UNIVERSIDADE FEDERAL DA PARAÍBA - UFPB
CENTRO DE CIÊNCIAS E TECNOLOGIA - CCT
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA - COPIN

DISSERTAÇÃO DE MESTRADO

MATERIALIZAÇÃO/UTILIZAÇÃO EFICIENTES DE VISÕES EM DATA WAREHOUSES

MÁRCIO FARIAS DE SOUZA

CAMPINA GRANDE, JUNHO DE 1999

Márcio Farias de Souza

MATERIALIZAÇÃO/UTILIZAÇÃO EFICIENTES DE VISÕES EM DATA WAREHOUSES

Dissertação submetida ao Curso de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraíba, como requisito parcial para a obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação
Linha: Banco de Dados

Marcus Costa Sampaio
Orientador

Campina Grande, Junho de 1999

S729m Souza, Marcio Farias de
Materializacao/utilizacao eficientes de visoes em data warehouses / Marcio Farias de Souza. - Campina Grande, 1999.
64 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Banco de Dados 2. Data Warehousing 3. Visoes Materializadas 4. Dissertacao - Informatica I. Sampaio, Marcus Costa II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 004.65(043)

MATERIALIZAÇÃO/UTILIZAÇÃO EFICIENTES DE VISÕES EM DATA WAREHOUSES

MÁRCIO FARIAS DE SOUZA

DISSERTAÇÃO APROVADA EM 01.06.1999

PROF. MARCUS COSTA SAMPAIO, Dr.
Orientador

PROF. ULRICH SCHIEL, Dr.
Examinador

PROFª MARIA LUIZA MACHADO CAMPOS, Ph.D
Examinadora

CAMPINA GRANDE – PB

"Instruir-te-ei e ensinar-te-ei
o caminho que deves seguir."
Salmos 32:8

"Não te deixarei nem
te desampararei."
Hebreus 13:5

Resumo

Dada a complexidade de consultas a um Data Warehouse (DW), surge a necessidade de se pré-computar certas operações onerosas, com seus resultados, chamados de visões materializadas, ficando armazenados no DW. Desta forma, pode-se esperar que os tempos de resposta de consultas que utilizam visões materializadas serão aceitáveis, uma vez que uma parte de suas operações já foi realizada. A questão da complexidade é então transferida para o processo de materialização de visões, que tem sido objeto de muitas pesquisas. O objetivo básico de diversos algoritmos com esta finalidade é a materialização simultânea de várias visões que têm dimensões (ou critérios de agregação) comuns. O problema com esta abordagem é que ela pode impor uma ordem de armazenamento das visões capaz de onerar o tempo de processamento das consultas que as utilizam, anulando, ao menos parcialmente, as vantagens da materialização. Nesta dissertação, apresentamos um algoritmo para a materialização/utilização eficientes de visões, permitindo materializar simultaneamente diversas visões sem perder de vista os custos de processamento das consultas ao DW.

Abstract

Given the complexity of many queries over a Data Warehouse (DW), it is necessary to precompute and store in the DW the answer sets of some onerous operations, so called *materialized views*. With this approach, one expects that queries using some of these materialized views will have acceptable processing costs, since at least some of their operations have already been computed. The complexity is then transferred to the view materialization process, the subject of much research. The main objective of several proposed algorithms is to simultaneously materialize the views which have common dimensions (that is, common aggregate criteria). The problem with this approach is that it often imposes a storage order for the views to the detriment of the processing cost of the queries using these materialized views. In this dissertation, we present an algorithm which allows the materialization of several views simultaneously without loosing sight of query processing costs.

Agradecimentos

Ao meu bom Deus pela presença constante em todos os momentos da minha vida.

Aos meus pais José Gomes e Izete Farias pelo apoio incondicional ao filho.

À minha irmã Cynnara Farias pelo afeto e compreensão.

Ao Prof. Dr. Marcus Costa Sampaio pela orientação, apoio e amizade sincera que sempre prevaleceu em nosso relacionamento.

Ao Prof. Eduardo Veloso pela ajuda e sugestões sobre a utilização de algoritmos combinatórios neste trabalho.

Aos professores Ulrich Schiel e Maria Luíza Campos, pelas sugestões e incentivo.

Aos professores e funcionários da COPIN e do DSC, pelos conhecimentos transmitidos e apoio na execução deste trabalho especialmente à Prof. Joseluce Cunha de Farias.

Às amigas Josenita Ramos e Águeda Miranda da COPEX pela divulgação dos resultados gerados neste trabalho.

À amiga Gilene Fernandes pela ajuda e apoio no desenvolvimento deste trabalho.

À Empresa de Processamento de Dados da Previdência Social – DATAPREV, especialmente aos colegas do DEAT.E, pela compreensão na fase final deste trabalho e apoio extraordinário para a conclusão.

À CAPES pelo suporte financeiro durante todo o trabalho de Mestrado.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O âmago desta Dissertação é um algoritmo para criação/utilização eficientes de visões materializadas em bancos de dados de apoio à decisão. Nesta Introdução, fazemos considerações sobre bancos de dados de apoio à decisão e sua problemática, damos a motivação e os objetivos da Dissertação, e apresentamos a estrutura da Dissertação, descrevendo sucintamente cada um dos seus capítulos.

1.1 OLAP e Data Warehouse

Tomadores de Decisão ("Decision Makers") necessitam de informações integradas, provenientes de diversas fontes de dados. Para atendê-los, prover acesso integrado a múltiplos e heterogêneos bancos de dados e outras fontes de informação tornou-se uma das áreas prioritárias de pesquisa em banco de dados. Uma das soluções para este problema segue a abordagem seguinte: a informação das fontes é periodicamente extraída, traduzida, filtrada, integrada, agregada e armazenada em um repositório centralizado, o *Data Warehouse (DW)*; quando uma consulta requerendo dados integrados e agregados é submetida, esta é avaliada diretamente no DW, sem a necessidade, portanto, de ter acesso às fontes primitivas de informação, um processo muito oneroso.

Podemos observar na abordagem resumida acima que, durante a interação usuário-sistema, o DW não se encontrará atualizado *stricto sensu*. Longe de se constituir em um problema, isto é mesmo necessário, pois requer-se de um DW que ele tenha *consistência temporal*, isto é, durante uma sessão (longa) de interação usuário-sistema, o DW não deve mudar de estado.

Com relação aos modelos de dados para DW, a tecnologia relacional, devido ao seu alto grau de maturidade e disseminação, vem se impondo também neste campo. Entretanto, o

projeto de um DW relacional é bastante diferente do projeto de um banco de dados relacional tradicional. Um DW é normalmente um banco de dados muito grande, mesmo se o nível de agregação das informações nele contidas for alto, pois pode guardar históricos de até 10 anos. Séries históricas são um requisito essencial de aplicações OLAP ("On-Line Analytical Processing"), desenvolvidas para auxiliar o processo de tomada de decisão **[KIMBALL 96]**. Os grandes DWs para aplicações OLAP podem ter de centenas de gigabytes a terabytes de tamanho. Consultas OLAP são em geral complexas, podendo acessar milhares de registros. Reduzir os custos de consultas OLAP é pois um importante objetivo de pesquisa.

1.2 Uma Típica Sessão OLAP[1]

Sistemas de apoio à decisão são aqueles que têm a capacidade de *alertar* os usuários (em geral, executivos das alta e média gerências) sobre a ocorrência de alguma *exceção*, dando-lhes os meios de achar as suas causas. Consultas OLAP inserem-se neste contexto: são atividades interativas usuário-sistema, em que o usuário pode querer mais detalhes ("drill down") ou menos detalhes ("drill up") de um relatório que exibe tendências de um negócio. Tendências são geralmente comparações de dados em um certo nível de agregação, através do tempo. O usuário pode ainda querer combinar relatórios logicamente interligados ("drill across") **[KIMBALL 96]**.

Considere a seguinte tabela de um DW relacional *Fatos(Produto#, Loja#, Tempo#, Venda)* onde Produto#, Loja# e Tempo# são respectivamente as chaves das *tabelas de dimensão* Produto, Loja e Tempo, enquanto que Venda é um atributo *aditivo*, isto é, que contém medidas numéricas ou fatos que podem ser somados através de todas as dimensões. Suponha também que o nível de agregação da tabela *Fatos* é *dia*, ou seja, cada linha da tabela exibe o total de vendas de um produto, de uma loja, num certo dia. A tabela de dimensão Tempo contém a semântica de cada valor de Tempo# (atributos Dia-normal, Fim-de-semana, Semana, Mês, e outros). Por sua vez, as tabelas de dimensão Produto e Loja descrevem, respectivamente, as semânticas de Produto# e Loja#.

Um usuário pode iniciar uma interação com o sistema desejando analisar as vendas de alguns produtos ao longo de vários meses. Se este tipo de análise fosse bastante comum e dado o gigantismo da tabela Fatos (para 30.000 produtos, 300 lojas e 10 anos pode chegar a mais de 100 gigabytes), uma *visão materializada* da mesma poderia ser *antecipadamente* criada, *PLT(Produto#, Loja#, Tempo#, Venda-Mensal)*, através da junção das tabelas Fatos e

[1] Alguns termos relacionados a DW utilizados nesta seção podem ter seu significado encontrado no Anexo I – Glossário de Termos.

Tempo, para uma lista de meses, e ordenada por Produto, Loja e Tempo. Como PLT seria ainda muito grande, outras visões materializadas da tabela Fatos também poderiam ser criadas, *PL(Produto#, Loja#, Venda-no-Período)*, ordenada por Produto e Loja, e *P(Produto#, Venda-no-Período)*, ordenada por Produto. Note que as tabelas PLT, PL e P são geradas de consultas do tipo *group by/order by* à tabela Fatos. Com o DW assim convenientemente preparado, esta seqüência de interações, consulta a PLT, depois a P ("drill up"), em seguida a PL ("drill down"), voltando a PLT ("drill down") serão respondidas de maneira mais eficiente.

É interessante observar que as visões materializadas de um DW devem ser transparentes aos usuários. No nosso exemplo, o usuário sempre submete suas consultas à tabela Fatos, que são então *desviadas* para as visões materializadas convenientes, via um software *navegador em agregados*[2] ("aggregate navigator") proposto inicialmente por **[KIMBALL 96]**.

Infelizmente, dada a natureza interativa de uma sessão OLAP, as atividades dos usuários não são completamente previsíveis. Voltando ao nosso exemplo, o usuário poderia, no meio da sua sessão, dirigir sua atenção para lojas. Podemos facilmente perceber que as ordens das visões materializadas PLT e PL não favorecem as novas operações OLAP desejadas pelo usuário, pelo menos na mesma medida em que favorecem a obtenção de informações sobre produtos. Por outro lado, é importante levar em conta que não é conveniente materializar um grande número de visões, para evitar o que se chama de *proliferação de visões materializadas*, levando a um DW de tamanho global "infinito".

Para agilizar o esforço de programação, os "group bys" correspondendo a cada uma das combinações de L, P e T podem ser calculados via um único operador, "cube by" **[GRAY 96]**. Para a implementação do "cube by", calcular separadamente cada "group by" do mesmo pode ser muito ineficiente. A idéia então é a de reutilizar um "group by" para calcular um outro "group by". Por exemplo, pode-se aproveitar o cálculo de PL para calcular, ao mesmo tempo, P.

Mas a questão ainda não está de todo respondida: todos os "group bys" possíveis são necessários? Desejando-se evitar a proliferação de visões, é melhor armazenar PL ou LP, tendo em vista a frequência e a importância relativas das consultas ao DW? Há restrições quanto ao tempo para a criação de todas as visões?

[2] - *Agregado* é uma outra denominação para visão materializada.

1.3 Motivação e Contribuição da Dissertação

Estamos diante de um problema combinatório, cuja questão a ser idealmente respondida é: qual o *custo global ótimo* de se materializar um conjunto de visões, considerando a possibilidade de se criar várias visões simultaneamente, e sem perder de vista os custos das consultas que vão fazer uso das visões? As vantagens de se diminuir os custos das consultas são óbvias. Reduzir os custos de criação das visões também é importante: embora este processo seja sempre realizado em modo "batch", ele pode consumir várias horas, e durante todo esse tempo o DW deve ficar indisponível aos usuários – reduzir de algumas horas sua duração pode ser uma prioridade, principalmente se o DW atende a uma região geográfica muito grande, com diversos fusos horários.

O objetivo desta dissertação é apresentar um algoritmo que seja uma solução para o problema combinatório enunciado. Como mostraremos, algumas vezes uma solução com custo mínimo não será possível, pois pode haver conflitos irreconciliáveis entre os dois objetivos, criação eficiente de visões *versus* consultas eficientes ao DW. Neste caso, o algoritmo procura gerar uma solução que seja *a mais aproximada possível* do custo global mínimo para a criação das visões. Outros algoritmos existentes na literatura são menos abrangentes, por não levarem em conta o uso que se faz das visões materializadas.

1.4 Estrutura e Apresentação dos Capítulos da Dissertação

Esta dissertação é constituída de seis capítulos, incluindo esta Introdução, e três apêndices. Descrevemo-los sucintamente, a seguir.

O Capítulo 2 é um breve introdução a DW, em que definimos a terminologia básica do assunto, e tecemos considerações sobre esquemas de dados para DWs e sobre visões materializadas em DWs, temas que nos interessam mais de perto. Um glossário de termos (Apêndice I) completa a introdução a DW.

No capítulo 3, mostramos algumas limitações da linguagem padrão de acesso a bancos de dados, SQL, em relação a aplicações OLAP, e que extensões se fazem necessárias para que as aplicações OLAP possam ser realizadas eficientemente. Definimos formalmente o operador relacional "cube by", que é a generalização multidimensional do operador "group by" utilizado na construção de agregados ou visões materializadas, para terminar com a discussão de alguns trabalhos encontrados na literatura para a computação de um "cube by", com suas limitações.

O Capítulo 4 é o coração da Dissertação, com a apresentação do nosso algoritmo *Quasi*-Ótimo para a materialização e a utilização eficientes de visões em DWs, permitindo materializar simultaneamente diversas visões, sem perder de vista os custos de processamento das consultas ao DW. O algoritmo ou otimiza o custo total de computação de um "cube by", ou então garante que o custo total para a sua computação é o mais próximo possível do custo ótimo.

No capítulo 5, detalhamos diversos aspectos da implementação do algoritmo *Quasi*-Ótimo.

Finalizamos a Dissertação com o Capítulo 6, com as conclusões e algumas sugestões de trabalhos futuros.

A Dissertação termina com mais dois apêndices: Apêndice II, com o pseudo-código do algoritmo *Quasi*-Ótimo, e o Apêndice III, com instruções para o uso do algoritmo.

Capítulo 2

Uma Breve Introdução a Data Warehouses

"Data Warehouse" é uma das áreas de sistema de informação de grande interesse atualmente. Neste capítulo, definimos a terminologia básica da área e mostramos que um "data warehouse" é parte de uma estrutura maior que inclui bancos de dados operacionais, sistemas de arquivos, software de extração e integração de dados, e ferramentas de apoio à decisão.

2.1 "Data Warehouse" e "Data Warehousing"

Em uma típica organização, os dados operacionais estão dispersos através de uma variedade de SGBDs e sistemas de arquivos, em formato às vezes largamente diferentes, e em uma variedade de plataformas de hardware. Uma padronização do software e do hardware não poderia ser cogitada, pois implicaria em um enorme desperdício de tempo e dinheiro. Por outro lado, os executivos no nível de gerência precisam utilizar, de maneira simples e rápida, dados integrados e derivados daqueles disseminados através da organização, que possam auxiliá-los nas atividades de tomada de decisão.

A necessidade da integração de múltiplos, distribuídos e heterogêneos bancos de dados e outras fontes de informação tem levado a um enorme esforço de pesquisa nos últimos anos. As abordagens iniciais para o problema de integração de dados baseavam-se em dois passos:

1. Aceitar uma consulta, determinar o conjunto apropriado de fontes para responder à consulta, e gerar as sub-consultas apropriadas ou comandos para cada fonte de informação.

2. Obter os resultados das fontes, traduzir, filtrar e integrar os dados, e retornar o resultado final para o usuário.

Esta primeira abordagem de integração é chamada de *Mediador* [WIEDERHOLD 1992], para referir-se aos módulos de software que decompõem consultas e combinam seus resultados. Apesar dessa abordagem garantir informação em tempo-real, o processamento de consultas é inexoravelmente ineficiente e demorado, se as fontes de informação são muitas e dispersas.

A alternativa ao enfoque Mediador que se revelou muito mais viável é a *abordagem antecipada* para a integração de dados. Ela funciona assim:

1. A informação das fontes de informação de interesse é extraída, traduzida, filtrada e integrada antecipadamente, e armazenada em um repositório centralizado. Esse repositório é chamado de "Data Warehouse" (DW).

2. Quando uma consulta é submetida, esta é avaliada diretamente no DW, sem ter acesso, portanto, às fontes.

Segundo [INMON 1997], *um DW é uma coleção de dados não-voláteis, invariantes em termos temporais, integrados e orientados a um assunto, utilizados no apoio a decisões gerenciais.*

O conjunto das etapas de extração, tradução, filtragem e integração forma o "back-end" de um DW. As ferramentas OLAP que têm acesso ao DW são o seu "front-end". O conjunto "back-end", DW e "front-end" compõe o ambiente de "Data Warehousing" (DWing) [INMON 1997; COREY 1998].

2.2 Processamento Analítico *versus* ProcessamentoTransacional

Um DW é voltado para *processamento analítico on-line* (OLAP – "On-Line Analytical Processing") [COOD 1995], cujos requisitos funcionais e de desempenho são diferentes dos das aplicações tradicionais de processamento transacional on-line (OLTP – "On-Line Transactional Processing").

Aplicações OLTP automatizam as operações do cotidiano do negócio de uma organização. Essas operações são estruturadas e repetitivas, consistindo de transações de curta duração, atômicas a falhas e isoladas. As transações requerem dados atualizados e detalhados. Consistência e recuperação em caso de falha do banco de dados operacional são fatores críticos. Bom desempenho é obtido pela maximização do número de transações executadas concorrentemente. Aplicações OLTP são suportadas por bancos de dados que chamaremos de operacionais

Aplicações OLAP são desenvolvidas para apoiar os tomadores de decisão da organização. Para essas aplicações, dados históricos, sumariados e consolidados são mais importantes que dados atômicos e detalhados. Aplicações OLAP são suportadas por DWs. Os grandes DWs para aplicações OLAP, por serem temporais, podem ter centenas de gigabytes à terabytes de tamanho. A carga maior de operações em um DW são consultas, geralmente complexas, envolvendo uma quantidade muito grande de registros do DW. Um conjunto de consultas OLAP, logicamente relacionadas, compõe uma sessão OLAP. Durante uma sessão OLAP, o DW deve ser invariante no tempo (*consistência temporal* **[Kimball 1996]**).

Terminamos esta seção com a Tabela 2.1, que contrasta as aplicações OLAP com as aplicações OLTP.

Características	OLAP	OLTP
Objetivo	Tomada de Decisão	Controle Operacional
Operação Típica	Análise	Atualização de Dados
Complexidade das Operações	Grande	Pequena
Nível de Agregação dos Dados	Alto	O mais baixo
Dados Históricos	Sim	Não
Frequência das Operações	Moderada	Alta
Usuário	Gestores do Negócio	Pessoal Operacional

Tabela 2.1: Características OLAP x OLTP

2.3 Análise Multidimensional

A atividade de tomada de decisão por um gerente de uma organização é muitas vezes baseada na *análise multidimensional*. Uma *dimensão* é um critério de agregação de dados numéricos

aditivos. A essência da análise multidimensional consiste em fazer-se comparações, ao longo do tempo, de dados sumariados ou agregados por diferentes dimensões, a fim de se descobrir tendências do negócio da organização.

A capacidade de "navegar" dentro das dimensões de um negócio é um aspecto fundamental da análise multidimensional. Um outro aspecto importante da análise multidimensional é a flexibilidade para que se responda a questões em *quatro espaços de apoio à decisão* [PARSAYE 1996]:

- *Espaço de Dados* (Qual é o preço de um determinado produto?)
- *Espaço de Agregação* (Quais são as vendas mensais de produtos para um loja?)
- *Espaço de Influência* (Que fatores influenciaram as vendas em uma determinada loja?)
- *Espaço de Variação* (Como têm se comportado as vendas nos três últimos meses?)

Nesta Dissertação, tratamos somente de espaço de agregação e de espaço de variação (a conjunção dos dois sendo chamada de *Espaço OLAP*). Espaço de dados é típico dos sistemas operacionais tradicionais. Espaço de influência é uma área de pesquisa ainda bastante embrionária, requerendo uma forte integração com algumas sub-áreas da área de inteligência artificial *(data mining)*.

Dentre as operações básicas de suporte à análise multidimensional destacamos "drill down", "drill up" e "drill across".

"Drill down" é a seleção iterativa de dados sumariados, onde cada iteração apresenta dados com um nível maior de detalhes. Considere um relatório de vendas anuais, inicialmente. Como resultado de operações "drill down", o relatório passa a ser de vendas mensais, em seguida de vendas semanais, e assim por diante. Neste exemplo, as operações "drill down" se realizam ao longo da dimensão tempo, da maior granularidade de tempo para a menor.

A operação *"drill up"* 'navega' no sentido oposto ao do da operação "drill down". Seja um relatório de vendas diárias, inicialmente. Como resultado de operações "drill up", o relatório passa a ser de vendas semanais, em seguida de vendas mensais, e assim por diante. Neste exemplo, as operações "drill up" se realizam ao longo da dimensão tempo, da menor granularidade de tempo para a maior.

A operação que combina relatórios diferentes contendo dimensões comuns em um simples relatório é feita através da operação *"drill across"*. A título de ilustração, pode-se desejar combinar ("to drill across") dois relatórios logicamente relacionados, relatório de vendas de produtos e relatório de distribuição dos mesmos produtos, para decidir-se por aumentar ou diminuir o ritmo de fabricação dos produtos.

Bancos de dados de suporte à análise multidimensional são chamados de *bancos de dados multidimensionais*. Os DWs devem, obviamente, ser bancos de dados multidimensionais. Como modelar bancos de dados multidimensionais é o assunto da próxima seção.

2.4 Modelagem de Bancos de Dados Multidimensionais

O projeto de bancos de dados multidimensionais (DWs, no nosso contexto) difere grandemente do projeto de bancos de dados para aplicações OLTP (BDs OLTP ou operacionais).

Os BDs OLTP são projetados para facilitar a atualização de dados, sua modelagem primando por evitar o mais possível dados redundantes, para que as transações possam modificar dados 'localmente'. Do fato de que os BDs OLTP são quase sempre relacionais, seus esquemas de dados consistem de diversas tabelas logicamente relacionadas, como é ilustrado na Figura 2.1.

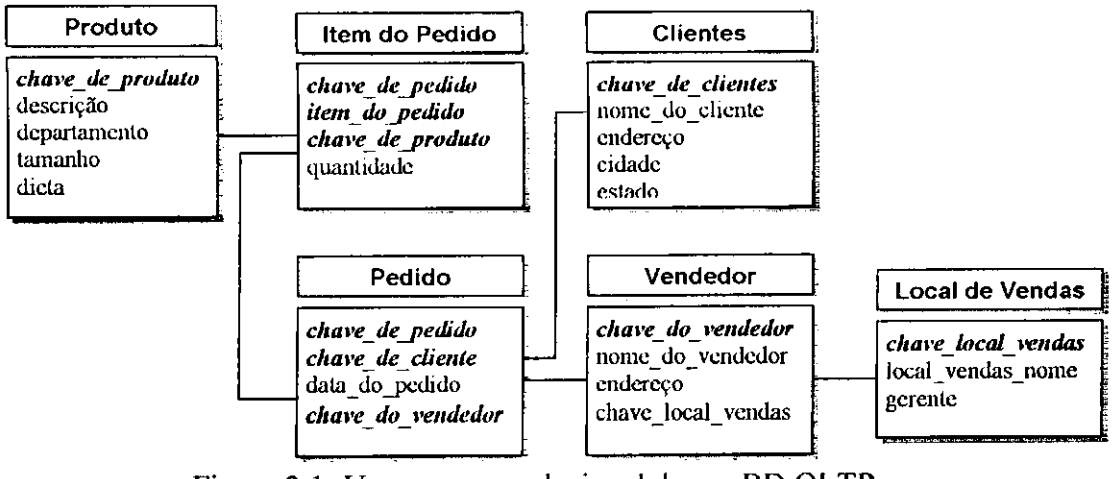


Figura 2.1: Um esquema relacional de um BD OLTP

As tabelas da Figura 2.1 encontram-se *normalizadas*, ou seja, contêm somente redundâncias absolutamente necessárias à consistência inter-tabelas. Por exemplo, na tabela Item_do_Pedido, o item de dados Chave_de_Pedido (chave_externa) é repetido do item de

dados homônimo da tabela Pedido (chave_primária[1]). Desta forma, dado um pedido da tabela Pedido, pode-se achar todos os ítens do pedido, na tabela Item_do_Pedido, através da ligação lógica chave_primária-chave_externa. Entretanto, ítens de pedidos encontram-se exclusivamente na tabela Item_do_Pedido, suas atualizações sendo, portanto, 'locais' à tabela Item_do_Pedido. Na figura, as linhas entre tabelas exprimem ligações lógicas.

Ainda em relação aos esquemas de dados para BDs OLTP, pode-se observar que eles são um tanto obscuros, no sentido de que geralmente não são compreensíveis de um simples exame.

Os requisitos de esquemas de dados para DWs são diferentes dos de esquemas de dados para BDs OLTP. Nos esquemas para DWs, as dimensões devem ficar bem destacadas, e devem ser facilmente compreendidos de um simples exame.

As matrizes bidimensionais (planilhas) são um bom "framework" para a modelagem de bancos de dados multidimensionais. Compare a expressividade da planilha da Tabela 2.3 com a da Tabela 2.2 (tabela relacional). Em ambas, têm-se duas dimensões, Produto e Região, e medidas aditivas associadas às dimensões (Vendas).

Produto	Região	Vendas
Produto1	Leste	50
Produto1	Oeste	60
Produto1	Sul	100
Produto2	Leste	40
Produto2	Oeste	70
Produto2	Sul	80
Produto3	Leste	90
Produto3	Oeste	120
Produto3	Sul	140
Produto4	Leste	20
Produto4	Oeste	10
Produto4	Sul	30

Tabela 2.2: Representação Relacional

	Leste	Oeste	Sul
Produto1	50	80	100
Produto2	40	70	80
Produto3	90	120	140
Produto4	20	10	30

Tabela 2.3: Representação Bidimensional

É evidente que a planilha é muito mais compreensivel, com os valores das dimensões bem destacados e bem 'arrumados'.

[1] - Dada uma tabela, um valor da sua chave primária identifica inequivocamente uma linha da tabela.

Infelizmente, na prática necessita-se de 3, 5, 10 ou até mais dimensões. Visualizar uma matriz com 3 dimensões (cubo) já é muito difícil. Acima de 3 (array multidimensional) é impossível, diretamente. Por outro lado, o modelo relacional, com tecnologia bastante amadurecida em termos de desempenho e confiabilidade, e com base instalada de praticamente 100% do mercado mundial, não podia ser simplesmente ignorada. A solução encontrada foi *simular* um array multidimensional por meio de tabelas relacionais. O tal esquema é chamado de esquema relacional com estrutura em estrela.

2.4.1 Esquema Relacional com Estrutura em Estrela

Um esquema relacional com estrutura em estrela ("star schema" **[Kimball 1996]**) consiste, como todo esquema relacional, de um conjunto de tabelas. Entretanto, vislumbra-se nele claramente dois tipos de tabela: uma tabela *central* de medidas numéricas ou *fatos* (tabela de fatos), geralmente com alta cardinalidade (a tabela-estrela), em torno da qual 'orbitam' diversas *tabelas de dimensão*, cada uma representando uma das dimensões de um negocio (as tabelas-planetas). As tabelas de dimensão têm geralmente cardinalidade pequena, quando comparadas com a da tabela de fatos. Quando se diz que uma tabela de dimensão 'orbita' em torno de sua tabela de fatos, entenda-se a existência de uma ligação lógica entre as duas tabelas (a tabela de fatos recebe a chave externa da tabela de dimensão). A Figura 2.2, extraída de **[KIMBALL 1996]**, ilustra um esquema em estrela para um DW de vendas.

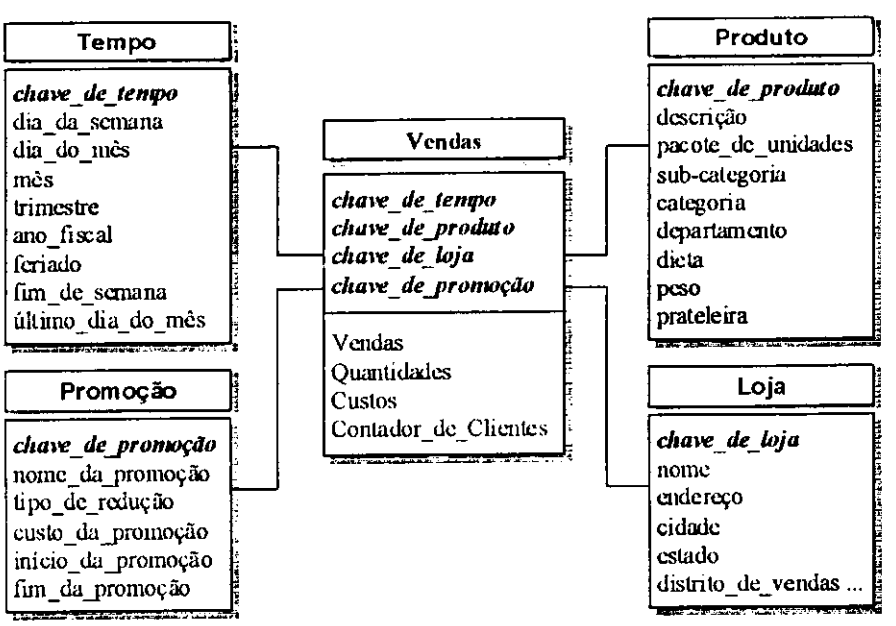


Figura 2.2: Esquema Relacional com Estrutura em Estrela para um DW de vendas

Nela, podemos ver a tabela-estrela Vendas, e as tabelas-planetas Produto, Promoção, Loja e Tempo.

A seguir, damos mais detalhes de tabelas de fatos e de dimensão, ilustradas com o exemplo da Figura 2.2.

Tabela de Fatos

Cada linha de uma tabela de fatos representa as medidas numéricas associadas ao negócio da empresa, ou fatos, para uma combinação dos valores das chaves primárias das tabelas de dimensão. O conjunto das chaves primárias das tabelas de dimensão (chaves externas da tabela de fatos) compõem a chave primária da tabela de fatos. No exemplo da Figura 2.2, a chave primária da tabelade fatos Vendas é o conjunto Chave_de_Tempo, Chave_de_Produto, Chave_de_Loja e Chave_de_Promoção.

Uma importante consideração sobre tabelas de fatos é que, em geral, elas são *temporais*, guardando históricos de fatos de 5, 10 ou até mais anos, muito frequentemente *dia-a-dia*. Desta forma, elas são absolutamente essenciais à geração, de maneira simples, de relatórios que exibem tendências de um negócio. Dado o seu aspecto temporal, a cardinalidade das tabelas de fatos geralmente é enorme (da ordem de gigabytes ou terabytes), e ela é determinante do tamanho de um DW. Impõe-se então que as tabelas de fatos sejam *normalizadas* (como as tabelas dos BDs OLTPs), para evitar toda redundância desnecessária.

Além da chave primária, os demais atributos de uma tabela de fatos contêm, a rigor, valores aditivos, semi-aditivos ou não-aditivos:

- *atributos aditivos*: seus valores podem ser somados ao longo de todas as dimensões. Exemplos: total de um produto por região, total do produto por loja, total do produto por um período de tempo, etc. Na Figura 2.2, Vendas, Quantidades e Custos são atributos aditivos.

- *atributos semi-aditivos*: a soma de seus valores ao longo de uma dimensão não têm significado em si, mas pode ser útil a outras operações aditivas. Exemplo: saldos de uma conta não devem ser somados ao longo do tempo; entretanto, a média dos saldos num certo período é uma informação muito útil. Na Figura 2.2, Contador_de_Clientes é um atributo

semi-aditivo (a soma de seus valores não tem significado em si, pois um cliente pode aparecer em mais de um contador).

- *atributos não-aditivos*: não podem (ou não faz sentido) ser somados. Em geral, valores percentuais são não-aditivos.

Outro importante aspecto do projeto de um esquema em estrela é a questão da *granularidade* da tabela de fatos. A granularidade diz respeito ao nível de detalhe dos fatos nela contidos. Quanto mais detalhe, mais baixo o nível de granularidade, e mais volumosa é a tabela de fatos. Quanto menos detalhe, mais alto o nível de granularidade, e menos volumosa, comparativamente, é a tabela de fatos. Assim, por exemplo, é preciso decidir se os fatos serão a nível de produto ou de categoria de produtos, a nível de loja ou de distrito de vendas, se são fatos diários, ou semanais, ou mensais, etc. Como veremos, na prática, poderemos ter *constelações* de estrelas, cada estrela (tabela de fatos) com uma granularidade diferente.

Finalmente, enquanto durar uma sessão usuário-sistema, as tabelas de fatos não devem mudar, ou devem ser estáticas. Na prática, as tabelas de fatos são atualizadas em modo "batch", em horários de pouca ou nenhuma atividade do sistema (quase sempre em intervalos de 24 horas, à noite).

Tabelas de Dimensão

As tabelas de dimensão devem conter atributos que sejam significativos para os cabeçalhos dos relatórios de análise de um negócio, que permitam restringir o acesso à tabela de fatos, e ainda, que sirvam de critério de agregação dos registros da tabela de fatos. Não sendo temporais, as tabelas de dimensão influem minimamente no volume de um DW, e nunca se deve comprometer o projeto das tabelas de dimensão com o fim de gerar economia de espaço. Em consequência, as tabelas de dimensão não devem ser necessariamente normalizadas.

Há uma outra razão para que as tabelas de dimensão não devam ser completamente normalizadas. Para restringir o aceso à tabela de fatos, os usuários fazem primeiro "browsing" nas tabelas de dimensão. Segundo **[Kimball 1996]**, em um dia de interação usuário-sistema, 80% das operações são "browsing" em tabelas de dimensão. Se as tabelas de dimensão fossem normalizadas, muito fragmentadas portanto, isto dificultaria em muito as operações "browsing".

Tabelas de dimensão não normalizadas geralmente apresentam relacionamentos hierárquicos (relacionamentos 1:N) entre seus atributos. Observe a tabela de dimensão Loja, do esquema em estrela da Figura 2.2. Temos a hierarquia de atributos *loja – cidade – estado* (uma cidade pode ter muitas lojas e uma loja existe em uma única cidade; um estado tem várias cidades e uma cidade existe em um único estado). Uma outra hierarquia é *loja – distrito_de_vendas – região_de_vendas*.

A existência de hierarquias permite definir "drilling up" e "drilling down" naturais ou implícitos. Uma operação "drilling" natural 'navega' em uma hierarquia de atributos de uma dimensão, de 'cima' para 'baixo' ("drill down"), ou de 'baixo' para 'cima' ("drill up"). Por exemplo, de vendas por loja para vendas por cidade é uma operação "drill up", enquanto que, de vendas por cidade para vendas por loja é uma operação "drill down". Deve ser deixado claro, entretanto, que as operações "drill up" e "drill down" não se restringem a hierarquias de atributos, podendo ser generalizadas **[Kimball 1996]**.

Com relação a valores de chaves de dimensão, recomenda-se que eles não tenham nenhum significado em si (valores internos ou "surrogates"), sua semântica sendo definida pelos valores dos demais atributos **[Kimball 1996]**. A explicação é que valores de chave devem servir unicamente para as ligações (lógicas) entre registros das tabelas de dimensão e registros da tabela de fatos. Mais geralmente, valores de chave devem servir unicamente para as ligações lógicas entre tabelas (os projetistas de BDs tradicionais não observam isto!).

Terminamos esta seção com uma pequena digressão sobre a dimensão Tempo, onipresente em todos os DWs.

A Dimensão Tempo

Uma tabela representando a dimensão Tempo pode, à primeira vista, parecer desnecessária em um DW, já que é possível definir um atributo do tipo Data (DD/MM/AAAA) para a tabela de fatos. Entretanto, uma data esconde muito de sua semântica. Por exemplo, 21/11/1998 representa explicitamente apenas três 'pontos' de tempo, o dia 21, o mês de novembro e o ano de 1998. Outros 'pontos' do tempo relacionados a esta data, que poderiam ser de muito interesse dos usuários, como semana, trimestre, feriado, etc., ficam completamente obscurecidos.

Como já dissemos, um dos requisitos fundamentais de um projeto de DW é a sua clareza. Particularmente, a informação histórica (séries temporais) deve estar bem explícita. Para levar isso em conta, todos os 'pontos' relevantes do tempo, como semana, trimestre, feriado, etc., devem ser explícitos através dos atributos da dimensão Tempo.

Em relação ao esquema da Figura 2.2, outros 'pontos' relevantes do tempo são ano fiscal, fim de semana e último dia do mês.

2.5 Visões Materializadas em DWs

O tamanho descomunal das tabelas de fatos em DWs é uma enorme barreira ao grande objetivo de um projeto de DW, que é o de permitir que as consultas OLAP aos DWs, geralmente muito complexas, possam ser processadas de maneira eficiente.

Da análise cuidadosa das consultas OLAP, pode-se perceber que, em sua imensa maioria, elas impõem a agregação de centenas ou milhares de linhas da tabela de fatos (por exemplo, a granularidade da tabela de fatos é *dia*, mas quer-se as vendas por ano fiscal), a um custo muito alto se essas agregações tiverem que ser feitas em tempo-real. Para contornar o problema, surgiu a idéia de se pré-computar agregados, pelo menos aqueles mais frequentemente usados pelas consultas OLAP.

Um agregado pré-computado é uma nova tabela de fatos, derivada da tabela de fatos *básica*, e chamada também de *visão materializada*. Uma visão materializada pode ter suas próprias tabelas de dimensão, chamadas de *dimensões derivadas* das dimensões *básicas*.

Retomemos o esquema da Figura 2.2. Suponhamos que a granularidade do tempo é *dia*, mas que é muito comum a emissão de relatórios de vendas mensais e trimestrais. Para facilitar a exposição, vamos considerar que a tabela básica Tempo é, na verdade, a tabela Dia. Poderiam então ser materializadas as visões Vendas-Mês e Vendas-Trimestre, derivadas *logicamente* da tabela de fatos básica, Vendas. Para Vendas-Mês, as tabelas de dimensão seriam Mês (tabela de dimensão derivada da tabela Dia), Loja e Produto (a tabela Promoção sendo suprimida). Para Vendas-Trimestre, as tabelas de dimensão seriam Trimenstre (derivada da tabela Dia), Loja, Produto e Promoção.

Em relação a outras tabelas de dimensão derivadas, poderíamos ter Categoria, derivada de Produto, e/ou Distrito_de_Venda, derivada de Loja, etc. Consequentemente, teríamos as tabelas de fatos derivadas Vendas-Categoria, Vendas-Distrito, Vendas-Distrito-Mês, Vendas-Distrito-Trimestre, podendo ser todas, além de outras tabelas de fatos derivadas representando outras combinações de dimensões derivadas.

Qual é o limite para a criação de visões materializadas? O que se pode dizer é que a explosão de visões materializadas pode levar um DW a um tamanho 'infinito'.

Deve ser notado que as tabelas Trimestre e Mês, por exemplo, poderiam ser derivadas cada uma da tabela Dia, em separado, paralelamente ou não, ou que Trimestre poderia ser derivada de Mês, além de várias outras possibilidades. Tudo vai depender da *qualidade* do algoritmo de criação de visões materializadas. Mais precisamente, que *plano* o algoritmo segue para criar as visões materializadas? O plano é ótimo, ou então quão próximo ou distante do plano ótimo ele é?

Outro desafio à construção de um bom algoritmo de criação de visões materializadas reside no fato de que muito provavelmente as consultas OLAP são *sensíveis* às *ordens* com que as visões materializadas são armazenadas no DW. Assim , não basta criar eficientemente as visões materializadas. É preciso também que a sua utilização, pelas consultas OLAP, seja eficiente!

É oportuno afirmar que as visões materializadas de um DW são completamente *transparentes* aos usuários. Isto quer dizer que os usuários só conhecem o DW básico (tabela de fatos básica e tabelas de dimensão básicas). É papel de um software, ao qual dá-se o nome de *'navegador' em agregados* ("aggregate navigator" **[Kimball 1996]**), *desviar* uma consulta para a(s) visão(ões) materializada(s) mais apropriada(s) a ela[2].

Na seção seguinte, discutimos um pouco mais o processo de criação de visões materializadas.

[2] - No sentido de tornar seu processamento eficiente.

Criação e Armazenamento de Visões Materializadas

Como vimos, uma visão materializada se confunde com uma tabela de fatos agregados de uma tabela de fatos básica. Um registro de uma tabela que é uma visão materializada representa, portanto, um agregado, ou sumário, de registros da tabela de fatos básica. É óbvio que, desta forma, a granularidade de uma visão materializada é sempre maior que a granularidade da tabela de fatos básica correspondente.

A pergunta que devemos responder agora é: a linguagem SQL, padrão de acesso a bancos de dados relacionais, estaria equipada dos operadores necessários à criação de visões materializadas? A resposta é: sim e não. Sim, porque agregados podem ser facilmente construídos por meio do operadores *"group by"* e *"sum"*. A Figura 2.3, extraída de **[GRAY 1996]**, é uma ilustração de como funcionam esses operadores.

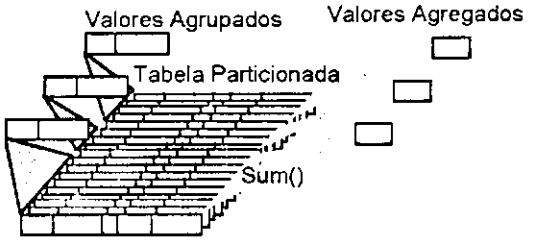


Figura 2.3: Lógica de Funcionamento dos Operadores *"group-by"* e *"sum"*.

Para exemplificar, considere novamente o esquema em estrela da Figura 2.2. Cada registro da tabela Vendas contém valores de vendas por dia, por produto, por loja e por promoção (suponha que a inexistência de uma promoção para um produto é uma 'promoção' fictícia). Com "group by" e "sum" podemos construir vários tipos de agregados, como:

- Vendas diárias por categoria de produtos e por loja
- Vendas mensais por produto e por loja
- Vendas diárias de produtos por categoria e por região de vendas
- Vendas mensais de produtos por categoria e por região de vendas

Capítulo 3

A Linguagem SQL e Operações OLAP

Neste capítulo, começamos com as limitações da linguagem SQL — padrão de acesso a bancos de dados relacionais — para fazer face aos requisitos de operações OLAP, em seguida, apresentamos um novo operador, "cube by", que é uma generalização do "group by" clássico, e encerramos o capítulo com uma discussão sobre as diferentes implementações do "cube by".

3.1 Limitações da Linguagem SQL

Diversas operações OLAP são difíceis, quando não impossíveis, de serem implementadas com a linguagem SQL-92, padrão de acesso a bancos de dados relacionais. As principais limitações da linguagem são:

- Impossibilidade de realizar operações "drill up" e "drill down" com totais e sub-totais;
- Inadequação para a apresentação multidimensional de dados.

Com relação à primeira limitação, considere o relatório de vendas da Figura 3.1.Nele, os dados de vendas de automóveis da marca 'Fiat Uno' estão sumariados por três critérios de agregação: vendas por modelo, vendas anuais por modelo, e vendas anuais de modelos, por cor.

Modelo	*Ano*	*Cor*	*Vendas Anuais por Modelo por Cor*	*Vendas Anuais por Modelo*	*Vendas por Modelo*
Fiat Uno	1997	preto	*50*		
		branco	*40*		
				90	
	1998	preto	*85*		
		branco	*115*		
				200	
					290

Figura 3.1: Relatório de vendas de automóveis.

O problema com este relatório é que sua representação não é relacional, ou tabular. Portanto, ele é 'estranho' à linguagem SQL. A representação relacional do mesmo relatório, com o código SQL que a gerou, é mostrada na Figura 3.2.

Modelo	*Ano*	*Cor*	*Vendas*
Fiat Uno	1997	preto	50
Fiat Uno	1997	branco	40
Fiat Uno	1998	preto	85
Fiat Uno	1998	branco	115

```
SELECT Modelo, Ano, Cor, SUM(Vendas)
FROM Vendas
WHERE Modelo = 'Fiat Uno'
GROUP BY Modelo, Ano, Cor
```

Figura 3.2: Representação relacional e código SQL do relatório da Figura 3.1.

Comparando os relatórios das figuras 3.1 e 3.2, o segundo perde as vendas anuais por modelo, e as vendas por modelo, devendo, portanto, ser rejeitado por insuficiência. A explicação para essa perda de totais reside na limitação do operador "group by", com sua incapacidade de realizar múltiplos níveis de agregação, dentro da mesma operação.

Pode ser argumentado que, com múltiplos "group bys", chega-se também ao relatório da Figura 3.1. Entretanto, pode-se também argumentar que (1) tratar os "group bys" independentemente um do outro é ineficiente, e (2) múltiplos "group bys" levam a códigos SQL muito detalhados e, portanto, complexos.

O relatório da Figura 3.1, embora dentro das especificações exigidas para o acompanhamento das vendas do automóvel da marca 'Fiat Uno', carece, no entanto, de um formato adequado à apresentação multidimensional dos dados de vendas. Muito melhor é a representação em forma de planilha da Figura 3.3, em que as dimensões Ano e Cor ficam bem evidenciadas.

Fiat Uno	1997	1998	Total
preto	50	85	135
branco	40	115	155
Total	90	200	290

Figura 3.3: Relatório da Figura 3.1, em forma de planilha.

Esta forma de mostrar os dados tem, porém, suas limitações. Se mais um modelo de carro (p.e., 'Fiat Pálio') tiver que ser analisado, teremos então uma terceira dimensão — Modelo — para a 'planilha', cuja representação 'plana' já não é tão simples. As dificuldades de apresentação aumentam à medida em que a 'planilha' tem quatro, cinco, ... dimensões. Infelizmente na prática a apresentação de dados acima ainda não é fácil de ser obtida com SQL padrão. A apresentação de uma matriz com 6 dimensões vai requerer a união de 64 (2^6) *group-bys* diferentes, que representam 64 leituras da tabela base, 64 ordenações e uma longa espera pelo resultado final [GRAY 1996].

Na sequência, trataremos somente das soluções para a primeira limitação da linguagem SQL, qual seja, a impossibilidade de realizar operações OLAP com totais e sub-totais. As soluções para o problema da visualização de dados multidimensionais fogem do escopo desta Dissertação.

3.2 Operador *Cube by*

Como foi explicado na seção anterior, o operador "group by" é muito limitado para suportar operações OLAP que requerem agregações em diferentes níveis de detalhe. Para remediar esta limitação, um novo operador, *"cube by"*, foi formalizado [GRAY 1996].

O operador *"cube by"* é uma generalização multidimensional do operador "group by", equivalendo a uma coleção de "group bys", sendo um para cada combinação das dimensões da lista de parâmetros do operador. Mais geralmente, um "cube by" associado a N dimensões resultará em 2^N "group by".

Considere a consulta:

```
SELECT SUM(vendas)
FROM vendas
CUBE BY modelo, ano, cor
```

Nesta consulta, as dimensões do "cube by" são *modelo, ano, cor*. Resultará em 2^3=8 "group bys", correspondendo às combinações um a um, dois a dois e três a três das dimensões, além da combinação 'vazia' ("null"). Cada combinação de dimensões é chamada

de 'cuboide' ("cuboid"). Podemos então nos referir a cuboides como modelo-ano-cor, modelo-cor, cor, etc. A cada cuboide, está associada uma função de agregação.

[DESHPANDE 1996] formalizou um cuboide da seguinte forma. Considere o conjunto de atributos $X = \{A_1, A_2, ..., A_k\}$ da relação $R = [X, Y]$ e a função de agregação F(). Um cuboid C(X) com *j* atributos $\{A_{i1}, A_{i2}, ..., A_{ij}\} \subseteq X$ é assim definido:

$$CUBOID(C(X)) = \{(a_{i1}, a_{i2}, ..., a_{ij}, F()) \mid \Pi_{Ai1,Ai2,...,Aij}(R) = [a_{i1}, a_{i2}, ..., a_{ij}]\}$$

Formalizaremos o operador "cube by" da seguinte forma. Dada a definição de cuboides, sejam X = {X1, X2, ..., Xn} e a relação R = [X1, X2, ..., Xn, Y]. Seja também $C_{m,n}(X)$ o conjunto de cuboides de *m* elementos dos *n* elementos de X. Temos então:

$$CUBE\text{-}BY\ (X1, X2, ..., Xn) = C_{1,n}(X) + C_{2,n}(X) + ... + C_{m,n}(X)$$

3.2.1 Alternativas de Armazenamento dos Cuboides de um *Cube By*

As seguintes alternativas de implementação são possíveis para o "cube by" [HARINARAYAN 1996]:

- Materialização de todos os cuboides de um "cube by" — a vantagem dessa estratégia é que consultas aos cuboides do cubo[1] materializado podem ser respondidas rapidamente. As principais desvantagens são, potencialmente, espaço proibitivo para o armazenamento do cubo, e tempo excessivo para a criação de todos os cuboides.. Cada cuboide materializado é chamado de *visão materializada*, ou simplesmente, visão.

- Materialização de nada — os cuboides são computados no momento da avaliação de uma consulta, o que reduz o espaço de armazenamento, mas pode gerar um custo excessivo de processamento da consulta.

- Materialização parcial do cubo — somente alguns cuboides, por algum critério, são materializados.

Encontram-se sistemas comerciais adotando as diversas estratégias acima. *Essbase* materializa o cubo inteiro, enquanto *BussinessObject* não materializa nada. *Metacube* materializa apenas parte do cubo. Claramente, cada estratégia tem seus benefícios e desvantagens.

[1] A partir deste ponto, confundiremos a expressão inglesa "cube by" com cubo

(modelo)

Modelo	Vendas
Fiat Uno	508
Fiat Pálio	433

(cor)

Cor	Vendas
vermelho	233
branco	369
azul	339

(ano)

Ano	Vendas
1997	343
1998	314
1999	284

(NULL)

Vendas
941

Figura 3.5: Os cuboides da consulta da Figura 3.4.

O conjunto de cuboides da Figura 3.5 é a representação relacional do ´cubo de dados´ da Figura 3.6.

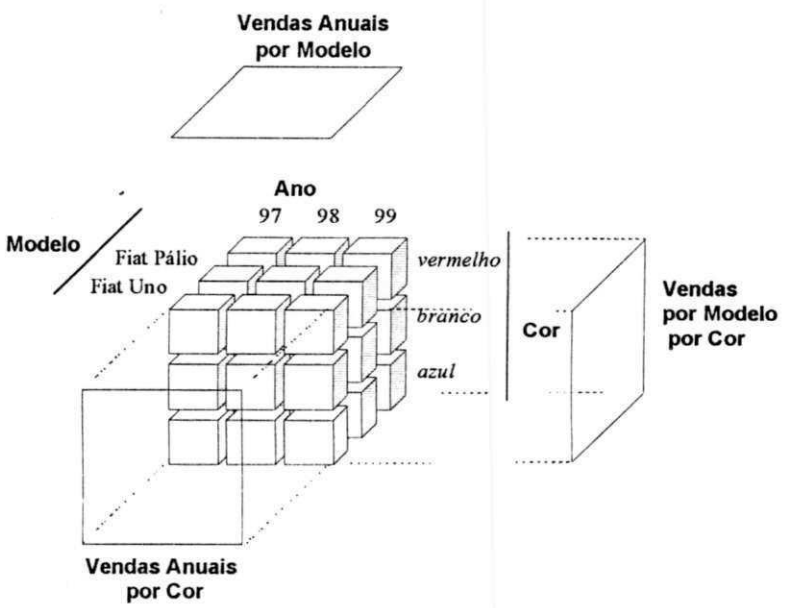


Figura 3.6: O ´cubo de dados´ correspondente aos cuboides da Figura 3.5.

A conclusão é que o "cube by" é um operador muito mais poderoso que o "group by", sendo extremamente adequado a consultas OLAP. Seu emprego imediato é em consultas do tipo "drill up" e "drill down", já que ele permite calcular agregações em diferentes níveis.

No restante deste capítulo, discutimos os diferentes métodos existentes na literatura para a computação do "cube by".

3.3 Métodos para a Computação do Cubo

O operador "cube by" generaliza o operador "group by", logo todos os métodos usados para computar "group bys" baseados em *sorts* ou *hash* [GRAEFE 1996] aplicam-se à computação do cubo.

As técnicas para a computação de um "cube by" estão relacionadas com as funções de agregação consideradas. No nosso trabalho assumiremos que as funções são *distributivas*, pois esta classe de funções engloba um maior número de funções de agregação definidas em SQL padrão. Uma função de agregação *F()* é distributiva se existir uma função *G()* tal que:

$$F(\{X_{i,j}\}) = G(\{F(\{X_{i,j} \mid i = 1,...,I\}) \mid j = 1,...,I\})$$

Funções *Count( ), Min( ), Max( ), Sum( )* são todas distributivas. Para estas funções, super-agregados podem ser computados de agregados. Por exemplo, se o conjunto *X* é a união dos conjuntos *Y* e *Z*, logo *sum(X) = sum(Y) + sum(Z)*. Essa relação não é válida para uma função não-distributiva tipo *Avg( )*

Computar o "cube by" requer a computação de todos os cuboides que formam o cubo. O cuboide básico, composto por todos atributos do "cube by", tem que ser computado a partir da relação original usando técnicas padrões para computação de group bys como *sorts* ou *hash*,

Pode-se estimar de modo analítico que a cardinalidade de um "cube by" com N atributos, onde cada atributo tem cardinalidade C_i será $\prod(C_i + 1)$. Por exemplo, para um "cube by" com 4 atributos onde cada atributo tem 4 valores distintos teremos um tamanho de (5 x 5 x 5 x 5) = 625 que é 2,4 vezes maior que a cardinalidade de um *group by* com os mesmos 04 atributos (4 x 4 x 4 x 4) = 256. Logo algoritmos eficientes para computar esse grande número de tuplas se faz necessário, uma vez que em DW reais a cardinalidade C_i dos atributos é da ordem de dezenas ou centenas.

Existe uma variedade de métodos baseados em *sorts* e *hash* para a computação do "cube by". A seguir iremos fazer uma análise dos principais métodos baseados em *sorts* encontrados na literatura apresentando limitações nestes.

3.3.1 Método Independente

A abordagem mais simples para computar um "cube by", consiste em gerar de modo independente os 2N cuboides do "cube by" a partir do cuboide base. Dessa forma, o cuboide base é lido e processado para cada um dos 2N cuboides a ser computado.

Suponha que desejemos computar o "cube by" com os atributos {A,B,C}, este resultará nos 8 cuboides mostrados na Figura 3.7 que serão todos computados a partir do cuboide com atributos (A,B,C).

select ... group by (A,B,C)	select ... group by (A)
select ... group by (A,B)	select ... group by (B)
select ... group by (A,C)	select ... group by (C)
select ... group by (B,C)	select ... (vazio)

Figura 3.7: Cuboides obtidos a partir de um "cube by" *A,B,C*.

Esse método tem baixa eficiência e mostra pouca escalabilidade com relação ao número de atributos uma vez que sempre vai ser necessário acessar os dados do cuboide base para computar os demais cuboides. Nessa abordagem não existe também nenhuma preocupação em como as visões serão utilizadas pelas consultas dos usuários após o processo de materialização. Uma otimização inicial seria usar cuboides menores (leia-se com menos atributos) ao invés do cuboide base.

3.3.2 Método Parente

Neste método proposto por [GRAY 1996] a otimização de se buscar cuboides menores para a computação de determinado cuboide é incorporada. Considere a computação de um "cube by" com os atributos (A, B, C, D). O cuboide (A, C) pode ser computado a partir do cuboide (A, B, C) ou do cuboide (A, C, D). Em geral, um cuboide com X atributos pode ser computado de um cuboide com Y atributos se $X \subset Y$. O método Parente usa a regra de se computar um cuboide com k – 1 atributos apenas com cuboides de k atributos, uma vez que quanto mais atributos um cuboides tiver maior este será. É melhor computar o cuboide (A) de (A,B) que de (A,B,C). Podemos ver esta hierarquia de cuboides formando um grafo de dependência ilustrado na Figura 3.8.

No método *Parente* cada cuboide de k – 1 é computado de seu parente com k atributos no grafo de dependência. Essa abordagem é melhor que a do método Independente uma vez que um cuboide parente é muito menor que o cuboide base, o maior de todos os cuboides.

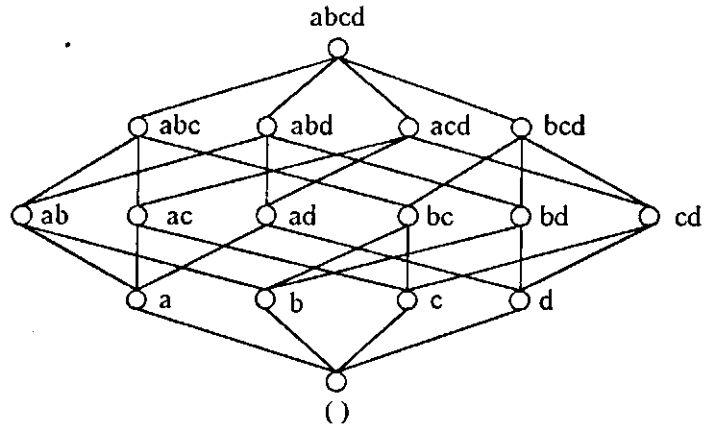


Figura 3.8: Grafo de dependência de um "cube by" *A,B,C,D*

Esse método porém não especifica que cuboide deve ser usado para derivar o cuboide desejado, uma vez que no grafo de dependência existem várias possibilidades de derivação para um cuboide. Além disso não é levado em consideração o uso que usuários farão através de consultas das visões materializadas.

3.3.3 Método Pipesort

O algoritmo Pipesort proposto em [SARAWAGI 1996] tenta otimizar o custo global de computação de um "cube by" usando estimativas do custo de derivação para determinar qual cuboide será usado para computar as tuplas de um cuboide descendente e em que ordem de atributos. A abordagem utilizada no Pipesort incorpora as seguintes otimizações:

"Smallest-parent": Esta otimização proposta em [GRAY 1996] sugere a computação de um cuboide a partir do menor cuboide previamente computado. Em geral um cuboide pode ser computado a partir de vários outros cuboides. Por exemplo, AB pode ser computado de ABC, ABD ou ABCD. Certamente ABC e ABD serão preferíveis em relação em ABCD uma vez que estes têm menos atributos e certamente são menores.

"Cache-results": Esta otimização consiste em manter na memória os resultados de um cuboide para a computação de outro cuboide reduzindo assim o número de I/O entre disco e memória. Por exemplo, enquanto ABC está na memória nós podemos computar AB a partir deste A.

"Amortize-scans": Esta otimização consiste em tentar diminuir o número de leituras em disco através da computação do número maior de "group bys" possíveis de uma única vez na memória. Por exemplo se o "group by" está armazenado em disco, podemos reduzir os custos de leitura a disco se todos os cuboides ABC, ACD, ABD e BCD forem computados em apenas uma leitura de ABCD.

Algoritmo Pipesort

O Pipesort otimiza a computação do cubo usando os dados ordenados em uma ordem particular para computar todos os "group-bys" que são prefixos dessa ordem. Por exemplo, se os atributos do "cube by" foram ordenados na forma *ABCD*, pode-se computar os group-bys *ABCD*, *ABC*, *AB* e *A* sem ordenações adicionais. O Pipesort também inclui uma otimização que permite a execução de múltiplos "group-bys" em pipeline, reduzindo assim o número de acessos a disco. No exemplo anterior, ao invés de computar os "group-bys" *ABCD*, *ABC*, *AB* e *A* separadamente, pode-se computa-los de modo pipeline da seguinte forma: cada vez que uma tupla de *ABCD* é computada, esta é propagada para computar *ABC*; cada vez que uma tupla de *ABC* é computada, esta é propagada para computar *AB*, e assim sucessivamente.

A seguir, explicaremos a entrada e a saída do algoritmo Pipesort.

Entrada: A entrada para o algoritmo Pipesort é o grafo de dependências gerada pelo "cube by", mostrada na Figura 3.9 , onde cada arco e_{ij} tem associado a si dois custos de derivação: O primeiro custo $S(e_{ij})$, custo de derivação indireta, é o custo de computar *j* a partir de *i* quando *i* não está ordenado. O segundo custo $A(e_{ij})$, custo de derivação direta, é o custo de computar *j* a partir de *i* quando *i* já estiver ordenado. O nível *k* do grafo de dependências representa todos os "group-bys" com *k* atributos. O nível 0 é representado pelo "group-by" vazio (all).

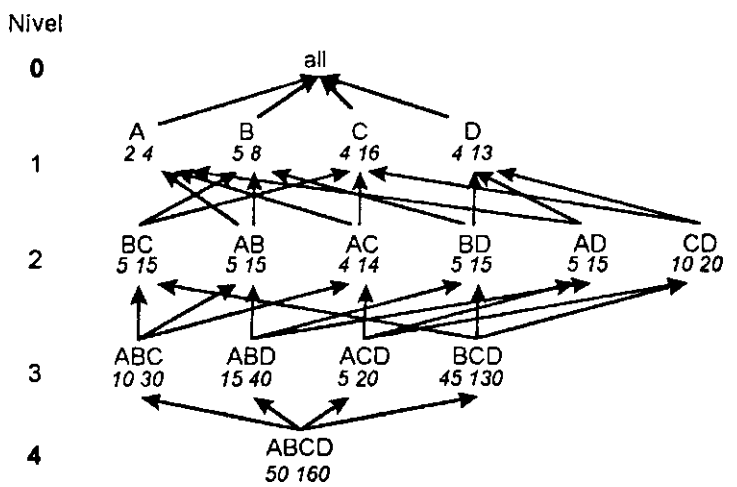


Figura 3.9: Um grafo de dependência para entrada do Pipesort

Saída: A saída para o algoritmo é um sub-grafo do grafo inicial, ilustrado na Figura 3.10, onde cada "group-by" *j* está conectado a um único "group-by" do nível anterior *i* que será usado para computar *j* com uma ordem de atributos estabelecida pelo algoritmo. Se a ordem dos atributos de um "group-by" *j* é prefixo de seu antecessor *i*, *j* pode ser computado de *i* sem ordenar *i*, e o arco e_{ij} é marcado com custo A. Caso contrário, *i* tem que ser ordenado para computar *j*, e o arco e_{ij} é marcado com custo S. O objetivo do algoritmo é encontrar um sub-grafo cuja soma global de custos de derivação seja mínima. Os vértices cujos arcos estão conectados com custo A, podem ter suas tuplas computadas através de pipeline, reduzindo o I/O entre memória e disco.

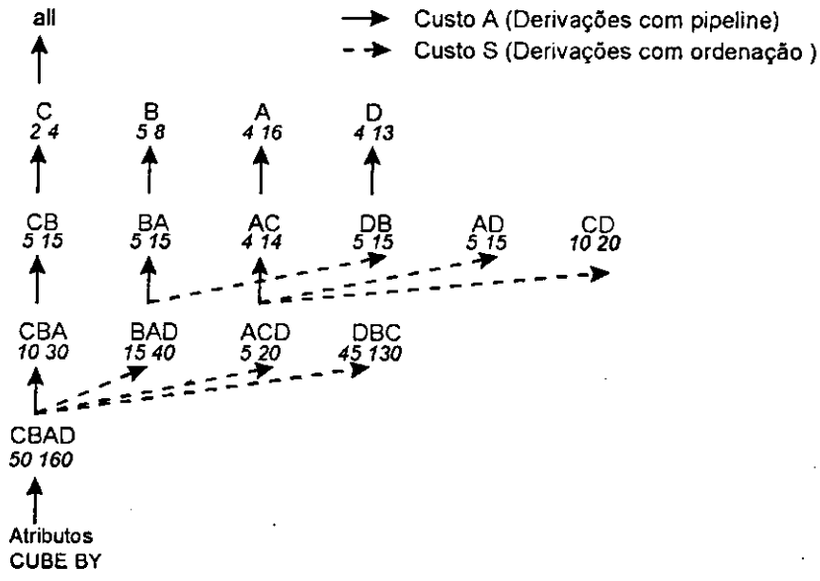


Figura 3.10: Exemplo de saida para o o Pipesort

3.4 Considerações Finais

Neste capítulo mostramos a necessidade de se ter visões materializadas em DW e os problemas associados ao operador SQL "group by" para criação destes. Vimos também uma generalização natural para o "group by" que é o operador "cube by", que trás consigo os conceitos de grafo de dependência, cuboides e computação do cubo.

Apresentamos alguns métodos para computação do cubo que apresentam a limitação dentre outras de apenas estarem interessados na criação eficiente das visões materializadas.

Capítulo 4

Algoritmo *Quasi*-Ótimo

Vimos no capítulo anterior que em um DW a presença de agregados é um requisito importante para a otimização de consultas OLAP, vimos também que o operador "cube by" é introduzido neste contexto para facilitar a construção de agregados. Agora, é preciso prover um método eficiente para a computação do cubo, que além de tentar considerar o custo global ótimo para se materializar visões, também considere o modo como estas visões serão utilizadas, isto é, não se deve perder de vista os custos das consultas OLAP ao DW que fazem uso destas visões.

Neste capítulo, apresentamos nosso algoritmo que é uma solução para a computação eficiente do cubo. Como mostraremos, o algoritmo não faz uma análise global do custo de computação, considerando apenas otimizações locais, por isso, algumas vezes uma solução com custo mínimo não será possível, pois pode haver conflitos irreconciliáveis entre os dois objetivos, criação eficiente de visões *versus* consultas eficientes ao DW.

Estamos propondo um algoritmo chamado de *Quasi*[1]-Ótimo que, ou otimiza o custo global do cálculo de um "cube by", ou então o custo global calculado aproxima-se o máximo possível do custo ótimo. O cálculo pode ser *total* (todos os cuboides) ou *parcial* (alguns cuboides). Para o cálculo do custo global, o algoritmo considera dois custos, custos de derivação direta e indireta de um cuboide a partir de outro cuboide, além de levar em conta o uso que os usuários fazem dos cuboides através de consultas.

Iniciamos com a apresentação da opção do algoritmo para o cálculo total de um "cube by". Descrevemos nas seções 4.1 e 4.2 a entrada e a saída do algoritmo, seguindo na seção 4.3 as principais idéias empregadas na construção do mesmo. Na seção 4.4, mostramos como o algoritmo calcula parcialmente um "cube by". Finalizamos com um exemplo ilustrando tudo

[1] – Palavra latina que significa *proximamente*.

o que foi explicado. O pseudocódigo do algoritmo encontra-se no Apêndice II desta Dissertação.

4.1 Grafo de Derivação

A entrada do algoritmo é um *grafo de derivação* $(G,\prec)$, onde cada vértice v_k de G é um cuboide de um "cube by". Os arcos de $\prec$ são *arcos dirigidos de derivação* (v_i,v_j), $i \neq j$, de v_i a v_j, se v_j tiver todas as dimensões de v_i, menos uma. Cada arco é rotulado com dois custos de derivação, *Indireto* e *Direto*. Dado o custo direto $D(v_i,v_j)$, isto significa que v_j é um prefixo de v_i, consequentemente v_j pode ser calculado diretamente de v_i, e simultaneamente com v_i. Por outro lado, se v_j não é um prefixo de v_i, então o custo de calcular v_j de v_i é indireto, $I(v_i,v_j)$, significando que v_i deve ser reordenado para daí gerar v_j diretamente. É então óbvio que o custo *D* para derivar um cuboide é sempre menor que o custo *I* para o mesmo fim.

Além do grafo de derivação, devem ser informadas, para alguns ou todos os cuboides, as *consultas ordenadas frequentes* a estes cuboides. Dado um cuboide, pode-se informar mais de uma consulta a ele.

As consultas ordenadas são aquelas que apresentam a cláusula "order by" na estrutura do seu "select". Isto é, as tuplas de retorno de uma consulta ordenada têm a ordem de acordo com os atributos do "order by". Assim como um cuboide pode ser identificado apenas com os atributos de agregação, as consultas ordenadas também são identificadas apenas com os atributos da cláusula "order by".

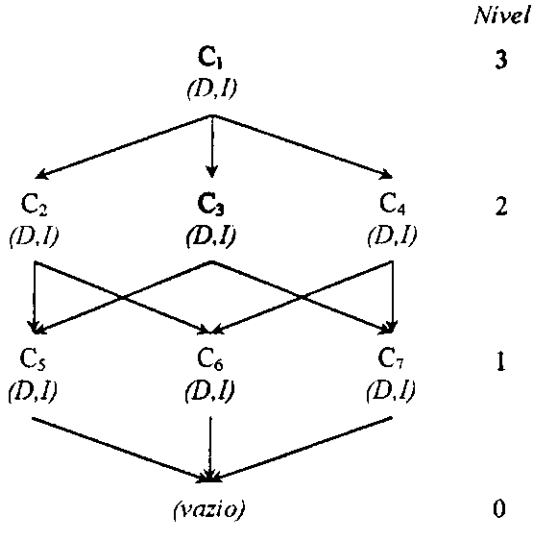


Figura 4.1: Um grafo de derivação, entrada para o algoritmo *Quasi*-Ótimo.

Seja o grafo de derivação da Figura 4.1 na página anterior. Considere o cuboide C2: os arcos de C2 (um cuboide *derivador*) para os cuboides C5 e C6 (cuboides *derivados*), respectivamente, são arcos de derivação, que indicam que C5 e C6 são derivados de C2, ou seja, C5 (C6) tem todas as dimensões de C2, menos uma. Estes arcos são rotulados de *(D, I)*: o custo de derivação de C5 (C6) é *D*, se C5 (C6) for prefixo de C2; se C5 (C6) não for prefixo de C2, então o custo é *I*.

É feita a hipótese bastante razoável de que, dado um cuboide derivador e todos os que lhe são derivados, todos os arcos (cuboide derivador – cuboide derivado) têm o mesmo custo *D* e o mesmo custo *I*. Assim, por exemplo, para os arcos ligando C2 a C5 e a C6, o valor de *(D, I)* é comum aos dois. Para evitar a repetição dos mesmos custos em vários arcos, optamos por colocá-los abaixo do cuboide derivador.

O nível *k* de um grafo de derivação representa todos os cuboides com *k* dimensões. Note que, dado um "cube by" de *n* dimensões, o número de níveis do grafo (G, $\prec$) é sempre *n+1* (nível 0, nível 1, ..., nível n). O nível 0 representa o cuboide *vazio*. O nível *n* representa o cuboide derivador dos cuboides dos outros níveis, direta ou indiretamente. Pode-se concluir pelo número de níveis que o grafo da Figura 4.1 é para um "cube by" de três dimensões.

4.1.1 Modelo de Custos

O modelo de custos apresentado em [HARINARAYAN 1996] serviu de base para o projeto do nosso modelo de custos de derivação. Utilizamos a idéia base de Harinaravan et al. de que o tempo para responder uma consulta *Q* está relacionado ao tamanho da visão *V* a partir da qual a consulta é respondida.

Seja o grafo de derivação (G, $\prec$), para derivar um cuboide *C* deste grafo utilizamos um cuboide antecessor C_A que encontra-se em um nível superior. Desta forma temos que percorrer toda a tabela de C_A para computar *C*. Definimos que o custo de derivação de um cuboide *C* a partir de um cuboide C_A é função do tamanho do cuboide C_A e do fato das ordens de C_A e *C* coincidirem ou não.

$$Custo(C_A, C) = C_T + C_O$$

O custo de derivar o cuboide *C* é a soma do *tamanho* do cuboide C_A usado para derivar *C* representado por C_T e uma constante C_O relacionada ao tempo utilizado para *ordenar* C_A. Esta constante está associada às características do hardware do DW (tamanho das memórias estável e volátil, velocidade do processamento). Quando consideramos o custo de derivação

direto o valor da constante C_O é nulo, pois a ordem de C_A e C coincidem, restando apenas o trabalho de derivação de C.

Os trabalhos de [SHUKLA 1996] e [HAAS 1995] apresentam um conjunto de procedimentos para se estimar o tamanho de um cuboide baseando-se em métodos estatísticos e analíticos.

Percebemos que existem outros fatores que influenciam o custo de derivação que não estão sendo considerados aqui. Entre eles, destacamos a presença ou não de índices nos atributos.

Modelos de custos mais sofisticados são certamente possíveis de serem formalizados, porém acreditamos que o modelo adotado nos permite o desenvolvimento de algoritmos com um bom desempenho.

4.2 Grafo de Derivação *quasi*-ótimo

Um dos objetivos do algoritmo é fazer com que as consultas fornecidas fiquem casadas com os cuboides correspondentes do grafo de derivação. O outro objetivo é explorar ao máximo a possibilidade de derivar cuboides direta e simultaneamente de outros, visando diminuir os custos de criação das visões materializadas. Atingidos plenamente os dois objetivos, então o custo global de cálculo de um cubo será ótimo.

Infelizmente, pode acontecer que os dois objetivos sejam conflitantes. Se este for o caso, então o algoritmo procurará casar todas as consultas, e ainda derivando direta e simultaneamente o maior número possível de cuboides, significando um custo global *quasi*-ótimo.

A saída do algoritmo é um sub-grafo, chamado de *grafo de derivação* quasi-*ótimo*, do grafo de derivação de entrada do algoritmo, no qual a soma global de todos os custos D e I é *mínima*, ou é um valor matematicamente o mais próximo possível do mínimo considerando níveis consecutivos, em relação aos dois objetivos fixados. Neste grafo, cada cuboide, à exceção do cuboide no nível mais alto, está ligado a um *único* cuboide no nível superior (o cuboide derivador), a ligação estando associada a um custo D ou I. O cuboide derivador de todos os outros cuboides, direta ou indiretamente, é o cuboide no nível mais alto do grafo de derivação *quasi*-ótimo. É importante perceber que trabalhamos com níveis consecutivos do grafo de derivação fazendo nossa otimização nestes níveis.

Um exemplo de um grafo de derivação *quasi*-ótimo (não é o único) para o grafo de derivação da Figura 4.1 é mostrado abaixo na Figura 4.2. Para fazer a distinção entre um valor *D* e um valor *I*, os arcos com custos *D* são cheios, enquanto que os arcos com custos *I* são pontilhados.

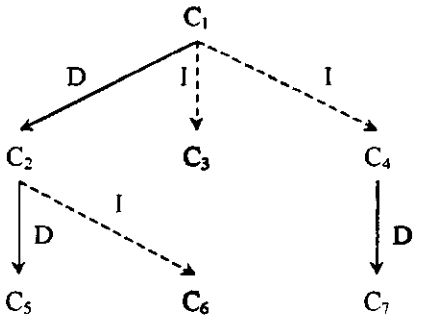


Figura 4.2: Um grafo de derivação *quasi*-ótimo.

Suponha que fossem informadas uma consulta ordenada a C1, outra a C2, e ainda uma a C5 e uma a C6. Se os dois objetivos do algoritmo tivessem sido plenamente alcançados, então entre todas as combinações pertinentes dos arcos do grafo da Figura 4.1, a combinação de arcos do grafo da Figura 4.2 teria um custo mínimo (em outras palavras, a soma de todos os custos dos arcos seria mínima), e, além disto, as consultas seriam casadas com os cuboides respectivos.

Em caso de conflito entre os dois objetivos, as consultas seriam ainda casadas com os seus cuboides mas então a soma de todos os custos dos arcos não seria exatamente a mínima. É importante dizer que em situações de conflito entre os dois objetivos o algoritmo privilegia as consultas fazendo a reordenação do cuboide de acordo com a ordem da consulta, pois acreditamos que o processo de consulta é mais importante que o de criação.

Para entender de que maneira um grafo de derivação *quasi*-ótimo é gerado, precisamos descrever como o algoritmo Q*uasi*-Ótimo funciona.

4.3 Descrição do Algoritmo

Para a obtenção de um grafo de derivação *quasi*-ótimo a partir de um grafo de derivação $(G, \prec)$, o algoritmo Q*uasi*-Ótimo trabalha por iterações: cada iteração considera dois níveis consecutivos de $(G, \prec)$, iniciando no nível $k = 0$ até o nível $k = n-1$, onde n é o número de

dimensões do "cube by" representado por $(G,\prec)$. Para cada nível k, o algoritmo encontra o menor custo de *derivar* o nível k do nível $k+1$. Este problema é resolvido com a ajuda de um algoritmo que implementa o chamado *método Húngaro* [PAPADIMITRIOU 1982], que determina o que se chama de associação mínima em um grafo bipartido dirigido.

Para dar uma idéia do que é o método Húngaro, a Figura 4.3(a) ilustra um grafo bipartido dirigido (V_1,V_2, A), onde as duas partes são os conjuntos de vértices $V_1=\{v_{11}, v_{12}, ..., v_{1i}\}$ e $V_2=\{v_{21}, v_{22}, ..., v_{2j}\}$. Cada arco $(v_{1i},v_{2j}) \in A$ tem um peso, $c_k > 0$.

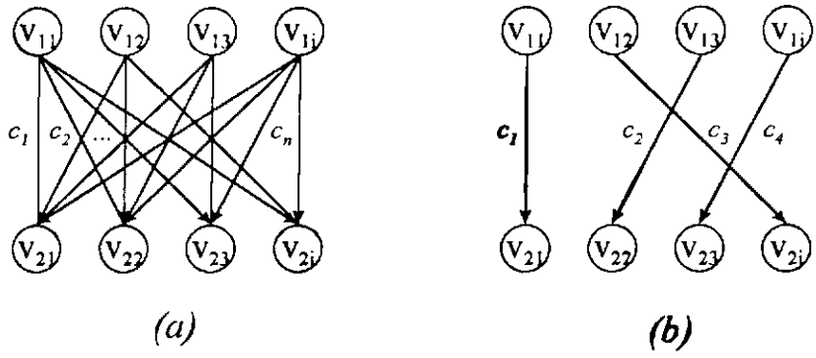


Figura 4.3: Grafo bipartido (a), e uma associação mínima entre as partes (b).

O método Húngaro encontra as associações (arcos, em verdade) um para um entre os elementos de V_1 e todos os elementos de V_2, tal que o valor da soma dos c_k das associações é mínima (Figura 4.3(b)).

Adaptando o método Húngaro ao nosso algoritmo, os elementos de V_1 são construídos a partir dos cuboides do nível $k+1$, enquanto que os elementos de V_2 o são dos cuboides do nível k do grafo de derivação.

Para cada iteração, o algoritmo *quasi*-ótimo utilizando a rotina *grafo_bipartido* primeiramente transforma o nível $k+1$ do grafo de derivação fazendo k cópias de cada vértice neste nível. Estas cópias são feitas a fim de representar todas as possibilidades de derivação. Os vértices copiados no nível $k+1$ são ligados a vértices do nível k com arcos representando os custos indiretos de derivação dos vértices do nível k (arcos pontilhados). Os arcos partindo dos vértices originais no nível $k+1$ representam custos diretos (arcos cheios).

Para exemplificar, mostramos na Figura 4.4 as cópias dos cuboides dos níveis *1* e *2* do grafo de derivação da Figura 4.1 (as cópias aparecem sublinhadas), com os novos arcos.

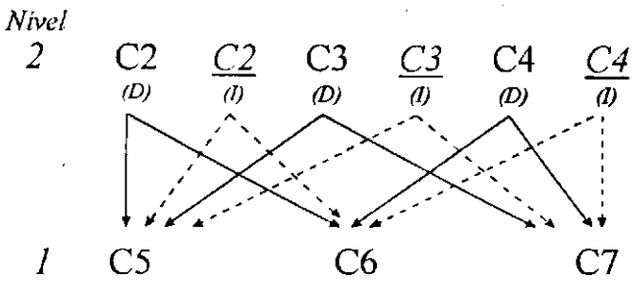


Figura 4.4: Um grafo bipartido extraído do grafo de derivação da Figura 4.1

Considere, por exemplo, o cuboide C5. Quatro arcos "chegam" a ele, representando todas as possibilidades para sua derivação: derivação direta de C2 ou de C3, ou derivação indireta de C2 ou de C3.

Neste ponto, chegamos a um grafo com as mesmas características de um grafo bipartido dirigido de entrada para o método Húngaro. Esquecendo por enquanto as consultas ordenadas fornecidas junto com o grafo de derivação, a questão que se coloca agora é: que subconjunto dos arcos do grafo da Figura 4.4 deve ser escolhido, tal que a soma dos seus custos é mínima? Note que igual raciocínio deve ser aplicado aos outros pares de níveis consecutivos (níveis *0* e *1*, e níveis *2* e *3*).

Com o auxílio da rotina *associação_de_custo_mínimo* do algoritmo *quasi*-ótimo, que implementa o método Húngaro, podemos chegar à(s) associação(ões) de custo mínimo exibidas na Figura 4.5 entre os dois conjuntos de cuboides do grafo bipartido da Figura 4.4

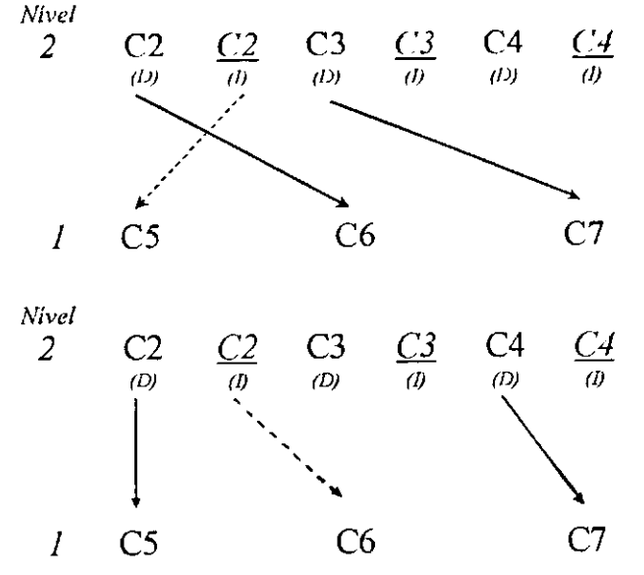


Figura 4.5: Associações de custo mínimo para o grafo bipartido da Figura 4.4

Como vemos, pode existir mais de uma associação com custo mímimo. Tiraremos proveito disto testando exaustivamente todas as alternativas para resolver da melhor forma possível conflitos entre a ordem dos cuboides e a ordem das consultas.

A entrada para a rotina *associação de custo mínimo* é a representação matricial do grafo bipartido que deve ter em sua linha 1 e coluna 1 os valores dos cuboides do nível k e k+1 respectivamente. As interseções dos valores destas linhas e colunas representam os valores dos custos *D* e *I*. Quando a interseção dos cuboides das linhas e colunas não corresponder a um arco colocamos um valor representando infinito na célula da matriz.

O próximo passo é fazer o casamento[2] entre as consultas ordenadas para o nível k+1 e os cuboides deste nível. Como será visto a seguir podem surgir conflitos entre as ordens dos cuboides e das consultas ordenadas. É necessário um procedimento que encontre estes conflitos de ordem e os solucione privilegiando as consultas.

Na próxima subseção mostraremos como detectar e solucionar conflitos na construção de um grafo de derivação *quasi*-ótimo.

4.3.1 Detecção e Solução de Conflitos

Neste momento chegamos à última parte do algoritmo que vai tratar da detecção e solução de conflitos entre as ordens dos cuboides e das consultas ordenadas freqüentes.

Existem dois tipos de conflito que são detectados pela rotina *detector_de_conflitos* do algoritmo *Quasi*-Ótimo: o primeiro é um conflito entre as diversas ordens de consulta a um mesmo cuboide; o segundo tipo de conflito é entre a ordem dos cuboides e a ordem das consultas ordenadas.

Para ilustrar o primeiro tipo de conflito e sua solução, imagine que sejam informadas duas consultas de ordem diversa ao cuboide C2. É evidente que somente uma delas poderá estar casada com C2, daí o conflito entre a ordem da outra com a ordem de C2.

A solução do conflito, pela rotina *solucionador_de_conflitos*, é a seguinte: a rotina escolhe a consulta que é casada com C2, mantendo a associação de custo mínimo entre os cuboides dos níveis *1* e *2*. Para a outra consulta, uma criteriosa escolha dos índices de acesso a C2 pode tornar aceitável seu custo de processamento.

[2] - Dizemos que uma consulta é casada com uma cuboide quando as ordens da consulta e do cuboide coincidem.

Para o segundo tipo de conflito e sua solução, considere que as ordens de C4 e da consulta a ele submetida sejam diferentes. Neste caso, a ordem de C4 é modificada para "casá-lo" com a consulta e, consequentemente, o custo direto de derivação de C7 é trocado pelo custo indireto de derivação de C7 (ou o arco de C4 para C7 muda de cheio para pontilhado). Como resultado, a associação entre os cuboides dos níveis *1* e *2* deixa de ser de custo mínimo. Note que a solução para os conflitos sempre privilegia as consultas.

É importante assinalar que, muitas vezes, um conflito do segundo tipo é resolvido ainda mantendo-se a associação de custo mínimo. Observe de novo o grafo bipartido da Figura 4.5: poderíamos trocar a ordem de C2, e portanto invertendo os custos de derivação de C5 (um arco pontilhado, agora) e de C6 (um arco cheio, agora) – é óbvio que a soma de todos os custos permanece inalterada.

As rotinas *detector_de_conflitos* e *solucionador_de_conflitos* são aplicadas para cada associação gerada pela rotina *associação_de_custo_mínimo*. A solução final para os níveis k+1 e k é aquela mínima ou a mais próxima possível da associação de custo mínimo.

Mais uma vez destacamos que no instante em que o algoritmo tem de escolher entre a ordem de um cuboide ou de uma consulta este escolherá a ordem da consulta, pois acreditamos que de nada adianta economizar o tempo de materialização das visões se não se levar em conta as consultas que poderiam utilizar as mesmas. Se não houvesse este cuidado, poderia ser que (1) uma visão viesse a ser muito pouco utilizada, ou sequer ser utilizada; e (2), uma visão e uma consulta que a utilizasse freqüentemente teriam ordens incompatíveis, levando à ordenação de arquivos temporários muito grandes, onerando desta forma excessivamente o custo de processamento da consulta.

4.4 Cálculo Parcial de uma Operação Cube By

Muitas vezes, o administrador de um DW está interessado em materializar somente algumas visões, dentre todas as visões correspondendo a cada um dos cuboides de um "cube by". Se esta for a sua opção, a saída do algoritmo é um grafo *parcial* de derivação *quasi*-ótimo, construído a partir do grafo de derivação *quasi*-ótimo se a opção fosse o cálculo total do "cube by". Este grafo parcial consiste somente dos cuboides correspondendo às visões materializadas desejadas, além dos cuboides intermediários necessários para gerá-los, e dos arcos pertinentes.

O refinamento seguinte consiste em trocar arcos pontilhados do grafo parcial por arcos cheios, onde isto for possivel. Forneceremos mais detalhes desta opção do algoritmo por meio do exemplo da subseção seguinte.

4.5 Um Exemplo de Aplicação do Algoritmo

Nesta seção iremos apresentar um exemplo detalhado ilustrando a aplicação do algoritmo *Quasi*-Ótimo em um DW relacional com quatro dimensões básicas, Loja (*L*), Produto (*P*), Promoção (*Pr*) e Tempo (*T*).

Sejam as sete consultas ordenadas muito frequentes plp_r[3], lp_rp, p_rpt, pl, lp_r, p e l ao DW relacional com a tabela *Fatos(L#, P#, Pr#, T#, atributos_de_agregação)* e as tabelas de dimensão *Loja*, *Produto*, *Promoção* e *Tempo*. O administrador do DW diante destas consultas deseja então materializar as visões[4] LPPr, PPrT, LP, LPr, P e L, do "*cube by* L, P, Pr, T", de modo a garantir o bom desempenho das consultas. Observe que estamos interessados em uma materialização parcial do cubo, poderíamos estar interessados no calculo total do cubo a fim de também assegurar o bom desempenho às consultas.

A entrada para o algoritmo *quasi*-ótimo é o grafo de derivação do *cube by* L, P, Pr, T da Figura 4.6, junto com a lista de consultas ordenadas dada.

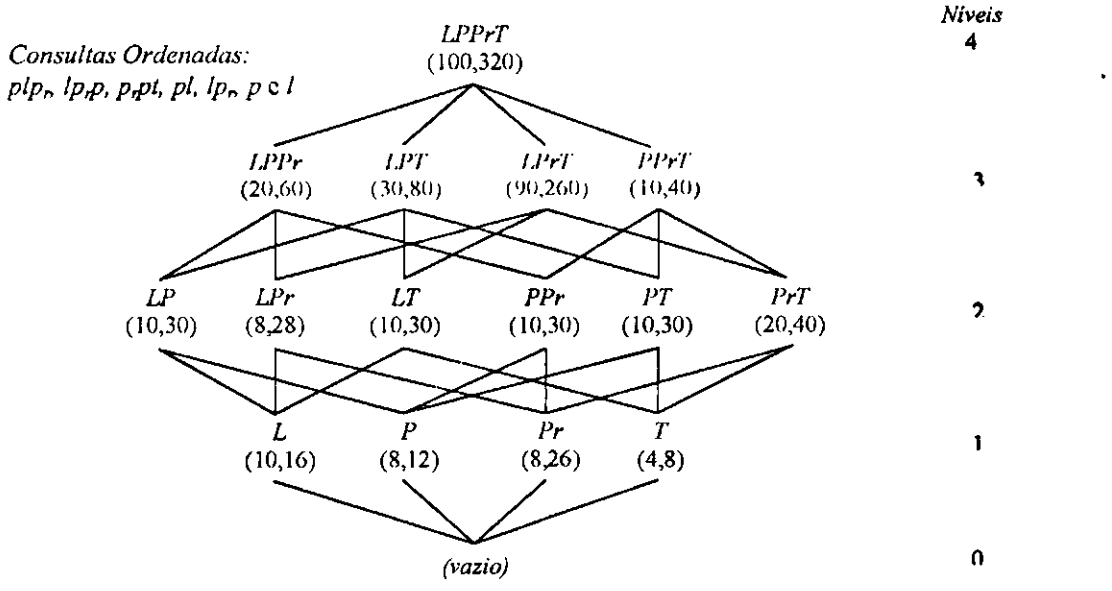


Figura 4.6: Grafo de derivação para "cube by L, P, Pr, T".

[3] - A notação plp_r indica a consulta ordenada *select ... order by P, L, Pr*.

[4] - É importante que se perceba que neste momento estamos interessados em materializar os cuboides que tem estes atributos. As ordens em que estes 6 cuboides serão materializados vai depender das consultas ordenadas e dos custos de derivação, e somente serão determinadas no final do processo.

Cada par de números é do tipo (D, I). Assim, por exemplo, (20, 60) sob o cuboide LPPr significa que o custo para derivar diretamente LP, ou LPr, ou PPr de LPPr é 20, enquanto que o custo para derivar indiretamente LP, ou LPr, ou PPr de LPPr é 60.

A primeira iteração do algoritmo com k = 0 considera os níveis 0 e 1 e as consultas *l* e *p* que são respondidas com os cuboides do nível 1. É fácil de ver que nesta iteração o trabalho do algoritmo é mínimo para encontrar a associação mínima e casar as consultas ordenadas, uma vez que não existem combinações possíveis para derivação dos cuboides no nível 0, e os cuboides do nível 1 têm apenas uma única ordem possível.

Logo ao final da primeira iteração teremos a seguinte parte do grafo de derivação *quasi*-ótimo mostrado na Figura 4.7 onde os cuboides com consultas ordenadas casadas estão sublinhados.

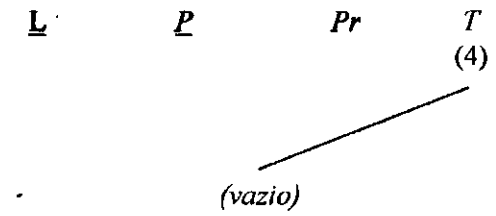


Figura 4.7: Grafo de derivação *quasi*-ótimo para os níveis 0 e 1.

Em seguida, considerando os níveis 1 e 2 com k = 1 e as consultas *pl* e lp_r, o algoritmo faz uma cópia adicional de cada cuboide no nível 2 ligando-os aos cuboides do nível 1 com arcos pontilhados representando custos indiretos como está sendo mostrando na Figura 4.8. Os arcos originais contínuos representam os custos de derivação diretos.

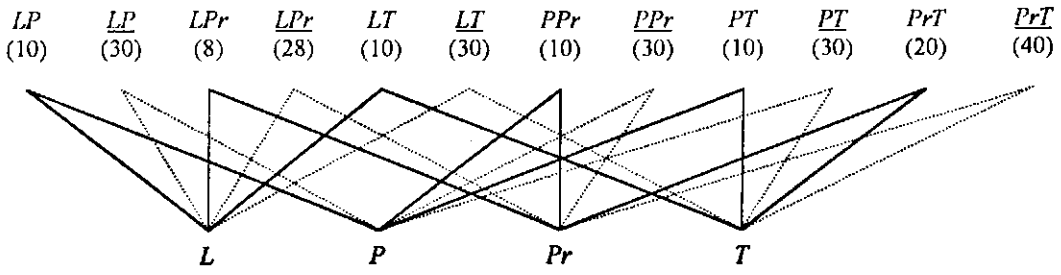


Figura 4.8: Grafo bipartido para níveis 1 e 2

O grafo bipartido da Figura 4.8 pode ser representado através de uma matriz bidimensional onde a linha 1 e a coluna 1 contêm os valores dos cuboides do nível 1 e 2 respectivamente e a interseção destas linhas e colunas representam o valor dos custos diretos e

indiretos. Esta matriz mostrada na Figura 4.9 servirá de entrada para o método Húngaro que encontrará uma associação de custo mínimo (valores destacados na matriz) com custo total igual a 38.

	L	P	Pr	T
LP	10	**10**		
LP	30	30		
LPr	**8**		8	
LPr	28		28	
LT	10			10
LT	30			30
PPr		10	**10**	
PPr		30	30	
PT		10		**10**
PT		30		30
PrT			20	20
PrT			40	40

Figura 4.9: Representação matricial do grafo bipartido

É importante dizer que a rotina *associação_de_custo_mínimo* encontrou nesta iteração 7 associações, com custo mínimo, candidatas a entrar no grafo de derivação *quasi*-ótimo. A associação escolhida é aquela que casa com o maior número de consultas. Neste exemplo as células, que representam arços, da matriz da Figura 4.9 destacadas são aqueles escolhidos dentre as soluções possíveis como resposta pois casam com as consultas frequentes.

No final da segunda iteração do algoritmo *Quasi*-Ótimo teremos mais uma parte acrescentada ao grafo de derivação *quasi*-ótimo mostrado na Figura 4.10. Mais uma vez os cuboides com consultas ordenadas aparecem sublinhados.

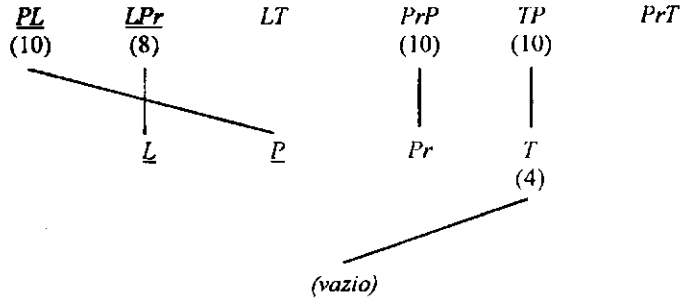


Figura 4.10: Parte do grafo de derivação *quasi*-ótimo para os níveis 0, 1 e 2.

A próxima iteração vai considerar os níveis 2 e 3 e as consultas ordenadas plp_r, lp_rp, p_rpt. Observe que as consultas ordenadas plp_r, lp_rp são feitas a um mesmo cuboide *PLPr* que vai implicar em um conflito do tipo 1 cuja solução vai casar apenas uma dessas consultas com o cuboide correspondente.

Realiza-se duas cópias de cada vértice no nível 3 e as respectivas ligações dos vértices novos e existentes com seus custos apropriados. Assim como ocorreu na iteração anterior o grafo bipartido resultante pode ser representado através de uma matriz cuja associação de custo mínima gerada pelo método Húngaro é mostrada na Figura 4.11.

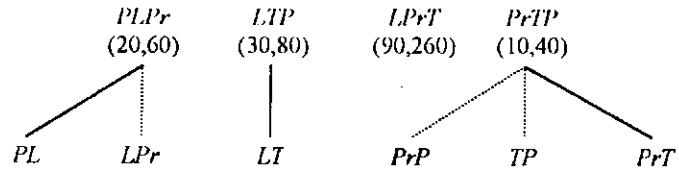


Figura 4.11: Associação de custo mínimo para o grafo bipartido dos níveis 2 e 3.

Observe que o cuboide *PLPr* foi escolhido nesta ordem privilegiando a consulta plp_r, em detrimento a consulta lp_rp, ocorrendo nesta situação um conflito entre as diversas ordens de consulta a um mesmo cuboide. Por outro lado, o cuboide PrTP está em conflito com a consulta p_rpt, neste caso a ordem do cuboide é modificada a fim de que este case com a consulta ordenada, consequentemente os custos dos arcos de derivação devem ser modificados. O resultado da terceira iteração juntamente com o restante do grafo de derivação *quasi*-ótimo é mostrado na Figura 4.12.

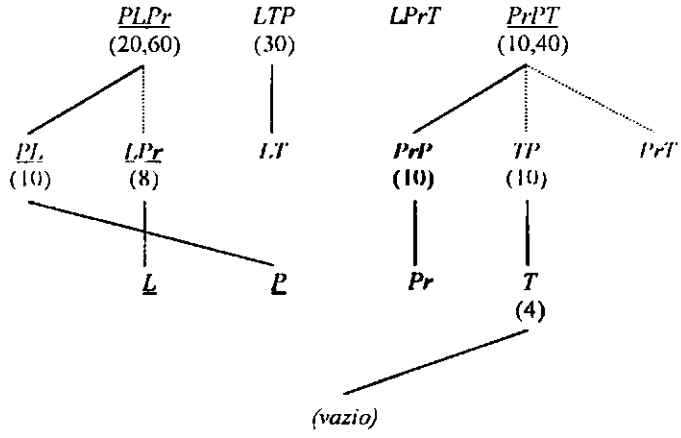


Figura 4.12: Parte do grafo de derivação *quasi*-ótimo para os níveis 0, 1 ,2 e 3.

Observe que o conflito entre a ordem da consulta e o cuboide foi resolvido ainda mantendo-se a associação de custo mínimo, pois a mudança nos arcos mantém ainda a soma dos custos de derivação igual.

Finalmente na última iteração onde são considerados os níveis 3 e 4 não havendo consultas ordenadas neste nível, o trabalho do algoritmo *quasi*-ótimo é mais simples, cabendo

apenas a escolha de um arco com custo de derivação direta enquanto os demais terão custo de derivação indireta.

A Figura 4.13 mostra o grafo de derivação *quasi*-ótimo completo gerado pelo algoritmo, cuja semântica é um plano de custo ótimo de construção de todas as visões materializadas de "cube by L, P, Pr, T", levando em conta as consultas dadas e a possibilidade de gerar simultaneamente várias visões.

Entre as várias alternativas para a construção de planos ótimos, o algoritmo escolheu este plano, que privilegia a consulta plp_r em detrimento de lp_rp, e em que há os casamentos entre P e p, L e l, PL e pl, LPr e lp_r, PLPr e plp_r, PrPT e p_rpt .

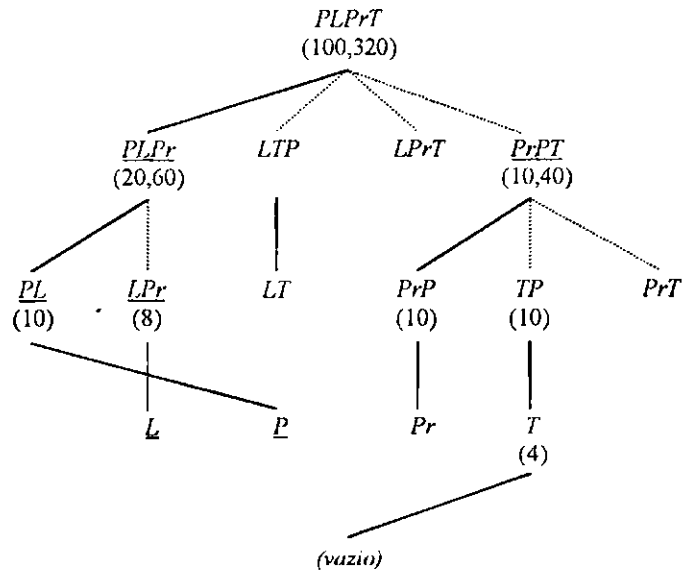


Figura 4.13: Um plano ótimo do *cube by* L, P, Pr, T, para a materialização de visões

Como exemplo de um plano que não é exatamente ótimo, acrescente-se a consulta ordenada lp_rpt às consultas dadas. Neste caso, teríamos o cuboide LPrPT, o custo de derivar PLPr de LPrPT seria indireto (320, em lugar de 100), e o custo global não seria mínimo.

Do grafo da Figura 4.13, o algoritmo gera o grafo parcial da Figura 4.14, onde somente os cuboides correspondentes às consultas dadas, bem como alguns cuboides intermediários necessários aos primeiros, são apresentados.

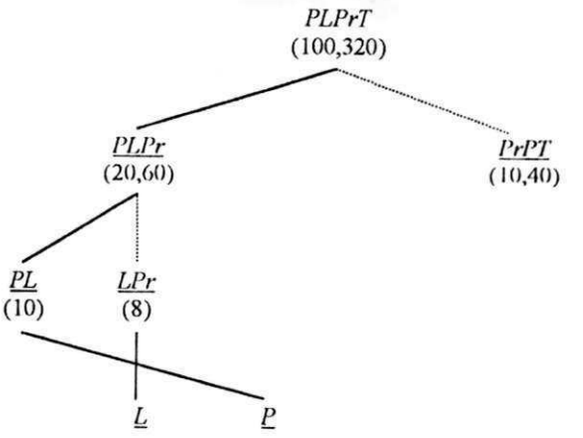


Figura 4.14: Grafo para o cálculo parcial de cube by L, P, Pr, T.

Note que o cuboide intermediário PLPrT é necessário para derivar direta e simultaneamente os cuboides PLPr, PL e P.

4.6 Conclusões

Apresentamos o nosso algoritmo cujo objetivo principal é permitir a construção de planos que servirão para a criação e utilização eficiente de visões em DW. No próximo capítulo mostraremos alguns detalhes na implementação de um protótipo para o algoritmo *Quasi*-Ótimo.

Capítulo 5

Aspectos da Implementação do Algoritmo *Quasi*-Ótimo

Neste capítulo, mostraremos alguns detalhes da implementação de um protótipo para o algoritmo *Quasi*-Ótimo descrito no capítulo anterior. A linguagem utilizada para desenvolver os programas foi Pascal, utilizando como ferramenta de desenvolvimento o *Borland Delphi 3.0*. O motivo principal para a escolha da linguagem Pascal e do ambiente de desenvolvimento *Delphi* foi a facilidade para o desenvolvimento de aplicações, e o suporte nativo a banco de dados.

Durante o processo de desenvolvimento do protótipo tivemos algumas dificuldades, principalmente na implementação de uma biblioteca de funções para a criação e manipulação das matrizes que são amplamente utilizadas pelo método Húngaro. A implementação do método Húngaro também exigiu um trabalho demasiado, uma vez que o pseudocódigo disponível [PAPADIMITRIOU 19982] estava muito generalizado e com poucos detalhes que ajudassem na atividade de implementação.

Mostraremos a seguir as estruturas utilizadas para armazenar os grafos de entrada e saída gerados pelo algoritmo. Em seguida, mostraremos as telas e comentaremos as principais funções implementadas no protótipo. Finalizaremos o capítulo com um breve comentário sobre a complexidade do algoritmo.

5.1 Representação do Grafo de Derivação

Utilizamos duas tabelas relacionais implementadas em *Paradox* para representar o Grafo de Derivação $(G, \prec)$. As tabelas *Vertices.db* e *Arcos.db* têm compondo as suas respectivas estruturas os atributos descritos abaixo nas Tabelas 5.1 e 5.2.

Nome	Descrição do Atributo
NU_VERTICE	Número seqüencial para identificar um vértice
NM_VERTICE	Nome do vértice. Os atributos que compõem os vértices de $(G,\prec)$ devem estar separados por vírgulas, ex.: L,P,Pr,T. O motivo disso é permitir que o algoritmo possa determinar a fronteira entre um atributo e outro.
NIVEL	Nível em que o atributo encontra-se em $(G,\prec)$.

Tabela 5.1: Estrutura da Tabela *Vertices.db*

Nome	Descrição do Atributo
Vt_Inicial	Este atributo contém um identificador de *Vertices.db* representando o vértice inicial em um arco.
Vt_Final	Este atributo contém um identificador de *Vertices.db* representando o vértice final em um arco.
Custo_D	Custo de Derivação Direto para o arco (Vt_Inicial,Vt_Final)
Custo_I	Custo de Derivação Indireto para o arco (Vt_Inicial,Vt_Final)

Tabela 5.2: Estrutura da Tabela *Arcos.db*

As saídas geradas pelo protótipo são duas tabelas do tipo *Paradox* chamadas *Arcos_Saida.db* e *Vertices_Saida.db*. A tabela *Vertices_Saida.db* tem estrutura igual a de *Vertice.db*, porém esta contém o conjunto dos vértices que forma o grafo de derivação *quasi-ótimo*. Estes vértices contém uma nova ordem nos atributos, que deve casar com as consultas ordenadas e eventualmente com o vértice de nível superior, que conecta-se a este com custo de derivação direto.

A tabela *Arcos_Saida.db* tem a estrutura da tabela de entrada *Arcos.db*, menos os atributos *Custo_D* e *Custo_I*, sendo estes substituídos pelo atributo *Custo* que tem em seu conteúdo o valor escolhido (Custo D ou I) para a derivação representada pelo arco (Vt_Incial, Vt_Final). A Tabela 5.3, abaixo mostra a estrutura da tabela *Arcos_Saida.db*.

Nome	Descrição do Atributo
Vt_Inicial	Este atributo contém um identificador de *Vertices.db* representando o vértice inicial em um arco.
Vt_Final	Este atributo contém um identificador de *Vertices.db* representando o vértice final em um arco.
Custo	Custo de Derivação Direto ou Indireto escolhido pelo algoritmo *Quasi-Ótimo*

Tabela 5.3: Estrutura da Tabela *Arcos_Saida.db*

A solução empregada para representar os grafos descrita acima mostrou-se muito eficiente para o propósito do algoritmo *Quasi*-Ótimo.

5.2 Uso do Protótipo

A Figura 5.1, ilustra a tela principal do nosso protótipo, nela estão presentes as opções utilizadas para a execução do algoritmo *Quasi*-Ótimo. A seguir descreveremos os principais componentes da Figura 5.1 e suas respectivas funcionalidades.

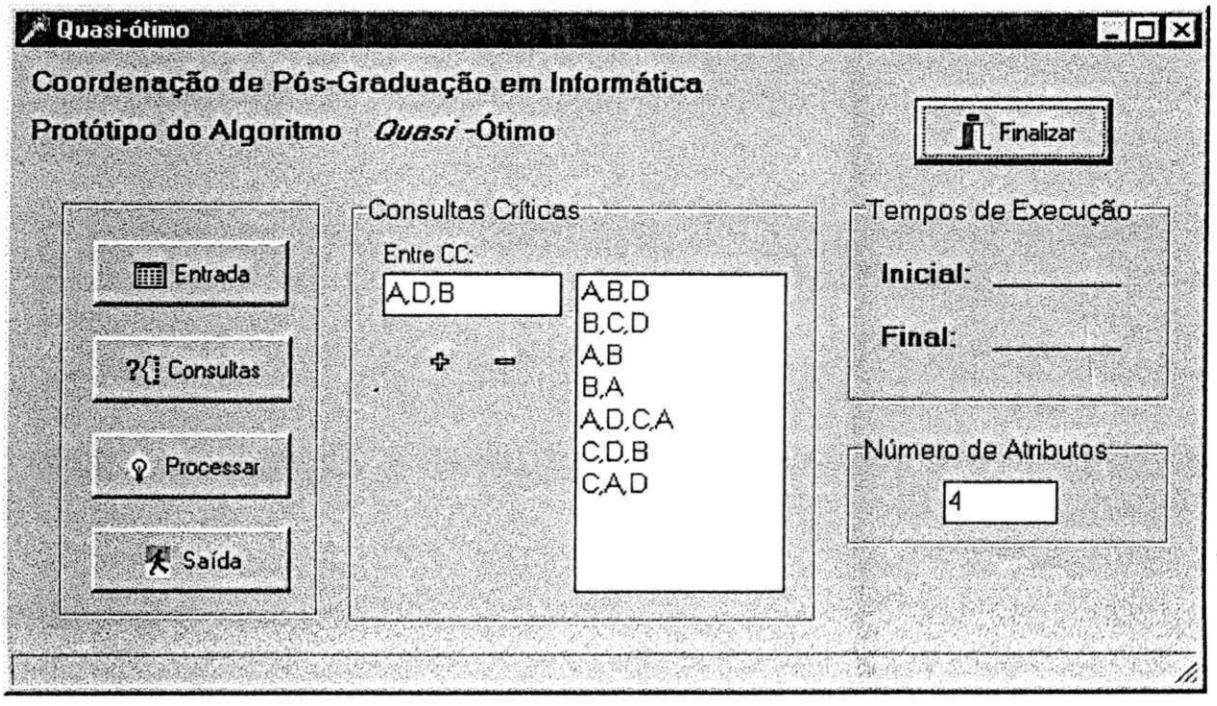
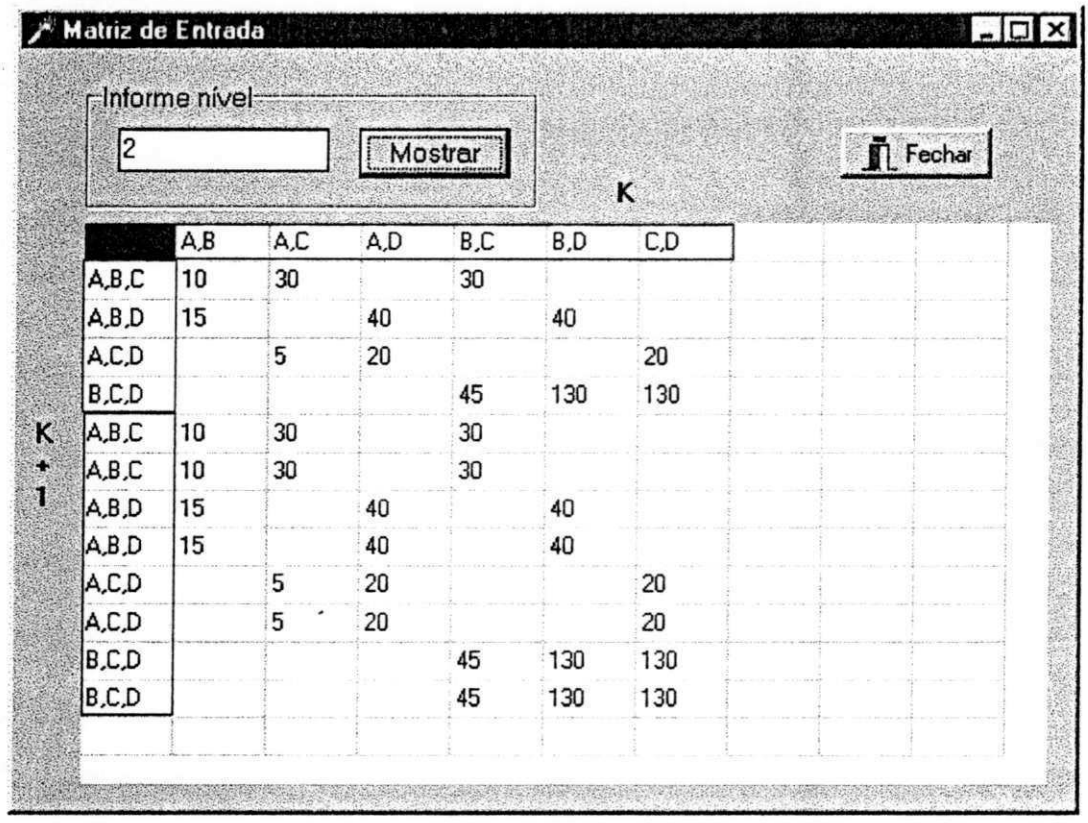


Figura 5.1: Tela principal do Protótipo Implementado

Através do botão *Consultas*, o usuário acessa a janela onde são informadas as consultas críticas, consultas estão que são armazenadas em uma lista encadeada. Estas consultas devem ser informadas com os atributos separados por vírgula, a fim de facilitar o desmembramento destas e a respectiva determinação de que cuboide casa com a consulta informada.

O botão *Entrada* abre o formulário mostrado na Figura 5.2. Nele é solicitado ao usuário que informe o nível de $(G, \prec)$ que deseja visualizar. Como resultado é mostrado ao usuário uma matriz representando os níveis K e K+1 do grafo de derivação. Na matriz apresentada ao usuário, já estão visíveis as K cópias feitas para cada vértice do nível K+1.

Na Figura 5.2 , podemos observar na interseção entre os vértices do nível K e K+1 os custos de derivação D e I. Os custos de derivação D estão presentes nos vértices originais, região cercada por um retângulo vermelho, enquanto que os arcos com custo de derivação I são os que são formados por vértices cópias, presentes na região formada pelo retângulo azul.



	A,B	A,C	A,D	B,C	B,D	C,D
A,B,C	10	30		30		
A,B,D	15		40		40	
A,C,D		5	20			20
B,C,D				45	130	130
A,B,C	10	30		30		
A,B,C	10	30		30		
A,B,D	15		40		40	
A,B,D	15		40		40	
A,C,D		5	20			20
A,C,D		5	20			20
B,C,D				45	130	130
B,C,D				45	130	130

Figura 5.2: Tela de consulta aos níveis K e K+1 de $(G, \prec)$

O botão *Processar* inicia a execução do algoritmo propriamente dito. A Figura 5.3. mostra o pseudocódigo do programa principal que é executado. A descrição das principais funções já foi feita no capítulo anterior.

```
Para k = 0 até N-1
   Grafo_Bipartido(níveis k, k+1);
   Associação_de_Custo-Mínimo(níveis k, k+1);
   Para cada sub-grafo do grafo bipartido com associação de custo mímimo(níveis k, k+1)
      Para cada cuboide Ck+1 no nível k+1 do sub-grafo com custo mínimo
         Se existir Ck no nível k e o arco D(Ck+1,Ck) então
            estabelecer a ordem de Ck+1 de tal maneira que Ck seja um prefixo de Ck+1
         Detector_de_Conflitos(Ck+1);
         Grafo_de_Derivação_Quasi-Ótimo(níveis k,k+1)
   Grafo_de_Derivação_Quasi-Ótimo_Final(níveis k,k+1)
```

Figura 5.3: Trecho principal do algoritmo *Quasi*-Ótimo

O botão *Saída*, quando pressionado após a execução do algoritmo, permite a visualização do Grafo de Derivação *Quasi*-Ótimo gerado. A visualização do grafo é feita por nível. É solicitado ao usuário que informe um nível. A matriz mostrada mostra os vértices do nível K informado e os vértices do nível K+1. Os vértices do nível K+1 são usados para gerar os respectivos do nível K com custos D ou I. Nos níveis mostrados, K e K+1, é garantido que o nível K é gerado com custo mínimo ou o mais próximo do mínimo e que o nível K+1 casa com todas as consultas ordenadas informadas para este nível. A Figura 5.4, mostra a tela de consulta à saída gerada pelo algoritmo *Quasi*-Ótimo.

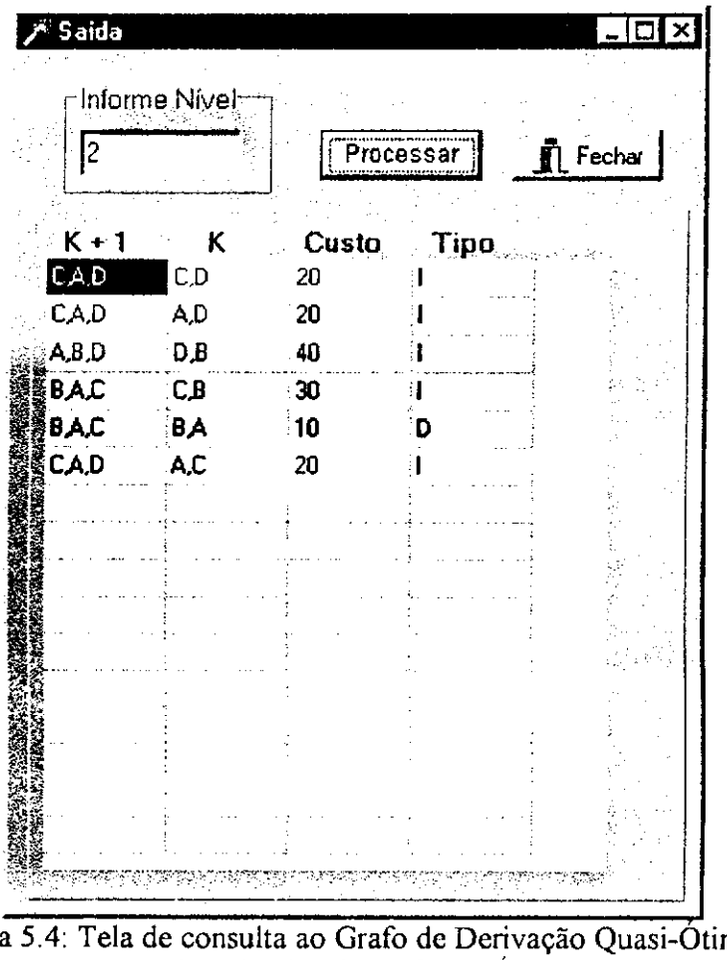


Figura 5.4: Tela de consulta ao Grafo de Derivação Quasi-Ótimo gerado pelo algoritmo *Quasi*-Ótimo

Em relação à complexidade o ponto crítico em termos de desempenho do algoritmo *Quasi*-Ótimo é a rotina *associação_de_custo_mínimo* que implementa o método Húngaro para o cálculo da associação de custo mínimo entre dois conjuntos de vértices de um grafo bipartido. A complexidade da rotina, para dois níveis consecutivos k e $k+1$ do grafo de derivação, é cúbica, $O(n^3)$, onde n é o número de vértices no nível $k+1$.

Capítulo 6

Conclusões e Perspectivas

O rápido crescimento do uso de sistemas de apoio à decisão nas empresas tem criado a necessidade de se integrar os mais variados dados espalhados pela empresa. Esta necessidade gerou uma das áreas de pesquisa em banco de dados com maior crescimento nos últimos anos, que é a integração de múltiplos bancos de dados em um único repositório de dados. Uma das abordagens para essa integração chamada de DW vem apresentando um grande crescimento nos últimos anos.

A integração de várias fontes de dados em um DW é parte de um processo maior chamado DWing que visa capacitar tomadores de decisão a tomar decisões rapidamente e corretamente acessando o DW.

Um DW é normalmente um banco de dados muito grande, mesmo se o nível de agregação das informações nele contidas for alto, pois pode guardar históricos de até 10 anos. Séries históricas são um requisito essencial de aplicações OLAP ("On-Line Analytical Processing"), desenvolvidas para auxiliar o processo de tomada de decisão. Os grandes DWs para aplicações OLAP podem ter de centenas de gigabytes a terabytes de tamanho. Consultas OLAP são em geral complexas, podendo acessar milhares de registros. Reduzir os custos de consultas OLAP é pois um importante objetivo de pesquisa.

A utilização de visões previamente materializadas em um DW é a maneira mais eficiente para responder a consultas OLAP. Porém um grande problema que surge é a determinação de que visões materializar, haja vista que a materialização de todas as possíveis visões geraria uma proliferação de visões difícil de se gerenciar.

É sabido que a escolha de que visões se materializar e em que formato é muito importante para a utilização satisfatória de um DW. Somente existindo as visões certas no DW é que as consultas OLAP terão um tempo de resposta satisfatório e permitirão a realização de interações plena entre usuário-sistema.

Nesta Dissertação, apresentamos um algoritmo para planejar a computação eficiente de múltiplos "group bys" de um "cube by". O operador "cube by" é intensivamente empregado em consultas do tipo OLAP a DWs. Os resultados de todos ou de alguns "group bys" de um "cube by" devem ser salvos no DW ou materializados (visões materializadas), visando a sua utilização posterior por consultas ao DW, reduzindo desta forma sensivelmente o custo de processamento das mesmas, comparativamente ao processamento em tempo real das visões virtuais, como é o caso das aplicações de bancos de dados convencionais.

Por computação eficiente de múltiplos "group bys", devemos entender o seguinte: por um lado, como as tabelas de um DW relacional são normalmente muito grandes, se cada "group by" fosse executado independentemente um do outro, então o tempo total de cálculo de todos eles (incluindo as operações de salvar seus resultados) seria igualmente muito grande, levando o DW a ficar inacessível a seus usuários por um periodo de tempo muito longo, enquanto todas as visões não tivessem sido materializadas. Para reduzir então este tempo, o algoritmo procura materializar simultaneamente o maior número possível de visões, pelo menor caminho, conseguindo com isto reduzir o tempo global de materialização de todas as visões.

Por outro lado, de nada adianta economizar o tempo de materialização das visões se não se levar em conta as consultas que poderiam utilizar as mesmas. Se não houvesse este cuidado, poderia ser que (1) uma visão viesse a ser muito pouco utilizada, ou sequer ser utilizada; e (2) uma visão e uma consulta que a utilizasse freqüentemente teriam ordens incompatíveis, levando à ordenação de arquivos temporários muito grandes, onerando desta forma excessivamente o custo de processamento da consulta.

O algoritmo *Quasi*-Ótimo leva então em conta estes dois custos, custo de criação de visões materializadas e custo de processamento de consultas a um DW. Ele tem como entrada um grafo chamado de *grafo de derivação* que indica que o resultado de um "group by" pode ser derivado *direta* ou *indiretamente* de um outro (isto é, caso não seja ou seja necessário reordenar o último para obter o primeiro), com os custos respectivos, além de uma lista de consultas frequentes ao DW que poderiam utilizar essas visões.

Como saída, ele gera um plano *ótimo* ou *quasi-ótimo* para a criação das visões, significando que neste plano o tempo de criação das visões é matematicamente o menor em relação a todos os outros tempos com o mesmo objetivo, ou próximo do menor tempo, e ainda ordenando essas visões para que, quando as consultas que podem utilizá-las forem submetidas ao DW, não haja necessidade de reordená-las.

Realizamos a implementação de um protótipo do algoritmo *Quasi*-Ótimo que serviu grandemente para validar as hipóteses iniciais deste trabalho e a realização de testes na geração de planos *ótimos* ou *quasi-ótimos*. A implementação do protótipo foi realizada considerando uma lista de consultas aleatórias e limitou-se à geração dos planos para criação das visões.

6.1 Trabalhos Futuros

Como um dos objetivos de nosso trabalho foi desenvolver um algoritmo que é uma solução eficiente para computação do cubo, uma proposta para trabalhos futuros é realizar uma análise detalhada de quão ótimo são os planos gerados pelo algoritmo.

Em termos de perspectivas de nosso trabalho, vislumbramos duas linhas de investigação. A primeira delas diz respeito à necessidade de realizar um grande número de testes com o algoritmo, baseados em casos próximos da realidade, para se determinar estatisticamente em que medida a solução apresentada pelo algoritmo é ótima. Dentre as soluções que não são ótimas, é preciso determinar estatisticamente o quanto elas *não são quasi*-ótimas (em outras palavras, as soluções cujos custos calculados são muito diferentes do custo ótimo). Dependendo destas observações estatísticas, poderemos pesquisar novas estratégias de otimização do algoritmo.

Finalmente, estamos trabalhando em um novo algoritmo que visa *atualizar* eficientemente o conjunto de visões materializadas de um "cube by". Para isto, ele trabalha também com grafos de derivação, mas somente para as alterações que devem ser feitas nas visões materializadas. Desta forma, dado um conjunto de dados de atualização de um DW, pode-se rapidamente derivar outros conjuntos de dados de atualização de visões materializadas do DW.

6.2 Considerações Finais

Realizamos um trabalho sobre otimização de consultas OLAP. Outros trabalhos nesta área têm usado a seguinte estratégia: cálculos necessários a consultas são pré-computados e armazenados no DW, com o propósito de propiciar custos aceitáveis às consultas que de outro modo teriam que efetuar esses cálculos em tempo real. A idéia básica de vários algoritmos com esta finalidade é a materialização simultânea de várias visões que têm critérios de agregação comuns. O problema com estes métodos é que eles podem impor uma ordem de armazenamento das visões capaz de onerar o custo das consultas que as utilizam, anulando, ao menos parcialmente, as vantagens da materialização.

Um grande projeto de pesquisa em DW está sendo desenvolvido na *Universidade de Stanford*, EUA. Sobre o operador "cube by" e visões materializadas, destacamos os trabalhos **[HARINARAYAN 96]** e **[GUPTA 96]**, que apresentam algoritmos para decidir que "group-bys" pré-calcular e indexar. Não existe nestes trabalhos, entretanto, a preocupação com a otimização de "group-bys" relacionados, ou dito de um outro modo, com a possibilidade de derivar "group bys" de outros "group bys".

Se não estivermos enganados, nosso trabalho é o primeiro que se preocupou em tratar de maneira integrada, desde o início, os dois problemas: materialização eficiente de visões e utilização eficiente de visões materializadas. As duas abordagens são às vezes conflitantes, ou seja, o que é o melhor para a materialização eficiente nem sempre é o melhor para o desempenho das consultas.

Acreditamos que nosso algoritmo será de grande valia, para futuros projetos de desenvolvimento de uma ferramenta OLAP para criação de agregados. Certamente nosso algoritmo integrando uma ferramenta OLAP que trabalhe em conjunto a um SGBD, poderá ser muito útil no desenvolvimento de DWs, proporcionando uma maior satisfação ao usuário final que esteja interagindo fazendo consultas OLAP.

Referências Bibliográficas

[BAZARAA 1990] Mokhtar S. Bazaraa, JohnJ. Jarvis, Hanif D. Sherali. Linear Programming and Network Flows – Second Edition, chapter 10, p. 499 –510, John Wiley & Sons, Inc, 1990

[CAMPOS 1997] L. M. Campos. *Data Warehouse*, Anais do XVII Congresso SBC – XVI Jornada de Atualização em Informática, Brasília 1997

[CHAUDHURI 1994] S. Chaudhuri, K. Shim. Including Group-By in Query Optimization. In Proceedings of the 20ª VLDB, p. 354-366, Chile, 1994

[COOD 1995] E. F. Codd, S. B. Codd, and C. T. Salley, Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate, E. F. Codd & Associetes, 1995

[COREY 1998] Michael J. Corey, Michael Abbey, Ian Abramson. Oracle8 Data Warehousing, Oracle Press, p. 45-64 1998

[DESHPANDE 1996] P.M.Deshpande, S.Agarwal, J.F.Naughton, and R.Ramakrishnan. Computation of multidimensional aggregates. *Technical Report Computer Science* TR1314, University of Wisconsin, Madison, Wisconsin, 1996

[EDELSTEIN 1997] Herb Edelstein, A Introduction to Data Warehousing. Planning and Designing the Data Warehouse, Pretice Hall, Inc p. 31-50

[FLAJOLET 1985] P. Flajolet, G.N. Martin. Probabilistic Counting Algorithms for Database Applications, Journal of Computer and System Sciences, 31(2): p. 182-209, 1995

[GILLETT 1976] Billy E. Gillett. Introduction to Operations Researsh. A Computer-Oriented Algorithmic Approach, chapter 3, p. 110-125, McGraw Hill, USA, 1976

[GRAEFE 1993] G.Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 25(2); p. 73-170, June 1993

[GRAY 1996] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube: A relacional aggregation operator generalizing group-by, cross-tab, and sub-totals. In Proceedings of the IEEE International Conference on Data Engineering, p. 152-159, 1996

[GUPTA 1996] H. Gupta et al. *Index Selection for OLAP*. Proc of the 13th Int. Conf. Ond Data Engineering, Birmingham, U.K., April 1997.

[HAAS 1995] P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of na Attribute, Proc. of the 21st Int. VLDB Conf., p. 311-322, 1995

[HARINARAYAN 1996] V. Harinarayan, A. Rajaraman, J. Ullman. Implementing Data Cubes Efficiently. In ACM SIGMOD Conference on Management of Data, June 1996

[INMON 1997] W. H. Inmon. Building the Data Warehouse, QED Pub. Group, 1997

[KIMBALL 1996] Ralph Kimball, The Data Warehouse Toolkit – Practical Techniques for Building Dimensional Data Warehouses. John Wily & Sons, Inc. 1996

[LEVIN 1997] Ellen J. Levin, Developing a Data Warehousing Strategy. Planning and Designing the Data Warehouse. p 53-91 Pretice Hall 97

[NASCIMENTO 1997] Rubens Nascimento Melo, Tutorial on Data Warehouse Technology, XII Simpósio Brasileiro de Banco de Dados, Fortaleza – Brasil

[O'NEILL 1995] P. O'Neill, G. Graefe. Multi-Table Joins Through Bitmapped Join Indexes. In SIGMOD Record, p. 8-11, September 1995.

[PAPADIMITRIOU 1982] C.H.Papadimitriou and K.Steiglits. Combinatorial Optimization: Algorithms and Complexity, chapter 11, p. 247-254. 1982

[PARSAYE 1996] Kamran Parsaye, The Four Spaces of Decision Support. DBMS Magazine, November 1996 p. 25

[SAMPAIO 1997] Marcus Costa Sampaio, Curso de Data Warehouse, COPIN/DSC, Campina Grande, 1997

[SARAWAGI 1996] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. *Technical Report RJ10026*, IBM Almaden Research Center, San Jose, CA, 1996

[SHUKLA 1996] A.Shukla, P.M.Deshpande, J.F.Naughton and K.Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In Proceedings of the 22ª VLDB Conference, India, 1996

[SOUZA 1998] Márcio F. Souza, Marcus Costa Sampaio. Materialização/Utilização Eficientes de Visões em Data Warehouses, XIII Simpósio Brasileiro de Banco de Dados, Maringá – Brasil p. 335-349

[SOUZA 1999] Márcio F. Souza, Marcus Costa Sampaio. *Efficient Materialization and Use of Views in Data Warehouses.* ACM SIGMOD Record, No. 01, Vol.28.

[TANLER 1998] Rick Tanler, Intranet Data Warehouse. Editora Infobook, 1998

[TREMBLAY 1975] J.P.Tremblay and R.Manahar. Discret Mathematical Structures with Applications to Computer Science. McGraw Hill Book , p. 130-149 New York, 1975

[WIEDERHOLD 1992] G. Wiederhold. Mediators in the architecture of future information systems. IEEE Computer, 25(3):38-49, March 1992.

[WITENBERG 1976] Juan Prawda Witenberg. Métodos y Modelos de Investigacion de Operaciones, Vol. 1 – Modelos Determinísticos, capítulo 3, p. 289-297, Editora Limusa, México, 1976

Apêndice I - Glossário de Termos

Estamos apresentando neste glossário o significado de vários termos e expressões utilizados ao longo da dissertação. Algumas destas expressões (tabela ordenada, consulta ordenada e consulta casada) não são de uso comum, foram criados por nós, enquanto que os demais termos e expressões são frequentemente encontrados na literatura sobre DW.

Consulta Casada: consulta *ordenada* a uma tabela, em que as ordens da consulta e da tabela coincidem.

Consulta Ordenada: consulta com a cláusula *order by*.

Cubo de Dados (Cubo): sinônimo de banco de dados (BD) *multidimensional*, em que o BD é visto como um *cubo de dados*, ou simplesmente *cubo*, de *n dimensões*: cada dimensão é um critério de agregação, e cada *célula* do cubo contém *medidas numéricas* ou *fatos* associados a um valor de cada uma das dimensões do cubo. Um cubo pode ser representado por um esquema relacional *em estrela*.

Cube by: operador associado a um conjunto de n dimensões de um cubo, calcula 2^n group bys para cada uma das combinações 1 a 1, 2 a 2, ..., n a n das n dimensões, incluindo *group by vazio*. Por exemplo, *select ... from tabela ... cube by d1, d2* resultará no cálculo de $2^2 = 4$ group bys, *group by d1, d2*, *group by d1*, *group by d2* e *group by vazio*.

Cuboid: cada uma das combinações das dimensões de um operador *cube by*. Para *cube by d1, d2*, os *cuboids* são *d1d2*, *d1*, *d2*, *vazio*.

Data Warehouse Relacional: cubo de dados em que o esquema é *em estrela*. Uma das dimensões é sempre o tempo, quase sempre por um longo período (5 anos, 10 anos, ...), razão pela qual um Data Warehouse é geralmente muito grande.

Esquema Relacional em Estrela ("Star Schema"): representação de um cubo de dados, composta de uma *tabela de fatos* (a estrela) e das *tabelas de dimensão* relacionadas com a tabela de fatos (os satélites da estrela). A Figura 1 é a ilustração de um esquema em estrela, com a tabela *tabela_fatos* e quatro tabelas de dimensão *d1*, *d2*, *d3* e *d4*.

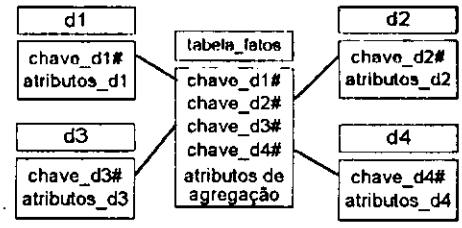


Figura A.1: Esquema Relacional em Estrela.

Tabela de Dimensão: tabela contendo a chave de uma dimensão, e os atributos para descrever a semântica de cada valor da chave.

Tabela de Fatos: tabela caracterizada por uma chave composta das chaves externas de cada dimensão, e por atributos geralmente *aditivos*, ou *atributos de agregação*.

Tabela Ordenada: uma tabela tem uma *ordem de armazenamento*, ou simplesmente uma *ordem*, se seus registros estão fisicamente agrupados ("clustered") por algum critério de ordenação envolvendo um ou vários de seus atributos. Se uma consulta ordenada é casada com a sua tabela, então seu custo de processamento é ótimo, e os otimizadores de consulta dos sistemas gerenciadores de bancos de dados procuram tirar proveito disto.

Visão Materializada: o nome que se dá a uma tabela ordenada correspondendo a um cuboid, e armazenada no DW. No restante do artigo e para simplificar, quando nos referirmos a um cuboid estaremos também fazendo menção à visão materializada a ele associada.

Apêndice II – PseudoCódigo do Algoritmo *Quasi*-Ótimo

Algoritmo *Quasi*-Ótimo para o cálculo de uma operação *cube by*

Entrada: Grafo de Derivação com custos D e I, e N dimensões
Lista de consultas ao DW
Opção: Cálculo total ou parcial da operação *cube by*

Saída: Grafo de Derivação *Quasi*-ótimo (completo ou parcial)

Para k = 0 **até** N-1
Grafo_Bipartido(níveis k, k+1);
Associação_de_Custo-Mínimo(níveis k, k+1);
Para cada sub-grafo do grafo bipartido com associação de custo mínimo*(níveis k, k+1)*
Para cada cuboid C_{k+1} no nível *k+1* do sub-grafo com custo mínimo
Se existir C_k no nível *k* e o arco $D(C_{k+1},C_k)$ **então**
estabelecer a ordem de C_{k+1} de tal maneira que C_k seja um prefixo de C_{k+1}
Detector_de_Conflitos(C_{k+1});
Grafo_de_Derivação_Quasi-Ótimo(níveis k,k+1)
Grafo_de_Derivação_*Quasi*-Ótimo_Final(níveis *k,k+1)* // O mais próximo da associação de custo mínimo

Se opção é cálculo parcial **então**
Grafo_de_Derivação_*Quasi*-Ótimo_Parcial*(Grafo_de_Derivação*_Quasi-*Ótimo*_Final*);*

Grafo_Bipartido(níveis k, k+1);
produzir *k* cópias de cada vértice no nível *k+1* do grafo de derivação;
Para cada vértice-cópia de um vértice-original
construir arcos do vértice-cópia para cada um dos vértices no nível *k*, tal que existem arcos ligando esses vértices ao vértice-original;
associar custos *D* aos arcos que partem dos vértices-originais e custos *I* aos arcos que partem dos vértices-cópias;

Associação_de_Custo-Mínimo(níveis k, k+1);
Entrada: grafo gerado por *Grafo_Bipartido(níveis k, k+1)*
Saída: Conjunto de sub-grafos do grafo bipartido, com custo mínimo, sem levar em conta a lista de consultas
Gerar o sub-grafo via o método Húngaro

Detector_de_Conflitos(C_{k+1});
Entrada: lista de consultas ao cuboid C_{k+1}
Case
Se existir uma ou mais consultas ao cuboid C_{k+1} e uma delas é casada com C_{k+1} **então**
// conflito do tipo 1
Solucionador_de_Conflitos(C_{k+1}, tipo_1);
Se existir uma ou mais consultas ao cuboid C_{k+1} e nenhuma delas é casada com C_{k+1} **então**
// conflito do tipo 2
Solucionador_de_Conflitos(C_{k+1}, tipo_2);

Solucionador_de_Conflitos(C_{k+1}, tipo_de_coflito);
Entrada: lista de consultas ao cuboid C_{k+1}
Case tipo_de_conflito
Se 1 **então**
escolher a consulta casada; //o custo do grafo de derivação *quasi*-ótimo para os níveis *(k+1) e k* é mínimo;
Se 2 **então**
escolher uma das consultas na lista de consultas
casar C_{k+1} com a consulta escolhida;
modificar convenientemente os arcos que partem de C_{k+1};
// o custo do grafo de derivação *quasi*-ótimo para os níveis *(k+1) e k* pode ser mínimo ou não;

Apêndice III – Instruções de Uso do Algoritmo *Quasi*-Ótimo

Alguns procedimentos precisam ser explicados para permitir a utilização do algoritmo. Neste Apêndice estaremos detalhando estas instruções de uso.

Configuração Inicial do Ambiente

- Versão do Delphi utilizada na codificação: *Delphi 3.0 c/s*
- Os componentes: math1.dcu, matrix.dcu e vector.dcu devem estar instalados no diretório C:\QUASI\COMPONENTES
- O arquivo *quasi.zip* que contém os fontes do protótipo deve ser descompactado no diretório *C:\QUASI* previamente criado. Os arquivos ARCOS.DB e VERTICES.DB devem ficar no sub-diretório *C:\QUASI\DATA*
- Criação de um *alias* no *DBExplorer* chamado Quasi que aponta para o diretório *C:\Quasi\Data (tipo PARADOX)*
- O arquivo de projeto do algoritmo chama-se *QUASI.DPR*, está localizado no diretório *C:\Quasi*

Instruções de Uso

1. A entrada do grafo de derivação é feita manualmente via *DBExplorer* nas tabelas ARCOS.DB e VERTICES.DB localizados no diretório C:\QUASI\DATA

A estrutura das tabelas é mostrada a seguir.

VERTICES

CÓDIGO	*NOME*	*NÍVEL*

Código: Seqüencial que deve ser informado

Nome: Nome do vértice (ex. A,B,C,D ou A,B,C ou A,B etc ...)

Obs.: Os atributos dos vértices devem estar separados por virgulas e apenas por elas. Isso é muito importante para permitir o funcionamento.

Nivel: O nível daquele vértice no grafo de entrada

ARCOS

VT_INICIAL	VT_FINAL	CUSTO_D	CUSTO_I

Vt_Inicial: Código de um vértice da tabela VERTICES

Vt_Final: Código de um vértice da tabela VERTICES

Obs.: VT_INICIAL e VT_FINAL formam um arco.

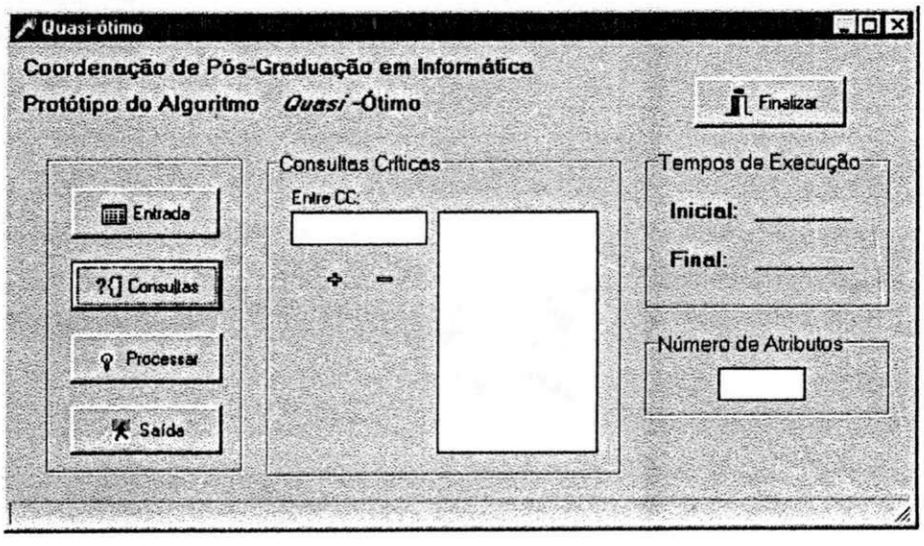
Os outros dois atributos (*CUSTO_D* e *CUSTO_I*) correspondem aos custos de derivação D e I.

Obs.: As tabelas ARCOS.DB e VERTICES.DB já estão previamente preenchidas com os vértices e arcos de um cube by com os atributos A,B,C,D.

Uma vez preenchido estas duas tabelas podemos executar o projeto *QUASI*.

Uma observação que deve ser feita é que apenas os arquivos *ARCOS.DB* e *VERTICES.DB* devem está no diretório C:\QUASI\DATA para permitir a execução correta do algoritmo. (O arquivo ARCOS2.DB não pode está no diretório C:\QUASI\DATA para execução inicial).

Vamos rapidamente apresentar os componetes da interface do protótipo do algoritmo. Maiores detalhes desta interface são encontrados na Capítulo 5 da Dissertação.



A caixa 'Número de Atributos' deve conter o número de atributos do grafo de derivação.

O botão 'Entrada' abre a tela que possibilita informar um nível do grafo de entrada. Será mostrado a matriz parcial de entrada para o Húngaro naquele nível.

O botão 'Consulta' abre uma janela na qual serão informadas as consultas críticas. Muita Atenção: As consultas críticas devem ser informadas em maiúsculo e com os atributos separados por vírgula. Isso é muito importante para que o algoritmo funcione perfeitamente.

Após informado o nível do grafo de entrada e as consultas críticas, selecione o botão 'Processar'.

Observações:

1. *Os arquivos ARCOS.DB e VERTICES.DB devem está OK.*
2. *O tempo de processamento dependerá da máquina e de quantos atributos tem o grafo, poderá variar de 25 segundos até 2 minutos. Logo a máquina cujo algoritmo executa é importante.*

Após o processamento, os tempos de execução (lado superior direito da tela) aparecerão com os valores do tempo inicial e final.

Um novo arquivo chamado ARCOS2.DB aparecerá no diretório C:\QUASI\DATA, este contém a saída do algoritmo.

O seu conteúdo poderá ser visualizado selecionando a opção 'Saída' do simulador. Na tela que será aberta para visualizar a saída basta informar o nível que se deseja visualizar do gráfico de saída.

O arquivo ARCOS2 contêm os atributos:

VT_INICIAL, VT_FINAL e CUSTO

que correspondem aos arcos VT_INICIAL e VT_FINAL e respectivo custo escolhido.

É importante dizer que na tabela VERTICES.DB, o nome dos vértices foram mudados após o processamento. Contendo casamento com as consultas críticas e prefixação para derivação.

Para uma próxima execução o arquivo ARCOS2.DB deve ser excluído e a tabela VERTICES.DB deve ter a ordem de seus vértices (cuboids) restabelecidas.