou Ruby. Além das linguagens previamente citadas, Hadoop Pipes é outra alternativa de implementação, pois é uma interface que permite utilizar funções na linguagem C++.

Uma grande variedade de companhias e organizações têm usado Apache Hadoop para seus ambientes tanto de pesquisa, quanto de produção [APA14]. Este software pode ser acessado e instalado de diversas formas. É possível baixá-lo em [APA14], instalá-lo e configurá-lo nos nodos. Pode-se ainda baixá-lo como máquina virtual, disponibilizado por grandes companhias como IBM

e Cloudera, já instalado e pré-configurado. Ou ainda, é possível usá-lo como serviço, de algum ambiente de Cloud Computing, como o serviço *Elastic Map Reduce* oferecido pela empresa Amazon.

2.3.2 Serviço Amazon Elastic MapReduce

O Amazon Web Services (AWS) é um conjunto de serviços de aplicativos e infraestrutura na nuvem, fornecida pela empresa Amazon. Ela oferece aos clientes a possibilidade de substituir os gastos e configuração de uma grande infraestrutura. Dentre os serviços do AWS pode-se destacar o AMAZON EC2, que permite executar servidores virtuais nas nuvens, o Storage S3, que armazena e recupera grandes quantidades de dados e o Elastic Map Reduce.

O serviço Elastic Map Reduce (EMR) permite analisar e processar grandes *datasets*. Ele distribui o processamento em *clusters* de servidores virtuais que rodam na nuvem da empresa Amazon. O *cluster* do EMR é gerenciado pela implementação de código aberto Apache Hadoop. De acordo com [AMA13], o EMR é um serviço que aprimorou o Apache Hadoop e outros projetos open-source para trabalhar no AWS. *Clusters* Apache Hadoop que rodam sobre o Amazon EMR usam instâncias do serviço Amazon EC2, executando máquinas virtuais para o *namenode* e outras máquinas virtuais para os *datanodes*. O serviço Amazon S3 também é usado para armazenar os dados de entrada e saída, além de oferecer outros serviços como o Amazon CloudWatch para monitorar o desempenho do *cluster*.

Desta forma, ao utilizar o Apache Hadoop em um ambiente como o Amazon Elastic MapReduce alguns benefícios de um ambiente Cloud Computing podem ser aproveitados. Facilmente pode-se regular o número de servidores virtuais necessários para processar os *jobs* (escalabilidade). Além disto, o serviço é cobrado de acordo com a quantidade e o tempo em que os servidores virtuais são utilizados. E por fim, é possível usufruir da integração de todos os serviços disponíveis no AWS.

2.4 Considerações Finais do Capítulo

Este capítulo discutiu o modelo de programação MapReduce, proposto originalmente por [DEA08]. O capítulo inicialmente apresenta o conceito de *big data* e evidencia a necessidade de novas tecnologias para lidar com grande quantidade de dados. Em sua sequência apresenta o modelo de programação MapReduce, explanando a respeito do seu sistema de arquivos distribuído (HDFS) e como as funções Map, Combine e Reduce estão inseridas no contexto deste modelo. O exemplo apresentado detalha explicitamente com os processos se comunicam no *cluster* de nodos. Logo, o entendimento destes tópicos são de extrema importância para a posterior compreensão de como o modelo de programação MapReduce é integrado aos algoritmos de Descoberta de Itens Frequentes.

3. DESCOBERTA DE ITENS FREQUENTES SOBRE DADOS COM INCERTEZA

Este capítulo trata da Descoberta de Itens Frequentes (*Frequent Itemsets Mining* - FIM) em contextos de incerteza, tema central desta tese. Ele serve de referencial bibliográfico para os diversos conceitos e temas que são apresentados neste e nos demais capítulos. O capítulo está dividido em três seções. Inicialmente a etapa de Descoberta de Itens Frequentes é discutida, inserindo-a no contexto da técnica de Análise de Associação. Na segunda seção o conceito de Incerteza dos Dados é definido e consequentemente associado, na última seção, à etapa de Descoberta de Itens Frequentes.

3.1 Descoberta de Itens Frequentes

Para Tan, Steinbach e Kumar [TAN09], *Knowledge Discovery on Database* (KDD) é todo o processo de transformação de dados puros em informação valiosa. E, de acordo com Fayyad [FAY86], a Mineração de Dados (*Data Mining*) é uma etapa de todo o processo de KDD. Esta etapa refere-se à aplicação de algoritmos para a extração de padrões e descoberta de conhecimento útil em dados, enquanto KDD engloba ainda as etapas de pré-processamento e pós-processamento, conforme Figura 3.1.

Dentro do processo de KDD, a etapa de pré-processamento é caracterizada por algumas tarefas. Dentre elas destacam-se a escolha da origem dos dados que serão garimpados; a fusão ou desmembramento dos mesmos; a limpeza de dados repetidos; a escolha de estratégias para referenciar os dados que estão ausentes ou com ruídos e a transformação dos diversos formatos a fim de padronizá-los. Estas atividades exercidas tendem a ocupar, no que diz respeito a tempo e custo, uma grande parte de todo o processo de KDD.

Após a aplicação dos algoritmos de Mineração de Dados é essencial que seja aplicada a etapa de pós-processamento. Sua responsabilidade é a de garantir que somente resultados válidos e úteis sejam direcionados aos tomadores de decisão. Nesta etapa, estes usuários finais precisam ter contato com sistemas que permitam visualizações flexíveis e análise dos dados sob diversos ângulos. As etapas de pré e pós-processamento, inseridas no processo de KDD, ficam mais claras na Figura 3.2.

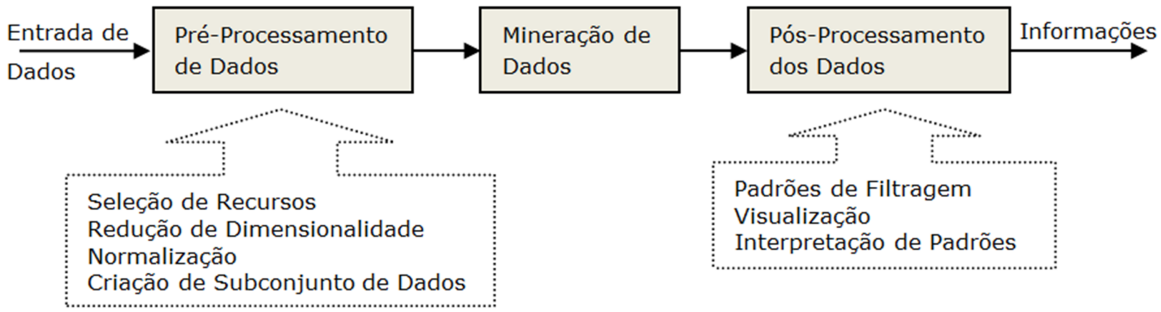

Figura 3.1: Visão do Processo de KDD, baseado em [TAN09].

De acordo com Han, Kamber e Pei [HAN11] muitas pessoas tratam Mineração de Dados como um sinônimo de KDD. No entanto, assim como [FAY86] e [TAN09], ele caracteriza Mineração de Dados como um passo do processo de KDD, o qual é responsável por descobrir conhecimento interessante em grandes conjuntos de dados, armazenados em bases de dados, *data warehouses* ou outros repositórios.

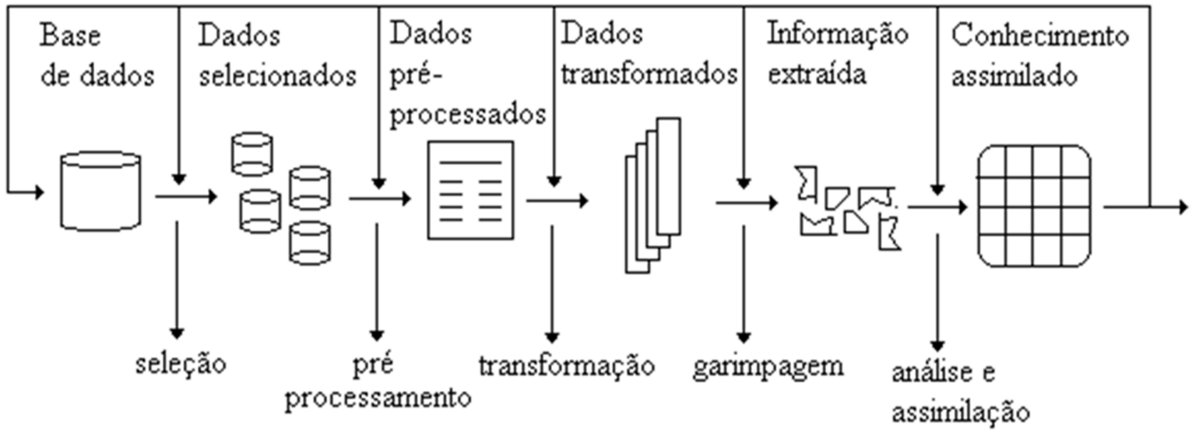

Figura 3.2: Visão do Processo de KDD, baseado em [FAY86].

Han, Kamber e Pei [HAN11] inserem o *data warehouse* como um elemento no processo de KDD. Eles descrevem como primeiro passo, o pré-processamento dos dados, tarefas de limpeza e integração. O resultado destas tarefas é armazenado em um *data warehouse*, e posteriormente os dados deste repositório são selecionados e transformados. Assim, as técnicas de mineração podem ser aplicadas e consequentemente ter seus valores avaliados e apresentados, em busca de conhecimento útil. O processo é interativo e iterativo, conforme exibido na Figura 3.3.

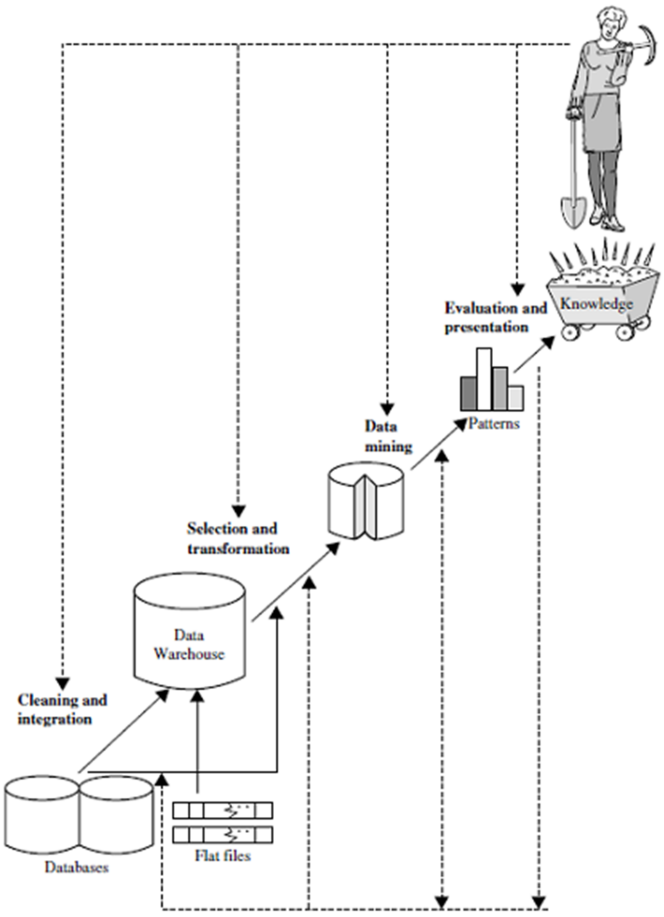

Figura 3.3: Visão do Processo de KDD, baseado em [HAN11].

3.1.1 Análise de Associação

Segundo Tan, Steinbach e Kumar [TAN09], as tarefas de mineração de dados são divididas em duas categorias principais: preditivas e descritivas. Na primeira o objetivo é, a partir de variáveis (atributos) independentes, prever o valor de uma determinada variável dependente (atributo alvo). Já a tarefa descritiva tem como objetivo descobrir padrões (correlações, tendências, grupos e anomalias) que estejam inseridos nos dados.

Uma das tarefas descritivas mais conhecidas e utilizadas da área de Mineração de Dados é a Extração de Regras de Associação [AGR93]. Esta tarefa consiste basicamente em varrer uma grande quantidade de dados já pré-processada e, através de fórmulas probabilísticas, descobrir correlações multidimensionais que associem itens do banco de dados. Segundo Agrawal, Imielinski e Swami [AGR93], esta tarefa garimpa uma grande coleção de transações com o objetivo de encontrar regras de associação entre conjuntos de itens, com um fator de confiança mínimo.

Para [TAN09] esta técnica é denominada "Análise de Associação" e é útil na descoberta de relacionamentos interessantes, escondidos em grandes conjuntos de dados. Um exemplo de relacionamento poderia ser expresso com a regra de associação: "Se computador e sistema operacional Windows então software antivírus [0.20, 0.92]". Esta regra indica que em 20% (suporte da regra) do total de compras realizadas, computadores, sistema operacional Windows e softwares antivírus, são adquiridos juntos. Além disso, 92% (confiança da regra) dos clientes que compraram computador e sistema operacional Windows, também compraram software antivírus.

3.1.2 Visão Geral de um Algoritmo de Análise de Associação

Dado um conjunto de transações $D = \{t_1, t_2, ..., t_n\}$, composto por um conjunto de itens $I = \{i_1, i_2, ..., i_m\}$, onde cada transação t_j específica é constituída por itens, ou seja, $t_j = \{i_1, i_2, ..., i_k\}$, ou ainda, $t_j \subseteq I$. Uma regra de associação R é uma implicação na forma $A => B$, onde $A \subset I$, $B \subset I$ e $A \cap B = \emptyset$. A ideia da Análise de Associação é aplicar algoritmos sobre grandes bases de dados (D), a fim de descobrir regras de associação significativas, em formatos semelhantes ao da Figura 3.4.

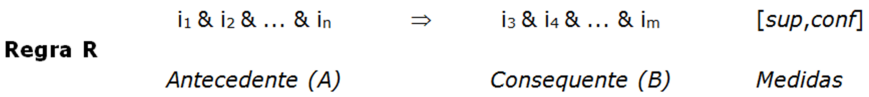

Figura 3.4: Especificação da forma de uma regra de associação.

Na regra R, exibida na Figura 3.4, *sup* é o suporte da regra, indicando a frequência de ocorrência dos itens desta regra dentro da base de dados D. O suporte de uma regra R é calculado dividindo-se o número de transações que contém os itens do antecedente e do consequente, pelo número de transações existentes na base de dados D. A Equação 3.1 ilustra o cálculo do suporte da regra R.

Já a confiança da regra, *conf*, denota a "força" da mesma, isto é, o grau de relacionamento do consequente em relação ao antecedente. Portanto, o cálculo da confiança da regra R é dado pela divisão do número de transações que contém o antecedente e o consequente, pelo número de transações nas quais somente o antecedente está presente. A Equação 3.2 descreve o cálculo desta medida.

$$sup = \frac{|(A \cup B|)}{|D|} \tag{3.1}$$

$$conf = \frac{|(A \cup B)|}{|(A)|} \tag{3.2}$$

De acordo com [TAN09], o cálculo do suporte e do grau de confiança, através de força bruta, para todas as regras de associação sobre uma base de dados, é algo extremamente custoso e inviável, pois o número de regras (|R|) para um conjunto com "m" itens seria dado pela Equação 3.3:

$$|R| = 3^m - 2^{m+1} + 1 \tag{3.3}$$

Em toda a aplicação de interesse são selecionadas somente as regras que possuem suporte e grau de confiança acima de mínimos estipulados, ou seja, acima de um limiar de importância em dado contexto. Logo, grande parte do esforço mencionado na geração de regras seria desperdiçado. Desta forma, algoritmos de regras de associação concentram-se em, inicialmente, gerar todos os conjuntos de itens com suporte maior ou igual a um suporte mínimo previamente especificado. Estes conjuntos são denominados de conjuntos de itens frequentes, ou itemsets frequentes e a união deles forma o grande conjunto de itens frequentes, representado por L. A partir de L, posteriormente, são recuperadas as regras com confiança acima de um limiar mínimo especificado.

3.1.3 Algoritmo Apriori

Descobrir o grande conjunto de conjuntos de itens frequentes (L) sobre uma base de dados grande e com suporte mínimo muito baixo requer elevado custo computacional. O algoritmo tradicional para descoberta do conjunto L é o Apriori, desenvolvido por Agrawal, Imielinski e Swami [AGR93]. Ele baseia-se na seguinte propriedade: se um conjunto de itens é frequente, então todos os seus subconjuntos também devem ser frequentes e, ao contrário, se um conjunto de itens é infrequente então todos os seus superconjuntos também o são [TAN09]. A codificação do Apriori está detalhada no Algoritmo 3.1.

Algoritmo 3.1: Algoritmo Apriori para Descoberta de Itens Frequentes

Entrada: D, *dataset* contendo as transações.
Entrada: $minsup$, mínimo suporte especificado.
Saída: grande conjunto L contendo todos os itemsets frequentes de tamanho k: $L = L_1 \cup L_2 \cup ... \cup L_k$.

1: $L_1 \leftarrow gera\_1 - itemsets()$
2: **for** $(k = 2; L_{k-1} != \emptyset; k++)$ **do**
3: $C_k = gera\_candidatos(L_{k-1})$
4: **for all** (transação $t_j \in D$) **do**
5: $C_t = gera\_subconjuntos(C_k, t_j)$
6: **for all** (candidatos $c \in C_t$) **do**
7: $c.count++$
8: **end for**
9: **end for**
10: $L_k = \{c \in C_k \mid c.count \geq minsup\}$
11: **end for**
12: return $L \cup = L_k$

O algoritmo Apriori faz k varreduras sobre o *dataset*, e a cada passo (varredura), ele gera todos os conjuntos de itens (itemsets) acima de um limiar probabilístico ($minsup$). Inicialmente os itemsets candidatos de tamanho 1 são gerados, conhecidos como 1-itemsets, a partir da primeira varredura no *dataset*. Logo em seguida, são considerados como frequentes apenas aqueles 1-itemsets com suporte maior ou igual ao $minsup$, sendo, portanto, gerado o conjunto L_1.

Após a descoberta do conjunto L_1 (1-itemsets frequentes), são gerados todos os conjuntos candidatos a 2-itemsets (C_2). Os elementos de C_2, com suporte maior ou igual ao $minsup$, farão parte do conjunto L_2. Ao final da k varredura sobre o *dataset* é gerado o grande conjunto L com a união de todos os conjuntos de itemsets frequentes gerados nos passos anteriores: $L = L_1 \cup L_2 \cup L3 \cup ... \cup L_k$.

O Algoritmo 3.1 tem duas funções: $gera\_candidatos(L_{k-1})$ e $gera\_subconjuntos(C_k, t_j)$. A primeira é responsável por criar todos os conjuntos candidatos de tamanho k, baseados no conjunto L_{k-1}, gerado no passo anterior. A função $gera\_subconjuntos(C_k, t_j)$ retorna somente àqueles conjuntos candidatos de tamanho k que estão presentes na transação t_j.

No pior caso esta abordagem poderá ter um conjunto L com $2^m - 1$ conjuntos de itens, que são todos os subconjuntos possíveis de I, onde m é o número de itens distintos em I. Além disso, a cada geração de C_k, existe a necessidade de varrer todo o *dataset* D. Por este motivo, para $k > 2$, o algoritmo Apriori aplica poda no conjunto de itens C_t, a partir da propriedade abaixo, conforme definida em [HAN11], a fim de diminuir o espaço de busca: *"Qualquer (k-1)-itemset que não é frequente não pode ser um subconjunto de um k-itemset frequente. Contudo, se qualquer (k-1)-subconjunto de um candidato k-itemset não está em L_{k-1}, então o candidato não pode ser frequente também e assim ele pode ser removido de C_k"*.

Exemplificando o Algoritmo Apriori

A fim de exemplificar o funcionamento do algoritmo Apriori, considere o *dataset* da Figura 3.5. Nesta figura estão ilustradas duas formas de representar um *dataset*. Na Figura 3.5 (a) são exibidas 10 transações, onde cada transação está associada a um conjunto de itens que foram comprados. Na Figura 3.5 (b) são exibidas as mesmas transações, no entanto, a representação é realizada de modo distinto. Neste caso, os itens adquiridos na transação são identificados pelo número 1 e os itens não comprados estão sinalizados com 0.

(a)

Transação	Itens
1	{Algodão, Esparadrapo}
2	{Algodão, Esparadrapo}
3	{Algodão, Esparadrapo, Mercúrio}
4	{Algodão, Esparadrapo, Mercúrio}
5	{Algodão, Esparadrapo, Mercúrio}
6	{Algodão, Esparadrapo, Mercúrio}
7	{ Esparadrapo, Mercúrio}
8	{Algodão, Esparadrapo, Mercúrio}
9	{Algodão, Mercúrio}
10	{Algodão, Esparadrapo, Mercúrio}

(b)

Transação	Algodão (A)	Esparadrapo (E)	Mercúrio (M)
1	1	1	0
2	1	1	0
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	0	1	1
8	1	1	1
9	1	0	1
10	1	1	1

Figura 3.5: Formas de representar o *dataset* que ilustra uma cesta de produtos: (a) *dataset* com os itens comprados descritos na transação; (b) *dataset* binário, onde 1 representa que o item foi adquirido na transação e 0 indica que o item não foi comprado.

A Figura 3.6 ilustra o funcionamento do algoritmo Apriori, exibindo todos os passos executados sobre o *dataset* da Figura 3.5. A partir de uma primeira varredura no *dataset*, o algoritmo Apriori descobre, no passo k=1, quais são os 1-itemsets frequentes, ou seja, aqueles que tem a métrica de suporte (frequência) maior ou igual a um suporte mínimo especificado. Os 1-itemsets frequentes formarão o conjunto L_1.

A partir do conjunto L_1, no passo k=2, são gerados todos os conjuntos candidatos a frequentes de tamanho 2, C_2 (1). É realizada uma nova varredura no *dataset* com o objetivo de calcular o suporte de cada um dos elementos de C_2 e verificar se o elemento tem suporte maior ou igual ao limiar mínimo de suporte e, desta forma, gerar o conjunto L_2 com todos os 2-itemsets frequentes

(2).

De posse do conjunto L_2, no passo k=3, são gerados os conjuntos candidatos a frequentes de tamanho 3, C_3 (3). Uma nova varredura no *dataset* é feita e para cada elemento de C_3 é calculado o seu suporte e, consequentemente, extraídos os elementos com suporte maior ou igual ao mínimo suporte, criando o conjunto L_3, com todos os 3-itemsets frequentes (4).

Neste exemplo, não há sentido em gerar os itemsets candidatos de tamanho 4 (C_4), pois só existem 3 elementos no *dataset* e o algoritmo para. Logo, o conjunto de itemsets frequentes (L) é representado pela união dos conjuntos frequentes de tamanho 1, 2 e 3: $L = L_1 \cup L_2 \cup L3$.

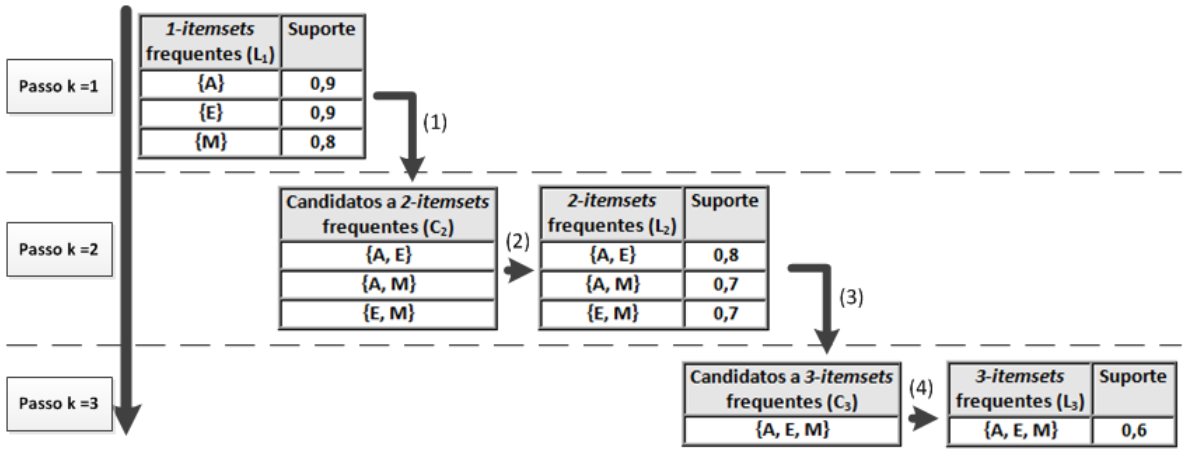

Figura 3.6: Funcionamento passo a passo do algoritmo Apriori.

Ao longo dos anos novos algoritmos baseados na ideia principal do Apriori têm sido criados. Muitos deles oferecem um melhor gerenciamento da memória principal e, consequentemente, um desempenho mais eficiente [AGR94] [SAV95], utilizando, por exemplo, estruturas de dados baseadas em *hash* ou árvores para armazenar os candidatos, ou ainda, procurando reduzir o número de varreduras realizadas no *dataset*. Outros algoritmos estenderam a ideia do Apriori, procurando extrair relacionamentos entre dados hierarquizados (múltiplos níveis) [HAN95], entre conjuntos de dados com atributos contínuos, e também com entidades mais complexas tais como sequências e grafos [HAN97].

3.2 Incerteza dos Dados

Aggarwal et al. [AGG09b] identificam mineração de dados como uma das áreas mais promissoras para pesquisa sobre incerteza nos dados. Os autores caracterizam a "descoberta de conjuntos de itens frequentes" como uma das técnicas com grande espaço para o desenvolvimento de pesquisas em um contexto de incerteza. Na revisão sistemática desenvolvida por Carvalho e Ruiz [CAR13], fica evidente o crescimento de publicações de artigos sobre o tema, a partir de 2007.

Os avanços tecnológicos em software, hardware e rede têm aumentado o volume das bases de dados. Muitos *datasets*, além de conter dados determinísticos, também armazenam dados com algum grau de incerteza associada. Tal incerteza pode ser gerada por uma série de motivos e são constantes em aplicações atuais.

A incerteza pode ser gerada em função dos métodos de coleta dos dados, tais como sensores. Durante o processo de captura, sensores podem gerar erros nos dados em função de sua imprecisão ao fazer a coleta, ou ainda durante a transmissão dos mesmos. Sensores que coletam dados específicos, tais como dados meteorológicos e de segurança, geram dados com incerteza. Frequentemente os dados dos sensores envolvidos na coleta necessitam ser consolidados. Desta consolidação é possível

estimar, por exemplo, a probabilidade "p" de os sensores detectarem um determinado objeto, com uma determinada margem de erro "e". Logo, tal incerteza é representada por probabilidades que são adquiridas, ou por meio de um especialista, ou através de algum método estatístico aplicado sobre os dados históricos.

A incerteza dos dados também pode estar relacionada ao desconhecimento da origem das fontes de informações, o que pode ocorrer por causa de formulários preenchidos incorretamente, leitura e escrita imprecisa de instrumentos diversos, problemas em sistemas de software, dentre outros fatores. Aplicações utilizadas para o reconhecimento de padrões, a fim de descobrir a presença ou ausência de objetos em imagens de satélite, são outros exemplos da existência de incerteza nos dados. Em função de erros e resoluções limitadas, a presença de um objeto em uma área no espaço é frequentemente incerta e probabilidades são utilizadas para representá-la.

Lakshmanan et al., em [LAK97], mostram um exemplo de *dataset* com dados incertos, a partir de uma base de dados contendo imagens, ilustrada na Tabela 3.1. Considerem-se algoritmos de processamento de imagens executando sobre imagens de vigilância. Dada uma imagem "im1.gif", tais algoritmos executam dois passos: tentam localizar faces na imagem "im1.gif" (segmentação) e posteriormente procuram combinar as faces encontradas no primeiro passo com imagens contidas em uma base de dados.

A tupla t_1, por exemplo, informa que uma face ocorre no arquivo de imagem "im1.gif". Esta face está localizada em um retângulo: canto rodapé-esquerda (5,10) e canto topo-direita (35,40). Para identificar quem está representado neste retângulo, de acordo com t_1, existem probabilidades associadas. A face pode ser de John, com 20% a 25% de certeza, ou Jim (35% - 40%) ou ainda Tom (40% - 45%).

Tabela 3.1: Exemplo de incerteza em um *dataset* de imagens (adaptado de [LAK97]).

tupla	**imagem**	**rodapé**	**esquerda**	**topo**	**direita**	**quem**	**IP**[1]	**SP**[2]
t_1	im1.gif	5	10	35	40	John Jim Tom	0.20 0.35 0.40	0.25 0.40 0.45
t_2	im2.gif	35	40	60	40	John Jim Ed	0.60 0.20 0.10	0.65 0.25 0.15
t_3	im3.gif	10	10	25	25	John Ed	0.30 0.60	0.35 0.65

Chui et al. [CHU08] ilustram outro exemplo de incerteza nos *datasets*. Tomem-se experimentos que testam microorganismos resistentes a certas drogas. Os resultados de cada teste são salvos em um *dataset*. Cada microorganismo é uma transação e as drogas são listadas como itens na transação. Aplicando algoritmos FIM sobre o *dataset*, é possível descobrir associações entre as drogas que apresentam resistência aos microorganismos. Na prática, devido aos possíveis erros de medição, diversos experimentos precisam ser conduzidos, a fim de obter a mais alta confiança nos resultados. Em cada caso, a existência de um item (droga) na transação deveria ser expressa como uma probabilidade. Por exemplo, se Streptococcus Pneumoniae (um microorganismo) mostra resistência a Penicilina (um antibiótico) 90 vezes nos 100 experimentos, a probabilidade de a propriedade "resistência à penicilina" existe em Streptococcus Pneumoniae em 90% dos casos. Este tipo de probabilidade é denominado de probabilidade existencial do item e foi caracterizada como tal, pela primeira vez, por Chui et al., em [CHU07].

Aggarwal et al. [AGG09a] afirmam e provam que algoritmos tradicionais como Apriori [AGR93], AprioriTid [AGR94] and FP-Growth [HAN00] não estão adaptados a lidar com as características

probabilísticas dos dados. Para os autores, novos algoritmos e técnicas têm sido desenvolvidas, a fim de capturar e trabalhar sob circunstâncias de incerteza, a partir da evolução destes algoritmos clássicos.

3.2.1 Representação da Incerteza nos Dados

Diversos trabalhos sobre incerteza nos dados armazenam esta informação de maneira distinta. Grande parte dos trabalhos, de acordo com a revisão sistemática de Carvalho e Ruiz [CAR13], representam esses dados conforme Figura 3.7. Cada item i, em uma transação t qualquer, tem uma probabilidade $P(i,t)$, onde $0 < P(i,t) <= 1$. Na Figura 3.7, por exemplo, a transação 4 indica que o paciente tem Obesidade (O), além de 80% e 90% de probabilidade de ter Depressão (D) e Insônia (I), respectivamente.

#	
1	(H 0.6)
2	(D 0.9) (O 0.9)
3	(D 1.0) (I 0.9) (O 1.0)
4	(D 0.8) (I 0.9) (O 1.0)
5	(D 1.0) (O 0.9)
6	(H 0.8) (I 0.5)
7	(I 0.6)
8	(D 0.6)
9	(D 0.8) (I 0.7) (O 0.9)
10	(H 0.7) (I 0.8)

Figura 3.7: *Dataset* onde cada transação denota a probabilidade de um paciente estar com sintomas de Depressão (D), Hipertensão (H), Insônia (I) e Obesidade (O).

Existem alguns trabalhos que lidam um pouco diferente com a representação dos dados com incertezas. O artigo de Liang Wang et al. [WAN10] varia a forma de representar os dados com incerteza. Além de executar testes sobre *datasets* com a mesma representação dos dados ilustrada na Figura 3.7, este trabalho também gera *datasets* cujas probabilidades estão associadas às tuplas, ou seja, a probabilidade de uma transação t_j é um valor entre 0 e 1, representada por $P(t_j) \in (0,1]$. Isto indica que esta tupla existe na base de dados com probabilidade igual a $P(t_j)$.

Outro trabalho que organiza os dados de modo distinto é o artigo de Leung e Sun [LEU11], que transforma bases de dados probabilísticas de seu formato usual, na horizontal, para um formato vertical. Neste artigo a base de dados é representada por uma coleção de itens, onde cada item i específico está associado a uma lista de transações (*tidlist*). A representação $tidlist(i) : t_1 : 0.9, t_2 : 0.8, t_3 : 0.2$ indica em quais transações o item i aparece. Neste caso, o item i aparece na transação t_1 com 90% de certeza, enquanto este mesmo item aparece nas transações t_2 e t_3 com 80% e 20% de certeza, respectivamente.

O artigo de Liu [LIU12] classifica os dados como *univariate uncertain data*, onde cada atributo, em uma transação, está associado a um intervalo quantitativo. Por exemplo, um sensor de baixa sensibilidade, usado para anotar a poluição atmosférica, pode registrar um intervalo quantitativo, ao invés de um valor preciso, para indicar a quantidade de partículas suspensas no ar, às 5h da manhã.

3.3 Descoberta de Itens Frequentes sobre Dados com Incerteza

Aggarwal et al. [AGG09b] afirmam que os estudos de geração de padrões frequentes sobre dados com incerteza têm estendido algoritmos conhecidos desta tarefa. Essencialmente, as novas abordagens dos tradicionais algoritmos utilizam diferentes estratégias para reduzir o espaço de busca a cada passo dos algoritmos, bem como, procuram criar diferentes estruturas computacionais para guardar e processar as probabilidades associadas aos itens.

Aggarwal et al. [AGG09b] classificam tais algoritmos em duas classes: *Generate-and-Test* e *Pattern-Growth*. Para Liu [LIU12] os algoritmos estão divididos em três classes: *Apriori-based*, *FP-growth-based* and *H-mine-based*. Associando-se as duas classificações, pode-se dizer que a classe *Apriori-based* corresponde à classe *Generate-and-Test*. Enquanto as classes *FP-growth-based* and *H-mine-based* estão contidas dentro da classe *Pattern-Growth*.

A revisão sistemática de Carvalho e Ruiz [CAR13] apresenta diversos estudos realizados nesta área e mostra que os algoritmos mais utilizados para descoberta de conjuntos de itens frequentes em contextos com incerteza são o UApriori [CHU07] (*Apriori-based*), UF-Growth [LEU08] (*FP-growth-based*), o UFP-Growth [AGG09b] (*FP-growth-based*) e UH-Mine [AGG09b] (*H-mine-based*). Estes algoritmos utilizam estratégias e métodos distintos a fim de melhorar seu desempenho em *datasets*, principalmente quando seus itens têm probabilidades existenciais baixas.

3.3.1 UApriori

O algoritmo UApriori é da classe *Apriori-based* (ou *Generate-and-Test*). Ele utiliza uma das estratégias mais abordadas para descoberta de itens frequentes, sobre um contexto de incerteza, que é a ideia de trabalhar com algoritmos baseados em Suporte Esperado [CHU07] [CHU08] [LEU08] [AGG09b] [CAL10] [LEU09a] [LEU09b] [LEU10a] [LEU10b] [LEU11] [LIN12a] [LIU12] [TON12]. Chui, Kao e Hung [CHU07] apresentaram pela primeira vez esta abordagem, adaptando o clássico algoritmo Apriori e denominando-o de UApriori.

3.3.2 Definição de Suporte Esperado

Em algoritmos tradicionais de FIM, a contagem de suporte de um itemset X é definida como o número de transações que contém X. Em um *dataset* com incerteza, o valor do suporte é indefinido, pois não há certeza se no mundo real a transação realmente contém X. Desta forma, a definição de suporte necessita ser redefinida. Chui, Kao e Hung [CHU07] redefiniram a medida de suporte, chamando-a de suporte esperado, a fim de contemplar a incerteza dos itens.

Dado um *dataset* com incerteza D (Tabela 3.2), o qual consiste de m transações $t_1, t_2, ...t_m$ e um conjunto de itens $I = \{i_1, i_2, ..., i_n\}$. Uma transação t_j qualquer contém um conjunto de itens observados de modo independente. Cada item $i \in t_j$ está associado a uma probabilidade $P_{tj}(i)$, onde $P_{tj}(i) \in (0, 1]$. $P_{tj}(i)$ indica a probabilidade de o item i estar presente na transação t_j. Se a probabilidade $P_{tj}(i) = 0$, o item i não estará presente na transação.

Tabela 3.2: *Dataset* D, com itens associados a probabilidades existenciais.

	i_1	i_2	**...**	i_{n-1}	i_n
t_1	$P_{t_1}(i_1)$	$P_{t_1}(i_2)$	...	$P_{t_1}(i_{n-1})$	$P_{t_1}(i_n)$
t_2	$P_{t_2}(i_1)$	$P_{t_2}(i_2)$	...	$P_{t_2}(i_{n-1})$	$P_{t_2}(i_n)$
...	...	...	...	...	...
t_{m-1}	$P_{t_{m-1}}(i_1)$	$P_{t_{m-1}}(i_2)$	...	$P_{t_{m-1}}(i_{n-1})$	$P_{t_{m-1}}(i_n)$
t_m	$P_{t_m}(i_1)$	$P_{t_m}(i_2)$	...	$P_{t_m}(i_{n-1})$	$P_{t_m}(i_n)$

Observando-se um item i qualquer em uma transação t_j, existem duas possibilidades de mundos possíveis: o item $i \in t_j$; ou de modo contrário, o item $i \notin t_j$. Estes dois mundos possíveis são chamados de W_1 e W_2, respectivamente. Não é conhecido qual destes dois mundos possíveis é o mundo real, mas sabe-se a probabilidade de cada mundo ser o verdadeiro: $P(W_1) = P_{tj}(i)$ e $P(W_2) = 1 - P_{tj}(i)$.

Tome-se a transação t_j com outros itens além de i. Por exemplo, faça k ser outro item em t_j, com probabilidade $P_{tj}(k)$. Se os itens i e k são independentes, então existem 4 mundos possíveis neste contexto:

W_1: i e k não estão presentes em $t_j : P(W_1) = 1 - [P_{tj}(i) \cdot P_{tj}(k)]$

W_2: i está presente e k não está presente em $t_j : P(W_2) = P_{tj}(i) \cdot [1 - P_{tj}(k)]$

W_3: i não está presente e k está presente em $t_j : P(W_3) = [1 - P_{tj}(i)] \cdot [P_{tj}(k)]$

W_4: i e k estão presentes em t_j: $P(W_4) = P_{tj}(i) \cdot P_{tj}(k)$

Estendendo este raciocínio e imaginando-se o *dataset* D com 2 itens (i e k) e 2 transações (1 e 2), o número de mundos possíveis chega a 16, conforme ilustra a Figura 3.8. Dessa forma, pode-se generalizar o número de mundos possíveis como sendo igual a $2^{n \cdot m}$, sendo m igual ao número de transações e n a quantidade de itens.

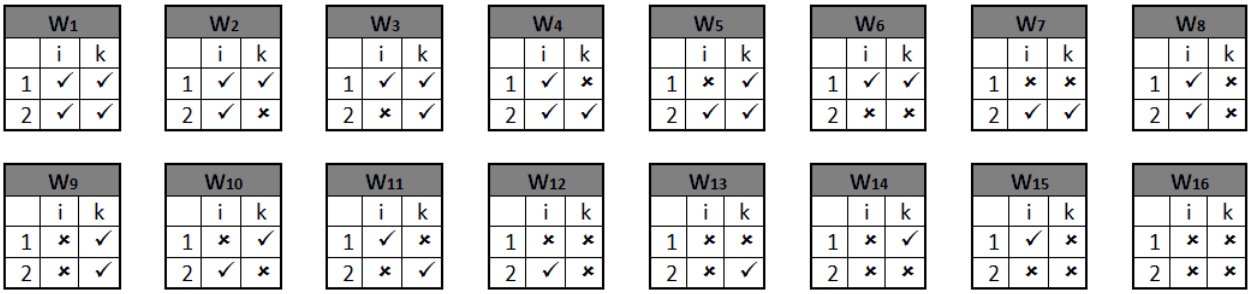

Figura 3.8: Representação dos 16 mundos possíveis para o exemplo com duas transações e dois itens (adaptado de [CHU07]).

De acordo com Chui, Kao e Hung [CHU07], dado um mundo possível qualquer W_i e um conjunto de itens (itemset) X, $P(W_i)$ é a probabilidade do mundo W_i e $S(X, W_i)$ é o suporte de X no mundo W_i. $T_{i,j}$ representa o conjunto de itens da j-ésima transação no mundo W_i. Desta forma, o cálculo da probabilidade de um mundo possível W_i existir, $P(W_i)$, é obtido por meio da Equação 3.4. Já o cálculo do suporte esperado de um itemset X, utilizando a probabilidade de um mundo possível existir, $SupEsp(X)$, é dado pela Equação 3.5.

$$P(W_i) = \prod_{j=1}^{m} \Big(\prod_{x \in T_{i,j}} P_{tj}(x) \cdot \prod_{x \notin T_{i,j}} (1 - P_{tj}(x)) \Big) \tag{3.4}$$

$$SupEsp(X) = \sum_{i=1}^{|W|} P(W_i) \cdot S(X, W_i) \tag{3.5}$$

Encontrar o suporte esperado de X, a partir das Equações 3.4 e 3.5, requer descobrir todos os mundos possíveis e calcular o suporte de X em cada mundo possível. Para [CHU07] tais tarefas são computacionalmente inviáveis, dado o número de mundos possíveis existentes. Portanto, a partir

de manipulações algébricas, [CHU07] demonstra que o suporte esperado de um itemset X pode ser calculado a partir da Equação 3.6. Conforme esta equação, o cálculo do suporte esperado de um itemset é realizado, adicionando-se o produto das probabilidades existenciais de cada item, que pertence ao itemset, para cada transação existente.

$$SupEsp(X) = \sum_{j=1}^{|D|} \prod_{x \in X} P_{t_j}(x) \tag{3.6}$$

Por meio do suporte esperado de cada itemset, Chui, Kao e Hung [CHU07] definem o conceito de um itemset frequente. *Um itemset X é frequente, se e somente se, seu suporte esperado não é menor do que $minsupesp \cdot m$, onde $minsupesp$ é um limiar mínimo de suporte especificado pelo usuário e m é o número total de transações do dataset.*

3.3.3 Itens com Probabilidades Existenciais Baixas

Chui et al., em [CHU07] e [CHU08], mostram que, se as probabilidades existenciais dos itens forem muito baixas, o algoritmo UApriori tem problemas para escalar em grandes bases de dados com incerteza. A fim de exemplificar a influência das probabilidades existenciais neste contexto, considere-se a Tabela 3.3, representando um *dataset* com 2 transações t_1 e t_2, contendo três itens i_1, i_2 e i_3, com baixas probabilidades existenciais. O suporte esperado do itemset I, de tamanho 3 (3-itemset) $I = i_1, i_2, i_3$, é calculado, de acordo com a Equação 3.6, multiplicando-se a probabilidade existencial de cada item em uma transação específica.

Tabela 3.3: Exemplo de *dataset* com baixas probabilidades existenciais em seus itens.

	i_1	i_2	i_3
t_1	0.002	0.005	0.001
t_2	0.005	0.001	0.001

Para a transação t_1, a multiplicação das probabilidades resultará em $0.002 \cdot 0.005 \cdot 0.001 = 0.00000001$. Este pequeno valor (0.00000001) será incrementado ao suporte esperado do item I, $SupEsp(I)$. Na transação t_2, o mesmo cálculo resultará em $0.005 \cdot 0.001 \cdot 0.001 = 0.000000005$. Este outro pequeno valor é adicionado novamente ao suporte esperado de I. Desta forma, o $SupEsp(I)$ será igual a $0.00000001 + 0.000000005 = 0.000000015$. Logo, se as baixas probabilidades existenciais são comuns no *dataset* D, incrementos insignificantes serão realizados e candidatos infrequentes só serão identificados como tal após o processamento da maioria das transações.

A fim de melhorar a escalabilidade deste algoritmo, principalmente quando lidam com probabilidades baixas, diversas técnicas têm sido propostas. Tais técnicas procuram reduzir o espaço de busca e, portanto, reduzir o número de itemsets gerados a cada passo do algoritmo. Chui [CHU07], além de definir o conceito de suporte esperado, também propõe um *framework* denominado *Data Trimming* para evitar suportes insignificantes de itemsets candidatos. A ideia geral deste método de poda é eliminar do *dataset* original aqueles itemsets com baixas probabilidades. Posteriormente, o *framework* faz a mineração dos dados sobre o *dataset* podado, diminuindo o custo computacional (CPU e I/O).

Em [CHU08], Chui e Kao exploram propriedades estatísticas das probabilidades existenciais dos itens a fim de diminuir a geração de itemsets candidatos. O raciocínio utilizado neste artigo é estimar progressivamente limites máximos do suporte esperado do itemset candidato, a cada transação processada. Consequentemente, em uma transação específica, se o limite máximo que o itemset

pode atingir for inferior ao limiar definido para o suporte mínimo, o itemset candidato já pode ser eliminado.

Usando *datasets* determinísticos diversos, alguns algoritmos surgiram baseados na ideia do Apriori, tais como FP-Growth [HAN00] e H-Mine [PEI01]. Estes algoritmos de descoberta de conjuntos frequentes são da classe *Pattern-Growth* e utilizam diferentes estruturas de dados para armazenar os itens e suas probabilidades. O algoritmo FP-Growth utiliza árvores (FP-tree) e o algoritmo H-Mine, por sua vez, lida com uma estrutura baseada em um array *hyper-linked*. Da mesma forma, em contextos com incerteza, evoluções sobre o algoritmo UApriori foram criadas.

3.3.4 UF-Growth

Este algoritmo é da classe *FP-growth-based*. Ele foi desenvolvido por Leung [LEU08] e baseia-se no algoritmo FP-Growth [HAN00]. A contribuição central do algoritmo FP-Growth em relação ao Apriori é que ele armazena o *dataset* dentro de uma árvore indexada chamada FP-tree. Desta forma, a árvore construída fica mais compactada do que o *dataset* original, o que diminui o espaço de busca. O algoritmo UF-Growth, assim como o FP-Growth, também constrói uma árvore indexada, denominada UF-tree, e posteriormente a minera para extrair os padrões frequentes diretamente dessa estrutura.

Para construir a UF-tree, o algoritmo varre o *dataset* e mapeia cada transação em um caminho na árvore. Como diferentes transações possuem itens comuns, os caminhos podem ser sobrepostos, permitindo uma compactação da árvore. A UF-tree é constituída de nodos que armazenam três informações: o item, o suporte esperado do item e o número de ocorrências do mesmo suporte esperado para cada item.

Inicialmente o algoritmo UF-Growth varre o *dataset* uma vez e guarda a soma do suporte esperado de cada item, ordenando-os de modo decrescente. Ele descobre os itemsets frequentes ($SupEsp(I) \geq minsupesp \cdot m$) de tamanho igual a 1, ou seja, os 1-itemsets. Os 1-itemsets infrequentes são descartados.

Considere-se um *dataset* com incerteza, representado pela Tabela 3.4 e com um suporte mínimo esperado ($minsupesp$) igual a 0.20. Após a primeira varredura sobre o *dataset*, os 1-itemsets são ordenados de acordo com seus suportes esperados e formam o conjunto $L = \{a : 2.70; b : 2.61; c : 2.51; d : 2.20; e : 2.14; f : 0.90\}$. Como o $minsupesp$ é igual a 0.20, os suportes esperados dos 1-itemsets frequentes precisam ser superiores a 1.20 ($0.20 \cdot 6 = minsupesp \cdot m$). Logo, o 1-itemset f: 0.90 é removido e os 1-itemsets remanescentes são $L = \{a : 2.70; b : 2.61; c : 2.51; d : 2.20; e : 2.14\}$.

Tabela 3.4: Exemplo de um *dataset* incerto usado para construção de uma UF-tree (adaptado de [LEU08]).

Transações	**Itens**
t_1	a:0.90; d:0.72; e:0.71; f:0.80
t_2	a:0.90; c:0.81; d:0.71; e:0.72
t_3	b:0.87; c:0.85
t_4	a:0.90; d:0.72; e:0.71
t_5	b:0.87; c:0.85; d:0.05
t_6	b:0.87; f:0.10

O próximo passo do algoritmo é varrer novamente cada transação, processando os itens na ordem de L dentro da UF-tree, criando um ramo na árvore para cada transação. Por exemplo, com a primeira transação t_1 é gerado o primeiro ramo da UF-tree, como mostra a Figura 3.9 (a). Este ramo tem três nodos (o item f de t_1 não é usado, pois ele não pertence a L). Cada

nodo é representado por um conjunto de três informações {item:probabilidade existencial:número de ocorrências no *dataset*}.

A segunda transação t_2 é avaliada $\{a:0.90; c:0.81; d:0.71; e:0.72\}$. Como o valor do suporte esperado do 1-itemset a é o mesmo de um ramo existente (o ramo de t_1 construído previamente), este nodo será compartilhado e o número de ocorrências é incrementado em uma unidade, conforme exibido na Figura 3.9 (b). É importante notar que os dois ramos criados até então compartilham um prefixo comum: $a:0.90$.

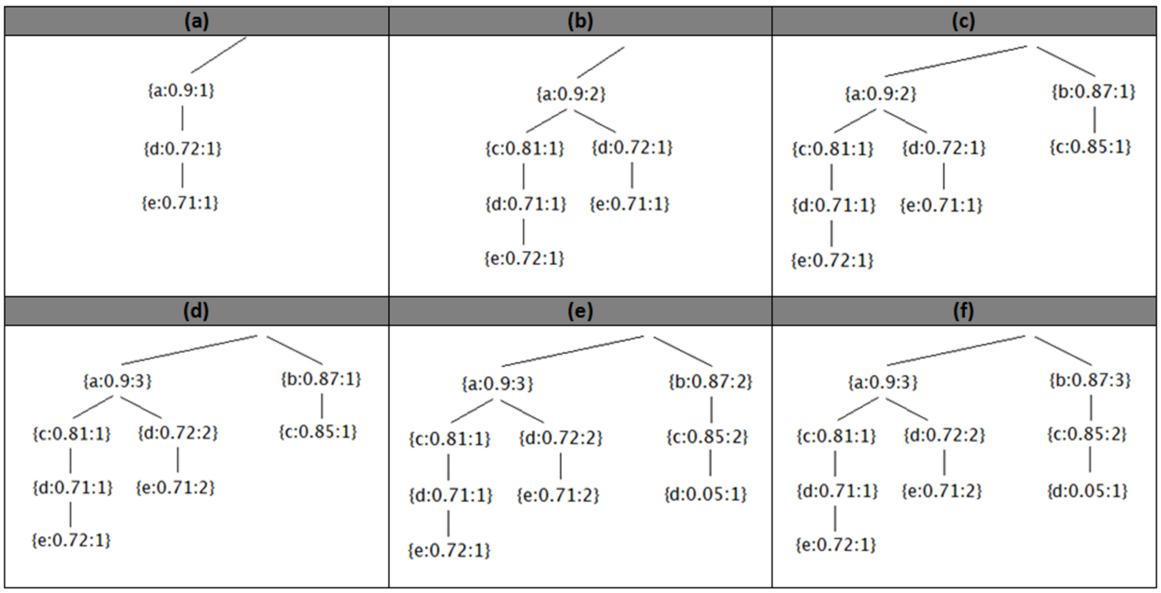

Figura 3.9: Passo a passo do algoritmo UF-Growth, ao construir a UF-tree, quando executa a segunda varredura no *dataset* com incerteza associada.

Seguindo-se com a transação t_3 $\{b:0.87; c:0.85\}$, o algoritmo UF-Growth gera um novo ramo porque ele inicia com o item b, ilustrado na Figura 3.9 (c). A transação t_4 $\{a:0.90; d:0.72; e:0.71\}$ também inicia com o item a e tem o mesmo suporte esperado do primeiro nodo, do primeiro ramo criado. Portanto, esta transação incrementa o nodo a com o valor 1. Os outros itens, $d:0.72$ e $e:0.71$, também possuem nodos com o mesmo suporte esperado e por esta razão, o ramo respectivo será usado, de acordo com a Figura 3.9 (d). Seus números de ocorrências também são incrementados em 1.

As transações t_5 $\{b:0.87; c:0.85; d:0.05\}$ e t6 $\{b:0.87; f:0.10\}$ também já estão presentes no ramo à direita com os mesmos suportes esperados. Portanto, os nodos deste ramo são adicionados em uma unidade, conforme ilustrado na Figura 3.9 (e) e (f). Observando-se os passos do algoritmo de construção da UF-tree, nota-se que os itens compartilham um nodo somente quando seus rótulos e probabilidades são os mesmos.

O próximo passo é aplicar o algoritmo UF-Growth recursivamente sobre a árvore UF-tree construída. Considere-se a árvore final UF-tree, mostrada na Figura 3.9 (f), e o mínimo suporte esperado igual a 0.20. O algoritmo UF-Growth inicia com o item e porque ele é o último item do conjunto L. O algoritmo descobre dois caminhos-prefixos até o item e : $\{(a:0.90), (c:0.81), (d:0.71)\}$ e $\{(a:0.90), (d:0.72)\}$. Tomando-se e como um sufixo, os dois caminhos-prefixos formam um padrão condicional (*conditional pattern base*) de e. Usando este padrão condicional, é possível construir a *{e}-projected-DB*, conforme Figura 3.10 (a), e calcular seu suporte esperado.

No primeiro caminho-prefixo de e, o item tem uma única ocorrência, com probabilidade existencial de 0.72 e no segundo caminho prefixo, ele aparece duas vezes com probabilidade existencial de 0.71. Sendo assim $SupEsp(\{e\}) = 2.14$, isto é, $1 \cdot 0.72 + 2 \cdot 0.71$. A partir destes caminhos, podem ser

geradas todas as combinações de padrões frequentes $\{a,e\}$, $\{d,e\}$ e $\{c,e\}$. A fim de calcular o suporte esperado (SupEsp) de uma extensão de um padrão frequente $X(X \cup \{y\})$, é necessário multiplicar o $SupEsp(\{y\})$ pelo suporte esperado de X, $SupEsp(X)$.

Portanto, o padrão $\{c,e\}$ é infrequente porque $SupEsp(\{c,e\})$ é igual a $(1 \cdot 0.72 \cdot 0.81) = 0.5832$, inferior ao resultado de $minsupesp \cdot m = 1.20$. Contudo os padrões $\{a,e\}$ e $\{d,e\}$ são frequentes, porque $SupEsp(\{a,e\}) = (1 \cdot 0.72 \cdot 0.9 + 2 \cdot 0.71 \cdot 0.9) = (0.648 + 1.278) = 1.926$ e $SupEsp(\{d,e\})$ é $(1 \cdot 0.72 \cdot 0.71 + 2 \cdot 0.71 \cdot 0.72) = (0.5112 + 1.0224) = 1.5336$. Ambos os suportes esperados são maiores do que o suporte mínimo especificado.

Posteriormente o algoritmo UF-Growth extrai da UF-tree o {d,e}-projected DB. Ele consiste de a, que expressa o padrão frequente $\{a,d,e\}$, com $SupEsp(\{a,d,e\}) = 3 \cdot 0.5112 \cdot 0.90 = 1.38024$, onde $0.5112 = (0.71 \cdot 0.72)$. A UF-tree para {d,e}-projected DB é exibida na Figura 3.10 (b).

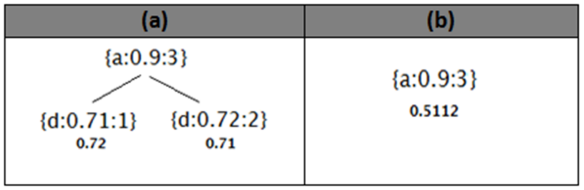

Figura 3.10: (a) A UF-tree para {e}-projected DB e (b) a UF-tree para {d,e}-projected DB.

Concluindo, o algoritmo UF-growth lida com os itens {d}, {c} e {b} para encontrar seus superconjuntos frequentes, conforme realizado para {e}. Consequentemente, ao aplicar o algoritmo UF-Growth sobre a UF-tree, são encontrados os seguintes padrões frequentes: $L = \{\{a\},\{a,d\},\{a,d,e\},\{a,e\},\{b\},\{b,c\},\{c\},\{d\},\{d,e\},\{e\}\}$.

3.4 Considerações Finais do Capítulo

Este capítulo faz um apanhado sobre a etapa que é o gargalo da técnica descritiva de mineração de dados denominada Análise de Associação: a Descoberta de Itens Frequentes. Contextualiza-se esta técnica dentro do processo de KDD e discute-se o tradicional algoritmo Apriori.

Posteriormente dá-se atenção ao crescimento de aplicações que envolvem a incerteza nos *datasets* e as maneiras de representá-la nos *datasets*. Desta forma, surge a necessidade de adaptação da etapa de Descoberta de Itens Frequentes sobre *datasets* com incerteza associada aos seus dados.

Na última seção dois algoritmos de Descoberta de Itens Frequentes que lidam com *datasets* com incerteza são apresentados: o UApriori e o UF-Growth. Eles evoluem os algoritmos tradicionais Apriori e FP-Growth, respectivamente. Nesta seção também elucida-se a definição de suporte esperado, redefinição do conceito de suporte feita pela primeira vez por Chui et al. [CHU07]. Ainda é abordado o problema de itens com probabilidades existenciais baixas no *dataset*, o que prejudica a escalabilidade dos algoritmos que lidam com este tipo de dados.

4. TRABALHOS RELACIONADOS

Este capítulo relata alguns trabalhos relacionados com a descoberta de conjuntos de itens frequentes. Ele está dividido em cinco seções. A primeira discute trabalhos que integram o algoritmo UApriori em *datasets* com incerteza associada. A segunda seção relata alguns trabalhos que utilizam o modelo de programação MapReduce com o algoritmo Apriori sobre *datasets* determinísticos. A terceira seção discute as vantagens de utilização do modelo de programação MapReduce em relação a outras abordagens para construção de algoritmos paralelos e distribuídos. A quarta seção apresenta um trabalho envolvendo um algoritmo de descoberta de itens frequentes, usando MapReduce sobre *datasets* com incerteza associada. A última seção enumera algumas considerações a respeito deste capítulo.

4.1 Trabalhos de FIM sobre *Datasets* com Incerteza Associada

Nesta seção são abordados e discutidos os experimentos de dois trabalhos bastante referenciados na área: Aggarwal et al. [AGG09b] e Tong et al. [TON12], que implementam o tradicional algoritmo Apriori sobre *datasets* com incerteza associada, utilizando a redefinição de suporte feita por [CHU07], detalhada na Seção 3.3.2, Equação 3.6.

4.1.1 Os Experimentos de Aggarwal et al.

O trabalho de Aggarwal et al. [AGG09b] estuda o problema de minerar itens frequentes sobre *datasets* com incerteza. Ele implementa e discute algoritmos das classes *Generate-and-Test* e *Pattern Growth*. Dentre suas observações, os autores evidenciam que o comportamento destas classes de algoritmos sobre *datasets* determinísticos é muito diferente de quando aplicados em *datasets* com incerteza associada. De modo mais preciso, os algoritmos UApriori e UH-Mine tem melhor desempenho do que algoritmos da classe *FP-growth-based*, baseados em árvores, quando aplicados sobre *datasets* com incerteza.

Aggarwal et al. [AGG09b] utiliza 4 *datasets* distintos em seus experimentos. Dois *datasets* são reais: connect4 e kosarak e dois são sintéticos: T40I10D100k e T25I15D320k. Os testes aplicados têm o propósito de analisar o tempo de execução e o consumo de memória dos algoritmos. Eles são realizados sobre todos os *datasets*, variando o mínimo suporte esperado. Sobre todos os *datasets* são executados os algoritmos UApriori, UH-Mine e UFP-Growth (variação do algoritmo UF-Growth).

Os *datasets* reais, connect4 e kosarak, são muito densos ($d = 0.33$) e muito esparsos ($d = 0.00019$), respectivamente. A densidade (d) de um *dataset* é calculada a partir da Equação 4.1, onde TMT e NID significam, respectivamente, Tamanho Médio de uma Transação e Número de Itens distintos no *Dataset*.

$$d = \frac{TMT}{NID} \tag{4.1}$$

Com relação aos testes aplicados sobre o *dataset* real e denso connect4, os tempos de execução dos algoritmos UH-Mine e UApriori foram muito semelhantes, com leve vantagem para o algoritmo UApriori. Em relação ao consumo de memória, o algoritmo UH-Mine teve um desempenho melhor do que o UApriori, e ambos foram mais eficientes do que o UFP-Growth.

Quando os testes foram aplicados no *dataset* real e esparso kosarak, os resultados foram semelhantes aos anteriores. Os algoritmos UApriori e UH-Mine tiveram um melhor tempo de processamento em relação ao algoritmo UFP-Growth, com pequena vantagem para o algoritmo UApriori à

medida que o suporte mínimo esperado decrescia. Houve um leve ganho de desempenho para o algoritmo UH-Mine, quando o suporte mínimo esperado era mais alto.

Os *datasets* sintéticos, T40I10D100K e T25I15D320k, foram gerados a partir da ferramenta *IBM Synthetic Data Generator* [AGR94]. As incertezas foram acrescentadas para cada item do *dataset* a partir de uma distribuição normal $N(\mu, \sigma^2)$. Neste artigo o valor da média (μ) foi gerado independentemente e aleatoriamente dentro da faixa [0.87,0.99], e o desvio padrão (σ) usou a faixa [1/21,1/12].

Os testes aplicados sobre o *dataset* T40I10D100K mostraram uma superioridade do algoritmo UH-Mine em relação aos demais algoritmos, tanto em tempo de execução, quanto em uso de memória. À medida que o suporte mínimo reduzia, a distância entre o tempo de execução do UH-Mine e do UApriori aumentou. Com relação ao consumo de memória, os experimentos mostraram que, devido a geração de muitos conjuntos candidatos, o algoritmo UApriori necessitou de mais memória do que o UH-Mine.

Nos testes realizados sobre o *dataset* T25I15D320k (maior *dataset* – em torno de 80,04 MB), com um mínimo suporte esperado de 0.5, todos os algoritmos exibiram escalabilidade linear. Os algoritmos UH-Mine e UApriori apresentaram melhor escalabilidade, tanto em consumo de memória, quanto em tempo de processamento do que o algoritmo UFP-Growth. O algoritmo UH-Mine consumiu menos recursos de memória do que o UApriori, mas em relação ao tempo de processamento, ambos algoritmos foram muito similares (em torno de 50s).

Analisando os testes de Aggarwal et al. [AGG09b] nota-se um melhor resultado dos algoritmos UApriori e UH-Mine, sobre todos os 4 *datasets* utilizados, em relação ao algoritmo UFP-Growth. A estratégia dos algoritmos da classe Pattern-Growth, de compactar o *dataset* em uma árvore indexada, embora muito eficiente quando usada sobre dados determinísticos [HAN00], não demonstrou a mesma eficiência quando aplicada sobre dados probabilísticos. Isto porque os itens do *dataset* podem estar associados a diferentes probabilidades existenciais, em diferentes transações, principalmente quando considerado um número grande de casas decimais. Com diferentes probabilidades, muitos ramos são criados para cada item na UF-tree e, desta forma, a árvore deixa de ficar tão compacta, prejudicando os algoritmos da classe *Pattern-Growth*, que são baseados em árvores.

4.1.2 Os Experimentos de Tong et al.

O artigo de Tong et al. [TON12] discute, dentre outros assuntos, o desempenho de algoritmos que descobrem itemsets frequentes baseando-se em seus suportes esperados. Os autores executam testes comparando os algoritmos UApriori, UFP-Growth e UH-Mine, os mesmos dos experimentos de Aggarwal et al. [AGG09b]. Embora o algoritmo Apriori seja mais lento do que os outros dois algoritmos, em versões sobre *datasets* determinísticos, sua versão utilizando dados com incerteza associada, ou seja, o UApriori, apresentou melhor desempenho dentre os algoritmos testados, principalmente quando os *datasets* eram densos, confirmando a mesma conclusão dos experimentos de [AGG09b].

Tong et al. [TON12] utilizou 5 *datasets* para testar a escalabilidade dos três algoritmos. Dois deles são *datasets* reais e densos, accidents e connect, dois reais e esparsos, gazelle e kosarak, e um *dataset* sintético denso T25I15D320k. Em seus experimentos, as probabilidades também foram inseridas nos *datasets* através da ferramenta *IBM Synthetic Data Generator*, com as médias (μ) e variâncias (σ^2) conforme exibidas na Tabela 4.1.

Nos experimentos de Tong et al. [TON12], o tempo de processamento dos algoritmos também são recuperados a partir do decréscimo gradual do mínimo suporte esperado. À medida que o suporte mínimo é reduzido, os algoritmos aumentam seu tempo de processamento. Os autores mostram que o algoritmo UFP-Growth é mais lento que os outros dois para todos os *datasets* reais. Demonstram

Tabela 4.1: Valores de média e variância usados no trabalho de Tong et al. [TON12].

dataset	μ	σ^2	d
Accidents	0.50	0.50	0.50
Connect	0.95	0.05	0.072
Gazelle	0.95	0.05	0.005
Kosarak	0.50	0.50	0.00019
T25I15D320k	0.90	0.10	0.025

também que o algoritmo UApriori tem um melhor desempenho sobre o algoritmo UH-Mine para os *datasets* densos (connect e accident), enquanto o UH-Mine obtém melhores resultados quando os testes são realizados sobre *datasets* esparsos (kosarak e gazelle).

O trabalho de Tong et al. [TON12] também discute a escalabilidade dos algoritmos a partir de testes sobre o *dataset* sintético T25I15D320k, variando o tamanho deste *dataset* de 20k até 320k. O tempo de processamento dos três algoritmos é linear. À medida que o tamanho do *dataset* cresce, o tempo de processamento do algoritmo UApriori aproxima-se do algoritmo UH-Mine. Ambos mostram-se mais eficientes do que o algoritmo UFP-Growth. Quanto à memória utilizada, o trabalho relata que o algoritmo UApriori tem um acréscimo de memória mais lento à medida que o *dataset* vai aumentando.

4.2 Trabalhos de FIM e MapReduce sobre *Datasets* Determinísticos

Esta seção discute outros três trabalhos que integram algoritmos FIM com o modelo de programação MapReduce. Estes trabalhos descobrem conjuntos de itens frequentes integrando o tradicional algoritmo Apriori com o modelo de programação MapReduce. Todos eles, Lin, Lee e Hsueh [LIN12b]; Li et al. [LI12]; Yahya, Hegazi e Ezat [YAH12], realizam seus experimentos sobre *datasets* determinísticos. Não foram encontrados, nas pesquisas e revisão sistemática [CAR13] realizadas, trabalhos envolvendo Apriori, MapReduce e *datasets* com incerteza associada.

4.2.1 Os Experimentos de Lin, Lee e Hsueh

Neste trabalho os autores Lin, Lee e Hsueh [LIN12b] elaboram três algoritmos distintos, todos baseados no Apriori, e os integram com o modelo de programação MapReduce. A utilização das funções Map e Reduce, em um ambiente paralelo e distribuído, ameniza as fraquezas do algoritmo Apriori: a geração de muitos candidatos, principalmente quando o valor de suporte mínimo é baixo e as múltiplas varreduras realizadas sobre o *dataset*.

Os algoritmos são nomeados de *Single Pass Counting* (SPC), *Fixed Passes Combined-counting* (FPC) e *Dynamic Passes Combined-counting* (DPC). O algoritmo SPC é uma conversão direta do algoritmo Apriori para uma versão MapReduce. Os outros dois algoritmos, baseados no SPC, procuram minimizar o número de varreduras no *dataset* e com isto reduzir o tempo de processamento.

O algoritmo SPC é constituído de k passos. A cada passo k, o algoritmo faz uma varredura no *dataset*. A função Map gera os conjuntos de k-itemsets candidatos (C_k) e, por consequência, as saídas $< X, 1 >$, onde X é um conjunto candidato existente na transação t_i. A função Reduce recebe a contagem de suporte de cada C_k e verifica aqueles que são maiores do que o suporte mínimo especificado, criando, no passo k, um conjunto de itens frequente denominado L_k. No passo $k+1$, os conjuntos candidatos (C_{k+1}) são gerados a partir do conjunto L_k (passo anterior). A função Map atua da mesma maneira sobre os conjuntos C_{k+1} e, quando a função Reduce verifica os suportes de cada C_{k+1}, é gerado o conjunto L_{k+1}. As Figuras 4.1 e 4.2 exibem os passos $k = 1$ e $k > 1$

do algoritmo SPC e como as funções Map, Combine e Reduce estão relacionadas. Considere-se o suporte mínimo igual a 0.25.

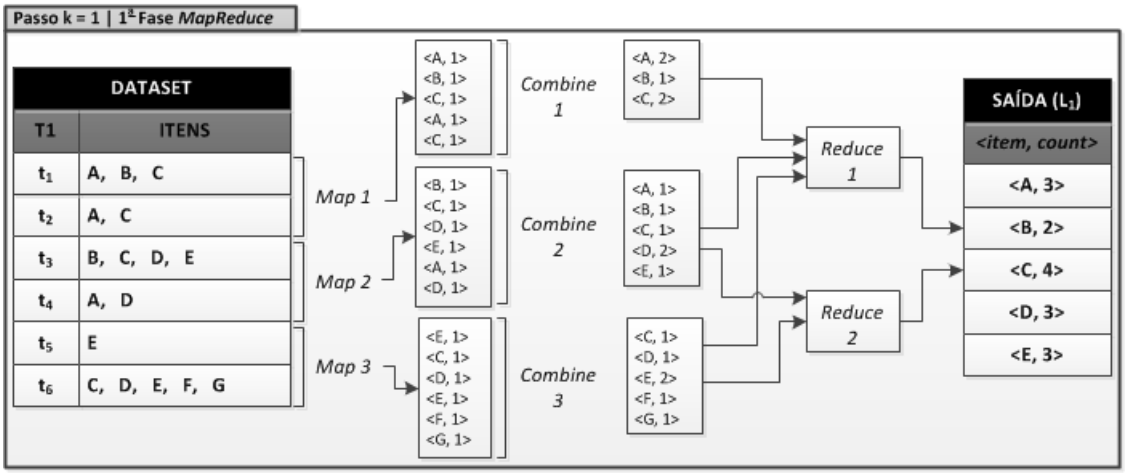

Figura 4.1: Execução do primeiro passo ($k = 1$) do algoritmo SPC (adaptado de [LIN12b]).

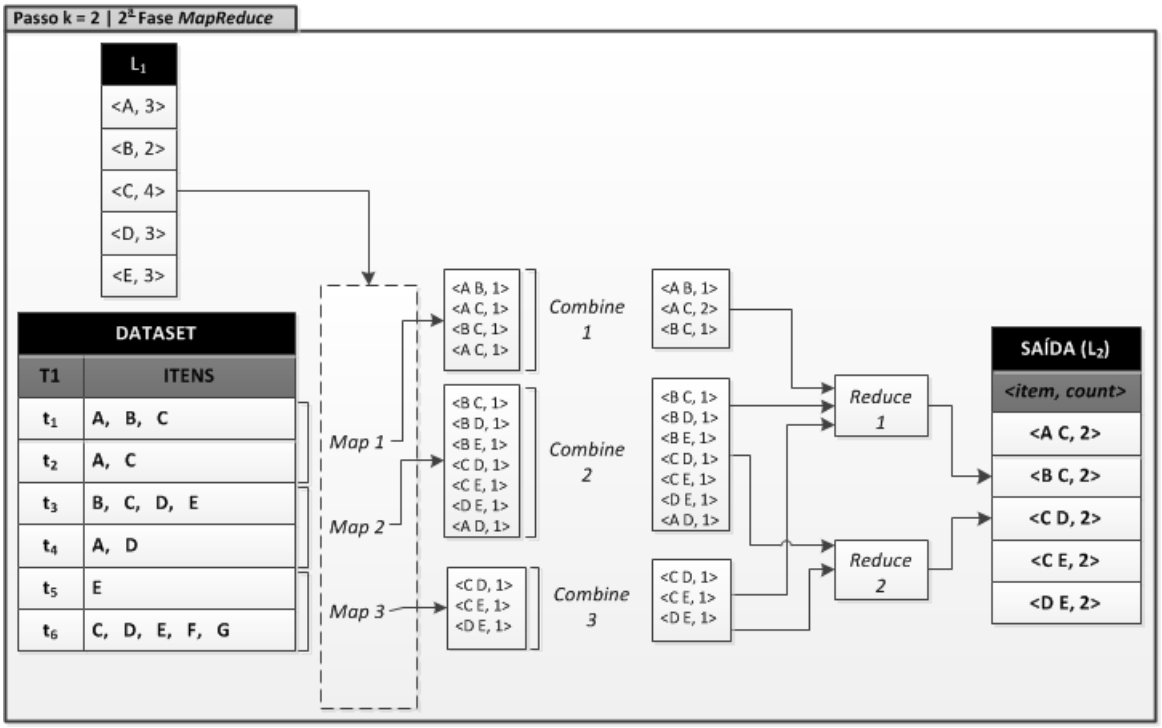

Figura 4.2: Execução do segundo passo ($k = 2$) do algoritmo SPC (adaptado de [LIN12b]).

Observa-se na Figura 4.1 que os 1-itemsets F e G são eliminados pela função Reduce e não fazem parte do conjunto L_1, pois seus suportes são inferiores a 0.25 ($= 0.16$). Nos passos subsequentes não existirão conjuntos candidatos contendo os itens F e G, a partir da aplicação da propriedade do tradicional algoritmo Apriori. Na Figura 4.2, a função Map 1, aplicada sobre seu *split* (Map 1 sobre t_1 e t_2, por exemplo), verifica quais os conjuntos candidatos de tamanho 2 (baseados em L_1) estão presentes nas transações. A mesma operação ocorrerá para as funções Map 2 e Map 3. A função Combine, associada a cada função Map, faz uma operação semelhante a um *group by*, somando os valores por itens encontrados. Posteriormente, as funções Reduce fazem o agrupamento do suporte

de todos os 2-itemsets e verificam quais deles são maiores do que o suporte mínimo. Os demais passos do algoritmo SPC são semelhantes ao passo 2, até que no passo k o conjunto L_{k-1} seja vazio.

Neste trabalho, os autores ainda desenvolvem dois outros algoritmos baseados no SPC, que são os algoritmos FPC e DPC. Estes dois possuem heurísticas aplicadas a partir do terceiro passo do algoritmo SPC. O algoritmo FPC combina conjuntos candidatos de diversos passos em somente uma fase MapReduce. No artigo, a partir do passo $k = 3$, são gerados os k-itemsets, (k+1)-itemsets e os (k+2)-itemsets em uma única fase MapReduce. Isto é, para o passo k=3, os conjuntos candidatos 3-itemsets, 4-itemsets e 5-itemsets são gerados e todos são avaliados a partir de uma única varredura no *dataset*. O objetivo desta heurística é diminuir o número de varreduras no *dataset* original, embora aumente o número de conjuntos candidatos produzidos a cada fase MapReduce.

O algoritmo DPC é proposto para encontrar um equilíbrio entre reduzir o número de fases MapReduce (menos varreduras no *dataset*) e também diminuir o número de conjuntos candidatos gerados. Neste algoritmo foi embutida uma variável *ct* que controla quantos (k+m)-itemsets serão gerados, a cada fase MapReduce, além dos k-itemsets. Isto difere do algoritmo FPC, que tem um número fixo de $m = 3$. A variável *ct*, inserida neste algoritmo, é dinâmica para cada fase MapReduce, pois está baseada no tamanho do conjunto de itens frequentes gerados no passo anterior $|L_{k-1}|$ e no intervalo de tempo para que L_{k-1} fosse gerado.

Os três algoritmos foram desenvolvidos com a API Java e seus experimentos foram realizados em um *cluster* com 4 nodos, onde cada nodo era um Intel Pentium Dual Core E6500 2.93GHz CPU, 4GB RAM e 500GB de disco rígido, usando uma distribuição Ubuntu 10.10. O *cluster* usou a versão 0.21.0 da implementação Apache Hadoop. Todos os testes foram configurados para rodar com 7 funções Map e 1 função Reduce.

Nos experimentos foram usados *datasets* reais: BMS-POS[1] e BMS-WebView-1[2], do repositório FIMI[3]. Também foram realizados testes em *datasets* sintéticos, gerados com o *IBM Synthetic Data Generator*. Segundo Lin, Lee e Hsueh [LIN12b], os experimentos em *datasets* sintéticos com diversos tamanhos, número de itens distintos e tamanho médio de itens por transação apresentaram resultados semelhantes. As execuções dos três algoritmos, sobre *datasets* sintéticos, utilizando suportes mínimos maiores do que 0.2, obtiveram tempos de execução semelhantes. À medida que o suporte foi diminuindo, os algoritmos FPC e DPC tornaram-se mais eficientes do que o algoritmo SPC. O algoritmo DPC teve menor tempo de execução em relação ao FPC, pois não gerou tantos conjuntos candidatos a cada passo.

Sobre *datasets* reais, o algoritmo FPC revelou uma fragilidade. Quando o suporte mínimo considerado é 0.125, tanto o FPC, quanto o DPC tiveram melhores desempenhos do que o algoritmo SPC. No entanto, quando o suporte mínimo é reduzido, o algoritmo FPC, com sua estratégia de gerar todos os conjuntos candidatos k-itemsets, (k+1)-itemsets e (k+2)-itemsets, a cada fase MapReduce, não produziu bons resultados. Nos experimentos sobre o *dataset* BMS-WebView-1, por exemplo, a geração de 3-itemsets, 4-itemsets e 5-itemsets, em uma fase MapReduce do algoritmo FPC, gerou um total de 431.883 candidatos. Enquanto isto, o número de conjuntos candidatos gerados no algoritmo original SPC foram 20.066 (17.137+2.688+241), para os mesmos passos. Ou seja, o algoritmo FPC desperdiça muito tempo gerando conjuntos candidatos que não se constituirão em conjuntos de itens frequentes ao final do algoritmo.

O trabalho de Lin, Lee e Hsueh [LIN12b] detalha com clareza a possibilidade de integrar o algoritmo Apriori com o modelo de programação MapReduce, por meio da implementação dos algoritmos

[1] Este *dataset* contém valores de pontos de venda de um grande varejista de eletrônicos.

[2] Este *dataset* contém diversos meses com cliques de usuários sobre dois websites de *e-commerce*.

[3] *Frequent Itemset Mining Dataset Repository*: http://fimi.cs.helsinki.fi/data/, repositório público de *datasets* usados em trabalhos de mineração de dados.

SPC, FPC e DPC. Abre espaço também para que novos experimentos possam ser realizados com estes algoritmos, variando o número de funções Map e Reduce. Experimentos em outros *datasets* reais também permitiriam identificar quais algoritmos têm um melhor desempenho em relação às densidades dos mesmos.

4.2.2 Os Experimentos de Li et al.

O trabalho de Li et al. [LI12] também reprojeta e converte o algoritmo Apriori para suportar o modelo de programação MapReduce. A proposta deste novo algoritmo é implementada e avaliada sobre a plataforma Amazon Elastic Cloud Computing (Amazon EC2). O artigo demonstra que o algorimto criado apresenta bons resultados experimentais.

O algoritmo funciona de modo semelhante ao algoritmo SPC [LIN12b]. Inicialmente o *dataset* é salvo no HDFS. O *dataset* é dividido em *splits*, onde cada um deles é alocado para uma função Map. A saída de cada função Map tem o formato <chave/valor>, onde o item é a chave e seu contador de frequência (suporte) é o valor. A saída da função Map é a entrada para a função Combine, que combina todos os valores relacionados a um item (chave) particular. A saída da função Combine serve como entrada para a função Reduce que soma os valores correspondentes a um item. Com tais somas, a função Reduce verifica se o suporte de um item é maior do que o suporte mínimo especificado. Em caso positivo, a função escreve o item e seu valor na saída. Este processo gera os 1-itemsets no primeiro passo.

A partir dos 1-itemsets gerados pela função Map, inicia-se a geração dos 2-itemsets. Portanto, a saída da função Map entra na função Combine, que posteriormente entrega as chaves e seus valores para a função Reduce fazer a soma e verificar quais 2-itemsets obedecem ao mínimo suporte especificado. Este processo é repetido para os diversos k-itemsets, até que nenhum conjunto candidato seja gerado no passo atual ou até que conjuntos de itens frequentes não sejam encontrados no passo anterior. Após k passos a saída contém os 1-itemsets, 2-itemsets, ... e os k-itemsets.

Os dados gerados pela função Map, a cada passo, são escritos em arquivos temporários no HDFS. Depois eles são capturados e processados pela função Combine. Por último, a saída da função Combine também é escrita em arquivos temporários para que a função Reduce possa processá-la. Enquanto os dados estão sendo processados pela função Reduce, eles são salvos em arquivos temporários até que, ao final do processamento, os dados são colocados em um arquivo permanente. A Figura 4.3 ilustra a saída de cada uma das funções aplicadas: Map, Combine e Reduce.

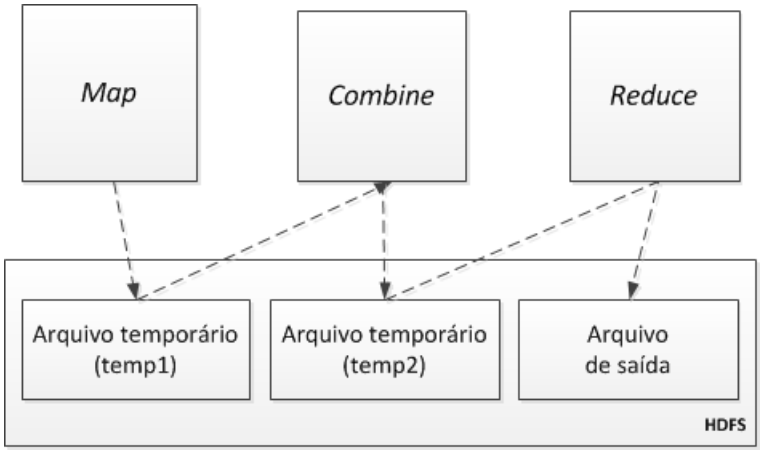

Figura 4.3: Fluxo de dados entre as funções Map, Combine, Reduce e o sistema de arquivos distribuídos HDFS (adaptado de [LI12]).

Este algoritmo é desenvolvido sobre a implementação Apache Hadoop, versão 0.20.0, dentro do

serviço Amazon EC2. Os arquivos de entrada e saída são salvos sobre o serviço de armazenamento Amazon S3. Os arquivos temporários de dados são armazenados no HDFS e o serviço Amazon Elastic MapReduce (Amazon EMR) é responsável por gerenciar o *cluster* Hadoop, rodar os *jobs* necessários, finalizar os *jobs* e mover os dados entre o Amazon EC2 e Amazon S3. Segundo os autores, a utilização do Amazon EMR elimina diversas dificuldades associadas à configuração do Apache Hadoop.

A Figura 4.4 exibe o fluxo de trabalho de um *job* dentro do serviço Amazon EMR. Uma requisição é realizada para iniciar o *job* (1). O serviço cria e inicia o *namenode* e os *datanodes* do *cluster* Hadoop (2). Os arquivos temporários são salvos no HDFS ou no Amazon S3 (3) e (4). O arquivo de saída final é armazenado no Amazon S3. Quando o *job* termina, uma mensagem é enviada ao usuário indicando que ele foi finalizado.

Os experimentos de Li et al. [LI12] foram realizados sobre 3 *datasets* reais: chess, connect e mushroom, obtidos do repositório UCI[4] e um *dataset* sintético T10I4D100k. O algoritmo foi aplicado sobre todos os *datasets* variando o suporte mínimo de 0.95 à 0.75. O *dataset* connect foi aquele que consumiu o maior tempo de execução do algoritmo. Embora o *dataset* sintético tenha um número maior de transações, o *dataset* real connect é o mais denso e, por este motivo, muitos passos do algoritmo são executados a fim de encontrar todos os itemsets frequentes.

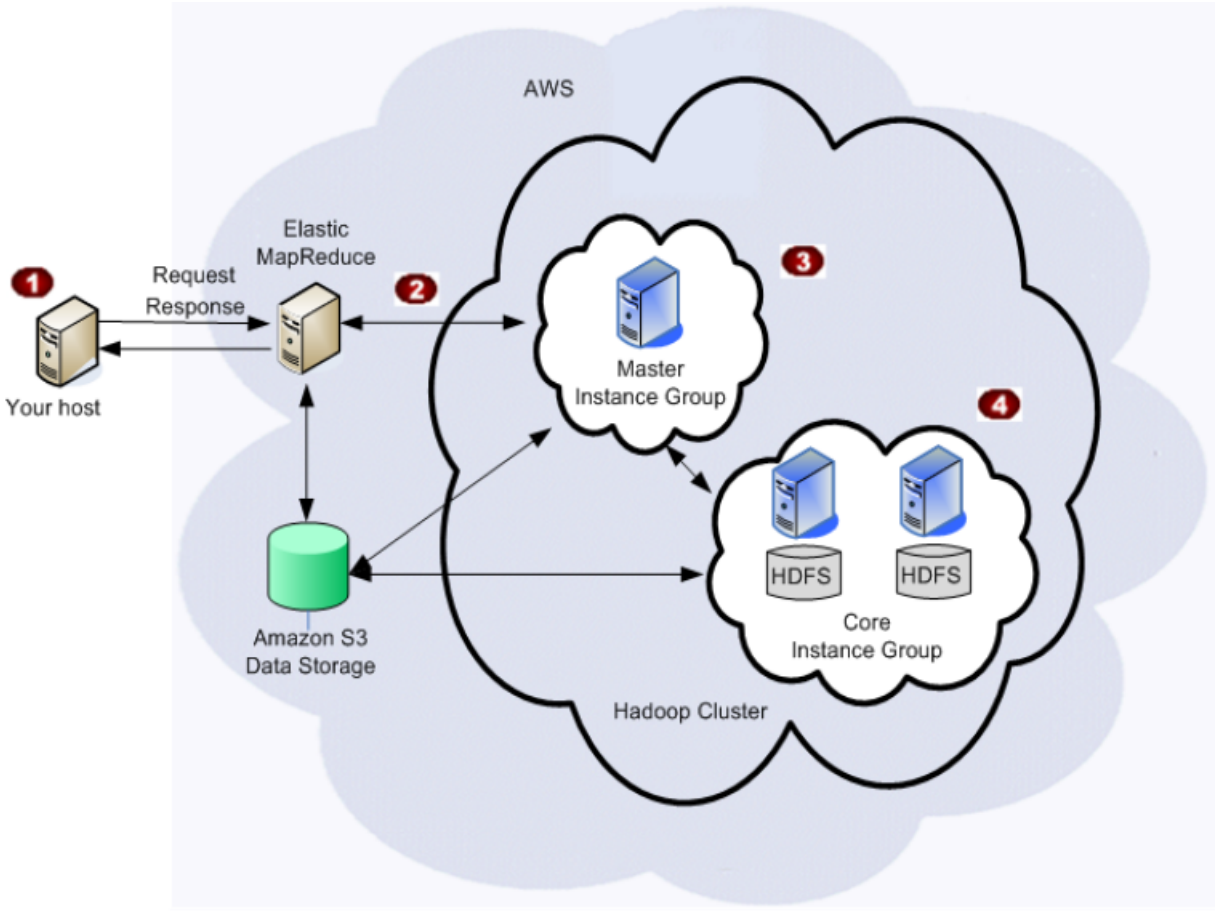

Figura 4.4: Fluxo do *job* na plataforma Amazon EC2, utilizando os serviços Amazon S3 e Amazon Elastic MapReduce (adaptado de [LI12]).

Os experimentos iniciais confrontaram os resultados sobre um *cluster* Hadoop de um único nodo

[4] *UCI Irvine Machine Learning Repository*: http://archive.ics.uci.edu/ml/, repositório público com 264 *datasets* para utilização da comunidade de machine learning.

e sobre uma máquina sem Hadoop. O desempenho do algoritmo sobre a máquina sem Hadoop foi melhor do que sobre o *cluster* de um nodo. Isto aconteceu porque o tempo gasto pelo Hadoop com outras atividades, tais como manter o *cluster* funcionando, dividir o *dataset* e escrever os resultados nas saídas, é grande. Logo, não compensa utilizá-lo para execução do algoritmo Apriori, quando o *cluster* tem apenas um nodo.

No entanto, quando o algoritmo é aplicado sobre *clusters* com 2, 4 e 8 nodos, os bons resultados dos experimentos são expressivos. Para os 4 *datasets*, quando utilizada uma implementação Hadoop com mais de um nodo, o algoritmo teve uma execução mais eficiente do que o algoritmo aplicado sobre uma máquina sem Hadoop. Quando utilizados 8 nodos, para o *dataset* denso connect, o algoritmo rodou cerca de 4 vezes mais rápido do que quando aplicado sobre uma máquina sem Hadoop.

4.2.3 Os Experimentos de Yahya, Hegazi e Ezat

Para Yahya, Hegazi e Ezat [YAH12], o Apriori é o algoritmo mais utilizado para encontrar itens frequentes em *datasets* transacionais. No entanto, quando os *datasets* são grandes, por varrer o *dataset* múltiplas vezes e gerar muitos itemsets candidatos, os custos de memória e processamento são muito altos. Além disso, quando o algoritmo roda sobre somente um processador, os recursos de CPU e memória ficam muito limitados, prejudicando seu desempenho.

Os autores afirmam que a integração do algoritmo Apriori e o modelo de programação MapReduce melhora o desempenho deste algoritmo. Eles classificam as iniciativas de integração dos algoritmos Apriori e MapReduce em duas classes: aqueles algoritmos que executam em 1-fase e os que rodam em k-fases. No algoritmo de 1-fase é necessário somente 1 *job* MapReduce para encontrar todos os k-itemsets [LI11]. Os algoritmos de k-fases executam em k *jobs* MapReduce para descobrir todos os k-itemsets [LIN12b] [LI12]. Ou seja, neste último caso, a fase 1 (*job* 1) descobre os 1-itemsets, a fase 2 (*job* 2) os 2-itemsets, e assim sucessivamente, até que a fase k (*job* k) descubra os k-itemsets.

O algoritmo implementado no trabalho de Yahya, Hegazi e Ezat [YAH12] é o MRApriori, também baseado no modelo de programação MapReduce. Este algoritmo é implementado em somente duas fases (dois *jobs* MapReduce), nas quais todos os k-itemsets frequentes são descobertos. Na fase 1, durante cada função Map, o algoritmo Apriori é aplicado sobre todo o *split*, ao invés de ser aplicado a cada transação, conforme os algoritmos 1-fase e k-fases. Ao final da fase 1, a função Reduce tem como saída todos os k-itemsets gerados, o conjunto frequente parcial L_p.

Na fase 2, a função Map recebe como entrada o arquivo com todos os k-itemsets gerados na fase anterior, ou seja, o conjunto L_p. A função Map conta a frequência de cada k-itemset sobre seu *split*, gerando uma lista chave/valor, onde a chave é um k-itemset e valor é o total de ocorrências desta chave no *split*. A função Reduce então processa as saídas das funções Map, somando as ocorrências encontradas de cada k-itemset e gerando todos os itemsets frequentes.

Neste trabalho o algoritmo MRApriori é comparado com outros dois algoritmos: 1-fase e k-fases. Os experimentos foram executados três vezes para cada algoritmo e suas médias foram calculadas. Estes experimentos foram realizados sobre uma única máquina com a seguinte configuração: Windows 7 64-bit, ferramenta Cygwin, Apache Hadoop versão 0.20.2 rodando em modo *stand-alone*. Todos os algoritmos foram implementados em Java e apenas um *dataset* sintético foi utilizado nos experimentos, T10I4D100k, gerado pela ferramenta *IBM Synthetic Data Generator* [AGR94].

Os resultados dos experimentos mostraram que o algoritmo MRApriori teve um melhor desempenho em relação aos outros dois algoritmos. O algoritmo 1-fase mostrou-se bastante ineficiente, pois consumiu muito tempo de processamento. Desta forma, os autores não o utilizaram em comparações, focando a discussão sobre os resultados do algoritmo MRApriori e do algoritmo de k-fases.

À medida que o mínimo suporte diminui, o tamanho máximo de itemsets frequentes aumenta

e, portanto, mais *jobs* são necessários. Logo, o algoritmo de k-fases precisa executar k *jobs*. Cada *job* tem um tempo de execução extra para iniciar, para instanciar a JVM (1 JVM é instanciada por *job*), exigindo um esforço extra ao nodo *master*, pois é ele que coordena e monitora os k *jobs*. Em contrapartida, no algoritmo MRApriori sempre são executados somente dois *jobs*, 2 fases MapReduce, reduzindo o trabalho extra do nodo *master*.

Em relação a este artigo duas observações podem ser realizadas. A primeira delas é que seus experimentos foram realizados sobre somente um nodo. De acordo com o trabalho de Li et al. [LI12], a execução de seu algoritmo sobre somente um nodo, usando Apache Hadoop, mostrou-se ineficiente. Além disso, os experimentos foram realizados sobre somente 1 *dataset* sintético, não permitindo conclusões sobre o comportamento dos algoritmos de acordo com a densidade dos *datasets*.

4.2.4 MPI e OpenMP

De acordo com Hayduk [HAY12], diversos algoritmos paralelos, baseados no Apriori, têm sido propostos usando as bibliotecas MPI (*Message Passing Interface*) [MPI14] e OpenMP (*Open Multi-Processing*) [OPE14]. Tais bibliotecas oferecem a possibilidade de troca de mensagens e acesso a memória compartilhada por meio de múltiplas *threads*. No entanto, ambas as bibliotecas requerem que os programadores explicitamente controlem a comunicação e sincronização entre os processos existentes. Além disto, os programadores frequentemente precisam lidar em seus códigos sobre questões relacionadas a concorrência, tais como *livelocks*, *deadlocks* e *race conditions*.

De modo contrário, utilizando-se o modelo de programação MapReduce, o programador não gerencia explicitamente a comunicação entre os processos. Ao invés disto, MapReduce oferece a funcionalidade de definir as tarefas de computação a partir de *jobs* definidos pelo usuário, usando as implementações das funções Map e Reduce. Além disto, MapReduce é tolerante a falhas, pois tem a capacidade de detectar e reagendar *jobs* que tiveram algum problema durante o processamento. De acordo com Hayduk [HAY12], em ambientes que utilizam as bibliotecas MPI e OpenMP, o processo de recuperação de falhas normalmente ocorre a partir da reinicialização do processamento que está sendo realizado, iniciando-se novamente o *job*. Como os *datasets* utilizados em FIM normalmente são grandes, esta operação despende bastante tempo.

Outros trabalhos também mencionam a complexidade de lidar com tais bibliotecas. Para Yahya, Hegazi e Ezat [YAH12], os algoritmos Apriori propostos com computação paralela e distribuída trazem consigo muitos problemas a serem resolvidos, que não existem nos algoritmos sequenciais. Tais problemas são o balanceamento de carga, a partição dos dados e sua distribuição entre os nodos, o monitoramento dos *jobs*, os parâmetros a serem trocados entre os nodos, dentre outras questões. Desta forma, uma considerável quantidade de esforço é necessária para resolver estes problemas. Por outro lado, ainda segundo Yahya, Hegazi e Ezat [YAH12], a implementação Apache Hadoop oferece simplicidade, escalabilidade e segurança para solucionar a maioria destes problemas.

Os autores Lin, Lee e Hsueh [LIN12b] também afirmam que os algoritmos paralelos de mineração de dados que têm sido propostos, apresentam novos problemas a serem resolvidos. Também citam como problemas o balanceamento de carga, a distribuição dos nodos, o gerenciamento dos *jobs* e os parâmetros a serem trocados entre os nodos. Para executar uma tarefa de mineração de dados sobre um *cluster*, o *dataset* deve ser dividido em *splits* e estes distribuídos aos nodos existentes no *cluster*. Além disto o *job* deve decidir em qual nodo executar. Se o balanceamento de carga não for realizado de modo eficiente, o tempo de execução será demorado por causa dos nodos mais lentos. Para Lin, Lee e Hsueh [LIN12b], com o surgimento de *cloud computing*, o modelo de programação MapReduce tornou-se uma das mais importantes técnicas para contornar estes problemas. Ele oculta problemas como a distribuição dos dados, tolerância a falhas e balanceamento de carga, fazendo com que o programador direcione seu foco no algoritmo a ser implementado.

4.3 Trabalhos de FIM e MapReduce sobre *Datasets* com Incerteza Associada

Além dos trabalhos mencionados, que integram FIM e MapReduce sobre dados determinísticos, foi encontrado um trabalho que integra estas duas áreas e as aplica sobre *datasets* com incerteza associada. O trabalho de Leung e Hayduk [LEU13] discute esta integração.

4.3.1 Os Experimentos de Leung e Hayduk

Este trabalho foi o único encontrado até o momento reunindo o modelo de programação MapReduce, algoritmos de descoberta de conjuntos de itens frequentes e *datasets* com incerteza. Seus autores propõem um algoritmo da classe *FP-growth-based* (baseado em árvore), denominado MR-Growth, que utiliza MapReduce para minerar padrões frequentes em grandes *datasets* com incerteza associada. É importante salientar que, devido à presença de probabilidades existenciais nos itens, o espaço de procura/solução para minerar itens frequentes sobre dados com incerteza é muito maior do que quando utilizado dados precisos. Os resultados experimentais deste trabalho mostram grande eficiência deste algoritmo em relação ao algoritmo sequencial UF-Growth.

Inicialmente, o MR-Growth lê o *dataset* e calcula o suporte esperado de cada 1-itemset usando MapReduce, conforme Equação 3.6. O algoritmo divide o *dataset* em diversas partições e as associa aos diferentes nodos do *cluster*. Durante a fase de mapeamento, a função Map recebe transações com o seguinte formato <código da transação, itens> como entrada. Para cada transação t_i, a função Map emite um par $< X, valor >$.

Nos algoritmos que trabalham sobre dados precisos, estes pares emitidos pela função Map são semelhantes a $< X, 1 >$ (conforme algoritmo SPC, discutido previamente), para cada ocorrência de X em t_i. No caso de minerar dados com incerteza, não é certo que X acontece em t_i pois, na verdade, X tem uma probabilidade existencial de ocorrer em t_i. Portanto, ao invés de a função Map do algoritmo MR-Growth emitir pares $< X, 1 >$, ela gera pares $< X, P(X, t_i) >$, para cada ocorrência de $X \in t_i$.

A fim de exemplificar o primeiro passo do algoritmo MR-Growth, tome-se como exemplo a Tabela 4.2, que representa um *dataset* com incerteza, contendo m transações ($m = 2$). Considere-se um mínimo suporte esperado (minsupesp) igual a 0.5. Para a transação t_1 a função Map tem como saída $\{< a, 0.5 >, < b, 0.5 >, < c, 1.0 >, < d, 1.0 >, < u, 0.5 >\}$ e para t_2 a saída é $\{< a, 0.5 >, < b, 0.5 >, < p, 0.5 >\}$. Após a função Map, os pares são integrados e ordenados, $\{< a, [0.5, 0.5] >, < b, [0.5, 0.5] >, < c, [1.0] >, < d, [1.0] >, < p, [0.5] >, < u, [0.5] >\}$, através da função Combine. Ao final da execução, esta função gera os seguintes conjuntos candidatos: $\{< a, 1.0 >, < b, 1.0 >, < c, 1.0 >, < d, 1.0 >, < p, 0.5 >, < u, 0.5 >\}$.

A saída da função Combine é a entrada para a função Reduce. Na função Reduce àqueles candidatos a 1-itemsets que possuem suporte esperado menor do que o mínimo estabelecido ($< minsupesp \cdot m$) são eliminados. Neste caso os elementos do conjunto L_1 são: $\{< a, 1.0 >, < b, 1.0 >, < c, 1.0 >, < d, 1.0 >\}$. Foram excluídos os 1-itemsets $< p, 0.5 >$ e $< u, 0.5 >$, pois seus suportes (0.5) são inferiores a 1 ($minsupesp \cdot m$).

Tabela 4.2: *Dataset* com 2 transações e itens com probabilidades existenciais.

Código da Transação	**Itens**
t_1	a:0.5, b:0.5, c:1.0, d:1.0 u:0.5
t_2	a:0.5, b:0.5, p:0.5

A partir do primeiro passo do algoritmo MR-Growth, há um estágio de agrupamento. Neste estágio os 1-itemsets são divididos em grupos distintos (*G-list*), onde cada grupo recebe um iden-

tificador. No artigo os 1-itemsets frequentes, $L_1 = \{< a, 1.0 >, < b, 1.0 >, < c, 1.0 >, < d, 1.0 >\}$, são divididos em dois grupos (em função de serem usados dois nodos no *cluster*), Grupo 1: $\{a, b\}$ e Grupo 2: $\{c, d\}$. Este estágio de agrupamento identifica as árvores condicionais que devem ser mineradas juntas sobre um único nodo.

Posteriormente, o algoritmo MR-Growth relaciona todas as transações com os grupos criados. Sobre cada nodo, o algoritmo MR-growth carrega a *G-list* para a memória principal, e cria um mapa reverso que mapeia os 1-itemsets frequentes para seus correspondentes IDs de grupo (GID). Para cada transação t_i no dataset, MR-growth substitui todos os itens das transações com seu GID correspondente, a partir do mapa reverso elaborado, criando uma nova lista do mesmo tamanho da transação. A Tabela 4.3 reproduz os itens do dataset (somente os 1-itemsets frequentes), de cada transação, com seus respectivos IDs de grupo (GID). Durante a função Map, as transações são identificadas por GID, conforme Tabela 4.3, e servem de entrada para a função Combine.

Tabela 4.3: Lista de grupos criada, em função da aplicação do algoritmo MR-Growth sobre o *dataset* ilustrado na Tabela 4.2.

Lista de Grupos	**ID do grupo (GID)**
1	<1, 1, 2, 2>
2	<1, 1>

A função Combine, de cada nodo, recebe as transações dependentes dos grupos no formato <key=gid, value=$t_1, t_2, ..., t_n$>, onde t_i representa a i-ésima transação. As transações são utilizadas para construção de uma UF-tree, criando uma árvore comprimida baseada em transações associadas ao grupo, cuja chave é igual ao GID.

A função Reduce, por sua vez, coleta as diversas árvores dependentes de grupo construídas após a aplicação das funções Combine sobre os diversos nodos. Posteriormente combina as árvores em uma única UF-tree gerando os padrões frequentes existentes. Sobre esta UF-tree é aplicado o algoritmo UF-Growth que minera os conjuntos de itens frequentes.

Para Leung e Hayduk [LEU13], a chave para a eficiência da execução do algoritmo desta abordagem é que ele nunca constrói a árvore principal sobre um único nodo. Cada nodo recebe uma partição do *dataset* principal (*split*) e constrói sua árvore localmente. O algoritmo constrói pequenas representações (árvores) disjuntas do *dataset* executando em diversos nodos. Posteriormente, a função Reduce agrupa todas as árvores em uma única UF-tree e aplica o algoritmo UF-Growth sobre esta.

Os experimentos usando o algoritmo MR-Growth foram realizados sobre dois ambientes. O primeiro deles em uma máquina Intel core i7, 4-core-processor, com 8GB de RAM, sobre o Windows7 64bits. O segundo ambiente usou um *cluster* Amazon EC2 com 11 nodos m2.xlarge (arquitetura 64 bits, 2 CPUs virtuais 17,1GB RAM). O algoritmo foi implementado na linguagem de programação Java e a versão do Apache Hadoop usada foi a 0.20.0.

Os experimentos foram realizados sobre *datasets* reais e sintéticos. Os *datasets* reais utilizados foram: accidents, connect4 e mushroom, existentes nos repositórios FIMI e UCI. Foram gerados também três *datasets* sintéticos usando a ferramenta *IBM Synthetic Data Generator* [AGR94]. Estes *datasets* foram construídos com cerca de 2 a 5 milhões de transações, com um tamanho médio de 10 itens por transação, em um domínio de cerca de 1000 itens.

Sobre 11 nodos no *cluster* Amazon EC2, o algoritmo MR-Growth obteve um desempenho bem melhor do que a sua versão sequencial UF-Growth. Com o maior *dataset* (5 milhões de transações), o algoritmo sequencial UF-Growth levou mais do que 120 mil segundos (33.33 horas), enquanto sua versão com MapReduce precisou de menos de 20 mil segundos (5.55 horas). O autor ainda exibe gráficos que, de modo geral, mostram o MR-Growth de 7 a 8.5 vezes mais rápido do que o

UF-Growth. O MR-Growth e o UF-Growth apresentam a mesma acurácia, pois foram gerados os mesmos conjuntos de itens frequentes.

O artigo cita ainda uma melhoria no algoritmo MR-Growth, explorando máquinas que possuem processadores *multi-core*, por meio da utilização do *framework* ForkJoin [LEA00]. Estes experimentos demonstraram o efeito de empregar múltiplas *threads* para minerar conjuntos de itens frequentes. Neste caso os testes foram aplicados sobre a máquina intel core i7, 4-core-processor, variando os números de *threads* de 1 a 4. Para *datasets* onde o tempo de execução do algoritmo é muito curto, o MR-Growth com ForkJoin não levou vantagem por executar em múltiplos *cores*. No entanto, quando o tempo de execução do algoritmo aumenta (mais do que 100 segundos), ocorreu uma melhoria linear no tempo de execução, à medida que foram utilizadas mais *threads*.

4.4 Considerações Finais do Capítulo

O trabalho de Han, Pen e Yin [HAN00] mostra que algoritmos da classe *FP-growth-based* têm melhor desempenho do que algoritmos da classe *Apriori-based*, sobre *datasets* determinísticos. No entanto, os trabalhos de Aggarwal et al. [AGG09b] e Tong et al. [TON12] constatam que os algoritmos de descoberta de itens frequentes comportam-se de modo diferente quando aplicados sobre *datasets* com incerteza. Quando o *dataset* tem probabilidades associadas aos seus itens, os algoritmos *Apriori-based* (UApriori) e *H-mine-based* (UH-Mine) têm melhor desempenho do que os algoritmos da classe *FP-growth-based* (UFP-Growth), que se utilizam de árvores compactadas para armazenamento do *dataset*.

Os trabalhos de Lin, Lee e Hsueh [LIN12b]; Li et al. [LI12]; Yahya, Hegazi e Ezat [YAH12] evoluem o tradicional algoritmo Apriori, aplicando diversas estratégias diferentes, integrando-o com o modelo de programação MapReduce. Nestes trabalhos a integração demonstrou melhoria no desempenho dos algoritmos à medida que o suporte mínimo vai sendo reduzido e, consequentemente, um número maior de itemsets frequentes é descoberto. Há relatos de boa escalabilidade destes algoritmos tanto em *clusters* locais, quanto em *clusters* que utilizam serviços de *cloud computing*. No entanto, todos os trabalhos encontrados não abordam o comportamento desta integração com *datasets* que têm incerteza associada. Estes três artigos, além do trabalho de Hayduk [HAY12] evidenciam também que a utilização do modelo de programação MapReduce diminui consideravelmente a complexidade do desenvolvimento de algoritmos paralelos e distribuídos quando comparado a outras tecnologias tais como MPI e OpenMP.

O único trabalho encontrado sobre FIM, MapReduce e Incerteza, de acordo com a revisão sistemática elaborada por Carvalho e Ruiz [CAR13], é o artigo de Leung e Hayduk [LEU13]. Este artigo implementa um algoritmo FIM da classe *FP-growth-based*, o MR-Growth, utilizando estruturas de árvore para armazenar o *dataset*, integrado com o modelo de programação MapReduce. O MR-Growth também obtém desempenho superior quando comparado ao seu equivalente sequencial, o UFP-Growth. Como este é um trabalho pioneiro relacionando FIM, MapReduce e Incerteza, outras discussões estão abertas e podem ser aprofundadas.

Desta forma, o trabalho de Leung e Hayduk [LEU13] deixa clara a viabilidade de usar MapReduce para minerar conjuntos de itens frequentes sobre dados incertos. No entanto, conforme mostrado por Aggarwal et al. [AGG09b] e Tong et al. [TON12] os algoritmos da classe *Apriori-based* e *H-mine-based* têm melhor comportamento sobre *datasets* com incerteza associada em relação aos algoritmos da classe *FP-growth-based*. Logo, abre-se um grande espaço de pesquisa para evoluir algoritmos das classes *Apriori-based* e *H-mine-based* com o modelo de programação MapReduce e avaliar seu desempenho sobre *datasets* com incerteza. Nesta tese serão explorados algoritmos da classe *Apriori-based*.

5. ALGORITMO UAPRIORIMRJOIN

Este capítulo formaliza o algoritmo UAprioriMRJoin, principal contribuição desta tese. Inicialmente é formalizado e apresentado um exemplo para melhor compreensão do algoritmo UAprioriMR, ideia tradicional do algoritmo UApriori, implementada sob o modelo de programação MapReduce (semelhante ao algoritmo SPC, desenvolvido por [LIN12b]). Em seguida, também é formalizado e exemplificado o algoritmo UAprioriMRByT, que altera o modo de geração de itemsets dos conjuntos candidatos, a cada passo $k > 1$. Posteriormente é realizada uma discussão sobre os benefícios de utilização do algoritmo UAprioriMRByT em relação ao algoritmo UAprioriMR, quando analisado o passo $k = 2$ do algoritmo. O capítulo termina com a formalização do algoritmo híbrido UAprioriMRJoin, a partir da composição das ideias dos algoritmos UAprioriMR e UAprioriMRByT.

5.1 O Algoritmo UAprioriMR

Os trabalhos de Li et al. [LI08]; Li et al. [LI12]; Lin, Lee e Hsueh [LIN12b]; Yahya, Hegazi e Ezat [YAH12]; e Kulkarni, Jagale e Rokade [KUL13] integram o modelo de programação MapReduce para descobrir conjuntos de itens frequentes em dados determinísticos. O algoritmo UAprioriMR, detalhado ao longo desta seção, baseou-se nestes trabalhos para também descobrir conjuntos de itens frequentes, mas sobre *datasets* com incerteza.

O algoritmo UAprioriMR foi implementado utilizando o modelo de programação MapReduce, sob a implementação Apache Hadoop. Ele foi desenvolvido com as tarefas Map, Combine e Reduce, que são coordenadas por meio do *job* MapReduce. Este *job* divide o *dataset* de entrada em diversos pedaços (*splits*), onde cada *split* é associado a um nodo do *cluster*. Cada *split* é processado independentemente por uma tarefa Map. Esta tarefa varre o seu *split* específico, criando como saída registros de pares $(chave\ valor)$, onde cada $chave$ é um itemset e o $valor$ é igual a probabilidade daquele itemset em uma transação específica.

Esta estratégia de divisão do *dataset* em pequenas porções (*splits*) auxilia a melhorar o desempenho de algoritmos baseados no Apriori tradicional. Isto porque este algoritmo tem como gargalo, durante sua execução, as diversas varreduras realizadas sobre o *dataset*. Dado que estas varreduras são realizadas em nodos diferentes, paralelizando o processamento sobre *datasets* menores (*splits*), o tempo de execução do algoritmo é reduzido.

Antes de entrar na tarefa Reduce, a saída da tarefa Map é exportada para uma tarefa denominada Combine. Esta última é responsável pelo cálculo do suporte esperado de cada itemset, $SupEsp(c)$, conforme a Equação 3.6. Posteriormente, a tarefa Combine entrega à tarefa Reduce pares $(chave\ valor)$, onde cada $chave$ representa um itemset c qualquer e o $valor$ significa o resultado do suporte esperado para o itemset c.

A tarefa Reduce captura a saída da tarefa Combine, agrupa os itemsets iguais, e verifica se cada itemset respeita o limiar de mínimo suporte esperado. Em caso positivo, o itemset e seu suporte esperado são escritos em um arquivo de saída, onde cada linha deste arquivo tem um itemset e o seu suporte esperado, no formato $(chave\ valor)$. Se $SupEsp(c)$ for inferior ao suporte mínimo esperado, o itemset c e seu $valor$ são descartados. Ao final, o arquivo de saída da tarefa Reduce lista o conjunto de itens frequentes de tamanho 1, os 1-itemsets.

O conjunto de itens frequentes 1-itemsets, L_1, permite que seja criado o conjunto de itens candidatos para a próxima iteração do algoritmo, ou seja, L_1 alimenta a entrada para a tarefa Map da segunda iteração do algoritmo. Ao final da iteração 2, após as tarefas Combine e Reduce serem aplicadas, um arquivo com o conjunto de itens frequentes de tamanho 2 (2-itemsets) é criado.

Este procedimento cíclico ocorre nas k próximas iterações até que a última iteração não gere um arquivo de saída com os k-itemsets. Desta forma, não havendo mais itens candidatos para a próxima iteração, o algoritmo termina. A Figura 5.1 ilustra de modo geral o algoritmo citado.

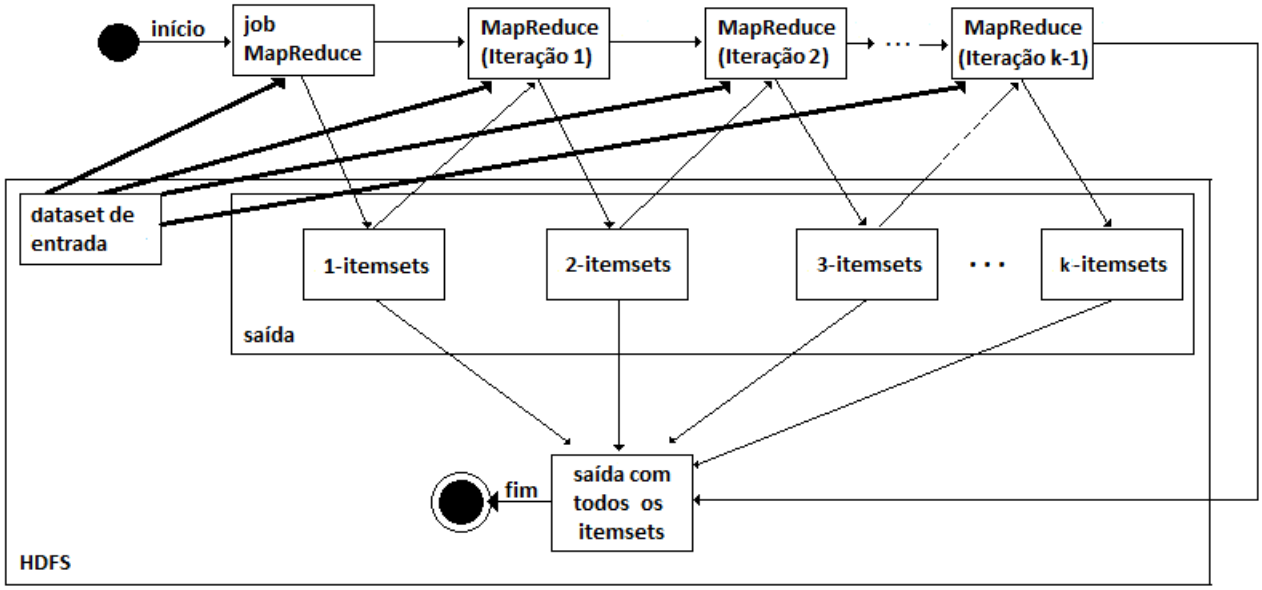

Figura 5.1: Arquitetura do algoritmo UAprioriMR na geração de itens frequentes usando MapReduce (adaptado de [LI12]).

5.1.1 Formalização do Algoritmo UAprioriMR

O algoritmo UAprioriMR executa em k passos. No passo $k = 1$ são descobertos os 1-itemsets. No passo seguinte os 2-itemsets são retornados. Este processo continua até que no passo $(k-1)$ não sejam gerados (k-1)-itemsets. A cada passo k do algoritmo, ele executa três funções: Map, Combine e Reduce. O passo $k = 1$ é diferente dos demais passos, portanto, é explicado à parte.

Passo $k = 1$

Dado um conjunto I contendo x itens, $I : \{i_1, i_2, ..., i_x\}$. Tome-se um *cluster* Q com n nodos. Considere-se um *dataset* D formado por m transações $\{t_1, t_2, ..., t_m\}$, onde cada transação t_j é formada por y pares $(i\ p)$, onde i representa um item qualquer $i \in I$ e p denota a probabilidade existencial de i em t_j. Portanto, t_j: $\{(i_1\ p_1), (i_2\ p_2), ..., (i_y\ p_y)\}$. A função $T(t_j)$ retorna o conjunto de itens i existentes na transação t_j, ou seja, $T(t_j) = \{i_1,\ i_2, ...,\ i_y\}$. A função $P(i, t_j)$ retorna a probabilidade existencial do item i na transação t_j. Um itemset c é um conjunto de itens i. Um itemset c com 1 item i qualquer é denominado 1-itemset, um itemset c com 2 itens i quaisquer é um 2-itemset, um itemset c com k itens quaisquer é um k-itemset. C_k é um conjunto de itemsets c de tamanho k.

O modelo de programação MapReduce divide o *dataset* D em n *splits* disjuntos: $S_1, S_2, ..., S_n$. Cada *split* S_q é enviado a um nodo q do *cluster* Q. Em cada nodo q roda uma função Map_q, que faz uma varredura sobre o *split* S_q. O código que implementa a função Map_q é representado pelo Algoritmo 5.1.

A função Map_q tem como saída o conjunto $O_{q,k}$ (linha 7), onde q significa o nodo do *split* onde a função Map está rodando e k denota o passo do algoritmo. O conjunto $O_{q,k}$ armazena todos os pares $(c\ p)$ do *split* S_q, conforme ilustrado na linha 4 do Algoritmo 5.1. Portanto, no passo $k = 1$,

Algoritmo 5.1: Função Map_q do Algoritmo UAprioriMR no Passo $k = 1$.

Entrada: S_q, *split* q onde cada linha é uma transação.
Saída: conjunto $O_{q,k}$ referente ao *split* S_q, contendo pares $(c\ p)$, onde c é um 1-itemset e p é a probabilidade existencial de c em uma transação t_j.

$Map_q(S_q)$
1: $O_{q,k} \leftarrow \emptyset$
2: **for all** (transação $t_j \in S_q$) **do**
3: **for all** (par $(c\ p) \in t_j$) **do**
4: $O_{q,k} \leftarrow O_{q,k} \cup (c\ p)$
5: **end for**
6: **end for**
7: return $O_{q,k}$

para cada transação t_j existente em S_q (linha 2), os pares $(c\ p)$ de t_j (linha 3) são inseridos dentro do conjunto $O_{q,k}$.

A saída da função Map_q, ou seja, o conjunto $O_{q,k}$ (linha 7), é associada à entrada da função $Combine_q$, a qual tem o código descrito no Algoritmo 5.2. Esta função, que roda no nodo q do *cluster* Q, agrupa a soma das probabilidades existenciais (p) de cada itemset c em $O_{q,k}$, exibido nas linhas 3 à 6 do Algoritmo 5.2. O conjunto A armazena cada itemset c encontrado em cada par $(c\ p)$ existente dentro do conjunto $O_{q,k}$ (linha 4). O vetor Sum, por sua vez, acumula as probabilidades existenciais p de cada itemset c (linha 5). A saída da função $Combine_q$ é o conjunto $O'_{q,k}$, o qual contém os itemsets, sem repetição, e para cada itemset, a soma de suas probabilidades existenciais, denominado suporte esperado do itemset ($supesp$), indicado na linha 10 do Algoritmo 5.2.

Algoritmo 5.2: Função $Combine_q$ do Algoritmo UAprioriMR.

Entrada: conjunto $O_{q,k}$ contendo pares $(c\ p)$, onde c é um itemset e p é a probabilidade existencial de c em uma transação t_j.
Saída: conjunto $O'_{q,k}$ contendo pares $(c\ supesp)$, onde c é um itemset e $supesp$ (suporte esperado) é a soma das probabilidades de c.

$Combine_q(O_{q,k})$
1: $O'_{q,k} \leftarrow \emptyset$
2: $A \leftarrow \emptyset$
3: **for all** (par $(c\ p) \in O_{q,k}$) **do**
4: $A \leftarrow A \cup c$
5: $Sum[c] \leftarrow Sum[c] + p$
6: **end for**
7: **for all** ($c \in A$) **do**
8: $O'_{q,k} \leftarrow O'_{q,k} \cup (c\ Sum[c])$
9: **end for**
10: return $O'_{q,k}$

A função Reduce tem o código semelhante à função $Combine_q$. Ela agrupa todas as saídas das funções $Combine_q$ dos diversos nodos existentes. Posteriormente, ela verifica quais são os itemsets que possuem suporte esperado maior do que o mínimo suporte esperado, gerando, desta forma, os 1-itemsets frequentes, conforme exibido nas linhas de 8 à 10, no código ilustrado pelo Algoritmo 5.3. Os 1-itemsets frequentes compõem o conjunto L_1, o qual é retornado pela função Reduce e inserido

no mecanismo *DistributedCache* do Apache Hadoop. O *DistributedCache* permite que os arquivos sejam copiados para os nodos onde as tarefas estão executando, os quais podem ser gravados no próprio sistema de arquivos local do nodo ou no HDFS.

Algoritmo 5.3: Função $Reduce$ do Algoritmo UAprioriMR.

Entrada: $O'_k = O'_{1,k} \cup O'_{2,k} ... \cup O'_{q,k} ... \cup O'_{n,k}$ contendo pares $(c\ supesp)$, onde c é um itemset e $supesp$ é o suporte esperado do itemset c.
Entrada: $minsupesp$, mínimo suporte esperado.
Saída: conjunto L_k contendo os itemsets frequentes de tamanho k, e seu suporte esperado.
$Reduce(O'_k, minsupesp)$
1: $L_k \leftarrow \emptyset$
2: $A \leftarrow \emptyset$
3: **for all** (par $(c\ supesp) \in O'_k$) **do**
4: $A \leftarrow A \cup c$
5: $Sum[c] \leftarrow Sum[c] + supesp$
6: **end for**
7: **for all** $(c \in A)$ **do**
8: **if** $(Sum[c] > (minsupesp \cdot |D|))$ **then**
9: $L_k \leftarrow L_k \cup (c\ Sum[c])$
10: **end if**
11: **end for**
12: return L_k

Passo $k > 1$

A função Map_q, nos passos posteriores do algoritmo UAprioriMR, funciona de modo distinto ao primeiro passo. O código do Algoritmo 5.4 apresenta a função Map_q para os passos posteriores ao primeiro. Isto ocorre, por exemplo, porque a geração dos 2-itemsets frequentes deve levar em consideração quais 1-itemsets frequentes foram produzidos. Desta forma, há uma função denominada $gera\_candidatos(L_{k-1})$ para criar o conjunto de itemsets candidatos de tamanho k, denominado C_k (linha 1), a partir do conjunto de itemsets frequentes do passo anterior L_{k-1}. Cada itemset c existente em C_k é constituído de k elementos i.

Após a criação de C_k, aplica-se ainda a função de poda do tradicional Apriori sobre este conjunto. A função $poda\_candidatos(C_k, L_{k-1})$, descrita na linha 2, é responsável por esta tarefa, com o objetivo de reduzir o número de itemsets $c \in C_k$. Ela elimina os k-itemsets que contenham algum (k-1)-itemset infrequente.

No passo $k = 2$, portanto, são gerados os 2-itemsets candidatos em C_2, a partir dos itens frequentes descobertos no passo $k = 1$, os 1-itemsets frequentes. Portanto, no passo k são gerados os k-itemsets, a partir dos $(k-1)$-itemsets frequentes. Para cada transação t_j, existente no *split* S_q (linha 4), é avaliado se cada itemset c do conjunto C_k (linha 5) está contido em t_j (linha 6). Caso este itemset c não exista na transação t_j ele é desconsiderado, ao contrário, caso ele esteja presente, é necessário calcular o suporte esperado do itemset c na transação t_j, aplicando-se a Equação 3.6 e armazenando este valor no vetor $Multi$, conforme linhas de 7 à 10 no Algoritmo 5.4. Esta avaliação é realizada sobre todas as transações do *split* S_q.

As saídas das funções Map_q (linha 15) são as entradas das funções $Combine_q$. As funções $Combine_q$ fazem a soma de todas as saídas das funções Map_q, agrupando-as por itemset ($chave$). Posteriormente, as saídas das funções $Combine_q$ servem de entradas para a função Reduce, que

Algoritmo 5.4: Função Map_q do Algoritmo UAprioriMR no Passo $k > 1$.

Entrada: S_q, *split* q, onde cada linha é uma transação.
Entrada: conjunto L_{k-1} contendo pares $(i\ p)$ de itemsets frequentes encontrados no passo anterior (k-1).
Saída: conjunto $O_{q,k}$ referente ao *split* S_q, contendo pares $(i\ p)$.

$Map_q(S_q, L_{k-1})$
1: $C_k \leftarrow gera\_candidatos(L_{k-1})$
2: $C_k \leftarrow poda\_candidatos(C_k, L_{k-1})$
3: $O_{q,k} \leftarrow \emptyset$
4: **for all** (transação $t_j \in S_q$) **do**
5: **for all** $(c \in C_k)$ **do**
6: **if** $(c \subseteq T(t_j))$ **then**
7: **for all** $(i \in c)$ **do**
8: $p \leftarrow P(i, t_j)$
9: $Multi[c] \leftarrow Multi[c] \cdot p$
10: **end for**
11: $O_{q,k} \leftarrow O_{q,k} \cup (c\ Multi[c])$
12: **end if**
13: **end for**
14: **end for**
15: return $O_{q,k}$

também faz um somatório do produto das probabilidades existenciais de cada item pertencente ao itemset (vide Equação 3.6) e seleciona os itemsets frequentes. As funções $Combine_q$ e Reduce dos passos $k > 1$ são iguais às respectivas funções do passo $k = 1$, já descritas nos algoritmos 5.2 e 5.3.

Ao final da execução de todos os k passos do algoritmo UAprioriMR, a função Reduce gera $k-1$ arquivos de saída. A união destes arquivos de saída representa o conjunto L, que mantém todos os itemsets frequentes de tamanhos $1, 2, ..., k-1$. Portanto,

$$L = \{\ Arquivo\_Saida\_Reduce_1 \cup Arquivo\_Saida\_Reduce_2 \cup ... \cup Arquivo\_Saida\_Reduce_{k-1}\ \}$$

5.1.2 Exemplificando o Algoritmo UAprioriMR

De acordo com Chui et al. [CHU07], um itemset c qualquer é frequente, se e somente se, o seu suporte esperado não é menor do que $minsupesp \cdot m$, onde $minsupesp$ é o suporte mínimo esperado e m é o número de transações do *dataset*. Tome-se um *dataset* com 10 transações $(m = 10)$, conforme ilustrado na Figura 5.2, onde cada transação denota a probabilidade de um paciente estar com sintomas de Depressão (D), Hipertensão (H), Insônia (I) e Obesidade (O). Por exemplo, o paciente da transação 3 tem Depressão e Obesidade, além de contar com probabilidade de 90% de estar sofrendo de Insônia.

Considere-se um suporte mínimo $(minsupesp)$ igual a 0.20 e o *dataset* da Figura 5.2. Os conjuntos de itens frequentes são aqueles itemsets com suporte esperado igual ou superior a 2.0 $(minsupesp \cdot m)$. Dado um *cluster* C composto por dois nodos $(n = 2)$ e o *dataset* dividido em dois *splits* de mesmo tamanho e disjuntos: S_1 e S_2. O *split* 1 (S_1) é direcionado para o nodo 1 e o *split* 2 (S_2) é transferido para o nodo 2, conforme ilustrado na Figura 5.3.

#	
1	(H 0.6)
2	(D 0.9) (O 0.9)
3	(D 1.0) (I 0.9) (O 1.0)
4	(D 0.8) (I 0.9) (O 1.0)
5	(D 1.0) (O 0.9)
6	(H 0.8) (I 0.5)
7	(I 0.6)
8	(D 0.6)
9	(D 0.8) (I 0.7) (O 0.9)
10	(H 0.7) (I 0.8)

Figura 5.2: *Dataset* onde cada transação denota a probabilidade de um paciente estar com sintomas de Depressão, Hipertensão, Insônia e Obesidade.

	NODO 1		NODO 2
#	SPLIT 1 (S_1)	#	SPLIT 2 (S_2)
1	(H 0.6)	6	(H 0.8) (I 0.5)
2	(D 0.9) (O 0.9)	7	(I 0.6)
3	(D 1.0) (I 0.9) (O 1.0)	8	(D 0.6)
4	(D 0.8) (I 0.9) (O 1.0)	9	(D 0.8) (I 0.7) (O 0.9)
5	(D 1.0) (O 0.9)	10	(H 0.7) (I 0.8)

Figura 5.3: *Dataset* original dividido em dois *splits* disjuntos: S_1 e S_2.

Passo $k = 1$

Em cada nodo, no passo $k = 1$, é executada uma tarefa Map_q, responsável por gerar um conjunto $O_{q,k}$ com uma lista de pares $(chave\ valor)$, onde a $chave$ é um 1-itemset e o $valor$ é a probabilidade existencial do 1-itemset em uma transação. Esta lista é armazenada em dois arquivos, cada um deles salvo no nodo local, conforme ilustrado na Figura 5.4.

#	Arquivo_Map_Split_1	#	Arquivo_Map_Split_2
1	(H 0.6)	1	(H 0.8)
2	(D 0.9)	2	(I 0.5)
3	(O 0.9)	3	(I 0.6)
4	(D 1.0)	4	(D 0.6)
5	(I 0.9)	5	(D 0.8)
6	(O 1.0)	6	(I 0.7)
7	(D 0.8)	7	(O 0.9)
8	(I 0.9)	8	(H 0.7)
9	(O 1.0)	9	(I 0.8)
10	(D 1.0)		
11	(O 0.9)		

Figura 5.4: Arquivos gerados após a execução da função Map_q em cada *split* q.

As saídas das tarefas Map_q, "Arquivo_Map_Split_1" ($O_{1,1}$) e "Arquivo_Map_Split_2" ($O_{2,1}$), são as entradas para cada tarefa $Combine_q$. Nesta tarefa os suportes esperados de cada 1-itemset são somados, para cada um dos *splits*, criando ao final da execução os arquivos semelhantes aos exibidos na Figura 5.5, ou seja, os conjuntos $O'_{q,k}$. O arquivo "Arquivo_Combine_Split_1" é o conjunto $O'_{1,1}$ e o arquivo "Arquivo_Combine_Split_2" é o conjunto $O'_{2,1}$.

A fim de exemplificar a saída das tarefas $Combine_q$, observa-se a linha correspondente ao itemset D, no arquivo "Arquivo_Combine_Split_1" ($O'_{1,1}$): $(D\ 3.7)$. Este par foi gerado a partir da aplicação da Equação 3.6: $0.9 + 1.0 + 0.8 + 1.0 = 3.7$. Ou seja, foi somada a probabilidade do itemset D existir, em cada transação do *split* 1 (transações $t_2,\ t_3,\ t_4\ e\ t_5$). Quanto ao itemset

#	Arquivo_Combine_Split_1	#	Arquivo_Combine_Split_2
1	(D 3.7)	1	(D 1.4)
2	(H 0.6)	2	(H 1.5)
3	(I 1.8)	3	(I 2.6)
4	(O 3.8)	4	(O 0.9)

Figura 5.5: Arquivos gerados após a execução da função $Combine_q$ em cada *split* q.

D, no arquivo "Arquivo_Combine_Split_2" ($O'_{2,1}$), é produzido o par $(D\ 1.4)$, também aplicando a soma das probabilidades do itemset D existir, em cada transação do *split* 2 (transações t_8 *e* t_9). As demais linhas dos arquivos "Arquivo_Combine_Split_1" e "Arquivo_Combine_Split_2" foram obtidas seguindo o mesmo procedimento.

Os arquivos gerados pelas tarefas $Combine_q$ são usados como entrada para a tarefa Reduce. A função Reduce soma os valores para cada 1-itemset, agrupando os valores em um único arquivo, de acordo com a Figura 5.6. Neste caso, o arquivo "Arquivo_Reduce" é o conjunto L_1, o qual é inserido posteriormente no *DistributedCache*. O par $(D\ 5.1)$, por exemplo, é obtido a partir da soma dos valores existentes para o itemset D nos arquivos "Arquivo_Combine_Split_1" e "Arquivo_Combine_Split_2" respectivamente, $(D\ 3.7)$ e $(D\ 1.4)$.

Na função Reduce são eliminadas aquelas linhas que não se adequam ao limiar de suporte mínimo esperado. Assim, os itemsets gerados são considerados frequentes caso seu suporte não seja inferior a 2.0 ($minsupesp \cdot m$). Neste exemplo, todos os 1-itemsets são frequentes, portanto, $L_1 = \{(D\ 5.1), (H\ 2.1), (I\ 4.4), (O\ 4.7)\}$.

#	Arquivo_Reduce
1	(D 5.1)
2	(H 2.1)
3	(I 4.4)
4	(O 4.7)

Figura 5.6: Arquivo consolidado após a execução da função Reduce.

Passo $k = 2$

O arquivo com os 1-itemsets frequentes (L_1) é então utilizado pela segunda iteração do algoritmo UAprioriMR. As tarefas Map_q, de cada nodo, geram os 2-itemsets candidatos, C_2. Cada função Map_q inicia gerando todos os itemsets do conjunto candidato de tamanho $k = 2$, a partir do conjunto L_1. Como L_1 tem 4 itemsets frequentes, o número máximo de itemsets do conjunto candidato de tamanho 2 (C_2), é dado pela equação de combinação 5.1: $C_{4,2} = \frac{4!}{(4-2)!\ \cdot\ 2!} = 6$. Os seis itemsets de C_2 são $\{(DH), (DI), (DO), (HI), (HO), (IO)\}$.

$$C_{n,p} = \frac{n!}{(n-p)!\ \cdot\ p!} \tag{5.1}$$

Após a geração de C_2 é realizada a poda sobre este conjunto. Só permanecem no conjunto C_2 aqueles itemsets onde todos os seus subconjuntos de tamanho 1 estão presentes no conjunto frequente L_1. Neste caso, os seis itemsets de C_2 continuam presentes no conjunto.

De posse do conjunto de itemsets candidatos C_2, cada transação t_j é avaliada perante todo o conjunto C_2, a fim de verificar se os itemsets de C_2 estão presentes na transação t_j e computar o suporte esperado de cada itemset $c \in C_2$. Ou seja, os 6 elementos de C_2 são comparados com cada uma das 10 transações existentes no *dataset*, fazendo ao final 60 comparações.

Ao final de todas as 60 comparações, a função Map_q gera um arquivo com pares $(chave\ valor)$. Neste arquivo cada $chave$ refere-se a um 2-itemset possível e $valor$ é o resultado da multiplicação das probabilidades existenciais de cada item que pertence ao 2-itemset candidato, de cada transação avaliada. A Figura 5.7 ilustra os dois arquivos resultantes de cada tarefa Map_q aplicada sobre cada um dos *splits* S_1 e S_2.

#	Arquivo_Map_Split_1	#	Arquivo_Map_Split_2
1	(DO 0.81)	1	(HI 0.40)
2	(DI 0.90)	2	(DI 0.56)
3	(DO 1.0)	3	(DO 0.72)
4	(IO 0.90)	4	(IO 0.63)
5	(DI 0.72)	5	(HI 0.56)
6	(DO 0.80)		
7	(IO 0.90)		
8	(DO 0.90)		

Figura 5.7: Arquivos gerados após a execução da função Map_q, na segunda iteração do algoritmo UAprioriMR.

Considere-se a saída de cada tarefa Map_q, o conjunto $O_{1,2}$ ("Arquivo_Map_Split_1"), quando esta tarefa é aplicada sobre o *split* 1 (S_1). O primeiro elemento do conjunto $O_{1,2}$ é o par $(DO\ 0.81)$. A probabilidade existencial do 2-itemset DO, exibida na linha 1 do arquivo "Arquivo_Map_Split_1", é o resultado da multiplicação entre as probabilidades existenciais dos itens D e O, que pertencem à segunda transação do *split* S_1 (Figura 5.3). Na linha 3 do arquivo "Arquivo_Map_Split_1" o 2-itemset DO tem probabilidade existencial igual a 1.0, pois na transação 3 do *split* S_1, o item D tem probabilidade de 1.0 e o item O tem probabilidade igual a 1.0. As probabilidades existenciais dos demais 2-itemsets são calculadas seguindo este mesmo procedimento (Equação 3.6).

Cabe salientar que na Figura 5.7 não aparecem em $O_{1,2}$ alguns itemsets do conjunto de candidatos C_2. O itemset DH, por exemplo, de C_2, não é exibido em $O_{1,2}$. Isto acontece porque os itens D e H, que compõem o elemento DH, não estão presentes juntos em qualquer transação do *split* S_1. O mesmo ocorre com os itemsets HI e HO, que também não estão presentes em $O_{1,2}$.

Embora o elemento DH não exista em nenhuma transação, é necessário fazer a comparação dele com todas as transações para verificar a sua existência e, caso exista, computar seu suporte esperado. Neste exemplo, é necessário que os 6 itemsets de C_2 sejam comparados com cada uma das 5 transações de cada *split*, gerando 30 comparações no *split* S_1 e mais 30 comparações em S_2.

Cada tarefa $Combine_q$, por sua vez, receberá o arquivo gerado pelas funções Map_q, os conjuntos $O_{q,k}$. A partir de $O_{q,k}$ estas tarefas geram os arquivos exibidos na Figura 5.8 (conjuntos $O'_{q,k}$). Ao final, cada 2-itemset distinto tem o seu suporte esperado calculado por *split*. Por exemplo, o resultado da Equação 3.6, para o 2-itemset DO, sobre todas as transações do *split* S_1 é $(0.81 + 1.0 + 0.80 + 0.90) = 3.51$. Já sobre as transações do *split* S_2 é $(0.72) = 0.72$. As demais linhas dos arquivos "Arquivo_Combine_Split_1" e "Arquivo_Combine_Split_2" são obtidas seguindo o mesmo raciocínio.

#	Arquivo_Combine_Split_1	#	Arquivo_Combine_Split_2
1	(DI 1.62)	1	(DI 0.56)
2	(DO 3.51)	2	(DO 0.72)
3	(IO 1.80)	3	(HI 0.96)
		4	(IO 0.63)

Figura 5.8: Arquivos gerados após a execução da função $Combine_q$, na segunda iteração do algoritmo UAprioriMR.

A tarefa Reduce, consequentemente, recebe os arquivos exibidos na Figura 5.8, conjuntos $O'_{q,k}$,

e gera o arquivo de saída exibido na Figura 5.9. Na saída desta função estão presentes os 2-itemsets frequentes (L_2), isto é, com suporte esperado maior do que o mínimo especificado (2.0).

Desta forma, é excluído o 2-itemset: HI, pois ele tem suporte esperado igual a 0.96, inferior a 2.0. O 2-itemset DH é eliminado, pois seus itens D e H não aparecem juntos em nenhuma das 10 transações. Pelo mesmo motivo de DH, o 2-itemset HO é descartado também. Logo, o conjunto frequente de tamanho 2 é igual a $L_2 = \{(DI\ 2.18), (DO\ 4.23), (IO\ 2.43)\}$.

#	Arquivo_Reduce
1	(DI 2.18)
2	(DO 4.23)
3	~~(HI 0.96)~~
4	(IO 2.43)

Figura 5.9: Arquivo gerado após a execução da função Reduce, na segunda iteração do algoritmo UAprioriMR.

Passo $k = 3$

A tarefa Map_q recebe a saída da tarefa Reduce, e gera os elementos 3-itemsets. O único 3-itemset gerado é o elemento DIO. A tarefa Map_q gera novamente dois arquivos com os 3-itemsets para cada *split*, Figura 5.10, ou seja, os dois conjuntos $O_{q,k}$. Cada conjunto $O_{q,k}$ serve de entrada para uma função $Combine_q$, a qual soma as probabilidades existenciais dos 3-itemsets existentes, gerando os conjuntos $O'_{q,k}$ (arquivos "Arquivo_Combine_Split_1" e "Arquivo_Combine_Split_2"), conforme a Figura 5.11. Por fim, a função Reduce exibe o suporte esperado do 3-itemset DIO no arquivo de saída ("Arquivo_Reduce"), e como este suporte é maior do que o $minsupesp$, o conjunto $L_3 = \{(DIO\ 2.124)\}$ é gerado, conforme demonstrado na Figura 5.12.

#	Arquivo_Map_Split_1	#	Arquivo_Map_Split_2
1	(DIO 0.90)	1	(DIO 0.504)
2	(DIO 0.72)		

Figura 5.10: Arquivos gerados após a execução da função Map_q, na terceira iteração do algoritmo UAprioriMR.

#	Arquivo_Combine_Split_1	#	Arquivo_Combine_Split_2
1	(DIO 1.62)	1	(DIO 0.504)

Figura 5.11: Arquivos gerados após a execução da função $Combine_q$, na terceira iteração do algoritmo UAprioriMR.

#	Arquivo_Reduce
1	(DIO 2.124)

Figura 5.12: Arquivo gerado após a execução da função Reduce, na terceira iteração do algoritmo UAprioriMR.

O algoritmo UAprioriMR finaliza neste momento porque o número de itens distintos é 3, logo, o número máximo de passos do algoritmo é 3. O conjunto L é composto pela união das saídas de cada função Reduce $(L_1 \cup L_2 \cup L_3)$. Portanto, neste exemplo, todos os itemsets frequentes são: $L = \{(D\ 5.1), (H\ 2.1), (I\ 4.4), (O\ 4.7)\ \cup\ (DI\ 2.18), (DO\ 4.23), (IO\ 2.43)\ \cup\ (DIO\ 2.124)\}$.

5.2 O Algoritmo UAprioriMRByT

O algoritmo UAprioriMRByT utiliza uma abordagem distinta do algoritmo UAprioriMR na geração dos itemsets dos conjuntos candidatos, para aqueles passos onde $k > 1$. O algoritmo UAprioriMR gera os itemsets do conjunto candidato de passo k, baseado no conjunto de itens frequentes gerados no passo anterior (L_{k-1}). Enquanto isso, o algoritmo UAprioriMRByT gera os itemsets do conjunto candidato à medida que cada transação vai sendo varrida pela função Map_q.

O algoritmo UAprioriMRByT funciona da mesma forma que o algoritmo UAprioriMR para o passo $k = 1$. Ou seja, as funções Map, Combine e Reduce são iguais àquelas exibidas nos algoritmos 5.1, 5.2 e 5.3, respectivamente. Quando o passo $k > 1$, a função Map_q do algoritmo UAprioriMRByT comporta-se de modo diferente da função Map_q do algoritmo UAprioriMR. As funções Combine e Reduce do algoritmo UAprioriMRByT, para os passos $k > 1$, são semelhantes às funções do algoritmo UAprioriMR. Por este motivo, esta seção se concentra em explicar a função Map_q do algoritmo UAprioriMRByT.

A função Map_q do algoritmo UAprioriMRByT, quando $k > 1$, lê cada linha do *split* associado ao nodo onde a função está executando, linha 2 do Algoritmo 5.5. Para cada linha lida são gerados os itemsets do conjunto candidato de tamanho k (linha 3). Após a geração destes itemsets, eles são podados aplicando-se a propriedade do algoritmo Apriori (linha 4). Ou seja, só permanecem como itemsets àqueles onde todos os subconjuntos de tamanho $k-1$ são gerados como itens frequentes no passo anterior. Para cada transação lida, os itemsets do conjunto candidato são gerados no formato $(chave\ valor)$, onde cada $chave$ é um k-itemset e o $valor$ é igual a probabilidade daquele k-itemset. O Algoritmo 5.5 ilustra o funcionamento da função Map_q do algoritmo UAprioriMRByT, quando $k > 1$.

Algoritmo 5.5: Função Map_q do Algoritmo UAprioriMRByT no Passo $k > 1$.

Entrada: S_q, *split* q, onde cada linha é uma transação.
Entrada: conjunto L_{k-1} contendo pares $(c\ p)$ de itemsets frequentes encontrados no passo anterior $k-1$.
Saída: conjunto $O_{q,k}$ referente ao *split* S_q, contendo pares $(c\ p)$.

$Map_q(S_q, L_{k-1})$
1: $O_{q,k} \leftarrow \emptyset$
2: **for all** (transação $t_j \in S_q$) **do**
3: $C_k \leftarrow gera\_candidatos(t_j)$
4: $C_k \leftarrow poda\_candidatos(C_k, L_{k-1})$
5: **for all** $(c \in C_k)$ **do**
6: **if** $(c \subseteq T(t_j))$ **then**
7: **for all** $(i \in c)$ **do**
8: $p \leftarrow P(i, t_j)$
9: $Multi[c] \leftarrow Multi[c] \cdot p$
10: **end for**
11: $O_{q,k} \leftarrow O_{q,k} \cup (c\ Multi[c])$
12: **end if**
13: **end for**
14: $limpa\_candidatos(C_k)$
15: **end for**
16: return $O_{q,k}$

No Algoritmo 5.5, para cada transação t_j do *split*, é gerado o conjunto de itemsets candidatos

C_k (linha 3). Após a geração dos itemsets candidatos, o conjunto C_k é podado a partir do conjunto frequente do passo anterior (L_{k-1}) (linha 4). De modo distinto, no algoritmo UAprioriMR, o conjunto C_k não é gerado para cada transação lida. Ele é gerado uma única vez, para cada passo k, e também podado usando o conjunto frequente do passo anterior (L_{k-1}).

Observe a estratégia de geração de conjuntos candidatos usada no algoritmo UAprioriMR para o passo $k > 1$. Quando o conjunto L_{k-1} é muito grande, o número de itemsets do conjunto candidato C_k é grande também, pois o número de itemsets de C_k é calculado pela Equação 5.1. Como toda transação t_j precisa ser comparada com todos os itemsets $c \in C_k$, a fim de calcular o suporte esperado de cada elemento c, a descoberta de conjuntos frequentes no passo k tende a ficar lenta, por causa do alto número de comparações a serem realizadas.

Já a estratégia de geração de candidatos por transação, proposta no algoritmo UAprioriMRByT, implementa a geração do conjunto de itemsets candidatos (C_k) por transação, e não mais a partir de L_{k-1}. Assim, quando o *split* tiver uma média baixa de itens por transação e $|L_{k-1}|$ for muito grande, a geração de itemsets em C_k por transação, tende a ser mais rápida do que produzir todos os itemsets a partir do conjunto de itemsets frequentes do passo anterior.

Após a geração dos conjuntos candidatos (C_k), para cada transação t_j, o suporte esperado de cada itemset $c \in C_k$ e contido na transação t_j é calculado, conforme pode ser visualizado nas linhas de 5 à 13 do Algoritmo 5.5. Os pares de itemset c e seu suporte esperado são inseridos no conjunto $O_{q,k}$ (linha 11). Após, o algoritmo esvazia o conjunto C_k, por intermédio da função $limpa\_candidatos(C_k)$, para posteriormente continuar sua execução descobrindo os conjuntos candidatos da próxima transação do *split* S_q.

5.2.1 Exemplificando o Algoritmo UAprioriMRByT

Considere o mesmo *dataset* de exemplo ilustrado na Figura 5.2. O passo $k = 1$ do algoritmo UAprioriMRByT funciona do mesmo modo que o algoritmo UAprioriMR, gerando, portanto, o mesmo conjunto frequente de tamanho 1: $L_1 = (D\ 5.1), (H\ 2.1), (I\ 4.4), (O\ 4.7)$, vide Figura 5.6. A diferença do algoritmo está na função Map_q para os passos k > 1.

No passo $k = 2$, ao invés de gerar C_2 a partir de L_1, o conjunto C_2 será gerado por transação. O conjunto $O_{q,k}$, resultante da função Map_q, sobre cada um dos *splits*, é o mesmo daquele gerado pelo algoritmo UAprioriMR, visualizado na Figura 5.7. Como as funções $Combine_q$ e Reduce são as mesmas do algoritmo UAprioriMR, as saídas delas também são iguais àquelas já ilustradas pelas Figuras 5.8 e 5.9, respectivamente.

No passo $k = 3$ a função Map_q varre cada transação novamente gerando o conjunto de itemsets candidatos de tamanho 3 (3-itemset). A saída desta função também é igual a saída da função Map_q do algoritmo UAprioriMR, representada na Figura 5.10. As funções $Combine_q$ e Reduce são as mesmas do algoritmo UAprioriMR e geram as mesmas saídas, exibidas nas Figuras 5.11 e 5.12.

Observa-se portanto, que as saídas das funções Map, Combine e Reduce são as mesmas em ambos os algoritmos: UAprioriMR e UAprioriMRByT. Todavia, o número de comparações de itemsets dos conjuntos candidatos com as transações dos *splits*, na função Map_q, para os passos $k > 1$, é diferente nos dois algoritmos.

5.3 UAprioriMR x UAprioriMRByT

Esta seção analisa o funcionamento da função Map_q de cada um dos dois algoritmos: UAprioriMR e UAprioriMRByT. A seção evidencia que o número de comparações de itemsets do conjunto candidato com as transações dos *splits*, na função Map_q, para os passos $k > 1$, é diferente nos dois algoritmos. Desta forma, auxilia a compreensão do algoritmo híbrido UApriorMRJoin proposto.

5.3.1 Análise da Função Map_q no Passo $k = 2$

Observe nas Figuras 5.7 e 5.10, o número de elementos de saída ($O_{q,k}$) da função Map_q, nos passos $k = 2$ e $k = 3$, do algoritmo UAprioriMR. No passo $k = 2$, são gerados 8 elementos em $O_{1,2}$ e 5 elementos em $O_{2,2}$. No passo $k = 3$, são produzidos 2 elementos em $O_{1,3}$ e 1 elemento em $O_{2,3}$. O algoritmo UAprioriMRByT produz o mesmo número de elementos em $O_{1,2}$, $O_{2,2}$, $O_{1,3}$ e $O_{2,3}$ que o algoritmo UAprioriMR.

A Tabela 5.1 exibe a saída da função Map_q e da função Reduce nos 3 passos executados pelos algoritmos UAprioriMR e UAprioriMRByT. Como saída da função Map_q são exibidos os números de elementos produzidos pela função Map_q, a cada passo k, em cada um dos dois *splits* S_1 e S_2: $O_{1,k}$ e $O_{2,k}$. Como saída da função Reduce, são mostradas as quantidades de itemsets frequentes do conjunto L_k, para cada passo k.

No passo $k = 1$, as funções Map, Combine e Reduce, nos dois algoritmos, são iguais e geram o mesmo número de elementos nos conjuntos $O_{q,k}$ ($O_{1,1}$ e $O_{2,1}$) e o mesmo número de itemsets no conjunto de itens frequentes L_1. Nos passos $k = 2$ e $k = 3$, os conjuntos frequentes descobertos (L_2 e L_3) também são os mesmos. Embora a função Map aplique uma abordagem diferente em cada um dos dois algoritmos, nos passos $k = 2$ e $k = 3$, o número de elementos gerados em $O_{1,2}$, $O_{2,2}$, $O_{1,3}$ e $O_{2,3}$, são iguais.

Tabela 5.1: Número de elementos gerados pelos algoritmos UAprioriMR e UAprioriMRByT nos conjuntos $O_{q,k}$ e L_k, após a aplicação das funções Map_q e Reduce, respectivamente.

$O_{1,1}$	$O_{2,1}$[1]	L_1	$O_{1,2}$	$O_{2,2}$	L_2	$O_{1,3}$	$O_{2,3}$	L_3
11	9	4	8	5	3	2	1	1

[1] $O_{2,1}$: Número de elementos do conjunto O, no passo $k = 1$, resultante da aplicação da função Map_q sobre o *split* 2.

Entretanto, a geração de itemsets nos conjuntos candidatos, implementada pelos algoritmos UAprioriMR e UAprioriMRByT, é realizada de maneira distinta. Isto afeta diretamente no número de comparações (verificações) a serem feitas entre cada itemset do conjunto candidato com as transações do *dataset*. Esta diferença no número de comparações realizadas influencia no desempenho dos algoritmos. A Tabela 5.2 exibe a diferença no número de itemsets gerados pelos conjuntos candidatos de cada um dos algoritmos UAprioriMR e UAprioriMRByT.

Tabela 5.2: Número de itemsets de L_k[1], C_k[2], C_{kp}[3] e $C_{kp}\ x\ D$[4], durante a aplicação dos algoritmos UAprioriMR e UAprioriMRByT.

Algoritmo	$\|L_1\|$	$\|C_2\|$	$\|C_{2p}\|$	$\|C_{2p}\ x\ D\|$	$\|L_2\|$	$\|C_3\|$	$\|C_{3p}\|$	$\|C_{3p}\ x\ D\|$	$\|L_3\|$
UAprioriMR	4	6	6	60	3	1	1	10	1
UAprioriMRByT	4	13	13	13	3	3	3	3	1

[1] L_k: Conjunto de itens frequentes L, no passo k.
[2] C_k: Conjunto candidato C, no passo k.
[3] C_{kp}: Conjunto candidato C, no passo k, após a aplicação da função de poda.
[4] $|C_{kp}\ x\ D|$: Número de comparações realizadas entre cada elemento do conjunto candidato C_{kp} com todas as transações existentes no *dataset* D.

Analisando a Tabela 5.2 no passo $k = 2$, o algoritmo UAprioriMR gera 6 itemsets no conjunto candidato C_2. Estes 6 itemsets passam pela função de poda, $poda\_candidatos(C_k, L_{k-1})$ e o conjunto candidato após a aplicação da poda, C_{2p}, continua com os mesmos 6 itemsets. Posteriormente, é necessário verificar a existência de cada um dos 6 itemsets do conjunto candidato C_{2p}

com cada uma das transações existentes, tanto no *split* S_1, quanto no *split* S_2. Estas verificações (comparações) na Tabela 5.2 estão representadas por $C_{2p}\ x\ D$. Como existem 5 transações em cada *split*, são realizadas 30 verificações por *split* e, consequentemente, 60 no total, conforme detalhado nas Tabelas 5.3 e 5.4.

As Tabela 5.3 exibe detalhadamente o que ocorre com cada transação t_j do *split* S_1. Tomando-se como exemplo a transação $t_j = 1$. O conjunto candidato já foi podado, e continua com 6 itemsets, C_{2p}: $\{(DH),(DI),(DO),(HI),(HO),(IO)\}$, precisa ter sua existência verificada na transação $t_j = 1$. Desta forma, para a transação $t_j = 1$ são realizadas 6 comparações, representadas na coluna $C_{2p}\ x\ D$. Consequentemente, são realizadas 30 verificações no *split* S_1. O mesmo ocorre com o *split* S_2, gerando outras 30 verificações, exibidas na Tabela 5.4.

Tabela 5.3: Itemsets gerados no conjunto C_{2p}, comparados com cada transação do *split* S_1, utilizando o algoritmo UAprioriMR.

$\mathbf{C_{2p}}$	$\mathbf{t_j}$	$\mathbf{C_{2p}\ x\ D}$	$\mathbf{\lvert C_{2p}\ x\ D\rvert}$
{(DH),(DI),(DO),(HI),(HO),(IO)}	1	$(DH) \in t_1?\ (DI) \in t_1?\ (DO) \in t_1?\ (HI) \in t_1?$ $(HO) \in t_1?\ (IO) \in t_1?$	6
	2	$(DH) \in t_2?\ (DI) \in t_2?\ (DO) \in t_2?,\ (HI) \in t_2?$ $(HO) \in t_2?\ (IO) \in t_2?$	6
	3	$(DH) \in t_3?\ (DI) \in t_3?\ (DO) \in t_3?,\ (HI) \in t_3?$ $(HO) \in t_3?\ (IO) \in t_3?$	6
	4	$(DH) \in t_4?\ (DI) \in t_4?\ (DO) \in t_4?,\ (HI) \in t_4?$ $(HO) \in t_4?\ (IO) \in t_4?$	6
	5	$(DH) \in t_5?\ (DI) \in t_5?\ (DO) \in t_5?,\ (HI) \in t_5?$ $(HO) \in t_5?\ (IO) \in t_5?$	6
			30

Tabela 5.4: Itemsets gerados no conjunto C_{2p}, comparados com cada transação do *split* S_2, utilizando o algoritmo UAprioriMR.

$\mathbf{C_{2p}}$	$\mathbf{t_j}$	$\mathbf{C_{2p}\ x\ D}$	$\mathbf{\lvert C_{2p}\ x\ D\rvert}$
{(DH),(DI),(DO),(HI),(HO),(IO)}	6	$(DH) \in t_6?\ (DI) \in t_6?\ (DO) \in t_6?\ (HI) \in t_6?$ $(HO) \in t_6?\ (IO) \in t_6?$	6
	7	$(DH) \in t_7?\ (DI) \in t_7?\ (DO) \in t_7?,\ (HI) \in t_7?$ $(HO) \in t_7?\ (IO) \in t_7?$	6
	8	$(DH) \in t_8?\ (DI) \in t_8?\ (DO) \in t_8?,\ (HI) \in t_8?$ $(HO) \in t_8?\ (IO) \in t_8?$	6
	9	$(DH) \in t_9?\ (DI) \in t_9?\ (DO) \in t_9?,\ (HI) \in t_9?$ $(HO) \in t_9?\ (IO) \in t_9?$	6
	10	$(DH) \in t_{10}?\ (DI) \in t_{10}?\ (DO) \in t_{10}?,$ $(HI) \in t_{10}?\ (HO) \in t_{10}?\ (IO) \in t_{10}?$	6
			30

Utilizando outra abordagem, o algoritmo UAprioriMRByT faz um número menor de verificações sobre a existência dos itemsets do conjunto candidato com cada uma das transações. Este número de verificações está representado pela coluna $|C_{2p}\ x\ D|$, nas Tabelas 5.5 e 5.6. Neste algoritmo, os itemsets do conjunto candidato são gerados por linha (por transação). Logo, não há um conjunto candidato C_{2p} prévio, gerado a partir de L_1, como existe no algoritmo UAprioriMR. Por este motivo $C_{2p} = \emptyset$ nas Tabelas 5.5 e 5.6.

Observe a transação $t_j = 2$, do *split* S_1, representada na Figura 5.3. Estão presentes nesta transação os itens D e O, ambos com probabilidades existenciais iguais a 0.9. Conforme a Tabela 5.5, o elemento (DO) é produzido para o conjunto candidato C_2. Somente este elemento pode ser gerado, pois só existem os itens D e O na transação $t_j = 2$. Após a aplicação da poda, o conjunto candidato (DO) permanece no conjunto C_{2p}, porque seus itemsets, D e O, pertencem ao conjunto de itemsets frequentes L_1. Desta forma, somente 1 verificação, $|C_{2p}\ x\ D|$, é realizada para a transação $t_j = 2$, que é a geração do conjunto (DO).

Observando um outro exemplo, a transação $t_j = 3$ da Figura 5.3. Esta transação tem três itens distintos: D, I e O. Portanto, para esta transação são gerados os três itemsets possíveis em C_2: $\{(DI),(DO),(IO)\}$. A poda também mantém os três itemsets em C_{2p}. E, neste caso, foram geradas 3 verificações na transação $t_j = 3$, representada na Tabela 5.5, na coluna $|C_{2p}\ x\ D|$. A transação $t_j = 1$ não gera nenhum conjunto candidato, pois possui apenas o item H no *dataset* original.

A Tabela 5.6 ilustra o mesmo raciocínio utilizando-se o *split* S_2. Portanto, pode-se notar que, com o algoritmo UAprioriMRByT, são geradas 8 verificações no *split* S_1 e 5 verificações no *split* S_2, totalizando 13 verificações sobre todas as transações do *dataset*, no passo $k = 2$. Este valor de verificações é menor do que as 60 verificações realizadas pelo algoritmo UAprioriMR.

Tabela 5.5: Itemsets do conjunto candidato C_{2p}, gerados a partir da leitura de cada transação do *split* S_1, utilizando o algoritmo UAprioriMRByT.

$\mathbf{C_{2p}}$	$\mathbf{t_j}$	$\mathbf{C_{2p}\ x\ D}$	$\mathbf{\lvert C_{2p}\ x\ D\rvert}$
$\emptyset$[1]	1	—[2]	0
	2	$(DO) \in t_2!$	1
	3	$(DI) \in t_3!\ (DO) \in t_3!\ (IO) \in t_3!$	3
	4	$(DI) \in t_4!\ (DO) \in t_4!\ (IO) \in t_4!$	3
	5	$(DO) \in t_5!$	1
			8

[1]O símbolo $\emptyset$ indica que não há C_{2p} gerado a partir do conjunto L_1.
[2]O símbolo — indica que não há verificação a ser feita pois existe apenas 1 item nesta transação.

Tabela 5.6: Itemsets do conjunto candidato C_{2p}, gerados a partir da leitura de cada transação do *split* S_2, utilizando o algoritmo UAprioriMRByT.

$\mathbf{C_{2p}}$	$\mathbf{t_j}$	$\mathbf{C_{2p}\ x\ D}$	$\mathbf{\lvert C_{2p}\ x\ D\rvert}$
$\emptyset$[1]	6	$(HI) \in t_6!$	1
	7	—[2]	0
	8	—[2]	0
	9	$(DI) \in t_9!\ (DO) \in t_9!\ (IO) \in t_9!$	3
	10	$(HI) \in t_{10}!$	1
			5

[1]O símbolo $\emptyset$ indica que não há C_{2p} gerado a partir do conjunto L_1.
[2]O símbolo — indica que não há verificação a ser feita pois existe apenas 1 item nesta transação.

A fim de compreender melhor a diferença do número de verificações realizadas pelos dois algoritmos, ao observar a Figura 5.3, nota-se que o *split* S_1 tem 11 itens em todas as suas 5 transações. Isto é, neste *split* há uma média de itens por transação igual a 2.2 ($\lceil 2.2 \rceil = 3$). Logo, o máximo de verificações, e consequentemente, o número máximo de itemsets em C_2, que podem ser obtidos

neste *split*, utilizando o algoritmo UAprioriMRByT, é igual a 15. Pois, multiplica-se o máximo de itemsets que cada transação pode gerar, $C_{3,2} = \frac{3!}{(3-2)! \cdot 2!} = 3$, pelo número de transações do *split* S_1, 5. Portanto, $3 \cdot 5 = 15$ verificações.

Da mesma maneira, o *split* S_2 tem 9 itens em suas 5 transações. Sua média de itens por transação é igual a 1.8 ($\lceil 1.8 \rceil = 2$). O máximo de verificações neste *split* é igual a 5. Isto porque multiplica-se $C_{2,2} = \frac{2!}{(2-2)! \cdot 2!} = 1$, pelas 5 transaçoes existentes no *split* S_2. Ou seja, no máximo, são geradas 5 verificações pelo algoritmo UAprioriMRByT em S_2.

Somando o máximo de 15 verificações que podem ser realizadas no *split* S_1, com as 5 verificações do *split* S_2, tem-se um máximo de 20 verificações que podem ser computadas. Este valor é inferior às 60 verificações realizadas pelo algoritmo UAprioriMR. Na realidade, como visto nas Tabelas 5.5 e 5.6, este número máximo de 20 verificações não foi atingido, pois foram produzidas 13 $(8 + 5)$ verificações.

Estendendo a proporções maiores o exemplo dado, tome-se o *dataset* T25I15D320k, utilizado nos experimentos do Capítulo 6 e descrito na Seção 6.1. Ele tem 320 mil transações, onde cada transação (linha) é formada por pares de itens e suas probabilidades existenciais. O *dataset* tem 995 itens distintos e uma média de 26.2 itens por transação. Considere também que, após o passo $k = 1$, o conjunto frequente L_1 tenha gerado 659 1-itemsets frequentes.

O algoritmo UAprioriMR então, no passo $k = 2$, ao gerar os itemsets do conjunto candidato C_2 tem, de acordo com a Equação 5.1, $C_{659,2} = \frac{659!}{(659-2)! \cdot 2!} = 216.811$ itemsets candidatos. Após a geração dos itemsets candidatos é aplicada a função de poda do tradicional algoritmo Apriori sobre o conjunto candidato gerado. A poda diz que, *"se um itemset c é frequente, com um mínimo suporte esperado, então seus subconjuntos também são frequentes, considerando o mesmo suporte"*. Portanto, se houver algum subconjunto do itemset c que seja infrequente, o itemset c pode ser descartado, pois ele não será um itemset frequente.

No entanto, no passo $k = 2$, a função de poda não é aplicável. Assim, tanto no pequeno exemplo ilustrado pela Figura 5.2, quanto no *dataset* T25I15D320k, todos os itemsets candidatos de tamanho 2, elementos de C_2, são gerados e nenhum deles é podado. Isto porque todos os subconjuntos de itemsets de tamanho 2, ou seja, os itemsets de tamanho 1, são frequentes. Exemplo: dado o itemset candidato de tamanho 2, DH, obviamente seus subconjuntos, os itemsets de tamanho 1, D e H, são frequentes. C_{kp} representa o conjunto de itemsets candidatos, no passo k, após a aplicação da função de poda.

Desta forma, o algoritmo UAprioriMR, no *dataset* T25I15D320k, faz 69,379,520,000 verificações, $|C_{2p}\, x\, D|$. Isto porque são $320,000$ transações multiplicado pelos $216,811$ itemsets de C_{2p}, $320,000 \cdot 216.811 = 69,379,520,000$. Esta quantidade muito grande de verificações, realizada pela função Map_q, no algoritmo UAprioriMR, prejudica o desempenho de sua execução.

O algoritmo UAprioriMRByT, por outro lado, faz um número significativamente menor de verificações, $|C_{2p}\ x\ D|$, do que o algoritmo UAprioriMR no passo $k = 2$. Como a média de itens por transação é igual a 26.2, são produzidos no mínimo $C_{26,2} = \frac{26!}{(26-2)! \cdot 2!} = 325$, e no máximo, $C_{27,2} = \frac{27!}{(27-2)! \cdot 2!} = 351$ itemsets em C_2. Considerando que nenhum itemset de C_2 é podado, C_{2p} tem no mínimo 325 e no máximo 351 itemsets. Como existem 320 mil transações no *dataset*, são realizadas no mínimo 104,000,000 $(320,000 \cdot 325)$ e no máximo 112,320,000 $(320,000 \cdot 351)$ verificações, valor substancialmente menor do que 69,379,520,000 (número de verificações gerado pelo UAprioriMR).

Logo, pode-se observar que, quando o número de itens por transação for pequeno e, em contrapartida, o conjunto candidato C_{2p}, gerado pelo algoritmo UAprioriMR, tiver muitos itemsets, o algoritmo UAprioriMRByT faz um número menor de verificações durante a aplicação da função Map_q. Assim, neste passo, o algoritmo UAprioriMRByT tem um melhor desempenho do que o

algoritmo UAprioriMR. Esta situação torna-se evidente no passo $k = 2$ porque neste momento o algoritmo UAprioriMR não consegue tirar proveito da poda. Nos passos $k > 2$, a poda, na maioria das vezes, permite a redução do número de itemsets dentro de $C_{k>2\ p}$ e, consequentemente, melhora o desempenho do algoritmo UAprioriMR.

5.3.2 Análise da Função Map_q no Passo $k = 3$

Usando o exemplo da Figura 5.3, no passo $k = 3$, o algoritmo UAprioriMRByT também produz um número menor de verificações do que o algoritmo UAprioriMR. O algoritmo UAprioriMR produz 10 verificações (5 em cada *split*), enquanto o algoritmo UAprioriMRByT executa 3 verificações (2 no *split* S_1 e 1 no *split* S_2), conforme pode ser visualizado nas Tabelas 5.7, 5.8, 5.9 e 5.10.

Tabela 5.7: Itemsets gerados no conjunto C_{3p}, comparados com cada transação do *split* S_1, utilizando o algoritmo UAprioriMR.

$\mathbf{C_{3p}}$	$\mathbf{t_j}$	$\mathbf{C_{3p}\ x\ D}$	$\mathbf{\lvert C_{3p}\ x\ D\rvert}$
$\{(DIO)\}$	1	$(DIO) \in t_1?$	1
	2	$(DIO) \in t_2?$	1
	3	$(DIO) \in t_3?$	1
	4	$(DIO) \in t_4?$	1
	5	$(DIO) \in t_5?$	1
			5

Tabela 5.8: Elementos gerados no conjunto C_{3p}, comparados com cada transação do *split* S_2, utilizando o algoritmo UAprioriMR.

$\mathbf{C_{3p}}$	$\mathbf{t_j}$	$\mathbf{C_{3p}\ x\ D}$	$\mathbf{\lvert C_{3p}\ x\ D\rvert}$
$\{(DIO)\}$	6	$(DIO) \in t_6?$	1
	7	$(DIO) \in t_7?$	1
	8	$(DIO) \in t_8?$	1
	9	$(DIO) \in t_9?$	1
	10	$(DIO) \in t_{10}?$	1
			5

Ao observar a Tabela 5.1, nota-se que o algoritmo UAprioriMR gera 3 elementos no conjunto de itens frequentes L_2, são eles: $L_2 = \{(DI), (DO), (IO)\}$. Os elementos gerados no conjunto candidato C_3 só podem ser aqueles no quais todos os seus subconjuntos estejam presentes em L_2. Por este motivo, $C_3 = C_{3p} = \{(DIO)\}$. É necessário verificar a existência deste elemento com as 5 transações existentes no *split* S_1. Desta forma, são produzidas 5 verificações no *split* S_1, conforme exibido na Tabela 5.7 e outras 5 no *split* S_2, de acordo com a Tabela 5.8, totalizando 10 verificações.

De forma diferente, o algoritmo UAprioriMRByT produz os conjuntos candidatos por transação. Logo, os itemsets produzidos em cada transação são aqueles demonstrados nas Tabelas 5.9 e 5.10. Observa-se que o algoritmo UAprioriMRByT fez uma verificação na transação $t_j = 3$ e outra verificação na transação $t_j = 4$, totalizando 2 verificações no *split* S_1 (Tabela 5.9).

Observando a Tabela 5.10 sobre o *split* S_2, nota-se que foi realizada apenas 1 verificação para geração de conjunto candidato na transação $t_j = 9$. Logo, o algoritmo UAprioriMRByT fez ao todo 3 verificações para geração de conjuntos candidatos no passo $k = 3$, valor este menor do que as 10 verificações realizadas pelo algoritmo UAprioriMR no mesmo passo. Logo, para o exemplo dado, o

Tabela 5.9: Itemsets do conjunto candidato C_{3p}, gerados a partir da leitura de cada transação do *split* S_1, utilizando o algoritmo UAprioriMRByT.

$\mathbf{C_{3p}}$	$\mathbf{t_j}$	$\mathbf{C_{3p}\ x\ D}$	$\mathbf{\lvert C_{3p}\ x\ D\rvert}$
$\emptyset$[1]	1	—[2]	0
	2	***[3]	0
	3	$(DIO) \in t_3$!	1
	4	$(DIO) \in t_4$!	1
	5	***[3]	0
			2

[1]O símbolo $\emptyset$ indica que não há C_{3p} gerado a partir do conjunto L_2.
[2]O símbolo — indica que não há verificação a ser feita pois existe apenas 1 item nesta transação.
[3]O símbolo *** indica que não há verificação a ser feita pois existem apenas 2 itens nesta transação.

Tabela 5.10: Itemsets do conjunto candidato C_{3p}, gerados a partir da leitura de cada transação do *split* S_2, utilizando o algoritmo UAprioriMRByT.

$\mathbf{C_{3p}}$	$\mathbf{t_j}$	$\mathbf{C_{3p}\ x\ D}$	$\mathbf{\lvert C_{3p}\ x\ D\rvert}$
$\emptyset$[1]	6	***[3]	0
	7	—[2]	0
	8	—[2]	0
	9	$(DIO) \in t_9$!	1
	10	***[3]	0
			1

[1]O símbolo $\emptyset$ indica que não há C_{3p} gerado a partir do conjunto L_2.
[2]O símbolo — indica que não há verificação a ser feita pois existe apenas 1 item nesta transação.
[3]O símbolo *** indica que não há verificação a ser feita pois existem apenas 2 itens nesta transação.

algoritmo UAprioriMRByT faz um número menor de verificações do que o algoritmo UAprioriMR, a fim de gerar a saída da função Map_q.

No entanto, a partir do passo $k > 2$, é importante salientar que os itemsets do conjunto candidato $C_{k>2}$ são podados no algoritmo UAprioriMR. Esta poda tende a ser eficiente nestes passos, reduzindo substancialmente o número de itemsets deste conjunto (gerando $C_{k>2\ p}$). Desta forma, nestes passos, para gerar a saída da função Map_q, o algoritmo AprioriMR pode fazer um número de verificações inferior ao número de verificações realizadas pelo algoritmo UAprioriMRByT.

5.4 O Algoritmo UAprioriMRJoin

Percebe-se que o algoritmo UAprioriMRByT pode obter melhor desempenho do que o algoritmo UAprioriMR, em alguns casos, principalmente no passo $k = 2$ (pela inexistência de poda). Tal característica evidencia-se quando a média de itens por transação do *dataset* for pequena. A fim de tirar proveito desta característica, elaborou-se o algoritmo híbrido UAprioriMRJoin (algoritmo 5.6). A ideia deste algoritmo é implementada na função Map_q nos passos $k > 1$. Portanto, no passo $k = 1$, as funções Map, Combine e Reduce são as mesmas do algoritmo UAprioriMR. Assim como as funções Combine e Reduce são iguais ao algoritmo UAprioriMR nos passos $k > 1$.

Imaginando que a função Map_q, descrita no Algoritmo 5.6, está executando no passo $k = 2$:

Algoritmo 5.6: Função Map_q do Algoritmo UAprioriMRJoin no Passo $k > 1$.

Entrada: S_q, *split* q, onde cada linha é uma transação.
Entrada: conjunto L_{k-1} contendo pares (c, p) de itemsets frequentes encontrados no passo anterior (k-1).
Entrada: avg, média de itens por transação.
Saída: conjunto $O_{q,k}$ referente ao *split* S_q, contendo pares (c, p).
$Map_q(S_q, L_{k-1}, avg)$
1: $C_k \leftarrow gera\_candidatos(L_{k-1})$
2: $C_k \leftarrow poda\_candidatos(C_k, L_{k-1})$
3: $C_{k\_t} \leftarrow calcula\_candidatos(avg, k)$
4: **if** $(|C_k| \geq \lambda \cdot C_{k\_t})$ **then**
5: UAprioriMRByT.$Map_q(S_q, L_{k-1})$
6: **else**
7: UAprioriMR.$Map_q(S_q, L_{k-1})$
8: **end if**
9: return $O_{q,k}$

na linha 2 do algoritmo, o conjunto C_k concentra todos os itemsets candidatos a itens frequentes L_2, gerados a partir de L_1 e já podados, conforme ideia tradicional do algoritmo UAprioriMR. Na linha 3, a variável C_{k_t} armazena o número de itemsets candidatos a L_2, calculando-o a partir do número médio de itens existentes por transação (avg). Este cálculo é realizado pela função $calcula\_candidatos(avg, k)$ que, aplicando a Equação 5.1, descobre o valor de C_{k_t}. Ou seja, considerando o passo $k = 2$, dada a média de itens por transação (avg), calcula-se o número de itemsets candidatos por transação, tomados 2 a 2.

O número de itemsets em C_k, $|C_k|$, e o número de itemsets por transação, C_{k_t}, servem para decidir se deve ser aplicada a função Map_q do algoritmo UAprioriMR, ou se deve ser aplicada a função Map_q do algoritmo UAprioriMRByT. Esta decisão é tomada na linha 4. Caso o número de itemsets candidatos em C_k seja maior do que o número de itemsets candidatos por transação, multiplicado por uma constante λ, deve ser executada a função Map_q do algoritmo UAprioriMRByT porque o número de verificações desta função produz um número menor de elementos do que o número de verificações da função Map_q do algoritmo UAprioriMR.

A constante λ serve como medida da oscilação do tempo de processamento de cada uma das abordagens. Existe diferença no tempo de processamento entre descobrir os itemsets do conjunto candidato a cada transação (abordagem UAprioriMRByT) e gerar estes itemsets uma única vez, a partir de L_{k-1} (abordagem UAprioriMR).

5.5 Considerações Finais do Capítulo

Neste capítulo foram formalizados e discutidos os algoritmos UAprioriMR, UAprioriMRByT e UAprioriMRJoin. Destas discussões destacam se algumas contribuições relacionadas aos algoritmos propostos:

1. A criação e implementação do algoritmo UAprioriMR. Embora existam outros trabalhos relacionados que adaptaram o algoritmo tradicional Apriori para trabalhar com o modelo de programação MapReduce (vide [LI08], [LI12], [LIN12b], [YAH12] e [KUL13]), não foram encontrados trabalhos que lidassem com incerteza no *dataset*. O algoritmo UAprioriMR implementa o modelo de programação MapReduce sobre *datasets* que possuem incerteza associada

aos seus itens;

2. A criação e implementação do algoritmo UAprioriMRByT. Este algoritmo modifica a maneira de criação dos itemsets dos conjuntos candidatos de cada passo k, gerando-os a cada transação lida do *dataset*;

3. A criação e implementação do algoritmo híbrido UAprioriMRJoin. Este algoritmo decide, a cada passo $k > 1$, qual abordagem utilizar: se usar a geração de itemsets dos conjuntos candidatos C_k, baseado em L_{k-1} (ideia do UAprioriMR) ou usar a geração de itemsets dos conjuntos candidatos por transação (abordagem do UAprioriMRByT).

6. EXPERIMENTOS

Este capítulo descreve os experimentos realizados com os algoritmos UAprioriMR, UaprioriMRByT e UAprioriMRJoin. Inicialmente é realizada uma caracterização de diversas variáveis envolvidas nos experimentos: datasets utilizados, configuração do laboratório experimental e a metodologia aplicada sobre os experimentos. Por fim, o capítulo discute o desempenho dos algoritmos UAprioriMR, UaprioriMRByT e UAprioriMRJoin e enfatiza o desempenho superior do algoritmo UAprioriMRJoin em relação ao algoritmo UAprioriMR.

6.1 *Datasets*

Para a realização dos experimentos foram escolhidos quatro *datasets* utilizados por Tong et al. [TON12]: accidents, connect, kosarak e T25I15D320k e o *dataset* kosarak_10. Os critérios de escolha dos *datasets* foram: a variedade de suas características e a ampla utilização destas bases em outros trabalhos sobre descoberta de conjuntos de itens frequentes em contextos de incerteza. À exceção do T25I15D320k, que é um *dataset* sintético, os demais são *datasets* reais. O *dataset kosarak_10*, criado para esta tese, é uma derivação do *dataset kosarak*, e é explicado o porquê de sua criação na Seção 6.6.3.

A partir dos repositórios UCI e FIMI, os quatro datasets reais originais podem ser recuperados. Estes *datasets* reais estão presentes em diversos estudos realizados na área de descoberta de conjuntos de itens frequentes [AGG09b] [CAL10] [SUN10] [WAN10] [TAN11] [TON12]. Nos experimentos realizados nesta tese, no entanto, foram usados *datasets* com incerteza já associada aos itens das transações, *datasets* estes disponibilizados pelo trabalho de Tong et al. [TON12]. Quanto ao *dataset* sintético, *T25I15D320k*, ele pode ser gerado a partir da ferramenta *IBM Synthetic Data Generator* [AGR94].

A ferramenta *IBM Synthetic Data Generator* é resultado do projeto *Quest*, do Centro de Pesquisa da IBM em Almaden. Ela foi descrita no artigo de Agrawal e Srikant [AGR94]. Seus arquivos originais não são mais disponibilizados pela IBM, mas há um projeto no *sourceforge*[1] e um site do pesquisador Chris Giannella[2], que disponibilizam os códigos-fonte em C++ e C, respectivamente. Este gerador permite criar bases de dados automáticas, configuráveis e com transações de usuários que simulam uma *market basket data*. A geração de um *dataset* sintético é realizada a partir da configuração de três elementos:

1. número médio de itens por transação (T);
2. tamanho médio de itens frequentes (I).;
3. número de transações na base (D).

Portanto, a base de dados sintética *T25I15D320k* tem uma média de 25 itens por transação, o tamanho médio de itens frequentes (itemsets) é igual a 15 e o *dataset* é constituído de 320 mil transações. Esta base de dados é utilizada no trabalho de [TON12] e tem a nomenclatura citada. No entanto, ao investigá-la com mais profundidade, descobriu-se que o número médio de itens por transação, na verdade, é igual a 26.2. A fim de não causar confusão será mantida a nomenclatura *T25I15D320k* para referir-se a este *dataset*.

[1] http://ibmquestdatagen.sourceforge.net/
[2] http://www.cs.loyola.edu/∼cgiannel/assoc_gen.html

Tabela 6.1: Descrição dos *datasets* reais utilizados nos experimentos da tese.

Dataset	**Repositório**	**Descrição**
Accidents	FIMI	Doado por Karolien Geurts, contém dados anônimos de acidentes de tráfego.
Connect	FIMI	Contém as posições legais e jogadas realizadas no jogo connect-4.
Kosarak	UCI	Fornecido por Ferenc Bodon, este *dataset* possui dados anônimos de cliques realizados sobre um portal de notícias húngaro.
Kosarak_10	—	*Dataset* gerado a partir do *dataset* kosarak, mantendo somente as transações com até 10 itens.

A Tabela 6.1 descreve os *datasets* reais. A Tabela 6.2 apresenta uma síntese de algumas características destes *datasets*: o número de transações/tuplas existentes, o número de itens distintos no *dataset*, o tamanho médio de uma transação e a densidade[3] do *dataset*.

Tabela 6.2: Características dos *datasets* utilizados nos experimentos.

Dataset	**Transações**	**Itens Distintos**	**Média de Itens por Transação**	**Densidade**
Accidents	340,183	468	33.8	0.072
Connect	67,557	129	43.0	0.33
Kosarak	990,002	41,270	8.1	0.00019
Kosarak_10	847,875	36,506	3.4	0.000093
T25I15D320k	320,000	995	26.2	0.026

Tabela 6.3: Valores de média e variância usados para inserir probabilidades nos *datasets*.

Dataset	**Média (μ)**	**Variância (σ^2)**
Accidents	0.50	0.50
Connect	0.95	0.05
Kosarak	0.50	0.50
Kosarak_10	0.50	0.01
T25I15D320k	0.90	0.10

Cabe salientar que em todos os *datasets* citados previamente, tanto reais, quanto sintéticos, não há probabilidade associada aos itens. As probabilidades, portanto, são adicionadas de forma aleatória, informando-se a média e a variância desejada. A Tabela 6.3 mostra as médias e variâncias usadas em cada *dataset*, valores iguais aos utilizados no trabalho de [TON12], com exceção do *dataset* kosarak_10. Desta forma, as Tabelas 6.1, 6.2 e 6.3 dão uma visão geral sobre as bases de dados utilizadas nos experimentos.

6.2 Laboratório Experimental

A fim de realizar os diversos experimentos necessários, foi instalado e configurado um laboratório para execução dos mesmos. Este laboratório está situado na Universidade Feevale e conta com

[3] densidade = Média de Itens pro Transação ÷ Itens Distintos

um total de 20 computadores (nodos). Todos os 20 nodos são iguais e apresentam a mesma configuração.

Em cada um dos 20 nodos foi instalada uma máquina virtual para que os experimentos ficassem isolados das outras atividades realizadas neste laboratório. A utilização da máquina virtual deve-se ao fato de o autor não possuir privilégios de administração sobre o laboratório. Com o uso das máquinas virtuais houve total flexibilidade para que o autor pudesse instalar e configurar o *cluster* Apache Hadoop. As configurações dos nodos e da máquina virtual estão descritas na Tabela 6.4.

Tabela 6.4: (a) Configuração dos nodos do laboratório. (b) Configuração da máquina virtual instalada em cada nodo.

(a)	**(b)**
Sistema Operacional Windows 7 Enterprise (SP 1) 64 bits	Sistema Operacional Ubuntu 12.04 64 bits
Intel® Core™ 2 Duo CPU E8400 @ 3.00GHz 2.99GHz	Intel® Core™ 2 Duo CPU E8400 @ 3.00GHz x 2
Memória RAM: 4GB	Memória RAM: 3GB
Hard Disk: 148GB	Armazenamento Virtual: 102,54GB
Sistema de Arquivos NTFS	Sistema de Arquivos ext4
Placa de rede Intel® 82567LM-3 e Placa de rede D-Link DGE-528T	Placa de rede em modo Bridge, utilizando a placa D-Link DGE-528T

A máquina virtual foi criada sobre o sistema operacional Linux, distribuição Ubuntu 12.04. Para montagem do *cluster* foi instalado nesta máquina virtual o Apache Hadoop, versão 1.1.2[4]. Posteriormente, instalou-se a Java Virtual Machine, versão 7[5].

O *hipervisor* ou monitor de máquina virtual utilizado nos experimentos foi o Oracle VM VirtualBox 4.1.18. Este hipervisor é uma solução profissional largamente usada para virtualização de máquinas e além disso é disponibilizada como software *open source*, sob a licença GNU GPL versão 2. O software Apache Hadoop foi instalado com sua versão *default* e não foi realizado *tuning*.

Após a instalação de todas as máquinas virtuais nos 20 nodos, foi realizada a configuração das mesmas. Elegeu-se a máquina a ser considerada o nodo mestre (*master*) e as demais foram configuradas como nodos escravos (*slaves*). Em todos os nodos foi instalado o servidor *ssh openssh-server*, pois a comunicação entre eles, dentro da implementação Apache Hadoop, é realizada por meio do serviço ssh. Para que durante a comunicação entre os nodos não fossem necessários dados de usuário e senha, cada nodo conhece a chave pública dos demais nodos pertencentes ao *cluster*.

O *cluster* tem seus nodos interligados por meio de um *switch* "Dell Power Connect 3324", com 24 portas 10/100 Mbs. A Figura 6.1 ilustra o laboratório experimental completo com os 20 nodos selecionados que compõem o *cluster*. Cabe salientar que todos os testes apresentados nesta tese foram executados sobre a máquina virtual destes nodos. Desta forma, procurou-se reduzir os viéses de interpretação inerentes aos experimentos.

6.3 Metodologia dos Experimentos

Nesta seção é discutida a condução dos experimentos. São evidenciadas questões como a quantidade de experimentos efetuados, os valores de mínimos suportes esperados utilizados, o número

[4]Foi realizado *download* do arquivo hadoop-1.1.2.tar.gz, no site http://hadoop.apache.org/

[5]Foi instalado o arquivo Jdk-7u25-nb-7_3_1-linux-x64.sh, cujo *download* pode ser realizado em http://www.oracle.com/technetwork/java/javase/downloads/index.html

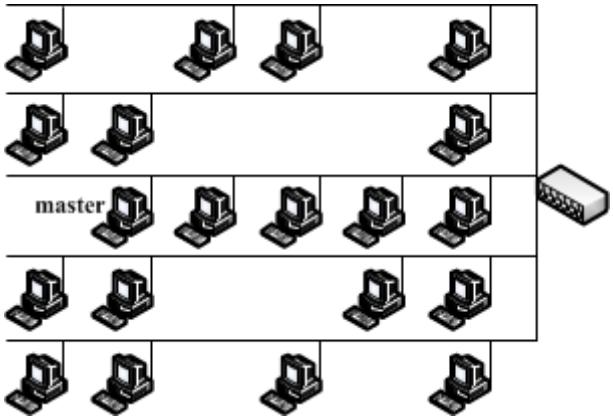

Figura 6.1: Laboratório experimental com 20 nodos, utilizado nos experimentos da tese.

de nodos em cada *cluster*, quais critérios foram definidos quanto a qualidade dos experimentos e o porquê de alguns experimentos serem descartados.

Todos os testes foram realizados sobre a máquina virtual instalada em cada um dos nodos e as bases de dados usadas são as detalhadas na Seção 6.1. A partir dos 20 nodos, foram executados testes sobre *clusters* de 1, 5, 10, 15 e 20 nodos. Desta forma foi possível discutir o comportamento dos algoritmos em relação ao crescimento do *cluster*, e, por consequência, analisar a métrica de *speedup* dos mesmos.

Para realização dos experimentos é importante definir a quantidade de funções map e reduce que serão executadas. Em todos os experimentos efetuados foram utilizadas uma função map por nodo e uma função reduce por *cluster*. Portanto, na execução de um *job* sobre um *cluster* de 5 nodos, foram iniciadas 5 funções map, uma para cada nodo, e uma função reduce somente para o *cluster* de 5 nodos.

6.3.1 Quantidade de Experimentos

Os experimentos foram realizados sobre os *datasets* entre o período de junho à novembro de 2014. Para cada *dataset*, os experimentos foram realizados em função dos algoritmos, bem como o número de nodos do *cluster* e o mínimo suporte esperado. A Tabela 6.5 ilustra o número de testes realizados para cada *dataset* versus o algoritmo aplicado. O *dataset* kosarak foi eliminado das comparações e o número de seus experimentos foi reduzido, pois ele apresentou resultados muito ruins em testes preliminares com o algoritmo UAprioriMRByT. Na Seção 6.6.3 é detalhado o problema dos testes sobre este *dataset* e o porquê da criação do *dataset* kosarak_10.

Tabela 6.5: Número de experimentos realizados com os algoritmos sobre cada *dataset*.

	Accidents	Connect	Kosarak	Kosarak_10	T25I15D320k	**Total**
UApriori sequencial	40	40	—[1]	—	40	**120**
UAprioriMR	200	200	160	160	200	**920**
UAprioriMRByT	140	—	—	—	200	**340**
UAprioriMRJoin	160	160	—	160	160	**640**
Total	**540**	**400**	**160**	**320**	**600**	**2020**

[1]O símbolo — significa que foram realizados testes preliminares sobre os *datasets*, mas não foram utilizados diretamente nas conclusões dos experimentos.

6.3.2 Mínimo Suporte Esperado (minsupesp)

Para cada *dataset* foram escolhidos alguns mínimos suportes esperados, de forma que estes permitissem a geração de conjuntos de itens frequentes de diversos tamanhos. Com esta diversidade de conjuntos de itens frequentes foi possível interpretar a execução de cada algoritmo no passo 1 (geração de 1-itemsets), no passo 2 (geração de 2-itemsets), ..., passo k-1 (geração de k-1-itemsets). É importante salientar que os mínimos suportes esperados variam de *dataset* para *dataset* porque estes conjuntos de dados possuem médias, variâncias e densidades distintas.

A Tabela 6.6 apresenta os valores de minsupesp usados nos testes, sobre os *datasets*, bem como o número gerado de conjuntos de itens frequentes de tamanho 1, 2, 3, ..., k-1. Por exemplo, para o *dataset* accidents foram utilizados os mínimos suportes esperados iguais a 0.3 (30%), 0.2 (20%), 0.1 (10%) e 0.05 (5%). Quando utilizado o mínimo suporte esperado igual a 0.05, foram gerados 75 1-itemsets, 624 2-itemsets, 1451 3-itemsets e 35 4-itemsets frequentes. No passo $k = 5$ o algoritmo não gera 5-itemsets frequentes.

Tabela 6.6: Número de itemsets frequentes, para cada mínimo suporte esperado de cada *dataset*.

Datasets	minsupesp	|1-itemsets|	|2-itemsets|	|3-itemsets|	|4-itemsets|	|5-itemsets|
Accidents	0.3	21	—[1]	—	—	—
	0.2	30	34	—	—	—
	0.1	48	294	48	—	—
	0.05	75	624	1451	35	—
Connect	0.9	0	—	—	—	—
	0.8	23	6	—	—	—
	0.7	29	230	145	—	—
	0.6	32	385	1699	1440	—
Kosarak	0.1	3	—	—	—	—
	0.05	4	2	—	—	—
	0.01	27	18	2	—	—
	0.005	54	45	10	—	—
	0.0025	156	140	34	1	—
Kosarak_10	0.005	26	14	2	—	—
	0.0025	45	41	4	—	—
	0.001	80	99	25	1	—
	0.0001	913	1229	414	46	1
T25I15D320k	0.1	8	—	—	—	—
	0.05	111	—	—	—	—
	0.025	361	—	—	—	—
	0.01	659	229	—	—	—

[1]O símbolo — significa que não foram gerados itemsets frequentes.

6.3.3 Número de Nodos Utilizados

Inicialmente foram realizados testes utilizando os algoritmos UAprioriMR e UAprioriMRByT, implementados com MapReduce. Estes testes foram executados sobre 1, 5, 10, 15 e 20 nodos. O algoritmo híbrido construído, UAprioriMRJoin, que também utiliza o modelo de programação MapReduce, foi executado sobre *clusters* com 5, 10, 15 e 20 nodos. Com este algoritmo não foram realizados testes sobre *cluster* com 1 nodo porque as execuções dos algoritmos UAprioriMR e

UAprioriMRByT, nesta configuração de *cluster*, mostraram resultados muito ruins, discutidos mais adiante na Seção 6.4.1 e exibidos nas Tabelas 6.8 e 6.9.

Os testes realizados com os algoritmos UAprioriMR e UAprioriMRByT, em *clusters* com 1 nodo somente, foram realizados sobre diferentes máquinas virtuais. Esta abordagem objetivou colher os resultados dos experimentos de modo mais rápido. Neste caso, diversos *clusters* foram montados com 1 nodo somente, a partir do ajuste dos arquivos de configuração do Apache Haddop.

Como foi possível observar na Tabela 6.5, muitos testes experimentais foram realizados, demandando, por consequência, bastante tempo para que finalizassem. Agravando esta questão, quando o número de nodos do *cluster* e o mínimo suporte esperado são pequenos, o tempo para conclusão da execução do experimento tende a ser muito grande. Em alguns casos, a execução completa demora muitas horas e até mesmo dias.

Portanto, a fim de coletar os resultados dos diversos experimentos de modo mais eficiente, o laboratório experimental foi configurado de modo diferente, dependendo do tipo de execução realizado. Quando as execuções ocorreram sobre 5 nodos, foram criados 4 *clusters*. Quando as execuções aconteceram sobre 10 nodos, 2 *clusters* foram montados. Nas execuções de 15 e 20 nodos, apenas 1 *cluster* foi montado. A Figura 6.2 exibe os 4 *clusters* de 5 nodos e a Figura 6.3 exibe os 2 *clusters* de 10 nodos.

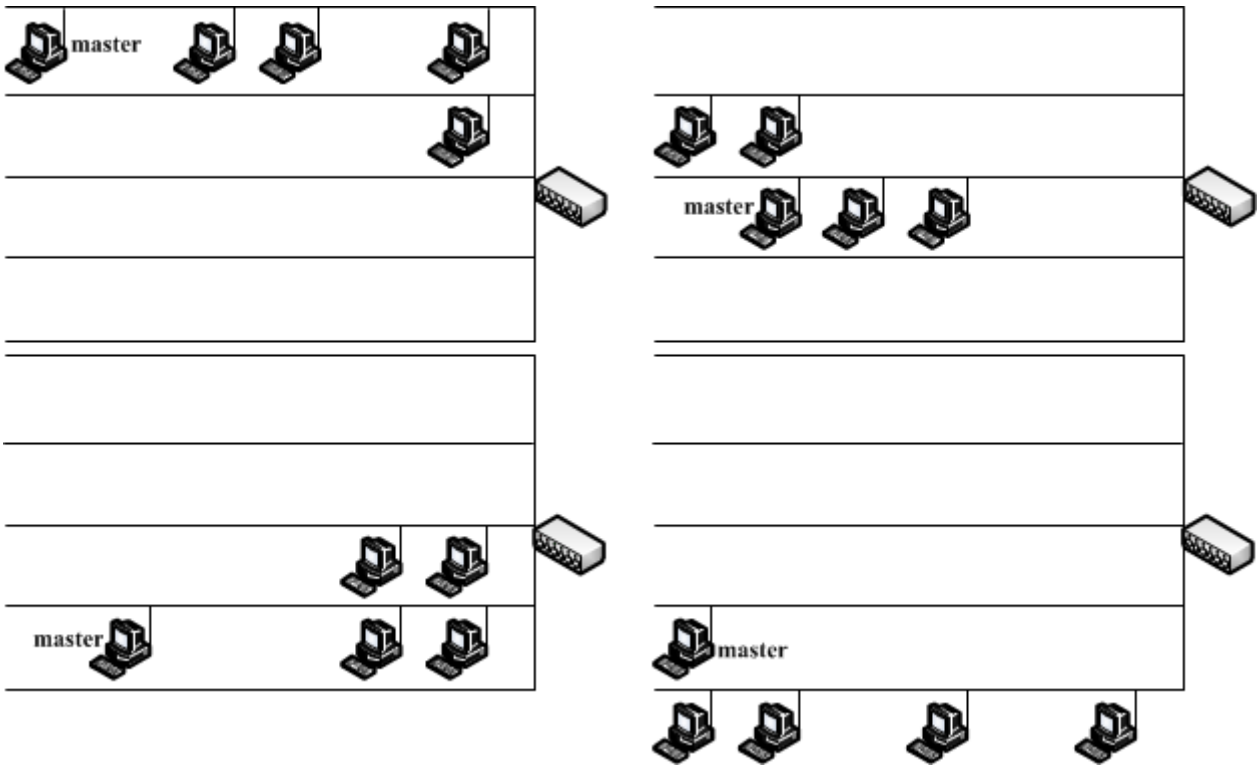

Figura 6.2: 4 *clusters* com 5 nodos cada.

Cabe salientar que os experimentos foram automatizados por *scripts* escritos em *shell script*. Estes *scripts* permitiram controlar, de modo automático, diversas variáveis: os mínimos suportes esperados, o tamanho do *cluster*, o número de experimentos a serem realizados, a inicialização e parada do *cluster*, dentre outras tarefas para a execução de cada um dos algoritmos. Três destes *scripts* estão detalhados no Apêndice A.

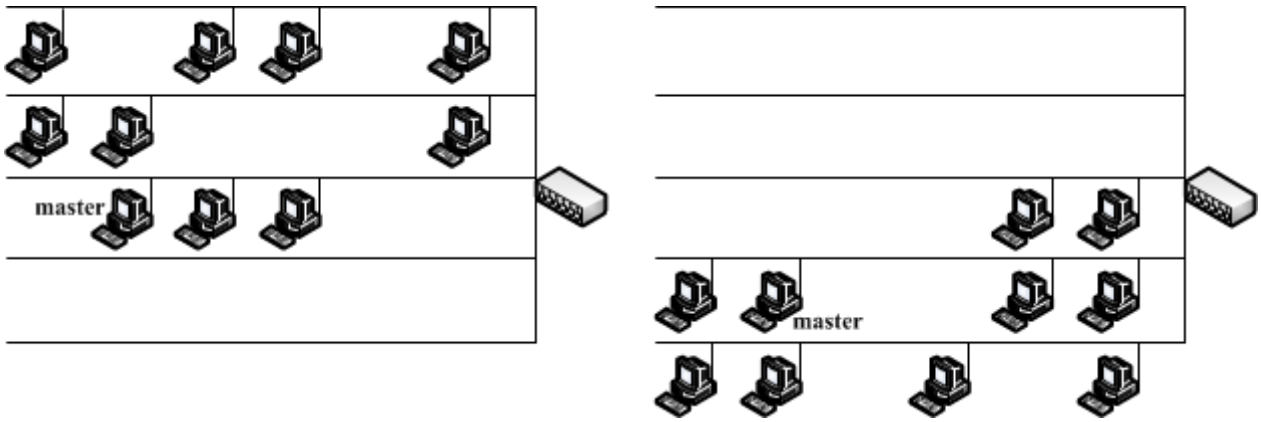

Figura 6.3: 2 *clusters* com 10 nodos cada.

6.3.4 Descarte de Experimentos

Definiu-se, inicialmente, coletar somente aqueles experimentos que não apresentaram qualquer tipo de falha. As falhas, durante um determinado experimento, podem ocorrer ou por causa de problemas nas máquinas virtuais; ou em alguma tarefa Map, Combine ou Reduce; ou algum cabo de rede desconectado; ou, ainda, em virtude de uma falta de energia. Sempre que identificados estes tipos de falha, o *job* parava ou demorava um tempo muito grande para finalizar. Portanto, estes experimentos foram descartados.

O Apache Hadoop guarda muitos *logs* durante a execução de seus testes. Durante os experimentos, algumas máquinas virtuais, por serem muito requisitadas, ficaram com o seu HD Virtual sobrecarregado. Quando o tamanho do HD Virtual se aproximou do tamanho do HD Real, a máquina virtual ficou bem mais lenta e os experimentos demoraram além do esperado. Tais experimentos foram descartados.

Outras vezes, por uso inadequado do laboratório, alguns cabos de rede foram danificados. Quando um nodo não consegue mais ser acessado no *cluster*, a implementação Apache Hadoop elege um outro nodo para realizar as tarefas que não foram concluídas. No entanto, esta delegação de tarefas demanda algum tempo, prejudicando o tempo final do experimento. Logo, os experimentos onde ocorreram estas falhas também foram eliminados.

Alguns episódios de falta de energia no laboratório também ocorreram. Os resultados dos experimentos realizados até o instante da queda de energia também foram desprezados. Ainda, algumas falhas aconteceram durante a execução de uma tarefa Map, Combine ou de uma tarefa Reduce. Sempre que estas falhas interferiram substancialmente no tempo de execução dos experimentos, eles foram abandonados.

6.3.5 Dados Capturados nos Experimentos

Para cada configuração de experimento (*dataset* x minsupesp x *cluster*), foram executados 10 testes e capturada a média destes. Os experimentos foram executados com *scripts* semelhantes àqueles descritos no Apêndice A, e todos os resultados foram registrados em planilhas eletrônicas. Desta forma, foram geradas 5 planilhas distintas, uma para cada *dataset*. Cada planilha contém uma série de colunas, com a configuração e os resultados obtidos dos experimentos. Estas colunas estão descritas na Tabela 6.7. O Apêndice B ilustra um exemplo do conteúdo das colunas destas planilhas eletrônicas.

Nas próximas seções são discutidos os experimentos realizados sobre os algoritmos e *datasets* já citados. Inicialmente são apresentados os resultados obtidos com a aplicação do algoritmo

Tabela 6.7: Colunas das planilhas eletrônicas utilizadas para registro dos resultados dos experimentos.

coluna	**descrição**
nodo/*cluster*	armazena o número do nodo onde foi executado o *job*, no caso de experimentos em *cluster* com 1 nodo somente/armazena o número do nodo *master* do *cluster*, quando este tem 5, 10, 15 ou 20 nodos.
data	registra a data de início do *job*.
hora	registra o horário de início do *job*.
minsupesp	limiar de mínimo suporte esperado.
|D|	número de registros existentes no *dataset*.
minsupesp · |D|	mínimo suporte esperado multiplicado pelo número de transações no *dataset*.
|nodos|	número de nodos do *cluster*.
|maps|	número de tarefas map executadas.
|reducers|	número de tarefas reducer executadas.
|*split*|	tamanho em bytes do *split*.
job 1	tempo de execução do algoritmo durante a execução do passo 1.
job 2	tempo de execução do algoritmo durante a execução do passo 2.
...	...
job k	tempo de execução do algoritmo durante a execução do passo k.
total	total do tempo de execução acumulado de todos os *jobs* executados.
|1-itemset|	número total de itemsets frequentes gerados de tamanho 1.
|2-itemset|	número total de itemsets frequentes gerados de tamanho 2.
...	...
|(k-1)-itemset|	número total de itemsets frequentes gerados de tamanho (k-1).

UAprioriMR. Posteriormente, os resultados do algoritmo UAprioriMRByT são exibidos e discutidos. Finalmente, a aplicação do algoritmo UAprioriMRJoin é mostrada, e as vantagens e desvantagens de sua utilização são analisadas.

6.4 Experimentos sobre o Algoritmo UAprioriMR

Esta seção apresenta os resultados dos experimentos realizados com o algoritmo UAprioriMR sobre os *datasets* reais e densos: accidents e connect; real e esparso: kosarak_10; sintético e denso: T25I15D320k. Embora este algoritmo seja similiar ao algoritmo SPC, proposto por [LIN12b], não foi encontrado nenhum trabalho que fizesse uma discussão do comportamento dele, usando o modelo de programação MapReduce, sobre *datasets* com incerteza associada. A Tabela 6.8 resume o tempo gasto pelos experimentos usando o algoritmo UAprioriMR sobre os quatro *datasets* citados e para as configurações de *cluster* com 1, 5, 10, 15 e 20 nodos.

6.4.1 Comportamento do Algoritmo UAprioriMR sobre um *Cluster* com 1 Nodo

Observa-se na Tabela 6.8 que não foram realizados experimentos sobre o *dataset* kosarak_10 em *clusters* com somente 1 nodo. Isto ocorreu porque os testes preliminares neste *dataset*, para esta configuração de *cluster*, foram muito demorados, haja vista que este *dataset* é muito esparso e muito grande (muitas transações). Desta forma, as discussões em *clusters* com 1 nodo, reservam-se aos experimentos realizados sobre os *datasets* accidents, connect e T25I15D320k.

Tabela 6.8: Média do tempo (em segundos) de execução do algoritmo UAprioriMR sobre os quatro *datasets*, para diferentes mínimos suportes esperados e número de nodos no *cluster*.

accidents	nodos				
minsupesp	1	5	10	15	20
0.3	663.6	149.1	97.0	80.3	72.0
0.2	1,446.4	313.8	186.7	149.9	130.2
0.1	8,350.4	1,681.7	858.9	600.2	484.8
0.05	40,244.0	7,219.9	3,715.7	2,532.5	1925.7
connect	nodos				
minsupesp	1	5	10	15	20
0.9	29.0	21.1	19.8	19.7	19.2
0.8	284.7	96.3	75.1	70.4	66.9
0.7	1,895.0	393.6	232.1	179.9	161.4
0.6	13,053.7	2,345.9	1,182.9	860.9	784.0
kosarak_10	nodos				
minsupesp	1	5	10	15	20
0.005	—[1]	133.4	102.9	91.9	94.3
0.0025	—	241.5	157.2	128.2	113.4
0.001	—	628.0	351.9	257.4	227.5
0.0001	—	63,482.1	31,646.4	21,061.2	16,007.7
T25I15D320k	nodos				
minsupesp	1	5	10	15	20
0.1	122.0	56.5	50.1	47.8	48.7
0.05	11,540.7	2,371.1	1,229.9	833.9	642.6
0.025	116,950.0	24,530.7	13,063.7	8,429.5	6,440.8
0.01	406,435.0	82,858.4	42,917.8	27,990.0	21,134.7

[1]O símbolo — significa que não foram exibidos os tempos com 1 nodo porque estes experimentos demoraram um tempo muito longo nos testes realizados e não foram concluídos.

Analisando-se os resultados obtidos pelo algoritmo UAprioriMR, conforme demonstrado na Tabela 6.8, fica evidente que o algoritmo tem um comportamento ineficiente quando utiliza-se um *cluster* com somente um nodo. Este comportamento é similar e confirma os testes realizados por Li et al. [LI12], nos quais os autores afirmam ser melhor utilizar um algoritmo sequencial sobre uma máquina sem Apache Hadoop do que utilizar um *cluster* Apache Hadoop com somente 1 nodo. Desta forma, quando utilizados *datasets* com incerteza, também não compensa utilizar o *cluster* com 1 nodo somente.

As Figuras 6.4 (a), (b), (c) auxiliam a visualização de que o algoritmo tem um desempenho ineficiente no *cluster* com somente 1 nodo. À medida que o limiar de suporte é reduzido, o número e o tamanho dos itens frequentes descobertos é maior. Consequentemente, a curva do tempo gasto nos experimentos, pelo algoritmo UAprioriMR, cresce exponencialmente, afastando-se substancialmente das curvas que representam os tempos gastos com 5, 10, 15 e 20 nodos. Este mesmo comportamento acontece para os três *datasets*: accidents, connect e T25I15D320k.

6.4.2 Comportamento do Algoritmo UAprioriMR sobre um *Cluster* com mais de 1 Nodo

À medida que o número de nodos cresce, o desempenho do algoritmo UAprioriMR melhora. As Figuras 6.4 (d), (e), (f) ilustram as mesmas curvas exibidas nas Figuras 6.4 (a), (b) e (c), porém,

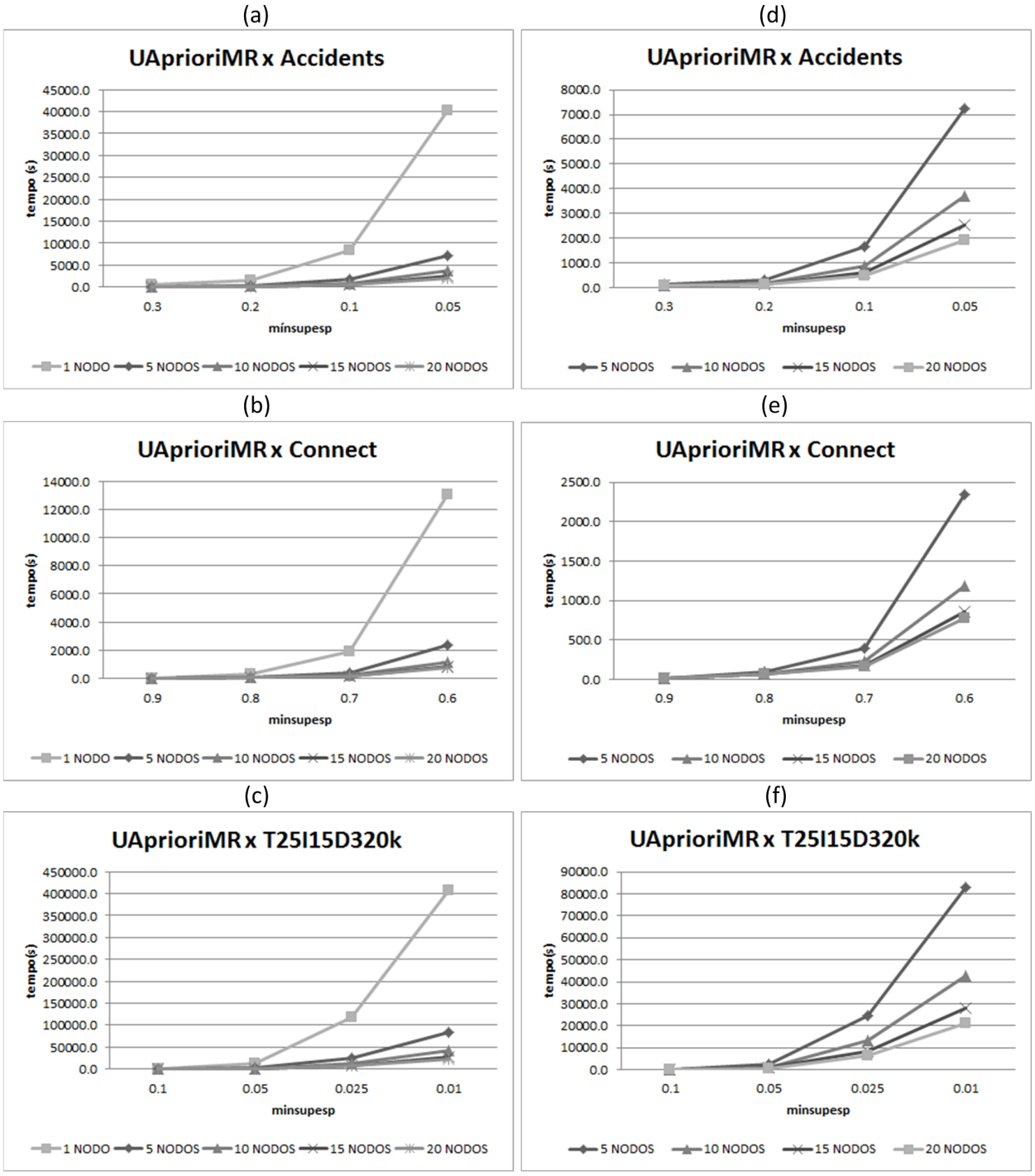

Figura 6.4: (a), (b), (c) - Comportamento do algoritmo UAprioriMR sobre os *datasets* accidents, connect e T25I15D320k em *clusters* de 1, 5, 10, 15 e 20 nodos; (d), (e) e (f) - Comportamento do algoritmo UAprioriMR sobre os *datasets* accidents, connect e T25I15D320k em *clusters* com mais de 1 nodo (5, 10, 15 e 20).

com a exclusão da curva do *cluster* com 1 nodo. Desta forma, é possível visualizar com mais clareza a relação entre os tempos de execução do algoritmo UAprioriMR sobre os demais *clusters*, para os mesmos *datasets*. Fica evidente que o tempo de execução do algoritmo é reduzido à proporção que o número de nodos aumenta.

A Figura 6.5 (a) mostra que ocorre o mesmo comportamento sobre o *dataset* kosarak_10

quando o *cluster* tem 5, 10, 15 e 20 nodos. A Figura 6.5 (b), também sobre o *dataset* kosarak_10, exibe um aumento significativo do tempo de execução do algoritmo para todas as configurações de *cluster* existentes. Isto ocorre porque neste gráfico foi incluído um teste com o mínimo suporte esperado muito baixo, igual a 0.0001. Neste teste são gerados 913 1-itemsets, 1,229 2-itemsets, 414 3-itemsets, 46 4-itemsets e 1 5-itemset frequentes, quantidades de itemsets frequentes muito superiores ao teste com mínimo suporte esperado 0.001, por exemplo, o qual gera 80 1-itemsets, 99 2-itemsets, 25 3-itemsets e 1 4-itemset frequentes.

Observa-se também nas Figuras 6.4 (d), (e), (f) e Figura 6.5 (a), que a distância entre os tempos de execução do algoritmo reduz à medida que o número de nodos cresce. Logo, de acordo com os testes, há um indício de que o tempo de execução do algoritmo tende a encontrar um tempo estável, mesmo com o acréscimo de diversos outros nodos ao *cluster*. Isto deve-se ao fato de a implementação Apache Hadoop necessitar de um *overhead* adicional para gerenciar o *cluster*.

(a) (b)

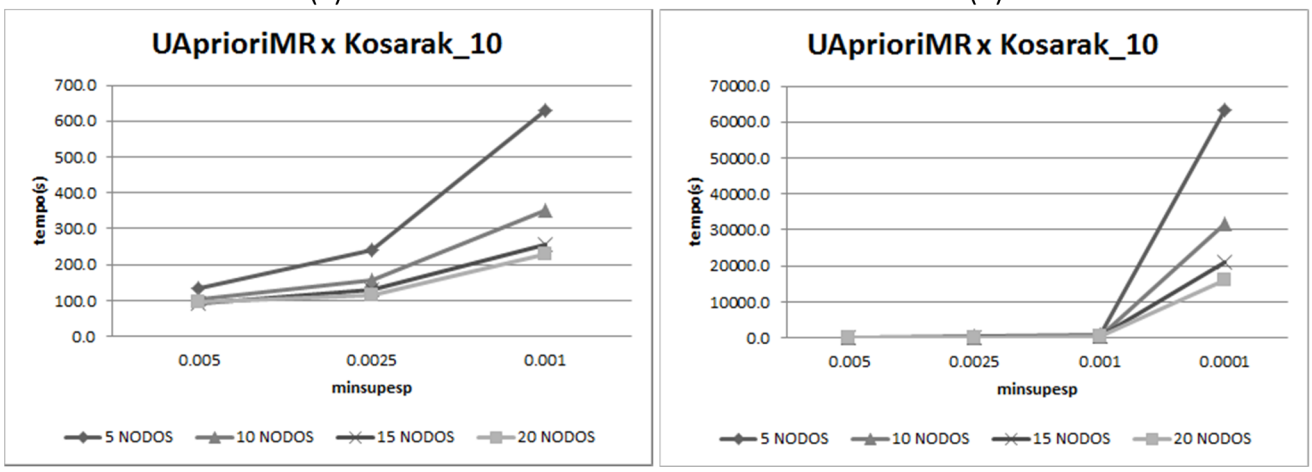

Figura 6.5: Comportamento do algoritmo UAprioriMR sobre o *dataset* kosarak_10.

6.4.3 *Speedup* do Algoritmo UAprioriMR

De acordo com Rauber e Rünger [RAU10], quando analisados programas paralelos, uma comparação com o tempo de execução de uma implementação sequencial é essencial, a fim de visualizar o benefício de utilizar-se o paralelismo. Esta comparação baseia-se frequentemente sobre a métrica denominada *speedup* $S(p)$. O *speedup* é a razão do tempo de execução do algoritmo sequencial $T(1)$ pelo tempo de execução do algoritmo paralelo $T(p)$, conforme mostrado na Equação 6.1. Nesta equação, $T(1)$ significa o tempo de execução do algoritmo sequencial executado em um processador e $T(p)$ denota o tempo de execução do algoritmo paralelo em p processadores.

$$S(p) = \frac{T(1)}{T(p)} \tag{6.1}$$

Para Rauber e Rünger [RAU10], a situação mais típica de um algoritmo paralelo é que ele não atinja um *speedup* linear, $S(p) = p$, pois a implementação paralela requer *overhead* adicional para gerenciamento do paralelismo. Este *overhead* é dado por operações tais como a troca de dados e a sincronização entre os processadores, bem como pelo tempo de espera causado por um balanceamento de carga desigual entre os processadores. Portanto, o algoritmo paralelo, em geral, tem um *speedup* sublinear: $1 < S(p) < p$. Raramente ocorre um *speedup* superlinear, onde $S(p) > p$ e é indesejável que ocorra um *speedup slowdown*, no qual $S(p) < 1$.

A fim de avaliar o *speedup* do algoritmo UAprioriMR, foi desenvolvido o algoritmo UApriori sequencial, baseado no artigo de Chui et al. [CHU07]. Após sua implementação, ele foi executado 10 vezes para cada mínimo suporte escolhido e para os *datasets* accidents, connect e T25I15D320k. O *dataset* Kosarak_10 não é abordado, pois em testes preliminares, os tempos do algoritmo UApriori sequencial foram muito longos. A Figura 6.6 exibe um resumo do *speedup* do algoritmo UAprioriMR em relação ao algoritmo UApriori sequencial.

Accidents	minsupesp			
nodos	0.3	0.2	0.1	0.05
5	3.6	3.8	4.4	4.7
10	5.6	6.4	8.5	9.1
15	6.7	7.9	12.2	13.3
20	7.5	9.1	15.1	17.5

Connect	minsupesp			
nodos	0.9	0.8	0.7	0.6
5	0.2	1.8	3.8	4.7
10	0.2	2.3	6.5	9.4
15	0.2	2.5	8.4	12.9
20	0.2	2.6	9.4	14.1

T25I15D320k	minsupesp			
nodos	0.1	0.05	0.025	0.01
5	1.1	4.7	4.8	4.9
10	1.2	9.0	9.0	9.6
15	1.3	13.3	13.9	14.6
20	1.3	17.2	18.2	19.4

Figura 6.6: *Speedup* do algoritmo UAprioriMR em relação ao UApriori sequencial, sobre os *datasets* accidents, connect e T25I15D320k.

Analisando-se a Figura 6.6 e os gráficos das Figuras 6.7 (a), (b) e (c), observa-se, com exceção do *dataset* connect e minsupesp igual a 0.9, que o algoritmo UAprioriMR apresenta um *speedup* sublinear, $1 < S(p) < p$, para todos os *datasets*, e todos os suportes mínimos esperados, em qualquer configuração de *cluster*. Os gráficos também auxiliam a percepção de que, à medida que o mínimo suporte esperado diminui e, portanto, mais itemsets frequentes são gerados, o *speedup* do algoritmo UAprioriMR cresce e aproxima-se de um *speedup* linear, principalmente nos experimentos realizados sobre o *dataset* maior e mais denso T25I15320k.

O *dataset* connect, com minsupesp igual a 0.9, sobre qualquer configuração de *cluster*, tem um *speedup slowdown* ($S(p) < 1$). Isto ocorre porque neste caso os algoritmos terminam sua execução já no primeiro passo, pois o minsupesp é alto e não são gerados itemsets frequentes de tamanho 1. Logo, os algoritmos terminam rapidamente sua execução. Nestes casos, o algoritmo UApriori sequencial tem melhor desempenho do que um algoritmo paralelo como o UAprioriMR.

Desta forma, os experimentos realizados com o algoritmo UAprioriMR, sobre *datasets* com incerteza, revelam que, assim como o algoritmo SPC [LIN12b] [LI12], que roda sobre *datasets* determínisticos, ele tem um desempenho muito ruim quando utilizado somente 1 nodo no *cluster*. No entanto, à medida que mais nodos são adicionados ao *cluster*, o desempenho do algoritmo tem uma melhora significativa, aproximando-se, quando utilizados suportes mínimos baixos, de um *speedup* linear.

6.5 Experimentos sobre o Algoritmo UAprioriMRByT

Nesta seção é feita uma análise sobre os experimentos realizados com o algoritmo UAprioriMRByT, que gera conjuntos candidatos a cada transação lida. Com este algoritmo foram realizados experimentos preliminares sobre os *datasets* accidents e T25I15D320k. A Tabela 6.9 resume o tempo gasto pelos experimentos usando o algoritmo UAprioriMRByT sobre os dois *datasets* citados e para as configurações de *cluster* com 1, 5, 10, 15 e 20 nodos. A partir do resultado destes experimentos, observou-se que o algoritmo UAprioriMRByT apresenta um desempenho ruim quando aplicado sobre o *dataset* accidents e um bom tempo de resposta quando aplicado sobre o *dataset* T25I15D320k, em relação ao algoritmo UAprioriMR.

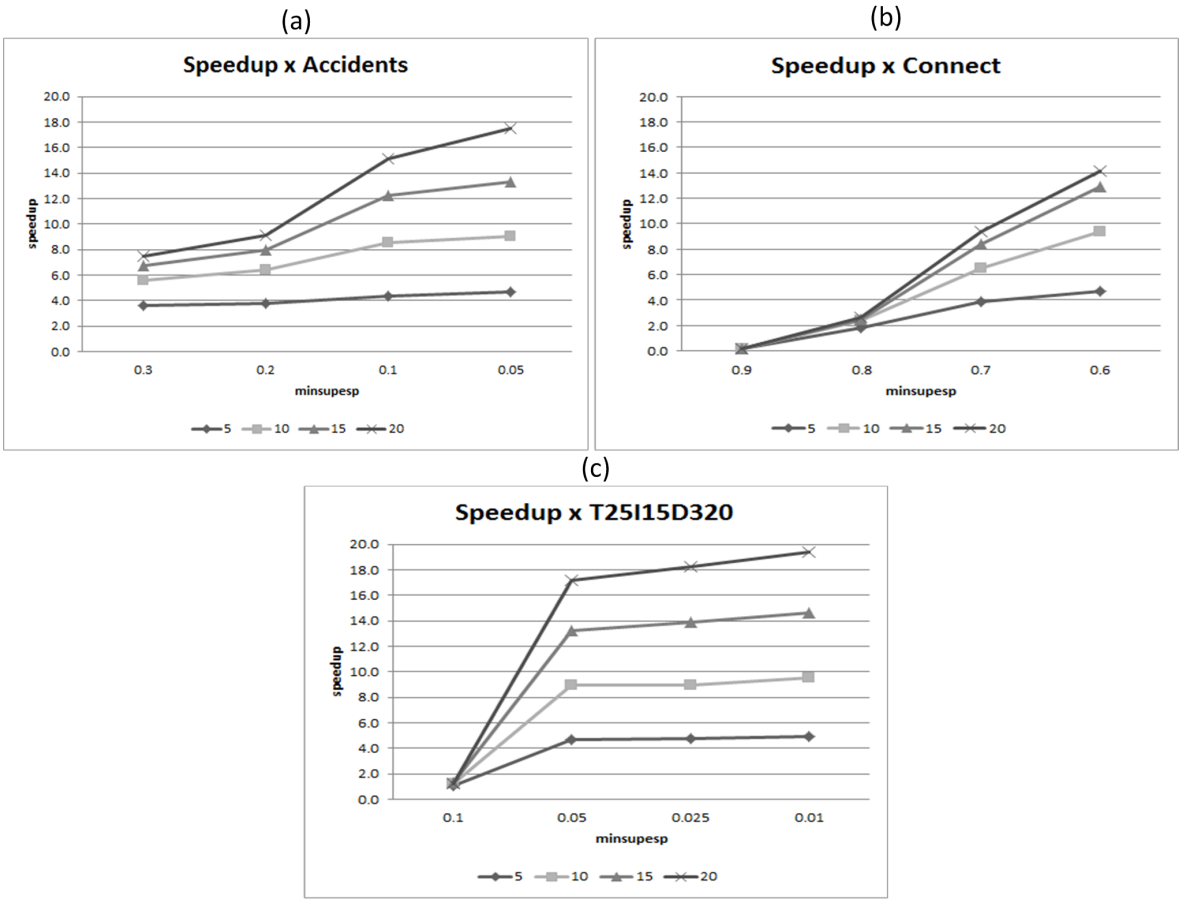

Figura 6.7: Gráficos com o *speedup* do algoritmo UAprioriMR em relação ao UApriori sequencial, sobre os *datasets* accidents, connect e T25I15D320k.

6.5.1 Desempenho do Algoritmo UAprioriMRByT sobre o *Dataset* Accidents

É possível observar na Tabela 6.9 que, sobre o *dataset* accidents, quando o mínimo suporte esperado é igual a 0.05, os tempos de execução foram muito longos e os experimentos não foram concluídos. Também pode-se notar no gráfico apresentado na Figura 6.8 que, para este *dataset*, o algoritmo UAprioriMRByT foi bastante ineficiente, obtendo um tempo de resposta maior do que o algoritmo UAprioriMR, à medida que os suportes mínimos esperados são reduzidos.

O *dataset* accidents tem 340,183 transações, 468 itens distintos e uma média de 33.8 itens por transação. Tome-se como exemplo a execução dos algoritmos UAprioriMR e UAprioriMRByT sobre este *dataset*, utilizando o mínimo suporte esperado igual a 0.1. O número de itemsets gerados, nos passos $k = 2$ e $k = 3$, após a aplicação de ambos algoritmos sobre o *dataset* accidents está detalhada a seguir e pode ser visualizada de modo resumido na Tabela 6.10.

Tome-se como exemplo a execução do algoritmo UAprioriMR sobre o *dataset* accidents. No passo $k = 1$, são gerados em L_1 48 itemsets frequentes. Como não há poda no passo $k = 2$ do algoritmo UAprioriMR, conforme já explicado na Seção 5.3, ele gera 1,128 itemsets no conjunto candidato C_{2p}, aplicando-se a Equação 5.1: $C_{48,2} = \frac{48!}{(48-2)! \cdot 2!} = 1,128$. Desta forma, as funções Map do algoritmo UAprioriMR geram 383,726,424 elementos para serem testados $(340,183 \cdot 1,128)$. Dos 1,128 itemsets em C_{2p}, e após a avaliação dos 383,726,424 elementos, somente 294 itemsets são frequentes em L_2.

Da mesma forma, no passo $k = 3$ são gerados 4,192,244 itemsets em C_3, $C_{294,3} = \frac{294!}{(294-3)! \cdot 3!} = 4,192,244$. No entanto, a partir deste passo, a poda dos itemsets em C_3 ocorre, conforme poda

Tabela 6.9: Média do tempo (em segundos) de execução do algoritmo UAprioriMRByT sobre os *datasets* accidents e T25I15D320k, para diferentes mínimos suportes esperados e número de nodos no *cluster*.

accidents	nodos				
minsupesp	1	5	10	15	20
0.3	709.7	187.0	116.7	98.0	76.0
0.2	20430.3	4253.3	2226.0	1486.0	1138.3
0.1	—	82952.0	40014.0	34886.0	20888.0
0.05	—	—	—	—	—
T25I15D320k	nodos				
minsupesp	1	5	10	15	20
0.1	286.7	95.3	66.0	58.7	59.0
0.05	1,758.7	422.0	219.3	164.0	143.0
0.025	5,158.0	987.0	527.7	374.7	298.3
0.01	137,680.0	15,878.0	7,778.0	5,420.5	4,164.7

[1]O símbolo — significa que não foram exibidos os tempos porque estes experimentos demoraram um tempo muito longo nos testes realizados e não foram concluídos.

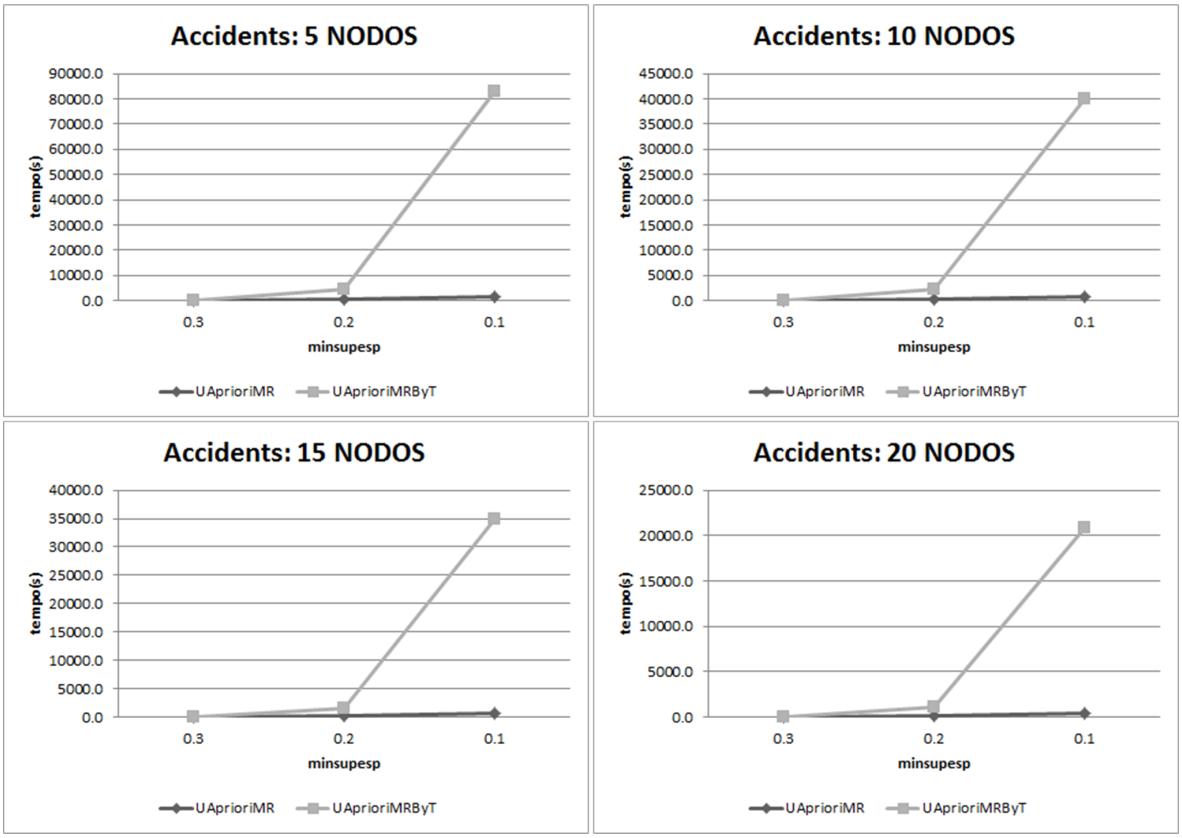

Figura 6.8: Gráficos comparativos com os tempos de execução dos algoritmos UAprioriMR e UAprioriMRByT sobre o *dataset* accidents.

do tradicional algoritmo Apriori e restam somente 1,716 itemsets no conjunto C_{3p}. Este número é 0.04% do total de itemsets do conjunto original C_3, ou seja, a poda neste passo foi extremamente eficiente. Logo, no passo $k = 3$, as funções Map do algoritmo produzem, 583,754,028 elementos $(340,183 \cdot 1,716)$ para serem testados. Dos 1,716 itemsets em C_{3p}, apenas 48 viram itemsets

frequentes em L_3.

Seguindo o mesmo raciocínio, para o passo $k = 4$, são gerados 194,580 itemsets em C_4, $C_{48,4} = \frac{48!}{(48-4)! \cdot 4!} = 194,580$. A poda elimina 99.98% dos itemsets de C_4, restando somente 35 itemsets. Novamente a poda foi bastante eficiente. Desta forma, as funções Map geram 11,906,405 elementos $(340,183 \cdot 35)$ a serem testados. Dos 35 itemsets de C_{4p}, nenhum vira itemset frequente, portanto $L_4 = \emptyset$ e o algoritmo para. A Tabela 6.10 resume os passos e os itemsets gerados pelo algoritmo UAprioriMR.

Tabela 6.10: Exemplo do número de itemsets gerados em L_k[1], C_k[2], C_{kp}[3] e $C_{kp}\ x\ D$[4], durante a aplicação dos algoritmos UAprioriMR e UAprioriMRByT sobre o *dataset* accidents, utilizando $minsupesp = 0.1$.

Algoritmo	$\lvert L_1 \rvert$	$\lvert C_2 \rvert$	$\lvert C_{2p} \rvert$	$\lvert C_{2p}\ x\ D \rvert$	$\lvert L_2 \rvert$	$\lvert C_3 \rvert$	$\lvert C_{3p} \rvert$	$\lvert C_{3p}\ x\ D \rvert$	$\lvert L_3 \rvert$
UAprioriMR	48	1,128	1,128	383,726,424	294	4,192,244	1,716	583,754,028	48
UAprioriMRByT	48	561	561	190,842,663	294	5,984	5,984	2,035,655,072	48

[1] L_k: Conjunto de itens frequentes L, no passo k.
[2] C_k: Conjunto candidato C, no passo k.
[3] C_{kp}: Conjunto candidato C, no passo k, após a aplicação da função de poda.
[4] $|C_{kp}\ x\ D|$: Número de comparações realizadas entre cada elemento do conjunto candidato C_{kp} com todas as transações existentes no *dataset* D.

Observe agora a execução do algoritmo UAprioriMRByT, também sobre o *dataset* accidents e com o mesmo mínimo suporte esperado igual a 0.1. Nota-se que este algoritmo tem um desempenho inferior ao algoritmo UAprioriMR. A Tabela 6.10 também resume os itemsets gerados pelo algoritmo UAprioriMRByT nos passos $k = 2$ e $k = 3$.

No passo $k = 1$, são gerados em L_1 os mesmos 48 itemsets frequentes (idem UAprioriMR). Neste caso, a geração de itemsets de C_2 é realizada por transação e a média de itens por transação neste *dataset* é 33.8 (≈ 34). O conjunto C_2, portanto, gera no máximo 561 itemsets por transação $C_{34,2} = \frac{34!}{(34-2)! \cdot 2!} = 561$. Considere que não houve poda neste passo e C_{2p} também é igual a 561 (todos os 2-itemsets gerados por transação). Consequentemente as funções Map produzem 190,842,663 elementos a serem testados $(340,183 \cdot 561)$. Este número de itemsets em C_{2p} é quase a metade (49.73%) dos itemsets candidatos gerados no mesmo passo do algoritmo UAprioriMR (1,128) e, por este motivo, o algoritmo UAprioriMRByT tem um melhor desempenho no passo $k = 2$, conforme pode ser visualizado na Figura 6.9.

No entanto, este comportamento eficiente do algoritmo UAprioriMRByT verifica-se somente no passo $k = 2$, pois a partir do passo $k > 2$, o algoritmo UAprioriMR aplica a poda tradicional do algoritmo Apriori e tem desempenho melhor do que o algoritmo UAprioriMRByT nestes passos. O passo $k = 2$ produz 294 itemsets frequentes em L_2. Contudo, como o algoritmo UAprioriMRByT gera os itemsets do conjunto candidato C_3 por transação, ele produz 5,984 itemsets em C_3, $C_{34,3} = \frac{34!}{(34-3)! \cdot 3!} = 5,984$, valor 348.72% maior do que os 1,716 itemsets de C_{3p} no algoritmo UAprioriMR. Desta forma, as funções Map do algoritmo UAprioriMRByT geram 2,035,655,072 elementos $(340,183 \cdot\ 5,984)$ no passo $k = 3$, prejudicando substancialmente o tempo de resposta do algoritmo UAprioriMRByT, conforme demonstrado na Figura 6.9. Seguindo o mesmo raciocínio, no passo $k = 4$, o algoritmo UAprioriMRByT também demonstra-se bastante ineficiente.

6.5.2 Desempenho do Algoritmo UAprioriMRByT sobre o *Dataset* T25I15D320k

Ao contrário do comportamento sobre o *dataset* accidents, o algoritmo UAprioriMRByT teve um melhor desempenho do que o algoritmo UAprioriMR, como pode observar-se na Figura 6.10.

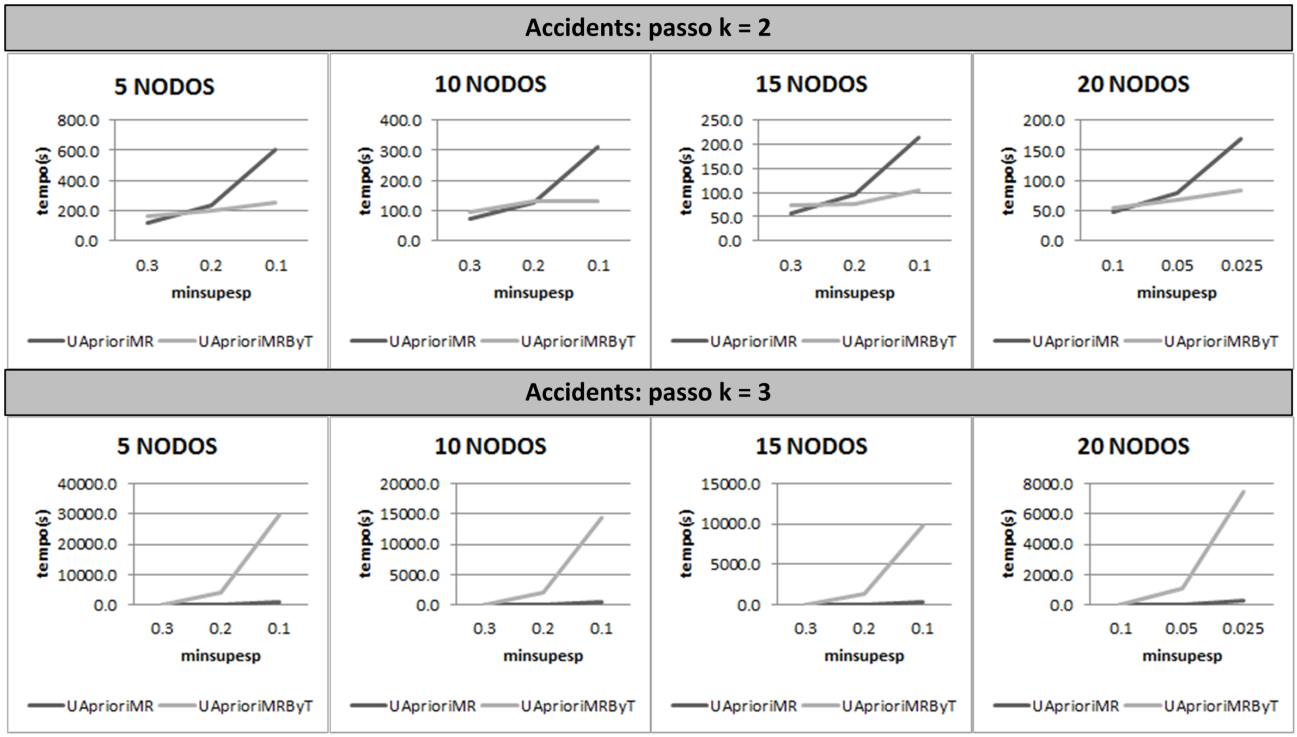

Figura 6.9: Gráficos comparativos com os tempos de execução dos algoritmos UAprioriMR e UAprioriMRByT, nos passos $k = 2$ e $k = 3$, sobre o *dataset* accidents.

Diferente do *dataset* accidents, todos os experimentos com os 4 mínimos suportes esperados distintos foram concluídos.

O *dataset* T25I15D320k tem 320,000 transações, 995 itens distintos e uma média de 26.2 itens por transação. Tome-se como exemplo a execução do algoritmo UAprioriMR sobre o *dataset* T25I15D320k para o mínimo suporte esperado igual a 0.01. O número de itemsets gerados, nos passos $k = 2$ e $k = 3$, após a aplicação de ambos algoritmos sobre o *dataset* T25I15D320k está detalhada a seguir e pode ser visualizada de modo resumido na Tabela 6.11.

Tabela 6.11: Exemplo do número de itemsets gerados em L_k, C_k, C_{kp} e $C_{kp}\ x\ D$, durante a aplicação dos algoritmos UAprioriMR e UAprioriMRByT sobre o *dataset* T25I15D320k, utilizando $minsupesp = 0.01$.

Algoritmo	$\|L_1\|$	$\|C_2\|$	$\|C_{2p}\|$	$\|C_{2p}\ x\ D\|$	$\|L_2\|$	$\|C_3\|$	$\|C_{3p}\|$	$\|C_{3p}\ x\ D\|$	$\|L_3\|$
UAprioriMR	659	216,811	216,811	69,379,520,000	229	1,975,354	434	138,880,000	0
UAprioriMRByT	659	325	325	104,000,000	229	2,600	2,600	832,000,000	0

No passo $k = 1$, são gerados em L_1 659 itemsets frequentes. Como não há poda, no passo $k = 2$ do algoritmo UAprioriMR, ele gera 216,811 itemsets no conjunto candidato C_{2p}, $C_{659,2} = \frac{659!}{(659-2)!\ \cdot\ 2!} = 216,811$. Assim, as funções Map do algoritmo UAprioriMR geram 69,379,520,000 elementos para serem testados $(320,000 \cdot 216,811)$. Dos 216,811 itemsets em C_{2p}, 229 são itens frequentes em L_2. No passo $k = 3$ são gerados 1,975,354 itemsets em C_3, $C_{229,3} = \frac{229!}{(229-3)!\ \cdot\ 3!} = 1,975,354$. No entanto, a partir deste passo, a poda dos itemsets ocorre, conforme propriedade do tradicional algoritmo Apriori e restam somente 434 itemsets no conjunto C_{3p}. Este número é 0.02% do total de itemsets do conjunto C_3, ou seja, a poda neste passo foi extremamente eficiente. Logo, as funções Map do algoritmo produzem 138,880,000 elementos $(320,000 \cdot 434)$ para serem testados no passo $k = 3$. E dos 434 itemsets nenhum vira itemset frequente em L_3.

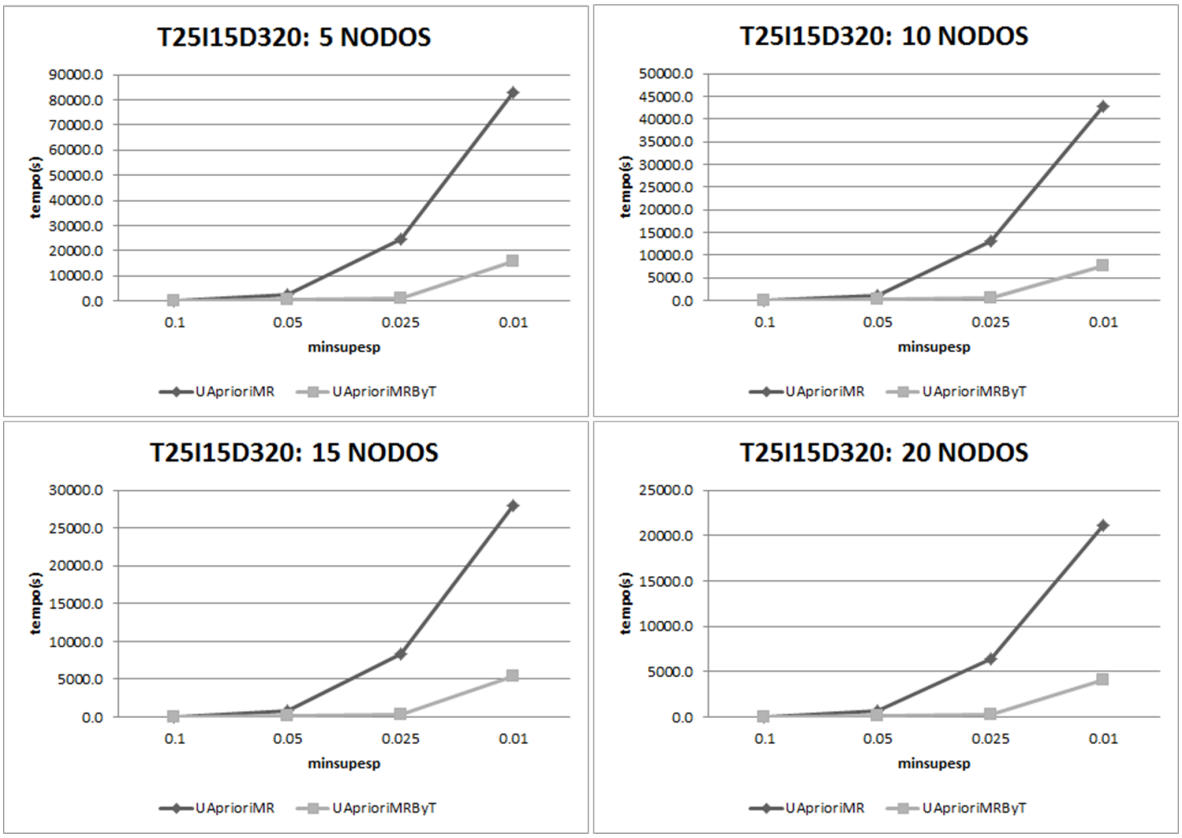

Figura 6.10: Gráficos comparativos com os tempos de execução dos algoritmos UAprioriMR e UAprioriMRByT sobre o *dataset* T25I15D320k.

Observe agora a execução do algoritmo UAprioriMRByT sobre o *dataset* T25I15D320k e para o mesmo mínimo suporte esperado igual a 0.01. Nota-se que ele tem um desempenho superior ao algoritmo UAprioriMR. No passo $k = 1$, são gerados em L_1 os mesmos 659 itemsets frequentes (idem UAprioriMR). Neste caso, a geração de itemsets de C_2 é realizada por transação. A média de itens por transação neste *dataset* é 26.2 (≈ 26). O conjunto C_2, portanto, gera 325 itemsets $C_{26,2} = \frac{26!}{(26-2)! \cdot 2!} = 325$. Considerando que não há poda, $C_{2p} = 325$. Consequentemente as funções Map produzem 104,000,000 elementos a serem testados $(320,000 \cdot 325)$. Este número de elementos é 14.99% dos elementos testados no mesmo passo do algoritmo UAprioriMR e, por este motivo, o algoritmo UAprioriMRByT tem um desempenho bem mais eficiente no passo $k = 2$, conforme pode ser visualizado na Figura 6.11.

No entanto, este comportamento eficiente do algoritmo UAprioriMRByT verifica-se somente no passo $k = 2$, pois a partir do passo $k > 2$ o algoritmo UAprioriMR aplica a poda tradicional do algoritmo Apriori e tem desempenho melhor, nestes passos, do que o algoritmo UAprioriMRByT. O passo $k = 2$ produz os mesmos 229 itemsets frequentes em L_2. Contudo, como o algoritmo UAprioriMRByT gera os itemsets do conjunto candidato C_3 por transação, ele produz 2,600 itemsets em C_3, $C_{26,3} = \frac{26!}{(26-3)! \cdot 3!} = 2,600$, valor 599.08% maior do que os 434 itemsets de C_{3p} produzidos no algoritmo UAprioriMR no mesmo passo. Desta forma, as funções Map do algoritmo UAprioriMRByT geram 832,000,000 elementos $(320,000 \cdot 2,600)$, prejudicando o tempo de resposta do algoritmo UAprioriMRByT no passo $k = 3$, conforme demonstrado na Figura 6.11.

Portanto, pode-se observar pelos experimentos realizados nos *datasets* accidents e T25I15D320k que o algoritmo UAprioriMRByT leva vantagem no passo $k = 2$ em relação ao algoritmo UAprioriMR. Isto ocorre em função deste último algoritmo não aplicar a propriedade de poda neste passo e, por-

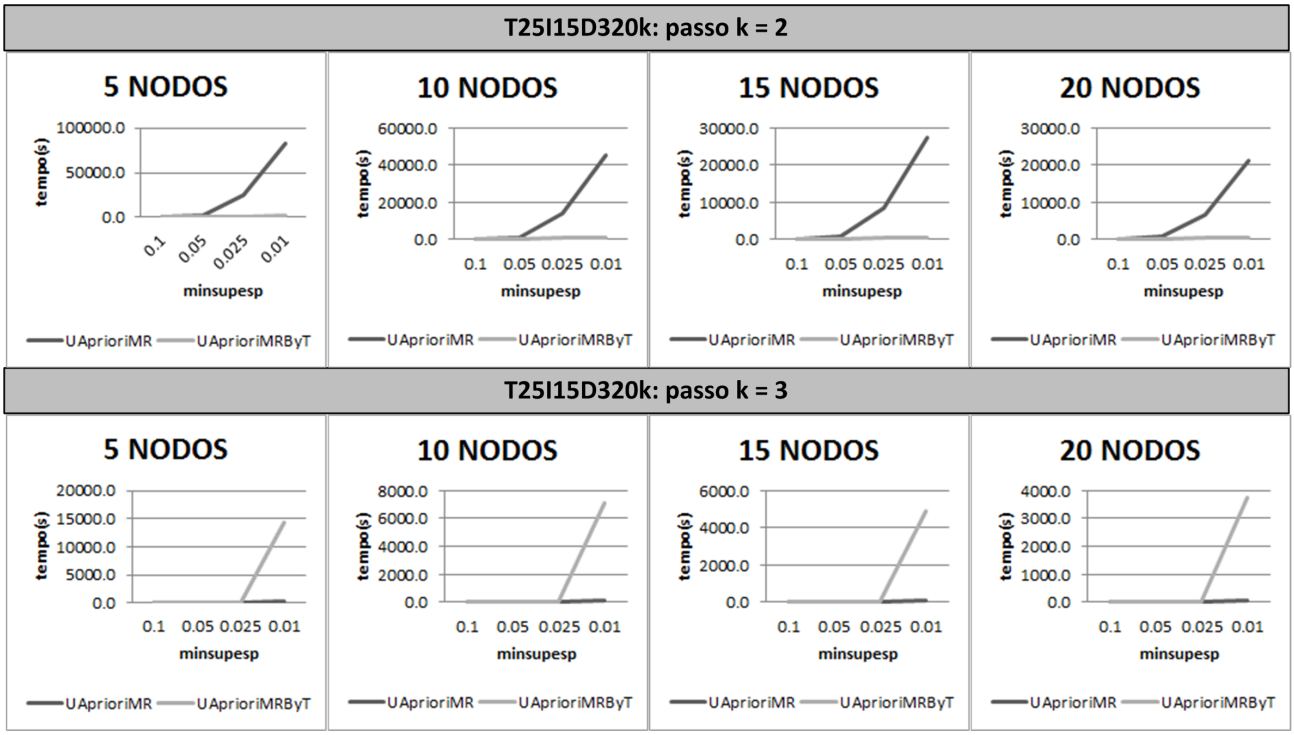

Figura 6.11: Gráficos comparativos com os tempos de execução dos algoritmos UAprioriMR e UAprioriMRByT, nos passos $k = 2$ e $k = 3$, sobre o *dataset* T25I15D320k.

tanto, a função Map gera um número maior de comparações a serem feitas entre os itemsets candidatos e cada transação existente no *dataset*. Por outro lado, quando aplicada a estratégia de geração de itemsets dos conjuntos candidatos por transação, implementada pelo algoritmo UAprioriMRByT, há, para os dois *datasets* testados, um tempo de resposta melhor no passo $k = 2$.

No entanto, para ambos *datasets*, em função da aplicação da poda, o comportamento do algoritmo UAprioriMR foi superior, nos passos $k > 2$, em relação ao algoritmo UAprioriMRByT. Desta forma, tais diferenças de desempenho, nos diferentes passos dos algoritmos motivaram o desenvolvimento e implementação do algoritmo híbrido UAprioriMRJoin e seus experimentos sobre os *datasets* accidents e T25I15D320k e também sobre os *datasets* connect e kosarak_10.

6.6 Experimentos sobre o Algoritmo UAprioriMRJoin

Percebe-se, a partir dos experimentos da seção anterior, que o algoritmo UAprioriMRByT obteve melhor desempenho no passo $k = 2$, enquanto o algoritmo UAprioriMR obteve melhor desempenho em relação aos demais passos ($k > 2$). Quando o *dataset* tem uma média de itens por transação mais baixa do que o número de itens frequentes de L_1 é eficiente utilizar a estratégia do algoritmo UAprioriMRByT no passo $k = 2$. Em caso contrário, utiliza-se a estratégia do algoritmo UAprioriMR.

Esta diferença de desempenho, observada no passo $k = 2$, para os dois *datasets* testados, pode ocorrer para os demais passos, a favor do algoritmo UAprioriMRByT. Isto acontece se a poda realizada pelo algoritmo UAprioriMR não for eficiente o suficiente a ponto de fazer com que o número de itemsets candidatos seja menor do que a estratégia de geração de itemsets por transação. Todavia, nos dois *datasets* testados na seção anterior, a poda sempre foi eficiente, fazendo com que o desempenho do algoritmo UAprioriMR fosse melhor, quando comparado ao algoritmo UAprioriMRByT.

Portanto, o algoritmo híbrido UAprioriMRJoin desenvolvido nesta tese escolhe, a cada passo, qual estratégia utilizar. A cada passo $k > 1$, o algoritmo calcula o número de itemsets dos conjuntos

candidatos que são gerados com base no número de itemsets frequentes do passo anterior (estratégia do UAprioriMR), e compara com o número de itemsets dos conjuntos candidatos gerados, calculados a partir da média de itens por transação (estratégia UAprioriMRByT). Com base nestes dois valores e considerando-se uma constante λ, ilustrada na linha 6 do Algoritmo 5.6, o algoritmo UAprioriMRJoin decide qual estratégia utilizar.

A constante λ é utilizada em função da diferença de desempenho e da diferença do número de itemsets candidatos gerados a cada passo, por cada uma das estratégias. A fim de encontrar o valor de λ, analisou-se o número de itemsets candidatos gerados no passo $k = 2$ dos algoritmos UAprioriMR e UAprioriMRByT. Foram analisadas as quantidades de itemsets candidatos, geradas por ambos algoritmos, quando aplicados sobre o *dataset* accidents, para 5, 10, 15 e 20 nodos, usando diversos mínimos suportes esperados.

Por exemplo, para o *dataset* accidents, com o mínimo suporte esperado igual a 0.3, foram realizados experimentos com 5, 10, 15 e 20 nodos. O algoritmo UAprioriMR gerou 210 elementos no conjunto candidato $C_{k=2}$, no passo $k = 2$, enquanto o algoritmo UAprioriMRByT gerou 561 elementos em $C_{k=2}$. Portanto, o algoritmo UAprioriMR tem aproximadamente 37% de elementos no conjunto $C_{k=2}$, em relação ao algoritmo UAprioriMRByT neste passo k=2. O tempo calculado após a execução destes algoritmos sobre 5, 10, 15 e 20 nodos foi muito semelhante. Fazendo uma relação dos tempos obtidos com os algoritmos UAprioriMR e UAprioriMRByT, para 5, 10, 15 e 20 nodos, foram coletados os seguintes valores 1.0, 0.94, 0.82 e 0.94, respectivamente. Logo, observa-se que mesmo tendo um número maior de elementos candidatos e, portanto, mais comparações a serem realizadas no *dataset* para calcular o suporte esperado de cada elemento, o tempo gasto pelo algoritmo UAprioriMRByT foi semelhante ao tempo gasto pelo algoritmo UAprioriMR.

Realizou-se a mesma observação para o mesmo *dataset* accidents, com mínimo suporte esperado igual a 0.2 e experimentos sobre *clusters* de 5, 10, 15 e 20 nodos. O algoritmo UAprioriMR gerou 435 elementos e o algoritmo UAprioriMRJoin 561 elementos em $C_{k=2}$. Logo, o algoritmo UAprioriMR gerou 78% de elementos do algoritmo UAprioriMRByT no passo $k = 2$. Neste caso, o tempo gasto pelo algoritmo UAprioriMR foi mais alto do que o algoritmo UAprioriMRByT. A relação entre os tempos de execução dos algoritmos UAprioriMR e UAprioriMRByT, em *clusters* de 5, 10, 15 e 20 nodos, foram 1.55, 1.36, 1.42 e 1.35, respectivamente. Desta forma, observa-se que mesmo com menos elementos no conjunto $C_{k=2}$, o algoritmo UAprioriMR levou mais tempo para executar do que o algoritmo UAprioriMRByT. Este mesmo comportamento ocorreu em experimentos com outros mínimos suportes esperados no *dataset* accidents e também com experimentos sobre o *dataset* T25I15D320k.

Observa-se que no passo $k = 2$, mesmo quando o número de itemsets candidatos gerados pelo UAprioriMR é maior do que os itemsets candidatos gerados pelo UAprioriMRByT, o tempo de execução deste último é melhor. No entanto, quando esta diferença entre os itemsets candidatos não é muito grande, o algoritmo UAprioriMR mostrou-se ainda uma melhor estratégia. Portanto, buscou-se identificar qual a melhor relação entre o número de itemsets candidatos gerados pelo algoritmo UAprioriMR e o número de itemsets candidatos produzidos pelo UAprioriMRByT para a escolha da estratégia de execução a ser tomada pelo algoritmo híbrido UAprioriMRJoin. Nos experimentos iniciais relatados anteriormente sobre o *dataset* accidents, o valor encontrado para λ foi igual a 0.4.

As médias dos tempos de execução dos experimentos, para avaliar o desempenho do algoritmo UAprioriMRJoin em relação ao UAprioriMR, foram capturadas sobre 4 *datasets*: accidents, connect, kosarak_10 e T25I15D320k e podem ser visualizadas na Tabela 6.12. Foram executados testes com 4 diferentes tipos de *clusters*: 5, 10, 15 e 20 nodos. Para cada *cluster* e *dataset*, foram usados 4 diferentes tipos de mínimos suportes esperados (minsupesp). Portanto, considerando-se o *dataset*, *cluster* e minsupesp, os experimentos têm 64 configurações diferentes. Para cada uma

Tabela 6.12: Média do tempo (em segundos) de execução do algoritmo UAprioriMRJoin sobre os quatro *datasets*, para diferentes mínimos suportes esperados e número de nodos no *cluster*.

accidents	nodos				
minsupesp	1	5	10	15	20
0.3	—[1]	150.9	102.0	81.9	74.0
0.2	—	230.4	153.4	122.9	109.3
0.1	—	1215.6	670.1	474.2	389.7
0.05	—	5899.6	3101.2	2113.7	1618.9
connect	nodos				
minsupesp	1	5	10	15	20
0.9	—	20.3	19.5	19.8	20.9
0.8	—	93.3	76.9	69.8	74.1
0.7	—	242.4	160.9	132.0	133.8
0.6	—	742.4	430.0	319.5	269.9
kosarak_10	nodos				
minsupesp	1	5	10	15	20
0.005	—	65.3	60.6	59.8	58.4
0.0025	—	89.4	82.1	77.5	79.4
0.001	—	100.7	88.6	84.3	83.2
0.0001	—	456.9	300.2	243.8	211.4
T25I15D320k	nodos				
minsupesp	1	5	10	15	20
0.1	—	57.8	49.3	48.8	49.2
0.05	—	306.7	175.9	135.8	114.6
0.025	—	718.5	391.7	286.8	233.6
0.01	—	1179.9	637.6	470.3	385.7

[1]O símbolo — significa que não foram realizados experimentos sobre *cluster* com 1 nodo, em função de os algoritmos UAprioriMR e UAprioriMRByT já terem evidenciado um desempenho ruim para esta configuração de *cluster*.

destas configurações foram executados 10 testes, totalizando portanto, 640 testes com o algoritmo UAprioriMR e outros 640 testes com o algoritmo UAprioriMRJoin. Ao final destes testes pôde-se discutir a análise estatística do algoritmo UAprioriMRJoin em relação ao algoritmo UAprioriMR, comparar os gráficos dos seus tempos de execução, bem como, fazer um comparativo sobre o *speedup* destes algoritmos.

6.6.1 Análise Estatística

Sobre os experimentos realizados, foram executados testes estatísticos, a fim de concluir se os dois algoritmos, UAprioriMR e UAprioriMRJoin, têm seus tempos de execução estatisticamente diferentes ou se suas diferenças ocorreram meramente ao acaso. Desta forma, é possível comparar o algoritmo UAprioriMRJoin, proposto neste trabalho, com o algoritmo UAprioriMR.

De acordo com Doane e Seward, em [DOA14], os testes estatísticos amostrais, baseados em duas amostras, comparam os parâmetros de duas populações por meio de suas estimativas amostrais e, desta forma, permitem que se façam inferências sobre as duas populações das quais as amostras são extraídas. Para o mesmo autor, quando os tamanhos das amostras são iguais, é maior também o poder do teste estatístico. Nos experimentos deste trabalho, as amostras dos algoritmos

UAprioriMRJoin e UAprioriMR têm o mesmo tamanho (10 testes para cada configuração de *dataset*, *cluster* e minsupesp).

J. Demšar, em [DEM06], afirma que o *teste de Wilcoxon signed-ranks* é uma alternativa não paramétrica, estatisticamente segura e robusta, em relação ao *teste t pareado*, ao fazer testes estatísticos comparativos entre dois classificadores. Ao invés de avaliar as diferenças entre dois classificadores, nesta tese são avaliadas as diferenças entre dois algoritmos de Análise de Associação, a partir do teste não paramétrico de *Wilcoxon*. Nesta tese, portanto, foram geradas as hipóteses abaixo, utilizando-se um nível de confiança igual a 95%, ou seja, um nível de significância de 5% ($\alpha = 0.05$).

Hipótese Nula, H_0: Não há diferença significativa entre as médias obtidas com ambos algoritmos, ou seja, a média de tempo de execução do algoritmo UAprioriMRJoin (μ_0) é igual a média de tempo de execução do algoritmo UAprioriMR (μ_1): ($\mu_0 = \mu_1$).

Hipótese Alternativa, H_1: Há diferença significativa entre as médias obtidas com ambos algoritmos, ou seja, a média de tempo de execução do algoritmo UAprioriMRJoin (μ_0) é diferente da média de tempo de execução do algoritmo UAprioriMR (μ_1): ($\mu_0 \neq \mu_1$).

Um teste de *Wilcoxon* foi executado sobre cada configuração de experimentos, com o propósito de avaliar a hipótese nula criada. Ou seja, para o *dataset* accidents, com 5 nodos e com mínimo suporte esperado igual a 0.3, foi executado um teste de *Wilcoxon* entre os dois algoritmos. Com o mesmo *dataset* e número de nodos também foram realizados testes estatísticos para os mínimos suportes esperados 0.2, 0.1 e 0.05. A mesma lógica seguiu-se para os demais *clusters* e *datasets*, gerando, ao final, 64 testes de *Wilcoxon*, um para cada configuração de experimento existente. Assim, foi gerada a Tabela 6.13, que ilustra se não houve (não) ou houve (sim) diferença estatística, para cada uma das 64 configurações de experimentos, entre as médias dos tempos de execução dos algoritmos UAprioriMRJoin e UAprioriMR.

Tabela 6.13: Resultado da significância estatística, aplicando-se o teste de *Wilcoxon*, para cada configuração de *cluster* e mínimo suporte esperado, sobre os *datasets* accidents, connect, kosarak_10 e T25I15D320k.

Accidents				
	Mínimo Suporte Esperado			
Nodos	0.3	0.2	0.1	0.05
5	Não	Sim	Sim	Sim
10	Não	Sim	Sim	Sim
15	Não	Sim	Sim	Sim
20	Não	Sim	Sim	Sim

Connect				
	Mínimo Suporte Esperado			
Nodos	0.9	0.8	0.7	0.6
5	Não	Não	Sim	Sim
10	Não	Não	Sim	Sim
15	Não	Não	Sim	Sim
20	Não	Não	Sim	Sim

Kosarak_10				
	Mínimo Suporte Esperado			
Nodos	0.005	0.0025	0.001	0.0001
5	Sim	Sim	Sim	Sim
10	Sim	Sim	Sim	Sim
15	Sim	Sim	Sim	Sim
20	Sim	Sim	Sim	Sim

T25I15D320k				
	Mínimo Suporte Esperado			
Nodos	0.1	0.05	0.025	0.01
5	Não	Sim	Sim	Sim
10	Não	Sim	Sim	Sim
15	Não	Sim	Sim	Sim
20	Não	Sim	Sim	Sim

Constata-se, por meio da Tabela 6.13, que na maioria dos testes estatísticos realizados (48 dos 64 = 75%) há diferença estatisticamente significativa entre a média dos tempos de execução dos

algoritmos UAprioriMRJoin e UAprioriMR. Ou seja, nestes 75% dos casos, a hipótese nula (H_0) é refutada, em favor de H_1. Apenas em 25% dos casos não houve diferença estatística entre as médias dos tempos de execução dos dois algoritmos.

Observa-se no *dataset* accidents que, quando o mínimo suporte esperado é igual a 0.3, para qualquer configuração de *cluster*, não há diferença estatisticamente significativa entre os dois algoritmos. Isto ocorre porque quando aplicado este minsupesp são gerados somente 21 itemsets frequentes no passo 1 e nenhum itemset frequente no passo 2. Com este número baixo de 1-itemsets frequentes, a média do tempo de execução gasto para geração dos itens candidatos a 2-itemsets é similar, tanto utilizando o algoritmo UAprioriMRJoin, quanto o UAprioriMR, fazendo com que não haja diferença significativa entre as médias dos seus tempos de execução.

No entanto, quando o mínimo suporte esperado é reduzido para 0.2, 0.1 e 0.05 há diferença estatisticamente significativa entre as médias dos tempos de execução dos dois algoritmos. Quando o minsupesp é 0.2, por exemplo, são gerados 30 1-itemsets e 34 2-itemsets frequentes. Para minsupesp igual a 0.1, são gerados 48 1-itemsets, 294 2-itemsets e 48 3-itemsets frequentes. Quando utiliza-se minsupesp igual a 0.05, são produzidos 75 1-itemsets, 624 2-itemsets, 1451 3-itemsets e 35 4-itemsets frequentes. Os gráficos da Figura 6.12 mostram que, à medida que o mínimo suporte esperado é reduzido e, por consequência, o número de itemsets frequentes cresce, o algoritmo UAprioriMRJoin tem melhor desempenho em relação ao algoritmo UAprioriMR.

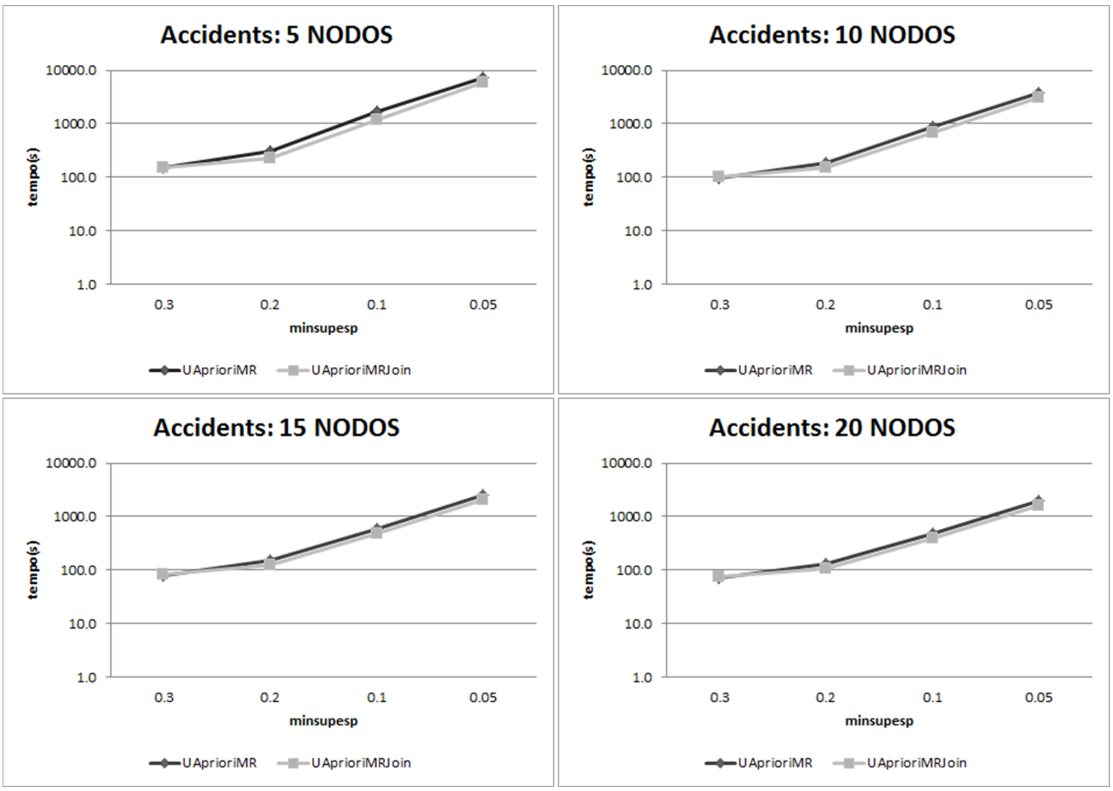

Figura 6.12: Gráficos comparativos entre as médias dos tempos de execução, em escala logarítmica, dos algoritmos UAprioriMR e UAprioriMRJoin sobre o *dataset* accidents.

Quanto aos testes estatísticos realizados sobre o *dataset* connect, observa-se que não houve significância estatística em dois dos quatro mínimos suportes esperados aplicados. Não houve diferença estatística quando utilizados os mínimos suportes esperados 0.9 e 0.8. No caso do minsupesp igual a 0.9 não foi gerado nenhum 1-itemset frequente e, portanto, os dois algoritmos concluíram sua tarefa já no primeiro passo, demorando poucos segundos (≈ 20 segundos para os *clusters* de 5,10,15 e 20 nodos). Para minsupesp igual a 0.8 foram gerados apenas 23 1-itemsets e 6 2-itemsets frequentes e

os tempos de execução de ambos algoritmos foram semelhantes em qualquer configuração de *cluster* também. Contudo, à medida que o número de itemsets frequentes cresce a cada passo dos algoritmos, o que ocorre para os mínimos suportes esperados iguais a 0.7 e 0.6, os testes de *Wilcoxon* mostram que há diferença estatística na média dos tempos de execução entre os dois algoritmos. Para minsupesp igual a 0.7 são gerados 29 1-itemsets, 230 2-itemsets e 145 3-itemsets frequentes. Já para minsupesp igual a 0.6 são produzidos 32 1-itemsets, 385 2-itemsets, 1699 3-itemsets e 1440 4-itemsets frequentes. A Figura 6.13 ilustra o melhor desempenho do algoritmo UAprioriMRJoin, à medida que o mínimo suporte esperado é reduzido.

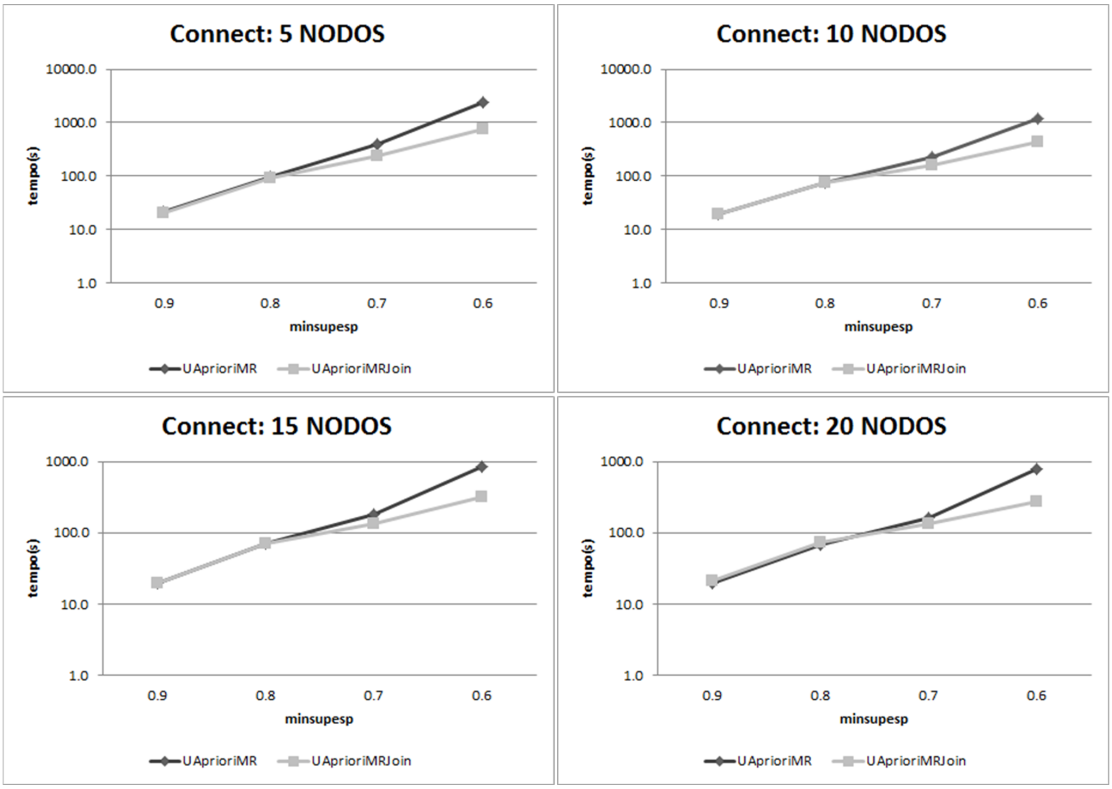

Figura 6.13: Gráficos comparativos entre as médias dos tempos de execução, em escala logarítmica, dos algoritmos UAprioriMR e UAprioriMRJoin sobre o *dataset* connect.

Percebe-se que, para o *dataset* kosarak_10, em todas as configurações de mínimos suportes esperados e número de nodos, houve diferença estatisticamente significativa entre as médias dos tempos de execução dos dois algoritmos. O gráfico da Figura 6.14 mostra que o algoritmo UAprioriMRJoin tem melhor desempenho para todos os mínimos suportes esperados e número de nodos utilizados. Quando o mínimo suporte esperado é igual a 0.005, são gerados 26 1-itemsets, 14 2-itemsets e 2 3-itemsets frequentes. Para minsupesp igual a 0.0025, são produzidos 45 1-itemsets, 41 2-itemsets e 4 3-itemsets frequentes. Com minsupesp igual a 0.001, são gerados 80 1-itemsets, 99 2-itemsets, 25 3-itemsets e 1 4-itemset frequentes. E, finalmente, para o mínimo suporte esperado igual a 0.0001, são gerados 913 1-itemsets, 1229 2-itemsets, 414 3-itemsets, 46 4-itemses e 1 5-itemset frequentes. Da mesma maneira, à medida que os mínimos suportes esperados reduzem e o número de itemsets frequentes aumentam, o desempenho do algoritmo UAprioriMRJoin é melhor do que o algoritmo UAprioriMR.

Ao observar o *dataset* sintético T25I15D320k, acontece um comportamento análogo aos outros 3 *datasets* testados. À medida que o mínimo suporte esperado diminui, é possível constatar que há diferença estatisticamente significativa quanto as médias dos tempos de execução dos dois algoritmos. O gráfico da Figura 6.15 exibe que o algoritmo UAprioriMRJoin tem melhor desempenho

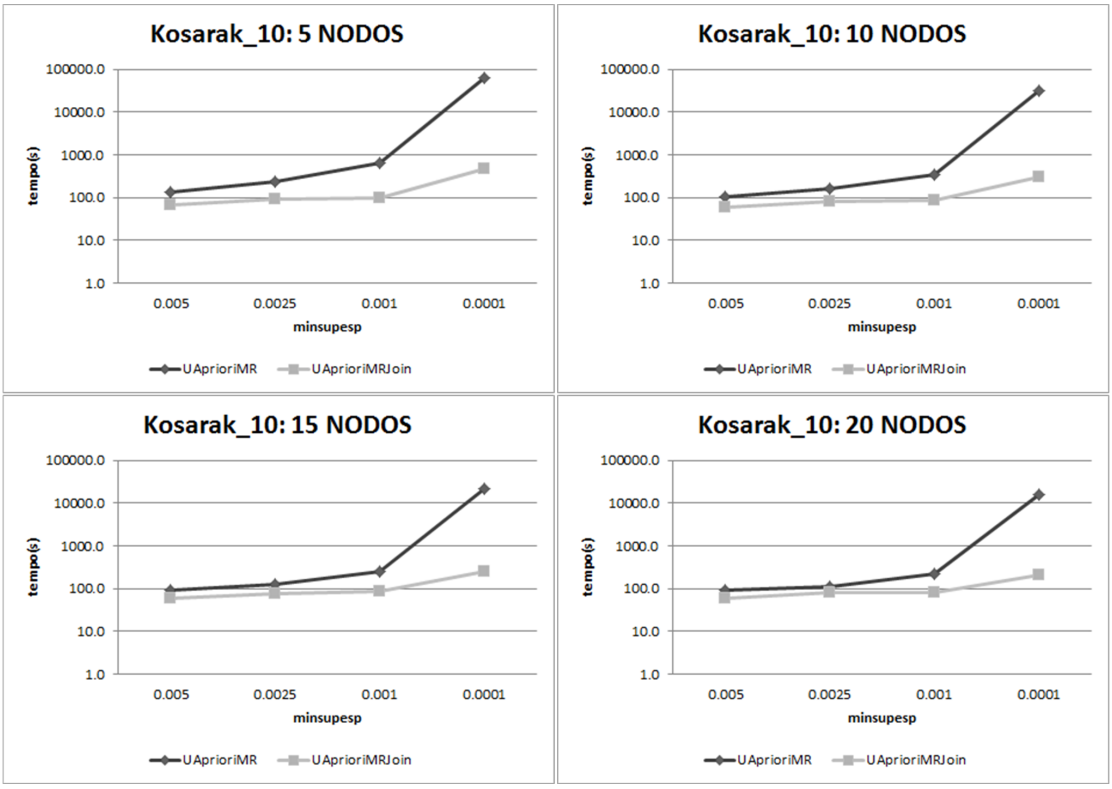

Figura 6.14: Gráficos comparativos entre as médias dos tempos de execução, em escala logarítmica, dos algoritmos UAprioriMR e UAprioriMRJoin sobre o *dataset* kosarak_10.

em relação ao algoritmo UAprioriMR para o *dataset* T25I15D320k. Não há diferença estatística apenas para o mínimo suporte esperado igual a 0.1. Isto ocorre porque, neste caso, os dois algoritmos geram apenas 8 1-temsets frequentes. Como são poucos 1-itemsets frequentes, as abordagens distintas de geração de itemsets candidatos a 2-itemsets frequentes não causam diferenças nos tempos de execução dos algoritmos, sendo semelhantes tanto para o UAprioriMRJoin, quanto para o UAprioriMR.

No entanto, quando utilizados os mínimos suportes esperados iguais a 0.05 e 0.025, são gerados, respectivamente, 111 e 361 1-itemsets frequentes. Para estes dois casos, é muito mais eficiente utilizar a abordagem de geração de candidatos por transação, já que a média de itens por transação deste *dataset* é de 26.2. Quando o minsupesp é reduzido para 0.01 a diferença de desempenho entre os dois algoritmos fica ainda mais evidente. Neste caso, são gerados 659 1-itemsets e 229 2-itemsets frequentes.

Desta forma, para os quatro *datasets* escolhidos, com tamanhos e densidades distintas, prova-se que há diferença estatísticamente significativa entre as médias dos tempos de execução dos dois algoritmos testados na maioria das configurações utilizadas. Esta diferença evidencia-se à medida que o mínimo suporte esperado é reduzido e, consequentemente, o número de itemsets frequentes aumenta.

6.6.2 *Speedup* do Algoritmo UAprioriMRJoin

Assim como foi realizada uma análise do *speedup* do algoritmo UAprioriMR na Seção 6.4.3, nesta seção é discutida a métrica *speedup* do algoritmo UAprioriMRJoin. Esta métrica também é avaliada com base no algoritmo UApriori sequencial, implementado com base no artigo de Chui et al. [CHU07]. A avaliação é realizada sobre os *datasets* accidents, connect e T25I15D320k. O

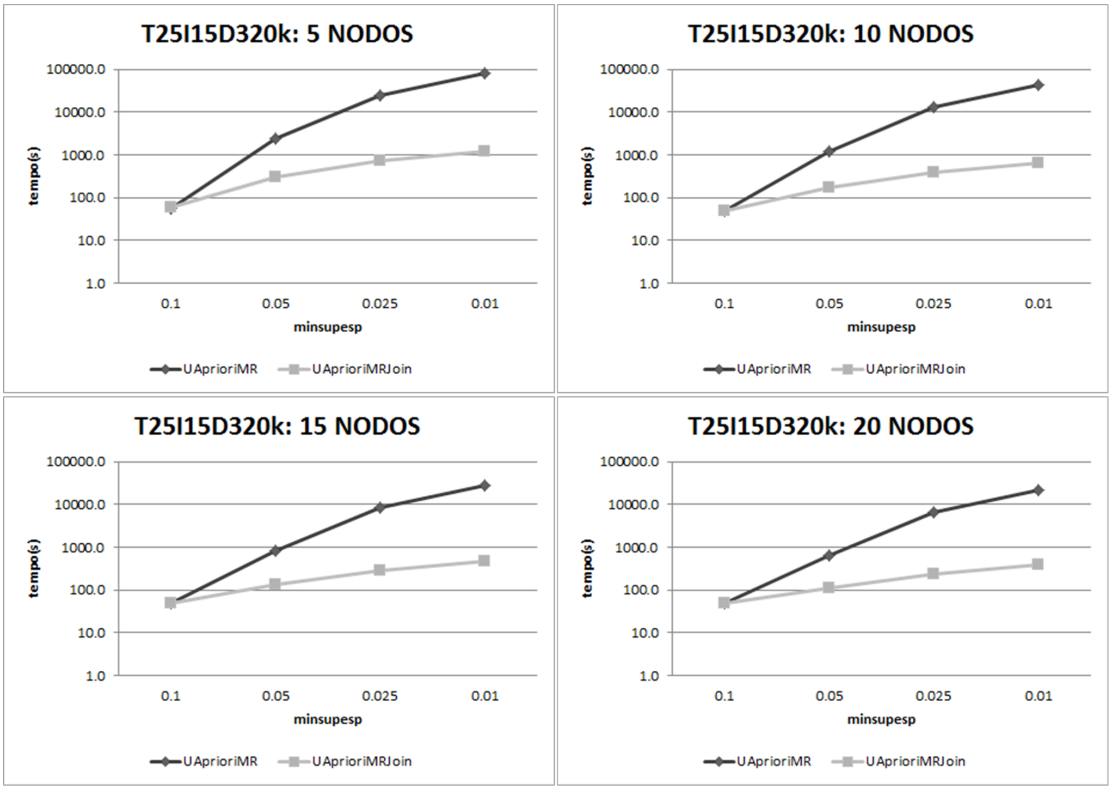

Figura 6.15: Gráficos comparativos entre as médias dos tempos de execução, em escala logarítmica, dos algoritmos UAprioriMR e UAprioriMRJoin sobre o *dataset* T25I15D320k.

dataset kosarak_10 não é abordado, pois em testes preliminares, os tempos do algoritmo UApriori sequencial foram muito longos.

Accidents	minsupesp			
nodos	0.3	0.2	0.1	0.05
5	3.6	5.2	6.0	5.7
10	5.3	7.8	11.0	10.9
15	6.6	9.7	15.5	16.0
20	7.3	10.9	18.8	20.8

Connect	minsupesp			
nodos	0.9	0.8	0.7	0.6
5	0.2	1.9	6.2	14.9
10	0.2	2.3	9.4	25.8
15	0.2	2.5	11.5	34.7
20	0.2	2.4	11.3	41.1

T25I15D320k	minsupesp			
nodos	0.1	0.05	0.025	0.01
5	1.1	36.0	163.2	347.5
10	1.2	62.8	299.4	643.0
15	1.3	81.4	408.9	871.8
20	1.3	96.4	502.1	1063.0

Figura 6.16: *Speedup* do algoritmo UAprioriMRJoin em relação ao UApriori sequencial, sobre os *datasets* accidents, connect e T25I15D320k.

A Figura 6.16 exibe um resumo da métrica *speedup* do algoritmo UAprioriMRJoin em relação ao algoritmo UApriori sequencial. Observa-se nesta figura que, à medida que o mínimo suporte esperado é reduzido, cresce o *speedup* do algoritmo UAprioriMRJoin. Em todos os *datasets*, para o menor mínimo suporte esperado utilizado, em todas as configurações de *clusters*, o algoritmo UAprioriMRJoin apresenta um *speedup* superlinear, ou seja, $S(p) > p$.

Chama a atenção o *dataset* T25I15D320k na Figura 6.16. Neste *dataset*, o algoritmo híbrido UAprioriMRJoin tem uma métrica de *speedup* superlinear bastante expressiva, quando utilizados os mínimos suportes esperados 0.05, 0.025 e 0.01. O *dataset* connect, para o mínimo suporte esperado igual a 0.6, também tem uma métrica de *speedup* superlinear bastante expressiva. Nota-se ainda que, nestes dois *datasets*, para os mínimos suportes esperados mencionados, à medida que o número de nodos cresce no *cluster*, a métrica de *speedup* torna-se ainda mais alta.

Este comportamento nestes dois *datasets* reflete a vantagem de utilizar o modelo de programação MapReduce para descobrir itens frequentes pois, como é sabido, um dos grandes gargalos do algoritmo UApriori sequencial são as diversas varreduras sobre o *dataset* analisado. Utilizando-se MapReduce, o *dataset* é dividido em *splits* com base no número de nodos do *cluster*. Portanto, em um *cluster* com 20 nodos, o *dataset* é dividido em 20 *splits* para que as funções Map executem em cada nodo, sobre um determinado *split*. A métrica de *speedup* deixa claro que esta abordagem apresenta benefícios na melhoria da métrica *speedup* do algoritmo UAprioriMRJoin.

Somente para o *dataset* connect, quando utilizado o mínimo suporte esperado igual a 0.9, há um *speedup slowdown* ($S(p) < p$). Isto porque, com este mínimo suporte esperado, nenhum 1-itemset frequente é gerado e o algoritmo termina rapidamente, em cerca de 20 segundos, caracterizando, desta forma, ser melhor o uso do algoritmo UApriori sequencial.

Os gráficos da Figura 6.17 auxiliam a percepção de que, à medida que o mínimo suporte esperado diminui e, portanto, mais itemsets frequentes são gerados, a métrica *speedup* do algoritmo UAprioriMRJoin cresce. E, à medida que mais nodos são adicionados ao *cluster*, a métrica *speedup* do algoritmo UAprioriMRJoin também aumenta.

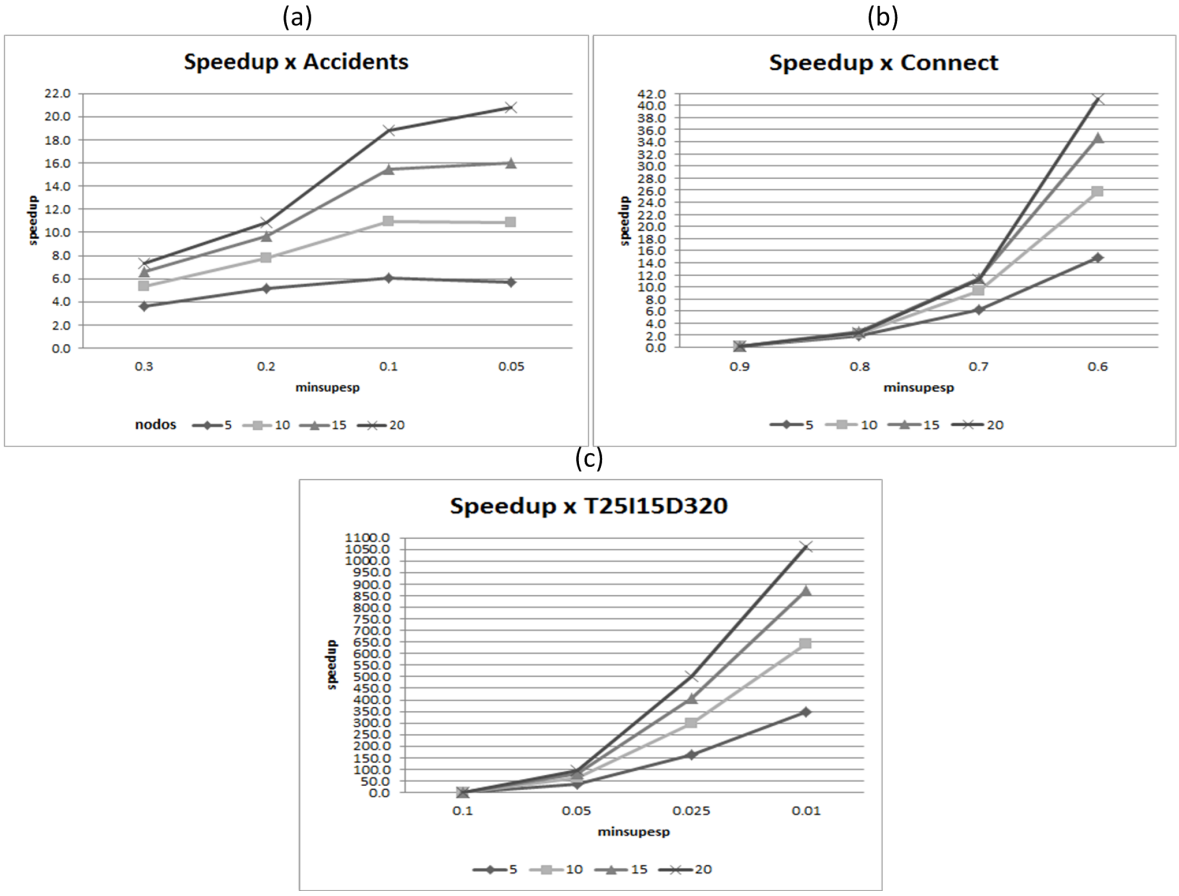

Figura 6.17: Gráficos com o *speedup* do algoritmo UAprioriMRJoin em relação ao UApriori sequencial, sobre os *datasets* accidents, connect e T25I15D320k.

A Figura 6.18 compara a métrica de *speedup* dos dois algoritmos para os três *datasets*: accidents, connect e T25I15D320k, em *clusters* de 5, 10, 15 e 20 nodos. Estes gráficos são os mesmos das Figuras 6.7 e 6.17, porém, agora comparando os algoritmos UAprioriMR e UAprioriMRJoin. Ambos algoritmos têm sua métrica de *speedup* aumentada, à medida que que o mínimo suporte esperado é reduzido. Observa-se também que, à proporção que o *cluster* cresce, a métrica de *speedup* dos dois algoritmos também cresce.

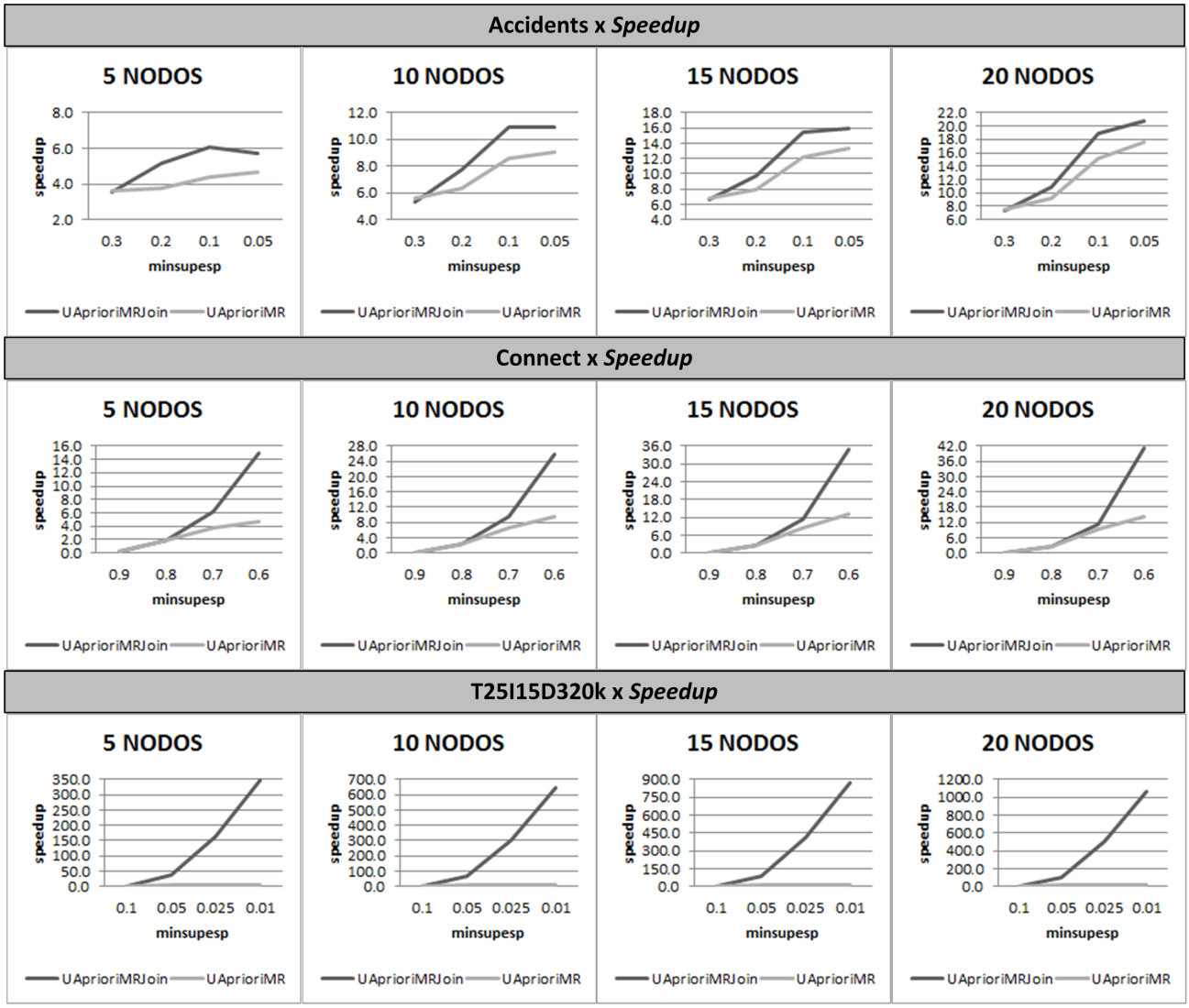

Figura 6.18: Gráficos comparativos com o *speedup* dos algoritmos UAprioriMR e UAprioriMRJoin, sobre os datasets accidents, connect e T25I15D320k, em clusters de 5, 10, 15 e 20 nodos.

A Figura 6.18 mostra que a métrica de *speedup* dos algoritmos UAprioriMRJoin e UAprioriMR é distinta. Na Figura 6.18 é possível observar que a métrica de *speedup* do algoritmo UAprioriMRJoin é maior do que a mesma métrica do algoritmo UAprioriMR na maioria dos casos. Inclusive, para os datasets connect e T25I15D320k, a distância entre as métricas de *speedup* dos dois algoritmos vai ficando maior, à medida que o mínimo suporte esperado é reduzido.

A partir da Figura 6.19 pode-se observar alguns aspectos relacionados ao *speedup* atingido pelos algoritmos UAprioriMR e UAprioriMRJoin em relação ao número de nodos presentes no *cluster*. No *dataset* accidents observa-se o *speedup* sublinear para todos os mínimos suportes esperados.

Quanto ao *dataset* connect, é possível observar que para ambos os algoritmos, quando o suporte mínimo esperado foi alto (0.9) o *speedup* foi *slowdown*. Para os demais suportes mínimos esperados, em ambos os algoritmos, o *speedup* foi sublinear. A exceção é o suporte mínimo esperado igual a 0.6 para o algoritmo UAprioriMRJoin, neste caso o *speedup* foi superlinear.

No *dataset* sintético T25I15D320k, para o algoritmo UAprioriMR, em todos os suportes mínimos esperados, o *speedup* foi sublinear. Neste mesmo *dataset*, o algoritmo UAprioriMRJoin, com exceção do mínimo suporte esperado 0.1, teve *speedup* superlinear.

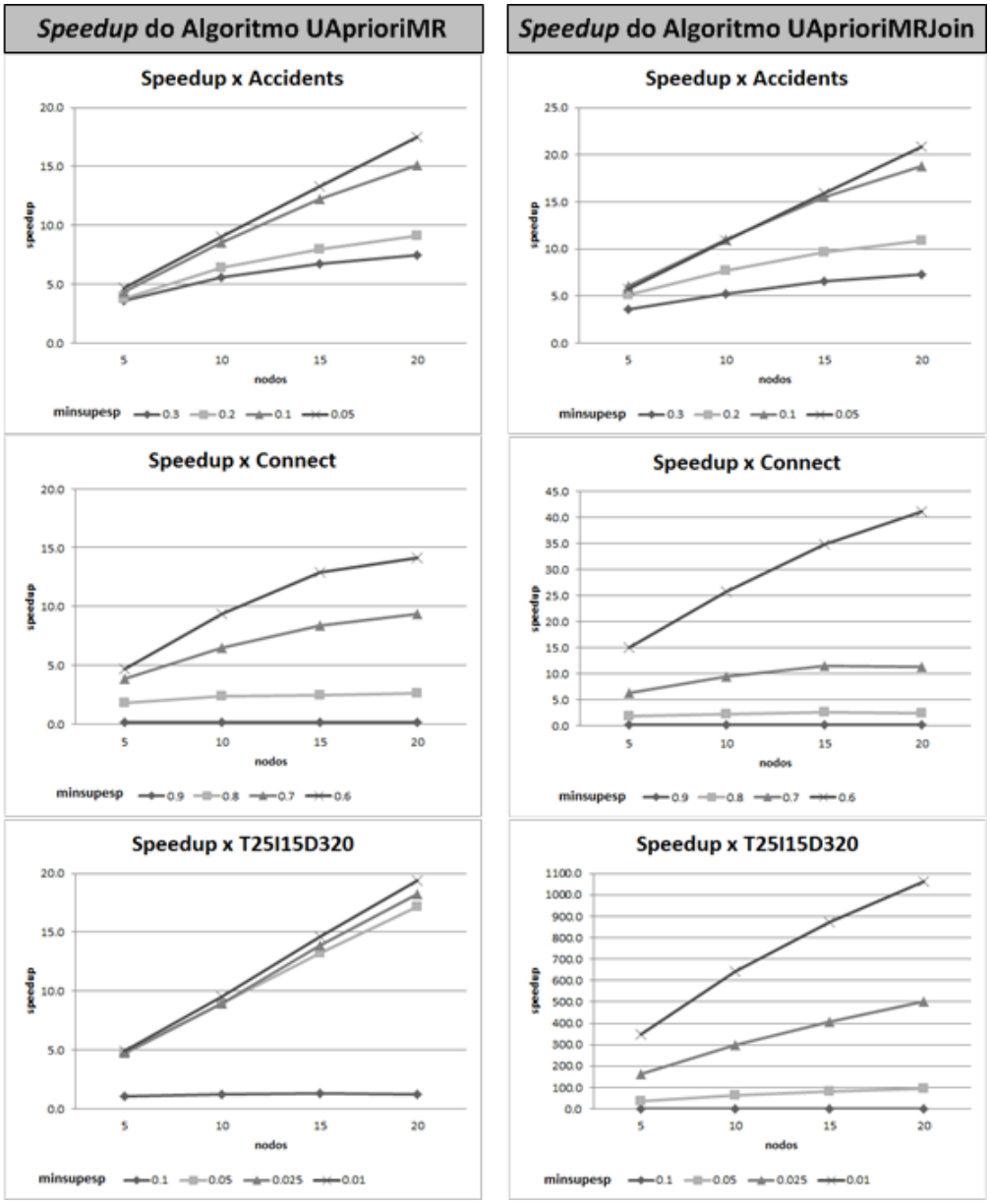

Figura 6.19: Gráficos com o *speedup* atingido em relação ao número de nodos do *cluster*, quando executados os algoritmos UAprioriMR e UAprioriMRJoin, usando os *datasets* accidents, connect e T25I15D320k.

6.6.3 Limitação do Algoritmo UAprioriMRJoin

Como já mencionado na Tabela 6.5, foram realizados 160 testes sobre o *dataset* kosarak, utilizando o algoritmo UAprioriMR. Com os demais algoritmos foram realizados somente alguns testes preliminares. Quando utilizado o algoritmo UAprioriMRByT, que gera conjuntos candidatos a cada transação, foi verificado um problema a partir do uso desta abordagem neste *dataset*.

O *dataset* kosarak é o maior dos conjuntos de dados utilizados nos testes. Ele tem cerca de 990 mil transações e 41 mil itens distintos, embora possua uma média de itens por transação baixa em relação aos demais *datasets*, apenas 8.1. Contudo, este *dataset* tem algumas transações que possuem um número muito grande de itens, diferindo sobremaneira das características apresentadas pelos demais *datasets*. A Tabela 6.14 exibe o número de itens das 5 maiores transações de todos os *datasets*. A Tabela 6.14 deixa evidente que o *dataset* kosarak possui transações muito maiores

do que a média de itens por transação deste *dataset*. Estas transações prejudicam o desempenho do algoritmo UAprioriMRByT e, consequentemente, o algoritmo híbrido UAprioriMRJoin.

Tomando-se como exemplo a maior transação do *dataset* kosarak, 2,498 itens. Se o algoritmo UAprioriMRJoin utilizar a estratégia de, ao varrer esta transação, gerar os itemsets candidatos, muitos itemsets são gerados. Por exemplo, gerar itemsets candidatos de tamanho 2, C_2, implica em produzir 3,118,753 itemsets, $C_{2,498,2} = \frac{2,498!}{(2,498-2)! \cdot 2!} = 3,118,753$, ou gerar itemsets candidatos de tamanho 3, C_3, implica em produzir 2,594,802,496 itemsets, $C_{2,498,3} = \frac{2,498!}{(2,498-3)! \cdot 3!} = 2,594,802,496$. Embora a maioria do *dataset* kosarak, 85.64%, tenha menos de 10 itens por transação, existem cerca de 4300 transações com um número de itens por transação igual ou superior a 100. Este comportamento do *dataset*, possuindo transações com muitos itens, prejudica o funcionamento do algoritmo UAprioriMRJoin, quando escolhida a estratégia de gerar itemsets candidatos por transação. Por este motivo, este *dataset* foi eliminado dos experimentos.

Tabela 6.14: Número total de itens das 5 maiores transações de cada um dos *datasets* estudados.

Dataset	**Quantidade de Itens das 5 Maiores Transações**					**Média de Itens por Transação**
	1^a	2^a	3^a	4^a	5^a	
Accidents	51	50	50	49	49	33.8
Connect	43	43	43	43	43	43
Kosarak	2,498	2,491	2,337	2,108	1,926	8.1
T25I15D320k	67	67	64	64	64	26.2

Desta forma, verifica-se que os bons resultados do algoritmo UAprioriMRJoin em relação ao algoritmo UAprioriMR devem levar em conta a variação do tamanho das transações existentes no *dataset*. A Tabela 6.15 ilustra a estatística descritiva dos *datasets* estudados. Fica evidente a grande variação do tamanho das transações no *dataset* kosarak. O seu coeficiente de variação, por exemplo, é muito alto em relação aos demais *datasets*. Seu coeficiente de variação indica que o desvio padrão do número de itens por transação representa 291.6% da média.

Tabela 6.15: Estatística descritiva dos *datasets*.

Dataset	**Média (μ)**	**Desvio padrão (σ)**	**Coeficiente de variação ($\frac{\mu}{\sigma}$)**	**Menor transação**	**Maior transação**
Accidents	33.8	2.9	8.7%	18	51
Connect	43.0	0.0	0.0%	43	43
Kosarak	8.1	23.6	291.6%	1	2,498
Kosarak_10	3.4	2.1	62.9%	1	10
T25I15D320k	26.2	9.1	34.0%	3	67

Como o *dataset* kosarak é muito grande e tem um número muito grande de itens distintos, características importantes para a análise dos experimentos sobre os algoritmos, foi criado um outro *dataset* a partir dele, o conjunto de dados kosarak_10. Neste *dataset* foram salvas somente as transações com 10 itens ou menos do *dataset* kosarak, eliminando-se todas as transações com mais de 10 itens. Mesmo assim, o *dataset* kosarak_10 ficou sendo o maior *dataset* dos experimentos, com 847,875 transações, das 990,002 existentes no *dataset* kosarak (85.64%). E dos 41,271 atributos distintos existentes em kosarak, restaram 36,506 (88.45%) itens distintos em kosarak_10.

6.7 Considerações Finais do Capítulo

Neste capítulo foram discutidos os experimentos realizados sobre os três algoritmos desenvolvidos e implementados neste trabalho: UAprioriMR, UAprioriMRByT e UAprioriMRJoin. Foram analisados os resultados dos experimentos sob diversos aspectos, a fim de esclarecer as vantagens e desvantagens de utilizar-se estes três algoritmos, enfatizando o algoritmo UAprioriMRJoin.

Inicialmente, apresentam-se os *datasets* sobre os quais foram aplicados os experimentos. Foram relatadas as características dos *datasets* utilizados e, ao longo do capítulo, os detalhes destes foram abordados com o objetivo de elucidar todas as análises realizadas. Além dos *datasets*, o laboratório experimental onde foram executados os experimentos também foi caracterizado, explicitando a montagem dos *clusters* de 1, 5, 10, 15 e 20 nodos. Esta seção é finalizada detalhando a metodologia dos experimentos, ou seja, como eles foram conduzidos.

Abriu-se uma seção para discutir os experimentos realizados sobre cada um dos algoritmos. Inicialmente foi abordado o algoritmo UAprioriMR. Este algoritmo demonstrou uma boa métrica de *speedup*, à medida que o número de nodos no *cluster* aumenta. O algoritmo UAprioriMR apresentou resultados ruins quando utilizado um *cluster* com 1 nodo somente. Este comportamento confirmou resultados similares de trabalhos anteriores [LI12], na utilização do modelo de programação MapReduce para lidar com a descoberta de conjuntos de itens frequentes.

Após o estudo do algoritmo UAprioriMR, foram analisados os resultados dos experimentos do algoritmo UAprioriMRByT sobre dois *datasets*. Os experimentos esclarecem que este algoritmo tem um bom desempenho, principalmente no passo $k = 2$, quando a média de itens por transação do *dataset* é menor do que o número de itemsets frequentes gerados no passo anterior. Quando esta situação ocorre, é gerado um número menor de itemsets candidatos e o algoritmo UAprioriMRByT é mais eficiente do que o algoritmo UAprioriMR. A partir desta constatação, desenvolveu-se e implementou-se o algoritmo híbrido UAprioriMRJoin.

Os experimentos sobre o algoritmo UAprioriMRJoin comprovam que, à medida que o mínimo suporte esperado é reduzido e, consequentemente, o número de itemsets frequentes aumenta, o algoritmo UAprioriMRJoin tem um melhor desempenho do que o algoritmo UAprioriMR. Além disso, o algoritmo UAprioriMRJoin demonstrou uma boa métrica de *speedup*, apresentando, em alguns casos, uma métrica de *speedup* superlinear, quando utilizados baixos mínimos suportes esperados e um número maior de nodos no *cluster*.

Foi apresentada também uma limitação do algoritmo UAprioriMRByT e, por consequência, do algoritmo híbrido UAprioriMRJoin. Estes algoritmos não mostraram-se eficientes ao lidar com *datasets* que têm transações muito grandes. O *dataset* kosarak tinha este perfil e o algoritmo UAprioriMRByT não obteve bons resultados.

Contudo, observando-se os quatros datasets sobre os quais foram realizados os experimentos, o *dataset* T25I15D320k tem a maior transação dentre todos eles: 67 itens. Para este número máximo de itens por transação, o algoritmo UAprioriMRJoin mostrou-se eficiente. Em todos os 4 datasets utilizados, o algoritmo UAprioriMRJoin apresentou melhor desempenho em relação ao algoritmo UAprioriMR, à medida que mais itemsets frequentes são produzidos. A partir dos experimentos, provou-se também, estatísticamente, utilizando-se o teste de *Wilcoxon*, que houve diferença significativa entre a média dos tempos de execução dos algoritmos UAprioriMR e UAprioriMRJoin, em favor deste último, quando os mínimos suportes esperados são reduzidos.

7. CONSIDERAÇÕES FINAIS E PERSPECTIVAS

A descoberta de conjuntos de itens frequentes (FIM), primeira etapa da tarefa de Análise de Associação, é um assunto bastante abordado pela comunidade científica [AGR93] [AGR94] [SAV95] [SRI96] [HAN00] [PEI01]. Isto ocorre porque esta etapa é o gargalo desta tarefa. Logo, diversos algoritmos, de diferentes abordagens, têm sido criados nas duas últimas décadas para lidar melhor com esta etapa da Análise de Associação. Além da comunidade científica, empresas também utilizam estes algoritmos com o objetivo de, por exemplo, buscar relações entre os itens comprados pelos seus clientes.

Os algoritmos tradicionais de FIM são aplicados sobre *datasets* que contém dados determinísticos. No entanto, atualmente tem havido um crescimento de *datasets* que armazenam probabilidades existenciais sobre os itens. Tais probabilidades denotam a incerteza de um determinado item pertencer a uma transação da base de dados. Os algoritmos tradicionais de FIM não se adaptam às bases com dados incertos [AGG09b]. Por este motivo, desde 2007, com o surgimento do algoritmo sequencial UApriori [CHU07], novas abordagens têm sido criadas para lidar com as probabilidades existenciais dos itens de um *dataset*.

Em função da grande quantidade de dados coletados nos dias atuais, o modelo de programação MapReduce tem se mostrado uma boa alternativa para executar sobre grandes *datasets*. Ele permite o desenvolvimento de algoritmos paralelos que executam sobre um *cluster* de nodos distribuídos. A implementação Apache Hadoop oferece aos programadores um ambiente propício para o desenvolvimento de algoritmos que utilizam o modelo MapReduce. Ela responsabiliza-se pela coordenação e gerenciamento da execução destes algoritmos, eximindo o programador de preocupações como a replicação dos dados, a tolerância a falhas e a comunicação entre os nodos presentes no *cluster*.

Alguns trabalhos têm concentrado esforços na integração de algoritmos FIM com o modelo de programação MapReduce. Ora estes trabalhos utilizam a implementação Apache Hadoop, ora manipulam esta implementação dentro do serviço oferecido pela empresa Amazon, o *Amazon Elastic MapReduce*. Trabalhos como [LI08], [LI12], [LIN12b], [YAH12] e [KUL13] evoluem o algoritmo tradicional Apriori para adaptar-se ao modelo MapReduce. Lin et al. [LIN12b] e Li et al. [LI12] apresentam resultados eficientes desta nova abordagem em relação à versão do tradicional Apriori sequencial, quando os algoritmos são aplicados sobre dados determinísticos.

Durante a pesquisa realizada na revisão sistemática desenvolvida por Carvalho e Ruiz [CAR13], foi encontrado apenas um trabalho relacionando FIM, MapReduce e incerteza nos *datasets*. O trabalho de Leung e Hayduk [LEU13] constrói um algoritmo denominado MR-growth, da classe *FP-growth-based*, baseado na construção de árvores UF-tree, e o aplica sobre *datasets* com incerteza associada, utilizando um *cluster* MapReduce.

Aggarwal et al. [AGG09b] e Tong et al. [TON12] afirmam que os algoritmos FIM, quando aplicados sobre *datasets* com incerteza, tem desempenho distinto dos resultados obtidos sobre *datasets* determinísticos. Estes dois trabalhos mostram, a partir de seus experimentos, que os algoritmos da classe *Apriori-based*, a qual faz parte o algoritmo UApriori, têm um melhor desempenho do que os algoritmos da classe *FP-growth-based*, ao contrário do que ocorre quando os *datasets* são determinísticos.

No entanto, os trabalhos de Chui et al. [CHU07] e Tong et al. [TON12] afirmam que o algoritmo UApriori apresenta problema de escalabilidade sobre grandes *datasets*. Este problema torna-se maior à medida que o mínimo suporte esperado é diminuído e, principalmente, se os itens presentes no *dataset* possuem probabilidades existenciais baixas. Perante este contexto, esta tese desenvolve, implementa e testa o algoritmo híbrido UAprioriMRJoin, o qual evolui o algoritmo UApriori,

integrando-o com o modelo de programação MapReduce, para lidar com *datasets* com incerteza associada e melhorar sua escalabilidade.

A partir dos experimentos realizados com o algoritmo UAprioriMRJoin foi possível responder a questão de pesquisa, afirmando que é possível desenvolver um algoritmo que apresente uma média de tempo de execução estatisticamente inferior à média de tempo de execução do algoritmo UAprioriMR. Desta forma, o objetivo desta tese, de desenvolver o algoritmo híbrido UAprioriMRJoin, conseguiu responder a questão de pesquisa.

7.1 Contribuições

A contribuição essencial desta tese é o desenvolvimento, implementação e testes realizados com o algoritmo híbrido UAprioriMRJoin. Nos experimentos realizados sobre os quatro *datasets*: accidents, connect, kosarak_10 e T25I15D320k provou-se, a partir de testes estatísticos, que o algoritmo UAprioriMRJoin tem melhores resultados do que o algoritmo UAprioriMR, desenvolvido em outros trabalhos [LIN12b] [LI12], à medida que o mínimo suporte esperado é reduzido e o número de nodos do *cluster* aumenta.

Nos trabalhos de Lin et al. [LIN12b] e Li et al. [LI12] o algoritmo UAprioriMR é abordado sobre *datasets* determinísticos. Nesta tese, o algoritmo UAprioriMR é implementado para que experimentos sobre *datasets* com incerteza possam ser realizados e, desta forma, comparar os resultados do algoritmo UAprioriMR com o algoritmo híbrido UAprioriMRJoin.

A tese também evidencia uma boa métrica de *speedup* do algoritmo UAprioriMRJoin. Ele obtém uma melhor métrica de *speedup* em relação ao algoritmo UAprioriMR à medida que o mínimo suporte esperado é reduzido e o número de nodos do *cluster* aumenta. A métrica de *speedup* do algoritmo UAprioriMRJoin é superlinear em alguns casos, como nos *datasets* T25I15D320k e connect, para alguns mínimos suportes esperados baixos.

O algoritmo híbrido UAprioriMRJoin tem melhor desempenho sobre todos os *datasets* envolvidos nos experimentos, à proporção que o mínimo suporte esperado decresce e o número de nodos do *cluster* aumenta. Ele tem melhor desempenho quando aplicado sobre o maior e mais esparso *dataset*: kosarak_10, com 847,875 transações, 36,506 itens distintos e densidade de $d = 0.000093$, quanto com o menor e mais denso *dataset*: *connect*, com 67,557 transações, apenas 129 itens distintos e densidade de $d = 0.33$.

O algoritmo UAprioriMRJoin também apresenta melhor desempenho quando aplicado sobre o *dataset* com menor média de itens por transação: o kosarak_10, com 3.4, até o maior deles: o connect, com uma média de 43 itens por transação. Nos dois outros *datasets*: accidents e T25I15D320k, com tamanho, média de itens por transação e densidade intermediárias, o algoritmo UAprioriMRJoin também apresenta melhor desempenho em relação ao algoritmo UAprioriMR.

O algoritmo híbrido UAprioriMRJoin evidencia um melhor desempenho em relação ao algoritmo UAprioriMR no passo $k = 2$. Isto porque o algoritmo UAprioriMR, neste passo, não consegue aproveitar-se da propriedade de poda. Portanto, quando o *dataset* tem um número médio de itens por transação menor do que o número de itemsets frequentes gerados no passo L_1, o algoritmo UAprioriMRJoin tem um melhor desempenho do que o UAprioriMR. Isto pode ocorrer nos demais passos $k > 2$ também. No entanto, nos testes realizados nesta tese, o algoritmo UAprioriMR sempre tem melhores resultados para $k > 2$, em função da aplicação da propriedade de poda.

Como contribuições secundárias há também o desenvolvimento, implementação e testes com o algoritmo UAprioriMR. A partir dos experimentos com este algoritmo é possível confirmar o bom desempenho dele sobre *datasets* com incerteza, quando comparado ao algoritmo UApriori sequencial. Este comportamento já havia sido mostrado pelos trabalhos de Lin et al. [LIN12b] e Li et al. [LI12], mas somente utilizando *datasets* determinísticos. Com os testes sobre o algoritmo UAprioriMR

afirma-se também que, utilizar MapReduce, em *cluster* com 1 nodo somente, não é uma boa abordagem. No entanto, à medida que o número de nodos no *cluster* aumenta, o desempenho do algoritmo melhora substancialmente.

Desta forma, foi possível constatar um bom desempenho da utilização de programação paralela em um ambiente distribuído, utilizando o modelo de programação MapReduce, em algoritmos de descoberta de itens frequentes sobre contextos com incerteza associada. A estratégia de divisão dos *datasets* em diversos *splits*, a posterior distribuição destes *splits* pelos nodos do *cluster* e aplicação de funções Map sobre cada *split*, apresenta bons resultados, pois auxilia os algoritmos de geração de itens frequentes a trabalhar com *datasets* menores (*splits*) e, por consequência, fazer varreduras menores nos *splits*.

7.2 Limitações

O algoritmo UAprioriMRJoin apresenta como limitação a utilização dele sobre *datasets* que contenham um número muito grande de itens por transação. Isto porque ele utiliza a estratégia do algoritmo UAprioriMRByT para gerar itemsets dos conjuntos candidatos, ou seja, produz estes itemsets para cada transação varrida. Desta forma, quando o número de itens por transação é muito grande, o número de itemsets dos conjuntos candidatos fica muito grande também, prejudicando o desempenho do algoritmo. O *dataset* kosarak evidencia esta limitação nos testes executados. Este *dataset* contém 4300 transações com mais de 100 itens e várias transações com um número de itens (2,498, por exemplo) muito além do tamanho médio de itens por transação (8.1).

Outra limitação desta pesquisa é que, por motivos financeiros e de tempo, os algoritmos não foram testados em um ambiente de *Cloud Computing* que ofereça uma implementação Apache Hadoop. Com a utilização de *Cloud Computing*, *datasets* maiores, bem como diferentes tipos de nodos no *cluster*, poderiam ter sido experimentados. Também seria possível aumentar e reduzir o número de nodos de modo mais prático do que como foi efetuado no laboratório experimental, pois os ambientes de *Cloud Computing* oferecem ferramentas que auxiliam os programadores e administradores nas tarefas de programar e gerenciar os nodos.

7.3 Perspectivas

Esta pesquisa não esgota-se com esta tese. Enxergam-se diversos trabalhos futuros que podem ser realizados a partir dos resultados obtidos, desde novos experimentos com o algoritmo UAprioriMRJoin, utilizando-se outras configurações, até a comparação dele com outros algoritmos de descoberta de itens frequentes. Destacam-se como perspectivas futuras:

- Ampliar a avaliação do comportamento do desempenho do algoritmo híbrido UAprioriMRJoin usando outras configurações: outros *datasets* reais e sintéticos, em *clusters* distintos, diversificando os mínimos suportes esperados.

- Avaliar o comportamento do desempenho dos algoritmos UAprioriMRJoin e UAprioriMR, variando o número de funções Map, Combine e Reduce, aplicadas sobre cada nodo. Por exemplo, como o laboratório utilizado nos experimentos possui máquinas dual core, será possível avaliar e comparar o desempenho do *cluster* quando utilizadas duas funções map por nodo.

- Investigar a implementação de um pré-processamento mais inteligente sobre os *splits*. Ou seja, interferir beneficamente na criação dos *splits* de modo a otimizar o algoritmo UAprioriMRJoin. Ao invés de deixar o trabalho de divisão dos *splits* como função exclusiva da implementação Apache Hadoop, que faz a divisão de modo sequencial, desenvolver um método que faça esta

divisão dos *splits* de acordo com as características do *dataset*: número de transações por *split*, média de itens por transação no *split*, média de *splits* por nodo, dentre outras.

- Avaliar o comportamento do desempenho dos algoritmos UAprioriMRJoin e UAprioriMR em um ambiente de *Cloud Computing*, alternando o poder de processamento e o número de nodos do *cluster*.

- Implementar o algoritmo MR-growth, desenvolvido por [LEU13], a fim de comparar seu desempenho com o algoritmo híbrido UAprioriMRJoin.

- Implementar o algoritmo UH-Mine integrando-o com o modelo de programação MapReduce, a fim de comparar seu desempenho com o algoritmo híbrido UAprioriMRJoin.

- Utilizar outra abordagem para armazenar os conjuntos de itens frequentes a cada passo do algoritmo. É possível, por exemplo, utilizar o HBase, um banco de dados distribuído, *open-source* e orientado a coluna, que integra-se com a implementação Apache Hadoop. Desta forma, poderá ser possível comparar os resultados do algoritmo UAprioriMRJoin salvando os itens frequentes no HDFS, por meio do *DistributedCache*, conforme implementado nesta tese, com o algoritmo UAprioriMRJoin usando o HBase.

REFERÊNCIAS BIBLIOGRÁFICAS

[AGG09a] C. C. Aggarwal. "Managing and Mining Uncertain Data". New York: Springer US, 2009, 491p.

[AGG09b] C. C. Aggarwal, Y. Li, J. Wang e J Wang. "Frequent pattern mining with uncertain data". In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '09, 2009, pp. 29–37.

[AGR93] R. Agrawal, T. Imielinski e A. Swami. "Mining association rules between sets of items in large databases". In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data - SIGMOD '93, ACM Press, 1993, pp. 207–216.

[AGR94] R. Agrawal e R. Srikant. "Fast Algorithms for Mining Association Rules". In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94), Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

[AMA13] Amazon Web Services. "Amazon Elastic MapReduce Developer Guide, API version: 2009-03-31". Kindle Edition, May, 2013.

[APA14] Apache Hadoop. "HDFS Architecture". Disponível em: <https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html>, Acesso em: agosto, 2014.

[BHA11] R.S. Bhadoria, R. Kumar e M. Dixit. "Analysis on probabilistic and binary datasets through frequent itemset mining". In: Proceedings of the 2011 World Congress on Information and Communication Technologies, WICT, 2011, pp. 263–267.

[CAL10] T. Calders, C. Garboni e B. Goethals. "Efficient pattern mining of uncertain data with sampling". In: Proceedings of the 14th Pacific-Asia Conference, PAKDD 2010, 2010, pp. 480–487.

[CAR13] J. V. Carvalho e D. D. A. Ruiz. "Discovering Frequent Itemsets on Uncertain Data: A Systematic Review". *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*, Berlin: Springer, vol. 7988, 2013, pp. 390–404.

[CHU07] C. Chui, B. Kao e E. Hung. "Mining Frequent Itemsets from Uncertain Data". In: Proceedings of the Eleventh Pacific-Asia Conference on Knowledge Discovery and Data Mining, Nanjing, China, 2007, pp. 47–58.

[CHU08] C. Chui e B. Kao. "A decremental approach for mining frequent itemsets from uncertain data". In: Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2008, pp.64–75.

[DEA08] J. Dean e G. Sanjay. "MapReduce: Simplified data processing on large clusters". *Communications of the ACM*, vol. 58(1), 2008, pp.107–113.

[DEM06] J. Demšar. "Statistical Comparisons of Classifiers over Multiple Data Sets". *The Journal of Machine Learning Research*, vol. 6, 2006, pp. 1–30.

[DOA14] D. Doane e L. Seward "Estatística Aplicada À Administração e Economia". 4a. ed. Bookman, 2014.

[FAY86] U. Fayyad, G. Piatetsky-Shapiro e P. Smyth. "From Data Mining to Knowledge Discovery: An overview". In: FAYYAD et al. Advances in Knowledge Discovery and Data Mining. G. Cambridge-Mass:AAAI/MIT Press, 1996.

[GAN11] J. Gantz e D. Reinsel. "Extracting Value from Chaos". IDC Analyze the Future, IDC IView, June, 2011. Disponível em: <http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf>, Acesso em: agosto, 2014.

[GAO11] F. Gao e C. Wu. "Mining frequent itemset from uncertain data". In: International Conference on Electrical and Control Engineering, ICECE 2011, pp.2329–2333.

[HAN95] J. Han, F. Yongjian. "Discovery of Multiple Level Association Rules from Large Databases". In: Proceddings of the 21st VLDB Conference, Zurique, Suiça, 1995, pp.420–431.

[HAN97] E. Han, G. Karypis e V. Kumar. "Min-Apriori: An Algorithm for Finding Association Rules in Data with Continuous Attributes". Disponível em: <http://www.cs.umn.edu/han>, Acesso em: março, 2013.

[HAN00] J. Han, J. Pei e Y. Yin. "Mining frequent patterns without candidate generation". In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data - SIGMOD '00, 2000, pp. 1–12.

[HAN11] J. Han, M. Kamber e J. Pei. "Data Mining Concepts and Tecniques". 3rd ed. Morgan Kaufman, 2011.

[HAY12] Y. Hayduk "Mining Frequent Patterns from Uncertain Data with MapReduce". Department of Computer Science The University of Manitoba Winnipeg, Manitoba, Canada, 2012.

[KOL13] J. Kolb e J. Kolb. "The Big Data Revolution". Applied Data Labs, 2013.

[KUL13] K. C. Kulkarni, R. S. Jagale e S. M. Rokade. "A Survey on Apriori Algorithm using MapReduce Technique". *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, Special Issue 4, 2013.

[LAK97] L. V. S. Lakshmanan, N. Leone, R. Ross, et al. "ProbView: a Flexible Probabilistic Database System". *ACM Transactions Database Systems*, vol. 22(3), 1997, pp. 419–469.

[LEA00] D. Lea. "A Java fork/join framework". ACM Java, 2000, pp. 36–43.

[LEU08] C. K. Leung, M. A. F. Mateo e D. A. Brajcsuk. "A tree-based approach for frequent pattern mining from uncertain data". In: 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2008, pp. 653–661.

[LEU09a] C. K. Leung e D. A. Brajcsuk. "Mining uncertain data for constrained frequent sets". In: Proceedings of the 2009 International Database Engineering & Applications Symposium, 2009, pp. 109–120.

[LEU09b] C. K. Leung e D. A. Brajcsuk. "Efficient algorithms for mining constrained frequent patterns from uncertain data". In: Proceedings of the 1st ACM SIGKDD Workshop on Knowledge Discovery from Uncertain Data, France, 2009, pp. 9–18.

[LEU10a] C. K. Leung, B. Hao e D. A. Brajcsuk. "Mining uncertain data for frequent itemsets that satisfy aggregate constraints". In: Proceedings of the 2010 ACM Symposium on Applied Computing, 2010, pp. 1034–1038.

[LEU10b] C. K. Leung e D. A. Brajcsuk. "uCFS2: an enhanced system that mines uncertain data for constrained frequent sets". In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium, 2010, pp. 32–37.

[LEU11] C. K. Leung e L. Sun. "Equivalence class transformation based mining of frequent itemsets from uncertain data". In: Proceedings of the ACM Symposium on Applied Computing, 2011, pp. 983–984.

[LEU13] C. K. Leung e Y. Hayduk. "Mining Frequent Patterns from Uncertain Data with MapReduce for Big Data Analytics". In: 18th International Conference, DASFAA 2013, China, April, 2013, pp. 22–25. *Lecture Notes in Computer Science/Database Systems for Advanced Applications*, Berlin: Springer, vol. 7825, 2013, pp. 440–455.

[LI08] H. Li, Y. Wang, D. Zhang, et al. "PFP: Parallel FP-growth for query recommendation". In: RecSys-08: 2008 ACM Conference on Recommender Systems, Switzerland, October, 2008, pp. 107–114.

[LI11] L. Li e M. Zhang "The Strategy of Mining Association Rule Based on Cloud Computing". In: Proceedings of the 2011 International Conference on Business Computing and Global Informatization (BCGIN '11), Washington, DC, USA, IEEE: pp. 475–478.

[LI12] J. Li, P. Roy, S. Khan, et al. "Data Mining Using Clouds: An Experimental Implementation of Apriori over MapReduce". In: 12th IEEE International Conference on Scalable Computing and Communications (ScalCom 2012), China, 2012.

[LIN12a] C. Lin e T. Hong. "A new mining approach for uncertain databases using CUFP trees". *Expert Systems with Applications*, vol. 39(4), March, 2012, pp. 4084–4093.

[LIN12b] M. Lin, P. Lee e S. Hsueh. "Apriori-based Frequent Itemset Mining Algorithms on MapReduce". In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, Malaysia, 2012, pp. 1–8.

[LIU12] Y. Liu. "Mining frequent patterns from univariate uncertain data". *Data and Knowledge Engineering*, vol. 71(1), 2012, pp. 47–68.

[MAD12] S. Madden. "From Databases to Big Data". *IEEE Internet Computing*, vol. 16(3), 2012, pp. 4–6.

[MIL13] E. Miller "An Overview of MapReduce and Its Impact on Distributed Data Processing". Amazon Media EU, Edição Kindle, 2013.

[MPI14] MPI Forum. "MPI: A Message-Passing Interface Standard Version 3.0". Disponível em: <http://www.mpi-forum.org>, Acesso em: outubro, 2014.

[OPE14] The OpenMP Architecture Review Board. "OpenMP: C and C++ Application Program Interface Version 4.0.". Disponível em: <http://openmp.org/wp/>, Acesso em: outubro, 2014.

[ORE12] "Big Data Now: Current Perspectives". 2012 Edition. O'Reilly Media Inc, 2012.

[PEI01] J. Pei, J. Han, H. Lu, et al. "H-mine: hyper-structure mining of frequent patterns in large databases". In: ICDM '01 Proceedings of the 2001 IEEE International Conference on Data Mining, 2001, pp. 441–448.

[RAU10] T. Rauber e G. Rünger. "Parallel Programming for Multicore and Cluster Systems". 2nd ed. Springer, 2010.

[SAV95] A. Savasere, E. Omiecinski e S. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases". In: 21st VLDB Conference, Swizerland, 1995, pp. 432–444.

[SHR13] G. Shroff "The Intelligent Web: search, smart, algorithms and big data". Oxford University Press, 2013.

[SIE13] S. Siewert. "Big data in the cloud: Data velocity". Disponível em: <http://www.ibm.com/developerworks/library/bd-bigdatacloud/>, Acesso em: novembro, 2013.

[SUN10] L. Sun, R. Cheng, D. Cheung, et al. "Mining uncertain data with probabilistic guarantees". In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2010, pp. 273–282.

[SRI96] R. Srikant e R. Agrawal. "Mining Quantitative Association Rules in Large Relational Tables". In: SIGMOD '96 Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, vol. 25(2), Canadá, 1996, pp. 1–12.

[TAN09] P. Tan, M. Steinbach e V. Kumar. "Introdução ao Data Mining". Rio de Janeiro: Ciência Moderna, 2009.

[TAN11] P. Tan, E.A. Peterson. "Mining probabilistic frequent closed itemsets in uncertain databases". In: Proceedings of the 49th Annual Southeast Regional Conference, 2011, pp. 86–91.

[TON12] Y. Tong, L. Chen, Y. Cheng, et al. "Mining frequent itemsets over uncertain databases". In: Proceedings of the VLDB Endowment, vol. 5(11), July, Turkey, 2012, pp. 1650–1661.

[WAN10] L. Wang, R. Cheng, S. Lee, et al. "Accelerating probabilistic frequent itemset mining: a model-based approach". In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, 2010, pp. 429–438.

[WHI12] T. White. "Hadoop: The Definitive Guide". 3rd Edition. O'Reilly, 2012.

[YAH12] O. Yahya, O. Hegazy e E. Ezat. "An Efficient Implementation of Apriori Algorithm based on Hadoop-MapReduce Model". *International Journal of Reviews in Computing*, vol. 12, 2012, pp. 59–67.

A. ALGUNS SCRIPTS EXECUTADOS NO *CLUSTER*

Neste apêndice estão detalhados três dos diversos *scripts* utilizados para executar os algoritmos UAprioriMR, UAprioriMRByT e UAprioriMRJoin. Eles permitem reduzir a intervenção do ser humano nos experimentos e são flexíveis o bastante para executar nas várias configurações de *cluster*. Os *scripts* detalhados neste apêndice foram aplicados sobre os *datasets* T25I15D320k, accidents e connect, respectivamente.

Script utilizado sobre o *dataset* accidents, pelo algoritmo UAprioriMR.

```
#!/bin/bash
# tamsplits: vetor que armazena os tamanhos dos splits para 5, 10, 15 e 20 nodos, ao trabalhar com o dataset Accidents.
tamsplits=(18875188 9437594 6291730 4718797)

# minsupesp: vetor que armazena os minimos suportes esperados aplicados sobre o dataset Accidents.
minsupesp=(0.3 0.2 0.1 0.05)

# k: controla quantos experimentos devem ser realizados.
for k in {0..9}; do

  # i: controla quantos nodos incluir no cluster.
  for i in {0..3}; do
    case $i in
          0) cp conf/slaves5 conf/slaves;;
          1) cp conf/slaves10 conf/slaves;;
          2) cp conf/slaves15 conf/slaves;;
          3) cp conf/slaves20 conf/slaves;;
    esac

    # j: controla quais suportes minimos esperados devem ser aplicados em cada execucao.
    for j in {0..3}; do
          # Para todo o cluster
          bin/stop-all.sh
          # Inicia todo o cluster
          bin/start-all.sh
          # Aguarda 10 segundos para iniciar todos os nodos do cluster.
          sleep 10
          # Instrucao para retirar o cluster do modo de seguranca.
          bin/hadoop dfsadmin -safemode leave
          # Aguarda 15 segundos para que todos os nodos do cluster saiam do modo de seguranca.
          sleep 15
          # Instrucao para remover os arquivos de saida da execucao anterior.
          /usr/local/hadoop/bin/hadoop dfs -rmr /user/hduser/UAprioriMRData-output*
          # Chamada para execucao do algoritmo UAprioriMR, capturando automaticamente o tamanho do split e o minimo suporte esperado.
      bin/hadoop jar UAprioriMR.jar UAprioriMR -r 1 /user/hduser/UAprioriMRData /user/hduser/UAprioriMRData-output ${minsupesp[$j]} 340183 ${tamsplits[$i]}
    done
  done
done
```

Script utilizado sobre o *dataset* T25I15D320k, pelo algoritmo UAprioriMRByT.

```
#!/bin/bash
# tamsplits: vetor que armazena os tamanhos dos splits para 5, 10, 15 e 20 nodos, ao trabalhar com o dataset T25I15D320k.
tamsplits=(16786228 8393114 5595410 4196557)

# minsupesp: vetor que armazena os minimos suportes esperados aplicados sobre o dataset T25I15D320k.
minsupesp=(0.1 0.05 0.025 0.01)

# k: controla quantos experimentos devem ser realizados.
for k in {0..9}; do

  # i: controla quantos nodos incluir no cluster.
  for i in {0..3}; do
    case $i in
          0) cp conf/slaves5 conf/slaves;;
          1) cp conf/slaves10 conf/slaves;;
          2) cp conf/slaves15 conf/slaves;;
          3) cp conf/slaves20 conf/slaves;;
    esac

    # j: controla quais suportes minimos esperados devem ser aplicados em cada execucao.
    for j in {0..3}; do
          # Para todo o cluster
          bin/stop-all.sh
          # Inicia todo o cluster
          bin/start-all.sh
          # Aguarda 10 segundos para iniciar todos os nodos do cluster.
          sleep 10
          # Instrucao para retirar o cluster do modo de seguranca.
          bin/hadoop dfsadmin -safemode leave
          # Aguarda 15 segundos para que todos os nodos do cluster saiam do modo de seguranca.
          sleep 15
          # Instrucao para remover os arquivos de saida da execucao anterior.
          /usr/local/hadoop/bin/hadoop dfs -rmr /user/hduser/UAprioriMRData-output*
          # Chamada para execucao do algoritmo UAprioriMRByT, capturando automaticamente o tamanho do split e o minimo suporte esperado.
      bin/hadoop jar UAprioriMRByT.jar UAprioriMRByT -r 1 /user/hduser/UAprioriMRData /user/hduser/UAprioriMRData-output ${minsupesp[$j]} 320000 ${tamsplits[$i]}
    done
  done
done
```

Script utilizado sobre o *dataset* connect, pelo algoritmo UAprioriMRJoin.

```
#!/bin/bash
# tamsplits: vetor que armazena os tamanhos dos splits para 5, 10, 15 e 20 nodos, ao trabalhar com o dataset Connect.
tamsplits=(4810343 2405172 1603448 1202586)

# minsupesp: vetor que armazena os minimos suportes esperados aplicados sobre o dataset Connect.
minsupesp=(0.9 0.8 0.7 0.6)

# k: controla quantos experimentos devem ser realizados.
for k in {0..9}; do

  # i: controla quantos nodos incluir no cluster.
  for i in {0..3}; do
    case $i in
          0) cp conf/slaves5 conf/slaves;;
          1) cp conf/slaves10 conf/slaves;;
          2) cp conf/slaves15 conf/slaves;;
          3) cp conf/slaves20 conf/slaves;;
    esac

    # j: controla quais suportes minimos esperados devem ser aplicados em cada execucao.
    for j in {0..3}; do
          # Para todo o cluster
          bin/stop-all.sh
          # Inicia todo o cluster
          bin/start-all.sh
          # Aguarda 10 segundos para iniciar todos os nodos do cluster.
          sleep 10
          # Instrucao para retirar o cluster do modo de seguranca.
          bin/hadoop dfsadmin -safemode leave
          # Aguarda 15 segundos para que todos os nodos do cluster saiam do modo de seguranca.
          sleep 15
          # Instrucao para remover os arquivos de saida da execucao anterior.
          /usr/local/hadoop/bin/hadoop dfs -rmr /user/hduser/UAprioriMRData-output*
          # Chamada para execucao do algoritmo UAprioriMR, capturando automaticamente o tamanho do split e o minimo suporte esperado.
      bin/hadoop jar UAprioriMRJoin.jar UAprioriMRJoin -r 1 /user/hduser/UAprioriMRData /user/hduser/UAprioriMRData-output ${minsupesp[$j]}  ${tamsplits[$i]} 34
    done
  done
done
```

B. CONTEÚDO DAS PLANILHAS DOS EXPERIMENTOS

Este apêndice ilustra uma porção do conteúdo de uma das planilhas onde os dados dos experimentos foram registrados. A planilha em questão mostra os dados capturados nos experimentos realizados sobre o *dataset* T25I15D320k, em um cluster de 10 nodos, para os mínimos suportes esperados de 0.1, 0.05, 0.025 e 0.01. A descrição dos atributos capturados está detalhada na tabela 6.7 do capítulo 6.

nodo/*cluster*	data	hora	minsupesp	|D|	minsupesp · |D|	|nodos|	|maps|	|reducers|	|*split*|	*job* 1	*job* 2	*job* 3	total	|1-itemset|	|2-itemset|
14	11/09/2014	12:23:10	0.1	320000	32000	10	10	1	8,393,114	25	26		51	8	
14	12/09/2014	00:29:11	0.1	320000	32000	10	10	1	8,393,114	23	26		49	8	
14	12/09/2014	03:03:52	0.1	320000	32000	10	10	1	8,393,114	23	25		48	8	
14	12/09/2014	05:30:03	0.1	320000	32000	10	10	1	8,393,114	23	26		49	8	
14	12/09/2014	07:57:37	0.1	320000	32000	10	10	1	8,393,114	24	26		50	8	
14	12/09/2014	09:46:41	0.1	320000	32000	10	10	1	8,393,114	23	25		48	8	
14	12/09/2014	12:20:26	0.1	320000	32000	10	10	1	8,393,114	23	26		49	8	
14	12/09/2014	14:12:04	0.1	320000	32000	10	10	1	8,393,114	24	25		49	8	
14	15/09/2014	12:26:41	0.1	320000	32000	10	10	1	8,393,114	24	26		50	8	
14	15/09/2014	12:53:53	0.1	320000	32000	10	10	1	8,393,114	24	26		50	8	
									Média	23.6	25.7		49.3		
14	11/09/2014	12:18:49	0.05	320000	16000	10	10	1	8,393,114	24	153		177	111	
14	12/09/2014	00:24:58	0.05	320000	16000	10	10	1	8,393,114	23	150		173	111	
14	12/09/2014	02:59:44	0.05	320000	16000	10	10	1	8,393,114	25	143		168	111	
14	12/09/2014	05:25:49	0.05	320000	16000	10	10	1	8,393,114	24	148		172	111	
14	12/09/2014	07:53:15	0.05	320000	16000	10	10	1	8,393,114	24	150		174	111	
14	12/09/2014	09:42:21	0.05	320000	16000	10	10	1	8,393,114	24	154		178	111	
14	12/09/2014	12:16:02	0.05	320000	16000	10	10	1	8,393,114	23	154		177	111	
14	12/09/2014	14:07:41	0.05	320000	16000	10	10	1	8,393,114	24	152		176	111	
14	15/09/2014	13:17:15	0.05	320000	16000	10	10	1	8,393,114	28	157		185	111	
14	15/09/2014	12:49:28	0.05	320000	16000	10	10	1	8,393,114	25	154		179	111	
									Média	24.4	148.7		175.9		
14	11/09/2014	12:10:39	0.025	320000	8000	10	10	1	8,393,114	23	380		403	361	
14	12/09/2014	00:17:16	0.025	320000	8000	10	10	1	8,393,114	22	359		381	361	
14	12/09/2014	02:52:02	0.025	320000	8000	10	10	1	8,393,114	24	360		384	361	
14	12/09/2014	05:18:02	0.025	320000	8000	10	10	1	8,393,114	23	360		383	361	
14	12/09/2014	07:45:23	0.025	320000	8000	10	10	1	8,393,114	24	364		388	361	
14	12/09/2014	09:34:32	0.025	320000	8000	10	10	1	8,393,114	23	365		388	361	
14	12/09/2014	12:08:35	0.025	320000	8000	10	10	1	8,393,114	24	359		383	361	
14	12/09/2014	13:59:38	0.025	320000	8000	10	10	1	8,393,114	26	368		394	361	
14	15/09/2014	13:08:26	0.025	320000	8000	10	10	1	8,393,114	25	387		412	361	
14	15/09/2014	12:41:16	0.025	320000	8000	10	10	1	8,393,114	29	372		401	361	
									Média	24.3	366.3		391.7		
14	11/09/2014	11:58:31	0.01	320000	3200	10	10	1	8,393,114	23	544	79	646	659	229
14	12/09/2014	00:05:08	0.01	320000	3200	10	10	1	8,393,114	24	540	79	643	659	229
14	12/09/2014	02:40:29	0.01	320000	3200	10	10	1	8,393,114	25	514	74	613	659	229
14	12/09/2014	05:06:13	0.01	320000	3200	10	10	1	8,393,114	23	521	78	622	659	229
14	12/09/2014	07:32:54	0.01	320000	3200	10	10	1	8,393,114	24	566	73	663	659	229
14	12/09/2014	09:22:54	0.01	320000	3200	10	10	1	8,393,114	23	515	78	616	659	229
14	12/09/2014	11:56:12	0.01	320000	3200	10	10	1	8,393,114	24	520	89	633	659	229
14	12/09/2014	13:47:14	0.01	320000	3200	10	10	1	8,393,114	23	542	89	654	659	229
14	15/09/2014	12:56:07	0.01	320000	3200	10	10	1	8,393,114	25	539	81	645	659	229
14	15/09/2014	12:29:07	0.01	320000	3200	10	10	1	8,393,114	28	534	79	641	659	229
									Média	24.2	532.7	77.3	637.6		