UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

FCT – Faculdade de Ciências e Tecnologia

DMEC – Departamento de Matemática, Estatística e Computação

Bacharelado em Ciência da Computação

Rafael Silva e Santos

Aplicação do Algoritmo *Rough Sets* em um *Data Warehouse*

Monografia para conclusão de curso

Orientador: Prof. Dr. Milton Hirokazu Shimabukuro

Presidente Prudente

2011

Termo de Aprovação

Rafael Silva e Santos

Aplicação do Algoritmo *Rough Sets* em um *Data Warehouse*

Monografia sob o título "Aplicação do Algoritmo *Rough Sets* em um *Data Warehouse*", defendida por Rafael Silva e Santos e aprovada em 15 de Dezembro de 2011, em Presidente Prudente, Estado de São Paulo, pela banca examinadora constituída pelos doutores:

____

Prof. Dr. Messias Meneguette Junior

Universidade Estadual Paulista "Júlio de Mesquita Filho"

____

Prof. Dr. Marco Antônio Piteri

Universidade Estadual Paulista "Júlio de Mesquita Filho"

____

Prof. Dr. Milton Hirokazu Shimabukuro

Universidade Estadual Paulista "Júlio de Mesquita Filho"

Presidente Prudente

2011

Agradecimentos

Manifesto minha gratidão:

A Deus, que sempre esteve ao meu lado em todas as tarefas e desafios ao longo do caminho.

Ao meu pai, Silvio Antônio dos Santos, que vive em memória, e sua ausência se faz presente todos os dias. Sem sua colaboração, a conclusão do curso não seria possível.

Aos meus familiares, amigos e docentes, por toda a força que me deram durante esse tempo.

Resumo

O projeto tem como objetivo desenvolver métodos de classificação dentro de um *Data Warehouse* para o auxílio de tomada de decisões.

Também temos como objetivo a redução de um grupo de atributos dentro de um *Data Warehouse*, mantendo as mesmas características do conjunto original.

Com essa redução, temos um menor custo computacional ao processar os dados, podemos identificar atributos que não são relevantes para certos tipos de situações, e por fim, somos capazes de reconhecer padrões na base de dados que ajudarão nas tomadas de decisões.

Para o alcance desses dois objetivos principais, foi implementado o algoritmo de *Rough Sets*.

Foi definido que o sistema gerenciador de banco de dados utilizado no projeto será o PostgreSQL, pois se trata de um sistema eficiente, consolidado, e por fim, é um sistema *open-source*, ou seja, de distribuição livre.

Palavras-chave: Data Warehouse. Rough Sets.Redução do sistema.

Abstract

This Project aims to develop methods for data classification in a Data Warehouse for decision-making purposes.

We also have as another goal the reduction of an attribute set in a Data Warehouse, in which a given reduced set is capable of keeping the same properties of the original one.

Once we achieve a reduced set, we have a smaller computational cost of processing, we are able to identify non-relevant attributes to certain kinds of situations, and finally we are also able to recognize patterns in the database that will help us to take decisions.

In order to achieve these main objectives, it will be implemented the Rough Sets algorithm.

We chose PostgreSQL as our data base management system due to its efficiency, consolidation and finally, it's an open-source system (free distribution).

Keywords: Data Warehouse. Rough Sets. Attributes Reduction.

SUMÁRIO

1. **Introdução..11**
1.1. Objetivos do trabalho..12
1.2. Organização do trabalho...12
2. **Fundamentação Teórica...13**
2.1. Considerações iniciais...13
2.2. *Data Warehouse*..15
2.2.1. OLTP x OLAP..16
2.2.2. Características de um DW...17
2.2.3. Tecnologias de um DW..18
2.2.4. Arquitetura de um DW...19
2.2.5. Modelagem de dados dimensional...20
2.2.6. Operadores OLAP...24
2.2.7. Projeto e implementação de um DW......................................27
2.2.7.1. Desenvolvimento de um DW..28
2.2.7.2. Arquitetura global do DW...28
2.2.7.3. Ferramentas ETC...31
2.3. Granularidade de dados...35
2.4. *Data Warehouse* usando o PostgreSQL.......................................35
2.4.1. PostgreSQL...35
2.4.2. Características do PostgreSQL que favorecem um *Data Warehouse*...36
2.5. *Rough Sets*...37
2.5.1. Idéia do algoritmo de *Rough Sets*...37
2.5.2. Sistemas de informação..38
2.5.3. Relação de indiscernibilidade...39
2.5.4. Aproximação de conjuntos...40
2.5.5. Qualidade de aproximações..41
2.5.6. Redução do sistema de informação e geração de regras......43
2.6. Método de Quine-McCluskey...44
2.7. Tecnologia JAVA..48

3. Implementação de *Rough Sets*..49

3.1. Considerações iniciais..49

3.2. Organização da implementação...49

3.3. Definição dos Sistemas de informações....................................50

3.3.1. Sistemas de informações..50

3.3.2. Atributos condicionais...51

3.3.3. Atributos de decisão...53

3.3.4. Requisitos do programa para os Sistemas de informações..53

3.4. *Rough Sets*..55

3.4.1. Classes de equivalência...55

3.4.1.1. Definição das classes e das relações de indiscernibilidade...55

3.4.2. Aproximação de conjuntos..58

3.4.2.1. Processamento do fato..58

3.4.2.2. Aproximações de conjuntos..58

3.4.2.3. Qualidade de aproximações.......................................62

3.4.3. Matriz de indiscernibilidade...62

3.4.4. Redução do sistema..65

3.4.5. Geração de Regras...74

3.4.6. Base de dados..75

3.4.7. Quantidade de atributos condicionais....................................75

4. Conclusão...77

4.1. Trabalhos Futuros...78

4.1.1. Faixa de valores dos atributos condicionais..........................78

4.1.2. Níveis de hierarquia das dimensões na escolha dos atributos condicionais...78

4.1.3. Simulação dos atributos de decisão......................................79

4.1.4. Restrição no armazenamento da matriz de discernibilidade.79

Referências Bibliográficas...81

Apêndice A – Definição da tabela das classes de equivalência............83
Apêndice B – Definir classes de equivalência...86
Apêndice C – Aproximações de conjuntos...87
Apêndice D – Matriz de discernibilidade..93
Apêndice E – Função de discernibilidade..98
Apêndice F – Encontrar redutos...100

Lista de Figuras

1. Sistema de suporte a decisão........13
2. Arquitetura de um *Data Warehouse*........20
3. Exemplo de uma modelagem multidimensional........22
4. Modelo estrela........23
5. Modelo floco de neve........24
6. Operação S*lice*........25
7. Operação S*lice* e *Dice*........25
8. Operação *Drill-down*........26
9. Operação *Roll-Up*........26
10. Operação *Pivot*........27
11. *Data Warehouse* centralizado........29
12. *Data Warehouse* descentralizado........30
13. DM lógicos........31
14. Captura de dados........32
15. Filtragem........33
16. Transformação........34
17. Carregamento e indexação........34
18. Combinação de minitermos........46

Lista de Tabelas

1. Sistema de suporte a decisão..15
2. Exemplo de um Sistema de Informação..39
3. Minitermos agrupados..46
4. Tabela de Quine-McCluskey..47

Capítulo 1

Introdução

Com o crescimento da automação dos processos empresariais, surge a necessidade de se ter um conjunto de tecnologias e ferramentas para a extração de informações de uma base de dados. As corporações veem hoje a necessidade de tomar decisões cada vez mais acertadas para os negócios, para possibilitar o aumento da lucratividade e da competitividade com os concorrentes, e para ter um conhecimento maior dos seus clientes (Machado, 2000).

Há a necessidade de um ambiente voltado à análise de dados e a obtenção de informações de modo sistemático e eficiente, para que as necessidades executivas do cliente, ou seja, ferramentas necessárias para a tomada de decisões sejam supridas, trazendo um benefício maior para o meio empresarial.

O *Data Warehouse* visa esse ambiente para consultas e análise de dados. Este ambiente utiliza uma modelagem multidimensional, e ferramentas OLAP (*On Line Analytical Processing*) para tratar as informações de maneira eficiente.

Em um ambiente com um grande volume de dados, é possível que em certos tipos de consultas, tenhamos um conjunto de atributos que são irrelevantes em seu contexto, ou seja, temos o mesmo resultado quando eliminamos esses atributos do nosso sistema de informação.

Embora o tempo de resposta de consultas realizadas em um *Data Warehouse* não seja um fator crucial, é possível diminuí-lo, usando somente atributos relevantes ao caso. Nesse contexto, entra a teoria de *Rough Sets*.

1.1. Objetivo do trabalho

O objetivo do projeto consiste em implementar o algoritmo de *Rough Sets* e aplicá-lo em um *Data Warehouse* para a classificação de dados, a otimização de consultas visando a utilização de atributos relevantes no contexto, e por fim a geração de regras e reconhecimento de padrões dentro do *Data Warehouse*.

1.2. Organização do trabalho

O Capítulo 2 contém toda a fundamentação teórica necessária para o entendimento de conceitos que estão presentes no projeto. Serão descritos conceitos e explicações sobre o *Data Warehouse,* suas características, tecnologias envolvidas, arquitetura e modelagens. O Capítulo 2 também descreve o algoritmo de *Rough Sets,* que será responsável por processar os dados contidos no *Data Warehouse,* classificá-los e por fim, reduzir o conjunto de informações, se possível. Por fim, temos a descrição de outro algoritmo, chamado de *Quine-McCluskey,* que será necessário para a redução do conjunto de informações.

O Capítulo 3 destina-se a descrever todo o desenvolvimento do projeto, desde a formação de parâmetros e informações necessárias do *Data Warehouse* até os passos necessários para a aplicação do algoritmo *Rough Sets.*

O Capítulo 4 discute sobre as restrições encontradas ao longo do projeto.

Por fim, no Capítulo 5 temos a conclusão do projeto, descrevendo os resultados e possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1. Considerações iniciais

Hoje a informação possui um imenso valor em vários contextos, como no setor empresarial no qual uma empresa deve tê-la em mãos para analisar os perfis de seus clientes, analisar o mercado e também seus concorrentes, ou no setor público, para proporcionar decisões mais eficientes com relação a suas tarefas e ao seu público-alvo.

Enfim, essas informações devem ser utilizadas com o objetivo de fornecer um suporte à tomada de decisões, que tem seu foco em monitorar e comparar situações atuais com passadas e estimar situações futuras (Machado, 2000).

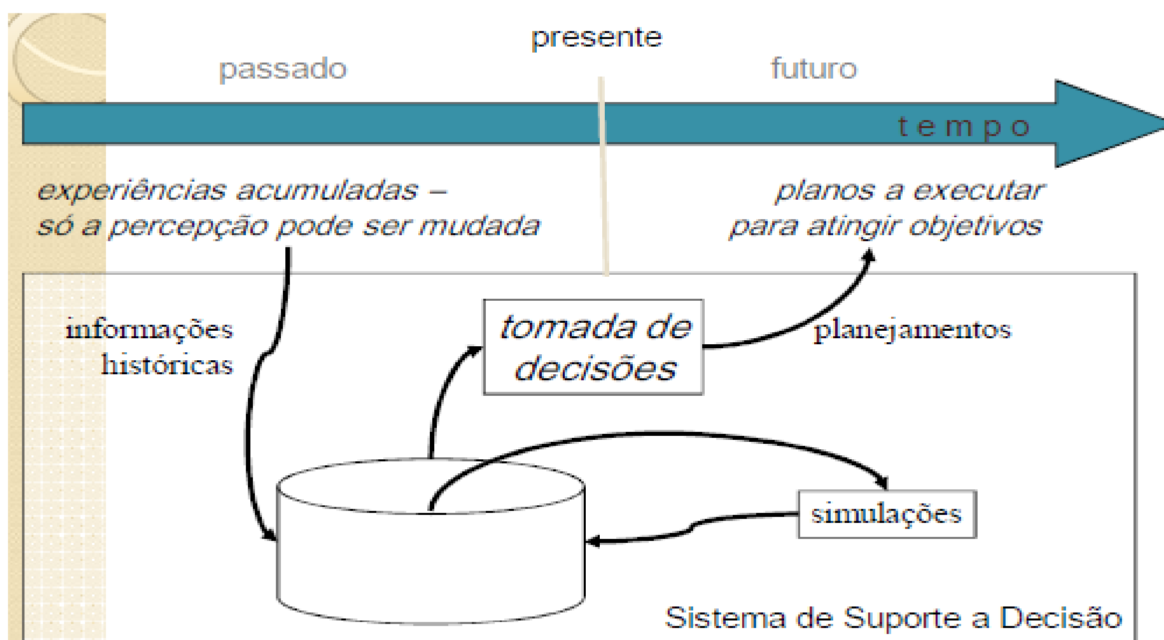


Figura 1. Sistema de suporte a decisão.

Fonte: Retirada de Fileto (2009).

A Figura 1 representa um esquema básico de um sistema de suporte a decisão. Tem-se uma linha do tempo, no qual no passado temos experiências acumuladas que servirão de parâmetros de entrada para o sistema. Essas informações são processadas, e com os dados da saída do sistema é possível ter uma visão mais ampla do problema, e tomar decisões mais eficientes para formular planos para atingir os objetivos.

Desde o começo do desenvolvimento de sistemas para o controle operacional de uma empresa, esses sistemas são feitos de maneira transacional, dando ênfase somente ao armazenamento e manipulação de dados, tendo seu direcionamento para automação de processos.

Porém existe pouca atenção a respeito da informação por diversos motivos. Um deles é que existem dificuldades em se diferenciar informação e dado, podendo acarretar problemas de modelagem e especificações do sistema. Outro motivo é o tratamento de informações em uma base transacional de dados, pois as informações são resultados de uma análise de um enorme volume de dados contidos em uma ou várias bases transacionais, e isso é uma tarefa extremamente cara (Machado, 2000).

Um sistema transacional, ou sistema *On Line Transaction Processing* (OLTP), é uma melhor opção quando é necessário controlar informações operacionais, como por exemplo, vendas, compras, contabilidade, sensoriamento e sistemas de tempo real.

Em um sistema transacional, não é visado um longo histórico de informações; este geralmente abrange de 60 a 90 dias.

Uma melhor opção para um sistema de suporte de decisão seria um sistema especialmente modelado para proporcionar um ambiente que possa tratar as informações de maneira mais eficiente, realizando consultas complexas de maneira mais rápida, pois se tem em mãos um grande volume de dados a serem analisadas, que representam o histórico de uma instituição.

A Tabela 1 mostra as principais diferenças entre um sistema transacional e um sistema de suporte a decisão, ilustrando a diferença entre os focos dos dois tipos de sistemas.

Tabela 1. Sistema de suporte a decisão.

Fonte: retirada de Fileto (2009).

Característica	**BD Transacional**	**BD Suporte à Decisão**
Objetivo	Atividades cotidianas	Análise do negócio
Uso	Operacional	Informativo
Processamento	OLTP	OLAP
Unidade de trabalho	Inclusão, alteração e exclusão.	Carga e consulta
Usuários	Operadores (muitos)	Gerência (poucos)
Interação dos usuários	Ações pré-definidas	Pré-definida e *ad-hoc*
Dados	Operacionais	Analíticos
Volume	(MB - GB)	(GB – TB)
Histórico	60 a 90 dias	Vários anos
Granularidade	Detalhada (baixa)	Detalhada e consolidada (alta)
Redundância	Não pode ocorrer	Pode ocorrer
Estrutura	Estática	Variável
Manutenção	Mínima	Periódica
Atualização	Contínua	Periódica

2.2. *Data Warehouse*

Um *Data Warehouse* (DW) visa esse ambiente para consultas e análise de dados. Este ambiente utiliza uma modelagem multidimensional e ferramentas OLAP (*On Line Analytical Processing*) para tratar as informações de maneira eficiente. Os dados utilizados nessa modelagem multidimensional podem ser derivados de vários sistemas transacionais.

Esse ambiente pode ser visto como um grande armazém que possui informações históricas sobre alguma instituição.

Combinando este ambiente com ferramentas OLAP, é possível extrair de forma eficiente informações importantes para tomada de decisões.

2.2.1. OLTP X OLAP

Seguem algumas características em que se diferenciam essas duas tecnologias, conforme cita Machado (2000):

Integração dos dados: Os dados contidos em sistemas transacionais geralmente não estão integrados, pois cada um deles possui um próprio padrão de definições;

Organização de dados: Em um ambiente de DW, a modelagem utilizada é a modelagem multidimensional, que busca atingir um melhor desempenho nas consultas;

Atualização de dados: Não existe sobreposição de registros provenientes do sistema transacional no DW. Se ocorrer uma mudança, um novo registro é inserido no DW, associando ao mesmo um elemento de tempo;

Ciclo de vida do desenvolvimento: O ciclo de vida do DW começa pelos dados. Somente quando é verificado que eles estão íntegros é que eles são inseridos no DW;

Uma máquina ou duas: Um DW requer uma expressiva quantidade de recursos, e geralmente é implantado em uma máquina separada do sistema OLTP;

Tempo de resposta: O tempo de resposta em um sistema transacional pode ser um fator crítico, enquanto em um DW, este tempo pode ser medido

em minutos, horas e até dias. Porém, isso não significa que o tempo no DW não seja importante, ele está ligado diretamente à produtividade;

Metadados: Em se tratando de um sistema transacional, os metadados são importantes para os desenvolvedores do sistema e o DBA (*Data Base Administrator*).

Os bancos de dados transacionais são usados por uma aplicação, podendo ter essas informações dentro da aplicação.

Em DW, os analistas de dados e os executivos procuram por fatos não usuais e correlações que serão conhecidas quando encontradas.

Os usuários finais de um DW devem ter amplo conhecimento do negócio; geralmente são as pessoas que pertencem a camadas gerenciais.

2.2.2. Características de um DW

Seguem algumas características importantes de um *Data Warehouse,* conforme cita Fileto (2009):

Orientado a assuntos: O DW agrupa as informações mais importantes para as tomadas de decisão, diferente de um sistema transacional, que mantém as informações das transações diárias;

Integrado: O DW pode ser derivado de várias bases transacionais, que pode ter diferentes tipos de nomenclaturas, formatos e estruturas. Então é necessário relacionar todos esses conceitos, para que o *Data Warehouse* seja consistente;

Séries temporais: As informações contidas em um DW representam uma *"fotografia"* do estado atual das fontes de dados, ou seja, são informações

que são retiradas das fontes periodicamente, diferentemente dos sistemas transacionais, que são atualizados a cada transação;

Não volátil: Um DW possui duas operações básicas: a carga dos dados e a consulta. Ou seja, não existe exclusão em um DW, a não ser que seja necessário realizar correções.

2.2.3. Tecnologias de um DW

Tecnologias incorporadas em um *Data Warehouse,* conforme cita Fileto (2009):

Ferramentas de ETC (Extração, Transformação e Carga): Essas ferramentas são a ponte entre os bancos de dados transacionais e o DW. Elas são responsáveis pela extração dos dados, realizando conversões, validações, correções e integração dos dados, e por fim, inserindo-os no DW. Mais detalhes na Seção 2.2.7.3;

Modelagem de dados multidimensional: Uma modelagem chamada multidimensional é apropriada para sumarizar e reestruturar dados, e apresentá-los em várias visões diferentes. Essa modelagem possui três elementos básicos como conceitos: fatos (coleção de itens de dados), dimensões (elementos de um fato, determinando o contexto) e por fim as medidas (variáveis). É comum usar um cubo para fazer a representação deste modelo. Mais detalhes na Seção 2.2.5;

Ferramentas de prospecção e análise de dados baseadas em *On-Line Analytical Processing* (OLAP);

Ferramentas de administração e manutenção do DW e os *Datamarts* (DM);

Os DMs são um subconjunto do DW, permitindo que os usuários acessem os dados de modo descentralizado e direcionando os dados a uma área específica. Usando uma analogia, podemos ver o DW como um grande armazém, onde os DMs são as prateleiras, tendo o propósito de uma melhor organização do armazém.

2.2.4. Arquitetura de um DW

A Figura 2 representa a arquitetura de um DW. Na parte inferior da figura temos as fontes de dados, das quais são tiradas as "fotografias" para a atualização do DW. Acima temos as ferramentas ETC para fazer a transição dos dados para o DW. Depois temos o DW, o catálogo de metadados e uma interface com o administrador do DW, que é responsável pela sua manutenção. As consultas no DW são feitas pelos Clientes, e é importante saber que esses clientes são pessoas que possuem um conhecimento avançado sobre o negócio, tendo competência suficiente para analisar os dados consultados e usá-los para planejamentos. Esses clientes realizam a pesquisa nos DMs.

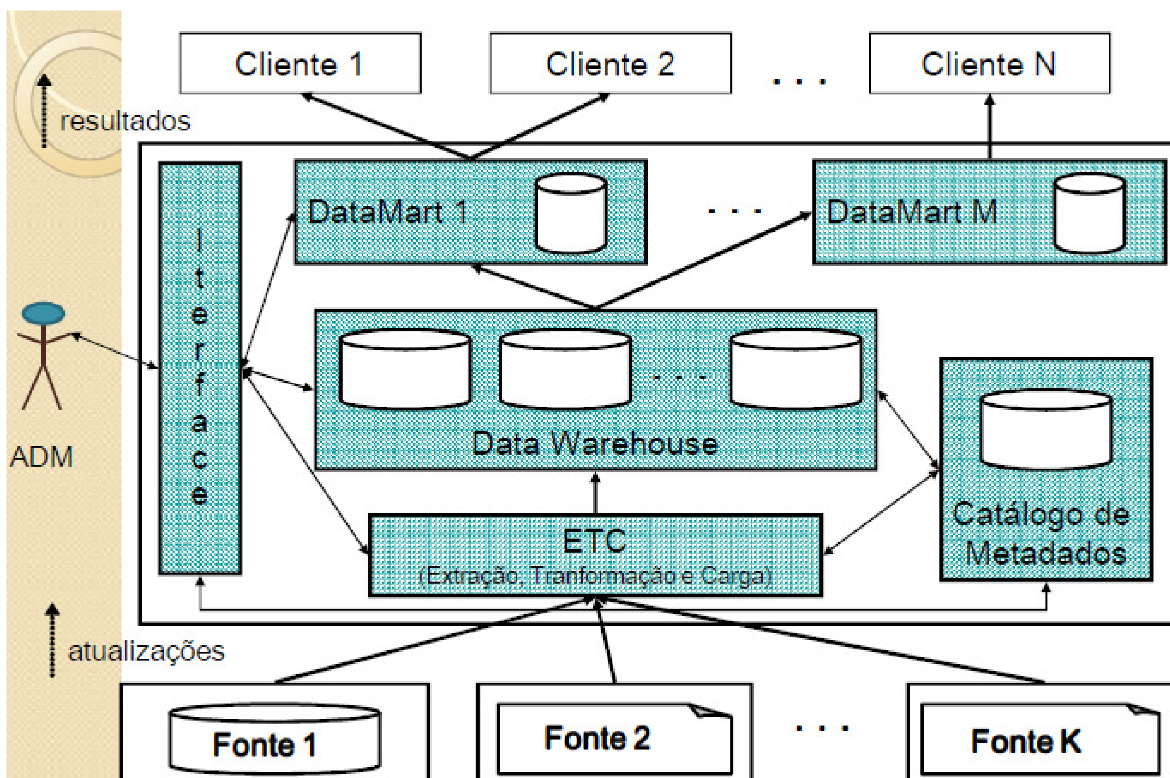


Figura 2. Arquitetura de um *Data Warehouse*

Fonte: retirada de Fileto (2009).

2.2.5. Modelagem de dados multidimensional

No esquema do DW, temos os fatos, as dimensões e as medidas.

Um **fato** é um item de uma perspectiva de um problema, um assunto do negócio que pode ser representado por valores numéricos. Eles que representam os dados quantitativos, ou seja, os registros das medidas.

Fatos possuem um caráter evolutivo, ou seja, seus valores mudam ao longo do tempo.

Outra característica de um fato é que é possível manter dados históricos com o passar do tempo.

Por exemplo: “O índice de aprovação de alunos vindos de escolas públicas em faculdades públicas”. O fato dessa frase é o índice de aprovação,

pois ele tem um valor numérico, tem um caráter evolutivo e podemos manter dados históricos sobre ele.

Um fato é implementado em uma tabela do *Data Warehouse* chamada de tabela fato.

Uma **dimensão** determina um contexto de um assunto, tendo um valor descritivo e classificatório. É basicamente um elemento que participa de um fato. Ela pode conter muitos membros, que são usados para classificar um fato. Por exemplo, temos em uma dimensão geográfica, temos membros como cidades, estados, regiões.

Geralmente esses membros são descritos com uma regra: Primeiro toma-se o fato, e depois tiramos as seguintes classificações: “onde”, “quando”, “o que” e “quem”.

Por exemplo, em um fato Produção: “Onde” significa que a produção será feita em uma máquina. “quando” refere-se à data da produção. “Quem” refere-se ao operador da máquina, e por fim, “o que” refere-se ao produto que será produzido.

Os membros de uma dimensão podem ser classificados em hierarquias. Por exemplo, em uma dimensão de tempo, temos no topo da hierarquia o ano, depois temos trimestres, depois os meses, depois os dias, etc.

Uma **medida** representa um atributo numérico que representa um fato. Elas são determinadas fazendo combinações das dimensões que participam de um fato, e estão localizadas como atributos de um fato (Machado, 2000).

Ela pode ser classificada em valores aditivos e não aditivos.

Os valores aditivos são os que podem se realizar operações de soma, subtração e média. Por exemplo: “números de crimes registrados”, ou “número de transplantes realizados”.

Os valores não aditivos não podem se realizar qualquer tipo de operação. Eles representam um desempenho de algum fato no tempo, obtidos de valores aditivos.

O modelo multidimensional é especifico para o processamento analítico da informação. As medidas são organizadas segundo dimensões e hierarquias de níveis.

A Figura 3 é um exemplo de uma modelagem multidimensional utilizando um cubo que representa o fato Vendas. Temos representadas três dimensões, ou dados qualitativos, que são o local, o produto e o tempo. A parte branca do cubo representa as medidas. Por fim, a parte colorida do cubo representa as operações sobre as medidas..

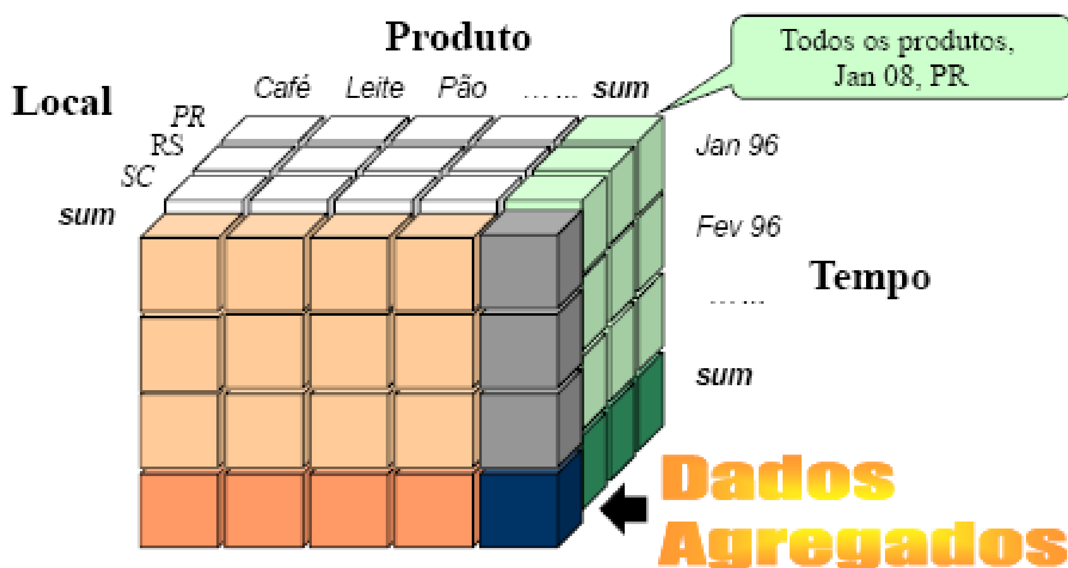


Figura 3. Exemplo de uma modelagem multidimensional.
Fonte: retirada de Fileto (2009).

Os DW podem ter dois tipos diferentes de modelagem: o *Star* (modelo em formato de estrela) e o *Snowflake* (formato de floco de neve).

No modelo *Star*, as instâncias são armazenadas em uma tabela contendo o identificador de instância, valores das dimensões descritivas para casa instância e as medidas para aquela instância (tabela de fatos). Além disso, pelo menos uma tabela é usada para cada dimensão, para armazenar dados sobre a dimensão. Este esquema é chamado de *Star* por apresentar uma tabela-fato dominante no centro do esquema, e as tabelas de dimensões

nas extremidades. A tabela-fato é ligada as demais tabelas por múltiplas junções, enquanto as tabelas de dimensões se ligam a tabela central por uma única junção. A tabela-fato é onde as medidas numéricas do fato representado são armazenadas. Cada uma destas medidas é tomada segundo uma interseção de todas as dimensões (Mussi, 2010).

A Figura 4 mostra o esquema Estrela, no qual no centro se encontra a tabela-fato (*Sales)*, e nas extremidades as tabelas de dimensões (*Item, Time, Location*e *Branch).* As medidas estão na tabela-fato, que são *Units_Sold, Dollars_Sold e avg_sales.*

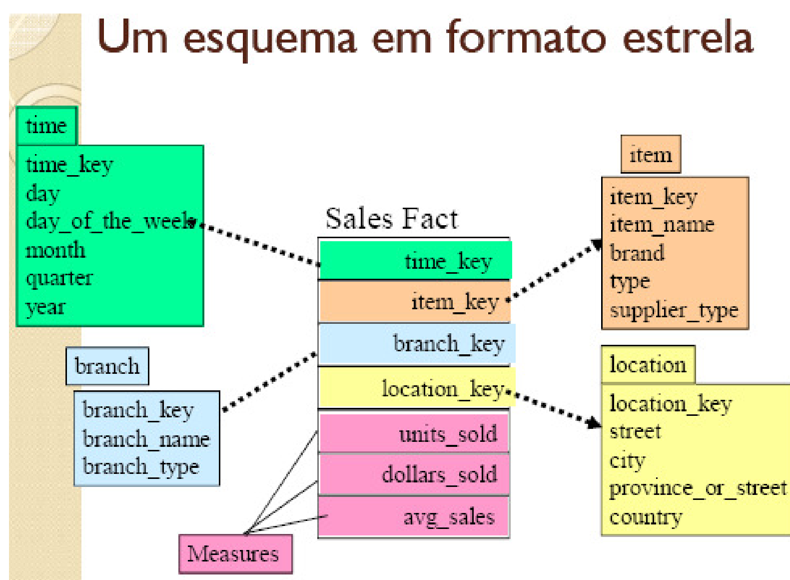


Figura 4. Modelo estrela.

Fonte: retirada de Fileto (2009).

O **modelo Floco de neve** consiste de uma extensão do modelo estrela, no qual cada uma das pontas da estrela passa a ser o centro de outras estrelas. Cada tabela de dimensão é normalizada, quebrando-se a tabela original ao longo de hierarquias existentes em seus atributos. (Mussi, 2010)

A Figura 5 mostra o esquema *Snowflake.* As tabelas de dimensões *Item* e *Location* são "quebradas", formando outras duas tabelas.

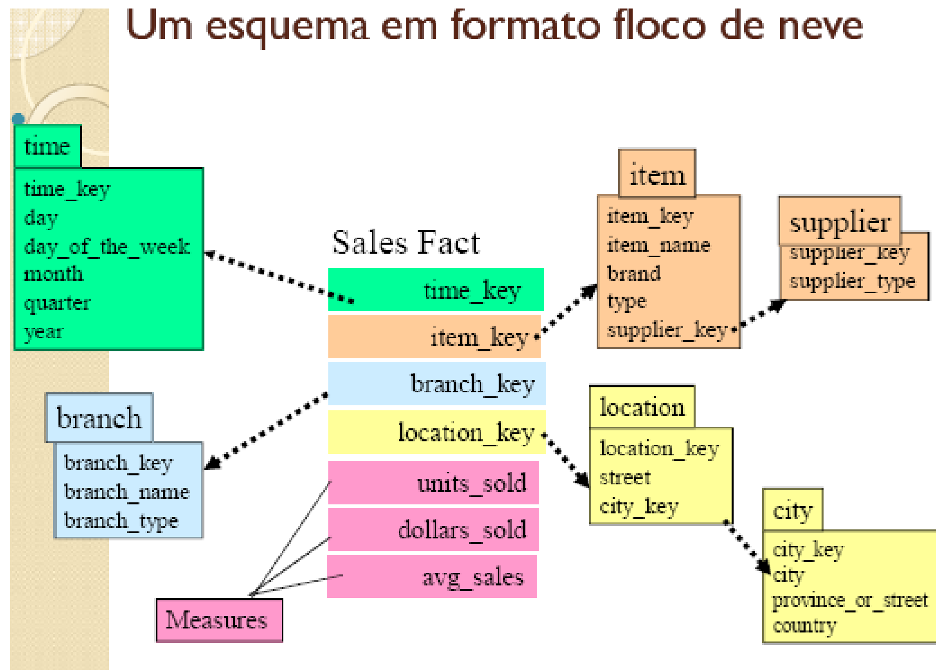


Figura 5. Modelo floco de neve.

Fonte: retirada de Fileto (2009).

2.2.6. Operadores OLAP

São os operadores que auxiliam nas consultas do DW, permitindo ao cliente visualizar os dados sob várias perspectivas:

Slice: projeta dados de uma determinada dimensão. Simplificadamente, esse operador extrai uma "fatia" do hipercubo.

A Figura 6 mostra um exemplo do operador *Slice.* O Produto selecionado foi o sapato (*shoes)*; o resultado mostra todas as medidas (variáveis) da operação nos meses determinados na dimensão meses (*months)*.

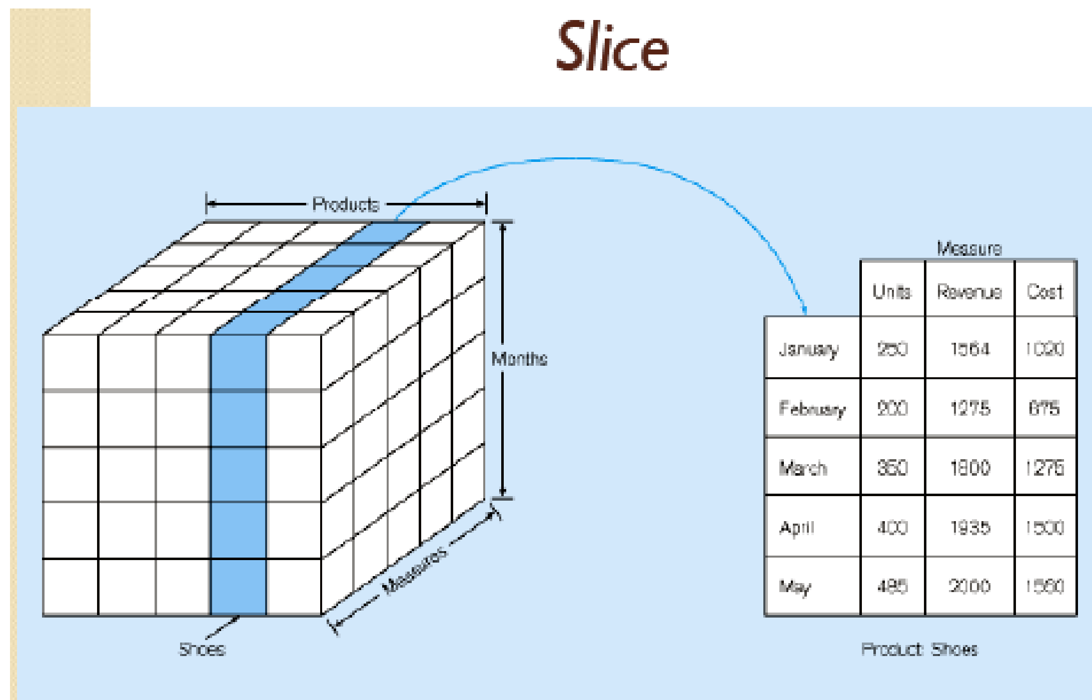


Figura 6. Operação S*lice*.

Fonte: retirada de Fileto (2009).

Dice: Retorna *Slices* consecutivos. Simplificadamente extrai um hipercubo menor. A Figura 7 mostra o uso do operador *dice*, no qual são selecionados candidatos de três diferentes estados (Paraná, Santa Catarina e São Paulo). Então, ao invés de somente "fatia" da dimensão, foram extraídas três "fatias".

		Measures
Curso	Candidato	num
+All cursos	-All Candidatos	318,468
	+Bahia	324
	+Mato Grosso	1,548
	+Minas Gerais	1,704
	+Parana	17,208
	+Rio de Janeiro	552
	+Santa Catarina	250,128
	+Sao Paulo	29,484

	Measures
Candidato	num
-All Candidatos	318,468
+Parana	17,208
+Santa Catarina	250,128
+Sao Paulo	29,484

Slice and Dice

Figura 7. Operação S*lice* e *Dice*.

Fonte: retirada de Fileto (2009).

***Roll-down* (ou *drill-down*):** Detalha os dados, descendo na hierarquia de uma dimensão. A Figura 8 representa uma operação *drill-down.*

	Measures	
Candidato	maxima	num
+All Candidatos	10.00	318,468

Drill-Down

	Measures	
Candidato	maxima	num
-All Candidatos	10.00	318,468
+Amapa	6.75	12
+Bahia	9.49	324
+Goias	9.80	1,248
+Mato Grosso	10.00	1,548
+Mato Grosso do Sul	9.80	2,256
+Minas Gerais	9.83	1,704
+Parana	10.00	17,208
+Rio de Janeiro	9.50	552
+Rio Grande do Sul	9.67	10,752
+Santa Catarina	10.00	250,128
+Sao Paulo	10.00	29,484

Figura 8. Operação *Drill-down*.

Fonte: retirada de Fileto (2009).

***Roll-up* (ou *drill-up*):** Sumarizam dados, subindo um nível na hierarquia. É o inverso do *Roll-down.* A Figura 9 representa uma operação *drill-up.*

	Measures	
Candidato	maxima	num
+All Candidatos	10.00	318,468

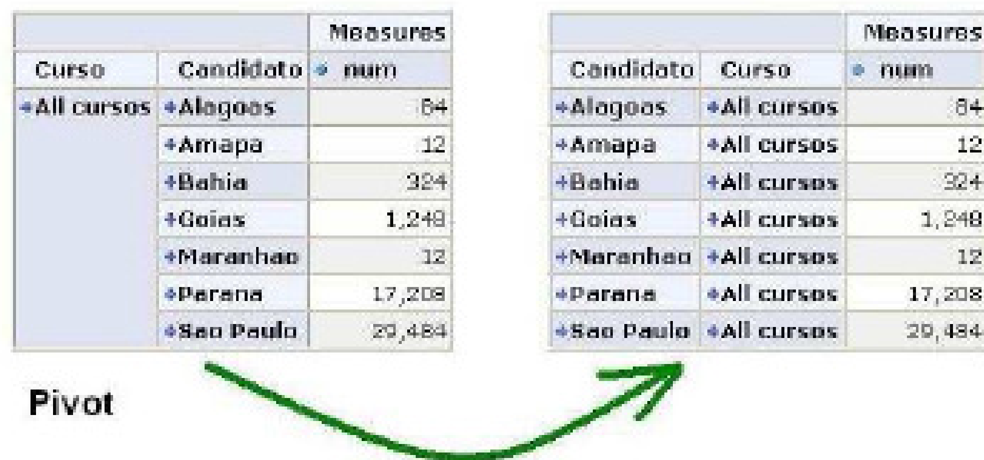
Roll Up

	Measures	
Candidato	maxima	num
-All Candidatos	10.00	318,468
+Amapa	6.75	12
+Bahia	9.49	324
+Goias	9.80	1,248
+Mato Grosso	10.00	1,548
+Mato Grosso do Sul	9.80	2,256
+Minas Gerais	9.83	1,704
+Parana	10.00	17,208
+Rio de Janeiro	9.50	552
+Rio Grande do Sul	9.67	10,752
+Santa Catarina	10.00	250,128
+Sao Paulo	10.00	29,484

Figura 9. Operação *Roll-Up*.

Fonte: retirada de Fileto (2009).

Pivot: Muda a posição das dimensões no hipercubo, fazendo um rearranjo nas tabelas. A Figura 10 representa uma operação *pivot.*



		Measures
Curso	Candidato	num
All cursos	Alagoas	84
	Amapa	12
	Bahia	324
	Goias	1,248
	Maranhao	12
	Parana	17,208
	Sao Paulo	29,484

		Measures
Candidato	Curso	num
Alagoas	All cursos	84
Amapa	All cursos	12
Bahia	All cursos	324
Goias	All cursos	1,248
Maranhao	All cursos	12
Parana	All cursos	17,208
Sao Paulo	All cursos	29,484

Figura 10. Operação *Pivot.*

Fonte: retirada de Fileto (2009).

2.2.7. Projeto e Implementação de um DW

Um DW geralmente possui um alto custo para ser implantado, tanto de dinheiro quanto de tempo, o que dificulta sua adoção em instituições de pequeno e médio porte. Outra dificuldade é que os requisitos para um DW não são conhecidos até que ele esteja parcialmente carregado ou já em uso.

Outra questão é a adequação do modelo entidade-relacionamento ao DW. Esta não é uma opção boa, pois o DW e o modelo E-R(1) possuem objetivos diferentes. O principal objetivo do modelo E-R é a eliminação de redundâncias, realizando fragmentações dos dados em diversas tabelas, o que acarreta em uma maior complexidade de consultas ao usuário final. E um DW tem por objetivo estruturas mais simples e com menor grau de normalização para agilizar as consultas.

(1) Modelo E-R: modelo baseado na percepção do mundo real, que consiste em um conjunto de objetos básicos chamados entidades e nos relacionamentos entre esses objetos.

2.2.7.1. Desenvolvimento do DW

O desenvolvimento de um DW passa basicamente pelas seguintes fases, conforme cita Fileto (2009):

- Planejamento
- Levantamento das necessidades e das fontes de dados
- Integração dos dados
- Modelagem dimensional
- Projeto físico do Banco de dados
- Projeto das transformações dos dados (Ferramentas ETC)
- Desenvolvimento das aplicações
- Validação e Teste
- Treinamento
- Implantação

2.2.7.2. Arquitetura global do DW

A arquitetura global pode ser fisicamente centralizada ou fisicamente distribuída nas instalações da empresa, conforme cita Fileto (2009).

A centralização física é utilizada quando uma empresa existe em um único local e o DW é administrado por um departamento de tecnologia de informação. Esse tipo de arquitetura requer um alto investimento em um servidor de alta capacidade de processamento e armazenamento.

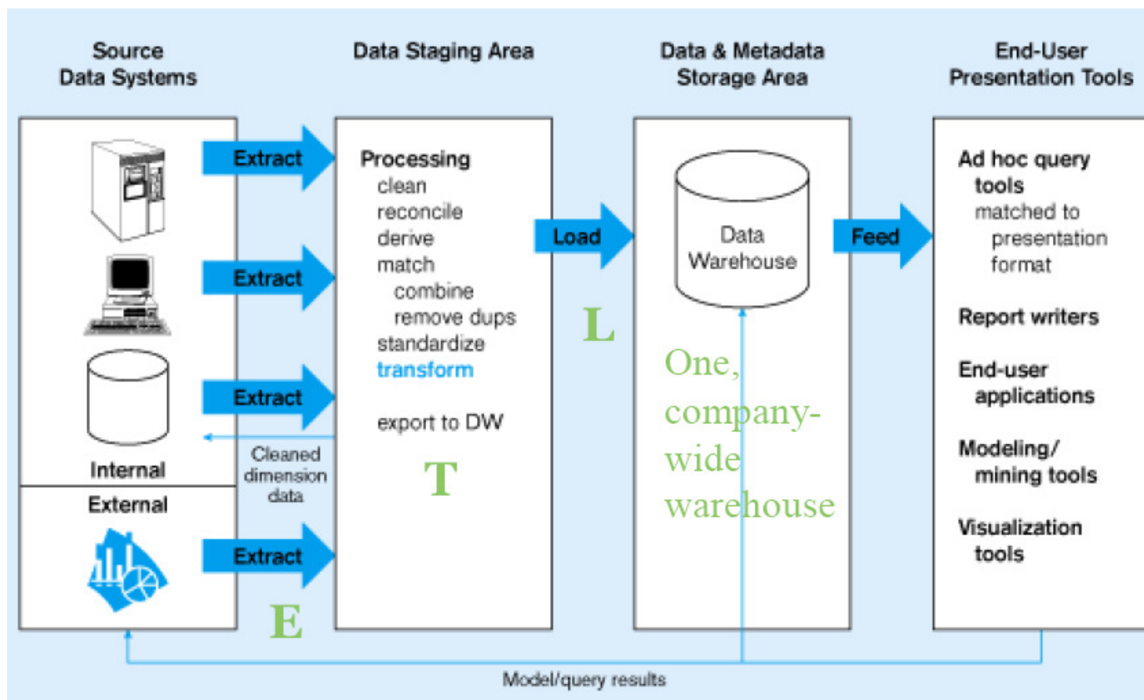


Figura 11. *Data Warehouse* centralizado.

Fonte: retirada de Fileto (2009).

A Figura 11 representa um DW centralizado. Na primeira coluna temos as fontes de dados do DW. Os dados são extraídos e enviados para o *Data Staging Area,* onde eles serão filtrados pelas ferramentas ETC.

Por fim, os dados são armazenados em um DW central, onde todas as consultas serão feitas somente nesse DW.

A distribuição física de um DW é utilizada quando a empresa possui diversos locais físicos e os dados em múltiplas instalações físicas, com administração também por um departamento de tecnologia de informação.

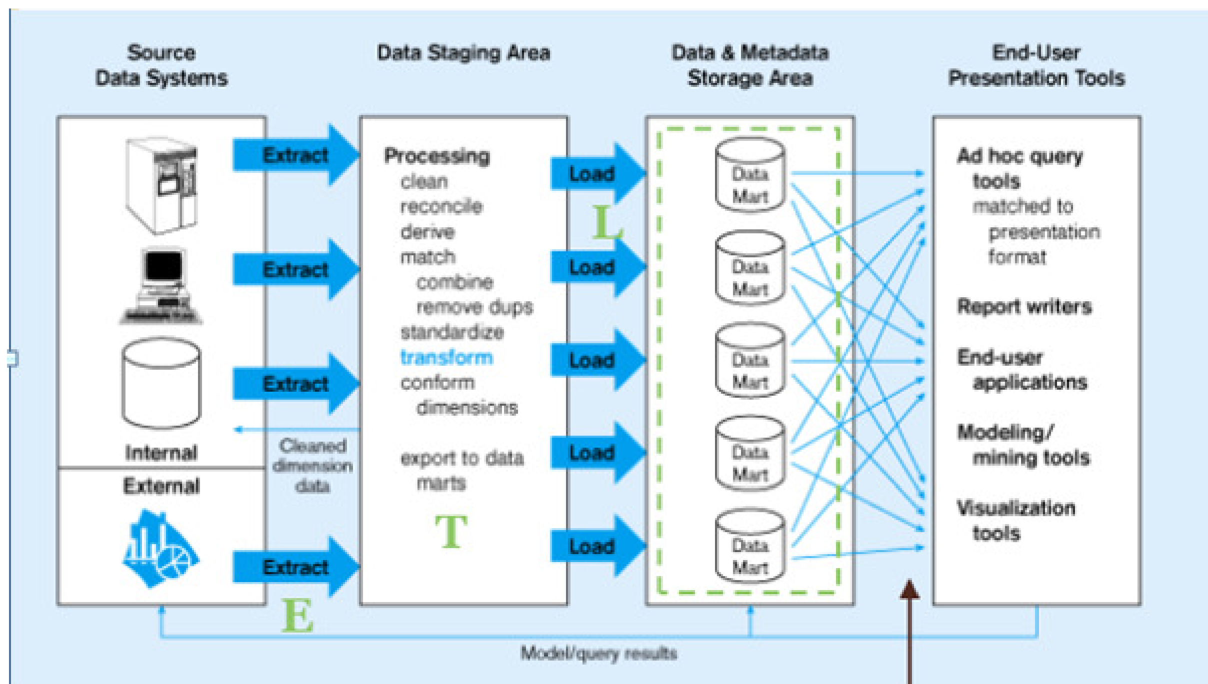


Figura 12. *Data Warehouse* descentralizado.

Fonte: retirada de Fileto (2009).

A Figura 12 representa um DW descentralizado ou distribuído. Na terceira coluna (Área de armazenamento), temos vários DMs, onde serão feitas todas as consultas. Os DMs são separados de acordo com áreas de interesse.

Temos ainda dois outros tipos de arquiteturas derivados do DW descentralizado. A arquitetura em nível de detalhes e DMs lógicos.

A arquitetura em nível de detalhes trabalha com dados sumarizados em um servidor, e dados detalhados em outro.

Por fim, a **arquitetura de DMs lógicos** representa os DMs como visões lógicas dos dados integrados no DW.

A Figura 13 representa os DMs lógicos. É interessante observar que o *Data Staging Area* e a Área de armazenamento são um só, o que facilita a alimentação dos DMs lógicos, tornando a alimentação dos DMs quase que ao mesmo tempo da extração dos dados nas fontes.

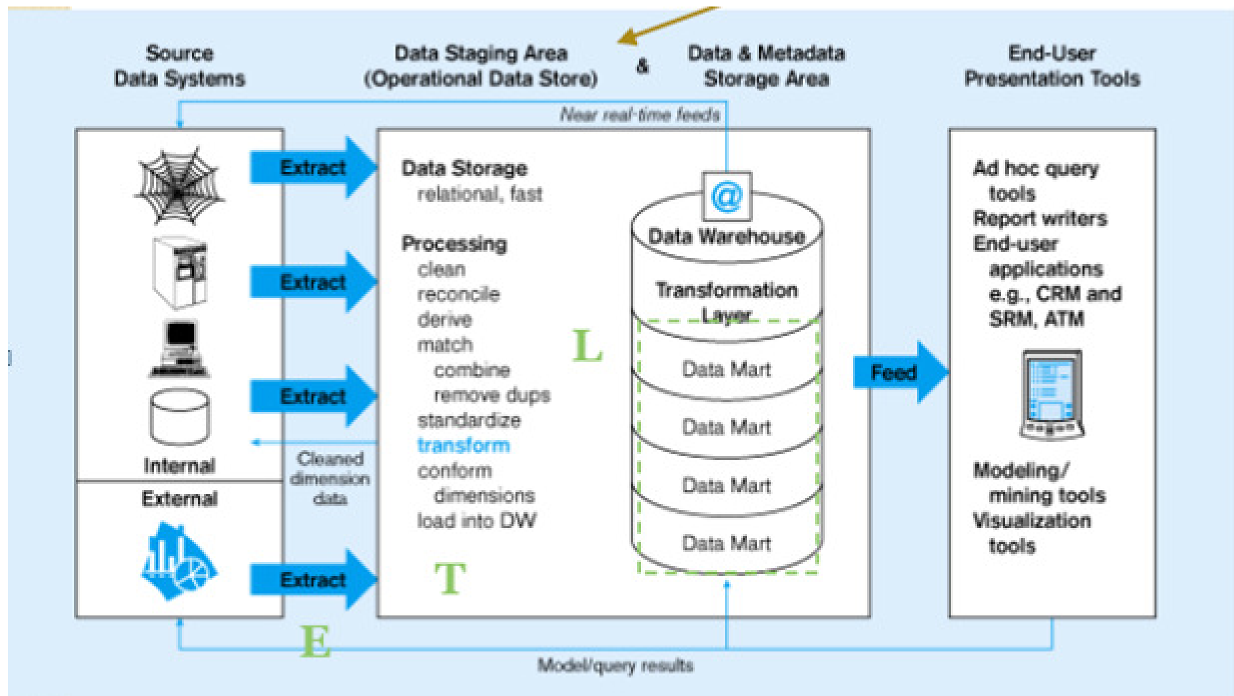


Figura 13. DM lógicos.

Fonte: retirada de Fileto (2009).

2.2.7.3. Ferramentas ETC

As ferramentas ETC possuem os seguintes objetivos:

Filtragem de dados: eliminam erros e elementos indesejados;

Integração de dados: correlacionam os dados de fontes de dados heterogêneas;

Conversão de dados: procedimento de conversão de formatos e unidades;

Condensação de dados: reduz o volume e otimiza o processamento;

Derivação de dados: define fórmulas para produzir novos valores a partir dos existentes.

Existem quatro passos a serem seguidos para realizar as tarefas descritas acima, conforme cita Fileto (2009):

O primeiro passo é a extração dos dados da fonte, capturando uma "fotografia" da fonte em um determinado momento. Essa captura pode ser feita também tendo o foco nos itens modificados desde a última extração. A Figura 14 representa o primeiro passo.

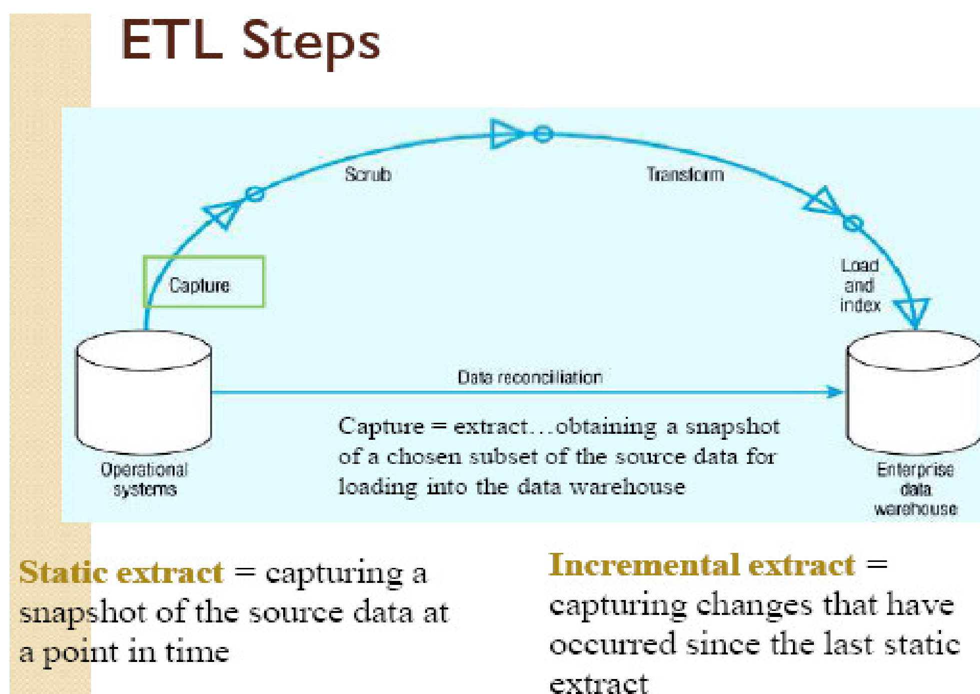


Figura 14. Captura de dados.

Fonte: retirada de Fileto (2009).

O segundo passo consiste em corrigir os erros vindos da extração, como palavras escritas erradas, uso incorreto dos campos, endereços que não se relacionam, dados perdidos, dados duplicados e inconsistências. A Figura 15 representa o segundo passo.

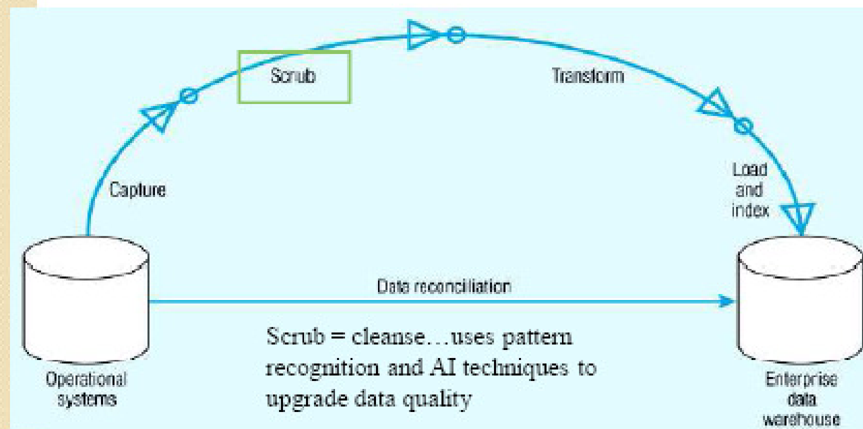


Figura 15. Filtragem.

Fonte: retirada de Fileto (2009).

O terceiro passo consiste na transformação dos dados. Os dados são convertidos do formato do sistema transacional para o formato do DW.

Em nível de registros, os dados são transformados através de operações de seleções, agrupamentos e agregações, e em nível de campo, um campo pode ser transformado em outro campo ou em vários campos e enfim, de vários campos em um campo. A Figura 16 representa o terceiro passo.

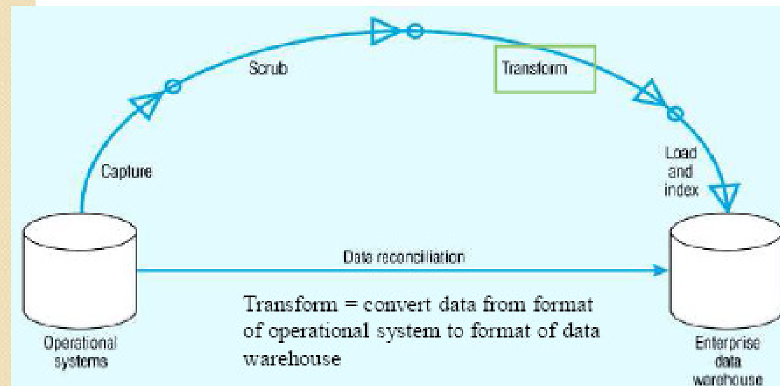


Figura 16. Transformação.

Fonte: retirada de Fileto (2009).

Por fim, o **último passo** consiste na carga e indexação dos dados. A Figura 17 representa o último passo.

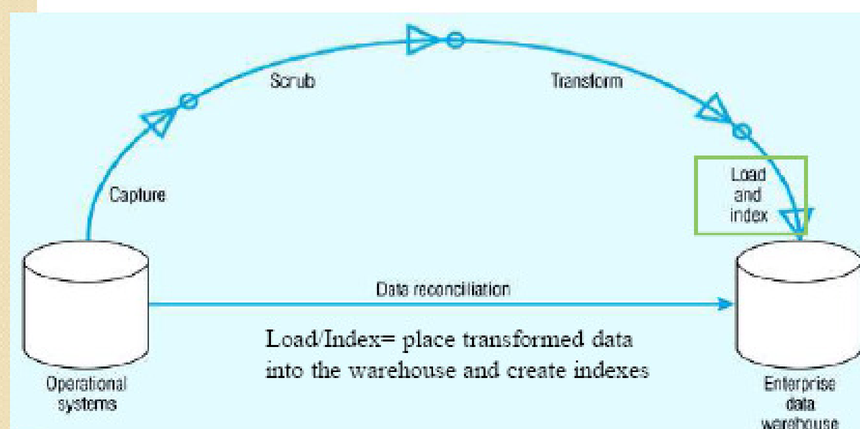


Figura 17. Carregamento e indexação.

Fonte: retirada de Fileto (2009).

2.3. Granularidade dos dados

Um dos elementos mais importantes da modelagem dos dados, seja qual for à arquitetura e a implementação usada, é a granularidade de dados.

A granularidade de dados representa o nível de sumarização dos dados e o nível de detalhamento dos dados; ou seja, quanto mais detalhes nós possuímos dos dados, menor é a granularidade. Conseqüentemente, quanto menor o nível de detalhamento dos dados, maior é a sua granularidade (Machado, 2000).

Isso influencia na construção do DW no aspecto de desempenho, ou seja, quanto maior o volume de dados, menor o desempenho do sistema.

É imprescindível a definição da granularidade dos dados do DW em um primeiro momento de sua construção, para facilitar a modelagem dos DMs. Uma vez definida a granularidade em um DM, ela deve ser utilizada para todo o DW, caso contrário, pode haver inconsistências no sistema.

Uma granularidade baixa é visada quando se procura manter um DW apenas como uma base histórica para a recuperação de dados, porém essa não é uma alternativa barata e não muito útil no ponto de vista da tomada de decisões, pois muitos dos dados não terão importância.

2.4. *Data Warehouse* usando o PostgreSQL

2.4.1. PostgreSQL

O PostgreSQL é um Sistema Gerenciador de Banco de Dados objeto-relacional *open-source,* desenvolvido na Universidade de Berkeley, Califórnia em 1986 e mantido por uma comunidade ativa de usuários.

A ferramenta possui vários recursos para a modelagem de projetos físicos de banco de dados, o que o torna comparável a produtos comerciais

similares de primeira linha. Hoje, o PostgreSQL e é utilizado por várias empresas de diversas áreas de atuação.

2.4.2. Características do PostgreSQL que favorecem um *Data Warehouse*

Seguem algumas características favoráveis ao uso do *Data Warehouse* com o PostgreSQL, conforme cita Berkus, Josh (2011):

- Cinco tipos diferentes de junções, podendo em uma consulta unirmos mais de 20 tabelas em uma consulta;

- Sub consultas são permitidas em qualquer cláusula da consulta;

- Consultas *“Windowing”:* Complemento das funções tradicionais de agregação, como SUM, AVG, COUNT que utilizam uma forma de acesso com menor custo computacional;

- Consultas recursivas;

- Uso de *tablespaces*, que possibilitam armazenar dados em diferentes locais (*Data Marts*);

- Para tratar casos de ETC, ao invés de usarmos o INSERT, considerado lento por inserir linha a linha, temos o COPY, considerado 3 a 5 vezes mais rápido, podendo ser executado em somente uma transação;

- Partição de tabelas: baseados no conceito de herança de tabelas;

2.5. *Rough Sets*

Para a tomada de decisões cada vez mais precisas, são necessários dados históricos para serem geradas probabilidades mais altas de acertos. Porém podem existir vários tipos de atributos que não fazem diferença para a tomada de uma decisão especifica: são os atributos irrelevantes. Eles fazem parte do conjunto de dados, porém podem ser descartados de tal forma que possa aumentar a eficiência de um modelo de decisão. Tendo nesse modelo de tomada de decisões somente os atributos relevantes ao contexto, podemos diminui a quantidade de atributos e conseqüentemente, a o tempo computacional.

A teoria de *Rough Sets* foi criada pelo cientista polonês Zdzisław I. Pawlak em 1982 (Patrício, Souza, Pinto, 2011), com o objetivo de gerar regras para um determinado conjunto de atributos, identificando um conjunto mínimo de atributos necessários através de aproximações inferiores e superiores.

2.5.1. Idéia do algoritmo de Rough Sets

A eliminação de atributos irrelevantes é feita através do processo de redução do sistema de informação. Para tal processo é usado a definição de redutos, ou seja, subconjuntos gerados a partir do conjunto de todos os atributos condicionais do nosso sistema que são capazes de proporcionar as mesmas propriedades de representação quando ela é feita por todos os atributos do sistema (Patrício, Souza, Pinto, 2011).

Pela teoria de *Rough Sets* também classificam-se objetos, agrupando-os em classes de equivalência quando estes possuem características similares.

Dizemos que dois objetos estão na mesma classe de equivalência quando os dois possuem os valores idênticos dos atributos levados em consideração.

Em caso de informações imprecisas, o algoritmo administra essas imprecisões e informações ruidosas e incompletas no sistema. Assim, os

objetos que não podem ser classificados através dos dados disponíveis são classificados através dos conceitos de aproximação inferior e superior (Patrício, Souza, Pinto, 2011).

2.5.2. Sistemas de informação

Um sistema de informação *A* possui a seguinte definição:

$$A = (U, C)$$

No qual *U* se refere aos registros contidos no sistema, e *C* são os atributos condicionais.

Pode ser que junto a esse sistema, temos também um atributo de decisão, ou seja, um atributo definido através do processamento dos atributos condicionais. Quando é esse o caso, a definição do sistema de informação é a seguinte:

$$\mathcal{R} = (U, C \cup D),$$

No qual *D* são os atributos de decisão do sistema.

A Tabela 2 representa um exemplo de um sistema de informação, no qual dez crianças, representadas por *U,* recebem cada uma um brinquedo com características diferentes, e no final elas tomam uma decisão sobre o brinquedo.

Tabela 2. Exemplo de um Sistema de Informação.

Fonte: retirada de Patrício, Souza, Pinto, (2011).

U	**Atributos Condicionais**					**Atributo de Decisão**
Criança	**Cor**	**Tamanho**	**Tato**	**Textura**	**Material**	**Atitude**
1	Azul	Grande	Duro	Indefinido	Plástico	Negativa
2	Vermelho	Médio	Moderado	Liso	Madeira	Neutra
3	Amarelo	Pequeno	Macio	Áspero	Pelúcia	Positiva
4	Azul	Médio	Moderado	Áspero	Plástico	Negativa
5	Amarelo	Pequeno	Macio	Indefinido	Plástico	Neutra
6	Verde	Grande	Duro	Liso	Madeira	Positiva
7	Amarelo	Pequeno	Duro	Indefinido	Metal	Positiva
8	Amarelo	Pequeno	Duro	Indefinido	Metal	Positiva
9	Verde	Grande	Duro	Liso	Madeira	Neutra
10	Verde	Médio	Moderado	Liso	Plástico	Neutra

2.5.3. Relação de indiscernibilidade

Considerando os atributos condicionais *C*, para todo subconjunto $B \subset C$ do sistema de informação *A,* uma relação de equivalência é associada, chamada relação de indiscernibilidade, definida como:

$$Ind_A(B) = \{(x, y) \in U^2 \mid \forall c \in B, c(x) = c(y)\},$$

No qual *c* se refere aos atributos condicionais de um determinado registro.

Assim, todo x e y são indiscerníveis entre si para todo atributo de *B* (Patrício, Souza, Pinto, 2011).

Resumindo, essa relação diz que dois objetos são idênticos considerando o conjunto de atributos *B.* O conjunto de todas as classes de equivalência determinadas por $Ind_A(B)$ é representado por:

$$^{U}/_{Ind_A(B)}$$

Um exemplo utilizando a Tabela 2:

Sendo *B = {textura}*, temos que:

$$^{U}/_{Ind_A(B)} = \{\{1,5,7,8\},\{2,6,9,10\},\{3,4\}\}$$

Cada grupo contendo os objetos indiscerníveis entre si são chamados de classes de equivalência. É importante lembrar que atributos de decisão não são analisados e nem agrupados nessas classes.

2.5.4. Aproximação de conjuntos

Analisando a Tabela 2, vamos verificar que atributos iguais aos das crianças 1 e 4 levam à uma atitude negativa, assim como atributos 2, 5 e 10 levam a uma atitude neutra, e por fim, 3, 7, 8 levam a uma atitude positiva. Mas por que não podemos afirmar nada sobre os registros 6 e 9?

Esses dois registros são indiscerníveis entre si para todos os atributos condicionais do sistema de informação. Porém o atributo de decisão é diferente, levando a uma inconsistência na geração de regras. Para a classificação de objetos nesse caso, usamos a teoria de aproximação.

Seja $A = (U, C)$ o sistema de informação, $B \subseteq C$e $X \subseteq U$, onde X é o conjunto de objetos ou de registros com respeito a *B,* isto é, *X* é obtido através das informações dos atributos de *B.* Assim define-se aproximação inferior de *X* em relação a *B*, denotado por$\underline{B}(x)$ e aproximação superior de *X* denotado por$\overline{B}(x)$como (Patrício, Souza, Pinto, 2011):

$$\underline{B}(x) = \{x \in U \mid {}^{U}/_{Ind_A(B)} \subseteq X\}, \text{ e}$$

$$\overline{B}(x) = \{x \in U \mid {}^{U}/_{Ind_A(B)} \cap X \neq \emptyset\}$$

- A aproximação inferior sugere que todos os objetos desse grupo são membros de *X;*

- A aproximação superior sugere que todos os objetos desse grupo podem ser membros de *X*;

- Região de fronteira sugere a subtração da aproximação superior pela inferior, ou seja, registros que não podem ser classificados em *X;*

- Fora da região sugere a subtração de todos os registros pela aproximação superior, ou seja, todos os registros que não são membros de *X;*

Um exemplo: Seja $\mathrm{B} = \{\mathrm{cor}, \mathrm{tato}, \mathrm{textura}, \mathrm{material}, \mathrm{tamanho}\}$, $^{U}/_{Ind_A}(B) = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6{,}9\}, \{7\}, \{8\}, \{10\}\}$

E $X = \{2, 5, 9{,}10\}$, são as crianças com atitudes neutras, temos que:

$\underline{B}(x) = \{\{2\}, \{5\}, \{10\}\}$, $\overline{B}(x) = \{\{2\}, \{5\}, \{10\}, \{6{,}9\}\}$, $RF(x) = \{\{6{,}9\}\}$, $U - \overline{B}(x) = \{\{1\}, \{3\}, \{4\}, \{7\}, \{8\}\}$

Essas aproximações são feitas para gerar as classes de equivalência e para classificar todos os objetos do sistema de informação de acordo com atributos de decisão.

2.5.5. Qualidade de aproximações

São percentuais obtidos através das próprias aproximações. Temos o coeficiente de imprecisão, coeficiente de qualidade de aproximação superior e coeficiente de qualidade de aproximação inferior.

O coeficiente de imprecisão mede a cardinalidade das duas aproximações.

$$\alpha_B(X) = \underline{B}(x) / \overline{B}(x)$$

Dentro do conceito desse coeficiente, temos as seguintes definições:

- Se $0 \leq \alpha_B \leq 1$ então o conjunto é definido como *Rough* (impreciso).

- Se $\alpha_B = 1$ então o conjunto é definido como *Crisp* (preciso).

O coeficiente de qualidade de aproximação superior mede a porcentagem dos objetos que podem ser membros de *X.*

$$\alpha_B(X) = \overline{B}(x) / U$$

O coeficiente de qualidade de aproximação inferior mede a porcentagem dos objetos que com certeza são membros de *X.*

$$\alpha_B(X) = \underline{B}(x) / U$$

Medindo o exemplo das aproximações, temos que:

$$\alpha_B(X) = \underline{B}(x) / \overline{B}(x) = \{2,5,10\} / \{2,5,10,6,9\} = 3/5 = 0{,}6$$

$$\alpha_B(X) = \overline{B}(x) / U = \{2,5,10,6,9\} / \{1,2,3,4,5,6,7,8,9,10\} = 5/10 = 0{,}5$$

$$\alpha_B(X) = \underline{B}(x)/_U = \{2,5,10\}/_{\{1,2,3,4,5,6,7,8,9,10\}} = 3/_{10} = 0,3$$

2.5.6. Redução do sistema de informação e geração de regras

Por fim, chegamos ao objetivo principal do algoritmo, que é reduzir o conjunto de atributos a fim de reduzir a complexidade do problema, evitando o desperdício de recursos e tempo computacional.

Para isso, devemos montar uma função de discernibilidade, que irá determinar o conjunto de atributos mínimos para diferenciar as classes de equivalência.

Antes de montar a função, precisamos de uma matriz de discernibilidade, que nos informa sobre os elementos que são diferentes de cada classe de equivalência, ou seja, para cada classe de equivalência, verificamos a classe anterior e recuperamos os atributos que diferem entre elas e inserimos na linha da classe corrente, na coluna da classe que estamos comparando.

A matriz de discernibilidade é definida por:

$$m_D(i,j) = \{b \in B \mid b(Cl(i) \neq b(Cl(j)\}, i,j = 1,2,\dots,n \text{ e} i \geq 1, j \leq n \; e \; n = \left| U/_{Ind_A(B)} \right|$$

No qual *cl* se refere às classes de equivalências geradas para um sistema de informação *A.*

A função de discernibilidade é definida por:

$$F_A(b_1^*, b_2^*, \dots, b_m^*) = \wedge \{\vee \; m_D^*(i,j) | i,j = 1,2,\dots,n, m_D(i,j) \neq \emptyset \}, m_D^*(i,j) = \{b^* | b \in m_D(i,j) \}$$

Essa função diz que para obtermos os redutos, precisamos fazer o "E" lógico de todas as células da matriz de discernibilidade, e o "Ou" lógico de todos os elementos das células.

Quando montamos a função e a simplificamos através de relações de álgebra relacional, chegamos ao resultado final, que é o subconjunto mínimo de atributos, usados para as definições das regras que serão aplicadas no sistema de informação.

Para todos os termos encontrados na função simplificada, devemos verificar se cada um deles mantém as mesmas propriedades do conjunto original. Caso afirmativo, podemos considerar esse termo como um reduto.

É importante lembrar que somente as regras determinísticas são válidas no sistema. São as regras que estão na aproximação inferior do sistema, ou seja, que não são imprecisas.

Para reduzir a função de discernibilidade, o projeto usou o algoritmo de Quine-McCluskey, pois este método possui uma boa solução computacional e pode ser usado para um maior número de variáveis, quando comparado com o mapa de Karnaugh.

2.6. Método de Quine-McCluskey

Este método, também chamado de Q-M ou de método tabular, foi inicialmente proposto por Quine em 1952 e posteriormente aperfeiçoado por McCluskey em 1956. Ele reduz expressões booleanas independente de quantas variáveis são usadas, além de ser um bom método para uma solução computacional (Crenshaw, Jack, 2004).

Basicamente o método consiste em recuperar minitermos, ou seja, linhas da tabela verdade cujo resultado da função resulta em '1', que serão candidatos a fazerem parte da função simplificada; agrupá-los pela quantidade de '1's em sua cadeia de *bits*, realizar combinações entre os grupos adjacentes até encontrarmos todos os primos implicantes possíveis, ou seja, termos que não podem mais ser combinados que são capazes de representar a função original.

O algoritmo deve seguir os seguintes passos, conforme cita Katekawa (2011):

- Listar os minitermos binários, separados pela quantidade de ‘1s’ presentes em cada minitermo;

- Combinar os minitermos adjacentes quando estes diferem em somente uma posição; quando esses minitermos são combinados, um novo minitermo é criado, e a posição que difere é marcada com um símbolo ‘_’ no novo minitermo;
  Este símbolo é considerado um terceiro bit para a uma futura combinação. Se um minitermo não pode ser combinado, ele é marcado como um primo implicante;

- Construir uma tabela contendo os primos implicantes nas linhas, e a linha de todos os minitermos nas colunas;

- Selecionar os primos implicantes essenciais, que são capazes de cobrir todas as colunas geradas. Os conjuntos desses primos devem ser escritos na forma soma de produtos, gerando assim a função simplificada.

Segue um exemplo para um melhor entendimento do método de Quine – McCluskey apresentado em Katekawa (2011):

Dada uma expressão booleana em sua forma canônica,

$$F(A,B,C,D) = \sum m\,(2,4,6,8,9,10,12,13,15)$$

No qual *m* corresponde aos minitermos da função.

Agrupamos os minitermos pelo numero de ‘1s’ de cada minitermo, conforme a Tabela 3.

Tabela 3. Minitermos agrupados.

Fonte: retirada Katekawa (2011).

Linha	Minitermos
2	0010
4	0100
8	1000
	—
6	0110
9	1001
10	1010
12	1100
	—
13	1101
	—
15	1111

A próxima etapa é combinar esses minitermos e gerar um novo minitermo com uma nova linha, e a combinação dos minitermos, com a posição do bit diferente marcada com o símbolo ‘_’. Quando um minitermo não pode mais ser combinado, ele é marcado como um primo implicante.

Linha	Minitermos
2	0010
4	0100
8	1000
	—
6	0110
9	1001
10	1010
12	1100
	—
13	1101
	—
15	1111

Linha	Minitermos
2,6 *	0_10
2,10 *	_010
4,6 *	01_0
4,12 *	_100
8,9	100_
8,10 *	10_0
8,12	1_00
	—
9,13	1_01
12,13	110_
	—
13,15 *	11_1

Linha	Minitermos
8,9,12,13 *	0_1_

Figura 18. Combinação de minitermos.

Fonte: retirada Katekawa (2011).

Quando nenhum termo pode mais ser combinado, montamos uma tabela de primos implicantes como linhas, e os minitermos como linhas:

Tabela 4. Tabela de Quine-McCluskey.

Fonte: retirada Katekawa (2011).

	2	4	6	8	9	10	12	13	15
8,9,12,13				*	*		*	*	
2,6	*		*						
2,10	*					*			
4,6		*	*						
4,12		*					*		
8,10				*		*			
13,15								*	*

Para escolhermos os primos implicantes que farão parte da função simplificada, devemos primeiro encontrar os primos essenciais, ou seja, aqueles que cobrem os minitermos únicos. Nesse caso, os minitermos 9 e 15 são cobertos unicamente pelos primos “8, 9, 12, 13” e “13, 15” respectivamente, então esses compõem os primos essenciais.

Esses primos cobrem também os minitermos ‘8’, ’12’ e ‘13’; então falta cobrir os minitermos ‘2’, ’4’, ’6’ e ‘10’. A idéia é achar o conjunto mínimo de primos que cobrem os minitermos que restam. Para encontrarmos o conjunto mínimo de minitermos, escolhemos os primos que cobrem o maior numero possível de minitermos restantes.

Quando encontramos todos esses primos, podemos relacioná-los com seus respectivos minitermos e encontramos a função reduzida:

$$F(A,B,C,D) = (8{,}9{,}12{,}13) + (13{,}15) + (2{,}10) + (4{,}6) =$$

$$F(A,B,C,D) = 0_1_ + 11_1 + _010 + 01_0 =$$

$$F(A,B,C,D) = A'C + ABD + B'CD' + A'BD'$$

No qual o símbolo ’ corresponde à negação da variável, ou seja, o *bit* zero.

Concluindo, esse método suporta um número ilimitado de variáveis e possui uma solução computacional favorável, porém seu uso se torna limitado quando se trata de um número extenso de variáveis, pois sua complexidade cresce exponencialmente quando aumentamos a quantidade de variáveis usadas na expressão, aumentando assim o número de implicantes primos, que normalmente já é grande. Mais detalhes na Seção 3.4.7.

2.7. Tecnologia JAVA

Para o desenvolvimento proposto no projeto, escolhemos a linguagem de programação JAVA.

Na maioria das linguagens de programação, é preciso compilar ou interpretar um programa para que ele seja executado em um computador. A linguagem JAVA é diferente: o compilador inicialmente transforma o programa em uma linguagem intermediária, chamada *bytecode.* Esse código é independente de plataforma, e é mais tarde interpretado por um interpretador JAVA. A compilação acontece somente uma vez; a interpretação acontece todas as vezes que seu programa é executado (Mengue, 2002).

Os *bytecodes* correspondem a instruções que serão dadas à máquina virtual JAVA, que é a plataforma em que iremos executar nossos programas.

Algumas características foram favoráveis para a escolha da linguagem. Seguem algumas delas, conforme cita Mengue (2002):

- Eficiência e extensibilidade da orientação de objetos;

- Alto desempenho, permitida pela Máquina virtual JAVA, que executa o código na maior velocidade possível;

- Independência de hardware e software para a execução de programas JAVA.

Capítulo 3

Implementação de *Rough Sets*

3.1. Considerações iniciais

A proposta teve por objetivos implementar o algoritmo de *Rough Sets* para a classificação de informações de acordo com a equivalência de seus atributos, redução um conjunto de informações para uma posterior otimização de tempo computacional e por fim, a geração de regras para um dado reduto encontrado em um *Data Warehouse*.

Também podemos aplicar o algoritmo desenvolvido em bases transacionais, levando em conta que o PostgreSQL é uma Sistema Gerenciador de Banco de Dados objeto–relacional e que a arquitetura da implementação permite adaptações de uma base transacional em alguns conceitos utilizados em um *Data Warehouse,* porém o uso de uma base transacional e suas adaptações não são o foco do projeto.

Em todo o desenvolvimento do sistema foi adotado o paradigma de software livre para viabilizar seu uso em diversos fins, tendo em vista o elevado custo de um projeto de uma ferramenta de suporte a decisão e sua restrição de acesso a grandes organizações.

3.2. Organização da Implementação

A implementação está dividida em três partes:

- Definição dos sistemas de informações, ou seja, o conjunto de dados em que o algoritmo será aplicado;

- Aplicação do algoritmo *Rough Sets* para um sistema de informação, seguindo as etapas do algoritmo de acordo com a ordem apresentada na implementação. Todas as etapas do algoritmo podem ser visualizadas através de tabelas após sua execução, permitindo uma maior clareza no funcionamento do algoritmo e deixando o usuário ciente de todas as modificações ocorridas em seu conjunto de dados durante o processo;

- Configuração com o banco de dados, que será usada para definirmos a base de dados do *Data Warehouse* escolhido para a aplicação do algoritmo.

3.3. Definição dos Sistemas de Informações

3.3.1. Sistemas de Informações

A primeira etapa consiste basicamente em definir o sistema de informação, porém de um modo personalizado.

Ao invés de levar a base de dados inteira em consideração, podemos relacionar somente partes do sistema que são interessantes para a sua aplicação.

O fato escolhido no SI se refere à tabela fato do *Data Warehouse* que iremos trabalhar.

Por fim, temos a definição das referências, ou seja, as dimensões que serão usadas para o SI. Elas representam todas as tabelas que fazem parte da base de dados, e podemos escolher quantas referências forem necessárias para a aplicação, inclusive a própria tabela fato.

.

3.3.2 Atributos condicionais

Os atributos condicionais são de fato todos os atributos levados em consideração para a aplicação do algoritmo. Eles podem pertencer tanto à tabela fato quanto às suas dimensões.

O programa contém uma restrição a respeito de níveis de hierarquia dentro de uma dimensão. Na verdade, podemos descer no máximo uma hierarquia da tabela fato, ou seja, não é possível escolhermos algum atributo da dimensão que está situado em hierarquias inferiores. Mais detalhes dessa restrição são descritas na Seção 4.1.2.

Os atributos condicionais podem ser tanto quantitativos quanto qualitativos. Em um sistema onde temos somente atributos nominais, fica fácil compararmos se um atributo é igual a outro ou não, basta que os dois sejam exatamente iguais.

Porém para uma aplicação voltada a atributos quantitativos, nem sempre um atributo precisa ser exatamente igual a outro para ambos serem considerados de uma mesma classe de equivalência. Uma aplicação pode requerer uma faixa de valores onde os atributos podem ser considerados "iguais".

Um exemplo é que podemos estabelecer uma faixa de valores onde um dado atributo pode representar a média geral de todos os registros. Então podemos descrever uma faixa de valores onde todos os registros cujo atributo escolhido está entre essa faixa podem ser considerados iguais.

Na implementação, tratamos os atributos condicionais com o tipo do atributo (0 – qualitativo, 1 – quantitativo), e também com faixas de valores (faixa_inicial e faixa_final) para atributos quantitativos.

Podemos escolher quantos atributos condicionais forem necessários para a aplicação, porém o algoritmo usado para a redução da função de discernibilidade gerada na etapa final do algoritmo possui certa limitação quanto ao numero de variáveis booleanas utilizadas na função em decorrência do tempo necessário para reduzir uma função com um número expressivo de variáveis.

Essas variáveis booleanas descritas na função de discernibilidade estão relacionadas com os atributos condicionais escolhidos nessa etapa, pois as essas variáveis podem representar todos os atributos condicionais, caso dado atributo escolhido não seja igual para todas as classes de equivalências criadas. Essa restrição é detalhada no Capítulo 4.

Quando definimos todos os atributos condicionais do SI e gravamos o registro, o programa cria em tempo de execução uma tabela contendo todos os atributos escolhidos, para em uma posterior etapa definirmos as classes de equivalência.

Essa tabela é criada com o nome do SI em conjunto com sua chave primária. Os atributos que compõe a tabela são os atributos condicionais definidos para o SI, cujos nomes são compostos pela referência dos atributos, o símbolo "_" e os nomes dos atributos.

Temos também como atributo dessa tabela a chave primária da classe de equivalência.

Sempre que gravamos um registro, o programa verifica se a tabela já existe; caso exista, o programa exclui essa tabela e cria uma nova com os atributos condicionais correntes.

Mais informações sobre a definição das classes de equivalências se encontram na Seção 3.4.1.

3.3.3. Atributos de decisão

No final dessa primeira etapa do programa, temos a definição dos atributos de decisão. São os atributos que classificam cada registro da tabela fato, e precisam estar presentes para uma melhor definição das regras definidas no SI na parte final do algoritmo.

Para um fim experimental, podemos também simular esses atributos de decisão dentro da tabela fato, para verificar as variações de aproximações de conjuntos para diferentes decisões. Essas simulações são feitas de forma aleatória e gravadas na tabela fato.

3.3.4. Requisitos do programa para os Sistemas de Informações

O programa adotou um padrão das tabelas para ser capaz de percorrer os atributos das tabelas da base de dados escolhidas e também das tabelas criadas em tempo de execução, pois não sabemos de início quais atributos serão definidos como condicionais no SI.

Como podemos ter vários tipos de modelagem de banco de dados, fica um tanto confuso poder identificar os atributos que representam chaves primárias e estrangeiras em tempo de execução, pois o arquiteto do banco de dados que define o nome de seus atributos, podendo ele ter seu próprio padrão dentro da base de dados.

Em todo o sistema, os atributos de uma tabela que são chaves primárias ou estrangeiras precisam iniciar com o nome da tabela onde o atributo é chave, e terminar com os seguintes caracteres: "_id". Com isso, podemos concluir qual tabela a chave faz parte, e que ela é uma chave, pois termina com os caracteres "_id".

Esse padrão é necessário principalmente para a identificação dos atributos condicionais nas tabelas que possuem chave estrangeira na tabela fato, ou seja, para identificarmos os atributos das dimensões do fato ao longo das etapas da aplicação do algoritmo de *Rough Sets.*

Como o objetivo do projeto não foca parametrizações para tornar o programa compatível com qualquer base de dados que for incluída, foi desenvolvido esse padrão, que pode ser considerado presente em muitas modelagens desenvolvidas em bancos objeto-relacionais. Tais parametrizações seriam também um tanto quanto exaustivas, pois o usuário final teria que conhecer toda a modelagem do banco de dados para parametrizar as chaves das referências e por fim poder incluir os atributos no SI.

Na definição dos atributos de decisão, precisamos ter um campo na tabela fato que mencione uma decisão tomada a respeito de cada registro informado na tabela fato, para então esse atributo ser vinculado ao atributo de decisão registrado no sistema.

As decisões são definições do algoritmo de *Rough Sets,* e esses precisam ser informados antes da aplicação do algoritmo, pois todas as aproximações de conjuntos e todas as regras geradas para o conjunto de atributos relevantes encontrados na parte final do algoritmo dependem das decisões informadas na tabela fato.

3.4. *Rough Sets*

3.4.1. Classes de Equivalência

3.4.1.1. Definição das classes e das relações de indiscernibilidade

Para definirmos as classes de equivalência, seguimos os passos abaixo:

- Agrupamos todas as referências contidas na definição de atributos condicionais;

- Para cada atributo, precisamos saber seu nome e sua referência, para buscá-lo nas tabelas da base de dados da aplicação e a faixa de valores que ele pode assumir, caso seja um atributo quantitativo;

- Antes de percorrermos as tabelas da aplicação, precisamos primeiro verificar se as classes e as relações já foram definidas anteriormente para apagar seus registros, pois elas podem ter sido geradas para outro conjunto de atributos condicionais;

- Percorremos a tabela fato para identificar os valores de todos os atributos condicionais pertencentes à tabela fato e a suas dimensões. Essa é a etapa principal do processamento, e possui os seguintes passos:

    - Para cada registro lido da tabela fato, temos que verificar se ele é equivalente aos registros já inseridos na tabela das classes de equivalência. Caso seja equivalente, não inserimos um registro novo na tabela, pois a classe já está definida. Caso contrário, devemos inserir o registro corrente;

O primeiro registro lido na tabela fato se torna o primeiro registro na tabela de classes de equivalências;

- No caso dos atributos quantitativos, devemos ter o cuidado de somente inserir em uma classe de equivalência somente aqueles que pertencem à faixa de valores definida. Caso um atributo de um registro lido não esteja entre essa faixa, devemos verificar se ele é exatamente igual ao atributo da classe corrente. Se não for exatamente igual, devemos incluir uma nova classe de equivalência;

- Após a definição de todas as classes de equivalência, precisamos definir a relação de indiscernibilidade para os atributos condicionais do SI e os registros da tabela fato;

- Para isso, usamos um processamento que cria uma tabela contendo todas as chaves primárias da tabela fato, suas respectivas classes de equivalência e por fim os atributos de decisão de cada registro, que serão usados posteriormente para definirmos as aproximações dos conjuntos;

- Usando essa tabela, agrupamos os registros por classes de equivalência, e chegamos à relação de indiscernibilidade listando todos os registros de acordo com sua classe de equivalência;

- Ao final do processamento, podemos visualizar uma tabela contendo todas as classes de equivalência encontradas, e a relação de indiscernibilidade gerada para aquele conjunto de informações; isso nos permite verificar quais campos são diferentes para cada classe de equivalência e verificar quais registros pertence à mesma classe de equivalência;

- É importante observar que a ordem descrita nas colunas da tabela visualizada deve estar na ordem dos atributos condicionais incluídos para o SI em questão, pois essa ordem terá que ser respeita para passos posteriores do algoritmo;

- Para esse processo, foi escolhida a estrutura de listas para todo o processo de definição das classes de equivalência e da relação de indiscernibilidade, pois foi necessária uma estrutura que contivesse outras variáveis necessárias para a comparação de atributos, como o nome dos campos, faixa de valores, e o próprio valor do atributo na aplicação, e que variasse seu tamanho de acordo com o tamanho de atributos condicionais;

- Por fim, esse processo também é usado quando precisamos definir as relações de indiscernibilidade de todos os termos encontrados na função de discernibilidade em sua forma reduzida para verificarmos se os termos são capazes de manter a mesma relação de indiscernibilidade contida no conjunto original de atributos condicionais. Nesse caso é utilizada uma tabela temporária semelhante à tabela das classes de equivalência, só que com os atributos do termo em questão ao invés do conjunto inteiro;

  Essa tabela também é excluída e criada a cada vez que precisamos definir as classes e as relações. Esse processamento no estágio de comparação de relações de indiscernibilidade entre termos é mais bem detalhado na Seção 3.4.4;

3.4.2. Aproximações de conjuntos

3.4.2.1. Processamento do fato

Essa etapa do algoritmo visa uma melhor visualização do conjunto de dados em relação a suas classes de equivalência e seus atributos de decisão.

Antes do cálculo de aproximações, precisamos primeiro realizar um processamento que une em uma tabela as informações necessárias para a definição das aproximações. Esse processamento consiste em unir em uma tabela os registros da tabela fato e das classes de equivalência.

Na tabela fato, precisamos de duas informações: a chave primária dos registros e seu atributo de decisão. Na tabela de classes, precisamos somente de sua chave primária.

Tendo em mãos essas informações, somos capazes de calcular todas as aproximações, e calcular também suas qualidades, e por fim, definir se o conjunto é classificado como *Rough,* isto é, se o coeficiente de imprecisão está entre zero e um, ou é classificado como *Crisp,* isto é, se o coeficiente de imprecisão é igual a um.

O processamento do fato é gravado na tabela "aproximacoes_conjuntos".

3.4.2.2. Aproximações de conjuntos

O programa adotou o conceito do *X* como o conjunto de dados que possui um atributo de decisão específico escolhido para o cálculo. Isso para que a aplicação tenha o foco em conceitos já especificados na tabela fato, ou seja, ao invés de escolhermos um conjunto de atributos qualquer para o cálculo das aproximações, o programa permite o usuário escolher um conjunto de atributos que pertencem aos atributos de decisão definidos no SI.

Uma vez processado o fato e escolhido o atributo de decisão, podemos escolher as aproximações que queremos calcular e por fim executar os cálculos.

O programa realiza o cálculo de todas as aproximações através do número mínimo de consultas SQL para cada tipo de aproximação.

A princípio, esses cálculos estavam sendo feitos através de algumas consultas no banco e processamentos em memória, porém o tempo de espera estava um tanto quanto elevado.

Uma vez realizado todas as definições das aproximações através de um conjunto mínimo de consulta SQL, temos um menor tempo de execução do cálculo da aproximação.

Todos os processamentos usados na definição de aproximações de conjuntos retornam os registros da tabela fato contidos nos conjuntos de aproximação escolhidos (inferior, superior, fronteira e fora da região), e esses são exibidos em uma tabela, que contém somente os registros. Podemos detalhar essa tabela com a opção de visualização do conjunto, e então temos o registro contendo todas as informações dos atributos condicionais.

Esse tipo de visualização favorece o suporte a decisões dentro da tabela fato escolhida, pois as aproximações definem o comportamento dos registros para cada tipo de atributo de decisão, e também fornece o cálculo da qualidade das aproximações. Mais detalhes desse cálculo de qualidade estão na Seção 3.4.2.3.

Seguem as definições das aproximações e a aplicação delas no programa:

Aproximação inferior: recupera todos os registros que possuem o atributo de decisão escolhido, e que para cada classe gerada desses registros, não existam atributos de decisão diferentes.

Em outras palavras, podemos dizer que esses registros são os que com certeza possuem o atributo de decisão igual ao escolhido.

No programa, usamos uma consulta e uma sub consulta para recuperar esses registros.

A primeira seleciona os registros que tem o atributo de decisão igual ao escolhido, e a segunda seleciona os registros que possuem o atributo de decisão diferente do escolhido. Por fim, selecionamos os registros contidos na primeira consulta que não estão contidos na segunda para obter o conjunto de registros contidos na aproximação inferior.

Aproximação superior: recupera os registros que podem estar dentro do conjunto do atributo de decisão escolhido, ou seja, são escolhidos os registros que para sua classe de equivalência, pelo menos uma contenha o atributo de decisão escolhido.

Em outras palavras, podemos dizer que os registros da aproximação superior são aqueles que com certeza estão na aproximação inferior e os que podem estar na aproximação inferior.

No programa, usamos duas consultas para recuperar esses registros.

A primeira identifica todas as classes de equivalência cujos registros possuem o atributo de decisão igual ao atributo escolhido. A segunda consulta identifica os registros da tabela fato que estão contidos na primeira consulta para chegar ao conjunto de registros que estão contidos na aproximação superior.

Região de fronteira: é o processamento de aproximação de conjuntos mais complexo. Precisamos identificar todos os registros cujas classes de equivalência se repetem e que pelo menos um desses registros deve possuir o atributo de decisão igual ao atributo escolhido e pelo menos um atributo de decisão que seja diferente do atributo escolhido.

Em outras palavras, podemos dizer que são os registros imprecisos, pois possuem classes de equivalências iguais com atributos de decisão diferentes. Esses registros são os ditos impossíveis de serem classificados em *X.*

Foi necessário um maior conjunto de instruções no programa para encontrar esses registros. Seguem os passos:

- Identificamos todas as classes de equivalência que se repetem;

- Percorremos todas as classes que se repetem; para cada classe, fazemos a contagem dos registros que pertencem à classe corrente e que o atributo de decisão seja igual ao atributo escolhido;

- Da mesma forma, fazemos a contagem dos registros que estão dentro da classe de equivalência corrente e que possuem o atributo de decisão diferente do atributo escolhido;

- Por fim, verificamos se de fato, os registros dentro da classe de equivalência corrente possuem pelo menos um atributo de decisão igual e pelo menos um atributo de decisão diferente do atributo escolhido. Se a resposta for sim, podemos considerar os registros dentro da região de fronteira.

Fora da região: Simplesmente escolhemos os registros cujo atributo de decisão é diferente do escolhido, porém tomando o cuidado de não escolher algum registro que faz parte da região de fronteira.

No programa, usamos uma consulta e uma sub consulta para recuperar esses registros.

A primeira consulta busca os registros cujo atributo de decisão seja diferente do atributo escolhido, e a segunda buscamos os registros as classes de equivalência que possuem pelo menos um atributo de decisão igual ao escolhido.

Por fim, retornamos os registros da primeira consulta que não estão contidos na segunda consulta, pois a segunda consulta retorna classes de equivalência que estão na região de fronteira.

3.4.2.3 Qualidade de Aproximações

Essas qualidades medem os percentuais dos resultados encontrados nas aproximações superiores e inferiores. Caso uma delas não seja escolhida para o cálculo, não podemos calcular todas as qualidades.

Coeficiente de imprecisão: o programa simplesmente recupera a quantidade de registros contidos na aproximação inferior e divide pela quantidade de registros da aproximação superior, e por fim classifica o coeficiente de imprecisão.

Coeficiente de qualidade de aproximação superior: o programa recupera a quantidade de registros contidos na aproximação superior e divide pela quantidade total de registros da tabela fato.

Coeficiente de qualidade de aproximação inferior: o programa recupera a quantidade de registros contidos na aproximação inferior e divide pela quantidade total de registros da tabela fato.

3.4.3. Matriz de discernibilidade

A matriz de discernibilidade é responsável por definir todos os atributos condicionais que estarão presentes no cálculo do reduto, e por formar a função de discernibilidade.

Mais uma vez no programa, precisamos percorrer as classes de equivalência, só que dessa vez, ao contrário de sua definição em que encontramos os atributos condicionais equivalentes, dessa vez devemos procurar os atributos condicionais que diferem entre duas classes.

O programa realiza duas consultas principais:

A primeira recupera todos os registros das classes de equivalência, e a segunda recupera todos os registros das classes de equivalência cujo que são menores que o registro corrente, ou seja, que estão acima do registro corrente. Assim, temos para cada registro corrente, todos os registros que estão contidos nas classes de equivalência menores, e então devemos comparar o registro corrente com todos os menores.

Ao comparar um par de registros, armazenamos todos os atributos condicionais que diferem entre eles na matriz, na posição linha igual à linha da classe corrente, e coluna igual à linha da classe menor que está sendo comparada.

Como iremos comparar a classe corrente somente com as classes menores, em cada linha da matriz de discernibilidade preenchemos somente até a posição da coluna anterior à linha corrente. Por exemplo, se estamos verificando a classe numero cinco, preenchemos as colunas até o numero quatro nessa linha, e todas as outras colunas dessa linha não são levadas em consideração.

Esse aspecto deve ser levado em consideração quando for necessário percorrer essa matriz, pois senão caímos no problema de ler posições irrelevantes, e em uma matriz com um elevado numero de linhas e colunas, isso se torna uma tarefa extremamente cara e desgastante.

Quando chegamos ao fim da primeira consulta, temos todas as posições da matriz preenchidas propriamente, e então podemos visualizá-la em forma de tabela.

A princípio, a matriz estava sendo armazenada em uma tabela no banco de dados, para uma posterior visualização e leitura da matriz sem necessitar de um novo processamento, caso o mesmo já tivesse sido realizado.

A tabela da matriz de discernibilidade estava sendo criada em tempo de execução porque não sabemos a priori as classes de equivalência que irão ser geradas.

Seus atributos foram definidos como o numero seqüencial das classes de equivalência, e o tipo do atributo teve que ser definido como *text*, pois não sabemos quantos atributos condicionais podem fazer parte de uma célula da matriz.

Porém com o crescimento dos registros da tabela fato, temos conseqüentemente o crescimento do numero de registros contidos na classe de equivalência e então o numero de linhas e colunas da matriz de discernibilidade também aumenta.

Obviamente, sempre temos a situação de no mínimo um atributo condicional diferir entre as classes, então nunca uma linha da matriz de discernibilidade será nula, com exceção da primeira que pela definição no algoritmo, é sempre nula.

O tamanho dessa matriz de discernibilidade é um grande problema quando armazenamos a matriz dessa forma no banco de dados, pois o PostgreSQL possui uma restrição em relação ao tamanho da tabela na relação linha x coluna. Mais detalhes sobre essa restrição se encontram na Seção 4.1.4.

Como não é o foco do projeto o tipo armazenamento dessa estrutura, foi decidido que a matriz estaria armazenada na memória, pois além de sua leitura ser mais eficiente, não temos o problema do limite do armazenamento no banco de dados.

A matriz foi definida como uma variável global no projeto, e instanciada quando fazemos a definição da matriz de discernibilidade.

Como o programa suporta a aplicação do algoritmo para vários SI, temos que ter o cuidado de limpar a matriz cada vez que trocamos o SI. Então a etapa de definição da matriz de discernibilidade precisa ser sempre

executada para o SI que estamos procurando os redutos, caso contrário corremos o risco de usar uma matriz de discernibilidade diferente da matriz do SI atual.

3.4.4. Redução do sistema

Essa é a etapa final do algoritmo, que consiste em encontrar um conjunto mínimo de atributos que mantém as mesmas características do conjunto original, isto é, a mesma relação de indiscernibilidade.

Para concluirmos essa etapa, precisamos recorrer à álgebra booleana, para montar a tabela verdade da função de discernibilidade, pois ela é gerada a partir da matriz de discernibilidade é uma função booleana, e para compreender conceitos que estão presentes no algoritmo de redução dessa função, chamado de Quine-McCluskey.
Seguem as etapas da redução do sistema:

- Montar a função de discernibilidade;
- Montar a tabela verdade e a função canônica dessa função;
- Reduzir a função de discernibilidade usando o algoritmo de Quine-McCluskey.

Definição da função de discernibilidade: A idéia dessa função é selecionar as células da matriz, e coluna a coluna, percorrer as linhas, fazendo o “ou” lógico entre os elementos da célula e o “e” lógico entre as linhas da matriz.

Após a definição da matriz de discernibilidade, podemos percorrer a matriz para gerar a função de discernibilidade, porém é preciso certo cuidado para percorrer essa matriz, pois ela contém várias células nulas, e percorrer essas seria tempo computacional desperdiçado.

Ao percorrer a matriz, fazemos dois laços de repetição: o primeiro é responsável por percorrer as colunas, e o segundo é responsável por percorrer as linhas, porém o segundo irá ser iniciado pela próxima posição do laço anterior. Isso significa que sempre iremos iniciar a linha com uma posição superior a coluna, pulando assim os valores nulos da matriz.

Como descrito anteriormente, para cada célula percorrida, fazemos o "ou" lógico entre todos os elementos desta, e concatenamos com o "e" lógico da próxima célula.

Por fim essa função é armazenada no banco de dados. É importante observar que quando essa função possui um elevado numero de caracteres, sua visualização em uma caixa de texto se torna lenta.

Definição da tabela verdade: Para aplicarmos o algoritmo de redução de expressões booleanas, primeiro precisamos montar a tabela verdade da função original.

O numero de variáveis booleanas dessa tabela é correspondente ao numero de atributos condicionais que se diferem pelo menos uma vez dos outros atributos, ou seja, que não seja igual para todas as classes de equivalência.

Duas funções são responsáveis pela criação da tabela verdade:

Uma que retorna a quantidade de elementos diferentes, para podermos saber a quantidade das linhas que iremos gerar na tabela verdade, e a outra que retorna os elementos que diferem, para podermos montar as colunas da tabela verdade.

A tabela verdade é criada em tempo de execução. Seu nome é composto pelo nome do SI, o símbolo "_", o código do SI e por fim a terminação "_tabela_ verdade".

Seus atributos são os seguintes:

- A letra "m" que representa a linha da tabela verdade;

- Todos os atributos condicionais que diferem pelo menos uma vez entre as classes de equivalência;

- O minitermo de cada linha, ou seja, a linha inteira representada com números binários;

- Usamos uma ordem de agrupamento, que indicará a quantidade de ‘1s’ em cada linha. Esse atributo irá ser usado posteriormente, na redução da função de discernibilidade;

- O resultado da função para aquela linha.

Uma vez definido os atributos da tabela, podemos gerar as linhas da tabela verdade, os minitermos e a ordem de agrupamento. Uma vez terminado todo o processamento, chamamos a função que calculará o resultado da função para cada linha.

Cálculo do resultado da função de discernibilidade: Para todas as linhas, precisamos gerar seu resultado. Então o programa irá recuperar a função de discernibilidade, separar termo a termo da função e calcular seu resultado.

Sabemos que o tipo da função de discernibilidade é uma soma de produtos (SOP), então, como os termos da função são um “ou” lógico entre todos os elementos contidos nele, basta que tenha somente um ‘1’ para que o resultado da função seja ‘1’, caso contrário será ‘0’.

Assim montamos todos os resultados dos termos e armazenamos em uma variável. Depois de percorrermos todos os termos da função de discernibilidade, temos que calcular o resultado da função inteira.

Sabemos que a função é uma SOP, então todos os termos são um “e” lógico entre eles mesmos. Tendo essa informação, basta que um elemento da função tenha ‘0’ como resultado para que a função seja ‘0’. Ou seja, a função só vai ter ‘1’ como resultado se todos os termos tiverem como resultado ‘1’.

Uma vez percorrida todas as linhas da tabela verdade e calculando seus resultados, estamos prontos para ir para a próxima etapa.

Definição da função canônica: Precisamos definir todas as linhas da tabela verdade que serão usadas no cálculo da simplificação da função. Para isso, usamos a forma canônica da função, que é escrita na seguinte forma:

$$F(variáveis) = \sum m \, (minitermos),$$

No qual as variáveis representam todos os atributos condicionais contidos na tabela verdade, e os minitermos representam todas as linhas da tabela verdade cujo resultado é igual a ‘1’.

Para montar essa função, o programa recupera todos os elementos presentes na função de discernibilidade para montar as variáveis. Logo após, temos que recuperar na tabela verdade, todas as linhas cujo atributo “função” seja igual a ‘1’, e unir a função canônica.

Após esses passos, estamos prontos para aplicar o algoritmo de Quine– McCluskey.

O programa adotou os seguintes passos principais para a aplicação do algoritmo de Quine-McCluskey:

Agrupar minitermos: o processo de agrupamento de minitermos de acordo com a quantidade de ‘1’s presentes em suas linhas foi facilitado quando criamos na tabela verdade o atributo “ordem_agrupamento”, pois ele indica quantos ‘1’s estão presentes na linha.

Uma variável chamada *Miniterms* é um *Arraylist* responsável por armazenar todos os minitermos em sua forma binária, o nome de suas linhas e por fim, um campo que indicará se um minitermo foi combinado ou não; precisamos dessa informação para encontrar os primos implicantes.

Cada informação no *Arraylist* também é uma lista, então estaremos preenchendo uma lista de listas.

O primeiro passo do agrupamento é recuperar todas as linhas descritas na forma canônica da função de discernibilidade, pois elas representam os minitermos na tabela verdade.

Depois disso, percorremos a tabela verdade para identificar os minitermos descritos em bytes, sua linha e sua ordem de agrupamento, que servirá para ordenar a lista. A lista então ficará ordenada de acordo com a quantidade de ‘1’s por minitermo.

Por fim, inserimos na lista os minitermos em bytes, a linha e por padrão o valor “*unchecked*” para o campo de combinação do minitermo de acordo com a ordem de agrupamento.

Por exemplo, podemos ter três minitermos com quantidade de ‘1’s igual a um, cinco minitermos com quantidade igual a dois e dois minitermos com quantidade igual a três respectivamente. Então o *Arraylist* será preenchido com da seguinte forma:

A primeira posição do *Arraylist* irá conter os três primeiros minitermos, a segunda posição os cinco minitermos e por fim, a terceira posição os dois minitermos.

Quando terminamos de percorrer todas as ordens de agrupamento das linhas recuperadas da função canônica, temos agrupamento de minitermos que serão combinados na próxima etapa do algoritmo.

Encontrar Primos implicantes: ao iniciar esse processo, temos que primeiro estabelecer algumas condições de parada do loop que percorrerá o *Arraylist,* pois ele irá crescer de acordo com novos agrupamentos formados, ou seja, para cada combinação encontrada, iremos inserir essa combinação no final da lista.

Por definição, temos que encerrar as buscas quando não for possível realizar mais nenhum agrupamento. Porém temos que ter em mente que usaremos dois *iterators* para percorrer o *Arraylist*, um do minitermo de ordem menor, e outro do minitermo de ordem maior, então devemos tomar cuidado com dois aspectos:

- Não podemos correr o risco de comparar o ultimo minitermo da lista com o próximo elemento que já é um novo agrupamento. Para contornar essa situação, devemos marcar a passagem de uma etapa de agrupamento para a outra, ou seja, quando passamos a percorrer os agrupamentos gerados a partir do anterior, porque devemos sempre pular mais uma posição dos dois *iterators,* para recuperar a posição correta; para marcarmos a passagem de etapas, o programa conta a quantidade de vírgulas presentes nas linhas dos minitermos que estão sendo comparados. Caso sejam diferentes, sabemos que uma etapa foi pulada;

- Devemos ter o cuidado de observar que sempre o segundo *iterator* estará um passo a frente do primeiro, então sempre devemos verificar se esse próximo agrupamento de minitermos existe para continuarmos. Caso não exista, significa que a busca acabou, pois não existem mais combinações possíveis.

Mesmo usando os *iterators*, devemos registrar também os índices da lista, para a cada nova iteração podermos situar os *iterators* na posição correta, pois como a lista cresce, eles perdem a referência. Precisamos manter os índices também para podermos marcar na lista os minitermos que já foram combinados.

Quando estamos comparando dois minitermos, primeiro devemos verificar se eles diferem somente em uma posição. Caso afirmativo, podemos fazer os seguintes passos:

- Fazer a marcação "*checked*" na lista na posição dos dois minitermos correntes usando o índice, caso ainda não tenha sido marcada;

- Guardamos um dos minitermos em bytes de um dos elementos comparados, recuperamos a posição do bit que difere entre os minitermos e por fim substituímos pelo caractere "_";

- Verificamos se esse novo minitermo já não existe na lista. Caso afirmativo, não iremos inseri-lo;

- Caso o novo minitermo não exista, podemos formar a linha do novo minitermo, unindo as linhas dos minitermos e separando-as por vírgula.

Fazemos esses passos até percorrermos todos os minitermos adjacentes ao primeiro *iterator,* armazenando em outras listas os resultados obtidos em cada comparação, e adicionando-os à lista principal no final das iterações entre os dois minitermos.

O *loop* é finalizado quando não temos mais nenhuma combinação possível de minitermos.

Criar a tabela de Quine-McCluskey: seguindo o roteiro, essa etapa irá ser importante para definirmos os primos implicantes que farão parte da função simplificada. A tabela será preenchida usando todos os primos implicantes encontrados na fase anterior.

Essa tabela será criada em tempo de execução, pois não sabemos quantos serão os minitermos.

O nome da tabela é composto pelo nome do SI, o símbolo "_", o código do SI e por fim a terminação "_quine_mccluskey".

Os atributos da tabela são os seguintes:

- O termo, que indicará a linha do minitermo marcado como primo implicante;

- A letra 'm' concatenada com a linha do minitermo. Incluiremos esse atributo para todas as linhas do minitermo. O tipo desses atributos será booleano;

  Um exemplo: Se temos as linhas 3, 5, 15, 22 como minitermos, os atributos terão os seguintes nomes: "m3", "m5", "m15" e "m22";

- Um atributo do tipo booleano para indicar se o primo implicante foi escolhido ou não;

- O minitermo do primo implicante.

Criamos também uma tabela auxiliar que contém os minitermos usados na tabela e um atributo indicando se ele foi coberto ou não, porque precisamos cobrir todos os minitermos presentes na tabela.

Antes de iniciar as inserções na tabela de quine_mccluskey, precisamos definir essa tabela de cobertura de termos.

O processo irá então percorrer a lista principal para encontrar todos os minitermos que não foram marcados como combinados.

Nessa busca, iremos inserir os registros na tabela de quine_mccluskey, incluindo na tabela a linha encontrada no atributo “termo”, marcando como “*true*” todas as linhas presentes nesse termo e por fim, incluindo o minitermo em bytes encontrado.

Criar a função reduzida: para esse processo, precisamos definir a condição de término como obter todas as linhas dos primos cobertas, ou seja, para cada primo implicante escolhido, marcamos as linhas que ele cobre, até que todas estejam cobertas. O processamento termina quando a quantidade de elementos a cobrir é igual à quantidade de elementos cobertos.

O primeiro passo é identificar os primos essenciais, ou seja, os primos cuja linha é marcada uma única vez em todos os primos implicantes.

Uma vez identificados, podemos marcar na tabela de quine_mccluskey o termo como escolhido como “*true*” e na tabela auxiliar de cobertura de termos, podemos marcar as linhas cobertas como “*true*”.

Depois de encontrar todos os primos essenciais, precisamos verificar se eles já foram capazes de cobrir todos os termos. Caso afirmativo, já temos o conjunto de primos implicantes para formar a função reduzida. Caso contrário, precisamos percorrer a tabela de quine_mccluskey para cobrir as outras linhas.

O programa não se preocupou em utilizar heurísticas para encontrar os melhores conjuntos de atributos para cobrir as linhas que restam ser cobertas, pois esse não é o foco do projeto. Porém o programa escolhe os primos que são capazes de cobrir o maior numero de linhas possíveis por iteração, marcando nas tabelas de quine_mccluskey e cobertura de termos que o termo e a linha foram cobertos, respectivamente.

Quando temos todas as linhas cobertas, estamos prontos para reduzir a função de discernibilidade.

O que devemos fazer nessa etapa é recuperar os minitermos em bytes dos primos implicantes escolhidos e associá-los às variáveis booleanas na tabela verdade, para recuperarmos os nomes dos atributos condicionais que cada bit representa.

A função simplificada terá a forma de Produto de somas (POS), onde os produtos são todos os minitermos em bytes encontrados, onde cada bit do minitermo é convertido para seu atributo condicional correspondente.

Com a função simplificada definida, podemos percorrê-la termo a termo para gerar para cada um deles uma classe de equivalência e a relação de discernibilidade, e comparar essa relação com a relação de indiscernibilidade do conjunto original, já definida no começo da aplicação do algoritmo.

A idéia é encontrar um termo que seja menor que o conjunto de atributos condicionais definidos no início, e que tenha a relação de indiscernibilidade idêntica à relação do desse conjunto já definido.

Se encontrado, esse termo será considerado um reduto do nosso sistema de informação. Ele é gravado na tabela que contém a função de discernibilidade do SI.

3.4.5. Geração de Regras

Essa é a última parte do algoritmo, que se resume em apresentar as classes de equivalência do conjunto de atributos condicionais reduzidos e seus respectivos atributos de decisão.

Todas as classes são apresentadas nas regras, porém somente as regras determinísticas são válidas, ou seja, as regras que estão na aproximação inferior do conjunto, pois podem existir classes de equivalência iguais para atributos de decisão diferentes.

Podemos provar que o reduto encontrado é de fato o menor possível definindo um novo SI contendo os atributos condicionais presentes no próprio reduto. Então percorremos todas as fases do programa até a nova redução do sistema. Teoricamente, não pode haver outro conjunto mínimo desse novo SI, então as regras desse permanecerão as mesmas.

3.4.6. Bases de dados

O programa trabalha com duas bases de dados distintas:

Temos uma base de dados que é responsável por gerenciar todas as etapas do algoritmo, armazenando todas as informações geradas pelo método de *Rough Sets*. Seu nome padrão é definido como “RS”.

A outra base de dados é a base de dados da aplicação, ou seja, o *Data Warehouse* que iremos aplicar o algoritmo.

Ela é configurada nos parâmetros do banco de dados, onde informamos o local da base, o usuário, seu nome e a senha do banco.

3.4.7. Quantidade de atributos condicionais permitidos

Podemos escolher quantos atributos condicionais forem necessários para a aplicação, o que torna o programa mais específico para certos tipos de situações que o usuário não deseja levar em consideração todos os atributos contidos no *Data Warehouse*.

Na verdade isso seria desnecessário e não faria sentido algum incluir atributos de controle nos atributos condicionais, como código de tabelas.

O algoritmo de Quine-McCluskey possui uma boa solução computacional por ter seu método como tabular, e por permitir mais variáveis do que o mapa de Karnaugh.

Porém o algoritmo possui certa limitação quanto ao número de variáveis booleanas utilizadas na função de discernibilidade, em decorrência do tempo necessário para reduzir uma função com um número expressivo de variáveis

. Essas variáveis booleanas descritas na função de discernibilidade estão relacionadas com os atributos condicionais escolhidos nessa etapa, pois as essas variáveis podem representar todos os atributos condicionais, caso dado atributo escolhido não seja igual para todas as classes de equivalências criadas.

A complexidade do algoritmo cresce exponencialmente quando aumentamos a quantidade de variáveis usadas na expressão, aumentando assim o número de implicantes primos, que normalmente já é grande.

Capítulo 4

Conclusão

Em todo o desenvolvimento do projeto, tanto da parte teórica quanto da implementação, o objetivo final foi reduzir um conjunto de atributos de um *Data Warehouse*.

O PostgreSQL conta com algumas características que favorecem um DW, porém sua essência é objeto-relacional, e isso o torna limitado ao realizar certos tipos de consultas em tabelas fato muito extensas.

Ao fim do projeto, podemos ver o objetivo completo, pois ao definir uma base de dados atendendo as especificações descritas na Seção 3.3.4, conseguimos reduzir os atributos definidos para essa base de dados, caso seja possível para o conjunto escolhido.

Mesmo encontrando algumas restrições de uso dos dois algoritmos principais utilizados no projeto, essas restrições puderam ser contornadas com o foco na redução do sistema de informação. Porém com um novo foco de projeto, podemos melhorar tempos computacionais e armazenamentos de dados.

4.1. Trabalhos futuros

4.1.1. Faixa de valores dos atributos condicionais

Conforme citado na Seção 3.3.2, podemos definir faixas de valores para os atributos condicionais, que serão utilizados para definirmos as equivalências. Porém, podemos ter certos tipos de situações onde precisamos estabelecer uma faixa de valores do tipo temporal para definir as classes de equivalência.

Esses valores podem ser representados pela dimensão de tempo em todas as suas hierarquias, para que possamos definir, por exemplo, uma faixa de valores de um trimestre, uma semana, alguns dias, enfim, o que for necessário para a cobertura completa de dimensões e medidas pelo programa.

4.1.2. Níveis de hierarquia das dimensões na escolha dos atributos condicionais

O programa contém uma restrição a respeito de níveis de hierarquia dentro de uma dimensão.

Não foi desenvolvida uma estrutura no programa que fosse capaz de adentrar as hierarquias das dimensões o quanto fosse necessário para possibilitar a escolha de qualquer um de seus atributos como um atributo condicional, pois o foco do programa se limitou a utilizar os atributos mais próximos da tabela fato.

Na verdade, podemos descer no máximo uma hierarquia da tabela fato, ou seja, não é possível escolhermos algum atributo da dimensão que está situado em hierarquias inferiores.

Essa estrutura poderá ser desenvolvia em trabalhos futuros, para que possamos ter um alcance completo de todos os atributos ligados às dimensões da tabela fato.

4.1.3. Simulação dos atributos de decisão

Citamos na Seção 3.3.3 que o programa é capaz de simular atributos de decisão dentro da tabela fato do *Data Warehouse*, para fins de experimentos de variações das aproximações de conjuntos.

Para melhor atender a experimentos, poderíamos pensar em certas regras de definição de decisões na tabela fato, atendendo certos tipos de situações na aplicação para que pudéssemos simular esses atributos de forma precisa. Porém essa simulação não foi o foco do objetivo proposto, podendo ser observado em trabalhos futuros.

4.1.4. Restrição no armazenamento da matriz de discernibilidade

O armazenamento da matriz de discernibilidade em sua forma original acaba se tornando um grande problema, pois o PostgreSQL possui uma restrição em relação ao tamanho da tabela na relação linha x coluna.

Temos as seguintes características no PostgreSQL:

- Tamanho máximo da base de dados: Ilimitado;
- Tamanho máximo de uma tabela: 32 TB;
- Tamanho máximo de uma linha: 1.6 TB;
- Tamanho máximo de um campo: 1 GB;

- Quantidade de linhas por tabela: Ilimitado;

- Quantidade máxima de colunas por tabela: 250 – 1600, dependendo do tipo da coluna;

- Tamanho máximo de índices por tabela: Ilimitado.

Analisando esses dados, confirmamos que não podemos armazenar uma matriz com mais de 1600 colunas. E como o tamanho da matriz de discernibilidade cresce proporcionalmente com a quantidade de registros da tabela fato, o número de registros teria que ser muito limitado, em se tratando de um *Data Warehouse*.

Então o armazenamento da matriz de discernibilidade no banco de dados se torna inviável.

Podemos pensar em outro tipo de armazenamento da matriz de discernibilidade, para que não tenhamos que processar uma matriz de discernibilidade já processada anteriormente para definirmos um reduto de um sistema de informação.

Referências bibliográficas

BERKUS, JOSH, Data Warehousing with PostgreSQL. Disponível em < http://pgexperts.com/document.html?id=49>. Acesso em 11 de Novembro de 2011

CRENSHAW, JACK, All about Quine–McCluskey. Disponível em <http://www.eetimes.com/discussion/programmer-s-toolbox/4025004/All-about-Quine-McClusky>. Acesso em 11 de Novembro de 2011

FILETO. Renato. Data Warehouse: Fundamentos, Ferramentas e Tendências Atuais Disponível em - < http://www.inf.ufsc.br/~fileto/Talks/IntrodDW-ERBD2009.pdf.pdf>. Acesso em 10 de março de 2010.

KATEKAWA, BRUNO - Método Quine-McCluskey – TABULAÇÃO. Disponível em <http://www.google.com.br/url?sa=t&rct=j&q=m%C3%A9todo%20quine-mccluskey%20-%20tabula%C3%A7%C3%A3o&source=web&cd=1&ved=0CB4QFjAA&url=http%3A%2F%2Fwww2.comp.ufscar.br%2F~bruno_katekawa%2FCircuitos%2520Digitais%2FAula%252005%2FM%25E9todos%2520de%2520Minimiza%25E7%25E3o%2520de%2520Circuitos%2520-%2520Tabula%25E7%25E3o%25202.doc&ei=HG_ITsCFGsba0QH0la3LBQ&usg=AFQjCNF5Y6_nlMh6zvhFtDfZ5xzxF46-WA&cad=rja>. Acesso em 1 de Novembro de 2011.

MACHADO, Felipe Nery Rodrigues. Projeto de Data Warehouse: Uma Visão Multidimensional. São Paulo: Érica, 2000. 252 p

MENGUE, FABIO – Curso de Java básico. Disponível em <ftp://ftp.unicamp.br/pub/apoio/treinamentos/linguagens/java_basico.pdf>. Acesso em 25 de novembro de 2011.

MUSSI, CAMILO – Data Warehouse – Da modelagem a implantação. Disponível em <http://www.fag.edu.br/professores/limanzke/Tecnologia%20da%20Informa%E7%E3o/DA%20TA%20WAREHOUSE.pdf>. Acesso em 20 de Outubro de 2010

PATRICIO, Cristian Mara M.M. PINTO, João Onofre Pereira. SOUZA, Celso Correia. Rough Sets, técnica de redução e geração de regras para classificação de dados. Disponível em <http://www.sbmac.org.br/eventos/cnmac/cd_xxviii_cnmac/resumos%20estendidos/cristian_patricio_ST18.pdf>. Acesso em 10 de Junho de 2011

Apêndice A – Definição da tabela das classes de equivalência

```
Public classAtributos_Condicionais {
Private int codigo;
Private int seq;
Private int tipo;
Private String referencia;
Private String atributo;
   // 0 - atributo qualitativo
   // 1 - atributo quantitativo
Private int cod_si;
Private Float faixa_inicial;
Private Float faixa_final;
Public Atributos_Condicionais (int codigo, int seq, int cod_si, String referencia, String atributo, int tipo, Float faixa_inicial, Float faixa_final) {
this.codigo = codigo;
this.seq = seq;
this.cod_si = cod_si;
this.referencia = referencia;
this.atributo = atributo;
this.tipo = tipo;
this.faixa_inicial = faixa_inicial;
this.faixa_final = faixa_final;
}
```

}

public void create_table (String vNomeTabela, Boolean vAll, String vAtributosParametros) {

// vNomeTabela se refere ao nome do SI definido, que será o nome da tabela

// vAll nos informa se a tabela será criada com todos os atributos condicionais escolhidos.

// Essa opção será usada quando criamos uma tabela temporária para os possíveis redutos do Sistema de Informação

// vAtributosParametros será usado quando precisamos restringir os atributos que serão criados para a tabela

Connection con;

PreparedStatement vStmt = null;

Atributos_Condicionais vItem = null;

Iterator it;

this.Search(vAll, vAtributosParametros); //atribui a variável da classe vLista todos os atributos condicionais

it = vLista.iterator();

try {

con = ConnectionFactory.getConnection();

//deleta a tabela temporária caso ela exista

if (!vAll) {

drop_table ("tmp_" + vNomeTabela);

}

/*--------CRIA A TABELA DE CLASSES DE EQUIVALÊNCIA-------------*/

//cria a tabela com o nome do sistema de informação (o primeiro elemento é o codigo da classe de equivalencia)

if (vAll) {

```
vSQL = "CREATE TABLE " + vNomeTabela + "( Cod_classe_equivalencia integer NOT NULL, 
";

} else {

vSQL = "CREATE TABLE tmp_" + vNomeTabela + "( Cod_classe_equivalencia integer NOT 
NULL, ";

}           //percorre os atributos da lista para gerar a tabela;

while (it.hasNext()) {

vItem = new Atributos_Condicionais((Atributos_Condicionais) it.next());

              //nome do campo

vSQL = vSQL + vItem.getReferencia() + "_" + vItem.getAtributo();

vSQL = vSQL + " character varying(200), ";

          }

if (vAll) {

vSQL = vSQL + " CONSTRAINT " + vNomeTabela + "_pk PRIMARY KEY 
(cod_classe_equivalencia))";

          } else {

vSQL = vSQL + " CONSTRAINT tmp_" + vNomeTabela + "_pk PRIMARY KEY 
(cod_classe_equivalencia))";

          }

vStmt = con.prepareStatement(vSQL);

vStmt.executeUpdate();

vStmt.close();

con.close();

       } catch (SQLException e) {

JOptionPane.showMessageDialog(null, e.getMessage());

}

}
```

Apêndice B – Definir classes de equivalência

```
/*--------------------------------SQL PRINCIPAL QUE PERCORRE A TABELA FATO DO DW---------
-----------*/
vSQL = "SELECT ";

while (i.hasNext()) { //percorre todos os atributos condicionais

        vItem = new tmp_Valores_Atributos((tmp_Valores_Atributos) i.next());
//atributo condicional corrente

        //verifica se o atributo é uma medida da tabela fato, ou se é uma dimensão; caso for
uma medida, não usaremos a terminação “_id”.

        if (!vItem.getReferencia().equals(vFato.toString())) {

                vSQL = vSQL + vItem.getReferencia() + "_id ,";
        }else {
vSQL = vSQL + vItem.getReferencia() + ".* ,";
}
}
        //tira a ultima virgula
        vSQL = vSQL.substring(0, vSQL.length() - 1);

//vFato se refere ao nome da tabela fato
vSQL = vSQL + " FROM " + vFato + " ORDER BY " + vFato + "_id";

vStmt = conApp.prepareStatement(vSQL);
rs = vStmt.executeQuery();
```

APÊNDICE C – Aproximações de conjuntos

```
Public List <Integer> DefineAproximacaoInferior() {
Connection con;
PreparedStatement vStmt;
ResultSet rs;
List<Integer> vRegistros;
vRegistros = new ArrayList<Integer>();
try {
con = ConnectionFactory.getConnection();
//Pega todas as classes que possuem o atributo de decisão escolhido
        //E que não possuem atributos de decisão diferentes para a mesma classe de
equivalencia
vSQL = "SELECT tabela_fato_id "
            + "FROM aproximacao_Conjuntos WHERE atributo_decisao = '" + vX + "'";
vSQL = vSQL + " AND classe_equivalencia_id NOT IN (";
vSQL = vSQL + " SELECT classe_equivalencia_id FROM aproximacao_Conjuntos"
            + " WHERE atributo_decisao <> '" + vX + "'";
vSQL = vSQL + " ORDER BY classe_equivalencia_id )";

        //  vSQL = vSQL + " GROUP BY classe_equivalencia_id HAVING COUNT(*) > 1 "
        //          + " ORDER BY classe_equivalencia_id )";
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
while (rs.next()) {
vRegistros.add(rs.getInt("tabela_fato_id"));
}
        }
rs.close();
con.close();
vStmt.close();
      } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
```

```
Return vRegistros;
}



Public List <Integer> DefineAproximacaoSuperior() {
Atributos_Decisao_DAO vItem;
Connection con;
PreparedStatement vStmt;
ResultSet rs;
List<Integer> vRegistros;
vRegistros = new ArrayList<Integer>();
String vDecisao = "";
try {
con = ConnectionFactory.getConnection();

//Pega todos os classes que possuem o atributo de decisão escolhido
//E que podem possuir atributos de decisão diferentes para a mesma classe de equivalencia
vSQL = "SELECT DISTINCT classe_equivalencia_id "
        + "FROM aproximacao_Conjuntos WHERE atributo_decisao = '" + vX + "'";
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
while (rs.next()) {
vDecisao = vDecisao + rs.getInt("classe_equivalencia_id") + ",";
}
}
if (!vDecisao.equals("")) {
vSQL = "SELECT tabela_fato_id FROM aproximacao_Conjuntos "
+ "WHERE classe_equivalencia_id IN (" + vDecisao.substring(0, vDecisao.length() - 1) + ")";
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
while (rs.next()) {
vRegistros.add(rs.getInt("tabela_fato_id"));
}
```

```
        }
      }

rs.close();
con.close();
vStmt.close();
    } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
Return vRegistros;
}


Public List <Integer> DefineRegiaoFronteira() {
Atributos_Decisao_DAO vItem;
Connection con;
PreparedStatement vStmt;
ResultSet rs;
ResultSet rs1;
List <Integer> vRegistros;
vRegistros = new ArrayList<Integer>();
int vCountIgual = 0;
int vCountDiferente = 0;
try {
con = ConnectionFactory.getConnection();
//Pega todos as classes que se repetem
vSQL = "SELECT classe_equivalencia_id FROM aproximacao_Conjuntos";
vSQL = vSQL + " GROUP BY classe_equivalencia_id HAVING COUNT(*) > 1 "
        + " ORDER BY classe_equivalencia_id";
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
while (rs.next()) {
//retorna a quantidade de classes que diferem no atributo de decisão
        //se tiver mais que um atributo de decisão para a mesma classe, esse registro está
na região de fronteira
```

```
		//primeiro a quantidade de registros diferentes do atributo
vSQL = "SELECT COUNT(*) AS contagem FROM aproximacao_Conjuntos "
			+ " WHERE classe_equivalencia_id = " + rs.getInt("classe_equivalencia_id")
			+ " AND atributo_decisao<> '" + vX + "'";
vStmt = con.prepareStatement(vSQL);
rs1 = vStmt.executeQuery();
if (rs1 != null) {
while (rs1.next()) {
vCountDiferente = rs1.getInt("contagem");
}
}
rs1 = null;
		//seguindo a quantidade de registros iguais ao atributo
vSQL = "SELECT COUNT(*) AS contagem FROM aproximacao_Conjuntos "
			+ " WHERE classe_equivalencia_id = " + rs.getInt("classe_equivalencia_id")
			+ " AND atributo_decisao = '" + vX + "'";
vStmt = con.prepareStatement(vSQL);
rs1 = vStmt.executeQuery();
if (rs1 != null) {
while (rs1.next()) {
vCountIgual = rs1.getInt("contagem");
}
}


		//Se tiver pelo menos um igual e pelo menos um diferente do atributo de decisão, significa que está na região de fronteira
if (vCountIgual> 0 &&vCountDiferente> 0) {
vSQL = "SELECT tabela_fato_id FROM aproximacao_Conjuntos WHERE classe_equivalencia_id = " + rs.getInt("classe_equivalencia_id");
vStmt = con.prepareStatement(vSQL);
			rs1 = vStmt.executeQuery();
if (rs1 != null) {
while (rs1.next()) {
vRegistros.add(rs1.getInt("tabela_fato_id"));
}
			}
```

```
                    }

                }
            }
rs.close();
con.close();
vStmt.close();
        } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
Return vRegistros;
}

Public List <Integer> DefineForaRegiao() {
Atributos_Decisao_DAO vItem;
Connection con;
PreparedStatement vStmt;
ResultSet rs;
List<Integer> vRegistros;
vRegistros = new ArrayList<Integer>();
try {
con = ConnectionFactory.getConnection();

//Pega todos as classes que não possuem o atributo de decisão escolhido.
vSQL = "SELECT tabela_fato_id "
                    + "FROM aproximacao_Conjuntos "
                    + "WHERE classe_equivalencia_idnot in"
                    + "(SELECT classe_equivalencia_id FROM aproximacao_Conjuntos"
                    + " WHERE atributo_decisao = '" + vX + "')";

vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
```

```
if (rs != null) {
while (rs.next()) {
vRegistros.add(rs.getInt("tabela_fato_id"));
}
            }
rs.close();
con.close();
vStmt.close();
        } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
Return vRegistros;
}
```

Apêndice D – Matriz de discernibilidade

```
public class Matriz_Discernibilidade {

public String vNomeTabela;
private String vSQL;
public long vCount;

public void DefineMatriz(intvCount) {
//vCount - dimensões da matriz
Classes_EquivalenciavConexao;
tmp_Valores_AtributosvAtributos = new tmp_Valores_Atributos();
Connection con;
PreparedStatement vStmt = null;
ResultSet rs;
ResultSet rs1;
Iterator i;
Iterator i2;
Iterator i3;
Iterator i4;
vConexao = new Classes_Equivalencia();
vConexao.vNomeTabela = this.vNomeTabela;
vAtributos.setCampos(vConexao.DefineColunas()); //recupera o nome de todos os atributos condicionais
String vAtributosDiferentes = new String();
String vColuna;
Int vTipoAtributo;
Float vFaixaInicial;
Float vFaixaFinal;
Int vPos;
Float vValorClasseMenor;
Float vValorClasseMaior;

 //variavel global
RoughSets.vMatrizInd = new String[vCount][vCount];
```

```
Int vLinhaMatriz = 0;
Int vColunaMatriz = 0;

            //inicia a matriz
RoughSets.vMatrizInd[0][0] = null;

//PERCORRE A TABELA DE CLASSES DE EQUIVALENCIA PARA ENCONTRAR OS
ATRIBUTOS DIFERENTES

try {
con = ConnectionFactory.getConnection();

          //Verifica o tipo, faixa inicial e faixa final de cada coluna da tabela e joga em outra lista
          i = vAtributos.getCampos().iterator();
          //pega os dados do atributo condicional
while (i.hasNext()) {
vColuna = new String();
vColuna = i.next().toString();
vPos = vColuna.indexOf(".");

              //separa a referencia do atributo

vSQL = "SELECT tipo,faixa_inicial,faixa_final FROM atributos_condicionais WHERE referencia
= '" + vColuna.substring(0, vPos) + "'";
vSQL = vSQL + " AND atributos = '" + vColuna.substring(vPos + 1, vColuna.length()) + "'";
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
if (rs.next()) {
vAtributos.getTipo().add(rs.getInt("tipo"));
vAtributos.getFaixa_inicial().add(rs.getFloat("faixa_inicial"));
vAtributos.getFaixa_final().add(rs.getFloat("faixa_final"));
}
vColuna = null;
}
          }
```

```
i = null;
rs = null;

        //SQL PRINCIPAL

vSQL = "SELECT * FROM " + vNomeTabela;
vStmt = con.prepareStatement(vSQL);
rs = vStmt.executeQuery();
if (rs != null) {
while (rs.next()) {

            //percorre os registros anteriores

vSQL = "SELECT * FROM " + vNomeTabela + " WHERE cod_classe_equivalencia< " +
rs.getInt("cod_classe_equivalencia") + "ORDER BY cod_classe_equivalencia";
vStmt = con.prepareStatement(vSQL);
            rs1 = vStmt.executeQuery();
if (rs1 != null) {
while (rs1.next()) {
vColunaMatriz = rs1.getInt("cod_classe_equivalencia") - 1;
                i = vAtributos.getCampos().iterator();
i2 = vAtributos.getTipo().iterator();
i3 = vAtributos.getFaixa_inicial().iterator();
i4 = vAtributos.getFaixa_final().iterator();
vAtributosDiferentes = "";
                //percorre os atributos e seus tipos
while (i.hasNext() && i2.hasNext() && i3.hasNext() && i4.hasNext()) {

//dados correntes
vColuna = i.next().toString().replace(".", "_"); //nome da coluna
vTipoAtributo = Integer.parseInt(i2.next().toString());
vFaixaInicial = Float.parseFloat(i3.next().toString());
vFaixaFinal = Float.parseFloat(i4.next().toString());

//SE FOR DIFERENTE, INSERE NA VARIAVEL DE ATRIBUTOS DIFERENTES
if (vTipoAtributo == 0) {
```

```
if (!rs.getString(vColuna).equals(rs1.getString(vColuna))) {
vAtributosDiferentes = vAtributosDiferentes + vColuna + ",";
}
} //quantitativo
else {
vValorClasseMenor = Float.parseFloat(rs1.getString(vColuna)); //valor da coluna
vValorClasseMaior = Float.parseFloat(rs.getString(vColuna));
                //se estiver na faixa de valores, significa que o valor é igual
                //nesse caso, temos que fazer uma segunda verificação para verificar se
o valor é igual

                //os dois não estão na faixa
if ((vValorClasseMenor<vFaixaInicial || vValorClasseMenor>vFaixaFinal) ||
(vValorClasseMaior<vFaixaInicial || vValorClasseMaior>vFaixaFinal)) {
if (vValorClasseMenor != vValorClasseMaior) {
vAtributosDiferentes = vAtributosDiferentes + vColuna + ",";
}
}
}

               /*------------------------------------------------------------------*/
}
            //INSERE NA TABELA OS CAMPOS DIFERENTES
               /*------------------------------------------------------------------*/
            //verifica se já existe a classe, para inserir ou atualizar

vAtributosDiferentes = vAtributosDiferentes.substring(0, vAtributosDiferentes.length() - 1);
RoughSets.vMatrizInd[vLinhaMatriz][vColunaMatriz] = vAtributosDiferentes;
i = null;
 i2 = null;
i3 = null;
i4 = null;
}
rs1.close();
}
vLinhaMatriz++;
          /*-----------------------------PROCESSAMENTO--------------------------*/
```

```
}
    }

rs.close();
vStmt.close();
con.close();


  } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
}
```

Apêndice E – Função de discernibilidade

```
Public String DefineFuncaoDiscernibilidade() {

Connection con;
PreparedStatement vStmt = null;
ResultSet rs;
Classes_Equivalencia vItem = new Classes_Equivalencia();
vItem.vNomeTabela = vTabela;

//retorna a quantidade de registros na tabela das classes de equivalência
Int qtdeClasses = vItem.RetornaCount();
String vFuncao = new String();
vFuncao = "";
vCelula = "";
int i;
int j;


try {
con = ConnectionFactory.getConnection();

        //começa da linha 1
for (i = 0; i <qtdeClasses; i++) {
for (j = i + 1; j <qtdeClasses; j++) {    //começa da coluna posterior a linha
vFuncao = vFuncao + "(" + roughsets.RoughSets.vMatrizInd[j][i].replace(",", "+") + ") . "; //ou
logico entre a célula e o e logico entre as linhas
}
}

       //GRAVA A FUNÇÃO DE DISCERNIBILIDADE
if (!vFuncao.equals("")) {
         //tira o ultimo produto
vFuncao = vFuncao.substring(0, vFuncao.length() - 3);
```

```
vSQL = "DELETE FROM funcao_discernibilidade WHERE cod_si = " +
vTabela.substring(vTabela.length() - 3, vTabela.length());
vStmt = con.prepareStatement(vSQL);
vStmt.executeUpdate();

vSQL = "INSERT INTO funcao_discernibilidade VALUES (" +
vTabela.substring(vTabela.length() - 3, vTabela.length()) + ",'" + vFuncao + "',",")";
vStmt = con.prepareStatement(vSQL);
vStmt.executeUpdate();
        }
vStmt.close();
con.close();
      } catch (SQLException e) {
JOptionPane.showMessageDialog(null, e.getMessage());
}
returnvFuncao;

}
```

Apêndice F – Encontrar redutos

```
//parte final do algoritmo
//percorre todos os termos da função reduzida
p1 = vP1.iterator(); //parenteses inicial
p2 = vP2.iterator(); //parenteses final

while (p1.hasNext() && p2.hasNext()) {
//pega os atributos entre os parenteses, já preparado para comparar no sql

par1 = Integer.parseInt(p1.next().toString());
par2 = Integer.parseInt(p2.next().toString());

vTermo = vFuncaoReduzida.substring(par1 + 1, par2); //pega o termo da função

vT.create_table(vTabela, false, vTermo); //cria uma tabela temporária de equivalência para cada termo da função reduzida

vClasses.Define_Classes_Equivalencia(false, vTermoSQL); //define as classes para cada termo

vAproximacao.Cria_Tabela_Classes_AD(false, vTermoSQL); //cria a tabela de processamento do fato

vRelacoesIndiscernibilidade.add(vAproximacao.CriaRelacaoIndiscernibilidade(false, vTermo)); //retorna a relação de indiscernibilidade do termo

vTermosRedutos.add(vTermo); //lista com todos os termos

}

        //comparar todos os termos da função reduzida com a função original

vRelacaoIndiscernibilidadeOriginal = vAproximacao.CriaRelacaoIndiscernibilidade(true, "");
```

```
vQtdeMenorAtributos = vFuncaoReduzida.length(); //pega o tamanho da função inteira para inicializar

iRelacoes = vRelacoesIndiscernibilidade.iterator();
iTermos = vTermosRedutos.iterator();

vRedutoCorrente = "";
vTermo = "";

while (iRelacoes.hasNext() &&iTermos.hasNext()) {
//relação de indiscernibilidade e termos correntes
vRedutoCorrente = iRelacoes.next().toString();
vTermo = iTermos.next().toString();



if (vRedutoCorrente.equals(vRelacaoIndiscernibilidadeOriginal)) {

        //se é um reduto, verificar se é o menor
if (vTermo.length() <= vQtdeMenorAtributos) {
vReduto = vTermo;
vQtdeMenorAtributos = vReduto.length();
}
}
}

 //se não achou nenhum reduto
if (vQtdeMenorAtributos == vFuncaoReduzida.length()) {
vReduto = "Nao existem redutos para esse conjunto de informações!";
}
        else {
          //Atualiza o reduto
vSQL = "UPDATE funcao_discernibilidade SET reduto = '" + vReduto + "' WHERE cod_si = " + vTabela.substring(vTabela.length() - 3, vTabela.length());
vStmt = con.prepareStatement(vSQL);
vStmt.executeUpdate();
}
```