Comparação de Algoritmos Paralelos para a Extração de Regras de Associação no Modelo de Memória Distribuída

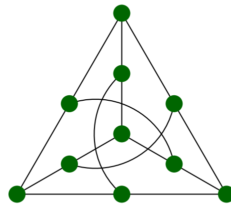
Marcos Alves Mariano

Dissertação de Mestrado

Orientação: Prof. Dr. Henrique Mongelli

Área de Concentração: Ciência da Computação

Durante a elaboração desse trabalho o autor recebeu apoio financeiro da CAPES.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
22 de novembro de 2011

Comparação de Algoritmos Paralelos para a Extração de Regras de Associação no Modelo de Memória Distribuída

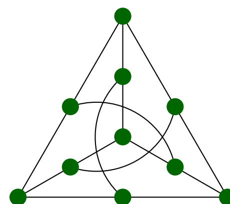
Marcos Alves Mariano

Dissertação de Mestrado

Orientação: Prof. Dr. Henrique Mongelli

Área de Concentração: Ciência da Computação

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, Curso de Mestrado em Ciência da Computação, Faculdade de Computação da Universidade Federal de Mato Grosso do Sul.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
22 de novembro de 2011

Resumo

Nos últimos anos, a extração de conhecimento a partir de grandes volumes de dados têm sido o objeto de estudo em muitas pesquisas. Com isso, diversas técnicas de mineração de dados foram desenvolvidas com o propósito de descobrir informações para auxiliar os gestores de empresas e organizações na tomada de decisões. Uma das técnicas mais predominantes na mineração de dados é a de extração de regras de associação, devido a sua eficiência e simplicidade no tratamento das informações. Com a utilização do paralelismo em diversos problemas computacionais, algoritmos paralelos para a mineração de dados foram construídos utilizando a técnica de extração de regras de associação. Dentre os algoritmos paralelos mais conhecidos, utilizando o modelo de memória distribuída, está o Apriori, o Eclat e o *FP-Growth*. Assim, o objetivo deste trabalho é implementar e comparar o desempenho dos algoritmos paralelos Apriori, Eclat e *FP-Growth* com diferentes números de processadores e tamanhos de bases de dados de entrada.

Palavras-chave: Regras de Associação, Apriori, Eclat, *FP-Growth*.

Abstract

In the last years, the extraction of knowledge from large amount of data have been the object of study in many surveys. Then, many data mining techniques have been developed in order to discover information to assist managers of companies and organizations in decision-making. One of the most prevalent techniques in data mining is the extraction of association rules, due to its efficiency and simplicity in managing information. With the utilization of the parallelism in many computational problems, works with algorithms for mining data were constructed using the technique of extracting association rules. Among the best most common algorithms using distributed memory model, is the Apriori, the Eclat and FP-Growth. So the main objective of this research is to implement and compare the performance of parallel algorithms Apriori, Eclat and FP-Growth with different processor numbers and sizes of input databases.

Keyword: Association Rules, Apriori, Eclat, FP-Growth.

Conteúdo

1 Introdução **11**

1.1 Motivação e Objetivos 11

1.2 Organização do Texto 12

2 Computação Paralela **13**

2.1 Conceitos Gerais 13

2.2 Modelos de Computação Paralela 14

2.2.1 Modelo de Memória Compartilhada 14

2.2.2 Modelo de Memória Distribuída 15

2.2.3 Modelos Realísticos 16

2.3 MPI (*Message Passing Interface*) 19

3 Mineração de Dados **20**

3.1 Conceitos Gerais 20

3.2 Regras de Associação 21

3.3 Estruturas de Dados 25

3.3.1 Projeção Vertical de Bases de Dados 25

3.3.2 *Hash-Tree* 26

3.4 Algoritmos Sequenciais 28

3.4.1 Algoritmo Apriori 28

3.4.2 Algoritmo AprioriTid 30

3.4.3 Algoritmo Partition 31

3.4.4 Algoritmo *FP-Growth* 34

3.4.5 Algoritmo Eclat 38

4 Algoritmos Paralelos para Extração de Regras de Associação 42

4.1 Conceitos Gerais 42

4.2 Algoritmo Apriori 42

4.3 Algoritmo Eclat 43

4.3.1 Fase de Inicialização 43

4.3.2 Fase de Processamento 44

4.4 Algoritmo *FP-Growth* 44

5 Bases de Dados Educacionais 48

5.1 Conceitos Gerais 48

5.2 Sistemas Nacionais de Avaliação e Informações Educacionais 49

5.2.1 Censo Escolar 49

5.2.2 ENCCEJA - (Exame Nacional para Certificação de Competências de Jovens e Adultos) 49

5.2.3 SAEB - (Sistema Nacional de Avaliação da Educação Básica) . . . 49

5.2.4 Provinha Brasil 50

5.2.5 ENEM - (Exame Nacional do Ensino Médio) 50

5.2.6 SINAES - (Sistema Nacional de Avaliação da Educação Superior) . 50

5.3 Projeto Web-PIDE 51

5.3.1 Formato dos Dados Disponibilizados pelo INEP 53

5.4 Extração de Regras de Associação de Dados Educacionais 57

6 Implementações e Resultados 60

6.1 Conceitos Gerais 60

6.2 Ambiente Computacional 60

6.3 Bases de Dados de Entrada 60

6.4 Bases de Dados de Saída 61

6.5 Tempos de Processamento 63

6.5.1 Tempo de Leitura 63

6.5.2 Tempo de Comunicação 63

6.5.3 Tempo de Computação Local 63

6.5.4 Tempo de Escrita 64
6.6 Análise dos Resultados Obtidos 64
6.6.1 Tempos de Processamento do Apriori 64
6.6.2 Tempos de Processamento do Eclat 67
6.6.3 Tempos de Processamento do *FP-Growth* 69
6.6.4 Comparativo do Desempenho das Implementações 72

7 Conclusão **76**

Referências Bibliográficas **78**

Lista de Figuras

2.1 Modelo BSP [12] 17

2.2 Modelo CGM [12] 18

3.1 Fases do processo de KDD [10]. 21

3.2 *Lattice* de $I = \{A, B, C, D\}$. 23

3.3 Exemplo de mineração de regras de associação de uma base de dados. . . . 24

3.4 Exemplo de base de dados no formato vertical. 26

3.5 Exemplo de *hash-tree*. 26

3.6 Exemplo de inserção de um itemset na *hash-tree*. 27

3.7 Exemplo de mapeamento de uma transação na *hash-tree*. 27

3.8 *FP-Tree* construída. 36

3.9 *FP-Tree* condicionada de “M”. 37

3.10 *itemsets* frequentes máximos. 39

3.11 Composição dos *itemsets* frequentes máximos em potencial. 40

4.1 Fase de Inicialização do Algoritmo Eclat 44

4.2 Exemplo da construção de *FP-Trees* locais a partir de uma base de dados particionada entre dois processadores. 46

4.3 Mineração de *itemsets* frequentes para o exemplo apresentado na Figura 4.1. 47

5.1 Arquitetura da plataforma Web-PIDE [28]). 52

5.2 Parte de um dos arquivos “Leia-me.pdf”do SAEB 2003 [1]. 54

5.3 Parte de um dos arquivos SAS do SAEB 2003 [1]. 55

5.4 Parte de um dos arquivos ASCII do SAEB 2003 [1]. 56

5.5 Parte do arquivo de variáveis selecionadas do SAEB 2003. 57

5.6 Parte da base de dados em itens da “Prova de Matemética da 4° Série - SAEB 2003”. 58

5.7 Parte dos itemsets frequentes e das regras de associação extraídos da “Prova de Matemética da 4° Série - SAEB 2003”. 59

6.1 Parte da base de dados de entrada com 256 transações. 61

6.2 Parte da base de dados de saída obtida com 256 transações. 62

6.3 Tempos de processamento do Algoritmo Apriori. 64

6.4 Tempos de processamento do Algoritmo Apriori com 256, 512, 1024 e 2048 transações. 65

6.5 Tempos de processamento do Algoritmo Apriori com 8192, 16384, 32768 e 65536 transações. 66

6.6 Tempos de processamento do Algoritmo Eclat. 67

6.7 Tempos de processamento do Algoritmo Eclat com 256, 512, 1024 e 2048 transações. 68

6.8 Tempos de processamento do Algoritmo Eclat com 8192, 16384, 32768 e 65536 transações. 69

6.9 Tempos de processamento do Algoritmo *FP-Growth*. 70

6.10 Tempos de processamento do Algoritmo *FP-Growth* com 256, 512, 1024 e 2048 transações. 70

6.11 Tempos de processamento do Algoritmo *FP-Growth* com 8192, 16384, 32768 e 65536 transações. 71

6.12 Tempos de processamento das implementações com 256, 2048, 8192 e 65536 transações. 72

6.13 *Speedup* das implementações com 256, 2048, 8192 e 65536 transações. . . . 74

Lista de Tabelas

3.1 Notação usada no algoritmo Partition. 32

3.2 Base de dados de transações 35

3.3 Base de dados condicionadas, *FP-Tree* condicionada e *Itemsets* frequentes. 38

6.1 Tempos de comunicação das implementações com 256 transações. 73

6.2 Tempos de computação local das implementações com 256 transações. . . . 73

6.3 Tempos de comunicação das implementações com 65536 transações. 73

6.4 Tempos de computação local das implementações com 65536 transações. . 74

Capítulo 1

Introdução

1.1 Motivação e Objetivos

Atualmente, empresas e organizações manipulam e armazenam grandes volumes de dados. Compreender esses dados, ou seja, conhecer a informação implícita nesses dados assume, cada vez mais, um papel importante no apoio à tomada de decisão. Diante disso, a área de mineração de dados vem atraindo muitos pesquisadores e se tornando extremamente importante no meio computacional.

Dentre várias técnicas existentes, a extração de regras de associação é uma das mais conhecidas na mineração de dados. Essa técnica consiste em identificar determinadas séries de padrões em bases de dados com grandes volumes de registros, permitindo, após a sua interpretação, adquirir conhecimentos acerca do problema em estudo.

Tradicionalmente, diversos algoritmos sequenciais para a mineração de dados empregando a técnica de regras de associação têm sido propostos. Porém, para manusear grandes quantidades de dados, um maior poder computacional é necessário (processador, memória e E/S) e que pode ser oferecido por arquiteturas projetadas para o processamento paralelo. Com isso, o desenvolvimento de algoritmos para a mineração de regras de associação, explorando paralelismo, têm sido objeto de estudos.

A arquitetura computacional empregando memória distribuída é a mais comumente utilizada na implementação de algoritmos paralelos para a extração de regras de associação. Nessa arquitetura, cada processador possui uma memória local, não sendo permitido que um processador obtenha acesso à memória de outro processador, e a comunicação entre os processadores é realizada apenas através da troca de mensagens.

Dentro desse contexto, o presente trabalho teve como objetivo realizar um estudo comparativo do desempenho de algoritmos paralelos para mineração de regras de associação em um modelo de memória distribuída. Cada algoritmo emprega uma estratégia diferente para determinar o conjunto de regras de associação a partir de uma base de dados de grande dimensão.

Os desempenhos dos algoritmos paralelos são comparados para determinar o mais eficiente deles em relação: (*i*) à quantidade de dados utilizada no processamento; e (*ii*) ao número de processadores utilizados no ambiente de execução. Os algoritmos foram implementados para um *cluster*, utilizando a linguagem de programação C++ e a biblioteca de comunicação MPI.

Para exemplificar a aplicação da mineração de regras de associação em uma base de dados real, utilizamos, como estudo de caso, os dados disponibilizados pelo INEP (Instituto Nacional de Estudos e Pesquisas Educacionais), nas quais são volumes de dados coletados periodicamente com a aplicação de avaliações educacionais em âmbito nacional. Dentre as avaliações educacionais mais conhecidas, estão: o SAEB (Sistema Nacional de Avaliação da Educação Básica), o ENEM (Exame Nacional do Ensino Médio) e o ENADE (Exame Nacional de Desempenho de Estudantes).

1.2 Organização do Texto

Este trabalho está organizado como segue. No Capítulo 2 são descritos os modelos de computação paralela empregados na construção de algoritmos. No Capítulo 3 são apresentados os principais conceitos da técnica de mineração de regras de associação, as estruturas de dados utilizadas, e os algoritmos sequenciais predominates na descoberta de regras de associação. No Capítulo 4 são apresentados os algoritmos paralelos de mineração de regras de associação utilizados como objetos de estudo no trabalho. No Capítulo 5 são descritas as definições e as características das bases de dados disponibilizadas pelo INEP, como também, é apresentada a aplicação da mineração de regras de associação em bases de dados educacionais. E, por fim, no Capítulo 6 são apresentados os resultados obtidos com as implementações dos algoritmos paralelos abordados no trabalho.

Capítulo 2

Computação Paralela

2.1 Conceitos Gerais

Para buscar soluções mais rápidas para problemas envolvendo grandes quantidades de informações, existem os sistemas de computação paralela. Algumas áreas como: previsão de tempo, processamento de imagens, inteligência artificial, entre outras, possuem problemas em que a computação sequencial exige uma solução dentro de tempos considerados “grandes”. Com a computação paralela, diversos problemas podem ser solucionados através da seguinte estratégia: o problema é dividido em partes menores, de modo que sejam independentes entre si, então, processando-as simultaneamente por um computador paralelo.

Um computador paralelo consiste em um conjunto de processadores, geralmente do mesmo tipo, interconectados de acordo com uma determinada topologia para permitir a coordenação de suas atividades e troca de dados [5]. Essa definição abrange desde supercomputadores paralelos, que possuem centenas de processadores, até um *cluster* de PC, implementado em um ambiente de rede.

Na computação paralela a construção de algoritmos modifica-se consideravelmente. As técnicas utilizadas em algoritmos sequenciais para resolver um problema não servem totalmente para a criação de algoritmos paralelos para o mesmo problema. Os recursos disponíveis são maiores e devem ser aproveitados ao máximo. Na construção dos algoritmos deve-se considerar a natureza dos problemas [25].

Alguns problemas são inerentemente paralelizáveis, enquanto outros apresentam um paralelismo menos transparente, exigindo estratégias mais elaboradas na construção dos algoritmos. Existem também aqueles em que o grau de paralelismo é muito baixo ou inexistente. No desenvolvimento dos algoritmos, deve-se considerar, também, a arquitetura paralela a ser utilizada, pois a sua utilização está ligada diretamente no desempenho dos algoritmos.

Este capítulo está organizado em duas seções. Na Seção 2.2 são apresentados os modelos de computação paralela empregados na construção de algoritmos e na Seção 2.3

é apresentada a biblioteca LAM-MPI, umas das mais utilizadas para a troca de mensagens em plataformas paralelas.

2.2 Modelos de Computação Paralela

Na construção de um algoritmo paralelo existem diversos fatores que devem ser considerados para se obter um desempenho superior à sua versão sequencial. Os fatores mais determinantes são [5]:

1. **Balanceamento de Carga**: Refere-se à divisão equitativa do trabalho entre os processadores disponíveis.
2. **Minimização de Comunicação**: Consiste na busca pelo menor número possível de comunicação entre os processadores.
3. **Sobreposição de comunicação e computação**: Consiste em atribuir novas tarefas aos processadores que estão ociosos aguardando pela resposta de outros processadores.

Na computação paralela o tempo de execução total de um programa é o fator considerado para a avaliação do seu desempenho. O tempo de execução de um programa depende de três componentes: tempo de computação, tempo de comunicação e tempo ocioso. O tempo de computação refere-se ao tempo gasto nas computações sobre os dados do problema. Nesse caso, o tempo ideal para um problema é que se tivermos p processadores em atividade, o tempo gasto na computação é t/p, em que t é o tempo gasto pelo algoritmo na versão sequencial. O tempo de comunicação é o tempo gasto pelos processadores para enviar e receber mensagens. Esse tempo é determinado pela latência e a largura da banda, em que a latência é o tempo gasto na preparação dos pacotes a serem enviados e a largura de banda é a velocidade real de transmissão. E por fim, o tempo ocioso é o tempo que um processador gasta aguardando dados de outros processadores, de modo que nenhum trabalho hábil é realizado.

Na computação paralela existe uma variedade de modelos que foram propostos. Porém, não há um modelo que seja amplamente utilizado para o projeto e análise dos algoritmos paralelos [5]. Os modelos mais empregados no desenvolvimento e análise de algoritmos paralelos são apresentados a seguir.

2.2.1 Modelo de Memória Compartilhada

O modelo de memória compartilhada consiste de um conjunto de processadores, cada um com uma memória local própria, executando em paralelo e comunicando-se através de uma memória global compartilhada. Cada processador é unicamente identificado por um número inteiro, e possui acesso a todos os endereços da memória compartilhada.

Nesse modelo, um processador pode executar uma instrução diferente ou não daquela executada por qualquer outro processador, em uma mesma unidade de tempo. Porém, o conjunto de dados a serem processados por um processador possivelmente será diferente daqueles processados por outros processadores em um mesmo instante. Os dados processados por um processador i e que serão utilizados por outros processadores são escritos pelo processador i na memória compartilhada.

Esse modelo oferece uma estrutura bastante atrativa para o desenvolvimento de algoritmos para a computação paralela. O modelo de memória compartilhada mais conhecido é o PRAM (*Parallel Random Access Machine*), o qual é uma extensão do modelo de computação sequencial de Von Neumam (RAM - *Random Access Machine*).

2.2.2 Modelo de Memória Distribuída

O modelo de memória distribuída, também chamado de modelo de rede, consiste de um conjunto de processadores onde a memória está distribuída entre os processadores e nenhuma memória global está disponível. Nesse modelo, a comunicação é feita através da troca de mensagens entre os processadores.

Na abordagem de troca de mensagens, a divisão dos dados e das tarefas entre os processadores, além do gerenciamento das comunicações, é de responsabilidade do programador. Se um processador i precisar de um dado processado pelo processador j, o processador j envia este dado para o processador i, através de uma mensagem pela rede de interconexão, e o processador i recebe este dado.

A topologia de interconexão dos processadores é incorporada pelo próprio modelo. O desempenho dessas topologias é dado em função de dois parâmetros: o diâmetro, que é a distância máxima entre quaisquer dois processadores, e o grau máximo, que corresponde ao número máximo de processadores ligados diretamente a um processador.

As principais topologias usadas no modelo de memória distribuída são [5]:

- **Lista linear**: Consiste de p processadores P_1, P_2, ..., P_p conectados linearmente, ou seja, P_i está conectado aos processadores P_{i-1} e P_{i+1}, se existirem. Possui diâmetro $p-1$ e seu grau máximo é 2.

- **Anel**: É uma lista linear onde o último processador está conectado ao primeiro. Possui diâmetro $\lfloor p/2 \rfloor$ e grau máximo 2.

- **Grade**: É uma versão bidimensional da lista linear. Consistem de $p = n^2$ processadores conectados numa grade $n \times n$, tal que, cada processador $P_{i,j}$ está ligado aos processadores $P_{i\pm1,j}$ e $P_{i,j\pm1}$, se existirem. Tem diâmetro $2(\sqrt{p}-1)$ e grau máximo 4.

- **Hipercubo**: Consiste de $p = 2^d$ processadores interligados em um cubo d-dimensional definido a seguir. Considere os p processadores rotulados como P_0, P_1, P_{p-1}. Seja a representação binária de i, $0 \leq i \leq p-1$ dada por $i = i_{d-1}i_{d-2}$ ... $i_2i_1i_0$. Dois

processadores estão conectados se, e somente se, seus índices diferem em apenas um *bit*. O hipercubo tem diâmetro e grau máximo $d = \log p$.

Nos modelos de memória distribuída, geralmente, a computação é efetuada usando um número fixo de processadores interconectados numa determinada topologia. A análise da complexidade dos seus algoritmos leva considera a comunicação e o tempo de processamento que são utilizados. No caso da comunicação, são analisados o número e o tamanho das mensagens. Com respeito ao tempo de processamento é feita uma análise local e uma análise global.

2.2.3 Modelos Realísticos

Nos algoritmos sequenciais, a principal medida de desempenho é o tempo de execução do algoritmo. Porém, no desenvolvimento de algoritmos paralelos devem-se considerar diversos outros parâmetros. Dentre esses parâmetros, podem-se destacar o número de processadores a serem utilizados na solução do problema, o tamanho do problema em questão, bem como a estratégia empregada na utilização desses dois parâmetros.

Com isso, diversos modelos de computação para o desenvolvimento de algoritmos paralelos foram construídos. Dentre os modelos desenvolvidos, os mais utilizados para a construção de algoritmos em máquinas paralelas reais são os modelos BSP e CGM [5], que estão descritos a seguir.

Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*) foi proposto por Valiant [29]. Além de ser um dos principais modelos realísticos, foi um dos primeiros a considerar os custos de comunicação e a abstração das características de uma máquina paralela através de um pequeno número de parâmetros.

Os parâmetros considerados no modelo BSP são:

- n: tamanho do problema;
- p: número de processadores com memória local;
- L: tempo máximo de um super-passo (periodicidade);
- g: descreve a taxa de eficiência de computação e comunicação, que corresponde à razão entre:
  - o número total de operações de computação local de todos os processadores (em unidades básicas de computação) em uma unidade de tempo e,
  - o número total de mensagens enviadas/recebidas (em palavras) em uma unidade de tempo.

O modelo BSP consiste de um conjunto de p processadores com memória local. Os processadores comunicam-se através de algum meio de interconexão, gerenciados por um roteador que pode transmitir mensagens ponto-a-ponto entre pares de processadores. Também oferece capacidade para sincronizar alguns, ou todos, os processadores em intervalos regulares de L unidades de tempo.

Um algoritmo BSP consiste de uma sequência de super-passos, conforme mostra a Figura 2.1. Em cada super-passo, cada processador executa uma combinação de computações locais, transmissão de mensagens e recebimento de mensagens de outros processadores. As tarefas de computação e comunicação podem ser separadas em super-passos de computação e super-passos de comunicação.

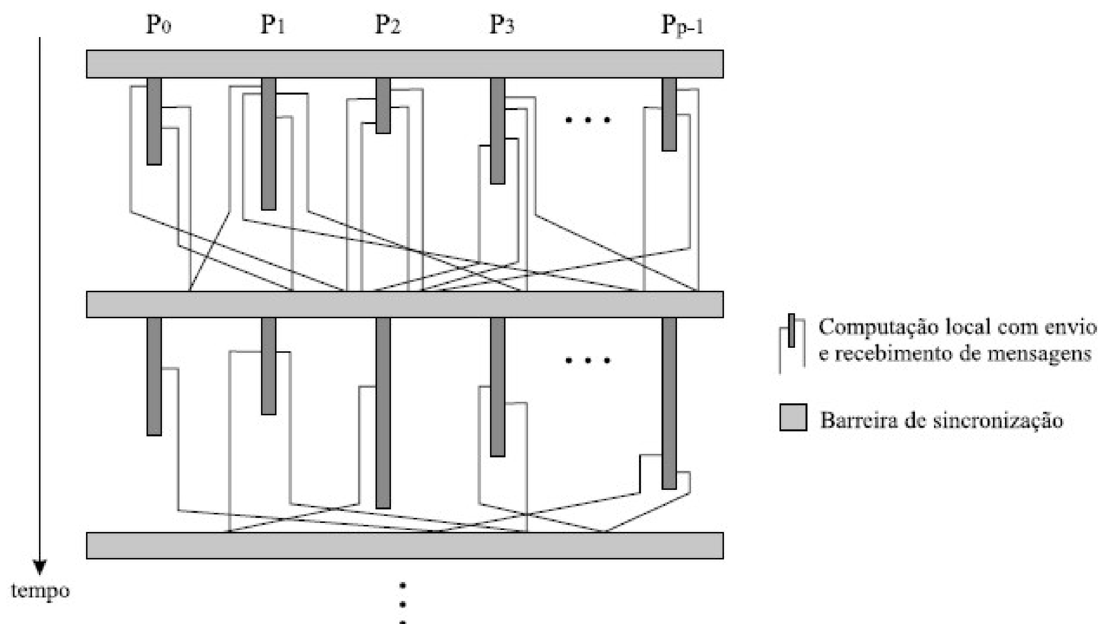


Figura 2.1: Modelo BSP [12]

Após cada período de L unidades de tempo, uma barreira de sincronização é efetuada para determinar se o super-passo foi completado por todos os processadores. Em caso afirmativo, a máquina executa o próximo super-passo; caso contrário, há outro período de L unidades de tempo para finalizar o super-passo. Isso assegura que os dados recebidos pelos processadores estarão disponíveis para o próximo super-passo.

Modelo CGM

O modelo CGM (*Coarsed Grained Multicomputer*) foi proposto por Dehne et al. [6]. Nesse modelo, os processadores podem estar conectados por qualquer meio de interconexão. O termo "*granularidade grossa*" (*coarsed grained*) refere-se ao fato do tamanho do problema ser consideravelmente maior que o número de processadores, ou seja, $n/p \gg p$.

Os parâmetros definidos no modelo CGM são:

- n: tamanho do problema;
- p: número de processadores com memória local do tamanho igual a n/p.

Um algoritmo CGM é constituído por uma sequência alternada de rodadas de computação local e comunicação global. Na rodada de computação, cada processador realiza o processamento dos dados disponibilizados localmente, geralmente, utilizando o melhor algoritmo sequencial para o problema em questão. Na rodada de comunicação, cada processador envia e recebe $O(n/p)$ dados no máximo. As rodadas de computação local e de comunicação entre os processadores são separadas por barreiras de sincronização, conforme mostra a Figura 2.2.

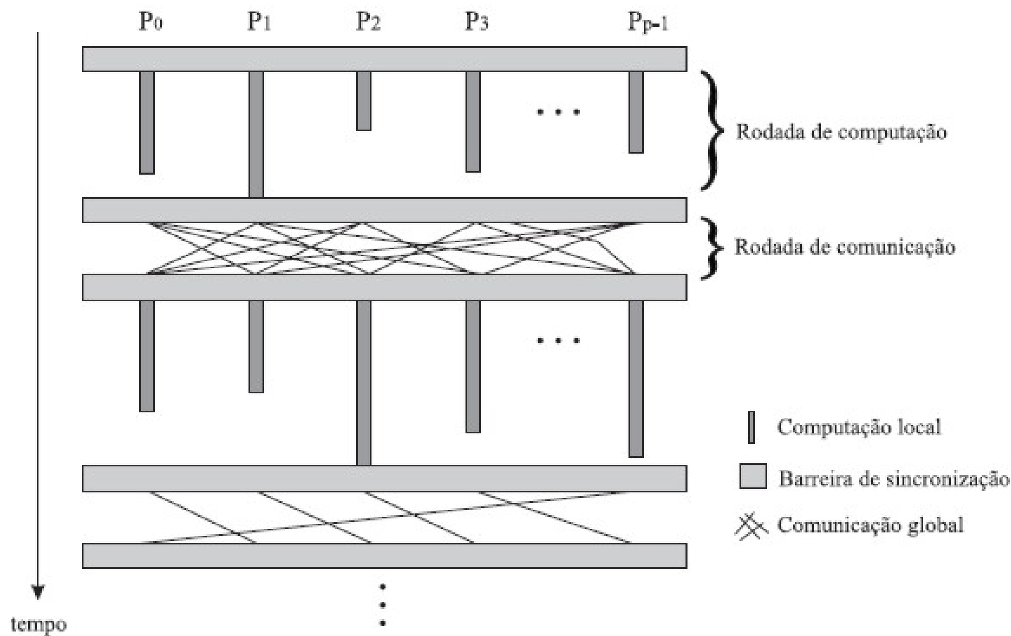


Figura 2.2: Modelo CGM [12]

O custo de um algoritmo CGM é determinado pela soma dos tempos gastos, tanto na computação local quanto na comunicação entre os processadores. Assim, o modelo CGM, como o modelo BSP descrito anteriormente, é chamado realístico por considerar o tempo gasto com a comunicação entre os processadores para computar o tempo total de execução dos algoritmos.

A característica que diferencia o modelo CGM do modelo BSP é que, além do número de parâmetros, no CGM os passos de computação local e de comunicação são separados por barreiras de sincronização, enquanto no BSP têm-se os passos de computação local junto com passos de comunicação entre processadores, porém os dados comunicados só estão disponíveis para o processamento após a sincronização.

2.3 MPI (*Message Passing Interface*)

O paradigma de troca de mensagens (*Message Passing*) consiste em um conjunto de funções que torna possível a criação, a gerência, a comunicação e a sincronização entre processos quando não existe uma memória compartilhada. Para permitir que linguagens como C, C++ e Fortran incorporassem esse paradigma, foram definidas extensões para essas linguagens, geralmente na forma de bibliotecas. Dentre as bibliotecas mais utilizadas para a troca de mensagens está o MPI (*Message Passing Interface*).

O MPI foi desenvolvido de modo a alcançar os seguintes objetivos:

- portabilidade do código fonte, permitindo que uma implementação seja usada em um ambiente heterogêneo sem alterações;
- prover uma interface de comunicação confiável, de tal forma, que o usuário não precise se preocupar com falhas na comunicação;
- definir uma interface que possa ser implementada em muitas plataformas sem mudanças muito significativas na parte de comunicação e do *software*.

Na biblioteca MPI, um conjunto básico de rotinas foi definido. Este conjunto inclui rotinas para:

- comunicação ponto-a-ponto e comunicação coletiva;
- suporte para grupo de processos (relaciona processos por grupos e os processos são classificados dentro de cada grupo);
- suporte para topologia de processos (definir uma estrutura topológica que descreve o relacionamento entre os processos).

Uma descrição detalhada de algumas das funcionalidades da biblioteca MPI pode ser encontrada em [5].

Capítulo 3

Mineração de Dados

3.1 Conceitos Gerais

De acordo com Elmasri e Navathe [9], mineração de dados é um termo que se refere à mineração ou descoberta de novas informações em uma grande quantidade de dados, usando padrões ou regras. As novas informações podem ser úteis para guiar decisões sobre atividades futuras, e para ser útil na prática, a mineração de dados deve ser realizada eficientemente em grandes arquivos ou bancos de dados.

A descoberta de conhecimento a partir de enormes volumes de dados é um desafio [24]. A mineração de dados, que tem como propósito enfrentar esse desafio, é considerada uma área fortemente interdisciplinar, já que agrega conceitos e técnicas de outras áreas para viabilizar a sua aplicação [10, 23]. Entre elas, pode-se destacar áreas como estatística, aprendizado de máquina, redes neurais e algoritmos genéticos [7]. Além disso, a mineração de dados pode ser relacionada a uma área mais ampla, conhecida como descoberta de conhecimento em bancos de dados (*Knowledge Discovery in Databases* - KDD). Segundo Mitra e Acharya [24], KDD é o processo não trivial que identifica padrões válidos, novos, potencialmente úteis e compreensíveis nos dados. O processo, como um todo, consiste em fazer com que dados em baixo nível tornem-se conhecimento em alto nível. O processo de KDD é dividido em várias fases [14] e as fases principais que o compõem são apresentadas a seguir:

1. **Seleção** - Seleciona da base de dados os dados relevantes para busca de padrões e conhecimento.

2. **Pré-processamento** - Organiza os dados e realiza o tratamento dos dados inconsistentes e incompletos.

3. **Transformação** - Transforma ou consolida os dados em formatos apropriados para o progresso de mineração de dados.

4. **Mineração de dados** - Processo essencial onde métodos inteligentes são aplicados para extrair padrões dos dados.

5. **Interpretação e avaliação** - Analisa os resultados da mineração e consolida o conhecimento descoberto para apresentar ao usuário.

A Figura 3.1 ilustra todo o processo de KDD em sequência.

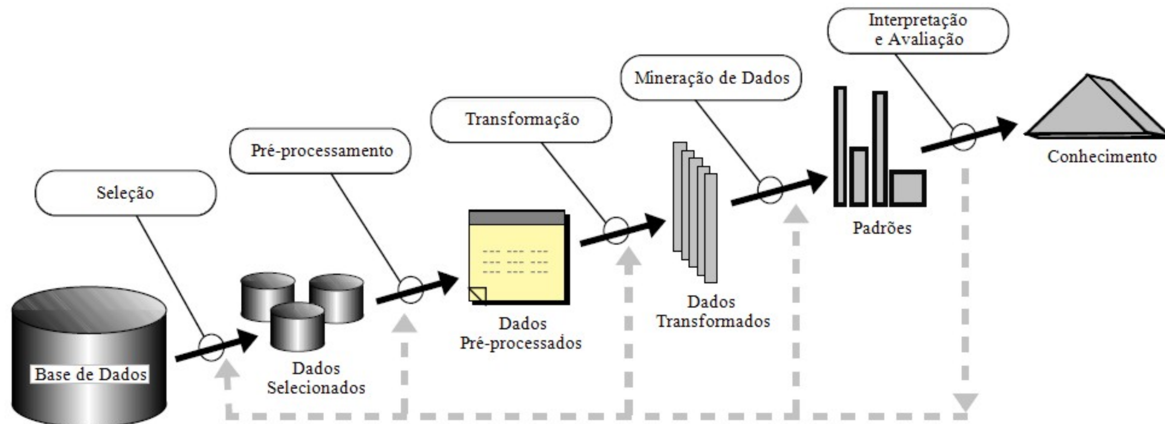


Figura 3.1: Fases do processo de KDD [10].

A mineração de dados é um passo essencial dentro do processo de KDD [22]. Por esse motivo, os termos mineração de dados e KDD são comumente tratados como sinônimos.

Na mineração de dados, não existe uma técnica específica que resolva todos os problemas abordados. Diferentes métodos servem para diferentes propósitos, cada método oferece suas vantagens e suas desvantagens. A familiaridade com as técnicas é necessária para facilitar a escolha de uma delas, de acordo com os problemas apresentados [31]. As técnicas de mineração mais conhecidas são: associação, classificação, agrupamento (*clustering*) e regressão. A tarefa de associação, que consiste na extração de regras de associação, é uma das mais utilizadas entre elas.

Este capítulo está organizado como se segue. Na Seção 3.2 é apresentada a técnica de regras de associação. Na Seção 3.3 são apresentadas as estruturas de dados utilizadas. Por fim, na Seção 3.4, são descritos alguns algoritmos sequenciais de extração de regras de associação.

3.2 Regras de Associação

Uma das técnicas mais predominantes em mineração de dados está na descoberta de regras de associação, inicialmente proposto por Agrawal [2]. No entanto, a base de dados deve ser tratada como uma coleção de transações, em que cada uma envolve um conjunto de itens [9]. Dessa forma, com a técnica de associação pode-se encontrar relações de associatividade entre os itens presentes nas transações da base de dados, ou melhor, descobrir que a presença de determinados itens podem implicar na ocorrência de outros itens [30].

Para uma melhor compreensão, a seguir são apresentadas algumas definições básicas para regras de associação.

- Seja I um conjunto de n itens, $I = \{i_1, i_2, \cdots, i_n\}$, e T um conjunto de m transações em uma base de dados, $T = \{t_1, t_2, \cdots, t_m\}$. Uma transação t_j, em que $1 \leq j \leq m$, é caracterizada como um conjunto R_j, tal que $R_j \subseteq I$, e possui um identificador único denominado *tid*.
- Um conjunto S, tal que $S \subseteq I$, é denominado um *itemset*. E um *itemset* composto por k itens é chamado de *k-itemset*.
- Todo *itemset* na base de dados é associado a um valor como parâmetro denominado **suporte**. A definição do valor é dada por $\sigma(S) = |\{R_j \mid 1 \leq j \leq m, S \subseteq R_j\}|$, em que refere-se ao número de transações nas quais o *itemset* ocorreu como um subconjunto.
- Dado um valor de suporte mínimo como entrada, denominado *min_sup*, todos os *itemsets* com valor de suporte maior ou igual ao valor *min_sup* são chamados de *itemsets* **frequentes**.
- O conjunto de todos os *k-itemsets* frequentes em uma base de dados é denominado de F_k.
- Um *itemset* frequente é chamado **máximo** se não for um subconjunto de qualquer outro *itemset* também frequente.

Uma regra de associação é uma expressão da forma $X \rightarrow Y$, em que X e Y são *itemsets*. O significado dessa regra é que as transações da base de dados que contém X tendem a conter Y também. O conjunto de itens que aparece à esquerda da seta (representado por X) é chamado de *antecedente* da regra. Já o conjunto de itens que aparece à direita da seta (representado por Y) é o *consequente* da regra. Assim, uma regra de associação tem o seguinte formato: $Antecendente \rightarrow Consequente$.

A cada regra de associação extraída de uma base de dados são atribuídos dois fatores: **suporte** e **confiança**. O **suporte** de uma regra é a probabilidade em comum de uma transação conter ambos X e Y, sendo definido como $\sigma(X \cup Y)/|T|$. E a **confiança** da regra é a probabilidade condicional de uma transação conter Y, dado que ela contém X, sendo definido como $\sigma(X \cup Y)/\sigma(X)$. Uma regra que possui o suporte maior ou igual que o valor *min_sup*, especificado como entrada, é denominada **frequente**, e é dita **forte** quando a sua confiança é maior ou igual que o valor da confiança mínima também dado como entrada, chamado *min_conf*.

Os algoritmos de mineração de regras de associação utilizam apenas as regras que apresentam suporte que satisfaz o valor *min_sup*, descartando as demais. As regras com alto grau de confiança são importantes porque possuem uma previsão bastante precisa da associatividade dos itens na regra. A mineração de regras de associação consiste em encontrar todas as regras que tenham um suporte maior ou igual que o valor *min_sup*, ditas regras frequentes, e aquelas que tenham uma confiança maior ou igual à *min_conf*, ditas regras fortes. Essa tarefa consiste em dois passos apresentados a seguir:

1. Encontrar todos os *itemsets* frequentes que tenham suporte mínimo. O espaço de busca para enumeração de todos os *itemsets* é 2^n, em que n é o número de de itens distintos.

2. Gerar regras fortes a partir de cada *itemset* frequente. A regra é gerada e sua confiança testada da seguinte forma: $X \backslash Y \rightarrow Y$, em que $Y \subset X$ e X é frequente. A complexidade do passo de geração da regra é $O(r.2^l)$, onde r é o número de *itemsets* frequentes, e l é o *itemset* frequente mais longo.

Seja, por exemplo, $I = \{$A, B, C, D$\}$ o conjunto de todos os possíveis itens em uma base de dados. Dessa maneira, tem-se $2^4 = 16$ possíveis *itemsets*. Na Figura 3.2 é apresentada uma estrutura, denominada *lattice*, que representa todos os possíveis *itemsets* do conjunto I. Nessa estrutura, a borda em destaque separa os *itemsets* frequentes (localizados na parte superior) dos *itemsets* não frequentes (localizados na parte inferior). A existência dessa borda é garantida pela propriedade de que para um *itemset* ser frequente, todos os seus subconjuntos também devem ser frequentes [4]. No entanto, o seu formato depende do suporte de cada *itemset* e do valor *min_sup* especificado.

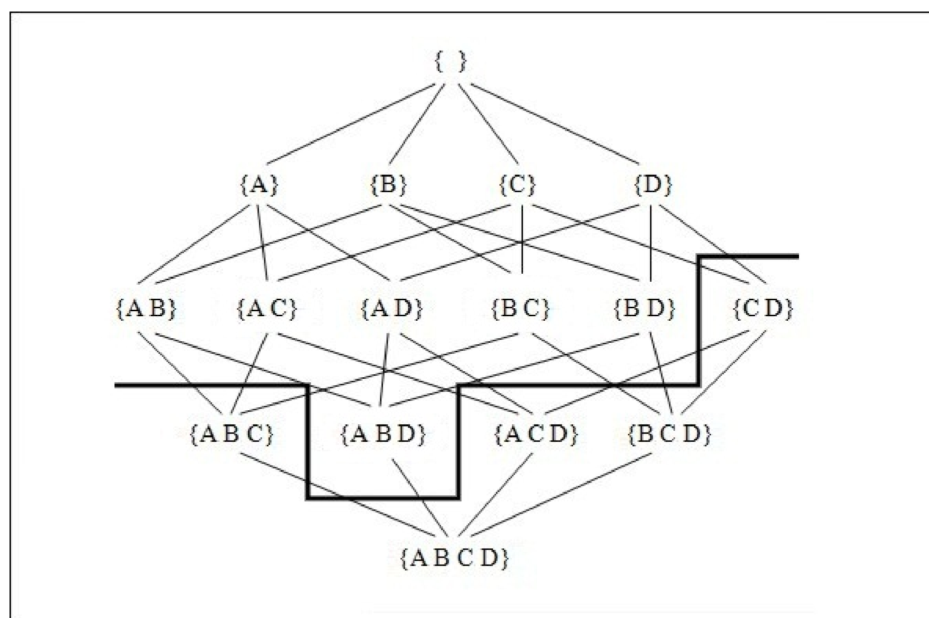


Figura 3.2: *Lattice* de $I = \{$A, B, C, D$\}$.

O principio básico dos algoritmos de mineração de regras de associação é empregar essa borda de maneira a diminuir a número de *itemsets* analisados quanto a serem frequentes ou não. Assim que a borda é encontrada, os algoritmos restrigem-se a determinar o suporte dos *itemsets* acima da mesma, e ignorar os que se encontram abaixo dela [16].

As estratégias mais comuns para se encontrar a borda entre os *itemsets* frequentes e não frequentes são a busca em largura e a busca em profundidade. A busca em largura determina o suporte de todos os (k-1)-*itemsets* antes de determinar o suporte dos k-*itemsets*. Em constraste, a busca em profundidade desce recursivamente a estrutura apresentada na Figura 3.2.

Um *itemset*, que possivelmente é frequente durante a fase em que se determina o seu suporte, é denominado *itemset* candidato. A abordagem mais comum para se calcular o suporte de um *itemset* candidato é contar diretamente na base de dados a sua ocorrência. Para esse propósito, um contador é iniciado com zero e todas as transações da base de dados são analisadas. Cada vez que o *itemset* candidato é reconhecido como subconjunto de uma transação, seu contador é incrementado.

Para ilustrar a técnica de descoberta de regras de associação, considere o exemplo mostrado na Figura 3.3. Cada transação na base de dados é composta por itens pertencentes ao conjunto $I = \{A, B, C, D, E\}$. A figura mostra todos os *itemsets* presentes em pelos menos três transações (*min_sup* = 50%). A figura também mostra o conjunto de todas as regras de associação com o valor *min_conf* = 100%.

Base de Dados com as Transações

Tid	Itens
1	A B D E
2	B C E
3	A B D E
4	A B C E
5	A B C D E
6	B C D

Itemsets Freqüentes (*min_sup* = 50%)

Suporte	Itemsets
100% (6)	B
83% (5)	E, BE
67% (4)	A, C, D, AB, AE, BC, BD, ABE
50% (3)	AD, CE, DE, ABD, ADE, BCE, BDE, ABDE

Itemsets Freqüentes Maximal = BCE, ABDE

Regras de Associação (*min_conf* = 100%)

A → B (4/4)	AB → E (4/4)	DE → B (3/3)
A → E (4/4)	AD → B (3/3)	AD → BE (3/3)
A → BE (4/4)	AD → E (3/3)	DE → AB (3/3)
C → B (4/4)	AE → B (4/4)	ABD → E (3/3)
D → B (4/4)	CE → B (3/3)	ADE → B (3/3)
E → B (5/5)	DE → A (3/3)	BDE → A (3/3)

Figura 3.3: Exemplo de mineração de regras de associação de uma base de dados.

Entretanto, após a fase de descoberta de regras de associação, os seguintes problemas podem ser encontrados: grande quantidade de regras geradas (causando dificuldades para a leitura das regras) e alta frequência de regras sem qualidade (apresentando informações redundantes ou sem sentido). Diante disso, uma outra fase na mineração de dados é requerida, denominada de pós-processamento.

Na fase de pós-processamento, algumas abordagens existentes podem ser empregadas para auxiliar na análise das regras extraídas. Dentre as abordagens mais conhecidas está o uso de taxonomias, na qual proporciona uma visão coletiva ou individual de como os itens podem ser hierarquicamente classificados, podendo ser utilizados para eliminar regras não interessantes e/ou redundantes [8].

3.3 Estruturas de Dados

Os diversos algoritmos de descoberta de regras de associação requerem uma estrutura de dados para determinar o suporte dos *itemsets* candidatos. Essas estruturas possibilitam a redução da quantidade de acessos a bases de dados durante a contagem do suporte dos *itemsets*. As estruturas de dados comumente utilizadas são projeção vertical de bases de dados e *hash-tree*.

3.3.1 Projeção Vertical de Bases de Dados

A maioria das bases de dados são compostas por um conjunto de transações que são armazenadas no formato horizontal. Nesse formato, cada *tid* na base de dados é seguido pelo conjunto de itens que formam a transação. Entretanto, esse formato impõe algumas restrições computacionais durante a fase de contagem dos suportes, uma vez que ele força que toda a base de dados seja acessada a cada valor de suporte determinado [30]. Na Figura 3.3, a base de dados como exemplo está no formato horizontal.

Uma técnica utilizada para diminuir os custos computacionais na determinação dos suportes é chamada de projeção vertical da base de dados. Através dessa técnica, uma estrutura de dados é obtida a partir de uma base de dados horizontal, de modo que as transações são dispostas em formato vertical. Nesse formato, um item da base de dados é associado a uma lista contendo os *tids* de todas as transações em que o item ocorreu na base de dados horizontal, sendo esta lista denominada *tidlist*. Na estrutura, um *itemset* S associado à sua *tidlist* é representado por $S.tidlist$.

A principal vantagem da base de dados estar representada em formato vertical está na determinação dos suportes dos *itemsets*. Uma vez que a base de dados horizontal teria que ser percorrida inteiramente para determinar o suporte de um *itemset* S, com a projeção vertical de dados basta verificar o tamanho da *tidlist* de S, definido por $\sigma(S) = |S.tidlist|/|\mathrm{T}|$, em que T é o conjunto de transações na base de dados horizontal. Como um *k-itemset* S é gerado unindo dois (k-1)-*itemsets* X e Y, definido por $S = X \cup Y$, a *tidilist* de S pode ser criada apenas fazendo a intersecção das *tidlists* dos dois *itemsets* geradores X e Y, definida como $S.tidlist = X.tidlist \cap Y.tidlist$. Através desse processo, pode-se gerar todos os possíveis *itemsets* com suas respectivas *tidlists*, iniciando a partir dos *itemsets* de tamanho 1. Além disso, a estrutura de dados no formato vertical é inicialmente construída com apenas uma varredura na base de dados, nos quais são criadas as *tidlists* referentes aos *itemsets* de tamanho 1. Outra vantagem observada é que quanto maior for um *itemset*, menor será a sua *tidlists*, o que implica em rapidez nas intersecções realizadas.

A estrutura de dados no formato vertical é utilizada pelo Algoritmo Eclat (algoritmo apresentado no modelo sequencial na Subseção 3.4.5 e no modelo paralelo na Seção 4.3) durante a mineração de regras de associação. A Figura 3.4 mostra, como exemplo, uma parte da estrutura de dados em formato vertical obtida a partir da base de dados da Figura 3.3.

Itemset	Lista de Tids
A	1 3 4 5
B	1 2 3 4 5 6
C	2 4 5 6
D	1 3 5 6
E	1 2 3 4 5
AB	1 3 4 5
AC	4 5
...	...

Figura 3.4: Exemplo de base de dados no formato vertical.

3.3.2 *Hash-Tree*

A *hash-tree* é uma estrutura de dados utilizada para facilitar a contagem de suporte dos *itemsets*. Na estrutura, o conjunto C_k, conjunto formado por *k-itemsets* obtidos a partir de um conjunto de n itens, é distribuído nas folhas da árvore, enquanto todos os nós internos são tabelas *hash* de dimensão k, de forma que cada nó aponta para no máximo k filhos.

Para adicionar um *k-itemset* na estrutura a árvore é percorrida a partir da raiz, de modo que, estando numa profundidade d, em que $d \leq k$, o próximo ramo a seguir é determinado aplicando uma função *hash* sobre o item na posição d do *k-itemset*. Repetindo o processo no máximo k vezes, o nó folha é encontrado, no qual este deverá armazenar o *k-itemset*. A capacidade de armazenamento de um nó folha é de $|C_k|/k$, em que $|C_k| \leq n!/((n\text{-}k)!/2)$. Quando a capacidade de armazenamento é excedida, o nó folha é convertido em um nó interno e os *itemsets* alocados são redistribuídos entre as novas folhas através da aplicação da função *hash*.

A Figura 3.5 mostra o exemplo de uma *hash-tree* construída a partir do seguinte conjunto de 3-*itemsets*: C_3 = {ADE, ABE, ACF, BCD, CEF, DEG, EFG, FHI, AEI}, em que cada nó folha tem capacidade para armazenar no máximo três *itemsets*, de modo que quando um quarto *itemset* é adicionado a um nó folha, ele é transformado em uma tabela *hash*.

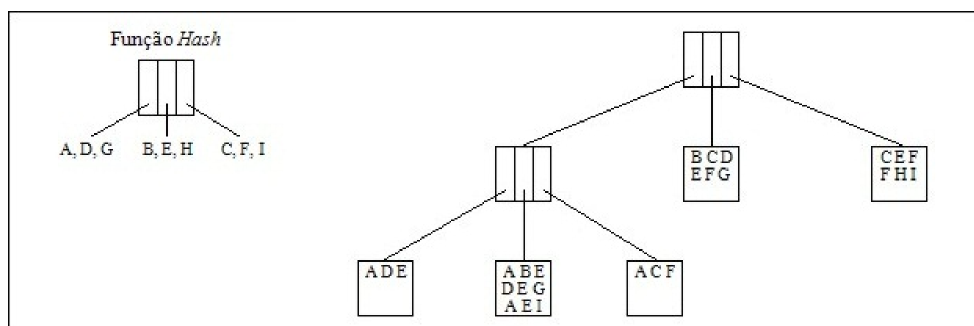


Figura 3.5: Exemplo de *hash-tree*.

A Figura 3.6 apresenta um exemplo de adição de um *itemset* na árvore da Figura 3.5, de maneira que a capacidade de armazenamento do nó folha é excedida.

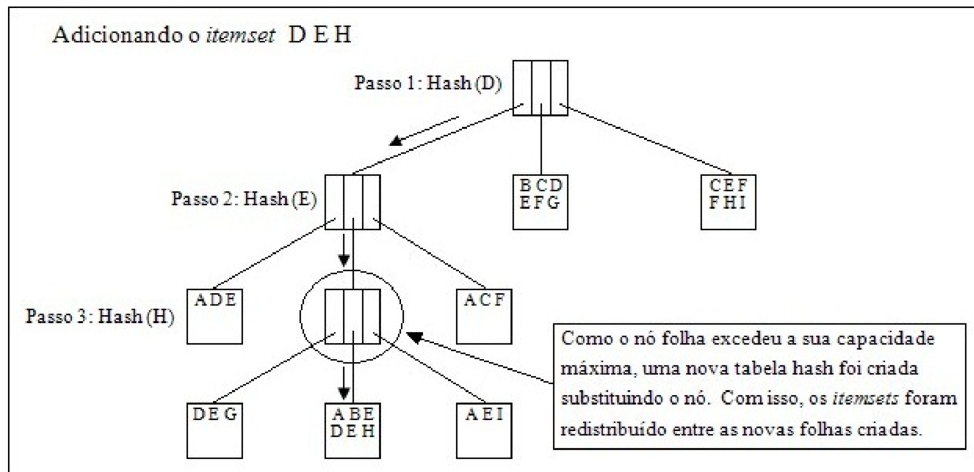


Figura 3.6: Exemplo de inserção de um itemset na *hash-tree*.

A principal vantagem da *hash-tree* é que todo *itemset* é acessado rapidamente para o incremento do seu suporte durante a leitura das transações, o que acelera o processo de determinação dos *itemsets* frequentes. Cada transação da base de dados é mapeada na *hash-tree* através da aplicação da função *hash* a cada nó da árvore, exceto nas folhas.

A estrutura de dados *hash-tree* é utilizada pelo Algoritmo Apriori (algoritmo apresentado no modelo sequencial na Subseção 3.4.1 e no modelo paralelo na Seção 4.2) durante a descoberta de regras de associação. A Figura 3.7 mostra o exemplo de uma transação sendo mapeada na árvore da Figura 3.6.

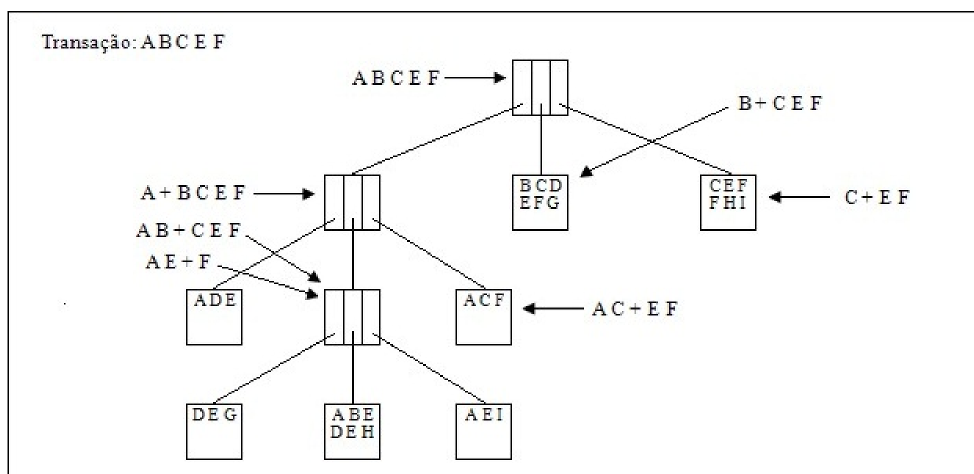


Figura 3.7: Exemplo de mapeamento de uma transação na *hash-tree*.

3.4 Algoritmos Sequenciais

Os primeiros algoritmos de mineração de regras de associação foram o AIS [2] e o SETM [17]. No entanto, logo surgiram outros algoritmos com grandes vantagens de eficiência em relação a estes. Hoje, dentre os algoritmos sequenciais mais conhecidos para a descoberta de regras de associação, estão: Apriori [4], AprioriTid [4], Partition [27], *FP-Growth* [15] e o Eclat [33].

3.4.1 Algoritmo Apriori

Entre os diversos algoritmos que realizam a mineração de dados buscando regras de associação, o mais conhecido e utilizado é o Algoritmo Apriori [4]. A sua criação representou um grande diferencial em relação aos seus predecessores, principalmente no que se refere ao desempenho e à estratégia de solução do problema de mineração de regras de associação [32]. Por esse motivo, o Algoritmo Apriori é considerado um algoritmo clássico, e a partir dele muitos algoritmos foram posteriormente criados, formando o que muitos chamam de "Família Apriori"[26].

A estratégia do Algoritmo Apriori é inicialmente identificar os conjuntos de *itemsets* frequentes (cujo suporte seja maior ou igual a *min_sup*) e, em seguida, construir regras de associação a partir desses conjuntos, nos quais possuam confiança maior ou igual a *min_conf*. A ideia inovadora desse algoritmo está na criação dos *itemsets* candidatos usando a propriedade de conjuntos que garante que, se um conjunto de itens não for frequente, então todos os seus superconjuntos também não são frequentes. Com isso, o algoritmo ganha em desempenho, já que não perde tempo analisando esses superconjuntos. Essa otimização é possível porque a busca em largura garante que os valores dos suportes de todos os subconjuntos de um *itemset* candidato são conhecidos antecipadamente.

Para determinar o suporte dos *itemsets* candidatos realiza-se uma varredura na base de dados para cada conjunto de *itemsets* candidatos de tamanho k. A parte crítica dessa tarefa está na procura pelos *itemsets* candidatos em cada transação da base de dados. Porém, esta tarefa pode ser realizada utilizando a estrutura de dados *hash-tree*, apresentada na seção anterior. A função principal do Algoritmo Apriori é apresentada a seguir.

1. **Entrada**: Uma base de dados D e o valor de suporte mínimo *min_sup*.
2. **Saída**: O conjunto L com todos os *itemsets* frequentes.
3. **Função** *apriori-main*(D, *min_sup*)
4. $L_1 = \{$conjunto dos *itemsets* frequentes de tamanho 1 contidos em $D\}$;
5. **para** ($k = 2$; $L_{k-1} \neq \emptyset$; k++)
6. C_k = *apriori-gen*(L_{k-1});
7.   **para** todas transações $t \in D$ **fazer**
8. C_t = *subset*(C_k, t);
9.   **para** todos candidatos $c \in C_t$ **fazer**
10.   *c.count*++;
11.   **fim para**
12.   **fim para**

13. $L_k = \{c \in C_k \mid c.count \geq min_sup\}$;
14. **fim para**
15. **retorne** $L = \cup_k L_k$;

Pode-se notar que o algoritmo faz uso de duas funções, *apriori-gen* e *subset*, em que cada uma realiza operações específicas. A função *apriori-gen*, chamada na linha 6 do algoritmo, recebe como argumento o conjunto L_{k-1} (composto por todos os (k-1)-*itemsets* frequentes) e retorna o conjunto C_k (composto por todos *k-itemsets* candidatos). A tarefa realizada por essa função é dividida em dois passos: *join* (junção) e *prune* (ajuste). No passo *join* são geradas as combinações através da junção de L_{k-1} com L_{k-1}. Já no passo *prune* são eliminados os *itemsets* candidatos que possuem algum subconjunto não frequente. Em outras palavras, é feito um ajuste no qual são removidos todos os *k-itemsets* $c \in C_k$ se existe um (k-1)-*itemset* s, em que $s \subset c$ e $s \notin L_{k-1}$. Esses dois passos podem ser melhores compreendidos a seguir.

1. **Função** *apriori-gen*(L_k)
// Passo 1 (*join*)
2. **para** cada *itemset* $l_1 \in L_{k-1}$ **fazer**
3. **para** cada *itemset* $l_2 \in L_{k-1}$ **fazer**
4. **se** ($l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge ... \wedge l_1[k\text{-}1] < l_2[k\text{-}1]$) **então**
5. $c = l_1[1] \,.\, l_1[2] \,...\, l_1[k\text{-}2] \,.\, l_1[k\text{-}1] \,.\, l_2[k\text{-}1]$;
6. **adicione** c em C_k;
7. **fim se**
8. **fim para**
9. **fim para**
// Passo 2 (*prune*)
10. **para** todos candidatos $c \in C_k$ **fazer**
11. **para** todos (k-1)-*subsets* $s \subset c$ **fazer**
12. **se** ($s \notin L_{k-1}$) **então**
13. **delete** c de C_k;
14. **fim se**
15. **fim para**
16. **fim para**
17. **retorne** C_k**;**

A função *subset*, chamada na linha 8 da função principal do algoritmo, recebe como argumento o conjunto C_k com os *itemsets* já podados e uma transação t da base de dados, e retorna o conjunto C_t composto pelos *itemsets* presentes em t. Essa função utiliza a *hash-tree* para encontrar os *itemsets* de C_k em uma transação t. Com isso, o suporte de cada *itemset* c em C_k é determinado no propósito de eliminar os *itemsets* que não são frequentes na base de dados de entrada.

Após gerar o conjunto L contendo todos os *itemsets* frequentes, pode-se iniciar a fase de extração de regras de associação. Para isso, o Algoritmo Apriori faz uso da função *ap-genrules*, mostrada a seguir.

1. **Entrada**: Um conjunto de *itemsets* L e a confiança mínima da regra *min_conf*.
2. **Saída**: O conjunto de regras R.
3. **Função** *ap-genrules*(L, *min_conf*)
4. **para** todos *k-itemsets* $\in L$ **fazer**
5. **para** ($i = k$-1; $i \geq 1$; $i--$)
6. **para** todos *i-itemsets* $\subset$ *k-itemset* **fazer**
7. conf = suporte(*k-itemset*) / suporte(*i-itemset*);
8. **se** (conf $\geq$ *min_conf*) **então**
9. **adicione** *i-itemset* $\rightarrow$ (*k-itemset* $-$ *i-itemset*) em R;
10. **fim se**
11. **fim para**
12. **fim para**
13. **fim para**
14. **retorne** R;

Nessa função, o conjunto L é recebido como argumento juntamente com o valor *min_conf* (valor de confiança mínima requisitada para as regras). A partir de cada *item-set* presente em L, são extraídos todos os subconjuntos de ítens possíveis, nos quais para cada subconjunto, calcula-se a confiança da regra a ser gerada, definida por: confiança = suporte(X)/suporte(Y), com $X = \{k\text{-}itemset \mid k\text{-}itemset \in L\}$ e $Y = \{1 \leq i < k \mid i\text{-}itemset \subset k\text{-}itemset\}$. Caso o valor da confiança satisfaça o valor mínimo requisitado, a regra pode então ser extraída, sendo especificada por: $Y \rightarrow X$-Y. Assim, todas as possíveis regras de associação são geradas a partir do conjunto de *itemsets* frequentes, e serão exibidas ao final da execução do algoritmo.

3.4.2 Algoritmo AprioriTid

O Algoritmo AprioriTid, também proposto por Agrawal e Srinkant [4], possui características idênticas ao Algoritmo Apriori. A única diferença é que esse algoritmo acessa a base de dados apenas uma única vez, na qual é determinado o conjunto L_1 (composto por todos *itemsets* frequentes de tamanho 1). Esse algoritmo utiliza uma estrutura denominada $\overline{C_k}$, que representa a base de dados, de modo que, para $k \geq 2$, a base de dados não precisa mais ser acessada para determinar o suporte dos *itemsets* candidatos.

A estrutura de dados $\overline{C_k}$ é composta por um conjunto de membros relacionados às transações da base de dados. Cada membro de $\overline{C_k}$ possui a forma $< tid, X_k >$, em que X_k representa um conjunto de *k-itemsets* presentes na transação identificada pelo valor *tid*. Para $k = 1$, $\overline{C_1}$ corresponde à própria base de dados D. Para $k \geq 2$, o membro de $\overline{C_k}$ correspondente a uma transação t é definido como $<t.tid, \{c \in C_k \mid c \subseteq t\}>$.

O Algoritmo AprioriTid utiliza a mesma função *apriori-gen* do Algoritmo Apriori para gerar o conjunto C_k, composto por todos os *k-itemsets* candidatos. A partir do conjunto C_k, o Algoritmo AprioriTid utiliza a estrutura $\overline{C_{k-1}}$ para determinar o suporte dos *k-itemsets* candidatos e, assim, obter o conjunto L_k. O Algoritmo AprioriTid é apresentado a seguir.

1. **Entrada**: Uma base de dados D e o valor de suporte mínimo *min_sup*.
2. **Saída**: O conjunto L com todos os *itemsets* frequentes.
3. **Função** *aprioritid-main*(D, *min_sup*)
4. L_1 = {Conjunto dos *itemsets* frequentes de tamanho 1 contidos em D};
5. C_1 = base de dados D;
6. **para** ($k = 2$; $L_{k-1} \neq \emptyset$; $k{+}{+}$)
7. $\quad C_k$ = *apriori-gen*(L_{k-1});
8. $\quad \overline{C_k} = \emptyset$;
9. $\quad$ **para** todas entradas $t \in \overline{C_{k-1}}$ **fazer**
10. $\qquad C_t = \{c \in C_k | (c\text{ - }c[k]) \in t.itemsets \wedge c \in C_k | (c\text{ - }c[k\text{-}1]) \in t.itemsets\}$;
11. $\qquad$ **para** todos candidatos $c \in C_t$ **fazer**
12. $\qquad\quad$ *c.count++*;
13. $\qquad$ **fim para**
14. $\qquad$ **se** ($C_t \neq \emptyset$) **então**
15. $\qquad\quad \overline{C_k} = \overline{C_k} + <t.tid, C_t>$;
16. $\qquad$ **fim se**
17. $\quad$ fim para
18. $\quad L_k = \{c \in C_k \mid c.count \geq min_sup\}$;
19. **fim para**
20. **retorne** $L = \cup_k L_k$;

Como pode-se observar, os *itemsets* de C_k são procurados em cada entrada t de $\overline{C_{k-1}}$, em que uma entrada contém um conjunto de *itemsets* contidos em uma transação t. Todos os *itemsets* de C_k presentes numa entrada t são armazenados em um conjunto temporário C_t, no propósito de facilitar a contagem dos suportes. Feito isso, o conjunto C_t, juntamente com o *tid* da transação t, são adicionados à estrutura $\overline{C_k}$. Assim, percorrida todas as entradas de $\overline{C_{k-1}}$, obtém-se toda a estrutura de dados $\overline{C_k}$, além do conjunto L_k composto pelos *k-itemsets* frequentes.

Após a determinação do conjunto L, composto por todos *itemsets* frequentes, pode-se realizar a extração de regras de associação. Para isso, o AprioriTid utiliza também a função *ap-genrules* do Algoritmo Apriori. Dessa forma, pode-se observar que a única diferença do Algoritmo AprioriTid em relação ao Algoritmo Apriori está na utilização de uma estrutura auxiliar, a qual evita os acessos constantes à base de dados, durante a contagem dos suportes.

3.4.3 Algoritmo Partition

O Algoritmo Partition, proposto por Saverese et al [27], utiliza a técnica de intersecções entre conjuntos para determinar os valores de suporte. De modo semelhante aos Algoritmos Apriori e AprioriTid, a contagem do suporte de todos os *itemsets* candidatos de tamanho k-1 é realizada antes de contar o suporte dos *itemsets* candidatos de tamanho k. O Algoritmo Partition utiliza as *tidlists* dos *itemsets* frequentes de tamanho k-1 para gerar as *tidlists* dos *itemsets* candidatos de tamanho k. Porém, pode ocorrer do tamanho dos resultados intermediários ficar maior que a memória física disponível. Para evitar esse problema, o Algoritmo Partition divide a base de dados em várias partes que são tratadas de maneira independente.

O tamanho de cada parte é escolhido de maneira que todas as suas *tidlists* intermediárias caibam na memória. Depois de determinar os *itemsets* frequentes em cada parte, uma passagem extra sobre a base de dados é necessária para garantir que os *itemsets* que são frequentes localmente também sejam frequentes em toda a base de dados. A chave para o funcionamento do algoritmo é que um *itemset* frequente em toda a base de dados é frequente em pelo menos uma das n partições [27]. Algumas notações utilizadas no algoritmo são apresentadas na Tabela 3.1.

Tabela 3.1: Notação usada no algoritmo Partition.

Notação	**Significado**
p_i	*i-ésima* partição da base de dados D
L_k^i	Conjunto local de *k-itemsets* frequentes em uma partição p_i
C^G	Conjunto global de *itemsets* candidatos
L^i	Conjunto local de *itemsets* frequentes em uma partição p_i
L_k^G	Conjunto global de *k-itemsets* frequentes
L^G	Conjunto global de *itemsets* frequentes

Na estratégia do algoritmo, inicialmente, a base de dados D é particionada logicamente em n partições, em que cada partição é identificada por p_i, e $0 \leq i < n$. Em seguida, o algoritmo é dividido em duas fases de execução.

A primeira fase é composta por n iterações, em que durante a iteração i, somente a partição p_i é considerada. Cada partição p_i é convertida para o formato vertical, em que cada item é associado a uma lista contendo o *tid* de todas as transações nas quais esse item ocorreu, ou seja, é construída a *tidlist* de todos os 1-*itemsets*. Com isso, o suporte de um *k-itemset* é obtido a partir da intersecção das *tidlists* de dois (k-1)-*itemsets*. Em cada partição p_i, é determinado o conjunto L^i que corresponde a todos os *itemsets* frequentes localmente, usando a definição que se um *itemset* é frequente em toda a base de dados, então ele também é frequente em pelos menos uma das partições. Com isso, o conjunto de *itemsets* candidatos é reduzido nessa fase. Após determinar o conjunto L^i em cada partição p_i, todos os n conjuntos são reunidos formando o conjunto C^G que corresponde ao conjunto global de *itemsets* candidatos.

Na segunda fase, o suporte de cada *itemset* candidato em C^G é novamente determinado, porém, agora, a contagem é feita utilizando toda a base de dados. Nessa fase, inicialmente, a base de dados é percorrida uma vez, de modo que, em cada partição p_i, são construídas as *tidlists* dos itens que estão presentes. A seguir, são realizadas n iterações para determinar o valor de suporte global dos *itemsets* candidatos. Em cada iteração i é calculado o suporte local de cada candidato na partição p_i, através da estratégia de intersecção das *tidlists* dos itens que compõem o *itemset*. Dessa forma, após percorrer todas as partições, tem-se o valor do suporte global de todos *itemsets* candidatos de C^G. Logo, é determinado o conjunto L^G, que corresponde ao conjunto global de todos *itemsets* frequentes. A função principal do Algoritmo Partition é apresentada a seguir.

1. **Entrada**: Uma base de dados D e o valor de suporte mínimo *min_sup*.
2. **Saída**: O conjunto L^G com todos os *itemsets* freqüentes.
3. **Função** *partition-main*(D, *min_sup*)
4. **particione** a base de dados D em n partições;

// Fase 1

5. **para** todas partições $p_i \in P$ *fazer*
6. L_i = *partition-gen*(p_i);
7. **fim para**
8. $C^G = \cup_{i=1,2,...,n} L^i$;

// Fase 2

9. **para** todas partições $p_i \in P$ **fazer**
10. *gen-count*(C^G, p_i);
11. **fim para**
12. $L^G = \{c \in C^G \mid c.count \geq min_sup\}$;
13. **retorne** L^G;

Esse algoritmo faz uso de duas funções, chamadas *partition-gen* e *gen-count*. A função *partition-gen*, chamada na linha 6 do algoritmo, corresponde à primeira fase do algoritmo, na qual, em cada partição p_i, são gerados todos os *itemsets* candidatos locais e determinado o conjunto daqueles que são frequentes. A função *partition-gen* é mostrada a seguir.

```
1.  Função partition-gen(p_i)
2.  L^i_1 = {Conjunto de todos itemsets frequentes de tamanho 1 contidos em p_i com
    suas tidlists};
3.  L^i = ∅;
4.  para (k = 2; L^i_{k-1} ∈ ∅; k++)
5.      para cada itemset l_1 ∈ L^i_{k-1} fazer
6.          para cada itemset l_2 ∈ L^i_{k-1} fazer
7.              se (l_1[1] = l_2[1] ∧ l_1[2] = l_2[2] ∧ ... ∧ l_1[k-1] < l_2[k-1]) então
8.                  c = l_1[1] . l_1[2] ... l_1[k-2] . l_1[k-1] . l_2[k-1];
9.                  se (existe (k-1)-subsets s ⊂ c, tal que s ∉ L_{k-1}) então
10.                     delete c;
11.                 senão
12.                     c.tidlist = l_1.tidlist ∩ l_2.tidlist;
13.                     se (|c.tidlist| / |p_i| ≥ min_sup) então
14.                         L^i_k = L^i_k ∪ {c};
15.                     fim se
16.                 fim se
17.             fim se
18.         fim para
19.     fim para
20. L^i = L^i ∪ L^i_k;
21. fim para
22. retorne L^i
```

A função *gen-count*, chamada na linha 10 do algoritmo, está relacionada à segunda fase do algoritmo, na qual é realizada a contagem global do suporte dos *itemset* candidatos. Para extrair as regras de associação de todos *itemsets* frequentes globalmente, o Algoritmo

Partition também pode utilizar a função *ap-genrules*, igualmente aos Algoritmos Apriori e AprioriTid. A função *gen-count* é apresentada a seguir.

1. **Função** *gen-count*(C^G, p_i)
2. **para** todos 1-*itemsets* $c \in p_i$ **fazer**
3.   **gerar** *c.lidlist*;
4. **fim para**
5. **para** ($k = 2$; $L^G_{k-1} \neq \emptyset$; $k{+}{+}$)
6.   **para** todos *k-itemsets* $c \in C^G$ **fazer**
7. $templist = c[1].tidlist \cap c[2].tidlist \cap ... \cap c[k].tidlist$;
8. $c.count = c.count + |templist|$;
9.   **fim para**
10. **fim para**

3.4.4 Algoritmo *FP-Growth*

O Algoritmo *FP-Growth* [15] é composto por duas fases de processamento. Na primeira fase é construída uma representação altamente condensada da base de dados, chamada *FP-Tree*. A geração da *FP-Tree* é feita através da estratégia de busca em profundidade e da contagem de ocorrência dos itens. Na segunda fase o *FP-Growth* utiliza a *FP-Tree* para determinar os valores de suporte para todos os *itemsets* frequentes. O processo de construção da *FP-Tree* e o Algoritmo *FP-Growth* são mostrados a seguir.

1. **Entrada**: Uma base de dados D e o valor de suporte mínimo *min_sup*.
2. **Saída**: O conjunto L com todos os *itemsets* frequentes.

// Fase 1 - Construção da *FP-Tree*

3. **Função** *FP-Tree*(D, *min_sup*)
4. **percorra** a base de dados D uma vez;
5. **determine** o conjunto de itens frequentes F e seus suportes;
6. **ordene** F em ordem decrescente em função do suporte e chamá-la de L;
7. **crie** a raiz da *FP-Tree* T e coloque como "*null*";
8. **para cada** transação t em D **faça**
9.   **selecione e ordene** os itens frequentes em t de acordo com a ordem de L, tornando a lista de itens frequentes em t igual a $[p|P]$, onde p é o primeiro elemento e P é o resto da lista;
10.   **execute** *insere_tree*($[p|P]$, T).
11.   **se** T tiver um filho N em que $N.nome_item = p.nome_item$ **então**
12.   **incremente** o contador de N por em 1;
13.   **senão**
14.   **crie** um novo nó N, e inicie seu contador com 1;
15.   **ligue** o seu *parent-link* a T, e seu *node-link* aos nós de mesmo (*nome_item*) através da estrutura dos *node-links*;
16.   **fim se**
17.   **se** P não for vazio **então**
18.   **chame** *insere_tree*(P, N) recursivamente;
19.   **fim se**
20. **fim para**

// Fase 2 - Mineração da *FP-Tree*
21. **Função** *FP-Growth*(*Tree*, α)
22. **se** *Tree* contém apenas um caminho P **então**
23. **para** cada combinação β de nós no caminho P **faça**
24. **gere** o padrão $\beta \cup \alpha$ com suporte = *min_sup* dos nós em β;
25. **fim para**
26. **senão**
27. **para** cada a_i na tabela de *node-links* de *Tree* **faça**
28. **gere** o padrão $\beta = a_i \cup \alpha$ com suporte = $a_i.suporte$;
29. **construa** a base de padrões condicionada de β e crie a *FP-Tree* condicionada de β chamada de $Tree_\beta$;
30. **se** ($Tree_\beta \neq \emptyset$) **então**
31. *FP-growth*($Tree_\beta$, β);
32. **fim se**
33. **fim para**
34. **fim se**

Para uma melhor compreenção do funcionamento do Algoritmo *FP-Growth* e da criação da *FP-Tree*, considere o exemplo a seguir. As duas primeiras colunas da Tabela 3.2 mostram um exemplo de base de dados de transações e a terceira coluna apresenta a lista de itens frequentes ordenados em função do suporte. No exemplo o valor de suporte mínimo é de 60%.

Tabela 3.2: Base de dados de transações

Tid	**Itens**	**Itens Frequentes (Ordenados)**
100	F A C D G I M P	F C A M P
200	A B C F L M O	F C A B M
300	B F H J O	F B
400	B C K S P	B C P
500	A F C E L P M N	F C A M P

Considerando o exemplo apresentado na Tabela 3.2, inicialmente, percorrendo a base de dados, é determinada a lista de itens frequentes {(F:4), (C:4), (A:3), (B:3), (M:3), (P:3)}, em que a forma (x:n) representa o item x e a quantidade n de ocorrências de x na base de dados. Para a construção da *FP-Tree*, a lista de itens é então ordenada em ordem decrescente em função do seu suporte e, na sequencia, é criada a raiz da *FP-Tree* tendo como rótulo "*null*". Feito isso, a base de dados é percorrida uma segunda vez, de forma que, a primeira transação leva à construção do primeiro ramo da árvore, {(F:1), (C:1), (A:1), (M:1), (P:1)}. Para a inserção da segunda transação, como a sua lista de itens frequentes (F, C, A, B, M) compartilha o mesmo prefixo (F, C, A) com o ramo existente (F, C, A, M, P), o contador de cada nó prefixo é incrementado, e um novo nó (B:1) é criado e ligado como filho de (A:2), e um outro nó (M:1) é criado e ligado como filho de (B:1). Para a terceira transação, como sua lista de itens frequentes (F, B) compartilha apenas o nó (F) com o ramo da *FP-Tree* iniciado com prefixo F, o contador de F é incrementado em 1 e um novo nó (B:1) é criado e ligado como filho de (F:3). A quarta transação leva à construção do segundo ramo da *FP-Tree*, {(C:1), (B:1), (P:1)}, já que essa transação não

tem um prefixo em comum com o ramo já existente. E para a última transação, já que sua lista de itens frequentes é idêntica ao da primeira, todos os nós que participam dessa lista têm o seu contador incrementado em 1. Para se percorrer mais facilmente a *FP-tree*, uma tabela é criada na qual cada item aponta para uma sequência de nós com mesmo nome através de uma estrutura chamada *node-links*. Um nó é ligado aos seus filhos através de um *parent-link*. A Figura 3.8 apresenta a *FP-Tree* criada.

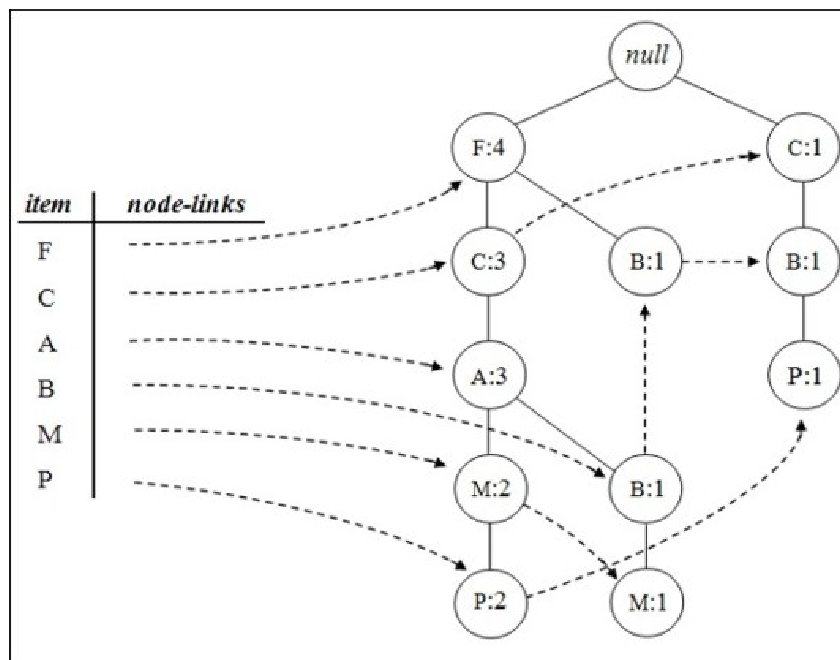


Figura 3.8: *FP-Tree* construída.

O processo de mineração da *FP-Tree* começa de baixo para cima na tabela de *node-links*. Iniciando pelo nó P, deriva-se o *itemset* (P:3) e os seguintes caminhos na *FP-Tree*: {F:4, C:3, A:3, M:2, P:2} e {C:1, B:1, P:1}. O primeiro caminho mostra que (F, C, A, M, P) aparece 2 vezes na base de dados. Apesar de (F, C, A) aparecer 3 vezes e (F) sozinho aparecer 4 vezes na base de dados, eles só aparecem em conjunto com P duas vezes. Assim, para se estudar quais itens aparecem com P, apenas o prefixo do caminho de P, {F:2, C:2, A:2, M:2}, é necessário. De maneira similar, (C, B, P) aparece apenas uma vez na base de dados, e o prefixo do caminho de P é: {C:1, B:1}. Os dois caminhos que prefixam P, {F:2, C:2, A:2, M:2} e {C:1, B:1}, formam a base de subpadrões de P, que é chamada de base de padrões condicionada (isto é, a base de subpadrões sobre a condição da existência de P). A construção de uma *FP-Tree* nessa base de padrões condicionada (chamada de *FP-Tree* condicionada) leva a apenas um ramo, (C:3). Dessa maneira o único *itemset* derivado é (CP:3). Assim a busca para *itemset* frequente associados a P termina.

Para o nó M, é derivado o *itemset* (M:3) e dois caminhos: {F:4, C:3, A:3, M:2} e {F:4, C:3, A:3, B:1, M:1}. Na análise de M, P não é incluído já que qualquer *itemset* frequente envolvendo P já foi analisado previamente. De maneira similar à analise de P, a base de dados condicionada de M é: {F:2, C:2, A:2} e {F:1, C:1, A:1, B:1}. Construindo uma *FP-Tree* baseada nessa base de dados condicionada, é derivada a *FP-Tree* condicionada de M, {F:3, C:3, A:3}, que contém apenas um ramo. Após isso, pode-se executar recursivamente

um algoritmo de mineração baseado em *FP-Tree* (como o *FP-Growth*), por exemplo, "*minere*({F:3, C:3, A:3}|M)".

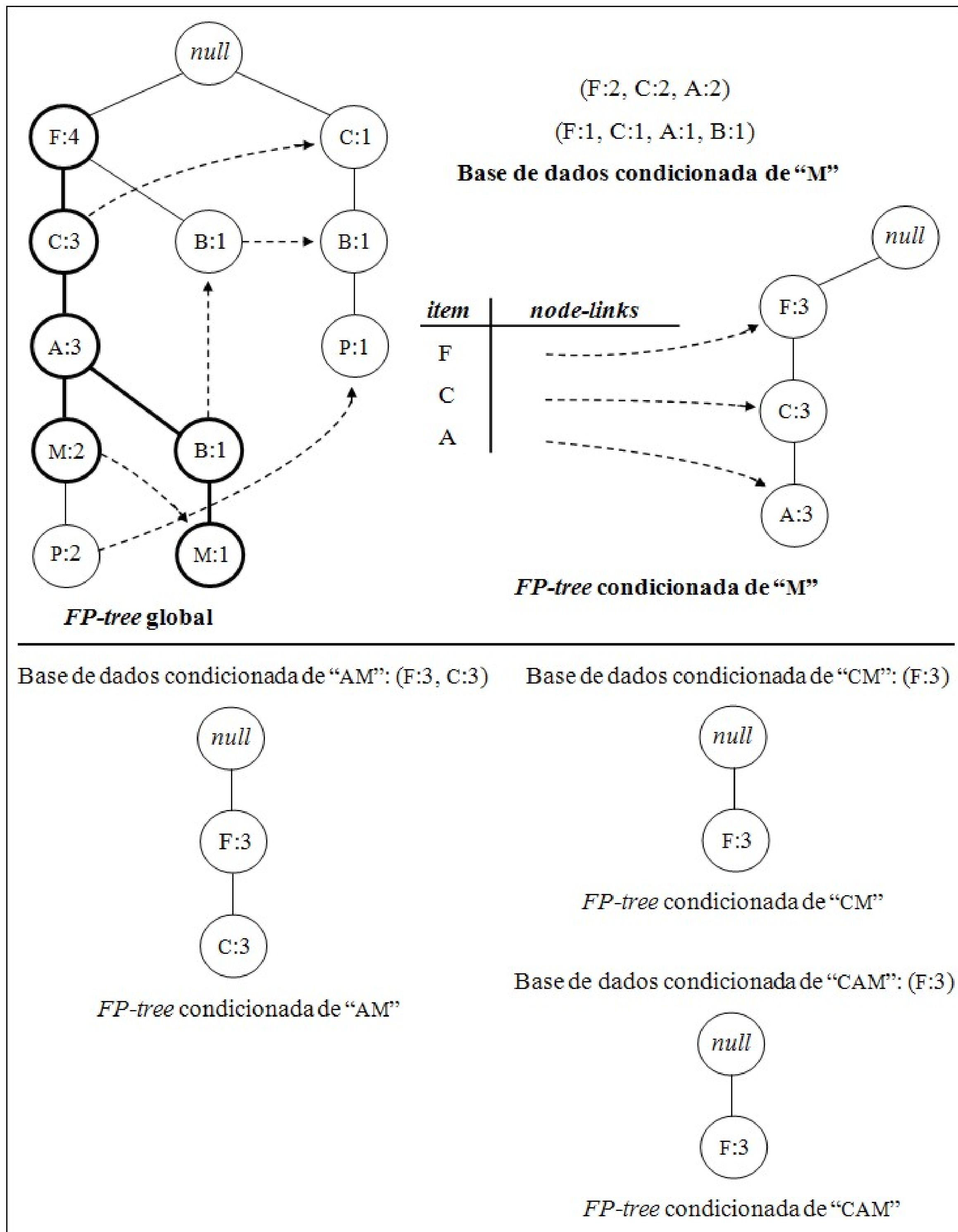


Figura 3.9: *FP-Tree* condicionada de "M".

A Figura 3.9 mostra que "*minere*({F:3, C:3, A:3}|M)"envolve minerar três itens (A), (C), (F) em sequência. O primeiro item deriva o *itemset* (AM:3) e chama "*minere*(F:3,

C:3|AM)". O segundo deriva o *itemset* frequente (CM:3) e chama "*minere*({F:3}|CM)". O terceiro deriva apenas o *itemset* frequente (FM:3). A chamada a "*minere*({F:3, C:3}|AM)" deriva (CAM:3), (FAM:3) e gera uma chamada para "*minere*({F:3}|CAM)", que deriva o *itemset* mais longo (FCAM:3). De maneira similar, a chamada "*minere*({F:3}|CM)" deriva o *itemset* (FCM:3). Dessa maneira, o conjunto de *itemsets* frequentes envolvendo M é {(M:3), (AM:3), (CM:3), (FM:3), (CAM:3), (FAM:3), (FCM:3), (FCAM:3)}. Isso indica que uma *FP-Tree* com apenas um caminho pode ser minerada através da combinação de todos os itens no caminho.

A mineração do nó B deriva (B:3) e três caminhos: (F:4, C:3, A:3, B:1), (F:4, B:1) e (C:1, B:1). Como a base de dados condicionada de B, {F:1, C:1, A:1}, {F:1} e {C:1}, não gera nenhum item frequente, a mineração para B termina. O nó A deriva um *itemset* frequente (A:3) e uma base de dados condicionada (F:3, C:3), uma *FP-Tree* com apenas um caminho. Dessa maneira pode-se gerar o conjunto de padrões frequentes através da combinação dos itens da base. Concatenando eles com (A:3), tem-se: {(FA:3), (CA:3), (FCA:3)}. O nó C deriva (C:4) e uma base de dados condicionada (F:3), gerando o padrão (FC:3). O nó F deriva apenas (F:4) e nenhuma base de dados condicionada. A Tabela 3.3 apresenta todas as bases de dados condicionadas, as *FP-Trees* condicionadas e os *itemsets* frequentes gerados.

Tabela 3.3: Base de dados condicionadas, *FP-Tree* condicionada e *Itemsets* frequentes.

Item	**Base de dados condicionada**	*FP-Tree* **condicionada**	*Itemsets* **frequentes**
P	{(F:2,C:2,A:2,M:2),(C:1,B:1)}	{(C:3)}|P	{(P:3,CP:3)}
M	{(F:2,C:2,A:2),(F:1,C:1,A:1,B:1)}	{(F:3,C:3,A:3)}|M	{(M:3),(AM:3),(CM:3), (FM:3),(CAM:3),(FAM:3), (FCM:3),(FCAM:3)}
B	{(F:1,C:1,A:1),(F:1),(C:1)}	∅	{(B:3)}
A	{(F:3,C:3)}	{(F:3),(C:3)}|A	{(A:3),(FA:3),(CA:3), (FCA:3)}
C	{(F:3)}	{(F:3)}|C	{(C:4),(FC:3)}
F	∅	∅	{(F:4)}

3.4.5 Algoritmo Eclat

O Algoritmo Eclat (*Equivalence Class Transformation*), introduzido por Zaki [33], utiliza uma estratégia que combina a busca em profundidade com intersecções entre conjuntos. Quando se utiliza a busca em profundidade, é necessário apenas manter em memória as *tidlists* dos *itemsets* correntemente em análise. Dessa maneira, dividir a base de dados como o Algoritmo Partition não é mais necessário. O Algoritmo Eclat utiliza uma otimização chamada de "intersecções rápidas". Na realização de uma intersecção entre duas *tidlists*, a *tidlist* resultante é composta pelos *itemsets* que possuem suporte maior que o mínimo especificado *min_sup*. Nas "intersecções rápidas", o processo de intersecção é interrompido quando se conhece que o resultado não atinge o suporte mínimo.

No exemplo mostrado na Figura 3.10, tem-se uma borda que separa os *itemsets* frequentes dos não frequentes, no qual acima dela encontram-se os *itemsets* frequentes. O

itemset {CD}, por exemplo, não é frequente, pois se encontra abaixo da borda. Um algoritmo de mineração de regras de associação deve determinar a borda de maneira eficiente, de modo que, apenas possam ser testados e enumerados os *itemsets* que são frequentes. A borda pode ser determinada precisamente pelos *itemsets* frequentes máximos. Lembrando que, um *itemset* frequente é máximo quando ele não é um subconjunto de nenhum outro *itemset* frequente. Na Figura 3.10, os *itemsets* frequentes máximos são apresentados dentro de uma elipse.

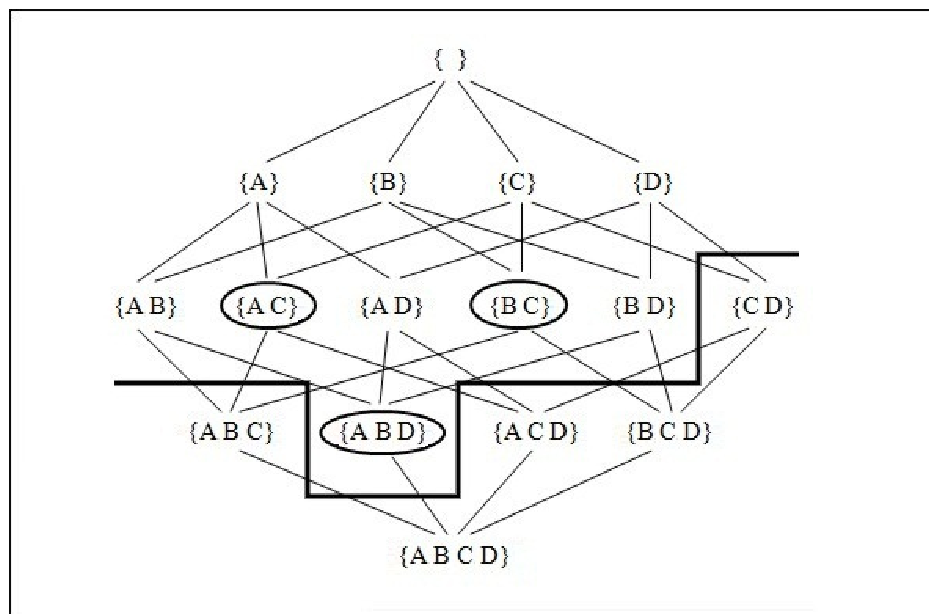


Figura 3.10: *itemsets* frequentes máximos.

Dado o conhecimento de *itemsets* frequentes máximos, seria desejável que um algoritmo obtivesse o suporte para eles e para os seus subconjuntos percorrendo apenas uma vez a base de dados. De maneira geral, não é possível determinar os *itemsets* frequentes máximos nos passos intermediários do algoritmo, mais é possível ter uma noção aproximada de quais serão estes *itemsets*. O Algoritmo Eclat utiliza uma técnica de clusterização para agrupar itens de modo a obter superconjuntos dos *itemsets* frequentes máximos (*itemsets* frequentes máximos em potencial), chamada de clusterização por classe de equivalência.

A clusterização por classe de equivalência ocorre da seguinte maneira. Seja L_2 = {AB, AC, AD, AE, BC, BD, BE, DE} o conjunto de *itemsets* frequentes de tamanho 2. Para gerar os *itemsets* candidatos de tamanho 3, deve-se combinar os *itemsets* de L_2, ou seja, C_3 = {ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE}. Assumindo então que L_{k-1} está ordenada lexicograficamente, pode-se particionar os *itemsets* de L_{k-1} em classes de equivalência baseado nos seus prefixos comuns de tamanho k-2. Os *itemsets* candidatos de tamanho k podem ser gerados a partir dos *itemsets* de uma classe, através da junção de pares de itens da classe e concatenação com o identificador da classe como prefixo. Para L_2 tem-se as seguintes classes de equivalência: [A] = {B, C, D, E}, [B] = {C, D, E} e [D] = {E}. A classe de equivalência [A] gera os *itemsets* {ABC, ABD, ABE, ACD, ACE, ADE} e [B] gera os {BCD, BCE, BDE}, no qual os *itemsets* gerados por uma classe são

independentes dos gerados por outra. Qualquer classe com apenas 1 elemento pode ser descartada, já que nenhum *itemset* pode ser gerado a partir dela, desta maneira, a classe [D] é eliminada.

Em qualquer passo intermediário do algoritmo, quando o conjunto de *itemsets* frequentes L_k com $k \geq 2$ tiver sido gerado, é possível gerar um conjunto de *itemsets* frequentes máximos em potencial através das classes de equivalência. Os *itemsets* frequentes máximos em potencial são gerados concatenando-se todos os itens de uma classe de equivalência, e tendo o identificador da classe como prefixo. A classe [A] gera o *itemset* frequente máximo em potencial ABCDE, e a classe [B] gera BCDE. Como BCDE é o subconjunto de ABCDE, ele é decartado como *itemset* frequente máximo em potencial. Quanto maior o valor de k, mais preciso o processo de clusterização. Note-se que para $k = 1$, todo o universo de itens será considerado como *itemset* máximo. No entando, para $k \geq 2$ é possível extrair um conhecimento mais preciso sobre a associação entre os itens.

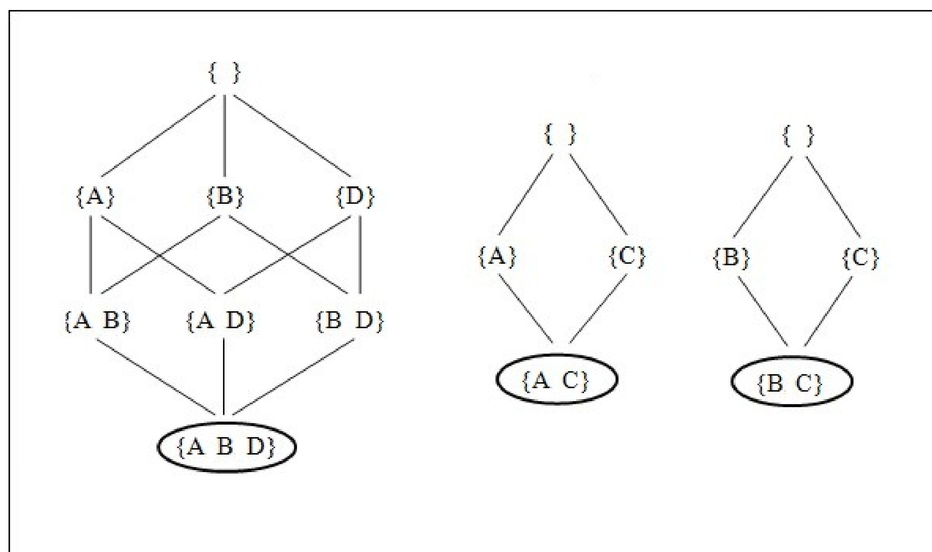


Figura 3.11: Composição dos *itemsets* frequentes máximos em potencial.

Uma vez determinado os *itemsets* frequentes máximos em potencial, devem ser encontrados os verdadeiros *itemsets* frequentes máximos. Cada *itemset* frequente máximo em potencial pode ser decomposto em *itemsets*, sendo este conjunto de *itemsets* mais o *itemset* frequente máximo em potencial um subconjunto dos *itemsets* da base de dados. Todos esses *itemsets* devem ser analisados para se determinar o verdadeiro *itemset* frequente máximo. A Figura 3.11 mostra os *itemsets* que compõe os *itemsets* frequentes máximos em potencial da Figura 3.10.

Os *itemsets* gerados são percorridos utilizando uma busca em largura como no Algoritmo Apriori. Como os *itemsets* gerados são um subconjunto dos *itemsets* da base de dados, pode-se dizer que em relação a toda a base de dados, o Algoritmo Eclat faz busca em profundidade. A seguir, o Algoritmo Eclat mostra como os *itemsets* gerados são percorridos. No passo 4 do algoritmo, é utilizada intersecções de *tidlists* e a otimização "intersecções rápidas".

1. **Entrada**: O conjunto F_k de *itemsets* frequentes de tamanho k e o valor de suporte mínimo *min_sup*.
2. **Saída**: O conjunto L de *itemsets* frequentes de tamanho l, em que $l > K$.
3. **Função** *Eclat*(F_k)
4. **para** todo *itemset* $c_i \in F_k$ **faça**
5. $F_{k+1} = \emptyset$;
6.   **para** todo *itemset* $c_j \in F_k$, em que $j > i$ **faça**
7. $s = c_i \cap c_j$;
8.   **se** (s.suporte $\geq$ *min_sup*) **então**
9. $F_{k+1} = F_{k+1} \cup \{s\}$;
10.   **fim se**
11.   **fim para**
12.   **se** ($F_{k+1} \neq \emptyset$) **então**
13.   *Eclat* (F_{k+1});
14.   **fim se**
15. **fim para**

Capítulo 4

Algoritmos Paralelos para Extração de Regras de Associação

4.1 Conceitos Gerais

A mineração de regras de associação é uma tarefa extremamente intensiva computacionalmente, em que a determinação dos *itemsets* frequentes é a fase que exige maior processamento. Dependendo das dimensões da base de dados, torna-se crucial o emprego da computação paralela. Para isso, existem alguns algoritmos paralelos para tornar a descoberta dos *itemsets* frequentes mais eficiente e em menor tempo de processamento.

Este capítulo está organizado como se segue. Na Seção 4.2 é apresentado o Algoritmo Apriori no modelo paralelo. Na Seção 4.3 é mostrado o Algoritmo Paralelo Eclat. Por fim, na Seção 4.4 é apresentado o Algoritmo Paralelo *FP-Growth*.

4.2 Algoritmo Apriori

O Algoritmo Apriori no modelo paralelo, também conhecido como CD (*Count Distribution*), foi desenvolvido por Agrawal e Shafer [3]. No algoritmo Apriori, a base de dados de entrada é particionada de forma que cada processador recebe m/p transações, em que m é o número de transações presentes na base de dados e p o número de processadores em operação.

Inicialmente, cada processador efetua uma varredura em sua base de dados local para determinar o valor de frequencia dos 1-*itemsets* presentes. Para obter o valor de frequencia global dos 1-*itemsets*, todos os processadores realizam uma operação de redução global com as frequencias locais. Com isso, determina-se o conjunto de todos os 1-*itemsets* frequentes.

Em seguida, cada processador cria uma árvore *hash* com o conjunto formado pelos 2-*itemsets* candidatos. Com uma varredura na base de dados local, cada processador

determina a frequencia local dos 2-*itemsets* candidatos e, efetuando uma redução global, determina-se o conjunto de todos os 2-*itemsets* frequentes. Ou seja, para cada conjunto de k-*itemsets* frequentes cria-se a árvore *hash* com o conjunto dos $(k+1)$-*itemsets* candidatos e, com uma varredura na base de dados local e uma operação de redução global, determina-se o conjunto dos $(k+1)$-*itemsets* frequentes. Assim, o processo é repetido até que todos os *itemsets* sejam gerados. Devido a essa estratégia, cada processador determina o conjunto de todos os *itemsets* frequentes, o que não torna a saída do algoritmo distribuída entre os processadores.

4.3 Algoritmo Eclat

O Algoritmo Eclat (*Equivalence Class Transformation*) no modelo paralelo, proposto por Zaki et al [33], é baseado em uma técnica que decompõe todo o espaço de busca original em partições menores, de forma que cada partição possa ser processada independentemente. Essa decomposição é feita dividindo o espaço de busca utilizando a técnica de clusterização por classes de equivalência, como utilizado no modelo sequencial.

No algoritmo, o espaço de busca refere-se a base de dados a ser processada, na qual é distribuída entre os processadores de forma que cada processador possa encontrar os *itemsets* frequentes independentemente, utilizando intersecções de *tidlists*. Para isso, uma projeção vertical da base de dados deve ser utilizada, na qual é construída com apenas uma varredura na base de dados.

O algoritmo Eclat, no modelo paralelo, consiste em duas fases, denominadas **fase de inicialização** e **fase de processamento**. Na fase de inicialização, a técnica de clusterização por classes de equivalencia é aplicada distribuíndo as classes determinadas entre os processadores. E na fase de processamento, a partir das classes de equivalência, cada processador determina o seu conjunto *itemsets* frequentes independentemente.

4.3.1 Fase de Inicialização

A fase de inicialização consiste de três passos. Primeiro, a base de dados é acessada, todos os itens frequentes são encontrados e as suas *tidlists* são construídas. Segundo, os 2-*itemsets* são computados realizando intersecções entre as *tidlists* dos itens frequentes. Terceiro, os 2-*itemsets* são agrupados em classes de distintas aplicando-se a técnica de clusterização por classes de equivalência, e assim o conjunto de classes independentes é criado.

Em seguida, as classes são distribuídas entre os processadores, de forma que, um grau de balanceamento de carga seja atingido por um valor de peso associado a cada classe de equivalência, baseado no número de elementos presentes na classe. Feito isso, as classes de equivalência são ordenadas em função dos pesos, e então, cada uma é atribuída ao processador com menor carga de trabalho (menor soma de pesos). Caso dois processadores atinjam uma mesma estimativa de carga de trabalho, o empate é desfeito selecionando o processador com o maior identificador. O processo referente a fase de inicialização é

apresentado na Figura 4.1.

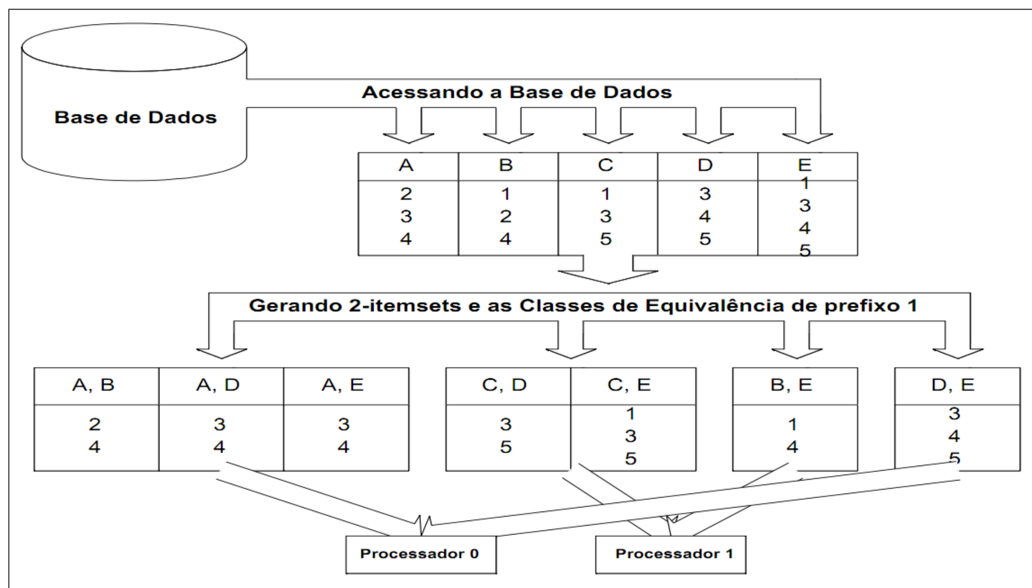


Figura 4.1: Fase de Inicialização do Algoritmo Eclat

4.3.2 Fase de Processamento

Após a fase de inicialização, as classes de equivalência estão disponíveis localmente em cada processador. Assim, cada um deles pode gerar independentemente todos os *itemsets* frequentes provenientes das classes que lhe foram lhe atribuídas. Cada processador mantém localmente todas as *tidlists* enquanto os *itemsets* são produzidos e a tarefa mineração de *itemsets*, como no modelo sequencial, é realizada.

À medida que cada processador vai terminando a tarefa de gerar seus *itemsets* frequentes, não é viável que esse processador assuma parte da tarefa ainda não concluída de outro processador. Em sistemas de memória distribuída, a movimentação de dados tem um custo muito alto, devido à alta taxa de comunicação. Por isso, é extremamente essencial que a distribuição da carga de trabalho entre os processadores seja equilibrada. Com essa estratégia de mineração, o conjunto global de *itemsets* frequentes é encontrado de forma distribuída entre os processadores.

4.4 Algoritmo *FP-Growth*

O Algoritmo *FP-Growth*, no modelo paralelo [21], utiliza a mesma estratégia adotada no modelo sequencial. Para minerar um conjunto de *itemsets* frequentes o algoritmo faz uso da estrutura de dados *FP-Tree*, em que transações repetidas na base de dados são compactadas na estrutura. Inicialmente, em cada transação da base de dados, os itens frequentes são ordenados em ordem decrescente sobre o seu valor de suporte. Feito isso,

cada transação é mapeada na *FP-Tree*, de forma que, os itens que formam um prefixo em comum com itens de outra transação já mapeada na estrutura possam ser agrupados.

Porém, devido à estrutura complexa e dinâmica da *FP-Tree*, frequentemente não é prático a construção de uma única estrutura para uma base de dados inteira. No entanto, múltiplas *FP-Trees* podem ser construídas facilmente em paralelo utilizando diferentes partições da base de dados, como também, bases de dados condicionadas podem ser extraídas e transformadas em *FP-Trees* condicionadas para todos os itens frequentes.

Em geral, o Algoritmo *FP-Growth* pode ser paralelizado no modelo de memória distribuída através dos seguintes passos:

1. A base de dados é dividida uniformemente em p partições horizontais, em que p é o número de processadores e cada partição é atribuída a um processador.

2. Cada processador varre a sua base de dados para determinar o valor de suporte local dos itens.

3. Uma operação de redução global é feita entre os processadores para determinar o suporte global dos itens. Em seguinda, o conjunto de itens frequentes é determinado e ordenado em ordem decrescente em função do suporte global.

4. Cada processador constrói localmente a tabela *node-links* e a *FP-Tree*. A tabela *node-link* é construída contendo os itens frequentes com os suportes globais. Já a *FP-Tree* é construída contendo os itens frequentes com os suportes locais.

5. A partir da *FP-Tree* local, cada processador gera as bases de dados condicionadas para os itens frequentes.

6. A partir de cada item frequente é construída uma *FP-Tree* condicionada global. Na realização dessa tarefa, cada item é atribuído a um dos processadores e todas as bases de dados condicionadas locais ligadas a este item são enviadas para o processador designado.

7. Em função de um item frequente, todas as bases de dados condicionadas recebidas pelo processador designado são concatenadas com a base condicionada local, e em seguida, a *FP-Tree* condicionada global é construída.

8. Cada processador percorre recursivamente as *FP-Trees* condicionadas para minerar os *itemsets* frequentes.

O Algoritmo Paralelo *FP-Growth* é dividido em duas fases de processamento. A primeira fase consiste nos passos de 1 a 4 apresentados anteriormente. Nessa fase, uma tabela *node-link* e uma *FP-Tree* são construídas em cada processador. Na construção da *FP-Tree*, os itens frequentes de cada transação da base de dados local são ordenados em ordem decrescente, em função do seu suporte, e adicionados na *FP-Tree*.

Para a inserção dos itens na *FP-Tree*, inicialmente é verificado se o primeiro item de cada transação ordenada existe como um dos filhos da raiz. Se existir, então o contador

para este nó deve ser incrementado. Caso contrário, um novo nó é então adicionado como filho para a raiz, no qual o nó deve conter o item e um contador com valor inicial igual a 1. Para os demais itens da transação ordenada, o mesmo processo é repetido considerando o nó corrente na *FP-Tree* como uma raiz temporária. Todos os nós na *FP-Tree* que possuem itens iguais são ligados formando uma lista encadeada, em que o nó cabeça é armazenado na tabela *node-links*. Um exemplo de construção paralela de *FP-Trees* é apresentado na Figura 4.2.

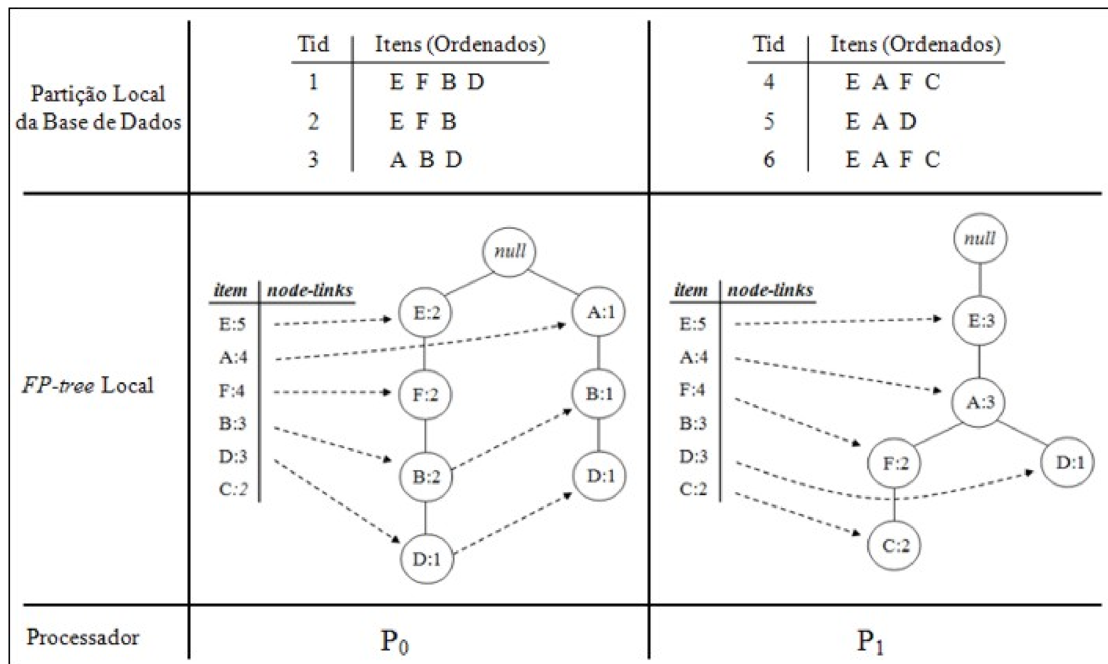


Figura 4.2: Exemplo da construção de *FP-Trees* locais a partir de uma base de dados particionada entre dois processadores.

A segunda fase de processamento, que consiste nos passos de 5 a 8, está relacionada à mineração dos *itemsets* frequentes da *FP-Tree* construída. O processo de mineração é realizado através de uma varredura *bottom-up* na *FP-Tree*, no qual é gerada a base de dados condicionada para cada item na tabela *node-links*. Uma base de dados condicionada é composta pelas listas de itens que formam um caminho na *FP-Tree*, partindo da raiz até o nó que precede o item em análise na tabela *node-links*.

Para efetuar a mineração dos *itemsets* frequentes, os itens armazenados na tabela *node-links* são distribuídos entre os processadores. A estratégia mais simplificada para divisão das tarefas, de modo que a carga de trabalho seja equilibrada, é realizar a atribuição em sequência de cada item presente na tabela *node-links* para os processadores em atividade. Por exemplo, seja n o número de itens presentes em uma tabela *node-links* e p o número de processadores. Cada processador p_i, em que $0 \leq i < p$, recebe o item presente em *node-links*$[j \underline{mod}\ p]$, em que $0 \leq j < n$. Essa forma de distribuição dos itens procura equilibrar a carga de trabalho entre os processadores, porque quanto menor a frequencia do item, maior o tamanho das suas estruturas de dados condicionadas.

Após a distribuíção dos itens, cada processador recebe dos demais processadores as bases de dados condicionadas referentes ao item lhe atribuído. Para cada item, todas as bases de dados condicionadas recebidas são agrupadas com a base de dados condicionada local e ordenadas em ordem decrescente em função do valor de suporte. Com isso, uma *FP-Tree* condicionada é criada seguindo os mesmos passos utilizados na construção da *FP-Tree* sobre um conjunto de transações.

A mineração de *itemsets* frequentes é realizada a partir de cada *FP-Tree* condicionada associada a um item frequente. Dessa forma, cada processador aplica os mesmos passos do algoritmo no modelo sequencial sobre a *FP-Tree* condicionada para a tarefa de mineração. A Figura 4.3 apresenta o processo de mineração de *itemsets* a partir do exemplo mostrado na Figura 4.2.

Item	E:5	A:4	F:4	B:3	D:3	C:2
Base de Dados Condicionada		(E:3)	(E:2) (E:2, A:2)	(E:2, F:2) (A:1)	(E:1, F:1, B:1) (A:1, B:1) (E:1, A:1)	(E:2, A:2, F:2)
FP-tree Condicionada		*item* | *node-links*: E:3 → null, E:3	*item* | *node-links*: E:4, A:2 → null, E:4, A:2	*item* | *node-links*: E:2, F:2 → null, E:2, F:2	*item* | *node-links*: A:2, B:2, E:2 → null, B:1, A:2, E:1, B:1, E:1	*item* | *node-links*: A:2, E:2, F:2 → null, A:2, E:2, F:2
Itemsets Frequentes Minerados	E:5	A:4 EA:3	F:4 EF:4 AF:2 EAF:2	B:3 EB:2 FB:2 EFB:2	D:3 ED:2 AD:2 BD:2	C:2 AC:2 EC:2 FC:2 AFC:2 EFC:2 AEC:2 AEFC:2
Processadores	P_0				P_1	

Figura 4.3: Mineração de *itemsets* frequentes para o exemplo apresentado na Figura 4.1.

Capítulo 5

Bases de Dados Educacionais

5.1 Conceitos Gerais

Através da necessidade de avaliar o setor educacional no país, o INEP (Instituto Nacional de Estudos e Pesquisas Educacionais), autarquia vinculada ao Ministério da Educação, estruturou os Sistemas Nacionais de Avaliação e de Informação, criando diversos censos, avaliações e pesquisas, com o objetivo de avaliar e acompanhar a evolução do ensino no país em todos os âmbitos: ensino fundamental, médio e superior. Assim, estes sistemas tornaram-se a ferramenta básica para o planejamento, monitoramento e acompanhamento das políticas públicas, subsidiando a tomada de decisões.

Os dados e as informações educacionais extraídas com a aplicação dos sistemas de avaliação são utilizados como instrumento principal no processo de avaliação da educação no país. Porém, este procedimento resulta em grandes volumes de dados que são acumulados ao longo dos anos e geralmente são armazenados em diferentes bases de dados de maneira não uniformizada e de difícil reuso e integração com as ferramentas dos sistemas de avaliação, o que dificulta substancialmente na sua reutilização para a tomada de decisão na gestão pública [28].

Diante disso, houve a proposta no presente trabalho de aplicar a mineração de regras de associação nas bases de dados educacionais visando a busca de informações úteis e escondidas, o que pode auxiliar no desenvolvimento de políticas voltadas para área da educação. A identificação de padrões nos dados educacionais pode revelar situações, por exemplo, em que o desempenho de alunos está ligado a fatores como: questões socioeconômicas, nível de escolaridade dos pais, estrutura física do estabelecimento de ensino, material didático ou formação dos professores.

Com a finalidade de promover pesquisas acadêmicas com os dados armazenados, o INEP, em parceria com a CAPES, criou o programa Observatório da Educação. O primeiro edital deste programa foi lançado em 2006, quando foi submetido e aprovado o projeto Web-PIDE. Para subsidiar os estudos e as pesquisas referente a educação no país, o INEP disponibilizou os dados de avaliações e censos para o projeto Web-PIDE.

Este capítulo está organizado como se segue. Na Seção 5.2 são apresentados os Sistemas Nacionais de Avaliação e Informações Educacionais. Na Seção 5.3 são apresentados o projeto Web-PIDE e as bases dados disponibilizadas pelo INEP. Por fim, na Seção 5.4 é mostrado o estudo de caso consistindo na mineração de regras de associação com os dados educacionais do INEP.

5.2 Sistemas Nacionais de Avaliação e Informações Educacionais

Atualmente em vigência, os Sistemas Nacionais de Avaliação e Informação Educacionais estão compostos pelos seguintes instrumentos de avaliação:

5.2.1 Censo Escolar

O Censo Escolar realiza o levantamento de informações estatístico-educacionais relativas à Educação Básica, em seus diferentes níveis (educação infantil, ensino fundamental e ensino médio) e modalidades (ensino regular, educação especial e educação de jovens e adultos).

O levantamento é feito anualmente junto a todos os estabelecimentos de ensino, das redes pública e particular, através do preenchimento de questionário padronizado. Por intermédio do Censo Escolar, o INEP atualiza anualmente o Cadastro Nacional de Escolas e as informações referentes à matrícula, ao movimento e ao rendimento dos alunos, incluindo dados sobre sexo, turnos, turmas, séries e períodos, condições físicas dos prédios escolares e equipamentos existentes. Além disso, são coletadas informações sobre o pessoal técnico e administrativo e pessoal docente, por nível de atuação e grau de formação.

5.2.2 ENCCEJA - (Exame Nacional para Certificação de Competências de Jovens e Adultos)

O ENCCEJA é uma avaliação que mede as competências e habilidades de jovens e adultos, residentes no Brasil e no exterior, que estão concluindo o Ensino Fundamental e Médio. A adesão a esta avaliação é opcional e realizada pelas secretarias de Educação dos Estados, Distrito Federal e Municípios.

5.2.3 SAEB - (Sistema Nacional de Avaliação da Educação Básica)

O SAEB foi a primeira iniciativa brasileira, em âmbito nacional, para se conhecer o sistema educacional em profundidade. É composto por dois processos: a Avaliação Nacional da Educação Básica (Aneb) e a Avaliação Nacional do Rendimento Escolar (Anresc).

- **Aneb**: teve sua primeira edição em 1991 e é realizada por amostragem das Redes de Ensino em cada unidade da Federação e tem foco nas gestões dos sistemas educacionais. Por manter as mesmas características, a Aneb recebe o nome do Saeb em suas divulgações;

- **Anresc**: criada em 2005 é mais extensa e detalhada que a Aneb e tem foco em cada unidade escolar. Por seu caráter universal, é divulgada com o nome de Prova Brasil. As duas avaliações acontecem simultaneamente e são aplicadas de dois em dois anos.

5.2.4 Provinha Brasil

A Provinha Brasil é uma iniciativa inaugurada pelo Ministério da Educação (MEC) no primeiro semestre de 2008, com o objetivo de oferecer aos professores, diretores, coordenadores e gestores das redes públicas de ensino um instrumento de diagnóstico do nível de alfabetização das crianças com idade entre seis e oito anos de idade. O INEP/MEC disponibiliza, anualmente, duas versões da Provinha Brasil. A primeira no 1º Bimestre do ano letivo e, a segunda, no 4º semestre do ano letivo, caracterizando, assim, o ciclo da prova.

5.2.5 ENEM - (Exame Nacional do Ensino Médio)

O ENEM foi criado em 1998 para avaliar anualmente os alunos que estão concluindo ou já concluíram o ensino médio. A proposta de sua criação era apenas em ter um mecanismo que avaliasse tanto o aluno quanto a qualidade de ensino da sua escola. Porém, em 2009, houve um processo de reformulação em que passou a permitir a sua utilização como forma de seleção unificada nos processos seletivos das universidades públicas.

Com a reformulação, a proposta do ENEM passou a ter, também, objetivos como: democratizar as oportunidades de acesso às vagas federais de ensino superior, possibilitar a mobilidade acadêmica e induzir a reestruturação dos currículos do ensino médio. Atualmente, o ENEM é exigido como pré-requisito para a participação dos programas federais ProUni (Programa Universidade para Todos) e FIES (Financiamento Estudantil).

5.2.6 SINAES - (Sistema Nacional de Avaliação da Educação Superior)

Criado em 2004, o SINAES é formado por três componentes principais: a Avaliação das Instituições de Ensino Superior, a Avaliação dos Cursos de Graduação e o Exame Nacional de Desempenho dos Estudantes. Este instrumento avalia todos os aspectos em torno de três eixos: o ensino, a pesquisa, a extensão, a responsabilidade social, o desempenho dos alunos, a gestão da instituição, o corpo docente, as instalações entre outros.

Avaliação das Instituições de Ensino Superior

Este sistema é dividido em duas modalidades:

- Auto-avaliação - coordenada pela Comissão Própria de Avaliação (CPA) de cada instituição e orientada pelas diretrizes e por um roteiro da auto-avaliação institucional.
- Avaliação externa - realizada por comissões designadas pelo INEP, que tem como referência os padrões de qualidade para a educação superior expressos nos instrumentos de avaliação e os relatórios das auto-avaliações.

Avaliação dos Cursos de Graduação

O INEP/MEC prevê que os cursos sejam avaliados periodicamente. Assim, os cursos de educação superior passam por três tipos de avaliação: para autorização, para reconhecimento e para renovação de reconhecimento. As avaliações dos cursos de graduação produzem indicadores que subsidiam tanto o processo de regulamentação, exercido pelo MEC, como garante transparência dos dados sobre qualidade da educação superior.

ENADE - (Exame Nacional de Desempenho de Estudantes)

O ENADE, aplicado pela primeira vez em 2004, o exame tem o objetivo de aferir o rendimento dos alunos dos cursos de graduação em relação aos conteúdos programáticos, suas habilidades e competências. O exame é realizado por amostragem e a participação ou, quando for o caso, sua dispensa pelo MEC, consta no histórico escolar do estudante. O INEP/MEC constitui a amostra dos participantes a partir da inscrição, pela própria instituição de ensino superior, dos alunos habilitados a fazer a prova.

5.3 Projeto Web-PIDE

Durante os últimos anos, vários censos e pesquisas nacionais foram realizados resultando na produção de múltiplas bases de dados. Através da integração desses dados é possível obter uma análise abrangente da situação educacional do país e prover subsídios nos debates sobre propostas de melhorias para educação brasileira. No entanto, essas informações encontram-se distribuídas em diferentes bases de dados e de maneira não uniformizada, fato que dificulta seu reuso e possíveis integrações com sistemas de avaliação institucional das universidades nacionais.

O fato das bases de dados armazenarem um grande volume de informações, também, dificulta sua análise e impossibilita uma tomada de decisão eficiente baseada nos dados das avaliações. Neste contexto, o projeto Web-PIDE tem o objetivo de especificar e implementar uma plataforma para compartilhar e integrar dados educacionais, visando facilitar

a consulta aos dados das avaliações e geração de hipóteses relativas aos dados armazenados. A Figura 5.1 apresenta a arquitetura da plataforma Web-PIDE.

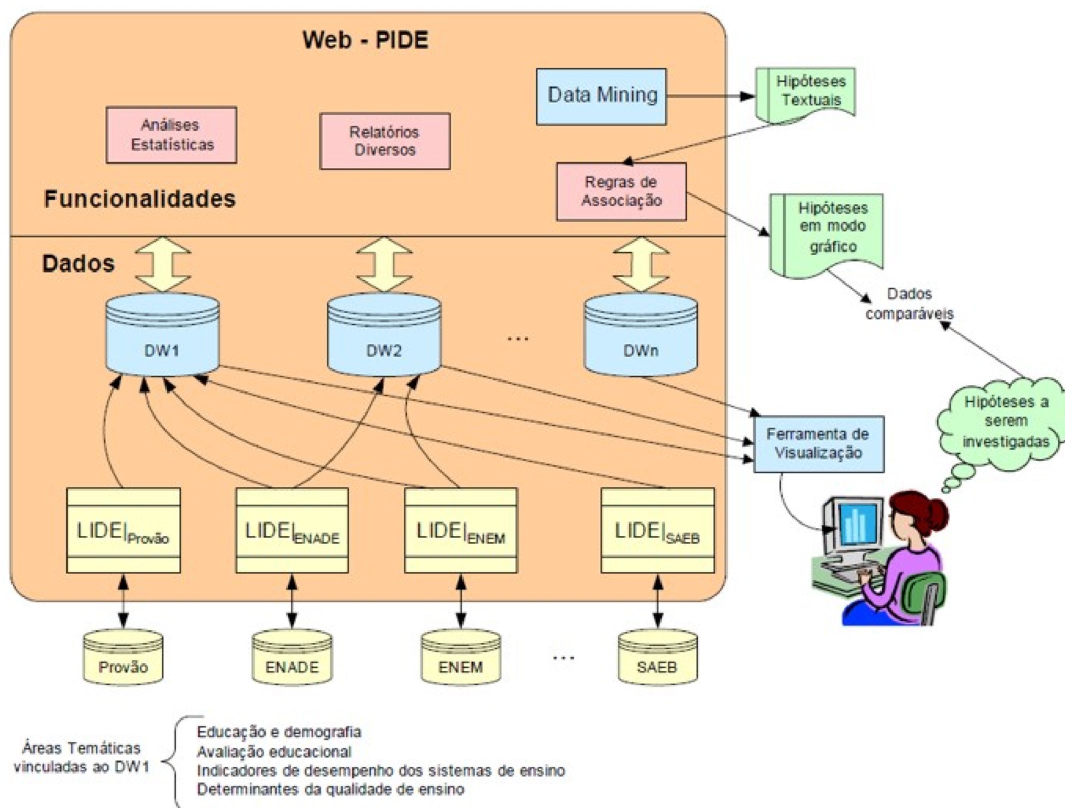


Figura 5.1: Arquitetura da plataforma Web-PIDE [28]).

Para integrar os dados educacionais, o projeto propôs uma linguagem de marcação intitulada LIDE (Linguagem para Integração de Dados Educacionais) para cada uma das bases de dados do INEP. As LIDEs são definidas com a linguagem de marcação XML (Extensible Markup Language), que torna os documentos legíveis para as pessoas e manipuláveis pelos computadores. A LIDE deve possuir um formato padrão que permita com que esses dados educacionais possam ser interpretados e armazenados em uma base de dados que dê suporte a um ambiente no qual várias funcionalidades poderão ser implementadas. Para completar a camada responsável pelos dados na arquitetura Web-PIDE, o projeto utiliza a tecnologia de Data Warehouse (DW) e de Data Marts (DM) para sistematizar e armazenar os dados históricos com o objetivo de facilitar a tomada de decisão pelos gestores. Essa tecnologia proporciona ao Ambiente de Apoio à Decisão (AAD), que no caso é a plataforma Web-PIDE, uma sólida e concisa integração dos dados para a realização de análises gerenciais, preocupando-se em integrar e consolidar as informações de fontes heterogêneas e fontes externas, sumarizando, filtrando e limpando esses dados, preparando-os para análise e suporte à decisão. Com o uso das tecnologias propostas para construir a camada de dados, a plataforma Web-PIDE deve apresentar facilidades para:

- Extração de dados educacionais de fontes heterogêneas (existentes ou externas);
- Transformação e integração dos dados antes de seu armazenamento na plataforma Web-PIDE;
- Visualização dos dados em diferentes níveis. Os dados do DW podem ou não ser extraídos para um nível mais específico, que são os DMs e, a partir desses, para um banco de dados individual; e
- Utilização de ferramentas voltadas para acesso com diferentes níveis de apresentação.

Dada a facilidade que pode ser alcançada com a construção da plataforma Web-PIDE, acredita-se que ela será um elemento facilitador para a comunidade interessada em trabalhar com esses dados, bem como servir de incentivo para a condução e suporte de trabalhos acadêmicos, o que, atualmente, não possui um número representativo.

5.3.1 Formato dos Dados Disponibilizados pelo INEP

As bases de dados são disponibilizadas pelo INEP em CD-ROM (http://portal.inep.gov.br/basica-levantamentos-acessar), no qual os dados estão organizados em:

- documentação dos microdados, presente em arquivos nomeados "leia-me", do tipo PDF (*Portable Document Format*).
- arquivos de entrada SAS (*Statistical Analysis Software*), que por sua vez definem as variáveis envolvidas nos microdados e são úteis por serem fáceis de manipular, possibilitando a leitura por outros software;
- arquivos no formato ASCII (*American Standard Code for Information Interchange*);

As Figuras 5.2, 5.3 e 5.4 mostram o formato de cada um dos arquivos disponibilizados pelo INEP.

.

DADOS ESTATÍSTICOS DO SISTEMA NACIONAL DE AVALIAÇÃO DA EDUCAÇÃO BÁSICA - Dicionário de Dados do arquivo MATEMATICA_04SERIE.TXT

Nome da Variável	Descrição	Variaveis Categoricas		Inicio	Tamanho	Tipo
		Categoria	Descrição			
VARIAVEIS DE CONTROLE (Bloco 1)						
MASCARA	Número gerado como código de acesso aos Estabelecimentos de Ensino Básico. Este código é o mesmo para todos os Microdados contendo informações sobre os Estabelecimentos no ano de referência. Por exemplo, usando a combinação ANO_MASCARA + MASCARA é possível obter os dados estatísticos da instituição a partir dos Microdados do Censo Escola. Este código, entretanto, não permite o acesso aos dados cadastrais dos Estabelecimentos (nome, endereço, etc). A mesma MASCARA para anos diferentes não identifica o mesmo Estabelecimento de Ensino.			1	8	Numérica
ANO_MASCARA	Ano da Máscara			9	8	Numérica
ANO	Ano de aplicação			17	4	Alfanumérica
SERIE	Série	04	4ª Série	21	2	Alfanumérica
DISC	Disciplina	M	Matemática	23	1	Alfanumérica
TURMA	Número da turma			24	2	Alfanumérica
ALUNO	Código do aluno			26	3	Alfanumérica
ESTRATO	Estrato da Amostra			29	12	Numérica
UPA	Unidade Primária de Amostragem			41	12	Numérica
DEP_ADM	Dependência Administrativa(Estadual/Municipal/Particular)	1	Estadual	53	8	Numérica
		2	Municipal			
		3	Particular			
		4	Federal			
LOCAL	Localização(Urbano/Rural)	1	Urbana	61	8	Numérica
		2	Rural			
REDE	Rede de Ensino(Publica/Particular)	1	Pública	69	8	Numérica
		2	Particular			
UF	Unidade da Federação	11	Rondônia	77	8	Numérica
		12	Acre			
		13	Amazonas			
		14	Roraima			
		15	Pará			
		16	Amapá			
		17	Tocantins			
		21	Maranhão			
		22	Piauí			
		23	Ceará			
		24	Rio Grande do Norte			
		25	Paraíba			
		26	Pernambuco			
		27	Alagoas			
		28	Sergipe			
		29	Bahia			
		31	Minas Gerais			
		32	Espírito Santo			
		33	Rio de Janeiro			
		35	São Paulo			
		41	Paraná			
		42	Santa Catarina			
		43	Rio Grande do Sul			
		50	Mato Grosso do Sul			
		51	Mato Grosso			
		52	Goiás			
		53	Distrito Federal			

Figura 5.2: Parte de um dos arquivos “Leia-me.pdf”do SAEB 2003 [1].

```
DATA MATEMATICA_04SERIE;

     INFILE 'D:\DADOS\ALUNOS\MATEMATICA_04SERIE.TXT' LRECL=894 MISSOVER;
     INPUT

@1    MASCARA     8.        /* MASCARA    */
@9    ANO_MASCARA 8.        /* ANO_MASCARA */
@17   ANO         $char4.  /* Ano de aplicação  */
@21   SERIE       $char2.  /* Série */
@23   DISC        $char1.  /* Disciplina  */
@24   TURMA       $char2.  /* Número da turma   */
@26   ALUNO       $char3.  /* Código do aluno   */
@29   ESTRATO     12.       /* Código do aluno   */
@41   UPA         12.       /* Unidade Primária de Amostragem     */
@53   DEP_ADM     8.        /* Dependência Administrativa(Estadual/Municipal/Particular)  */
@61   LOCAL       8.        /* Localização(Urbano/Rural)    */
@69   REDE        8.        /* Rede de Ensino(Publica/Particular)  */
@77   UF          8.        /* Unidade da Federação    */
@85   UFSUD       8.        /* Unidade da Federação para SUDAAN    */
@93   REGIAO      8.       /*  Região      */
@101  TAM_MUNIC   8.       /*  Tamanho do Município    */
@109  REG_METROP  8.       /*  Localizado em Região Metropolitana?*/
@117  TAM_CID     8.       /*  Tamanho/perfil da Cidade     */
@125  TUR_BE      8.       /*  Turma tem aluno Bolsa Escola?*/
@133  ALU_BE      8.       /*  Aluno tem Bolsa Escola?*/
@141  PESO_AC     8.4      /*  Peso calibrado(usado para expansão)*/
@149  CADERNO     8.       /*  Caderno de Provas */
@157  BLOCO1      8.       /*  Bloco 1 do caderno      */
@165  BLOCO2      8.       /*  Bloco 2 do caderno      */
@173  BLOCO3      8.       /*  Bloco 3 do caderno      */
@181  RESP_BL1    $char13.    /*    Respostas do aluno no Bloco 1 do Caderno */
@194  RESP_BL2    $char13.    /*    Respostas do aluno no Bloco 2 do Caderno */
@207  RESP_BL3    $char13.    /*    Respostas do aluno no Bloco 3 do Caderno */
@220  GAB_BL1     $char13.    /*    Gabarito do Bloco 1 do Caderno     */
@233  GAB_BL2     $char13.    /*    Gabarito do Bloco 2 do Caderno     */
@246  GAB_BL3     $char13.    /*    Gabarito do Bloco 3 do Caderno     */
@259  PROFIC      12.5  /*     Proficiência      */
@271  ESTAGIO     15.   /*     Estágios de desempenho  */
@286  A041_001    8.    /*     Sexo  */
@294  A041_002    8.    /*     Como você se considera? */
@302  A041_003    8.    /*     Qual a sua idade? */
@310  A041_004    8.    /*     Você ainda vai fazer aniversário ate o final deste ano?    */
@318  A041_005    8.    /*     Qual e o mês do seu aniversário?    */
@326  A041_006    8.    /*     Na sua casa tem televisão em cores?*/
@334  A041_007    8.    /*     Na sua casa tem radio?  */
@342  A041_008    8.    /*     Na sua casa tem videocassete?*/
@350  A041_009    8.    /*     Dentro de sua casa tem banheiro?    */
@358  A041_010    8.    /*     Na sua casa tem quartos para dormir?      */
@366  A041_011    8.    /*     Na sua casa tem geladeira?    */
@374  A041_012    8.    /*     Na sua casa tem freezer junto a geladeira?      */
@382  A041_013    8.    /*     Na sua casa tem freezer separado da geladeira? */
@390  A041_014    8.    /*     Na sua casa tem maquina de lavar roupa?  */
@398  A041_015    8.    /*     Na sua casa tem aspirador de pó?    */
@406  A041_016    8.    /*     Na sua casa tem automóvel/ carro?   */
@414  A041_017    8.    /*     Na sua casa tem computador com internet? */
@422  A041_018    8.    /*     Na sua casa tem computador sem internet? */
@430  A041_019    8.    /*     Além dos livros escolares, quantos livros há em sua casa?  */
```

Figura 5.3: Parte de um dos arquivos SAS do SAEB 2003 [1].

```
22922841    2002200304M 2  2    15611   22922841   3   1   2   15
22922841    2002200304M 2  4    15611   22922841   3   1   2   15
22922841    2002200304M 2  6    15611   22922841   3   1   2   15
22922841    2002200304M 2  8    15611   22922841   3   1   2   15
22922841    2002200304M 2 10    15611   22922841   3   1   2   15
22922841    2002200304M 2 12    15611   22922841   3   1   2   15
22922841    2002200304M 2 14    15611   22922841   3   1   2   15
22922841    2002200304M 2 16    15611   22922841   3   1   2   15
22922841    2002200304M 2 18    15611   22922841   3   1   2   15
22922841    2002200304M 2 20    15611   22922841   3   1   2   15
22926872    2002200304M 1  2    15611   22926872   3   1   2   15
22926872    2002200304M 1  4    15611   22926872   3   1   2   15
22926872    2002200304M 1  6    15611   22926872   3   1   2   15
22926872    2002200304M 1  8    15611   22926872   3   1   2   15
22926872    2002200304M 1 10    15611   22926872   3   1   2   15
22926872    2002200304M 1 12    15611   22926872   3   1   2   15
22926872    2002200304M 1 14    15611   22926872   3   1   2   15
22926872    2002200304M 1 16    15611   22926872   3   1   2   15
22926872    2002200304M 1 18    15611   22926872   3   1   2   15
22926872    2002200304M 1 20    15611   22926872   3   1   2   15
22926872    2002200304M 1 22    15611   22926872   3   1   2   15
22926872    2002200304M 1 24    15611   22926872   3   1   2   15
22926872    2002200304M 1 26    15611   22926872   3   1   2   15
22926872    2002200304M 1 28    15611   22926872   3   1   2   15
22926966    2002200304M 1  1    15611   22926966   3   1   2   15
22926966    2002200304M 1  3    15611   22926966   3   1   2   15
22926966    2002200304M 1  5    15611   22926966   3   1   2   15
22926966    2002200304M 1  7    15611   22926966   3   1   2   15
22926966    2002200304M 1  9    15611   22926966   3   1   2   15
22926966    2002200304M 1 11    15611   22926966   3   1   2   15
22926966    2002200304M 1 13    15611   22926966   3   1   2   15
22926966    2002200304M 1 15    15611   22926966   3   1   2   15
22926966    2002200304M 1 17    15611   22926966   3   1   2   15
22926966    2002200304M 1 19    15611   22926966   3   1   2   15
22926966    2002200304M 1 21    15611   22926966   3   1   2   15
22945682    2002200304M 1  1    52611   22945682   3   1   2   52
22945682    2002200304M 1  3    52611   22945682   3   1   2   52
22953690    2002200304M 1  2    41311   22953690   3   1   2   41
22953690    2002200304M 1  4    41311   22953690   3   1   2   41
22953690    2002200304M 1  6    41311   22953690   3   1   2   41
23018762    2002200304M 2  2    25611   23018762   3   1   2   25
23018762    2002200304M 2  4    25611   23018762   3   1   2   25
23018762    2002200304M 2  6    25611   23018762   3   1   2   25
23018762    2002200304M 2  8    25611   23018762   3   1   2   25
23018762    2002200304M 2 10    25611   23018762   3   1   2   25
23018762    2002200304M 2 12    25611   23018762   3   1   2   25
23018762    2002200304M 2 14    25611   23018762   3   1   2   25
23018762    2002200304M 2 16    25611   23018762   3   1   2   25
23018762    2002200304M 2 18    25611   23018762   3   1   2   25
23018762    2002200304M 2 20    25611   23018762   3   1   2   25
23018762    2002200304M 2 22    25611   23018762   3   1   2   25
23026454    2002200304M 1  1    22111   23026454   3   1   2   22
23026454    2002200304M 1  3    22111   23026454   3   1   2   22
23026454    2002200304M 1  5    22111   23026454   3   1   2   22
24161722    2003200304M 1  5    28311   24161722   3   1   2   28
24161722    2003200304M 1  7    28311   24161722   3   1   2   28
24161722    2003200304M 1  9    28311   24161722   3   1   2   28
24161722    2003200304M 1 11    28311   24161722   3   1   2   28
24161722    2003200304M 1 13    28311   24161722   3   1   2   28
24161731    2003200304M 1  2    28311   24161731   3   1   2   28
24161731    2003200304M 1  4    28311   24161731   3   1   2   28
24161740    2003200304M 1  1    28311   24161740   3   1   2   28
24161740    2003200304M 1  3    28311   24161740   3   1   2   28
24161740    2003200304M 1  5    28311   24161740   3   1   2   28
24161740    2003200304M 1  7    28311   24161740   3   1   2   28
```

Figura 5.4: Parte de um dos arquivos ASCII do SAEB 2003 [1].

5.4 Extração de Regras de Associação de Dados Educacionais

Com os dados educacionais disponibilizados pelo INEP foi realizado um estudo de caso aplicando-se a mineração de regras de associação na base de dados da "Prova de Matemática da 4° Série - SAEB 2003" (Arquivo ASCII como visualizado na Figura 5.4). Como os dados não estão representados no formato de itens, houve a necessidade de realizar um pré-processamento para convertê-los.

O pré-processamento foi efetuado por um programa, desenvolvido em C++, que obteve como entrada o arquivo ASCII e gerou como saída o arquivo no formato *.txt*, apresentado na Figura 5.6. No pré-processamento, primeiro foram selecionadas as informações julgadas relevantes, ou seja, aquelas informações que pode contribuir para uma tomada de decisão pelos gestores da educação. Por exemplo, as informações do aluno, como, a dependência administrativa da escola em que estuda (municipal, estadual, particular ou federal), a região do país em que reside (norte, nordeste, sudeste, sul ou centro-oeste), se é beneficiado pelo programa bolsa escola (sim ou não), entre outras.

DADOS ESTATÍSTICOS DO SISTEMA NACIONAL DE AVALIAÇÃO DA EDUCAÇÃO BÁSICA - Dicionário de Dados do arquivo MATEMATICA_04SERIE.TXT

Código	Nome da Variável	Descrição	Variáveis Categóricas		Início	Tamanho
			Categoria	Descrição		
VARIÁVEIS DE CONTROLE (Bloco 1)						
10	DEP_ADM	Dependência Administrativa(Estadual/Municipal/Particular)	1	Estadual	53	8
			2	Municipal		
			3	Particular		
			4	Federal		
11	LOCAL	Localização(Urbano/Rural)	1	Urbana	61	8
			2	Rural		
12	REDE	Rede de Ensino(Publica/Particular)	1	Pública	69	8
			2	Particular		
13	UF	Unidade da Federação	11	Rondônia	77	8
			12	Acre		
			13	Amazonas		
			14	Roraima		
			15	Pará		
			16	Amapá		
			17	Tocantins		
			21	Maranhão		
			22	Piauí		
			23	Ceará		
			24	Rio Grande do Norte		
			25	Paraíba		
			26	Pernambuco		
			27	Alagoas		
			28	Sergipe		
			29	Bahia		
			31	Minas Gerais		
			32	Espírito Santo		
			33	Rio de Janeiro		
			35	São Paulo		
			41	Paraná		
			42	Santa Catarina		
			43	Rio Grande do Sul		
			50	Mato Grosso do Sul		
			51	Mato Grosso		
			52	Goiás		
			53	Distrito Federal		
14	REGIAO	Região	1	Norte	93	8
			2	Nordeste		
			3	Sudeste		
			4	Sul		
			5	Centro-Oeste		

Figura 5.5: Parte do arquivo de variáveis selecionadas do SAEB 2003.

Na sequência do pré-processamento, um código identificador foi associado a cada variável selecionada, conforme pode ser visto na Figura 5.5. Com isso, unindo-se o código da variável com o seu conteúdo pôde-se constituir um item. Por exemplo, para a variável "dependência administrativa da escola" concatenou-se o código 10 com o valor presente no arquivo ASCII ("1", "2", "3" ou "4'"). O item 103, por exemplo, informa que o aluno estuda em uma escola particular. A base de dados na forma de itens foi construída de modo que cada linha representa as informações de um aluno, sendo cada linha denominada de uma transação. A base de dados pode ser visualizada na Figura 5.6.

Arquivo Editar Localizar Visualizar Formatar Linguagem Configurações Macro Executar Plugins Janela ?

BD_MATEMATICA_04SERIE.txt

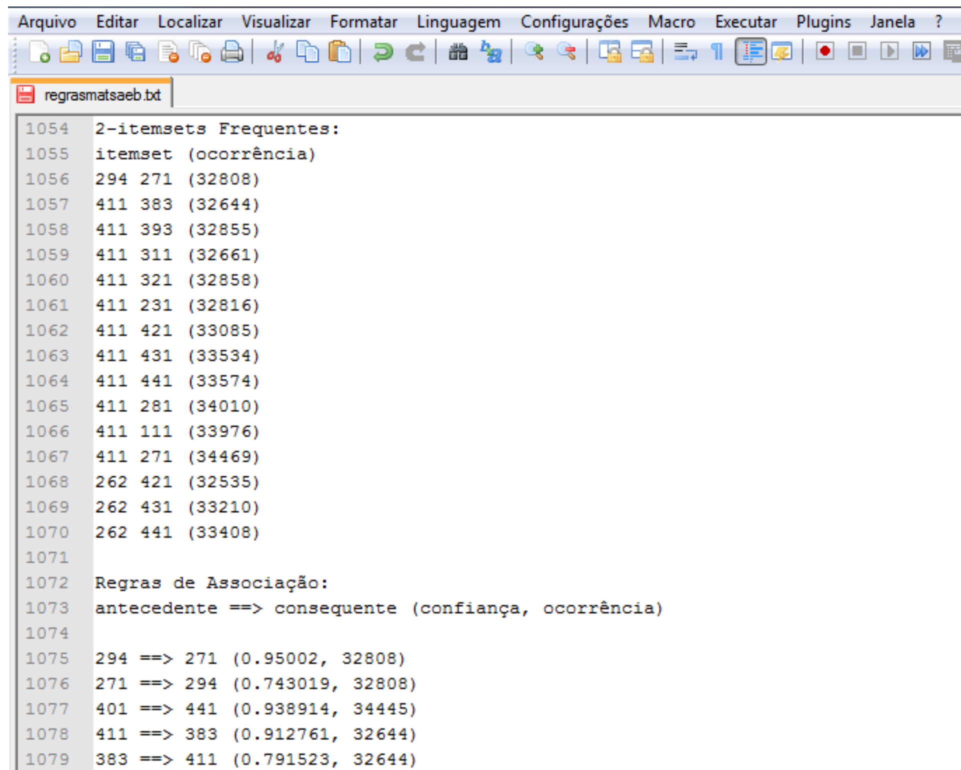
```
1  103 111 122 1315 141 152 161 173 182 191 201 212 223 231 241 251 261 271 281 292 303 311 321 339 341 351
2  103 111 122 1315 141 152 161 174 183 192 201 213 223 231 241 253 261 271 281 292 305 311 321 339 341 351
3  103 111 122 1315 141 152 162 173 182 192 201 212 223 231 242 251 262 271 281 292 306 311 321 339 341 351
4  103 111 122 1315 141 152 161 173 184 191 201 213 223 231 241 252 261 271 281 292 306 311 321 339 342 351
5  103 111 122 1315 141 152 161 173 184 191 201 211 223 231 241 251 261 271 281 292 305 311 321 339 341 351
6  103 111 122 1315 141 152 162 174 183 192 201 213 223 231 241 251 262 271 281 294 301 311 321 330 341 351
7  103 111 122 1315 141 152 161 174 183 191 201 213 223 231 241 254 262 271 281 292 306 311 321 339 341 351
8  103 111 122 1315 141 152 162 173 183 191 201 213 223 231 241 251 261 271 281 292 305 311 321 339 341 351
9  103 111 122 1315 141 152 161 173 183 191 201 213 222 231 242 251 261 271 281 292 303 311 321 339 341 351
10 103 111 122 1315 141 152 161 174 184 191 201 213 223 231 241 253 262 271 281 291 305 311 321 339 341 351
11 103 111 122 1315 141 152 162 173 181 191 201 211 222 231 241 254 261 271 282 294 303 313 321 338 342 352
12 103 111 122 1315 141 152 162 173 181 192 201 212 222 231 241 254 261 271 281 294 304 311 321 335 341 351
13 103 111 122 1315 141 152 164 173 183 192 201 211 222 231 242 254 262 271 281 294 304 311 321 337 341 351
14 103 111 122 1315 141 152 162 173 182 191 201 212 222 231 241 254 262 271 282 292 305 311 321 339 341 351
15 103 111 122 1315 141 152 162 173 181 191 202 214 224 231 242 254 262 271 282 294 306 311 321 334 341 351
16 103 111 122 1315 141 152 162 173 182 195 201 211 224 231 241 251 262 271 282 294 304 311 321 335 341 351
17 103 111 122 1315 141 152 161 173 182 191 201 211 221 231 241 251 262 271 282 294 304 311 321 335 341 351
18 103 111 122 1315 141 152 162 173 181 191 201 211 221 231 241 254 262 271 282 294 305 311 321 330 342 351
19 103 111 122 1315 141 152 162 175 181 191 202 211 222 231 241 254 262 271 282 291 305 311 321 334 341 351
20 103 111 122 1315 141 152 161 173 182 191 202 211 222 231 241 254 262 271 282 294 304 311 322 337 341 351
21 103 111 122 1315 141 152 162 173 181 191 202 214 224 231 242 254 262 271 282 294 304 311 321 332 341 351
22 103 111 122 1315 141 152 161 172 182 192 201 211 222 231 241 251 262 271 282 294 304 311 321 332 341 351
23 103 111 122 1315 141 152 164 173 181 191 202 214 222 231 242 251 262 271 281 294 304 311 321 335 343 351
24 103 111 122 1315 141 152 162 173 182 192 201 212 223 231 241 251 261 271 282 292 307 312 321 339 341 351
25 103 111 122 1315 141 152 162 172 183 193 201 211 222 231 242 254 262 271 281 292 304 311 321 337 342 351
26 103 111 122 1315 141 152 162 173 182 193 202 211 223 231 241 254 262 271 281 294 307 311 321 339 341 351
```

Figura 5.6: Parte da base de dados em itens da "Prova de Matemética da 4° Série - SAEB 2003".

Após o pré-processamento realizado com os dados da "Prova de Matemática da 4° Série - SAEB 2003", foi aplicada a mineração de regras de associação utilizando a implementação do algoritmo Apriori com um valor de frequência na ordem de 70%. A base de dados em itens foi composta por 38980 transações. Com isso, foram considerados "frequentes" os *itemsets* que ocorreram em pelo menos 27286 transações. Como resultado obteve-se o conjunto composto pelos *itemsets* frequentes juntamente com as regras de associação extraídas, conforme pode ser visto em parte na Figura 5.7.

Em uma análise realizada com as regras de associação, observou-se que grande parte das regras não apresentaram informações relevantes, ou seja, não revelaram informações que possam contribuir na melhoria do sistema educacional no país. Muitas regras extraídas mostraram informações redundantes, como exemplo, a regra *231→271*, que indicou que 99% dos alunos que possuem geladeira em casa também possuem energia elétrica. Outras regras obtidas não demonstraram nenhum significado, por exemplo, a regra *411→281*, que indicou que 95% dos alunos que fazem a lição da escola em casa também possuem água que chega pela torneira onde mora. Porém, existiram regras que revelaram informações significativas, por exemplo, a regra *441→393*, que informou que 82% dos alunos que re-

cebem incentivos do professor para estudar nunca deixaram de frequentar a escola. Na Figura 5.7 pode-se visualizar algumas regras de associação juntamente com o seu valor e confiança e de ocorrência. Por exemplo, a regra *294→271*, obtida a partir do *itemset 294 271*, possui ocorrência em 32808 transações, nas quais 95% das transações que ocorreu o item *294* também ocorreu o item *271*.




```
1054 2-itemsets Frequentes:
1055 itemset (ocorrência)
1056 294 271 (32808)
1057 411 383 (32644)
1058 411 393 (32855)
1059 411 311 (32661)
1060 411 321 (32858)
1061 411 231 (32816)
1062 411 421 (33085)
1063 411 431 (33534)
1064 411 441 (33574)
1065 411 281 (34010)
1066 411 111 (33976)
1067 411 271 (34469)
1068 262 421 (32535)
1069 262 431 (33210)
1070 262 441 (33408)
1071
1072 Regras de Associação:
1073 antecedente ==> consequente (confiança, ocorrência)
1074
1075 294 ==> 271 (0.95002, 32808)
1076 271 ==> 294 (0.743019, 32808)
1077 401 ==> 441 (0.938914, 34445)
1078 411 ==> 383 (0.912761, 32644)
1079 383 ==> 411 (0.791523, 32644)
```

Figura 5.7: Parte dos itemsets frequentes e das regras de associação extraídos da "Prova de Matemética da 4° Série - SAEB 2003".

Capítulo 6

Implementações e Resultados

6.1 Conceitos Gerais

Neste capítulo são apresentados os resultados obtidos com as implementações dos Algoritmos Paralelos Apriori, Eclat e *FP-Growth* para a mineração de *itemsets* frequentes. Na Seção 6.2 é apresentado o ambiente computacional utilizado para realizar os experimentos com as implementações. Na Seção 6.3 são apresentadas as bases de dados de entrada utilizadas com as implementações nos experimentos. Na Seção 6.4 são apresentadas as bases de dados de saída construídas pelas implementações. Na Seção 6.5 é feita uma abordagem sobre os tempos de processamento dos experimentos. Por fim, na Seção 6.6 é efetuada uma análise dos resultados obtidos.

6.2 Ambiente Computacional

Os Algoritmos Paralelos Apriori, Eclat e *FP-Growth*, apresentados no Capítulo 4, foram implementados na linguagem de programação C++, utilizando o sistema operacional Linux. Para a tarefa de comunicação entre processadores foi utilizada a biblioteca MPI (*Message Passing Interface*).

As implementações foram executadas no *PC Cluster* (*HPCVL - High Performance Computing Virtual Laboratory*) [20] da **Carleton University**, Canadá. O *PC Cluster* (*HPCVL*) é composto de 256 processadores em 64 nós com 4x2.2 GHz Opteron Cores e 8 GB RAM por nó. Todos os nós são interconectados através de um *switch* SuperX Foundry usando Gigabit Ethernet.

6.3 Bases de Dados de Entrada

Para realizar os procedimentos de validação dos resultados de mineração e de coleta de tempos de processamento, foram definidas nove bases de dados com tamanhos diferentes

para serem processadas pelas implementações dos Algoritmos Apriori, Eclat e *FP-Growth*. As bases de dados foram construídas em arquivos no formato *.txt* nos quais cada linha de texto correspondia a uma transação com 32 itens. Cada transação foi obtida a partir de um conjunto inicial de 64 itens, identificados de 1 a 64, e tomados em quantidade de 32 itens para compor cada transação, de forma que cada item tivesse uma frequencia de 50% na base de dados. Os itens nas transações aparecem em ordem crescente em cada uma delas. A Figura 6.1 mostra parte de uma base de dados de entrada.


```
1   1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63
2   2 3 5 8 9 11 14 15 17 20 21 23 26 27 29 32 33 35 38 39 41 44 45 47 50 51 53 56 57 59 62 63
3   1 4 5 7 10 11 13 16 17 19 22 23 25 28 29 31 34 35 37 40 41 43 46 47 49 52 53 55 58 59 61 64
4   2 4 5 8 10 11 14 16 17 20 22 23 26 28 29 32 34 35 38 40 41 44 46 47 50 52 53 56 58 59 62 64
5   1 3 6 7 9 12 13 15 18 19 21 24 25 27 30 31 33 36 37 39 42 43 45 48 49 51 54 55 57 60 61 63
6   2 3 6 8 9 12 14 15 18 20 21 24 26 27 30 32 33 36 38 39 42 44 45 48 50 51 54 56 57 60 62 63
7   1 4 6 7 10 12 13 16 18 19 22 24 25 28 30 31 34 36 37 40 42 43 46 48 49 52 54 55 58 60 61 64
8   2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64
9   1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63
10  2 3 5 8 9 11 14 15 17 20 21 23 26 27 29 32 33 35 38 39 41 44 45 47 50 51 53 56 57 59 62 63
11  1 4 5 7 10 11 13 16 17 19 22 23 25 28 29 31 34 35 37 40 41 43 46 47 49 52 53 55 58 59 61 64
12  2 4 5 8 10 11 14 16 17 20 22 23 26 28 29 32 34 35 38 40 41 44 46 47 50 52 53 56 58 59 62 64
13  1 3 6 7 9 12 13 15 18 19 21 24 25 27 30 31 33 36 37 39 42 43 45 48 49 51 54 55 57 60 61 63
14  2 3 6 8 9 12 14 15 18 20 21 24 26 27 30 32 33 36 38 39 42 44 45 48 50 51 54 56 57 60 62 63
15  1 4 6 7 10 12 13 16 18 19 22 24 25 28 30 31 34 36 37 40 42 43 46 48 49 52 54 55 58 60 61 64
16  2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64
```

Figura 6.1: Parte da base de dados de entrada com 256 transações.

Foram construídas bases de dados com 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 e 65536 transações. Cada uma das base de dados foi utilizada para a execução de todas as implementações utilizando-se 1, 2, 4, 8, 16 e 32 processadores.

De acordo com a estratégia de mineração de cada implementação, as bases de dados de entrada foram fornecidas de maneira compartilhada ou particionada para os processadores. Para as implementações dos Algoritmos Apriori e *FP-Growth* as bases de dados de entrada foram particionadas de forma que cada processador recebeu m/p transações, em que m é o número de transações na base de dados e p o número de processadores em execução. Com a implementação do Algoritmo Eclat, as bases de dados de entrada foram compartilhadas pelos processadores em operação.

6.4 Bases de Dados de Saída

Durante a execução das implementações dos Algoritmos Apriori, Eclat e *FP-Growth*, cada processador em atividade gerou a sua base de dados de saída, que consistiu no conjunto de *itemsets* com frequência mínima de 50% nas transações das bases de dados de entrada. Devido cada item estar presente em 50% das transações, os *itemsets* frequentes determinados tiveram um valor de suporte igual à $m/2$, em que m é número de transações na base de dados de entrada.

De acordo com a estratégia de mineração utilizada pela implementação, as bases de dados de saída foram construídas em arquivos no formato *.txt*, possuindo o conjunto de *itemsets* minerados de modo distribuído ou não-distribuído entre os processadores. Porém,

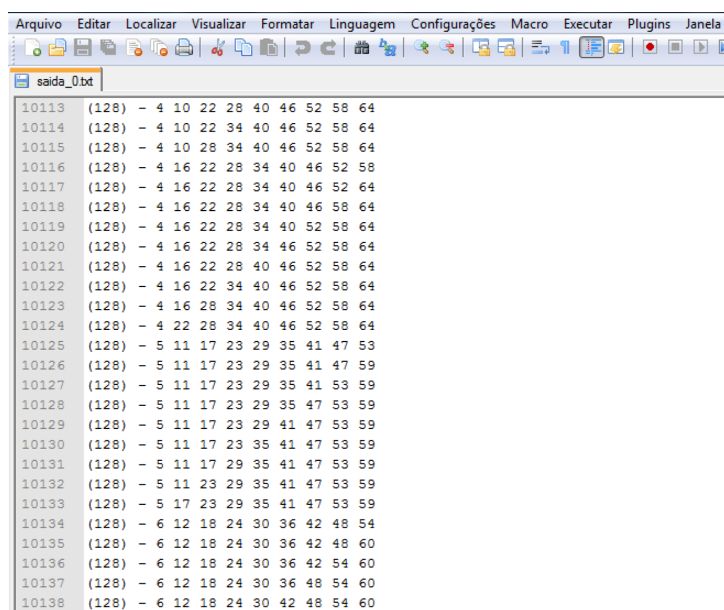
os *itemsets* determinados foram iguais na forma distribuída e não-distribuída.

Na implementação do Algoritmo Apriori, a saída foi igual em todos os processadores. Isso porque, durante a sua execução, cada processador possui uma cópia do conjunto de todos os k-*itemsets* candidatos gerados, e através de uma redução global, todos processadores determinam a frequência dos k-*itemsets*. Dessa forma, todos os processadores possuem o mesmo conjunto de *itemsets* frequentes no final da execução.

Na implementação do Algoritmo Eclat, a saída ocorreu de maneira distribuída entre os processadores. Na sua estratégia de mineração, inicialmente, todos os 1-*itemsets* são distribuídos entre os processadores, e em seguida, todos os k-*itemsets* são determinados a partir dos (k-1)-*itemsets* presentes localmente em cada processador. Dessa forma, cada processador possui uma parte do conjunto total de *itemsets* frequentes minerados no final do processo de execução.

Por fim, na implementação do Algoritmo *FP-Growth*, a saída também ocorreu de forma distribuída entre os processadores. Com a utilização das estruturas associadas *FP-Tree* e *node-links* para a mineração de *itemsets*, cada processador é responsável pela determinação da frequência de um subconjunto de *itemsets*. Com isso, o conjunto de todos os *itemsets* frequentes é obtido de forma distribuída entre todos os processadores em atividade.

A Figura 6.2 apresenta uma pequena parte da base de dados de saída obtida a partir do processamento da base de dados de entrada com 256 transações. Observa-se que são mostrados 9-*itemsets* com um valor de suporte equivalente a 128.




```
10113 (128) - 4 10 22 28 40 46 52 58 64
10114 (128) - 4 10 22 34 40 46 52 58 64
10115 (128) - 4 10 28 34 40 46 52 58 64
10116 (128) - 4 16 22 28 34 40 46 52 58
10117 (128) - 4 16 22 28 34 40 46 52 64
10118 (128) - 4 16 22 28 34 40 46 58 64
10119 (128) - 4 16 22 28 34 40 52 58 64
10120 (128) - 4 16 22 28 34 46 52 58 64
10121 (128) - 4 16 22 28 40 46 52 58 64
10122 (128) - 4 16 22 34 40 46 52 58 64
10123 (128) - 4 16 28 34 40 46 52 58 64
10124 (128) - 4 22 28 34 40 46 52 58 64
10125 (128) - 5 11 17 23 29 35 41 47 53
10126 (128) - 5 11 17 23 29 35 41 47 59
10127 (128) - 5 11 17 23 29 35 41 53 59
10128 (128) - 5 11 17 23 29 35 47 53 59
10129 (128) - 5 11 17 23 29 41 47 53 59
10130 (128) - 5 11 17 23 35 41 47 53 59
10131 (128) - 5 11 17 29 35 41 47 53 59
10132 (128) - 5 11 23 29 35 41 47 53 59
10133 (128) - 5 17 23 29 35 41 47 53 59
10134 (128) - 6 12 18 24 30 36 42 48 54
10135 (128) - 6 12 18 24 30 36 42 48 60
10136 (128) - 6 12 18 24 30 36 42 54 60
10137 (128) - 6 12 18 24 30 36 48 54 60
10138 (128) - 6 12 18 24 30 42 48 54 60
```

Figura 6.2: Parte da base de dados de saída obtida com 256 transações.

6.5 Tempos de Processamento

Os tempos de processamento consistem nos tempos execução das implementações dos Algoritmos Apriori, Eclat e *FP-Growth* para realizar o cumprimento das tarefas de leitura de transações nas bases de dados de entrada, comunicação de dados entre os processadores, computação de dados locais e escrita de *itemsets* frequentes nas bases de dados de saída. O tempo de processamento de cada implementação foi determinado considerando a média dos tempos de execução, em segundos, dos processadores envolvidos. Para uma melhor visualização dos tempos obtidos, foram determinados os tempos de leitura, comunicação, computação e escrita de maneira independente.

6.5.1 Tempo de Leitura

O tempo de leitura é o tempo em segundos utilizado pelos processadores para efetuarem a leitura das transações nas bases de dados de entrada. Abordando as estratégias de leitura das implementações, no Algoritmo Apriori, cada processador realiza uma varredura nas transações presentes na base de dados local toda a vez em que um conjunto de *k-itemsets* candidatos é gerado. No Algoritmo Eclat, cada processador realiza a leitura das transações na base de dados de entrada apenas uma única vez, na qual é utilizada para criar a estrutura de dados no formato vertical. Na implementação do Algoritmo *FP-Growth*, cada processador realiza duas varreduras em sua respectiva base de dados local, em que a primeira varredura é utilizada para determinar a frequência local dos *itemsets* e a segunda varredura é utilizada para mapear as transações da estrutura *FP-Tree*.

6.5.2 Tempo de Comunicação

O tempo de comunicação representa o custo obtido pelas implementações em segundos para a tarefa de troca de informações entre os processadores em operação. Abordando as implementações realizadas, no Algoritmo Apriori, a cada conjunto de *k-itemsets* de candidatos gerados ocorre uma comunicação entre todos os processadores para determinar a frequência de cada *itemset*. Na implementação do Algoritmo Eclat não existe nenhuma comunicação entre os processadores devido às características da sua estratégia de mineração de *itemsets*. Na implementação do Algoritmo *FP-Growth* ocorrem duas etapas de comunicações entre os processadores, em que, inicialmente, é realizada uma redução global para determinar a frequência dos 1-*itemsets* que são armazenados na estrutura *node-links* e, em seguida, uma comunicação global para atualizar os valores de suporte de cada *itemset* na estrutura *FP-Tree*.

6.5.3 Tempo de Computação Local

O tempo de computação local é o tempo em segundos utilizado por cada processador para realizar a tarefa de processamento dos dados contidos na memória local. Para uma entrada de m transações, durante o processo de execução, cada processador mantém localmente

m/p transações, em que p é o número de processadores em operação. Portanto, quanto maior o número de processadores menor o volume de dados presente a cada processador.

6.5.4 Tempo de Escrita

O tempo de escrita é tempo obtido pelos processadores em segundos para realização da tarefa de escrita nas bases de dados de saída. Durante a execução das implementações cada processador gera a sua base de dados local, na qual contém o conjunto de *itemsets* frequentes determinados em sua computação local. Conforme descrito na Seção 6.4, cada implementação possui uma estratégia diferente para a mineração dos *itemsets* frequentes, e, com isso, as bases de dados de saída também resultam em formatos diferentes entre as implementações.

6.6 Análise dos Resultados Obtidos

Nesta seção, são apresentados os resultados dos experimentos realizados com as implementações dos Algoritmos Apriori, Eclat e *FP-Growth*. Nos experimentos utilizou-se um suporte no valor de 50%, ou seja, foram minerados os *itemsets* que ocorrem em 50% das transações nas bases de dados de entrada. Os tempos de processamento obtidos pelas implementações são analisados e apresentados a seguir.

6.6.1 Tempos de Processamento do Apriori

Nos experimentos realizados com a implementação do Algoritmo Apriori foram utilizadas nove bases de dados de entrada com tamanhos diferentes, na qual foram processadas com 1, 2, 4, 8, 16 e 32 processadores. A Figura 6.3 ilustra os tempos de processamento obtidos pelo Algoritmo Apriori nos experimentos.

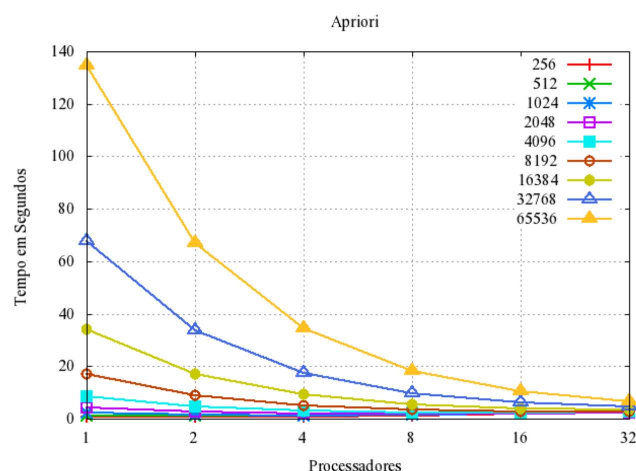


Figura 6.3: Tempos de processamento do Algoritmo Apriori.

Observa-se na Figura 6.3 que as entradas com maiores quantidades de transações (8192, 16384, 32768, 65536 transações) possuem uma diminuição significativa nos tempos de processamento conforme aumenta-se o número de processadores. No gráfico foram considerados os tempos totais de processamento obtidos pela implementação, que inclui os tempos de leitura, comunicação, computação e escrita.

Para uma melhor visualização dos tempos de processamento do Algoritmo Apriori, foram obtidos separadamente os tempos de leitura, comunicação, computação e escrita. Para todas as base de dados de entrada foram realizados experimentos no propósito de analisar o desempenho da implementação. Os gráficos dos tempos de processamento obtidos são apresentados nas Figuras 6.4 e 6.5.

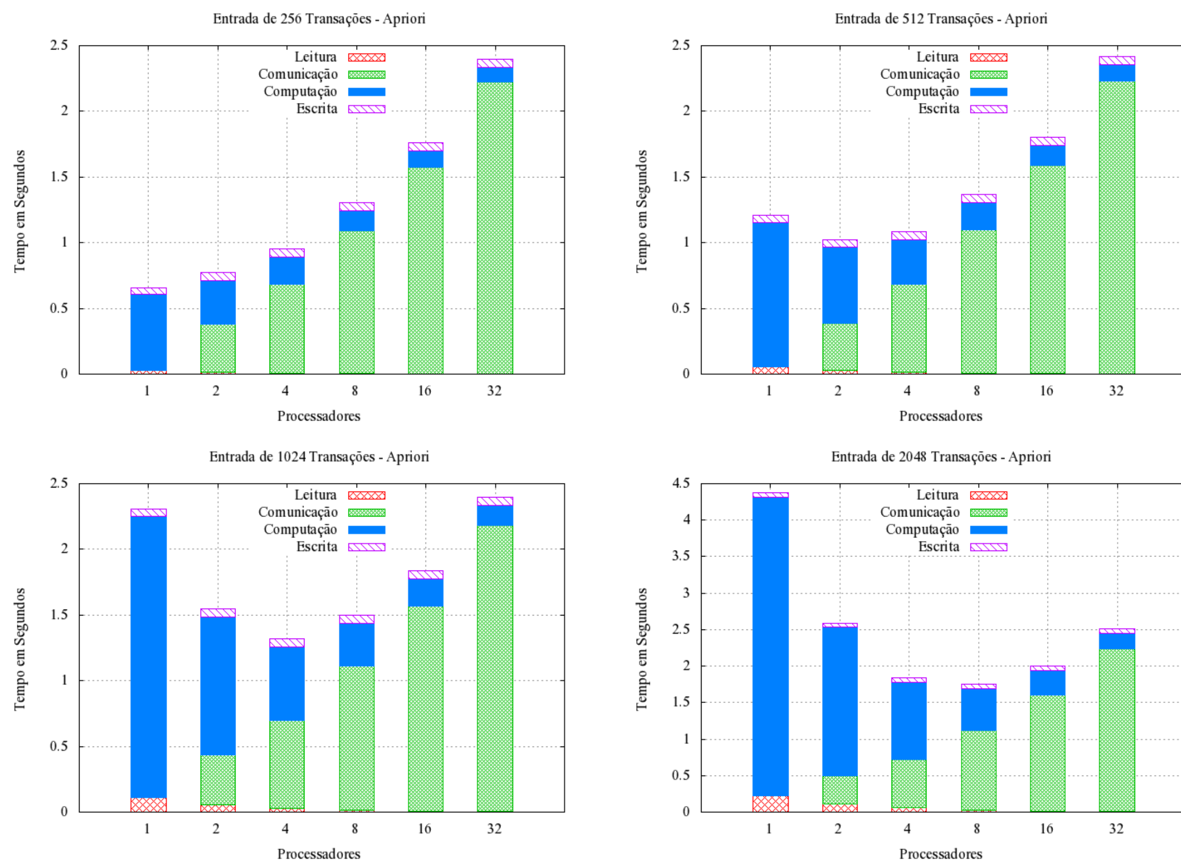


Figura 6.4: Tempos de processamento do Algoritmo Apriori com 256, 512, 1024 e 2048 transações.

Na Figura 6.4, observa-se que com a base de dados de 256 transações existe um crescimento no tempo total de processamento conforme aumenta-se o número de processadores. Percebe-se que o fator determinante para o crescimento do tempo total é o tempo de comunicação entre os processadores, de modo que, os tempos de computação local são inferiores aos tempos de comunicação, exceto com 1 processador onde não há comunicação. Dessa forma, a entrada com 256 transações é pequena para se obter reduções de tempo no processamento paralelo. Observa-se ainda, que conforme aumenta-se o número de processadores, os tempos de computação local decrescem (devido à divisão da carga

de trabalho), os tempos de escrita permanecem estáveis (devido à estratégia da implementação, em que todos processadores determinam a mesma saída) e os tempos de leitura não são visualizados (consequência do pequeno volume de transações processadas).

Observando o gráfico que ilustra o experimento realizado com 512 transações, percebe-se que existe uma redução no tempo de processamento quando a implementação é executada com 2 processadores. Nesse caso, o tempo de comunicação é menor que o tempo de processamento da carga de trabalho local. A mesma situação já não ocorre com as demais quantidades de processadores. Observa-se, também, que o tempo de leitura da base de dados de entrada pode ser visualizado com 1 e 2 processadores, isso em decorrência do aumento da quantidade de transações em relação ao gráfico obtido com 256 transações. No gráfico do experimento com 1024 transações, percebe-se que a redução de tempo de processamento ocorre com 2 e 4 processadores. Já no gráfico obtido com a base de dados de 2048 transações, a redução do tempo de execução ocorre com 2, 4 e 8 processadores, ou seja, conforme aumentou-se o volume de transações no processamento, o desempenho da aplicação obteve um crescimento.

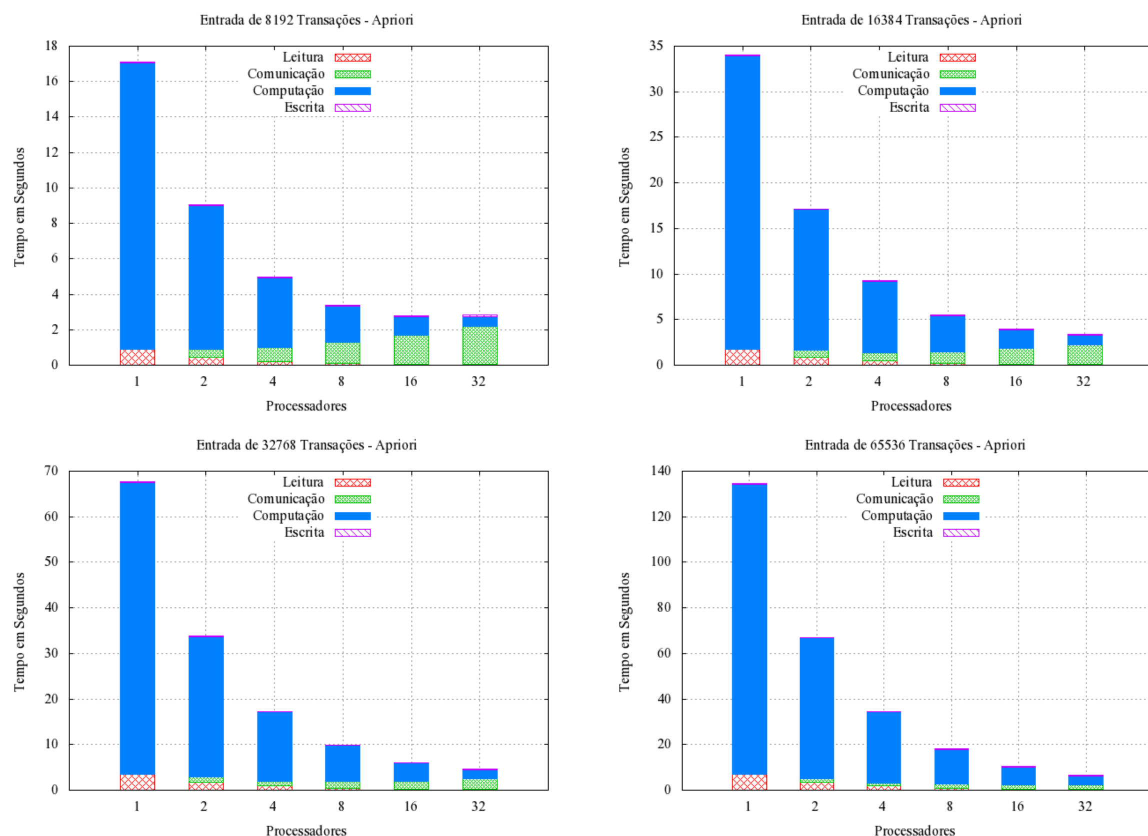


Figura 6.5: Tempos de processamento do Algoritmo Apriori com 8192, 16384, 32768 e 65536 transações.

Na Figura 6.5 os experimentos confirmam que, conforme o tamanho das bases de entrada aumenta, reduz-se o tempo de execução da implementação do Algoritmo Apriori no ambiente paralelo. Para a base de dados com 8192 transações o tempo de processamento

é reduzido com 2, 4, 8 e 16 processadores. Em relação à maior base de dados utilizada nos experimentos, ou seja, a entrada com 65536 transações, a redução de tempo de processamento ocorre com todas as quantidades de processadores utilizadas (2, 4, 8, 16 e 32 processadores).

Concluindo a análise da implementação do Algoritmo Apriori, observa-se que para entradas com uma quantidade de transações inferior a 16384 não se obtém reduções de tempos de processamento quando executadas com até 32 processadores. Isso porque o tempo de comunicação não é compensado pelo tempo de computação local durante o processamento de uma baixa quantidade de transações. Com relação aos tempos de leitura, percebe-se que conforme o número de processadores em atividade aumenta, os custos de leitura são reduzidos, isso devido a divisão do conjunto de transações entre os processadores. Analisando o tempo de escrita, observa-se que os custos permanecem estáveis quando se aumenta a quantidade de processadores em operação, pois as saídas são iguais em todos os processadores. Outro fator a ser observado nos tempos de escrita é que mesmo com o aumento da quantidade de transações no processamento, o tempo de escrita permanece estável, pois a quantidade de *itemsets* frequentes minerados permanece igual, o que cresce é apenas o valor de frequência dos *itemsets*.

6.6.2 Tempos de Processamento do Eclat

Na implementação do Algoritmo Eclat foram realizados experimentos com diferentes bases de dados de entrada, em que cada base de dados foi processada com 1, 2, 4, 8, 16 e 32 processadores. A Figura 6.6 ilustra o desempenho do Algoritmo Eclat no ambiente paralelo.

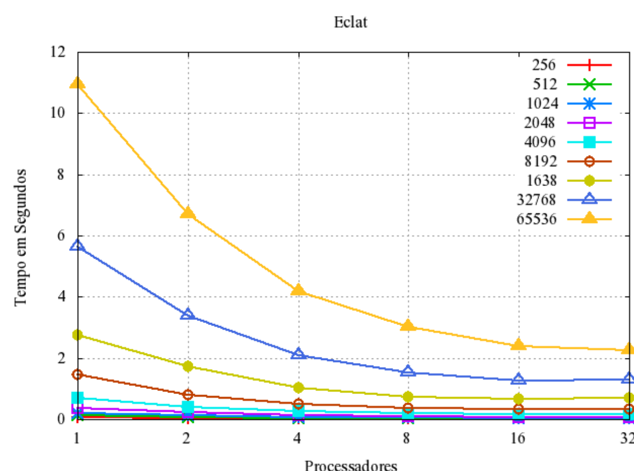


Figura 6.6: Tempos de processamento do Algoritmo Eclat.

Observa-se na Figura 6.6 que, para as entradas com maiores quantidades de transações, há uma redução no tempo de execução conforme cresce no número de processadores em operação. Nesse experimento, utilizou-se o tempo total de processamento, ou melhor, considerou-se os tempos de leitura, comunicação, computação e escrita. Os tempos de

processamento de forma individual são apresentados nas Figuras 6.7 e 6.8.

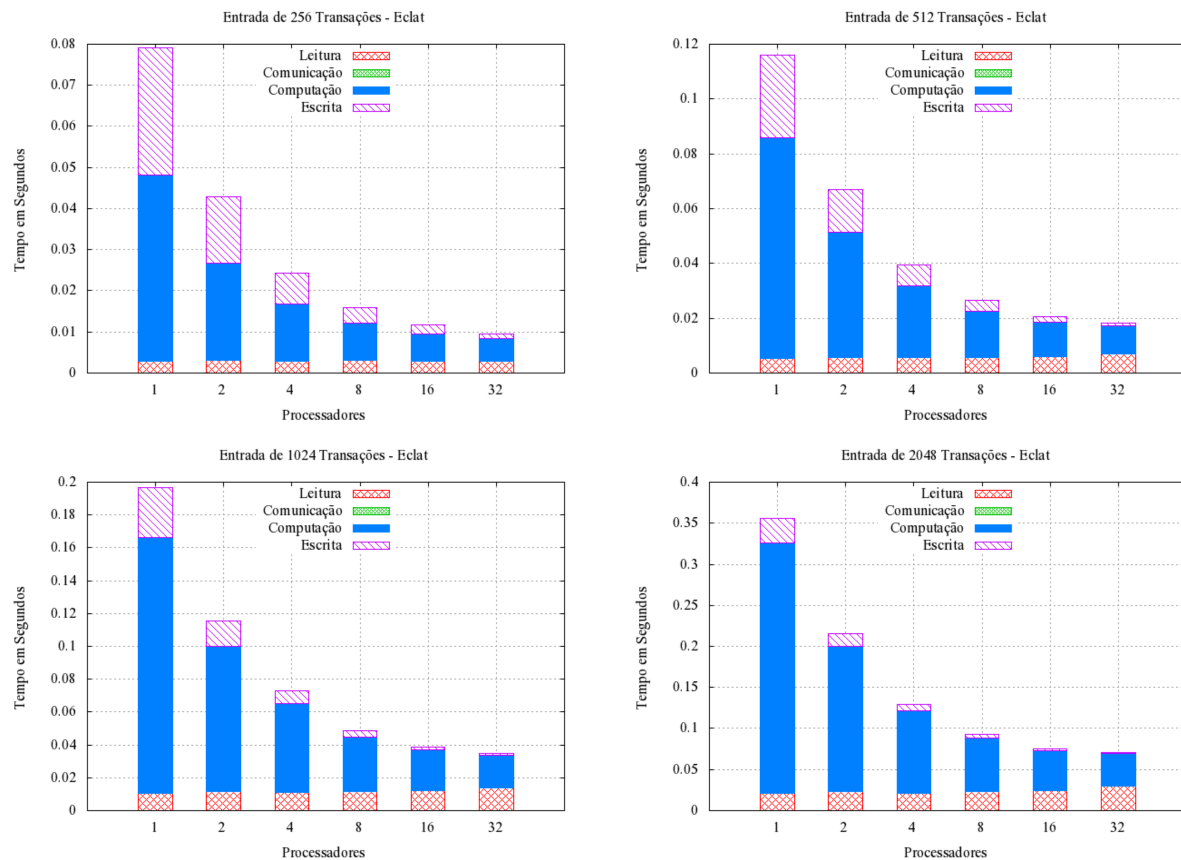


Figura 6.7: Tempos de processamento do Algoritmo Eclat com 256, 512, 1024 e 2048 transações.

Na Figura 6.7, observa-se no gráfico obtido com 256 transações que, a redução do tempo de processamento ocorre conforme o número de processadores aumenta, utilizando no experimento 1, 2, 4, 8, 16 e 32 processadores. Isso ocorre por não ser necessário comunicação entre os processadores durante a execução da implementação, o que é uma das principais características do Algoritmo Eclat. Percebe-se, também, que o tempo de leitura permanece igual para todas as execuções com diferentes quantidades de processadores no ambiente de execução, pois a base e dados de entrada não é particionada entre os processadores, ou seja, a base de dados original é varrida igualmente por todos os processadores. Com relação à saída, observa-se que o tempo para gerar as bases de dados de saída diminui gradativamente conforme o número de processadores em operação aumenta, uma vez que a base de dados de saída é gerada de forma distribuída entre os processadores, ou seja, quanto maior o número de processadores em atividade, menor a quantidade de *itemsets* minerados por cada processador no ambiente.

Analisando a estratégia do Algoritmo Eclat com os experimentos realizados, percebe-se que os ganhos de tempo no processamento paralelo ocorrem independente do volume de transações processadas. Por não haver comunicação entre processadores, são computados

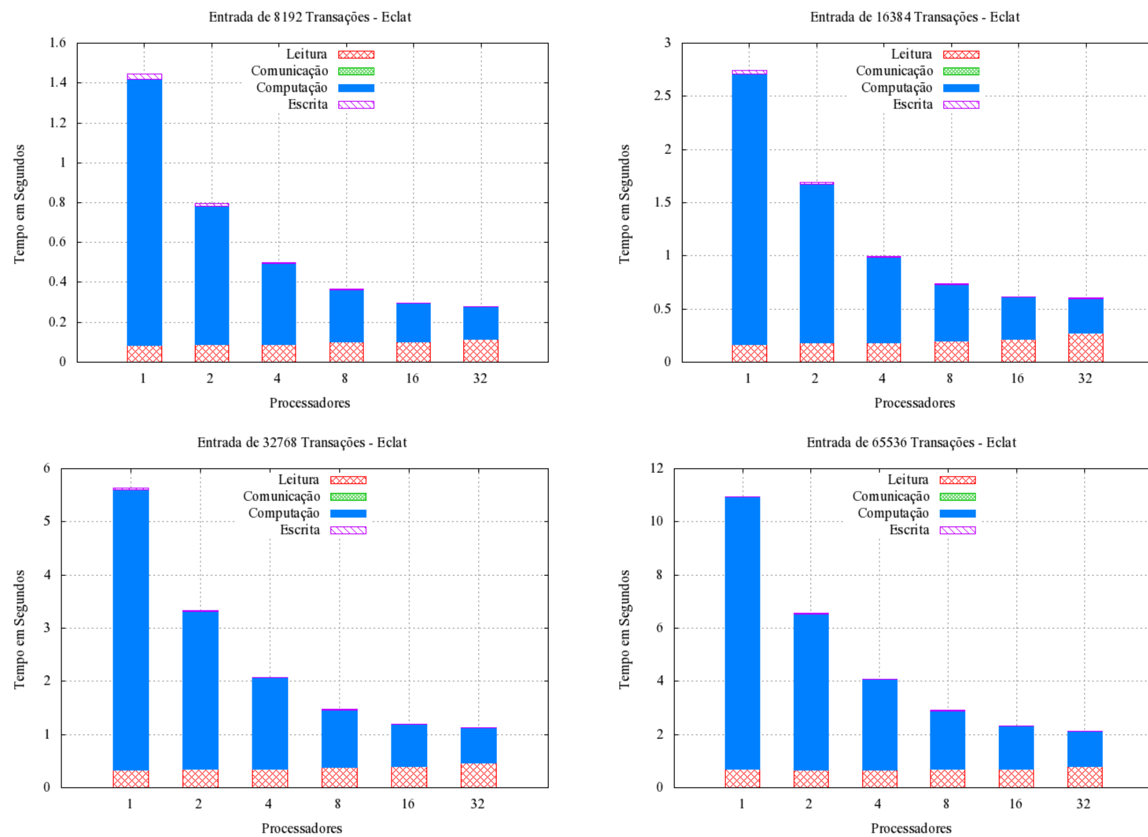


Figura 6.8: Tempos de processamento do Algoritmo Eclat com 8192, 16384, 32768 e 65536 transações.

apenas o tempo de leitura (um tempo fixo para cada tamanho de base de dados), o tempo de computação (tempo gasto para processar as classes de *itemsets* que lhe foram atribuídas) e o tempo de escrita (tempo gasto para gerar a base de dados de saída). É importante também lembrar que o tempo de escrita não cresce como os outros tempos quando se aumenta o volume de dados a ser processado, pois, independente do tamanho da base de dados, a quantidade de *itemsets* distintos permanece igual, ou melhor, o que cresce é o valor de frequência de cada *itemset*, o que não causa um aumento no volume de dados para a escrita.

6.6.3 Tempos de Processamento do *FP-Growth*

Com a implementação do Algoritmo *FP-Growth* foram realizados experimentos com 1, 2, 4, 8, 16 e 32 processadores, em que foram utilizados diferentes tamanhos de bases de dados como entrada. A Figura 6.9 mostra os tempos de processamento obtidos pela implementação, no qual observa-se que, com as entradas de maiores quantidades de transações (8192, 16384, 32768 e 65536 transações) os tempos de processamento da implementação do Algoritmo *FP-Growth* decrescem com o aumento do número de processadores.

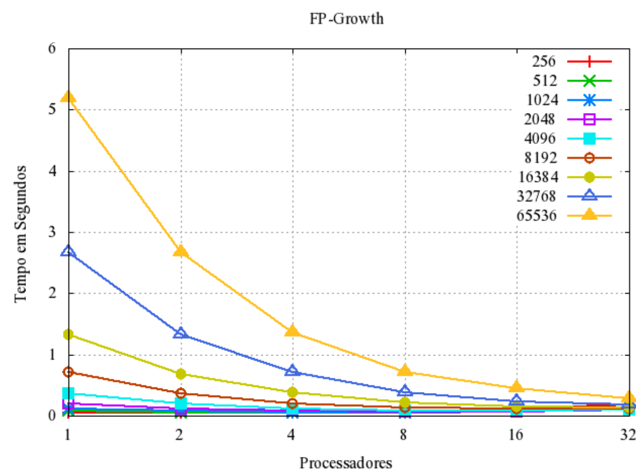


Figura 6.9: Tempos de processamento do Algoritmo *FP-Growth*.

Na Figura 6.9, os tempos de processamento incluem os tempos de leitura, comunicação, computação e escrita. Os tempos de processamento obtidos de maneira separada são apresentados nas Figuras 6.10 e 6.11.

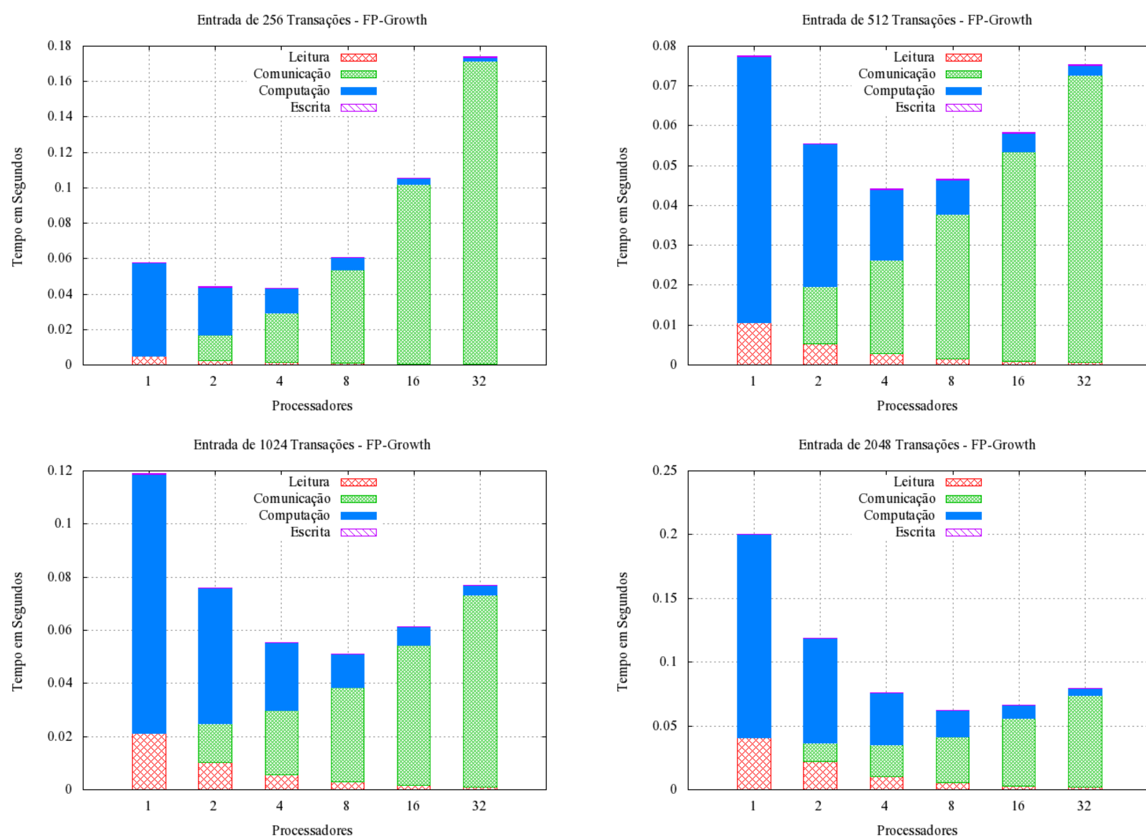


Figura 6.10: Tempos de processamento do Algoritmo *FP-Growth* com 256, 512, 1024 e 2048 transações.

Na Figura 6.10, observando o experimento realizado com 256 transações, percebe-se que a implementação do Algoritmo *FP-Growth* não proporciona reduções crescentes de tempo de processamento quando executada com diferentes números de processadores, pois o custo da comunicação realizada entre os processadores é maior que o custo do processamento local. Visualizando os demais gráficos presentes nas Figuras 6.10 e 6.11, percebe-se que conforme ocorre o aumento na quantidade de transações de entrada, o tempo de processamento decresce quando é utilizado um maior número de processadores no ambiente.

Com relação aos tempos de leitura, no Algoritmo *FP-Growth* as bases de dados de entrada são particionadas entre os processadores. Visualizando os gráficos, percebe-se que os tempos de leitura decrescem conforme o número de processadores aumenta. Analisando os tempos de escrita, observa-se que os custos para gerar as bases de dados de saída são estritamente baixos quando comparados aos demais tempos. Isso se deve à estratégia de escrita utilizada pelo Algoritmo *FP-Growth*, na qual as bases de dados de saída são construídas durante o percurso na estrutura *FP-Tree* quando são determinados os *itemsets* frequentes.

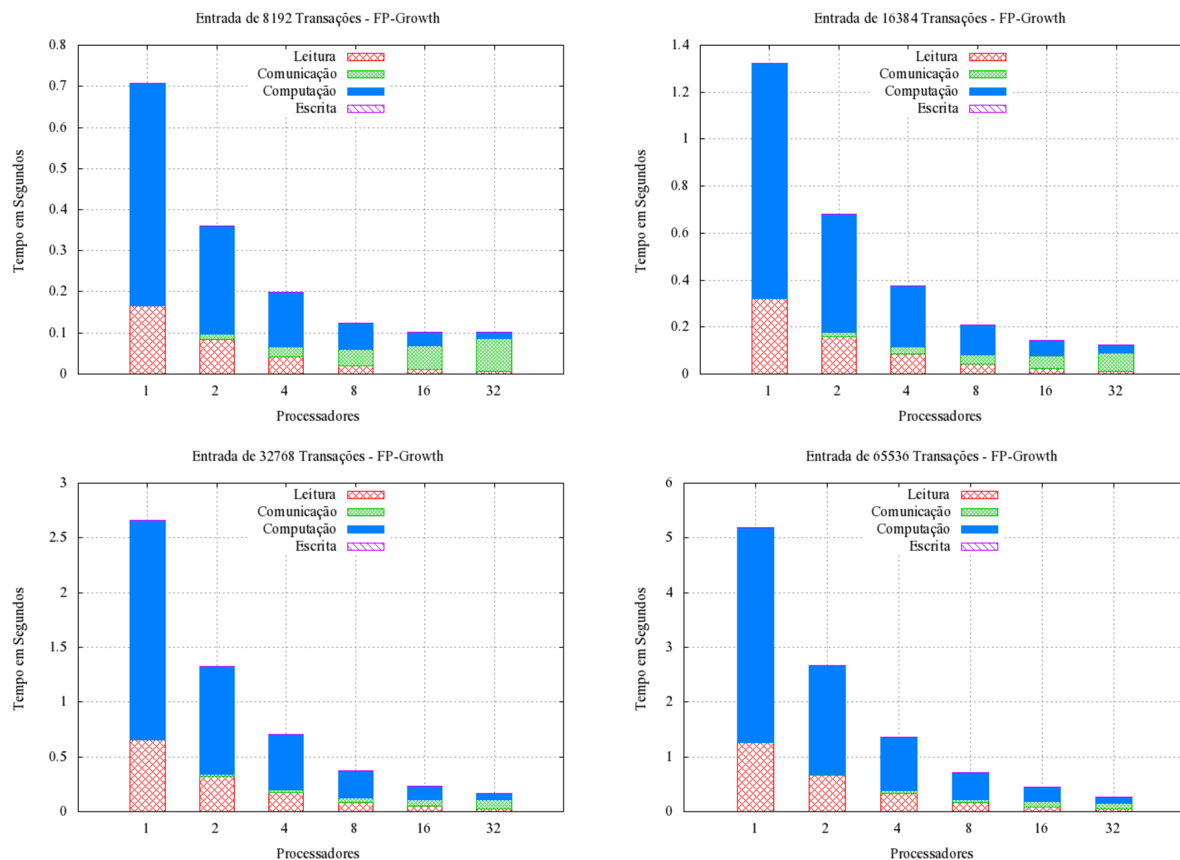


Figura 6.11: Tempos de processamento do Algoritmo *FP-Growth* com 8192, 16384, 32768 e 65536 transações.

6.6.4 Comparativo do Desempenho das Implementações

Realizando uma comparação entre os resultados obtidos pelas implementações dos Algoritmos Apriori, Eclat e *FP-Growth*, foram construídos gráficos com tempos de processamento e os *speedups* (ganhos) das implementações para algumas bases de dados de entrada. Os tempos de processamento e os *speedups* incluem os tempos de leitura, comunicação, computação e escrita, nos quais foram determinados com 1, 2, 4, 8, 16 e 32 processadores. Os gráficos dos tempos de processamento das implementações são apresentados na Figura 6.12. Os gráficos contendo os *speedups* são visualizados na Figura 6.13.

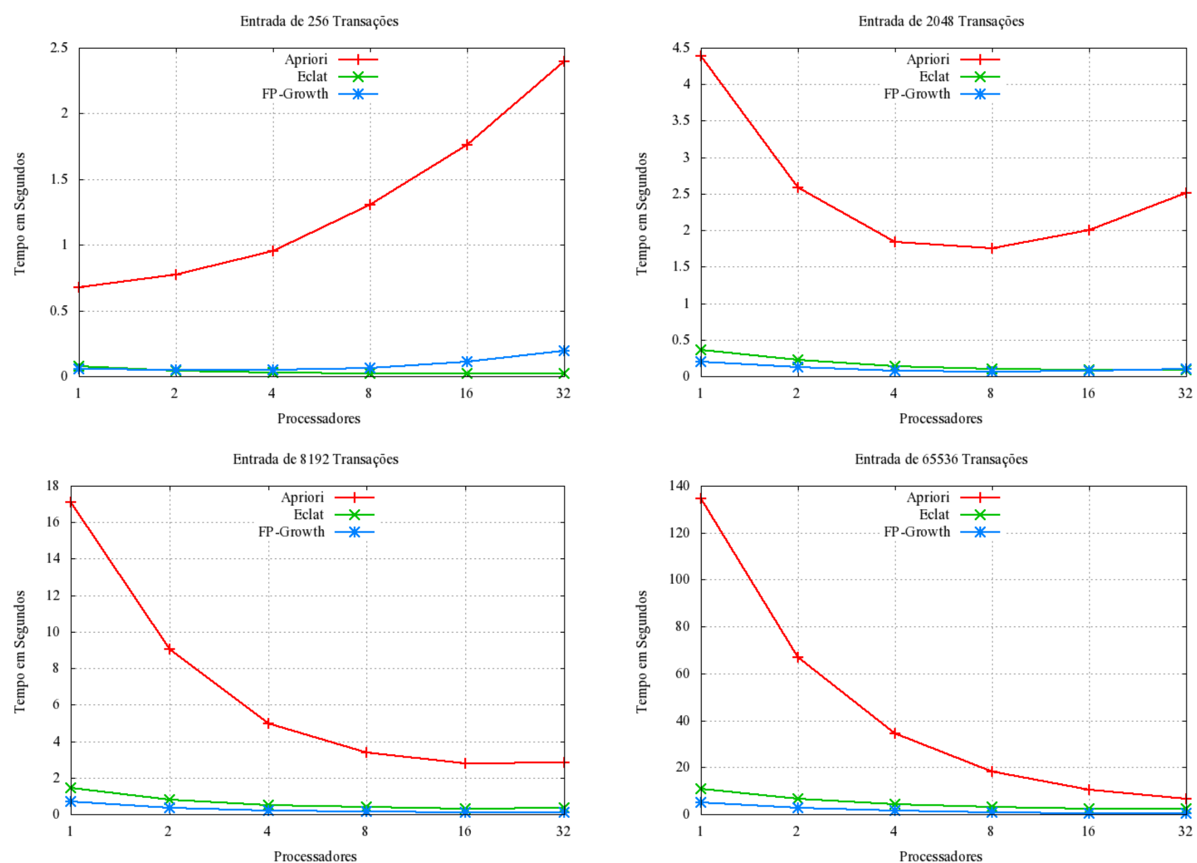


Figura 6.12: Tempos de processamento das implementações com 256, 2048, 8192 e 65536 transações.

Na Figura 6.12, observa-se nos gráficos apresentados, que os tempos de processamento do Algoritmo*FP-Growth* são inferiores aos tempos do Algoritmo Apriori e do Algoritmo Eclat com bases de dados a partir de 2048 transações. Percebe-se, também, que os tempos de processamento do Algoritmo Apriori são superiores aos tempos do Algoritmo Eclat e do Algoritmo *FP-Growth* para todos os tamanhos de bases de dados de entrada. A superioridade dos tempos do Algoritmo Apriori em relação aos tempos do Algoritmo Eclat e do Algoritmo *FP-Growth* está nos tempos de comunicação e de computação local. Com a base de dados de 256 transações, o tempo de comunicação do Algoritmo Apriori é superir ao tempo de comunicação do Algoritmo *FP-Growth* em aproximadamente 26 vezes

com 2 processadores e em 13 vezes com 32 processadores. No tempo de computação local, o Algoritmo Apriori é superior ao Algoritmo *FP-Growth* em aproximadamente 12 vezes com 2 processadores e em 52 vezes com 32 processadores. Os tempos de comunicação e de computação local com 256 transações são apresentados nas Tabelas 6.1 e 6.2.

Tabela 6.1: Tempos de comunicação das implementações com 256 transações.

Processadores	**Apriori**	**Eclat**	*FP-Growth*
1	0.000	0.000	0.000
2	0.3627	0.000	0.0141
4	0.6745	0.000	0.0279
8	1.0860	0.000	0.0526
16	1.5733	0.000	0.1011
32	2.2238	0.000	0.1712

Tabela 6.2: Tempos de computação local das implementações com 256 transações.

Processadores	**Apriori**	**Eclat**	*FP-Growth*
1	0.5762	0.0453	0.0524
2	0.3335	0.0238	0.0272
4	0.2111	0.0138	0.0137
8	0.1528	0.0091	0.0071
16	0.1235	0.0067	0.0048
32	0.1096	0.0055	0.0021

Nas Tabelas 6.3 e 6.4 estão, respectivamente, os tempos de comunicação e de computação local das implementações com a bases de dados de 65536 transações. Analisando os tempos de comunicação, percebe-se que a implementação do Algoritmo Apriori é superior à implementação do Algoritmo *FP-Growth* em aproximadamente 71 vezes com 2 processadores e em 25 vezes com 32 processadores em execução. Em relação aos tempos de computação local, observa-se que o Algoritmo Apriori é superior ao Algoritmo *FP-Growth* aproximadamente 31 vezes com 2 e 32 processadores.

Tabela 6.3: Tempos de comunicação das implementações com 65536 transações.

Processadores	**Apriori**	**Eclat**	*FP-Growth*
1	0.0000	0.0000	0.0000
2	1.4472	0.0000	0.0203
4	1.5425	0.0000	0.0447
8	1.6743	0.0000	0.0499
16	1.9973	0.0000	0.0967
32	2.1745	0.0000	0.0861

Para visualizar o desempenho das implementações dos Algoritmos Apriori, Eclat e *FP-Growth* foram construídos gráficos dos *speedups* (ganhos) para algumas bases de dados de entrada. O *speedup* é dado por: $S_p = t_1/t_p$, em que p é o número de processadores, t_1 é o tempo de processamento obtido com 1 processador e t_p é o tempo de processamento

Tabela 6.4: Tempos de computação local das implementações com 65536 transações.

Processadores	**Apriori**	**Eclat**	*FP-Growth*
1	127.2549	10.2426	3.9340
2	61.8744	5.8677	1.9861
4	31.1206	3.4129	0.9886
8	15.4060	2.2050	0.4964
16	7.7470	1.6027	0.2529
32	3.9455	1.3216	0.1243

obtido com p processadores. A Figura 6.13 mostra os *speedups* das implementações com a base de dados de 256, 2048, 8192 e 65536 transações.

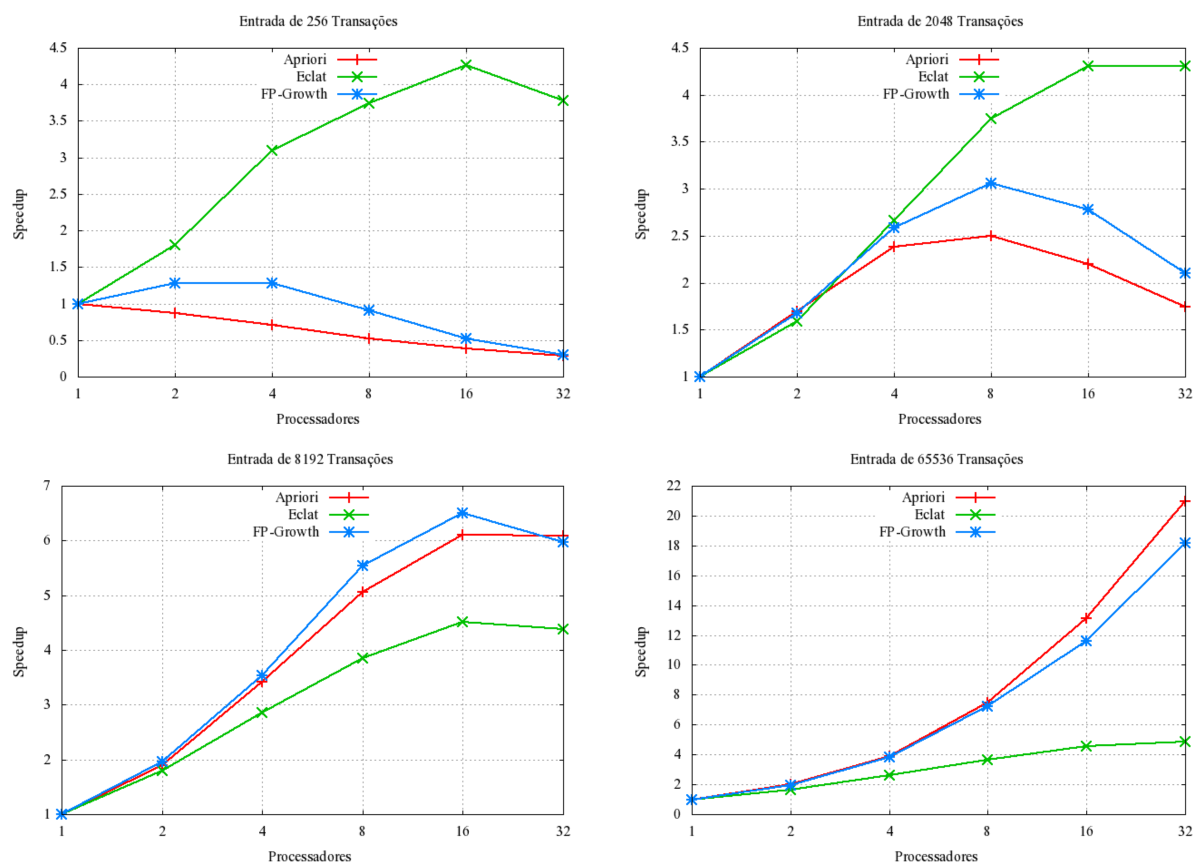


Figura 6.13: *Speedup* das implementações com 256, 2048, 8192 e 65536 transações.

No gráfico do *speedup* com 256 transações, observa-se que a implementação do Algoritmo Eclat é a que apresenta o maior desempenho. As implementações dos Algoritmos Apriori e *FP-Growth* não obtêm ganhos de desempenho no ambiente paralelo com essa base de dados, confirmando que, no processamento com pequenas quantidades de transações, os tempos de comunicação não são compensados pelos tempos de computação local.

Analisando os demais gráficos, percebe-se que o desempenho de todas implementações crescem conforme o tamanho da base de dados de entrada aumenta. A implementação que obtêm um maior crescimento no desempenho é a do Algoritmo Apriori, de modo que, quando processada com quantidades pequenas de transações, é a implementação que possui o menor desempenho, porém, conforme o tamanho da bases de dados de entrada aumenta, é a implementação que apresenta o maior ganho de desempenho. A implementação do Algoritmo Eclat é que possui o melhor desempenho quando processada com pequenas quantidades de transações, porém, à medida que o tamanho das bases de dados aumenta, é a implementação que apresenta o pior desempenho. Com relação à implementação do Algoritmo *FP-Growth*, observa-se que é a implementação que apresenta o desempenho mais estável, comparado ao desempenho das implementações dos Algoritmos Apriori e Eclat.

Capítulo 7

Conclusão

Nos últimos anos, empresas e organizações vêm buscando, cada vez mais, extrair conhecimentos a partir de volumes de dados que são acumulados ao longo do tempo. A busca pelo conhecimento tem ocorrido pela necessidade de mecanismos que possam auxiliar os gestores em suas tomadas de decisões. Com isso, técnicas de mineração de dados têm sido desenvolvidas no propósito de alcançar uma maior eficiência na extração dos conhecimentos. Dentre as técnicas mais empregadas na mineração de dados está a técnica de extração de regras de associação, devido a sua simplicidade de eficiência para a implementação. Atualmente, existem diversos algoritmos de mineração de dados que emprega a técnica de extração de regras de associação, dentre os mais conhecidos estão o Apriori, o Eclat e o *FP-Growth*.

Como a mineração de dados é aplicada geralmente em grandes volumes de dados, um maior poder computacional é requerido. E com a aplicação do paralelismo em diversos problemas computacionais, algoritmos para a extração de regras de associação no modelo paralelo têm sido desenvolvidos. Diante isso, o presente trabalho teve como objetivo realizar uma comparação entre os algoritmos Apriori, Eclat e *FP-Growth*, no modelo paralelo, afim de verificar qual o mais eficiente em relação a diferentes quantidades de dados processados, como também, em relação ao número de processadores utilizados no ambiente de execução.

Nos experimentos realizados, concluiu-se que, dentre os algoritmos paralelos Apriori, Eclat e *FP-Growth*, o algoritmo Apriori é o que requer maiores tempos de execução para qualquer volume de dados submetido em um processamento comparativo. Porém, o algoritmo Apriori é o que apresenta uma melhora mais acentuada no desempenho, conforme aumenta-se o número de processadores, quando é requerido um processamento com grandes quantidades de dados.

O algoritmo Eclat, por não utilizar comunicação entre os processadores envolvidos, é o algoritmo que apresenta uma maior redução no tempos de processamento, conforme aumenta-se o número de processadores, quando são utilizados volumes de dados considerados pequenos. Porém, também é o algoritmo que possui menor desempenho nos experimentos realizados com grandes quantidades de dados.

Já o algoritmo *FP-Growth* não apresenta reduções de tempo de processamento, com pequenas quantidades de dados, quando comparados aos tempos do algoritmo Eclat. Porém, entre os algoritmos, é o que obtém o menor tempo de processamento quando executado com grandes quantidades de dados. Com relação a ganhos de desempenho, o algoritmo *FP-Growth* apresenta resultados próximos aos obtidos pelo algoritmo Apriori com grandes volumes de dados.

A contribuição deste trabalho é apresentar, entre os algoritmos paralelos Apriori, Eclat e *FP-Growth*, qual o algoritmo mais adequado para ser utilizado de acordo com o volume de dados a ser processado. Como trabalhos futuros, pode-se: (*i*) pesquisar e implementar outros algoritmos paralelos de extração de regras de associação e comparar os resultados alcançados com os resultados do Apriori, Eclat e *FP-Growth*; (*ii*) realizar análises comparativas dos algoritmos Apriori, Eclat e *FP-Growth* utilizando diferentes bases de dados reais. (*iii*) Implementar os algoritmos Apriori, Eclat e *FP-Growth* utilizando o conceito de computação em nuvem ("*cloud computing*").

Referências Bibliográficas

[1] Microdados SAEB 2003. Manual do usuário. *Ministério da Educação*, INEP, 2011. [Acesso em 10 de Abril de 2011]. Disponível em: <http://portal.inep.gov.br/basica-levantamentos-acessar>.

[2] R. Agrawal, T. Imielinshi, 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, páginas 207–216. ACM, New York, NY, USA, 1993. ISBN 0-89791-592-5.

[3] R. Agrawal e J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, IEEE Educational Activities Department, Piscataway, NJ, USA, 1996. ISSN 1041-4347.

[4] R. Agrawal e R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, páginas 487–499. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994. ISBN 1-55860-153-8.

[5] E. Cáceres, H. Mongelli, e S. Song. Algoritmos paralelos usando cgm/pvm/mpi: Uma introdução. In *Anais do XXI Congresso da Sociedade Brasileira de Computação, Jornadas de Atualização de Informática*, volume 2, páginas 219–278. 2001.

[6] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6:298–307, 1994.

[7] J. Deogun, V. Raghavan, A. Sarkar, e H. Server. Data mining: Trends in research and development. In *Rough Sets and Data Mining: Analysis for Imprecise Data*, páginas 9–45. Kluwer Academic Publishers, 1996.

[8] M. Domingues e S. Rezende. Pós−processamento de regras de associação usando taxonomias. In *Infocomp Journal Of Computer Science*, volume 4, páginas 26–37. 2005. [Acesso em 14 de Julho de 2011]. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v4.2/art04.pdf>.

[9] R. Elmasri e S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 6° edição, 2010. ISBN-13 978-0136086208.

[10] U. Fayyad, G. Piatetsky-Shapiro, e P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, 1996.

[11] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, e R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press/The MIT Press, 1996. ISBN-13 978-0262560979.

[12] S. Götz. Communication-efficient parallel algoritms for minimum spanning-tree computation. *Tese de Doutoramento*, Universitat-Gesamthochschule Parderborn, 1998.

[13] E. Han, G. Karypis, e V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Engineering*, 12:337–352, IEEE Computer Society, Los Alamitos, CA, USA, 2000. ISSN 1041-4347.

[14] J. Han e M. Kamber. *Data Mining - Concepts and Techniques*. Morgan Kaufmann, 2° edição, 2005. ISBN-13 978-1558609013.

[15] 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, páginas 1–12. ACM, New York, NY, USA, 2000. ISBN 1-58113-217-4.

[16] J. Hipp, U. Güntzer, e G. Nakhaeizadeh. Algorithms for association rule mining a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, ACM, New York, NY, USA, 2000. ISSN 1931-0145.

[17] M. Houtsma e A. Swami. Set-oriented mining for association rules in relational databases. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, páginas 25–33. IEEE Computer Society, Washington, DC, USA, 1995. ISBN 0-8186-6910-1.

[18] M. Jannuzzi. *Indicadores Sociais no Brasil. Conceitos, Fontes de Dados e Aplicações*. Alínea Editora, 4° edição, 2009. ISBN-13 978-8575163689.

[19] V. Kumar, A. Grama, A. Gupta, e G. Karypis. *Introduction to Parallel Computing*. Addison Wesley, 2° edição, 2003. ISBN-13 978-0201648652.

[20] PC Cluster (HPCVL − High Performance Computing Virtual Laboratory). [Acesso em 03 de Maio de 2011]. Disponível em: <http://www.dehne.carleton.ca/research/lab-tour/hpcvl-cluster>.

[21] J. Li, Y. Liu, W. Liao, e A. Choudhary. Parallel data mining algorithms for association rules and clustering. In *International Conference on Computer Science and Software Engineering*. 2006.

[22] H. Mannila. Methods and problems in data mining. In *International Conference on Database Theory*, páginas 41–55. Dephi, Greece, 1997.

[23] E. Marchiori. Data mining. *Encyclopedia of Life Support Systems (EOLSS)*, 2000.

[24] S. Mitra e T. Acharya. *Data Mining: Multimidia, Soft Computing, and Bioinformatics*. Wiley-Interscience, 1° edição, 2003. ISBN-13 978-0471460541.

[25] H. Mongelli e R. Terada. Algoritmos paralelos para solução de problemas lineares. In *Relatório Técnico RT-MAC-9506, Instituto de Matemática e Estatística da Universidade de São Paulo*. São Paulo, SP, Brasil, 1995.

[26] S. Orlando, P. Palmerini, e R. Perego. Enhancing the apriori algorithm for frequent set counting. In *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '01, páginas 71–82. Springer-Verlag, London, UK, 2001. ISBN 3-540-42553-5.

[27] A. Saverese, E. Omiecinski, e S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, páginas 432–444. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. ISBN 1-55860-379-4.

[28] M. Turine, M. Machado, M. Santos, e R. Gondim. Videweb: Um ambiente para visualização de dados educacionais na plataforma web-pide. 2007.

[29] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, ACM, New York, NY, USA, 1990. ISSN 0001-0782.

[30] A. Veloso, B. Coutinho, G. Menezes, B. Pôssas, W. Meira, M. Carvalho, e C. Amorim. Mineração assíncrona de regras de associação em sistemas de memória compartilhada-distribuída. In *Anais do 2° Workshop em Sistemas Computacionais de Alto Desempenho (SBAC-PAD)*, páginas 9–16. Pirenópolis, GO, Brasil, 2001.

[31] I. Witten e E. Frank. *Data Mining Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2° edição, 2005. ISBN-13 978-0120884070.

[32] M. Wojciechowski e M. Zakrzewicz. On efficiency of dataset filtering implementations in constraint-based discovery of frequent itemset. In *JCKBSE Conference*. Maribor, Slovenia, 2002.

[33] M. Zaki, S. Parthasarathy, e W. Li. A localized algorithm for parallel association mining. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, páginas 321–330. ACM, New York, NY, USA, 1997. ISBN 0-89791-890-8.