Índice

Capítulo 1 - Introdução a Sistemas de Bancos de Dados

Neste capítulo inicial faremos uma pequena introdução aos conceitos de bancos de dados. Há muitas definições para bancos de dados:

- Consiste em um sistema de manutenção de informações por computador que tem por objetivo mantê-las e disponibilizá-las aos seus usuários quando solicitadas (C.J. DATE).
- Coleção de dados, organizados, integrados (não duplicados), que constituem uma representação natural, sem imposição de restrições ou modificações para todas as aplicações relevantes.(PALMER).

Há autores que fazem distinção entre **Dado e Informação**. Os **dados** "*são um conjunto de fatos distintos e objetivos, relativos a eventos*" (DAVENPORT & PRUSAK, 1998) que constituem a base para a criação da informação, sendo obtidos, por exemplo, de registros ou transações. Eles não possuem significados inerentes, descrevem apenas parte do acontecido e não fornecem condições para qualquer julgamento, interpretação ou base para tomada de ação.

As informações são "*dados dotados de relevância e propósito*" (DAVENPORT & PRUSAK, 1998), constituindo-se em um fluxo de mensagens que tem por objetivo exercer algum impacto sobre o julgamento e o comportamento do seu receptor.

Tabela 1 - Características de dado e informação

Dado	Informação
• Simples observações sobre o estado do mundo • Facilmente estruturado • Facilmente obtido por máquinas • Freqüentemente quantificado • Facilmente transferível	• Dados dotados de relevância e propósito • Requer unidade de análise • Exige consenso em relação ao significado • Exige necessariamente a mediação humana

A Tabela 1e a Figura 1, ilustram as definições acima:

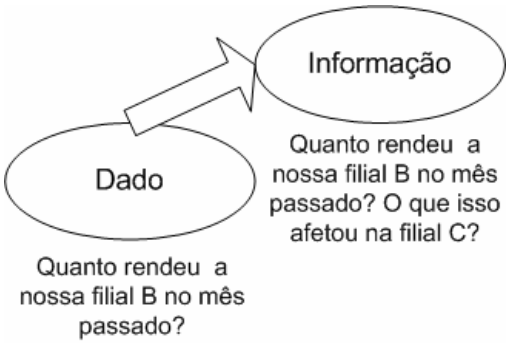


Figura 1 - Exemplo de dado e informação

1.1 SGBD – SISTEMAS GERENCIADORES DE BANCOS DE DADOS

Um SGBD é uma coleção de programas que permite a definição, construção e manipulação de Bases de Dados para as mais diversas finalidades.

De forma resumida, podemos dizer que:

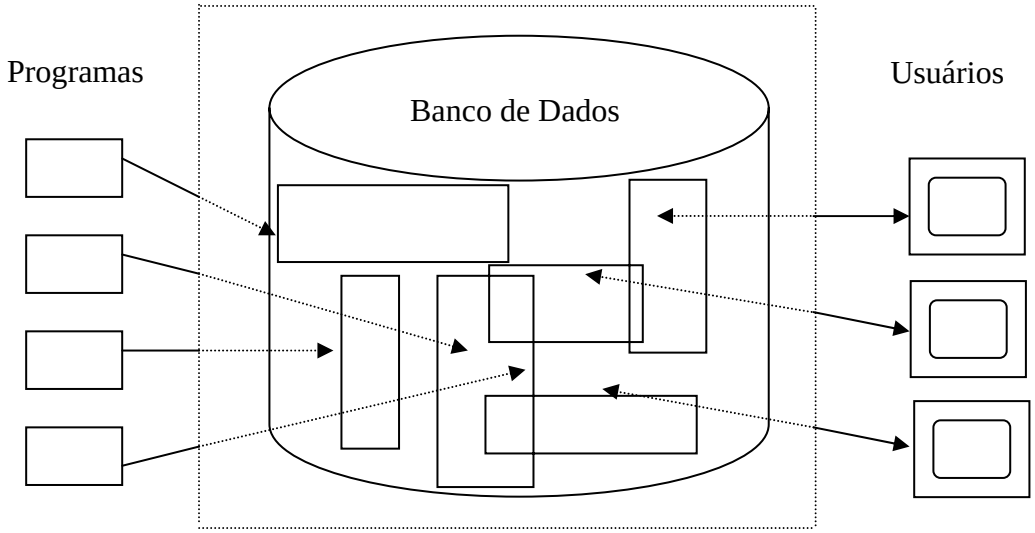
SGBD = Coleção de dados inter-relacionados + Camada de *software* para acessá-los

Alguns dos principais objetivos de um SGBD são prover um ambiente que seja adequado e eficiente para recuperar e armazenar dados bem como prover os usuários com uma visão abstrata destes.

Não devemos confundir um SGBD, Sistema Gerenciador de Banco de Dados, com a parte do Sistema Operacional chamada de Gerenciador de Arquivos. Veja algumas das desvantagens de um Gerenciador de Arquivo frente a um SGBD.

1. **Redundância e inconsistência de dados**. Arquivos com formatos diferentes, diferentes linguagens de programação, elementos de informação duplicados em diversos arquivos.
2. **Dificuldade no acesso aos dados**. Dados recuperados de forma complexa, inconveniente e ineficiente.
3. **Isolamento de dados**. Dados intimamente relacionados podem estar mantidos separadamente (Ex.: Planilhas Excel).
4. **Anomalias de acesso concorrente**. Dados acessados por diferentes programas aplicativos – é uma tarefa complicada a supervisão e controle desta concorrência.
5. **Problemas de segurança**. Difícil definição de **níveis** de visibilidade para usuários.
6. **Problemas de integridade**. Restrição de integridade nos valores dos atributos é de difícil manutenção.

Abaixo temos a representação básica e simplificada da estrutura de funcionamento de um sistema de banco de dados.



1.2 Aplicação de Exemplo

Nosso estudo estará embasado em uma aplicação bancária. Desenvolvemos um banco de dados relacional para expressar os correntistas de um banco. Os esquemas abaixo detalham o banco de dados (projetos lógico e físico). Os dados de demonstração são sintéticos.

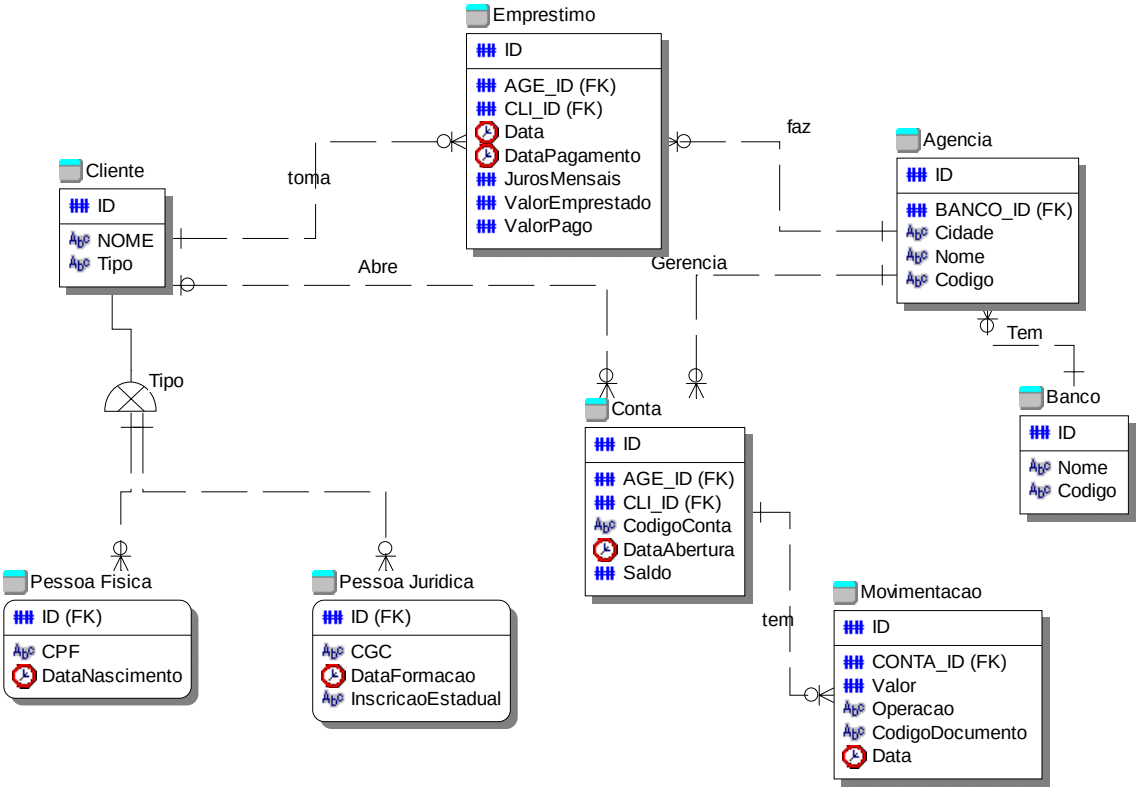


Figura 2 - Esquema Lógico do Banco de Dados de Aplicação Bancária

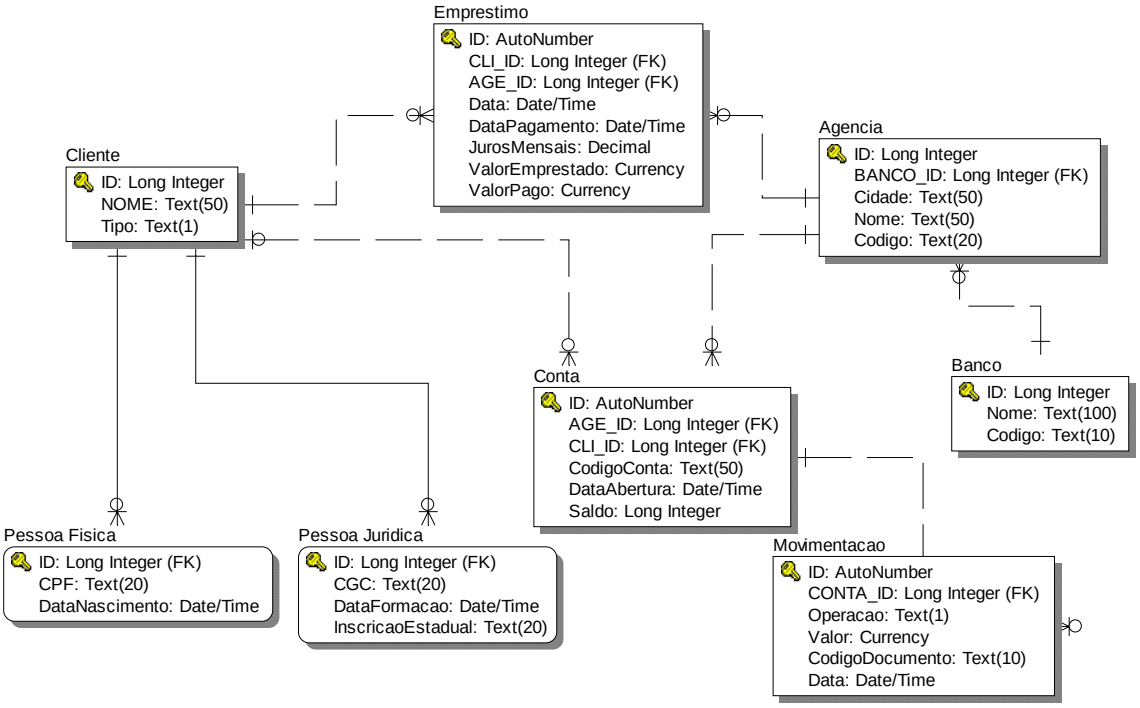


Figura 3 - Esquema Físico do Banco de Dados de Aplicação Bancária

Exemplos de dados da Tabela BANCO

ID	Nome	Codigo
1	Banco do Brasil	001
2	Banco Real	356
3	Banco Bradesco	789
4	Banco Itau	129

Exemplos de dados da Tabela AGENCIA

ID	BANCO_ID	Cidade	Nome	Codigo
1	1	Rio de Janeiro	Ag Centro	4891-1
2	1	Rio de Janeiro	Ag Tijuca	3872-1
3	1	Sao Paulo	Ag Tiete	3098-2
4	2	Salvador	Ag Ondina	8987-2
5	2	Sao Paulo	Ag Tiete 2	2121-0
6	2	Sao Paulo	Ag Pinhes	213-A
7	3	Rio de Janeiro	Ag Borel	2874-2
8	3	Brasilia	Ag Ministerios	2129-0
9	3	Salvador	Ag Barra	9827-2
10	4	Sao Paulo	Ag Barra Funda	2233-2
11	4	Sao Paulo	Ag Ministerios	0983-0
12	4	Sao Paulo	Ag Barra	9354-2

Exemplos de dados da Tabela CLIENTE

ID	NOME	Tipo
1	Leonardo	F
2	Lucro Certo SA	J
3	Maria	F
4	Joao Pedro	F
5	Mudancas Joe	J
6	Armando Jose	F

Exemplos de dados da Tabela PESSOA FISICA

ID	CPF	DataNascimento
1	07898281790	6/8/1977
3	12345678912	10/1/1980
4	98765432109	26/2/1969
6	56941236420	17/10/1940

Exemplos de dados da Tabela PESSOA JURIDICA

ID	CGC	DataFormacao	InscricaoEstadual
2	87628871/0001	1/12/1957	29389
5	98767266/0002	6/7/1997	56416

Exemplos de dados da Tabela CONTA

ID	AGE_ID	CLI_ID	CodigoConta	DataAbertura	Saldo
1	1	4	3652-8	13/1/2002	90821
2	2	4	3323-4	23/12/1998	12000
3	3	1	3215-9	6/1/1997	1000
4	4	3	9873-8	3/3/2000	50000
5	5	5	4982-1	18/8/1999	33312
6	6	6	3177-2	14/7/2003	800
7	7	2	3212-1	9/6/2002	1300
8	8	2	0098-0	20/5/2001	47893
9	9	1	3277-1	3/10/2002	78933

Exemplos de dados da Tabela MOVIMENTACAO

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
1	1	R$ 500,00	C	x998	10/1/2000
2	1	R$ 1.500,00	D	x997	30/1/2000
3	1	R$ 1.500,00	C	x998	20/5/2000
4	1	R$ 4.500,00	D	x997	10/1/2001
5	2	R$ 1.800,00	C	3872	10/1/2000
6	2	R$ 1.900,00	C	2908	30/1/2000
7	2	R$ 100,00	C	2398	20/5/2000
8	2	R$ 50,00	C	9823	10/1/2001
9	9	R$ 100,00	C	2111	10/1/1998
10	8	R$ 200,00	D	3444	11/5/1999
11	7	R$ 350,00	C	5783	14/7/2002
12	6	R$ 450,00	D	9234	15/11/2000
13	5	R$ 550,00	C	9324	15/10/1997

Exemplo da Tabela EMPRESTIMO

ID	AGE_ID	CLI_ID	Data	DataPagamento	JurosMensais	ValorEmprestado	ValorPago
1	5	1	12/12/2003		3	R$ 5.000,00	R$ 0,00
2	11	5	4/11/2002		1	R$ 500.000,00	R$ 0,00
3	1	6	22/8/2003		5	R$ 15.000,00	R$ 10.000,00
4	5	3	12/12/2002	12/5/2003	3	R$ 50.000,00	R$ 50.000,00
5	12	3	14/12/2002	12/5/2003	1	R$ 20.000,00	R$ 20.000,00
6	3	1	21/8/2002	21/8/2003	5	R$ 44.000,00	R$ 44.000,00
7	6	2	7/5/2002	7/5/2003	3	R$ 89.000,00	R$ 89.000,00
8	6	2	4/4/2002	30/4/2002	1	R$ 3.000,00	R$ 3.030,00
9	10	1	22/10/2002	31/12/2002	5	R$ 15.000,00	R$ 15.000,00

Capítulo 2 - Linguagens de Consulta

Uma linguagem de consulta é a linguagem através da qual usuários e programas solicitam informações de um banco de dados. De uma maneira geral, elas apresentam **nível mais alto** quando comparadas a linguagens de programação (Fortran, C, Pascal etc).

Os operadores de uma linguagem de consulta são capazes de manipular conjuntos inteiros de objetos simples, sem a restrição de um registro de cada vez.

Neste contexto, linguagens de consultas podem ser:

1. **Procedurais** – usuários informam ao sistema a sequência de operações que deve ser enviada ao banco de dados, tal seqüência irá computar o resultado desejado. Exemplos: Álgebra Relacional e PLSQL (linguagem procedural da Oracle que estende o SQL).
2. **Não-Procedurais ou Declarativas** – usuários especificam a informação que desejam sem se preocupar em informar a maneira como ela será recuperada. Exemplos: SQL e QBE.

Uma linguagem de consulta completa ainda contempla facilidades para inserção, remoção e atualização de informações, bem como criação e remoção de objetos em um banco de dados.

2.1 Álgebra Relacional

A álgebra relacional é uma linguagem de consulta procedural muito importante porque serve como base para a implementação eficiente de linguagens declarativas (mais simples). É a base matemática dos bancos de dados relacionais, que são os bancos de dados convencionais mais utilizados nos dias atuais.

Como estamos no mundo relacional, os objetos do banco de dados são tabelas (relações) que apresentam relacionamentos entre si. Assim, a linguagem consiste em um **conjunto de operações** que tomam uma ou mais relações como entrada e produzem uma nova relação como resultado (mundo fechado).

Abaixo temos algumas propriedades das relações para melhor entendimento de álgebra relacional:

- As tuplas de uma relação não têm uma ordem definida. Formam um conjunto.
- Uma tupla é formada por um conjunto de pares (<atributo>, <valor>) e também não têm uma ordem explícita.
- Todos os valores são atômicos, não há atributos multivalorados (chamamos esta regra de primeira forma normal).
- Não há tuplas duplicadas. Não pode haver duas linhas com os mesmos valores em uma tabela.
- Cada chave candidata deve ser única para cada tupla.
- Nenhuma chave primária pode ser nula.
- Uma tupla em uma relação que referencia outra relação deve referenciar uma tupla existente naquela relação (chamamos esta regra de integridade referencial).

As operações da álgebra relacional podem ser divididas em dois grupos:

1. Operações Fundamentais
    a. SELEÇÃO (unária)
    b. PROJEÇÃO (unária)
    c. PRODUTO CARTESIANO (binária)
    d. RENOMEAÇÃO (unária)
    e. UNIÃO (binária)
    f. DIFERENÇA DE CONJUNTOS (binária)
2. Operações Derivadas das Fundamentais (e muito importantes)
    a. INTERSEÇÃO DE CONJUNTOS
    b. JUNÇÃO
    c. DIVISÃO
    d. *ASSIGNMENT*

A partir de agora, examinaremos cada operação da álgebra relacional.

2.1.1 Seleção

A operação de seleção constrói uma nova tabela usando um subconjunto horizontal de uma tabela existente, isto é, todas as linhas de uma tabela que satisfaçam a determinada condição.

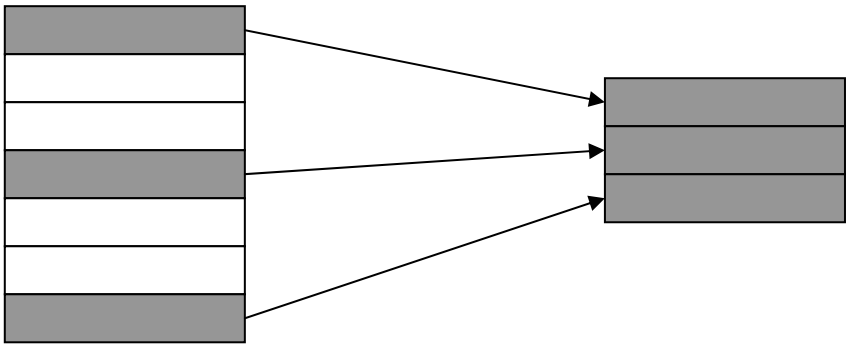
A condição é expressa como uma combinação lógica (booleana) de termos, sendo cada termo uma comparação singela que pode se tornar verdadeira ou falsa para uma determinada linha, inspecionando-se aquela linha isoladamente.

Se um termo envolver uma comparação entre valores de dois atributos dentro da mesma linha, então esses atributos têm que estar definidos no mesmo domínio. Por exemplo, não é possível comparar diretamente um número com uma string.

A operação de seleção

- É unária;
- É usada para selecionar um subconjunto das tuplas em uma relação, subconjunto este de tuplas que satisfazem a um **predicado** especificado (condição de seleção);
- É representada pela letra grega sigma (σ);
- Possui o predicado subscrito em σ, sendo o predicado uma combinação booleana de termos;
- Possui a relação argumento entre parênteses seguindo o σ;
- Em geral é denotada da seguinte forma: σ<condição de seleção>(<nome da relação>).

Exemplo:



A	B	C
1	Y	a
3	X	c
2	Y	b

$\sigma_{B=y \wedge A>1}(R)$

A	B	C
2	Y	b

As operações lógicas permitidas são =, != (<>), <, <=, >, >=.

Os conectores lógicos são "^" (operador "E" – *and*) e "v" (operador "OU" – "*or*").

Outros exemplos:

Selecione as agências do "Rio de Janeiro".

$\sigma_{cidade='Rio\ de\ Janeiro'}(AGENCIA)$

ID	BANCO_ID	Cidade	Nome	Codigo
1	1	Rio de Janeiro	Ag Centro	4891-1
2	1	Rio de Janeiro	Ag Tijuca	3872-1
7	3	Rio de Janeiro	Ag Borel	2874-2

Encontrar as movimentações de crédito maiores do que 1000.

σoperacao='C' ^ valor>1000 (MOVIMENTACAO)

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
3	1	R$ 1.500,00	C	x998	20/5/2000
5	2	R$ 1.800,00	C	3872	10/1/2000
6	2	R$ 1.900,00	C	2908	30/1/2000

2.1.2 Projeção

A operação de projeção forma uma nova tabela usando um subconjunto vertical de algumas colunas de uma tabela, subconjunto obtido pela seleção de atributos especificados, extraindo colunas específicas e removendo qualquer linha duplicata no conjunto de colunas extraídas.

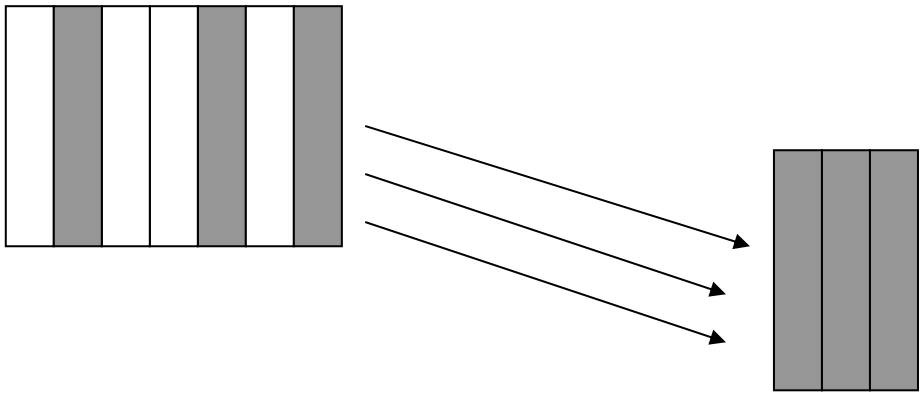
A projeção nos fornece a maneira de permutar (reordenar) os atributos de uma dada relação. Nenhum atributo pode ser especificado mais de uma vez em uma operação de projeção. A omissão da lista de nomes de atributos é equivalente à especificação de uma lista contendo todos os nomes de atributos da relação dada, na sua ordem correta, da esquerda para a direita. Em outras palavras, tal projeção é idêntica à relação dada.

Observação: qualquer operação da álgebra relacional é um processo de construção de tabela. Reconhecendo este fato e ainda que, matematicamente falando, uma relação (tabela) é um conjunto – um conjunto de linhas – e por definição conjuntos não podem conter elementos em duplicata, justifica-se o fato de que em uma projeção as linhas em duplicata sejam extraídas do resultado final.

A operação de projeção

- É unária;
- É usada para retornar a própria relação argumento, mas com certos atributos (colunas) deixados de fora;
- É representada pela letra grega pi (Π);
- Possui os atributos que se deseja que apareçam no resultado como subscritos de Π;
- Possui a relação argumento entre parênteses seguindo o Π;
- Em geral é denotada da seguinte forma: Π<lista de atributos>(<nome da relação>).

Exemplo:



A	B	C
1	y	a
3	x	c
2	y	b

$\Pi_{B,C}(R)$

$\Pi_{B}(R)$

B	C
y	A
x	C
y	B

B
y
x

Repare a remoção de duplicatas!

Os nomes dos atributos devem ser separados por vírgula (“,”).

É possível agrupar operadores:

A	B	C
1	y	a
3	x	c
2	y	b

$\Pi_{B}(\sigma_{B=y \wedge A>1}(R))$

B
y

Outros exemplos:

Mostre o CPF das pessoas fisicas.

Π cpf (PESSOA FISICA)

cpf
07898281790
12345678912
56941236420
98765432109

Mostre os códigos de bancos existentes em “Salvador”.

Devemos fazer uma seleção de todas as agências de Salvador, em seguida projetar o código dos bancos.

Π banco_id (σcidade=“Salvador”(AGENCIA))

banco_id
2
3

Observe que neste caso, se houver mais de uma agência na cidade, o código do banco aparecerá somente uma vez no resultado.

2.1.3 Produto Cartesiano

Esta operação combina atributos (colunas) a partir de diversas relações. Trata-se de uma operação binária muito importante e custosa. Esta operação nos mostra todos os atributos das relações envolvidas.

O resultado do produto cartesiano é uma nova relação em que cada tupla é uma combinação de tuplas das relações envolvidas.

Nesta operação, a tendência é que o tamanho do resultado seja bem grande (operação custosa). Por exemplo, se uma tabela “A” tem “y” tuplas e uma tabela “B” tem “z” tuplas, então o resultado (A X B) terá “y*z” tuplas.

A operação de produto cartesiano:

- É binária;
- É representada pela letra xis (X);
- Tem uma representação natural para os nomes dos atributos quando não existe ambigüidade (vide exemplo abaixo);
- Em geral é denotada da seguinte forma: R1 X R2.

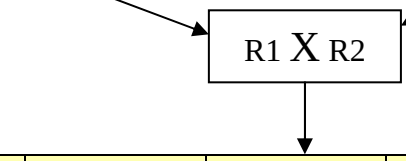
Exemplo:

R1

A	B	C
1	y	a
3	x	c
2	y	b

R2

D	C
1	Vinho
2	Champagne



A	B	R1.C	D	R2.C
1	y	a	1	Vinho
1	y	a	2	Champagne
3	x	c	1	Vinho
3	x	c	2	Champagne
2	y	b	1	Vinho
2	y	b	2	Champagne

Como implementar uma correlação entre os atributos A e D em R1 e R2 especificamente? (JUNCAO)

Outros Exemplos:

CLIENTE = (codigo, nome)

FIADO = (codigo, valor)

Se quisermos saber os nomes dos clientes e seus fiados, deveríamos fazer:

Π codigo,nome,valor (σCLIENTE.codigo=FIADO.codigo (Cliente X Fiado))

Supondo as tabelas CLIENTE e FIADO abaixo, podemos representar das etapas desta consulta.

CLIENTE

codigo	nome
1	Chico
2	Ana

FIADO

codigo	valor
1	5,00
1	10,00
2	4,00
2	8,00

CLIENTE X FIADO

CLIENTE.codigo	nome	FIADO.codigo	valor
1	Chico	1	5,00
1	Chico	1	10,00
1	Chico	2	4,00
1	Chico	2	8,00
2	Ana	1	5,00
2	Ana	1	10,00
2	Ana	2	4,00
2	Ana	2	8,00

σCLIENTE.codigo=FIADO.codigo (CLIENTE X FIADO)

CLIENTE.codigo	nome	FIADO.codigo	valor
1	Chico	1	5,00
1	Chico	1	10,00
2	Ana	2	4,00
2	Ana	2	8,00

Encontrar uma combinação de serviços, indicando pessoas físicas para pessoas jurídicas venderem produtos ou serviços.

PESSOA FISICA X PESSOA JURIDICA

pessoafisica.ID	CPF	DataNascimento	pessoajuridica.ID	CGC	DataFormacao	InscricaoEstadual
1	07898281790	6/8/1977	2	87628871/0001	1/12/1957	29389
1	07898281790	6/8/1977	5	98767266/0002	6/7/1997	56416
3	12345678912	10/1/1980	2	87628871/0001	1/12/1957	29389
3	12345678912	10/1/1980	5	98767266/0002	6/7/1997	56416
4	98765432109	26/2/1969	2	87628871/0001	1/12/1957	29389
4	98765432109	26/2/1969	5	98767266/0002	6/7/1997	56416
6	56941236420	17/10/1940	2	87628871/0001	1/12/1957	29389
6	56941236420	17/10/1940	5	98767266/0002	6/7/1997	56416

2.1.4 Renomeação

A operação de renomear uma tabela é usada sempre que uma relação aparece mais de uma vez em uma consulta. É representada pela letra grega ρ (ro) . A forma geral é:

ρ <novo nome> (TABELA)

Exemplo:

Encontrar as agências que são da mesma cidade que a agência ID número 6.

Para obtermos a cidade da agência 6 fazemos:

σ id=6 (AGENCIA)

ID	BANCO_ID	Cidade	Nome	Codigo
6	2	Sao Paulo	Ag Pinhes	213-A

Entretanto, para encontrar outras agências com esta mesma cidade, é necessária uma nova referência à tabela agência. Se novamente inserirmos a palavra "AGENCIA" na expressão, geraremos uma ambigüidade, então é necessário utilizar renomeação.

ρ AGENCIA2 (AGENCIA)

Finalmente obtemos:

σ agencia.cidade = agencia2.cidade (σ id=6 (AGENCIA) X ρ AGENCIA2 (AGENCIA))

ID	BANCO_ID	Cidade	Nome	Codigo
12	4	Sao Paulo	Ag Barra	9354-2
11	4	Sao Paulo	Ag Ministerios	0983-0
10	4	Sao Paulo	Ag Barra Funda	2233-2
6	2	Sao Paulo	Ag Pinhes	213-A
5	2	Sao Paulo	Ag Tiete 2	2121-0
3	1	Sao Paulo	Ag Tiete	3098-2

Exercício: Para obtermos o resultado acima, falta uma projeção na expressão acima? Onde e como a colocar?

2.1.5 União

A operação binária união é representada, como na teoria dos conjuntos, pelo símbolo **U**. A união de duas relações R e S (R U S) é o conjunto de todas as tuplas pertencentes a R, ou a S, ou a ambas.

A operação de união é a primeira que requer que as duas relações operando sejem "compatíveis de união", isto é, elas têm que ser do mesmo grau (mesmo número de atributos), digamos n, e os atributos das duas relações têm que ser retirados do mesmo domínio (não precisam ter o mesmo nome).

Lembre-se que, como na teoria dos conjuntos, duplicatas são eliminadas em uma operação de união.

Exemplo:

Encontrar todos os clientes, com informações detalhadas.

É necessário obtermos dados de pessoas físicas e de pessoas jurídicas, assim:

$\Pi_{id,cpf,datanascimento}(PESSOA\ FISICA)\ U\ \Pi_{id,cgc,dataformacao}(PESSOA\ JURIDICA)$

id	cpf	datanascimento
1	07898281790	6/8/1977
2	87628871/0001	1/12/1957
3	12345678912	10/1/1980
4	98765432109	26/2/1969
5	98767266/0002	6/7/1997
6	56941236420	17/10/1940

O nome da coluna veio como “cpf” por que?

2.1.6 Diferença de Conjuntos

A diferença entre duas relações R e S (nesta ordem), (R – S), é o conjunto de todas as tuplas t pertencentes a R, mas não a S.

Exemplo:

Encontrar os códigos de bancos que não têm agências no ‘RIO DE JANEIRO’.

$\Pi_{ID}(BANCO) - \Pi_{BANCO\_ID}(\sigma_{cidade='RIO\ DE\ JANEIRO'}(AGENCIA))$
(todos os bancos) – (bancos com agências no rio de janeiro)

Na primeira parte projetamos todos os códigos dos bancos e na segunda parte, projetamos os códigos de bancos das agências existentes no Rio de Janeiro.

Se um banco tem agência no Rio de Janeiro, então seu código aparece na primeira parte E na segunda, fazendo com que ele NÃO apareça no resultado final.

Se um banco não tem agência no Rio, então seu código só aparece na primeira parte, fazendo com que entre no resultado final, exemplificado abaixo.

id
2
4

Observação importante: na diferença de conjuntos, bem como na união, as relações devem ser “compatíveis de união”.

2.1.7 Interseção de Conjuntos

A operação é representada pelo símbolo ∩. A interseção de duas relações R e S (R ∩ S) é o conjunto de todas as tuplas pertencentes simultaneamente a R e a S. A interseção poder ser expressa através de diferença de conjuntos como:

$R \cap S = R - (R - S)$

Exemplos:

Encontre o código dos clientes que tem empréstimo e conta na agência de ID 3.

$\Pi_{CLI\_ID}(\sigma_{age\_id=3}(EMPRESTIMO)) \cap \Pi_{CLI\_ID}(\sigma_{age\_id=3}(CONTA))$

(clientes que têm empréstimo) ∩ (clientes que têm conta)

cli_id
1

2.1.8 Junção

É uma das operações mais importantes da álgebra relacional. Simplifica a operação de produto cartesiano através do operador de seleção, que fica implícito.

A operação de junção forma, a partir de duas tabelas que possuem, cada uma, uma coluna definida em um domínio comum, uma tabela nova e mais ampla, na qual cada linha é formada pela concatenação de duas linhas, uma de cada tabela original, sendo que as duas linhas têm o mesmo valor naquelas duas colunas.

Se uma linha de uma das tabelas originais não tiver correspondente na outra, ela simplesmente não participa do resultado.

Na verdade esta definição corresponde apenas a uma das várias uniões possíveis – ou seja, a junção na qual a condição de junção está baseada na igualdade entre valores na coluna comum – e é por isso conhecida como “equi-join”. É possível também definir, por exemplo, uma junção “maior do que”, uma junção “não-igual”, etc.

A operação de junção é representada da seguinte forma:

$R1 \rhd\lhd_{(condição)} R2$

É equivalente a:

$\sigma_{condição}(R1 \times R2)$

Exemplos:

Encontre o nome dos clientes que têm empréstimos, o valor do empréstimo e o ID da agência que emprestou o dinheiro.

$\Pi_{Nome,ValorEmprestado,AGE\_ID}(CLIENTE \rhd\lhd_{(id=cli\_id)} EMPRESTIMO)$

nome	valoremprestado	age_id
Leonardo	R$ 5.000,00	5
Leonardo	R$ 44.000,00	3
Leonardo	R$ 15.000,00	10
Lucro Certo SA	R$ 89.000,00	6

Lucro Certo SA	R$ 3.000,00	6
Maria	R$ 50.000,00	5
Maria	R$ 20.000,00	12
Mudancas Joe	R$ 500.000,00	11
Armando Jose	R$ 15.000,00	1

O identificador interno da Agência não tem valor para um usuário (é um campo de controle). Vamos estender a junção para contemplar mais uma tabela, pois não queremos mais o ID da Agência, mas o seu código e cidade.

$$\Pi_{Nome,ValorEmprestado,Código,\ Cidade}\ (\ CLIENTE \rhd\lhd_{(id=cli\_id)} EMPRESTIMO \rhd\lhd_{(age\_id=id)} AGENCIA\ )$$

nome	valoremprestado	codigo	cidade
Leonardo	R$ 5.000,00	2121-0	Sao Paulo
Mudancas Joe	R$ 500.000,00	0983-0	Sao Paulo
Armando Jose	R$ 15.000,00	4891-1	Rio de Janeiro
Maria	R$ 50.000,00	2121-0	Sao Paulo
Maria	R$ 20.000,00	9354-2	Sao Paulo
Leonardo	R$ 44.000,00	3098-2	Sao Paulo
Lucro Certo SA	R$ 89.000,00	213-A	Sao Paulo
Lucro Certo SA	R$ 3.000,00	213-A	Sao Paulo
Leonardo	R$ 15.000,00	2233-2	Sao Paulo

Restrições podem ser colocadas em qualquer relação envolvida na junção. Por exemplo, se desejarmos ver os clientes que pediram empréstimos no “Rio de Janeiro”, temos a seguinte consulta.

$$\Pi_{Nome,ValorEmprestado,Código,\ Cidade}\ (\ \sigma_{cidade=`RiodeJaneiro'}(CLIENTE \rhd\lhd_{(id=cli\_id)} EMPRESTIMO \rhd\lhd_{(age\_id=id)} AGENCIA)\ )$$

nome	valoremprestado	codigo	cidade
Armando Jose	R$ 15.000,00	4891-1	Rio de Janeiro

Repare que a junção é ASSOCIATIVA!

2.1.9 Divisão

A operação divisão, representada por ÷, serve para consultas que incluem frases com “para todos”.

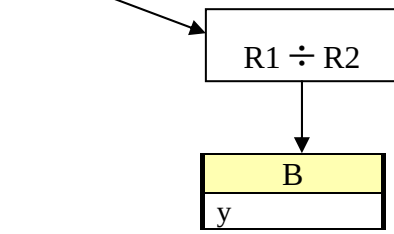
Exemplo:

R1 (numerador)

B	C
y	a
y	c
y	b
x	a

R2 (denominador)

C
a
b
c



B
y

Suponha que desejamos encontrar os clientes que têm conta em TODAS as agências da cidade de ‘Brasilia’.

Em primeiro lugar, obtemos todas as agências de ‘Brasilia’ (denominador):

$AUX1 \leftarrow \prod_{ID} (\sigma_{cidade='Brasilia'} (AGENCIA))$

id
8

Em segundo lugar, temos que obter os clientes e as agências nas quais eles têm conta (numerador):

$AUX2 \leftarrow \prod_{Nome,AGE\_ID} (CLIENTE \rhd\lhd_{(id=cli\_id)} CONTA)$

nome	age_id
Leonardo	3
Leonardo	9
Lucro Certo SA	7
Lucro Certo SA	8
Maria	4
Joao Pedro	1
Joao Pedro	2
Mudancas Joe	5
Armando Jose	6

Agora precisamos encontrar os clientes que apareçam em AUX2 com cada ID de agência em AUX1. Escrevemos esta consulta da seguinte forma:

$AUX2 \div AUX1$

NOME
Lucro Certo SA

2.1.10 ASSIGNMENT

É a operação utilizada no exemplo da divisão acima, usada para reduzir a complexidade das expressões. Utiliza o operador ←.

2.2 QBE e QUEL

2.3 SQL (Structured Query Language)

A linguagem SQL é uma linguagem de consulta declarativa. Esta linguagem se tornou **padrão** nos produtos relacionais do mercado, daí a sua importância.

O Departamento de Pesquisas da IBM desenvolveu a SQL como forma de interface para o sistema de BD relacional denominado SYSTEM R, início dos anos 70. Em 1986 o American National Standard Institute (ANSI), publicou um padrão SQL.

A linguagem SQL apresenta diversas partes:

1. Comandos DDL (*Data Definition Language*) → provêem comandos para criação e manutenção de relações no SGBD. Exemplos: criar esquemas, criar tabelas, criar índices, modificar tabelas, remover índices etc.
2. Comandos DML (*Data Manipulation Language*) → provêem uma forma declarativa de recuperar informações no banco de dados. A linguagem SQL é baseada na álgebra e no cálculo relacional. A SQL ainda traz comandos DML para inserção, atualização e remoção de tuplas.
3. Comandos Embutidos → são usados dentro de linguagens de programação como C, Pascal, Cobol etc.
4. Definições de Visões → comandos para criação de visões.
5. Segurança → comandos para definir permissões nos objetos do banco de dados: relações, atributos, visões, procedimentos etc.
6. Integridade → comandos que implementam as restrições de integridade dos dados.
7. Controle de Transação → comandos que especificam início e fim de transações.

2.3.1 Estrutura Básica da DML do SQL

Uma expressão básica em SQL apresenta três cláusulas básicas: SELECT FROM WHERE.

- SELECT → lista atributos que são desejados no resultado (análogo à projeção em álgebra relacional).
- FROM → lista as relações que serão usadas na consulta. As relações listadas nesta cláusula formam um produto cartesiano implícito no resultado.
- WHERE → corresponde ao predicado lógico que cada tupla deve satisfazer para entrar no resultado (análogo à seleção em álgebra relacional).

Uma consulta típica em SQL tem a forma:

```
SELECT DISTINCT A1, A2, A3,..., An
FROM R1, R2, R3,..., Rm
WHERE P
```

Cada A (A1, A2... An) corresponde a atributos existentes nas relações R (R1, R2... Rm). P é um predicado lógico, por exemplo: “A2 > 40”.

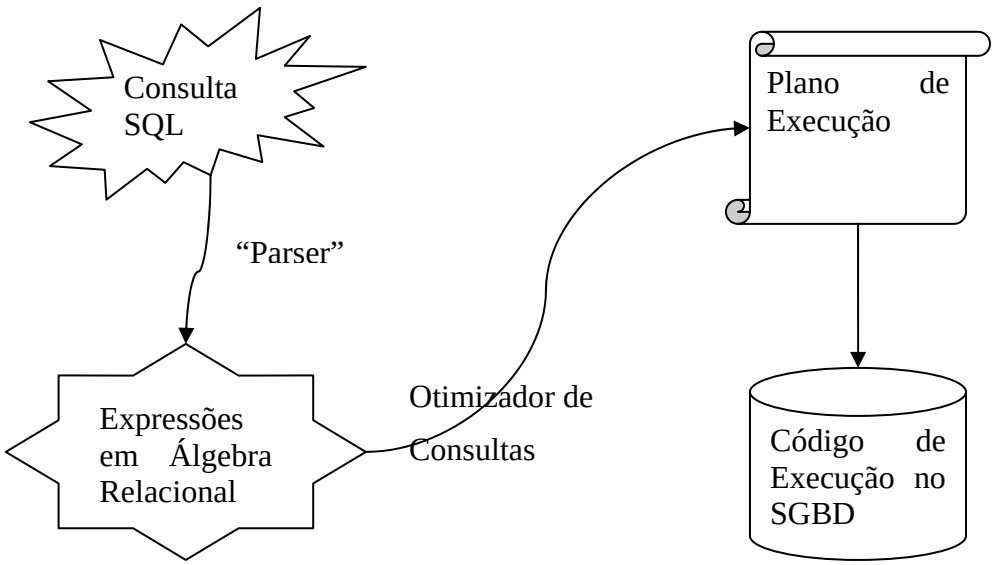
A consulta básica SQL acima tem um correspondente em álgebra relacional, que é o seguinte:

$\Pi_{A1, A2, A3,\ldots,An} (\sigma_P (R1 \times R2 \times R3 \times \ldots \times Rm))$

(projeção) (seleção) (produto cartesiano)

Observações:

- Se a cláusula WHERE é omitida, o predicado P é considerado como sempre verdadeiro.
- A lista de atributos pode ser substituída por um asterisco (*) que indica que queremos todos os atributos.
- **O resultado do SQL é uma relação**!
- Embora conceitualmente falando exista o produto cartesiano, os processadores da linguagem conseguem identificar junções e otimizam o processamento das consultas, através do otimizador de consultas.



2.3.1.1 A cláusula SELECT

Comecemos com um exemplo: encontre os nomes dos bancos na tabela Banco.

```
SELECT Nome
FROM Banco
```

No SQL a **eliminação de duplicatas não é automática** por ser ineficiente em muitos casos. Em SQL devemos explicitamente declarar que queremos a remoção de duplicatas, com a cláusula DISTINCT.

Exemplo: Encontre os IDs internos de clientes que têm empréstimos (terá uma tupla na tabela Emprestimo).

```
SELECT DISTINCT cli_id
FROM Emprestimo
```

cli_id
1
2
3
5
6

Sem a clásula distinct:

```
SELECT cli_id
FROM Emprestimo
```

Ou

```
SELECT ALL cli_id
FROM Emprestimo
```

cli_id
1
5
6
3
3
1
2
2
1

A lista de atributos pode ser substituída por asterisco (*):

```
SELECT *
FROM CLIENTE
```

ID	NOME	Tipo
1	Leonardo	F
2	Lucro Certo SA	J
3	Maria	F
4	Joao Pedro	F
5	Mudancas Joe	J
6	Armando Jose	F

É permitido o uso de operadores (aritméticos, concatenação de strings, uso de funções etc) nas cláusulas SELECT.

```
SELECT ID, (Saldo * 1.1) as SaldoMais10Porcento
FROM Conta
WHERE Saldo * 1.1 > 5000
```

É possível renomear o nome do campo.

É possível usar expressões na cláusula WHERE.

ID	SaldoMais10Porcento
1	99903,1
2	13200
4	55000
5	36643,2
8	52682,3
9	86826,3

2.3.1.2 A cláusula WHERE

Os predicados da cláusula WHERE são expressões lógicas (booleanas) e retornam verdadeiro ou falso para cada tupla processada.

É permitido o uso de conectores lógicos (AND, OR e NOT).

Exemplo: Recupere as movimentações de crédito superiores a 1000 nas contas correntes.

```
SELECT *
FROM Movimentacao
WHERE (Operacao = 'C') and (Valor > 1000)
```

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
3	1	R$ 1.500,00	C	x998	20/5/2000
5	2	R$ 1.800,00	C	3872	10/1/2000
6	2	R$ 1.900,00	C	2908	30/1/2000

Expressões aritméticas podem ser usadas também neste ponto. As expressões podem envolver constantes ou valores de campos da tupla.

Exemplo: Recupere os empréstimos de quem pagou um valor mais alto do que pegou emprestado.

```
SELECT *
FROM EMPRESTIMO
WHERE ValorPago > ValorEmprestado
```

ID	AGE_ID	CLI_ID	Data	DataPagamento	JurosMensais	ValorEmprestado	ValorPago
8	6	2	4/4/2002	30/4/2002	1	R$ 3.000,00	R$ 3.030,00

Para faixas de valores, o operador **between** pode ser usado.

Exemplo: Recupere as contas com saldo entre 1000 e 2000 (inclusive).

```
SELECT *
FROM CONTA
WHERE Saldo between 1000 and 2000
```

ID	AGE_ID	CLI_ID	CodigoConta	DataAbertura	Saldo
3	3	1	3215-9	6/1/1997	1000
7	7	2	3212-1	9/6/2002	1300

Exercício: como implementar o between sem usar a palavra “between”?

2.3.1.3 A cláusula FROM

Define um produto cartesiano entre as relações listadas.

Exemplo (de produto cartesiano na seção 2.1.3):

Encontrar uma combinação de serviços, indicando pessoas físicas para pessoas jurídicas venderem produtos ou serviços.

PESSOA FISICA X PESSOA JURIDICA

```
SELECT *
FROM PESSOAFISICA, PESSOAJURIDICA
```

PESSOAFISICA.ID	CPF	DataNascimento	PESSOAJURIDICA.ID	CGC	DataFormacao	InscricaoEstadual
1	07898281790	6/8/1977	2	87628871/0001	1/12/1957	29389
1	07898281790	6/8/1977	5	98767266/0002	6/7/1997	56416
3	12345678912	10/1/1980	2	87628871/0001	1/12/1957	29389
3	12345678912	10/1/1980	5	98767266/0002	6/7/1997	56416
4	98765432109	26/2/1969	2	87628871/0001	1/12/1957	29389
4	98765432109	26/2/1969	5	98767266/0002	6/7/1997	56416
6	56941236420	17/10/1940	2	87628871/0001	1/12/1957	29389
6	56941236420	17/10/1940	5	98767266/0002	6/7/1997	56416

As junções devem ser expressas na cláusula WHERE (lembre-se da definição de junção).

Exemplo: Recupere os nomes dos clientes pessoas físicas que têm alguma conta.

SELECT DISTINCT cliente.nome

FROM Cliente, Conta

WHERE (cliente.id = conta.cli_id) AND (cliente.tipo = ‘F’)

nome
Armando Jose
Joao Pedro
Leonardo
Maria

Exemplo 2: Recupere as movimentações da conta Código ‘3323-4’

```
SELECT movimentacao.*
```

Semi-Junção. Não recuperou campos de “Conta”.

```
FROM Movimentacao, Conta
WHERE
 (movimentacao.conta_id = conta.id) and conta.CodigoConta = ‘3323-4’
```

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
5	2	R$ 1.800,00	C	3872	10/1/2000
6	2	R$ 1.900,00	C	2908	30/1/2000
7	2	R$ 100,00	C	2398	20/5/2000
8	2	R$ 50,00	C	9823	10/1/2001

2.3.1.4 A Renomeação

O SQL provê um mecanismo para implementação da renomeação, tanto em nível de atributos como de relações.

A palavra “AS” precede o novo nome desejado.

Exemplo:

```
Select Nome as NomedoBanco, Codigo as CodigodoBanco
From Banco
```

NomedoBanco	CodigodoBanco
Banco do Brasil	001
Banco Real	356
Banco Bradesco	789
Banco Itau	129

Exemplo 2: com junção.

```
SELECT m.*
FROM Movimentacao as M, Conta as C
WHERE
 (m.conta_id = c.id) and c.CodigoConta = ‘3323-4’
```

(O resultado é o mesmo da consulta que ilustou a semi-junção)

Exemplo 3: Encontrar as agências que são da mesma cidade que a agência código número 6 (já vimos em álgebra relacional).

```
SELECT A2.*
FROM Agencia as A1, Agencia as A2
WHERE (A1.ID = 6) AND (A1.Cidade = A2.Cidade)
```

Observação:

- Em diversas implementações do SQL, a palavra “as” pode ser omitida.

2.3.1.5 Operadores de Strings

O operador para encontrar padrões de strings é indicado pela palavra "**like**". Como nos principais sistemas operacionais, existem caracteres especiais para uso com o operador **LIKE**.

- O "%" pode ser substituído por qualquer substring.
- O "_" (underscore ou underline) pode ser substituído por qualquer caracter.
- Caracter de "escape" → '\' (barra invertida) busca por % ou _ (no seu valor específico).

Por exemplo, a expressão "___%" é verdadeira para qualquer string com pelo menos três posições preenchidas.

Os padrões são sensíveis a maiúsculas (*case sensitive*), assim "Leo" é diferente de "leo".

É possível usar a negação – NOT LIKE.

Exemplo: Recupere os Bancos cujo nome começa termina com 'Brasil'.

```
SELECT *
FROM Banco
WHERE Nome like '%Brasil'
```

ID	Nome	Codigo
1	Banco do Brasil	001

Ainda existe o operador de concatenação de Strings "||" (duas barras verticais seguidas).

Exemplo: Formatar o Nome o CPF das pessoas físicas em uma única coluna.

```
SELECT (C.Nome || ' – ' || F.CPF) as Nome_e_CPF
FROM Cliente C, PessoaFisica F
WHERE C.ID = F.ID
```

Nome_e_CPF
Leonardo –07898281790
Maria –12345678912
Joao Pedro –98765432109
Armando Jose –56941236420

2.3.1.6 Ordenação de Resultado

Para ordenar um resultado de SQL, usamos a cláusula ORDER BY após a cláusula WHERE.

Para ordenação ascendente, usamos a palavra ASC, para descendente DESC. "ASC" é o default e pode ser omitido.

Exemplo: Recupere as movimentações de contas de maneira descrescente na Data de Movimentação

```
SELECT *
```

```
FROM Movimentacao
ORDER BY Data DESC
```

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
11	7	R$ 350,00	C	5783	14/7/2002
8	2	R$ 50,00	C	9823	10/1/2001
4	1	R$ 4.500,00	D	x997	10/1/2001
12	6	R$ 450,00	D	9234	15/11/2000
7	2	R$ 100,00	C	2398	20/5/2000
3	1	R$ 1.500,00	C	x998	20/5/2000
6	2	R$ 1.900,00	C	2908	30/1/2000
2	1	R$ 1.500,00	D	x997	30/1/2000
5	2	R$ 1.800,00	C	3872	10/1/2000
1	1	R$ 500,00	C	x998	10/1/2000
10	8	R$ 200,00	D	3444	11/5/1999
9	9	R$ 100,00	C	2111	10/1/1998
13	5	R$ 550,00	C	9324	15/10/1997

Exemplo 2: Liste as pessoas físicas na ordem da sua data de nascimento

```
SELECT *
FROM PessoaFisica
ORDER BY DataNascimento
```

ID	CPF	DataNascimento
6	56941236420	17/10/1940
4	98765432109	26/2/1969
1	07898281790	6/8/1977
3	12345678912	10/1/1980

Observação: o uso de ordenação pode ser custoso e só deve ser usado quando necessário. A ordenação tem complexidade O (N log N).

2.3.2 Operações com Conjuntos

Os operadores de conjuntos da álgebra relacional estão presentes no SQL, são eles: *union*, *intersect* e *except (minus)*.

Exemplo:

Encontre os códigos dos clientes que têm conta.

```
SELECT DISTINCT CLI_ID FROM CONTA
```

Encontre os códigos dos clientes que têm conta ou empréstimo.

```
(SELECT CLI_ID FROM CONTA)
UNION
(SELECT CLI_ID FROM EMPRESTIMO)
```

Encontre os códigos dos clients que têm conta e empréstimo.

```
(SELECT CLI_ID FROM CONTA)
INTERSECT
(SELECT CLI_ID FROM EMPRESTIMO)
```

Encontre os códigos dos clientes que têm conta, **mas não têm** empréstimos.

```
(SELECT CLI_ID FROM CONTA)
EXCEPT
(SELECT CLI_ID FROM EMPRESTIMO)
```

Observações:

- "UNION" elimina duplicatas por default. Para manter as duplicatas, é necessário usar "UNION ALL". O mesmo vale para "intersect" e "except".
- Nem todas as implementações de SQL apresentam estes operadores de conjuntos.
- A operação "Except" advém do SQL-92. Seu nome na primeira versão (SQL-86) era "Minus", mas a operação é a subtração de conjuntos. Só nome foi modificado.

2.3.3 Funções de Agregação

Em SQL é possível computar valores de funções em grupos de tuplas através de funções de agregação. As funções de agregação são as seguintes (SQL padrão):

- Count → número de tuplas
- Sum → Somatório
- Max → Valor Máximo
- Min → Valor Mínimo
- Avg → Média Aritmética

As funções agregadas retornam um valor somente, para todo o grupo.

Exemplo: Calcule o saldo médio das contas existentes.

```
SELECT AVG(SALDO) as SaldoMedio
FROM Conta
```

SaldoMedio
35117,6666666667

Exemplo 2: Recupere o valor máximo e mínimo movimentado nas contas.

```
SELECT MAX(Valor) as maximo, MIN(Valor) as minimo
FROM Movimentacao
```

maximo	minimo
R$ 4.500,00	R$ 50,00

Exemplo 3: Recupere o número de contas existentes no banco.

```
SELECT COUNT(*) as numerodecontas
FROM CONTA
```

numerodecontas
9

Nos exemplos acima, tratamos a relação toda como um grupo, calculando uma média, valor máximo, valor mínimo e contagem da tabela inteira. **É possível "quebrar" a relação em grupos com algum tipo de significado**. Os grupos são formados a partir da cláusula "GROUP BY" que deve ser escrita após a cláusula "WHERE" e antes de um possível "ORDER BY".

Exemplo: Recupere o saldo médio por Nome de Agencia, ordene por este valor.

```
SELECT A.Nome, AVG(C.Saldo) as SaldoMedioAgencia
FROM Conta as C, Agencia as A
WHERE (C.AGE_ID = A.ID)
GROUP BY A.Nome
ORDER BY AVG(C.Saldo) DESC
```

Nome	SaldoMedioAgencia
Ag Centro	90821
Ag Barra	78933
Ag Ondina	50000
Ag Ministerios	47893
Ag Tiete 2	33312
Ag Tijuca	12000
Ag Borel	1300
Ag Tiete	1000
Ag Pinhes	800

Exemplo 2: Encontre o número de correntistas por cidade.

```
SELECT A.Cidade, Count(DISTINCT C.CLI_ID) as NumeroCorrentistas
FROM Agencia as A, Conta as C
WHERE (C.AGE_ID = A.ID)
GROUP BY A.Cidade
```

Cidade	NumeroCorrentistas
Brasilia	1
Rio de Janeiro	3
Salvador	2
Sao Paulo	3

Utilizamos “DISTINCT” pois uma pessoa com mais de uma conta na cidade, só deve ser contada uma única vez.

Exemplo 3: Encontre as contas cujo valor máximo de movimentação excedeu 1380 reais nas operações de crédito.

```
SELECT CONTA_ID, MAX(Valor)
FROM Movimentacao
WHERE Operacao = ‘C’
GROUP BY CONTA_ID
HAVING MAX(VALOR) > 1380
```

CONTA_ID	Expr1001
1	R$ 1.500,00
2	R$ 1.900,00

A cláusula “HAVING” é aplicada após a formação dos grupos. É uma espécie de segundo “WHERE”. A cláusula “HAVING” normalmente envolve função de agregação.

Exemplo 4: Encontre o saldo médio de conta corrente por nome de cliente que tenha mais de uma conta.

```
SELECT c.NOME, Avg(cc.Saldo) AS MédiaDeSaldo
FROM cliente AS c, conta AS cc
WHERE c.ID=cc.cli_id
GROUP BY c.NOME
HAVING Count( DISTINCT cc.CodigoConta ) > 1
```

NOME	MédiaDeSaldo
Pedro	16283,4935064935
Usinagens INC	11740,4252136752

Neste exemplo, usamos uma função de agregação no SELECT e outra no HAVING!

Observações:

- Durante o processamento da consulta, as tuplas que satisfazem a cláusula WHERE são alocadas nos grupos definidos pela cláusula ORDER BY.

- Após a criação dos grupos (processamento de todas as tuplas) a cláusula HAVING é aplicada para cada grupo, sendo uma segunda "filtragem".
- Somente os grupos que satisfazem a cláusula HAVING são enviados como resposta à consulta.

2.3.4 Valores Nulos (NULL)

O "nulo" representa a **ausência de valor** e não deve ser confundido com zero, strings vazias, datas iniciais etc. A ausência de valor não pode ser **COMPARADA** convencionalmente nem figurar em funções de agregação. Qualquer comparação com o valor nulo retorna falso, o que exclui a tupla de um possível resultado.

O SQL provê uma expressão especial para comparação de nulos: "is null" e "is not null".

Para testar se um campo tem valor nulo em uma cláusula WHERE, usamos "WHERE campo IS NULL".

Observação: existem comandos DDL que impedem que uma coluna possa assumir o valor nulo.

Exemplos:

Tabela "TesteNulos" (os IDs 3 e 5 têm "Numero" como NULL)

ID	NOME	Numero
1	ABC	10
2	DEF	15
3	GHI	
4	JKL	30
5	MNO	
6	PQR	10

Observe as consultas e os resultados:

```
SELECT COUNT(NUMERO) FROM TESTENULOS
```

Expr1000
4

```
SELECT AVG(NUMERO) FROM TESTENULOS
```

Expr1000
16,25

```
SELECT * FROM TESTENULOS WHERE NUMERO = NULL
```

ID	NOME	Numero

```
SELECT * FROM TESTENULOS WHERE NUMERO IS NULL
```

ID	NOME	Numero
3	GHI	
5	MNO	

2.3.5 SubConsultas Aninhadas (*Subqueries*)

Em algumas situações é importante a utilização de subconsultas. Embora sua implementação seja, na maioria dos casos, bastante custosa, seu funcionamento é bastante importante.

2.3.5.1 Pertencer ao Conjunto

Para relações de pertinência, as operações “IN” e “NOT IN” são usadas.

Exemplos:

Encontrar os IDs de clientes que têm empréstimos:

```
SELECT DISTINCT C.ID
FROM CLIENTE C
WHERE C.ID IN (SELECT E.CLI_ID FROM EMPRESTIMO E)
```

ID
1
2
3
5
6

Como escrever esta consulta de outra forma? Sem subconsulta? Já fizemos antes...

Mais de um atributo pode ser testado, ou seja, cada elemento na operação IN se comporta como uma tupla e não como um valor atômico.

Encontrar o par de IDs agencia e cliente que figuram em “conta” e “empréstimo”.

```
SELECT DISTINCT C.CLI_ID, C.AGE_ID
FROM CONTA C
WHERE (C.CLI_ID, C.AGE_ID) IN
            (SELECT E.CLI_ID, E.AGE_ID FROM EMPRESTIMO E)
```

Observação: a consulta acima não roda no MSAccess, mas funciona no Oracle, por exemplo.

Encontrar os IDs de clientes que não têm empréstimos:

```
SELECT DISTINCT C.ID
FROM CLIENTE C
WHERE C.ID NOT IN (SELECT E.CLI_ID FROM EMPRESTIMO E)
```

ID
4

Outra maneira de calcular aqueles que não têm empréstimos?

2.3.5.2 Comparação de Conjuntos

Para comparar um valor contra um conjunto, usamos este tipo de operação.

Exemplos:

Encontrar as movimentações de créditos da conta ID = 2 maiores que **alguma** movimentação da conta ID = 1

```
SELECT M.*
FROM MOVIMENTACAO M
WHERE
  M.Operacao = 'C'
  AND M.CONTA_ID = 2
  AND M.Valor > some
                (SELECT M2.valor
                 FROM Movimentacao M2
                 WHERE M2.CONTA_ID = 1)
```

ID	CONTA_ID	Valor	Operacao	CodigoDocumento	Data
5	2	R$ 1.800,00	C	3872	10/1/2000
6	2	R$ 1.900,00	C	2908	30/1/2000

Encontrar as movimentações de créditos da conta ID = 2 maiores que **TODAS** as movimentações da conta ID = 1

```
SELECT M.*
FROM MOVIMENTACAO M
WHERE
  M.Operacao = 'C'
  AND M.CONTA_ID = 2
  AND M.Valor > ALL
                (SELECT M2.valor
                 FROM Movimentacao M2
```

```
WHERE M2.CONTA_ID = 1)
```

ID	CONTA_ID	Operacao	Valor	CodigoDocumento	Data

Encontrar os ID de Agência que têm o saldo médio em Conta Corrente maior que a média geral do banco central.

Nota: Não é possível usar MAX(AVG(SALDO))!!

Para encontrar o grupo de médias por agência, a solução é simples:

```
select age_id, avg(saldo)
from conta
group by age_id
```

age_id	Expr1001
1	90821
2	12000
3	1000
4	50000
5	33312
6	800
7	1300
8	47893
9	78933

Para encontrar a média geral, a solução também é simples:

```
select avg(saldo)
from conta
```

Expr1000
35117,666667

Finalmente...

```
select age_id, avg(saldo)
from conta
group by age_id
having avg(saldo) >= all (select avg(saldo) from conta)
```

age_id	Expr1001
1	90821
4	50000
8	47893
9	78933

O máximo saldo médio é dado pela consulta abaixo:

```
select age_id, avg(saldo)
from conta
group by age_id
having avg(saldo) >= all
            (select avg(saldo) from conta group by age_id)
```

age_id	Expr1001
1	90821

Por que? :)

2.3.5.3 Teste de Relações Vazias

O construtor “EXISTS” testa se o resultado da subconsulta retorna algo. Caso retorne, “EXISTS” é verdadeiro. É possível utilizar “NOT EXISTS”.

Exemplos:

Encontre as contas que têm movimentação em 2002.

```
SELECT C.*
FROM Conta C
WHERE EXISTS
      (SELECT M.ID
       FROM Movimentacao M
       WHERE M.Data >= #01-01-2002#
         AND M.Data <= #31-12-2002#
         AND M.CONTA_ID = C.ID)
```

ID	AGE_ID	CLI_ID	CodigoConta	DataAbertura	Saldo
7	7	2	3212-1	9/6/2002	1300

Encontre as contas que não têm movimentação em Janeiro de 2003.

```
SELECT C.*
FROM Conta C
WHERE NOT EXISTS
      (SELECT M.ID
       FROM Movimentacao M
       WHERE M.Data >= #01-01-2003#
         AND M.Data <= #31-01-2003#
         AND M.CONTA_ID = C.ID)
```

ID	AGE_ID	CLI_ID	CodigoConta	DataAbertura	Saldo
1	1	4	3652-8	13/1/2002	90821
2	2	4	3323-4	23/12/1998	12000
3	3	1	3215-9	6/1/1997	1000
4	4	3	9873-8	3/3/2000	50000
5	5	5	4982-1	18/8/1999	33312
6	6	6	3177-2	14/7/2003	800
7	7	2	3212-1	9/6/2002	1300
8	8	2	0098-0	20/5/2001	47893
9	9	1	3277-1	3/10/2002	78933

2.3.6 Relações Derivadas

Às vezes é importante ver o resultado de uma consulta como uma relação existente na cláusula FROM. Esta construção é permitida no SQL, colocando-se uma subconsulta na cláusula FROM.

Exemplos:

Implementando “na mão” o Having...

```
select r.age_id, r.saldomedio
from
    (select age_id, avg(saldo) as saldomedio
     from conta
     group by age_id)
     as r
where r.saldomedio > 32750
```

age_id	saldomedio
1	90821
4	50000
5	33312
8	47893
9	78933

2.3.7 Modificando os Dados

Até o presente momento nós apenas vimos meios de consultar dados existentes no banco de dados. A linguagem SQL também apresenta uma forma declarativa para **inserir, atualizar e remover** dados em SGBDS.

2.3.7.1 DELETE

A operação de remoção é expressa pelo comando **DELETE** e tem a mesma estrutura de um comando SELECT, a diferença é que as tuplas selecionadas na cláusula WHERE não serão mostradas ao usuário, mas sim removidas permanentemente do banco de dados.

A estrutura do comando DELETE em SQL é a seguinte:

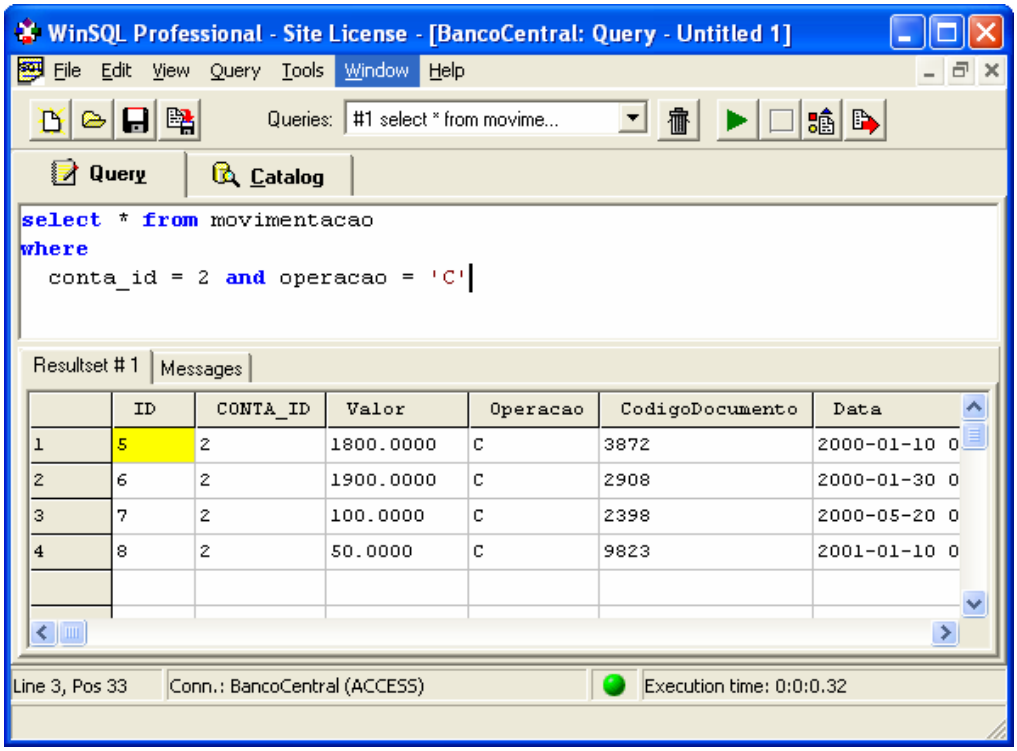
```
DELETE FROM <<TABELA>>
WHERE <<CONDICAO>>
```

Repare que um comando **DELETE** sem cláusula **WHERE** remove todas as tuplas de uma tabela.

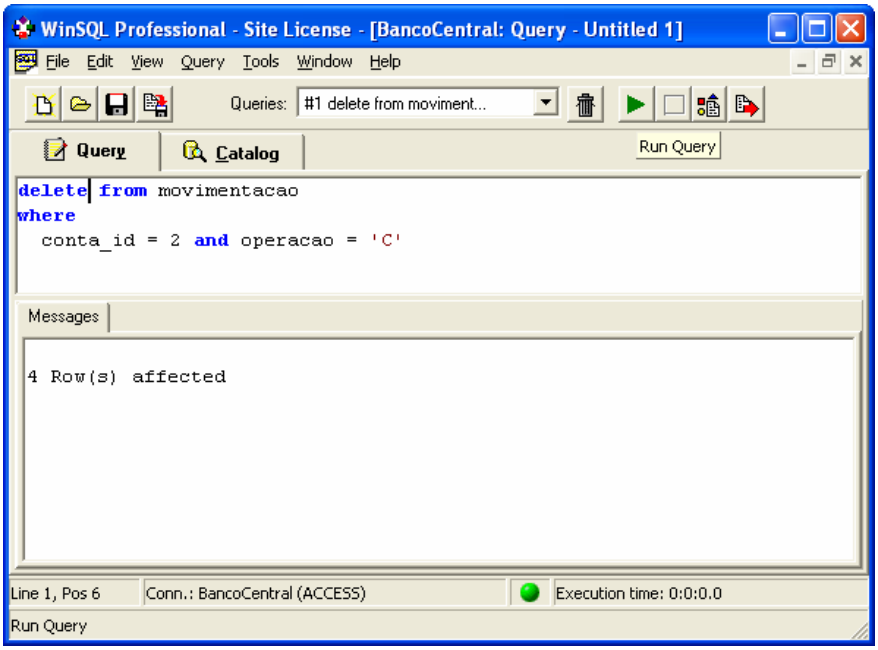
Exemplo:

Remover as movimentações de crédito da conta com CONTA_ID = 2.

Repare o comando SELECT:



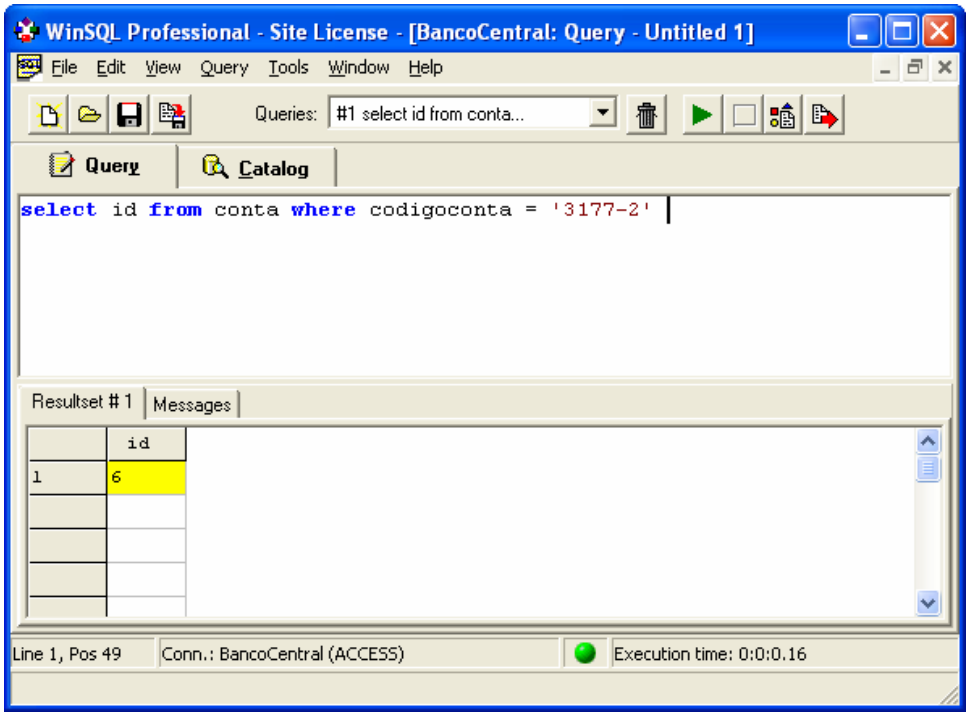
Substituindo “SELECT *” por “DELETE”, temos:



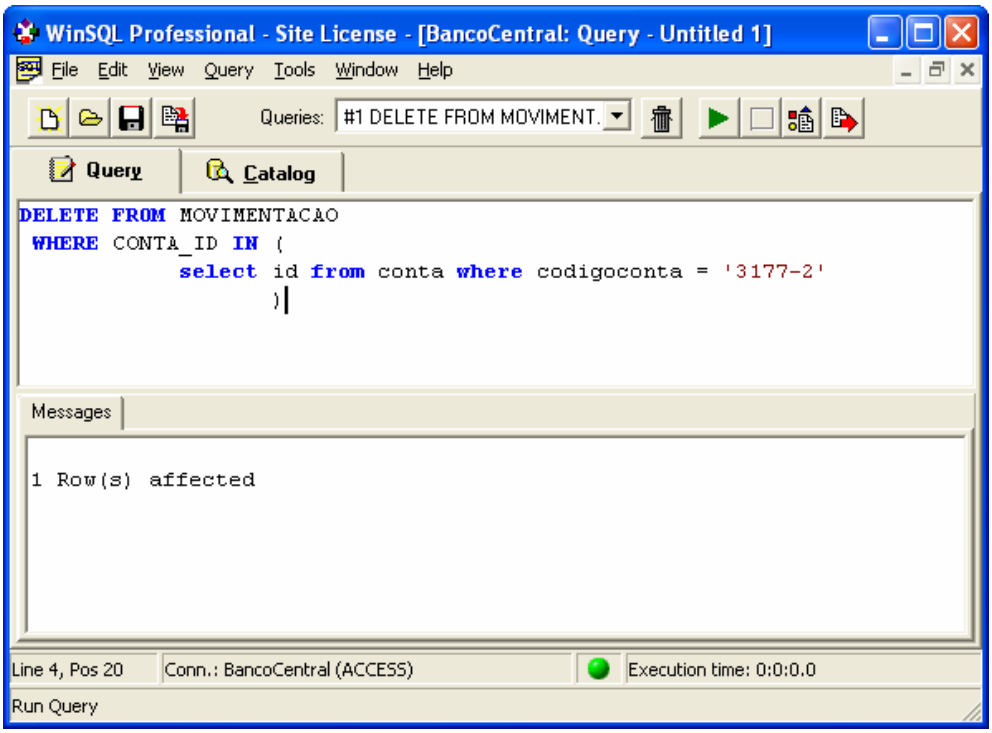
Um comando DELETE pode somente remover tuplas de uma única relação por vez. Entretanto, a sua cláusula WHERE pode conter subconsultas que acessam outras relações existentes na base de dados, por exemplo:

Exemplo:

Pegando o ID interno de um determinado Codigo de Conta



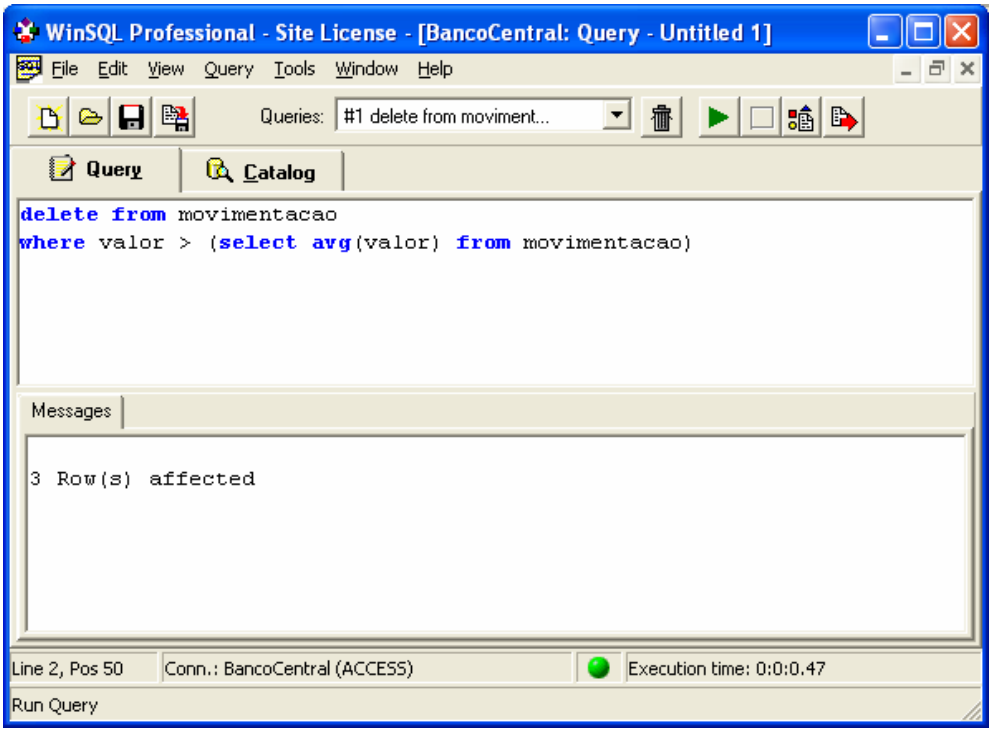
Apagando as movimentações desta conta:



Como o comando DELETE altera a base de dados, este pode modificar resultados de funções de agregação (count, avg, sum, min, max etc).

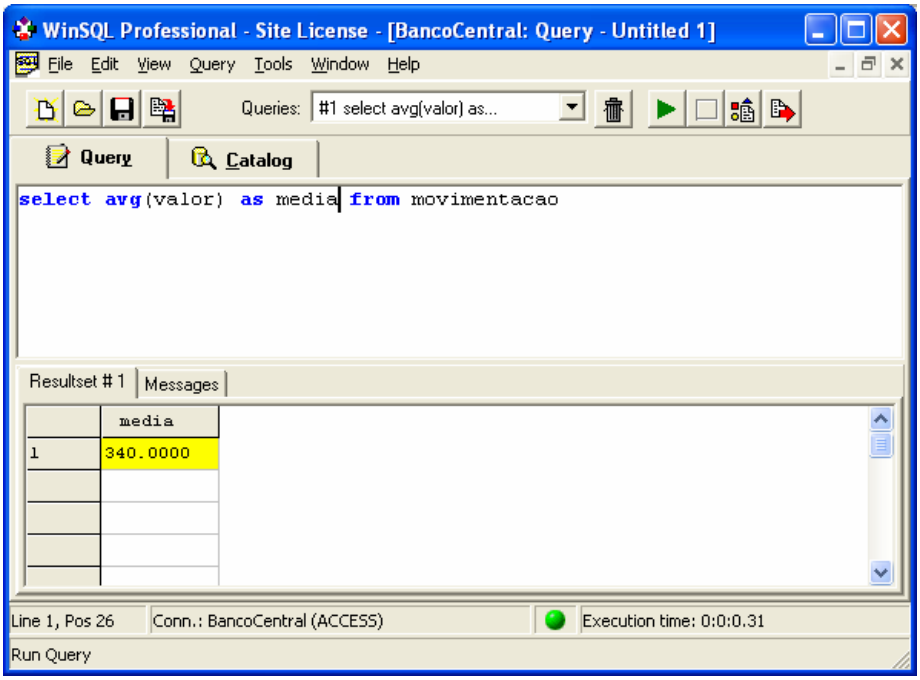
Exemplo:

Como apagar as movimentações maiores do que a média de movimentações existentes na base de dados? Quando apagamos uma tupla, a média muda!!!

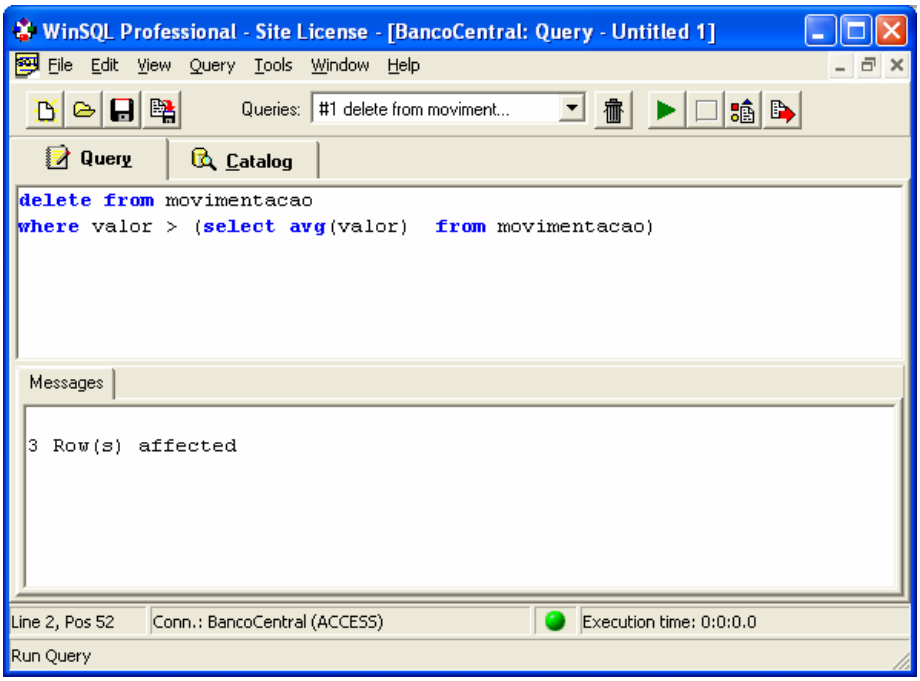


A solução encontrada **pelo SGBD é realizar primeiramente a subconsulta**, calculando o valor médio. Só então as tuplas que satisfazem a cláusula principal do comando WHERE são removidas.

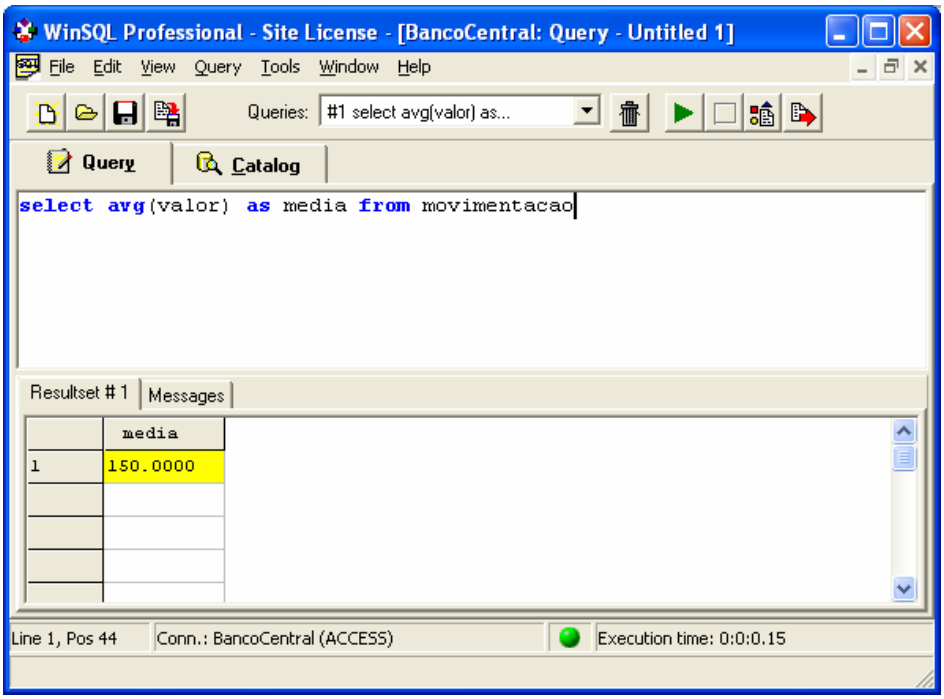
A média mudou...



Se aplicarmos o comando de novo, novas tuplas serão removidas:



Assim como a média...



2.3.7.2 INSERT

O comando **INSERT** permite que novas tuplas sejam inseridas em uma relação do banco de dados.

É possível informar completamente a tupla ou inserir em uma tabela o resultado de uma consulta SELECT.

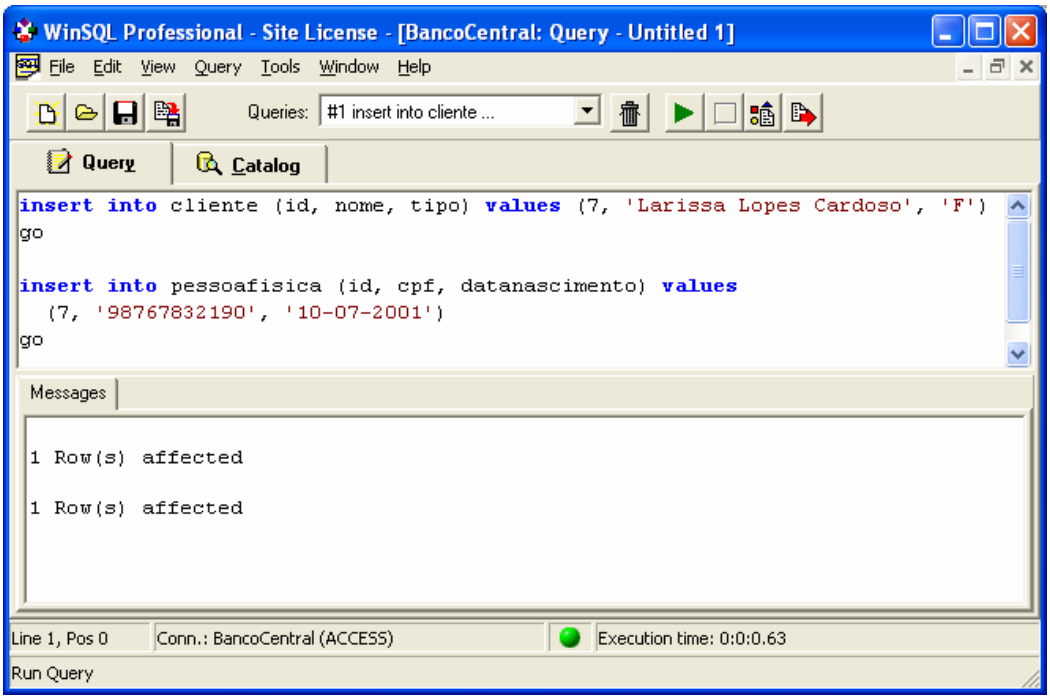
A estrutura do comando INSERT em SQL é a seguinte:

```
INSERT INTO <<TABELA>> (<<CAMPOS>>)
VALUES (<<VALORES PARA OS CAMPOS>>)
Ou
INSERT INTO <<TABELA>> (<<CAMPOS>>)
SELECT <<CAMPOS COMPATIVEIS>> FROM <<TABELA>>
```

Campos omitidos recebem o valor NULL.

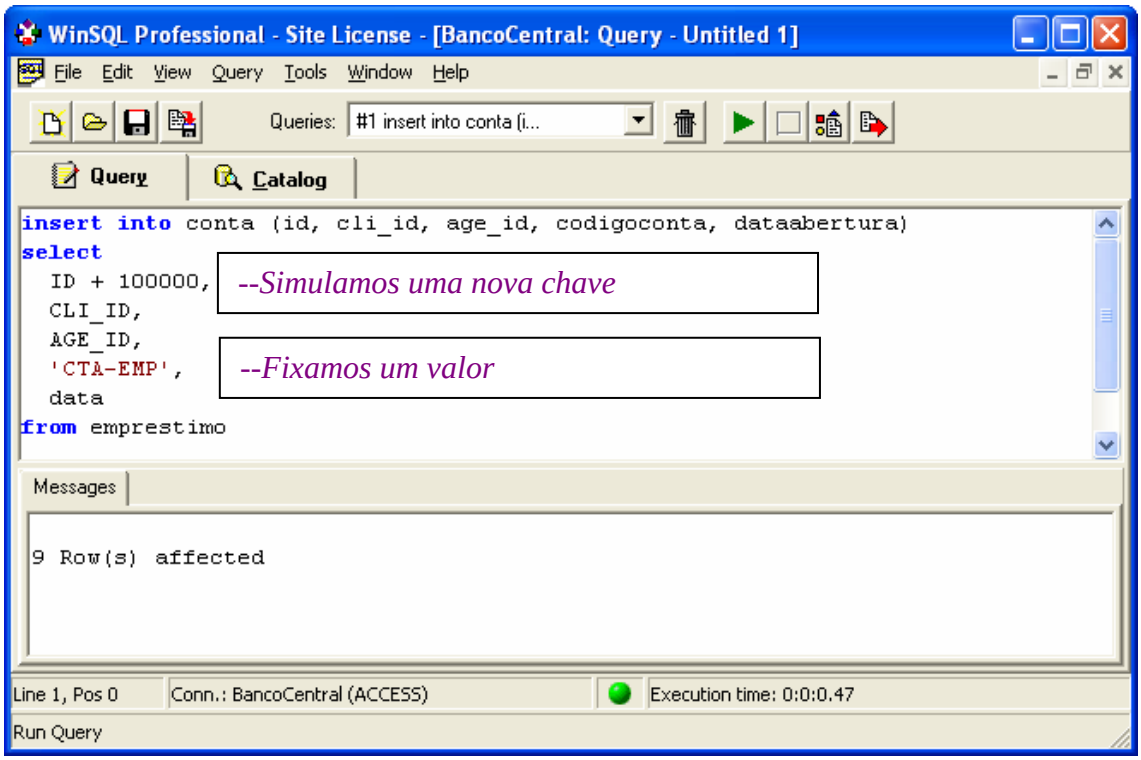
Exemplo: Inserir um novo cliente pessoa física na base.

Esta operação consiste na inserção de uma nova tupla na tabela Cliente e de uma outra tupla relacionada em PessoaFisica:



Se deu a louca no Banco Central (praxe) e eles resolvem transformar os empréstimos em Contas?

Inserimos as “novas” contas diretamente da tabela EMPRESTIMO.



Boa prática: ao usar um INSERT a partir de um SELECT, primeiro faça o SELECT sozinho para ter noção que a operação será a desejada.

2.3.7.3 UPDATE - Atualizações

O comando **UPDATE** permite que tuplas existentes no banco de dados tenham determinados (pelo menos um) campos modificados.

O comando UPDATE também utiliza uma cláusula WHERE que irá guiar a operação de atualização, ou seja, somente tuplas que gerem verdadeiro na cláusula WHERE terão seus valores potencialmente modificados.

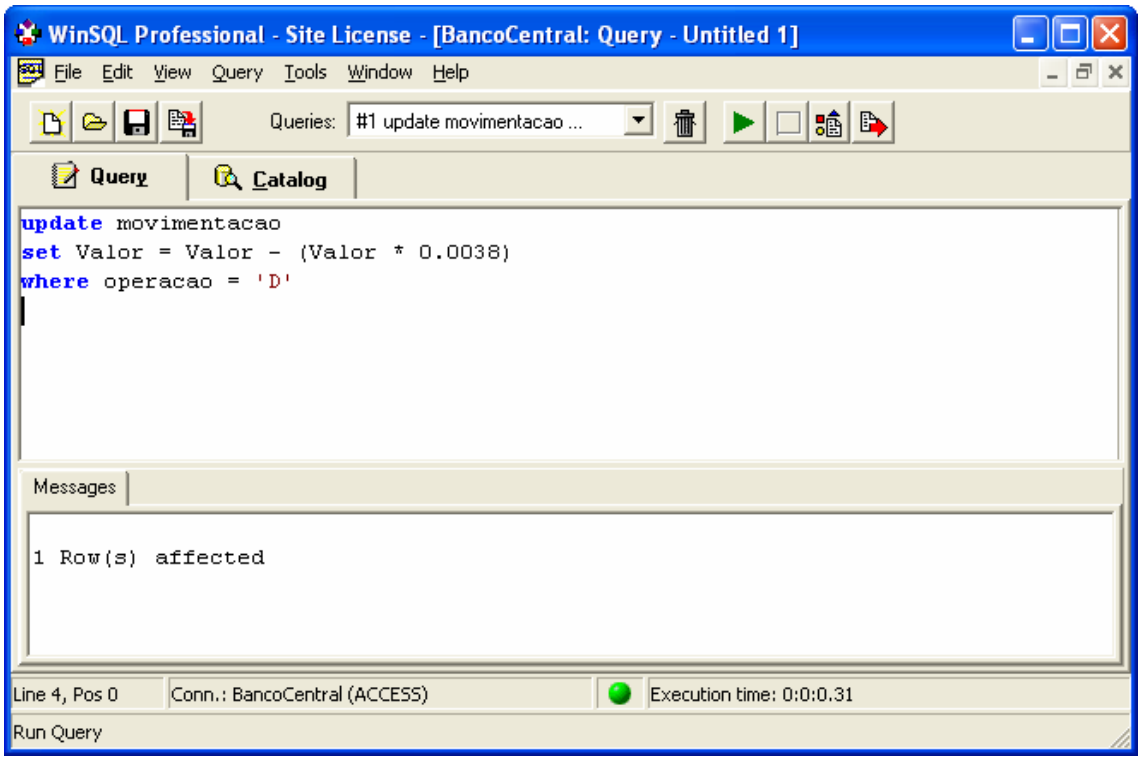
Em geral, uma cláusula WHERE válida em um comando SELECT, também é válida em um comando UPDATE.

A estrutura do comando UPDATE em SQL é a seguinte:

```
UPDATE <<TABELA>>
SET <<CAMPO1>> = <<NOVO VALOR 1>>,
    <<CAMPO2>> = <<NOVO VALOR 2>>,
    ...
WHERE <<CONDICAO>>
```

Exemplos:

Tirar o CPMF (subtraindo 0.38%) das movimentações de débito em conta:



2.3.8 DDL (Linguagem de Definição de Dados)

O SQL provê meios declarativos para a definição de objetos dentro do banco de dados. Por objetos, entenda-se:

- Domínios
- Esquemas
- Restrições de Integridade
- Índices para Tabelas
- Segurança
- Modos de armazenamento físico

2.3.8.1 Tipos de Domínios Existentes no SQL-92

char(n) (or character(n)): fixed-length character string, with user-specified length.

varchar(n) (or character varying): variable-length character string, with user-specified maximum length.

int or integer: an integer (length is machine-dependent).

smallint: a small integer (length is machine-dependent).

numeric(p, d): a fixed-point number with user-specified precision, consists of p digits (plus a sign) and d of p digits are to the right of the decimal point. E.g., numeric(3, 1) allows 44.5 to be stored exactly but not 444.5.

real or double precision: floating-point or double-precision floating-point numbers, with machine-dependent precision.

float(n): floating-point, with user-specified precision of at least n digits.

date: a calendar date, containing four digit year, month, and day of the month.

time: the time of the day in hours, minutes, and seconds.

É possivel criar um novo tipo de dados, semelhantemente ao mecanismo usado em linguagens de programação:

create domain person-name char(20)

2.3.8.2 Definição de Esquemas

Uma tabela pode ser definida em SQL através do comando “CREATE TABLE”.

```
CREATE TABLE (
  A1 D1,
  A2 D2,
  ...
  AN DN,
  <<restrições de integridade>>
  )
```

As restrições de integridade incluem integridade referencial (foreign key) e chaves primárias e checagem de valores.

Exemplos (sintaxe do MS Access):

```
CREATE TABLE Banco(
    ID                  INTEGER,
    Nome                VARCHAR(100),
    Codigo              VARCHAR(10)
)
GO
CREATE TABLE Agencia(
    ID                  INTEGER,
    BANCO_ID            INTEGER,
    Cidade              VARCHAR(50),
    Nome                VARCHAR(50),
    Codigo              VARCHAR(20)
)
```

```
GO
CREATE TABLE Cliente(
    ID                  INTEGER,
    NOME                VARCHAR(50),
    Tipo                VARCHAR(1)
)
GO
CREATE TABLE PessoaFisica(
    ID                  INTEGER,
    CPF                 VARCHAR(20),
    DataNascimento      DATETIME
)
GO
CREATE TABLE PessoaJuridica(
    ID                  INTEGER,
    CGC                 VARCHAR(20),
    DataFormacao        DATETIME,
    InscricaoEstadual   VARCHAR(20)
)
GO

CREATE TABLE Emprestimo(
    ID                  COUNTER NOT NULL,
    CLI_ID              INTEGER,
    AGE_ID              INTEGER,
    Data                DATETIME,
    DataPagamento       DATETIME,
    JurosMensais        DOUBLE,
    ValorEmprestado     CURRENCY,
    ValorPago           CURRENCY
)
GO
CREATE TABLE Conta(
    ID                  COUNTER NOT NULL,
    AGE_ID              INTEGER,
    CLI_ID              INTEGER,
```

```
    CodigoConta          VARCHAR(50),
    DataAbertura         DATETIME,
    Saldo                INTEGER
)
GO
CREATE TABLE Movimentacao(
    ID                   COUNTER NOT NULL,
    CONTA_ID             INTEGER,
    Operacao             VARCHAR(1),
    Valor                CURRENCY,
    CodigoDocumento      VARCHAR(10),
    Data                 DATETIME
)
GO
```

Para remover tabelas em uma base de dados, é usado o comando “DROP TABLE”.

Exemplo:

```
create table filme
(
 id integer primary key,
 nome varchar(50)
)
GO
drop table filme
GO
```

Para modificar tabelas, usamos o comando “ALTER TABLE”. Por modificação leia-se adição, exclusão e modificação de atributos e inserção e remoção de restrições de integridade.

Exemplo:

```
drop table filme
go
```

```
create table filme
(
 id integer primary key,
 nome varchar(50)
)
GO

alter table filme alter column nome varchar(50) not null
GO

drop table ator
go

create table ator
(id integer primary key,
 nome varchar(60) not null
 )
GO

drop table participacao
go

create table participacao
 (
  filme_id integer,
  ator_id integer,
  data date not null
 )
GO

ALTER TABLE participacao ADD
 CONSTRAINT [FK_filme] FOREIGN KEY
 (
  filme_id
 ) REFERENCES filme (id)
GO
```

```
alter table participacao
drop CONSTRAINT FK_filme
go
```

Observação: A maioria dos SGBDs apresenta pequenas variações na sintaxe empregada nos comandos da DDL. Uma leitura no manual de cada produto resolve o problema.

Capítulo 3 - Processamento de Consultas

Neste capítulo nos concentraremos no estudo de técnicas de indexação e do processamento de junções em bancos de dados.

Em diversas consultas estudadas nos capítulos anteriores, apenas um pequeno conjunto das tabelas era solicitado como resultado. Por exemplo, encontrar todos as agências do banco com ID = 293 significa efetuar a consulta: "SELECT * FROM AGENCIA WHERE BANCO_ID = 293". Quando estamos falando de um enorme número de bancos, esta consulta pode ficar custosa, pois a única maneira de resolvê-la será percorrendo toda a tabela "Agência" a cada execução da consulta.

Quando é necessário o acesso direto e rápido a determinados registros, precisamos usar estruturas auxiliares que irão desempenhar esta tarefa. A estas estruturas auxiliares, damos o nome de índices.

3.1 Conceitos Básicos

Um índice para uma tabela funciona como um catálogo em uma biblioteca, onde fichas ordenadas por autor facilmente nos permitem encontrar os livros de um determinado escritor, sem que tenhamos que procurar em todos os livros das estantes.

Embora a analogia acima seja completamente válida, no mundo dos bancos de dados, as técnicas de indexação são mais sofisticadas.

Existem dois tipos de índices:

- Ordenados (*Ordered indices*) → são baseados em uma lista ordenada de valores dos campos.
- de Dispersão (*Hash indices*) → são baseados em uma dispersão dos valores dos campos através de uma função de HASH. Cada valor de campo é mapeado para uma posição (*bucket*), de forma que ela possa ser acessada em tempo constante.

Assim como nas livrarias, é possível ter mais de uma estrutura de índice por tabela, visto que podemos ter as fichas ordenadas por autor, editora, nome do livro, ano de publicação, assunto etc. Além disso, nos bancos de dados, um índice pode ser construído sobre um ou um conjunto de mais atributos.

Até agora só vimos maravilhas no uso de índices, entretanto o funcionamento do banco de dados não é composto apenas de operações de leitura (consultas). Desta forma, é necessário avaliar o impacto do uso de índices nas operações de inclusão / remoção e no quesito de espaço de armazenamento.

Os métodos de indexação serão avaliados segundo os seguintes critérios:

1. Tipo de Acesso → o índice é bom para acesso direto, para busca em faixas de valores etc?
2. Tempo de Acesso → tempo para encontrar um item ou um conjunto de itens, usando o índice.
3. Tempo de Inclusão → tempo para inserir um novo item no índice (inclui o tempo de busca da posição correta dentro da própria estrutura de índice).

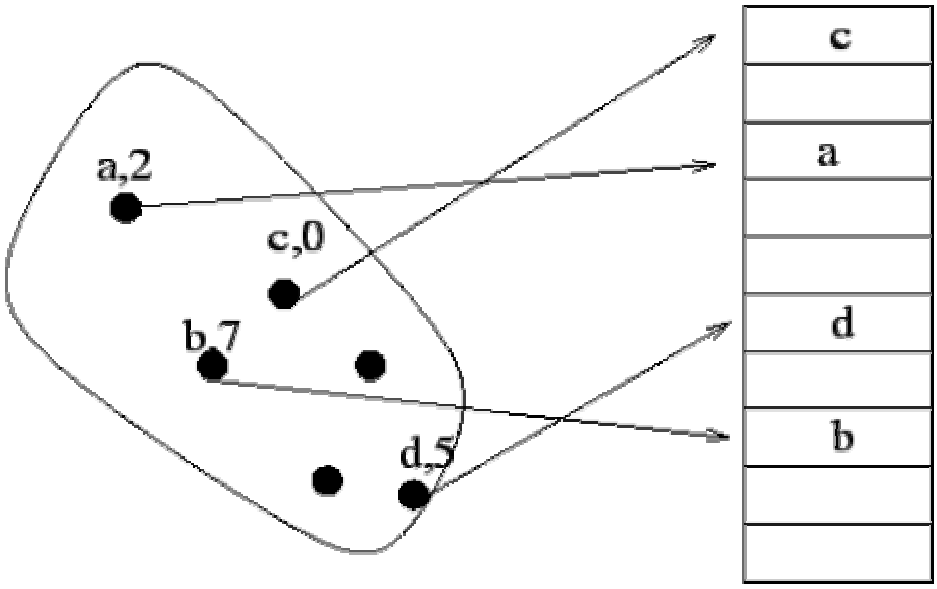
4. Tempo de Remoção → tempo para remover um item do índice (inclui o tempo para encontrar o ítem a ser removido e a atualização das estruturas).
5. Uso de Espaço → espaço ocupado pela estrutura de indexação.

Nas próximas seções, estaremos examinando diversas técnicas de indexação. Não existe uma técnica melhor em todas as situações, ou seja, é importante identificar a melhor maneira de indexação, dada a forma de consulta aos dados e a natureza da aplicação do banco de dados.

3.1.1 Revisão HASHING

``Hashing" é uma técnica que busca realizar as operações de inserção, remoção e busca em tempo constante.

Imagine que cada elemento em um conjunto possua um número associado a ele e que quaisquer dois elementos distintos possuam números associados diferentes. Desta forma, poderiamos armazenar os elementos em um array na posição indicada pelo número associado ao elemento.



Se conseguirmos associar a cada elemento a ser armazenado um número como no exemplo acima, poderemos realizar as operações de inserção, remoção e busca em tempo constante.

A função que associa a cada elemento de um conjunto U um número que sirva de índice em uma tabela (array) é chamada função hash.

Uma função hash deve satisfazer as seguintes condições:

- Ser simples de calcular.
- Assegurar que elementos distintos possuam índices distintos.
- Gerar uma distribuição equilibrada para os elementos dentro do array.

3.2 Tipos de Organização de Arquivos

Seguindo a analogia feita na seção anterior, vamos estudar os tipos existentes de organização de arquivos. No mundo dos bancos de dados relacionais, é possível estabelecer uma analogia “arquivo – tabela”, de forma que uma organização

específica de arquivo pode ser uma opção de implementação física de tabela em um banco de dados.

Atualmente, os SGBDs do mercado têm seus modos nativos de organização de arquivos, mas é importante conhecer a maneira como isso é feito.

3.2.1 Arquivos com Registros Não – Ordenados

- Arquivos são organizados como “heap” (estrutura de dados) ou pilha.
- Os registros são desordenados e se organizam como foram inseridos. Assim, novos registros são inseridos sempre no final do arquivo.
- MUITO EFICIENTE para inserção.
- MUITO CARO para a seleção de um determinado registro (precisa buscar em todos os registros para encontrar).
- Quando existem muitas remoções, de tempos em tempos o arquivo deve ser reorganizado.

3.2.2 Arquivos com Registros Ordenados

- São arquivos seqüenciais em que os registros ficam fisicamente ordenados dentro do arquivo. A ordenação é baseanda no valor de um (ou um conjunto) de seus atributos.
- Arquivos com registros ordenados são MUITO EFICIENTES para: (i) recuperar registros na ordem dos valores dos atributos de ordenação, (ii) achar o próximo registro de uma seqüência, (iii) usar um critério de busca baseado no valor do atributo de ordenação.
- Este tipo de organização, não traz NENHUMA VANTAGEM para: acesso aleatório (randômico), (ii) acesso ordenado baseado no valor de atributos que não os que compõem a chave de ordenação.
- Este tipo de organização acarreta um processamento CARO para INSERÇÃO e REMOÇÃO. Por que?

3.2.3 Arquivos HASH ou Diretos

- Um atributo é usado como chave de HASH a qual uma função é aplicada para cálculo do endereço do bloco de disco no qual o registro será armazenado.
- Este método é MUITO EFICIENTE para recuperação de um registro específico baseado no valor do atributo usado como chave de HASH.
- Este método é MUITO CARO para recuperar registros de maneira ordenada (mesmo que seja usada a chave de HASH) e para recuperar registros usando outros atributos que não os da chave de HASH.

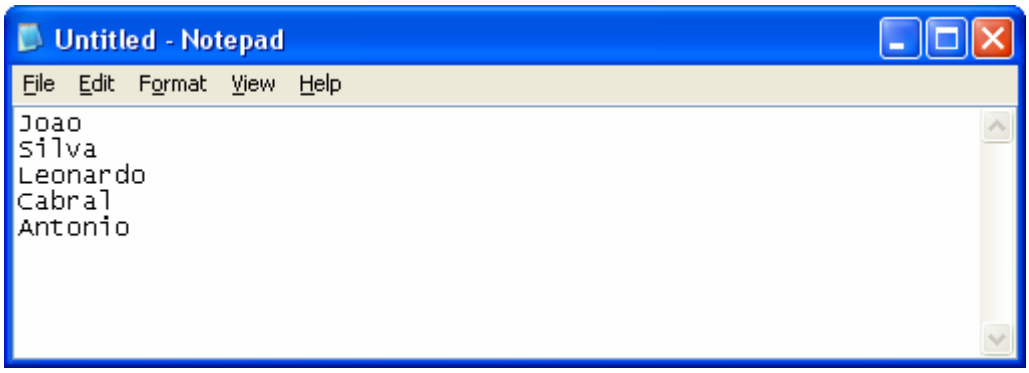
3.2.4 Arquivos com Índices

- Requerem estruturas de dados **ADICIONAIS** auxiliares além do arquivo contendo os registros de dados. Utilizam atributos indexadores (chaves de busca) para encontrar o endereço do registro desejado.

3.2.5 Como indexar uma tabela?

Como acessar diretamente uma tupla em uma tabela? Sem essa resposta, não é possível pensar em utilizar um método de indexação.

Imagine que o seguinte arquivo texto no bloco de notas seja um arquivo com registros não-ordenados:



Imaginando que este texto seja uma tabela com apenas um atributo e que cada linha represente uma tupla, podemos assumir que **usar o número da linha** seja uma forma de acesso direto a uma tupla. Assim, uma estrutura rudimentar de índice ordenado seria a seguinte:

Antonio	5
Cabral	4
João	1
Leonardo	3
Silva	2

Repare que este índice rudimentar é um índice ordenado (os nomes estão ordenados alfabeticamente).

Nos SGBDs, cada tupla existente em uma tabela apresenta um **identificador físico**, usado para acesso direto pelos índices. No Oracle, por exemplo, este identificador se chama “ROWID”.

Devemos entender que, quando disponível, este “campo virtual” é um controle interno do banco de dados e não deve ser utilizado dentro de aplicações.

Nome	ROWID
Joao	88755
Silva	29761
Leonardo	12828
Cabral	19976
Antonio	21987

A mesma estrutura rudimentar de índice acima, agora usa os identificadores físicos das tuplas. Para entendermos a indexação, somente temos que conhecer a existência destes identificadores físicos, e não a maneira como eles são criados / mantidos (temas de interesse para um curso de criação de sistemas de bancos de dados).

Antonio	21987

Cabral	19976
João	88755
Leonardo	12828
Silva	29761

3.3 Índices Ordenados

Para implementar um eficiente acesso aleatório (não necessariamente seqüencial) aos dados, estruturas de índice são utilizadas.

Um arquivo de dados ou uma tabela podem conter diversos índices, mas cada um deve apresentar uma **chave de busca** diferente. Por chave de busca, entenda-se a lista de campos que estarão sendo indexados.

Todos os índices são baseados nos mesmos conceitos básicos: chaves e campos de referência (ponteiros para posições físicas). Inicialmente vamos estudar índices simples, consistindo simplesmente de vetores com chaves e campos de referência. Mais tarde vamos estudar índices com estruturas mais complexas, especialmente árvores.

Uma das principais vantagens dos índices é poder manter a maior parte possível do índice em memória principal, que é MUITO MAIS RÁPIDA QUE O DISCO, local onde os dados realmente estão armazenados.

Nos índices ordenados, os valores da chave de busca encontram-se em uma ordem bem definida.

Denomina-se **ÍNDICE PRIMÁRIO** aquele que é especificado sobre o campo chave da ordenação de um arquivo com tipo de organização "**com Registros Ordenados**". O campo chave é usado para ordenar fisicamente os registros do arquivo no disco, e cada registro **TEM UM VALOR ÚNICO PARA ESTE CAMPO**. A chave de busca é normalmente, mas não necessariamente, a chave primária.

ÍNDICE SECUNDÁRIO é aquele especificado sobre qualquer campo de um arquivo que não o de ordenação física.

Como um arquivo só pode ter uma ordenação física possível, só é possível a existência de UM ÍNDICE PRIMÁRIO associado ao mesmo. Um arquivo pode ter vários índices secundários.

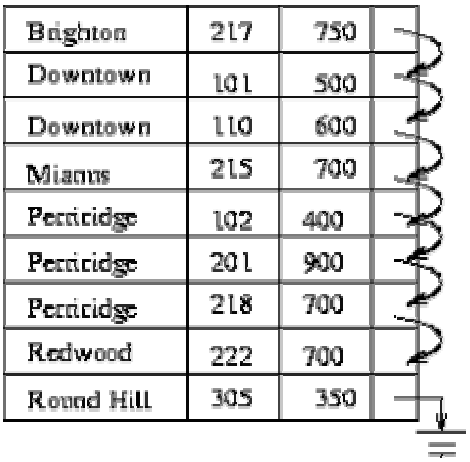


Figura 4 - Exemplo de Índice Primário

3.3.1 Índices Densos e Esparsos

Dentro do universo dos índices ordenados, encontramos uma classificação. Um índice ordenado pode ser denso ou esparso (não denso).

Em um índice **DENSO**:

- Cada valor distinto da chave tem a correspondente entrada no índice.
- Cada entrada no índice contém um valor da chave de busca e um ponteiro para o registro relacionado.
- Também chamado de COMPLETO.

Em um índice **ESPARSO**:

- Apenas **ALGUNS** dos valores distintos da chave têm uma entrada no índice.
- Para encontrar o registro com o valor "K" da chave: Encontra-se a entrada do índice com o maior valor MENOR QUE "K" (quase chegando). Partindo do registro apontado pela entrada do índice encontrada, percorre-se o arquivo seqüencialmente.
- Também chamado de PARCIAL.

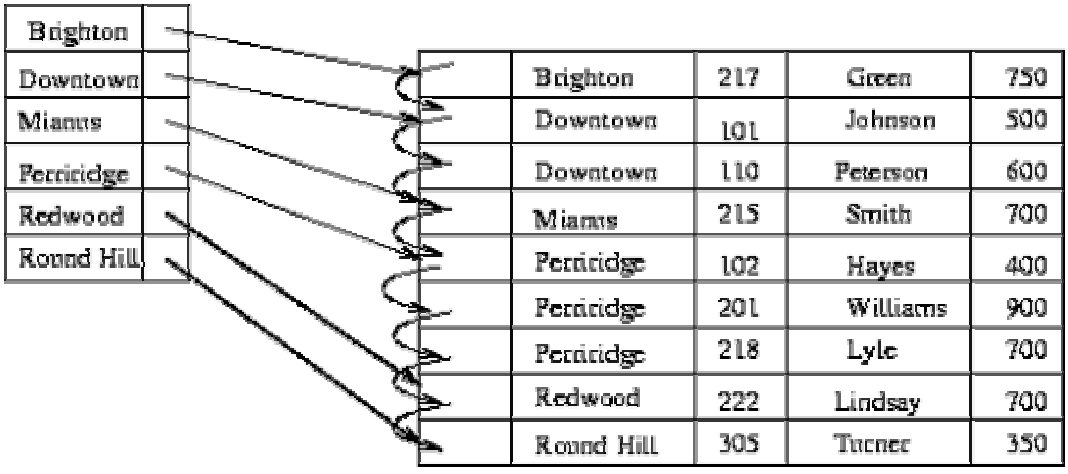


Figura 5 - Índice DENSO a esquerda

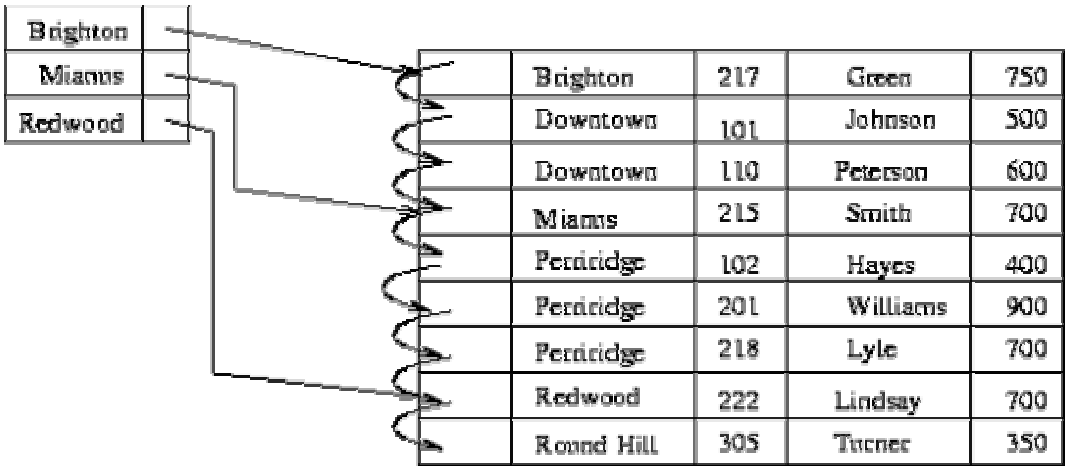


Figura 6 - Índice ESPARSO a esquerda

Quais seriam as vantagens de um índice não denso? Menos espaço de armazenamento, menos tempo na atualização e na remoção.

Índices densos tendem a ser mais rápidos para encontrar a informação.

3.3.2 Índices de múltiplos níveis

Em diversas situações, mesmo com a construção de um índice esparso, o tamanho da estrutura fica proibitivo, imagine o seguinte cenário:

1. O arquivo tem 100.000 registros com valores diferentes para o campo a ser indexado.
2. O bloco de disco tem capacidade para armazenar um bloco de índice com 10 registros. Assim, o índice terá 10.000 blocos, o que torna qualquer algoritmo que o percorra inevitavelmente lento. Lembre-se que entruturas de índice requerem que o número de acessos a disco seja baixo.

Nestas situações, a solução para minimizar o problema da indexação é a construção de um índice esparso SOBRE o índice padrão, de forma que este guie a busca pela chave indexada.

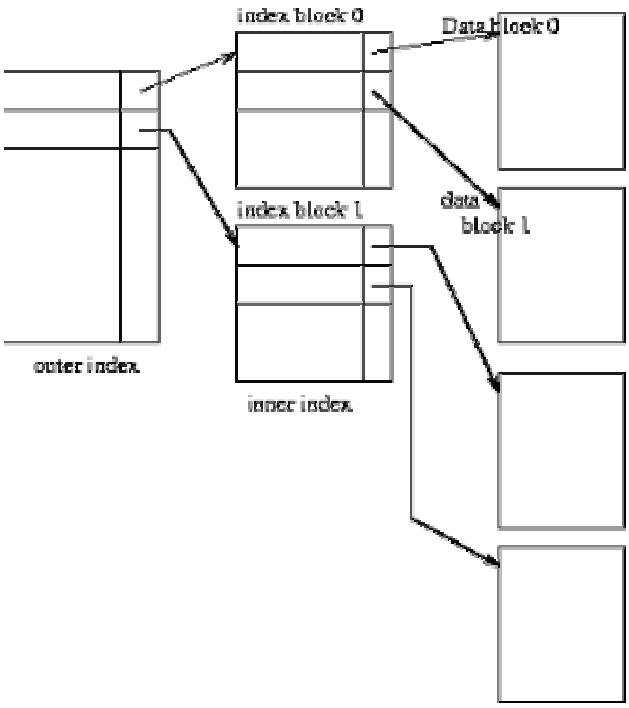


Figura 7- Índice Multi-Nivel com um nó de índice esparso

Uma busca binária pode ser feita no índice "outer" para encontrar o bloco de índice denso apropriado. Uma vez encontrado o bloco de índice denso correto, fazemos uma busca dentro deste e por fim encontramos a informação.

Quando o número de valores por indexar é muito alto, utilizamos mais de um nível (procedimento corriqueiro).

Como nada no mundo é perfeito, inclusões, atualizações e remoções nos dados devem ser refletidas em **TODOS OS NÍVEIS DO ÍNDICE**.

A idéia é que cada bloco de disco corresponda a um bloco de índice (eficiência na leitura).

3.3.3 Atualização de índices

Toda estrutura de índice (não importa seu tipo) deve ser atualizada ao longo que operações aconteçam no arquivo.

3.3.3.1 Remoção

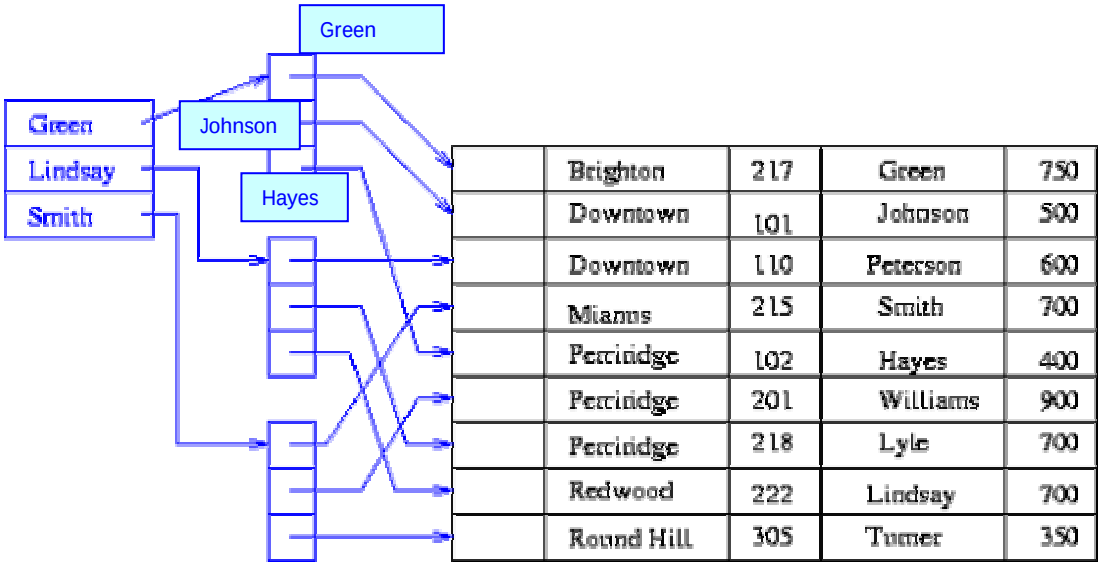
- Encontre o registro (faça uma busca)
- Se este é o último registro com a referida chave de busca, delete a entrada do índice.
    - Para índices densos → este procedimento é como apagar um registro do arquivo.
    - Para índices esparsos → substitua o valor a ser removido pelo próximo valor possível (obedecendo a ordem). Caso esta operação não seja possível, apague a entrada do índice esparso.

3.3.3.2 Inclusão

- Encontre o bloco para inserção (faça uma busca)
  - Para índices densos → caso o valor de chave não exista, insira o valor de chave e atualize as referências.
  - Para índices esparsos → caso a inclusão acarrete na criação de um novo bloco de índice correspondente, então crie uma nova entrada de índice esparso e atualize as referências, caso contrário, nada é feito.

3.3.4 Índices Secundários

Conforme exposto acima, os índices secundários são aqueles especificados sobre qualquer campo de um arquivo que não o de ordenação física. Assim, para construir um índice secundário, é necessário armazenarmos ponteiros para todos os registros do arquivo (deve ser DENSO).



Um índice secundário melhora as consultas sobre atributos que não compõem a chave primária de uma tabela.

Como já visto, esta classe de índices deve ser atualizada a cada operação no arquivo correspondente, assim, é necessário usar bom senso para decidir se um índice será útil no dia a dia do banco de dados (melhora a leitura penalizando as inclusões e remoções).

3.4 Índices em Árvore

Na medida em que um arquivo cresce em demasia, as estruturas de índice estudadas até o momento têm seu desempenho degradado (marcas de remoção, movimentação de blocos etc). O desempenho volta a ficar razoável quando as estruturas de índice são re-organizadas.

Árvores-B (Bayer&MCCreight 1972) às vezes também chamadas Árvores-Bayer ou Árvores Multivias, foram originalmente concebidas para a implementação de mecanismos de indexação por chave primária em memória secundária (DISCO).

Permite um número menor de nós (menor altura) e, por conseguinte menos acessos a disco. Implementações comuns utilizam arquivo de índices com ponteiros para arquivo de registros de dados.

Árvore binária de busca.

3.4.1 Árvores B

A árvore B, utilizando o recurso de manter mais de uma chave em cada nó da estrutura, proporciona uma organização de ponteiros tal que as operações de leitura, inclusão e remoção são executadas rapidamente.

Sua construção assegura que as folhas se encontram todas em um mesmo nível, **não importando a ordem de entrada de dados**.

As árvores B são largamente utilizadas como forma de armazenamento em memória secundária. Diversos sistemas comerciais de bancos de dados, por exemplo, as empregam.

Seja *d* um número natural. Uma *árvore B de ordem d* é uma árvore ordenada que é vazia, ou que satisfaz as seguintes condições:

1. a raiz e uma folha ou tem no mínimo dois filhos;
2. cada nó interno (não raiz e não folha) possui no mínimo *d+1* filhos;
3. cada nó tem no máximo *2d + 1* filhos;
4. todas as folhas estão no mesmo nível.

Em uma árvore B, cada nó é denominado página. A estrutura apresentada satisfaz ainda as seguintes propriedades:

a. Seja *m* o número de chaves em uma página P não folha. P tem *m+1* filhos. Conseqüentemente, cada página contém entre *d* e *2d* chaves, com exceção da raizo que possui entre *1 e 2d* chaves.

b. Em cada página P, as chaves estão ORDENADAS.

c. Os ponteiros de cada página são organizados como uma árvore de busca, em que os intervalos de valores são respeitados.

3.4.1.1 Buscas

A busca em uma árvore B é semelhante a empregada em árvores binárias de busca.

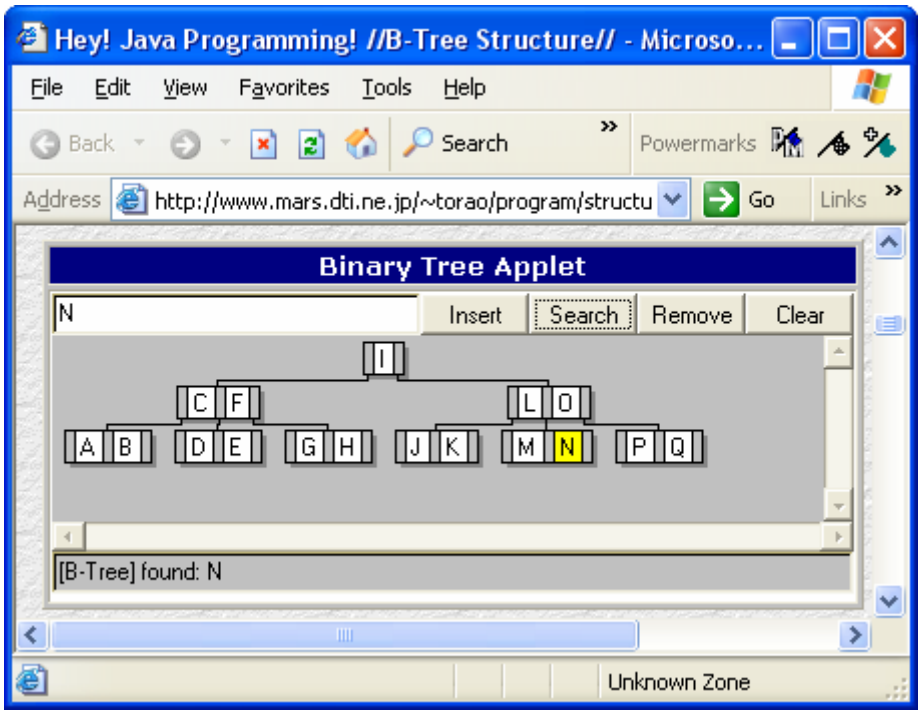


Figura 8 - Árvore B de ordem 2

Observe, na figura 8, a busca da chave “N” na árvore B de ordem 2. De início, a chave procurada (“N”) é comparada com a única chave existente na raiz (“I”). Como “N” é maior que “I”, então a busca segue recursivamente para a próxima página (que contém as chaves maiores do que “I” – “L” e “O”).

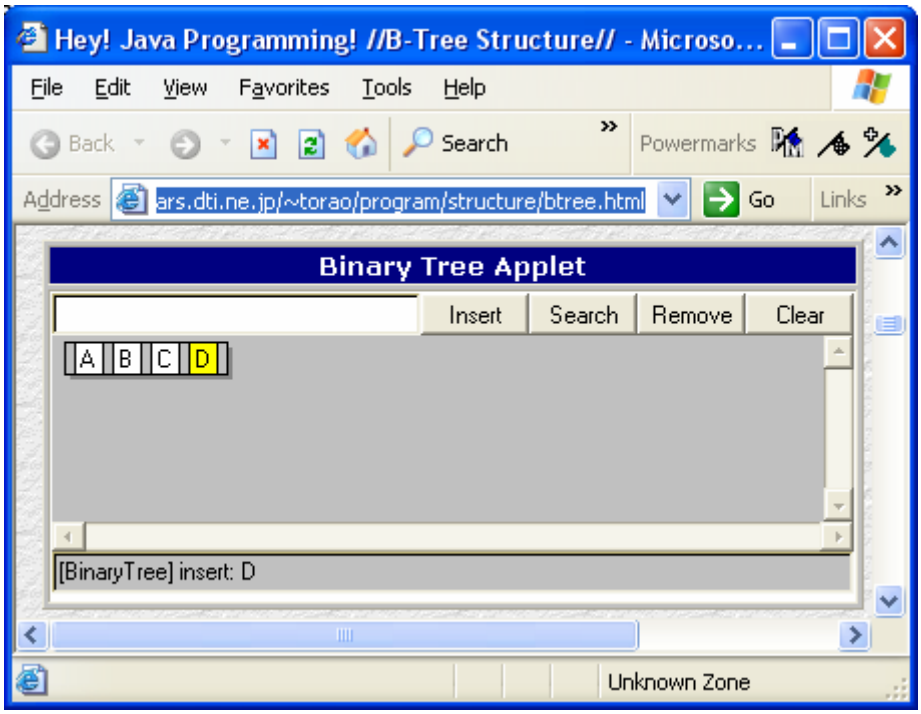
A operação é repetida para a página filha. Na próxima página do caminho, deve-se procurar a posição adequada, testando-se o valor “N” com todas as chaves armazenadas no nó até que seja encontrado o ponteiro conveniente para o prosseguimento da busca (ou a chave ser encontrada). Neste exemplo, a busca ainda não encontra o valor “N” e passa para a terceira página, segundo o ponteiro que demarca as chaves maiores do que “L” e menores do que “O”.

Finalmente, na terceira leitura de página do disco, a chave “N” é encontrada na folha.

3.4.1.2 Inserção de Dados

Considere agora o problema de inserir uma nova chave X em uma árvore B. **Uma NOVA CHAVE, SEMPRE É INSERIDA EM UMA PÁGINA FOLHA.** O primeiro passo consiste em executar uma busca para encontrar a posição (folha) onde “X” deveria estar. Vamos construir passo a passo a árvore da figura 8, inserindo as letras do alfabeto em ordem. Lembre-se que estamos trabalhando com ordem 2 (cada nó interno tem no máximo (2d+1) = 5 ponteiros).

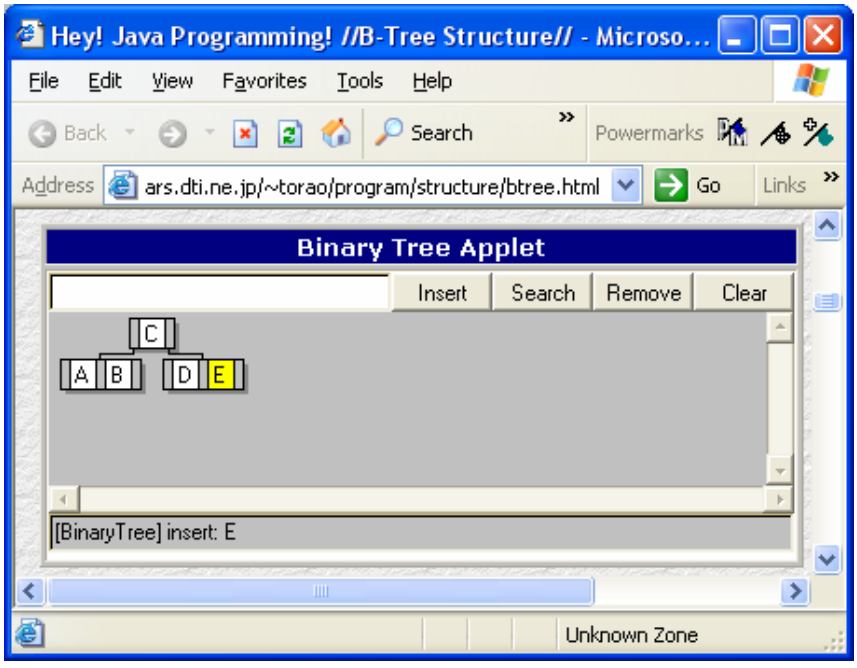
No caso trivial, inserimos “A”, “B”, “C” e “D”. A árvore fica com a seguinte estrutura.



Ao inserir a chave “E”, a regra de tamanho de um nó estará sendo violada, pois o nó já apresenta 2d (4) chaves antes da inserção. A solução para este problema é regorganizar as páginas, processo conhecido por cisão de página.

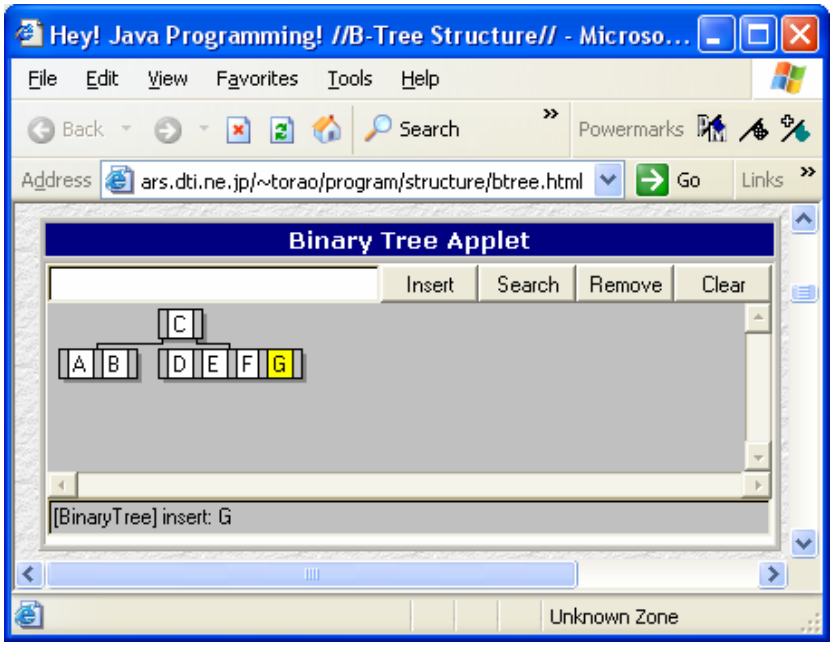
A cisão da página, executada em P, transforma uma página com excesso de chaves em **DUAS**. Em P permanecem as d primeiras entradas. As últimas chaves dentre as d+1 restantes serão alocadas em uma nova página. A chave “central” será alocada no nível acima da árvore.

Neste exemplo, a página P vai conter as chaves “A” e “B”. Teremos um irmão criado para P que vai conter “D” e “E” (o novo valor). E o valor central passa para o nível acima “C”, criando uma nova raiz para a árvore. Vide figura abaixo.

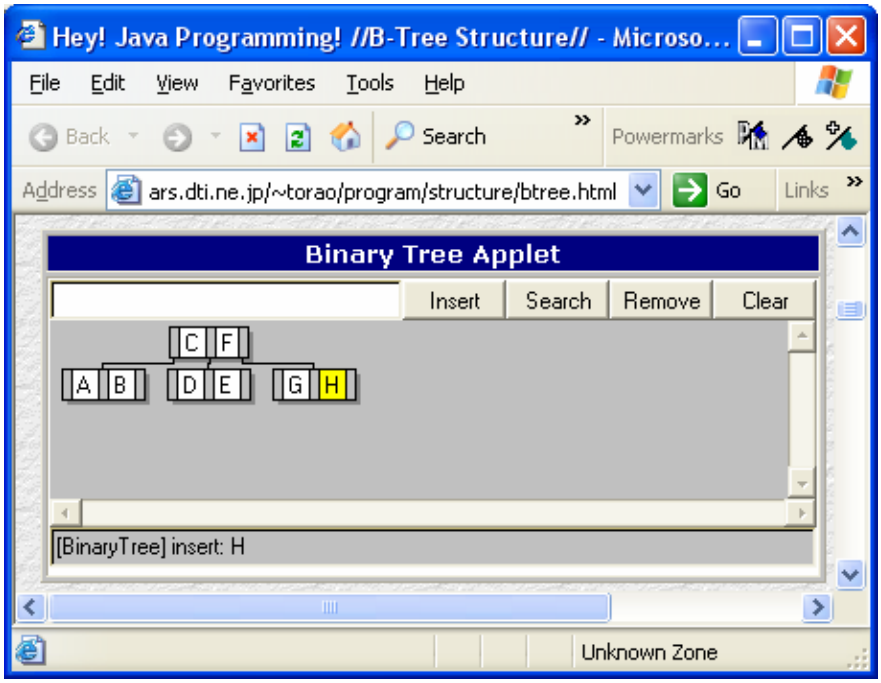


Repare que agora temos mais espaço para a inclusão de novas chaves nas folhas da arvore.

Continuando com a construção da árvore, agora inserimos “F” e “G”, para quase completarmos novamente a página. Estas novas inclusões desconhecem a existência das chaves “A” e “B”, pois elas estão no outro ramo da árvore.



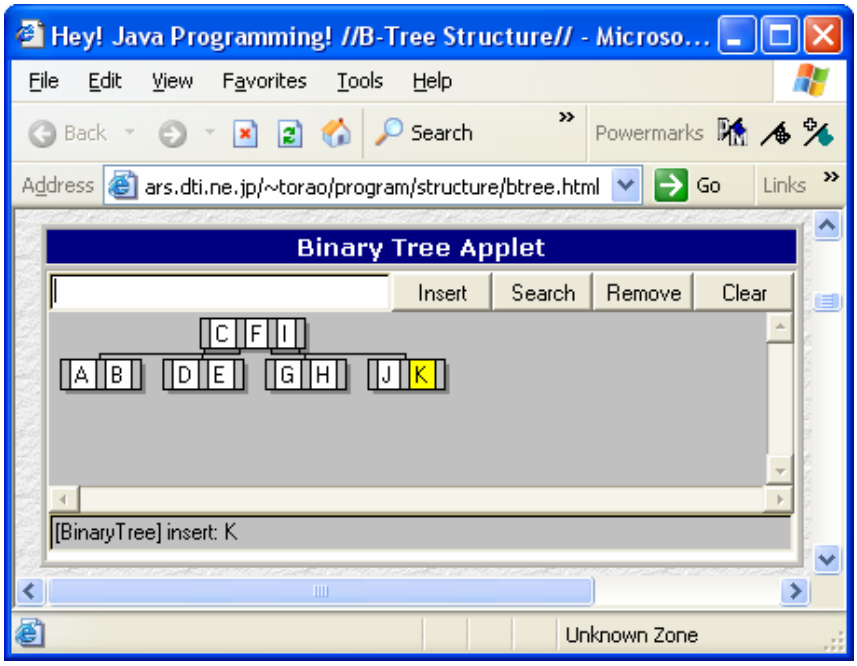
Quando “H” for inserida, uma nova cisão de páginas vai ocorrer:



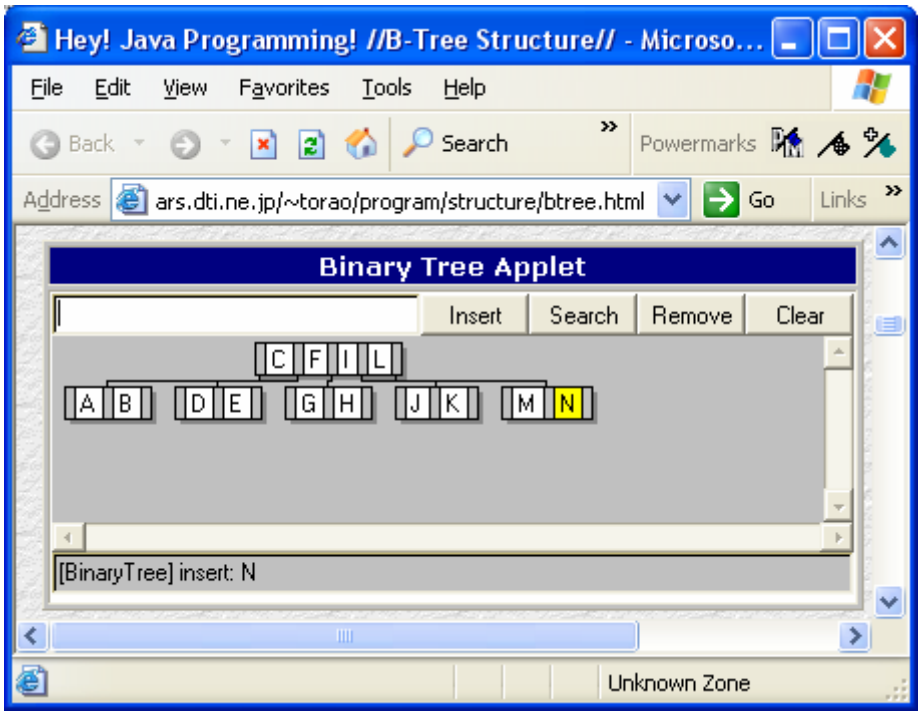
O nó cheio apresentava “D”, “E”, “F”, “G” e “H”. Conforme a regra geral da árvore B, as d (2 neste exemplo) primeiras (D e E) permanecem na página. As d últimas (G e H) vão para uma nova página e o termo central (F) sobe de nível,

compondo com C a raiz da árvore. Repare que na primeira cisão de páginas a raiz foi modificada enquanto que nesta, a altura da árvore não se modificou.

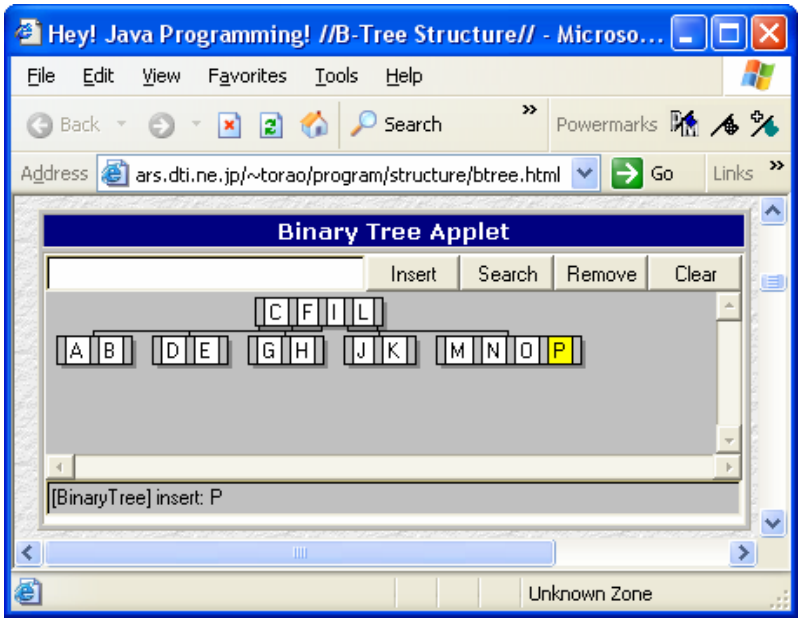
Com a continuidade das inclusões neste mesmo raciocínio, quando inserirmos a chave “K”, temos a seguinte posição após a cisão:



Da mesma forma, quando inserirmos “N”, a árvore B de ordem 2 toma o seguinte formato:

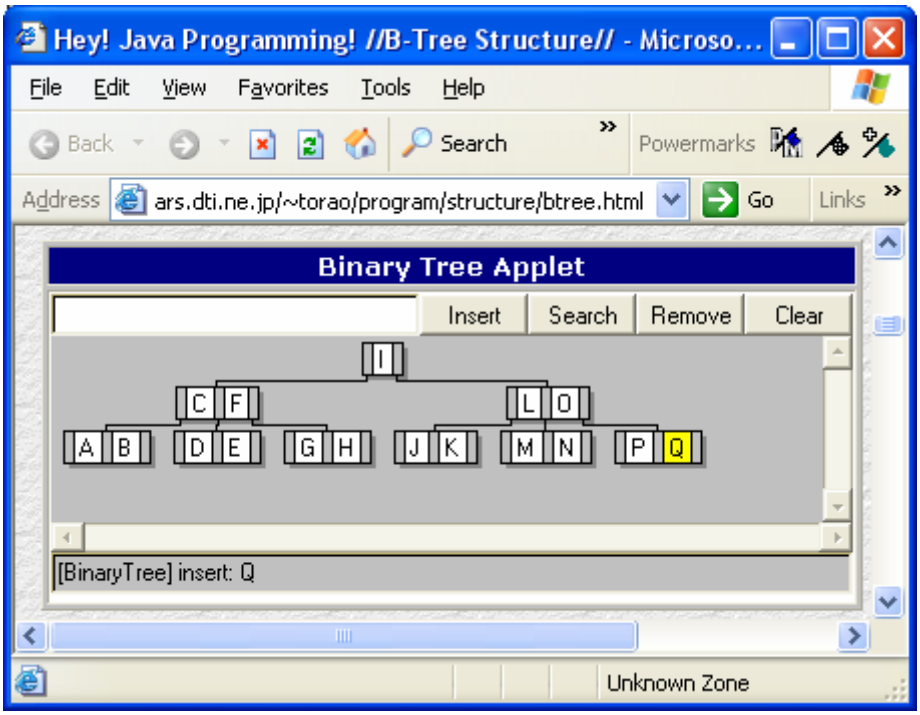


Chegando ao ponto mais interessante, podemos ver que, após a inserção de “O” e “P” a árvore vai ficar como a figura abaixo:



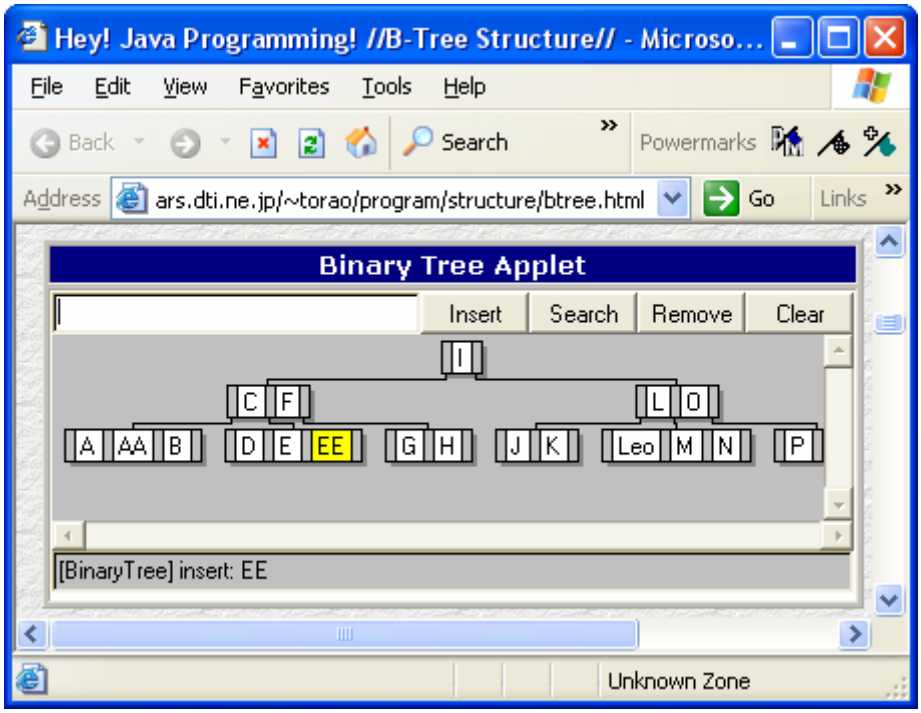
No momento de inserir “Q”, aplicando o algoritmo de cisão, teremos uma página com “M” e “N”, outra com “P e “Q” e o valor “O” sobe de nível, chegando até a raiz. **MAS NESTE MOMENTO, a raiz também está completa!!**

Com a raiz completa, não há outra alternativa senão a cisão propagada de página. A propagação da cisão é automática e funciona sempre que o nó pai está cheio. Assim, a raiz cheia (composta de C, F, I, L e O) sofrerá uma cisão, gerando dois irmãos (C, F) e (L, O) e uma nova raiz (I).

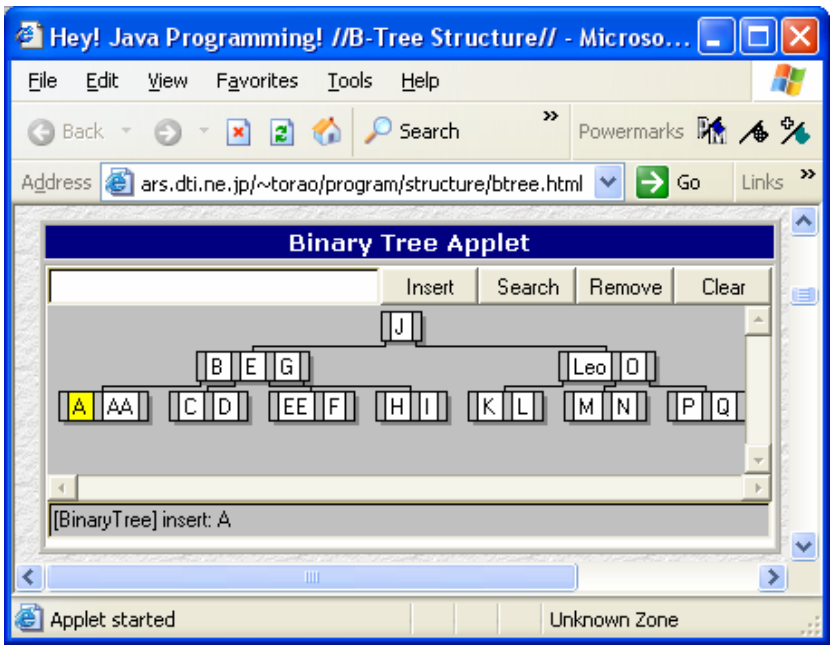


É interessante notar que o resto da árvore B não sofre modificações. Somente o caminho da folha até a raiz pode sofrer cisão de páginas.

Note também que existe espaço para inclusão de novas chaves sem a necessidade de novas cisões. Vamos inserir “AA”, “Leo”, “EE” etc.



Esta árvore foi construída com a inserção em ordem. Vamos reconstruí-la com a inserção na ordem inversa, ou seja, “EE, “Leo”, “AA”, “Q”, “P”, ... , “B”, “A”.



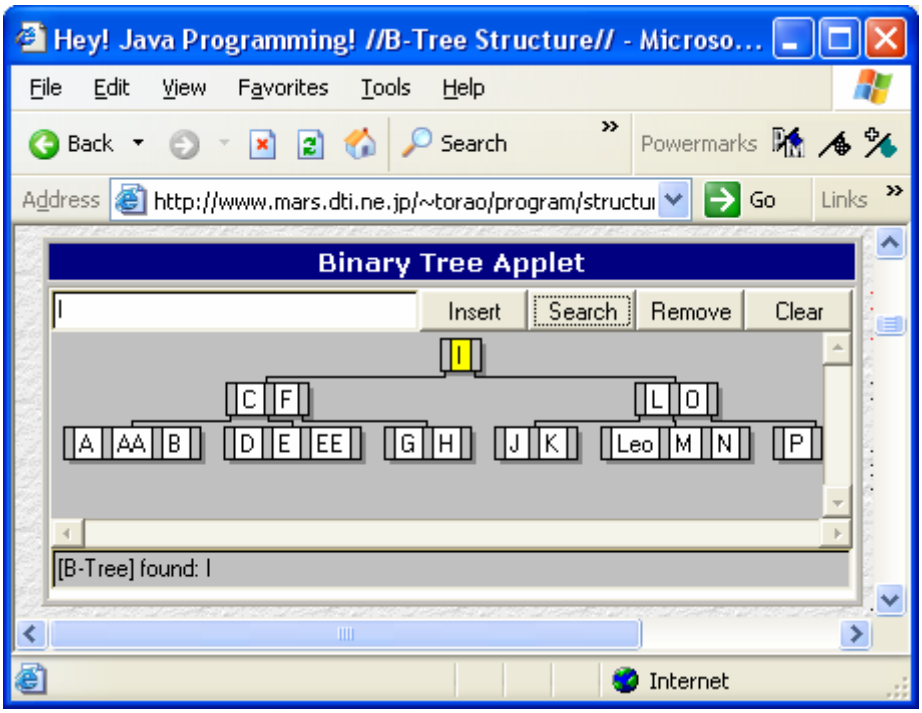
A estrutura é um pouco diferente, mas apresenta a mesma altura, o que garante um bom desempenho nas buscas.

3.4.1.3 Remoção de Dados

Já vimos que uma árvore B se auto-ajusta na medida que novos nós são inseridos nas páginas. O processo de cisão de páginas garante que a árvore está coerente com a definição formal em todos os momentos de sua existência.

Assim como a inserção pode tornar um nó cheio demais (com mais de *2d* chaves), a remoção pode torná-lo vazio demais (subutilizado) (com menos de *d* chaves). Em certas situações, a remoção de dados poderá ocasionar na reorganização da árvore B, conforme veremos adiante.

Considere a árvore B de ordem d=2 construída abaixo:



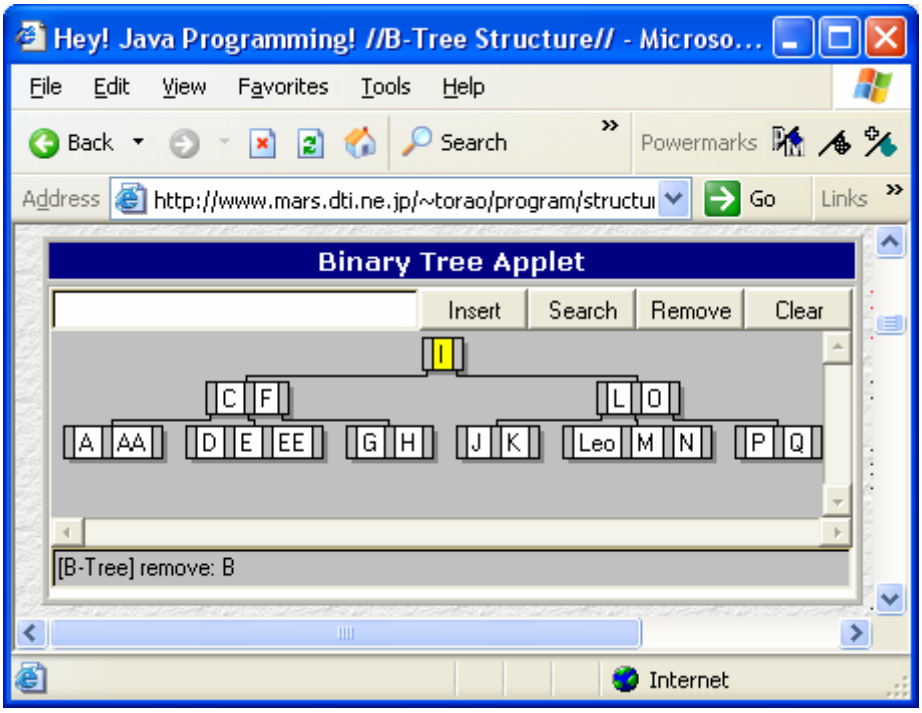
O algoritmo BÁSICO da remoção de nós é o seguinte:

- Se a chave “x” a ser removida estiver em uma folha, a chave é simplesmente removida.
- Se a chave “x” NÃO se encontra em uma folha, o espaço de “x” no nó interno deve ser preenchido pelo valor imediatamente maior na árvore, digamos “y”. A chave “y” NECESSARIAMENTE deve existir em uma folha.

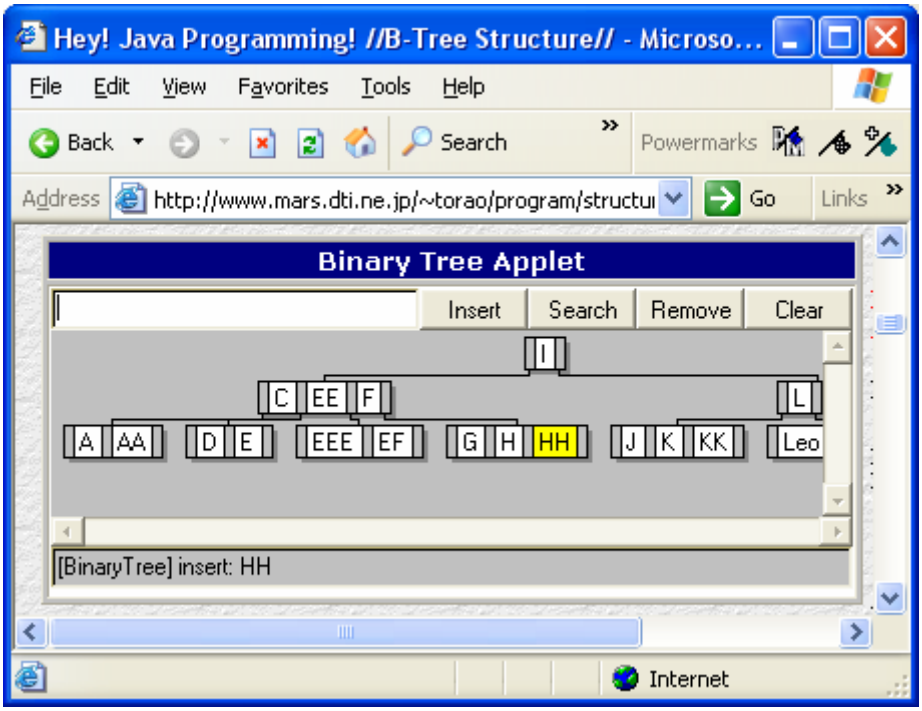
Com a segunda regra acima bem definida, **o problema da remoção pode estar restrito ao caso em que a informação é removida de uma folha**. Imagine, na árvore acima, que é necessário remover a chave “EE”. Como ela existe em uma folha, o procedimento é imediato. Caso a chave a ser removida seja “F”, que se encontra em um nó interno, a chave “G” (imediatamente maior) tomará o lugar de “F” no nó interno e bastará apagarmos “G” para completarmos o processo (não podemos ter duas chaves “G” neste exemplo).

Vamos apagar a chave “B” da árvore acima.

- “B” existe em uma folha → não existe necessidade de mover chaves.
- Apagamos “B”.
- A página folha ainda tem no mínimo d=2 chaves.



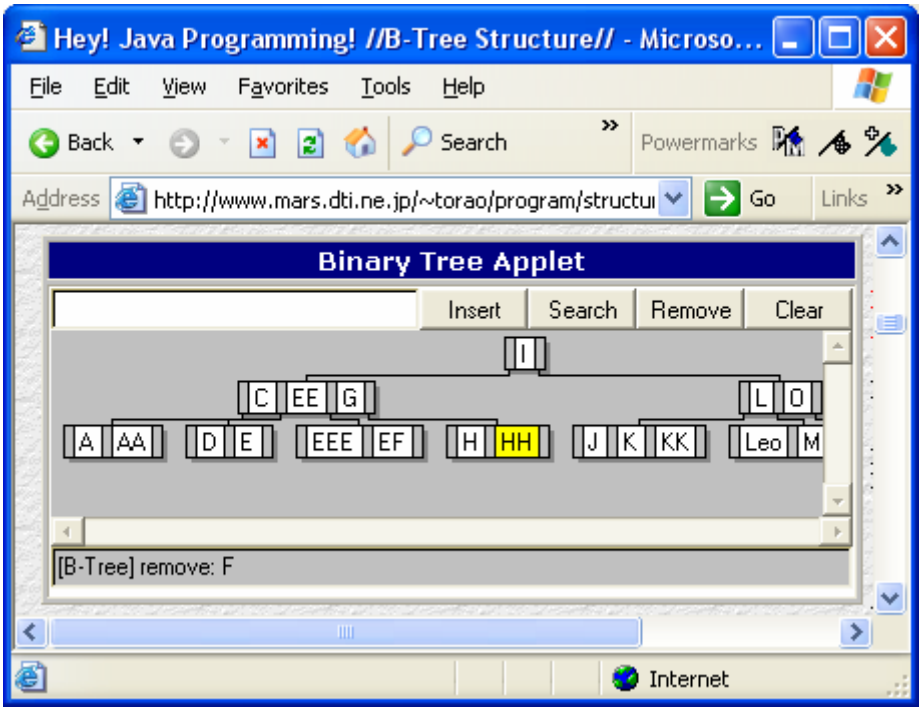
Para passarmos para o caso mais complicado, imagine que houve novas inserções de chaves, e que a árvore está como abaixo:



Vamos apagar a chave “F” da árvore acima.

- “F” é um nó interno → procuramos a maior chave maior que “F” que exista em folha. Encontramos “G”.
- Colocamos “G” no lugar de “F”. (Com isso já apagamos “F”)
- Apagamos “G” da folha.

Observe o resultado (“G” está no nó interno):



Prosseguindo com as remoções, vamos apagar a chave “D”.

- “D” é chave existente em folha.
- Apagamos “D”.
- VIOLAMOS A REGRA DA ÁRVORE B. A página fica com 1 chave somente (“E”).

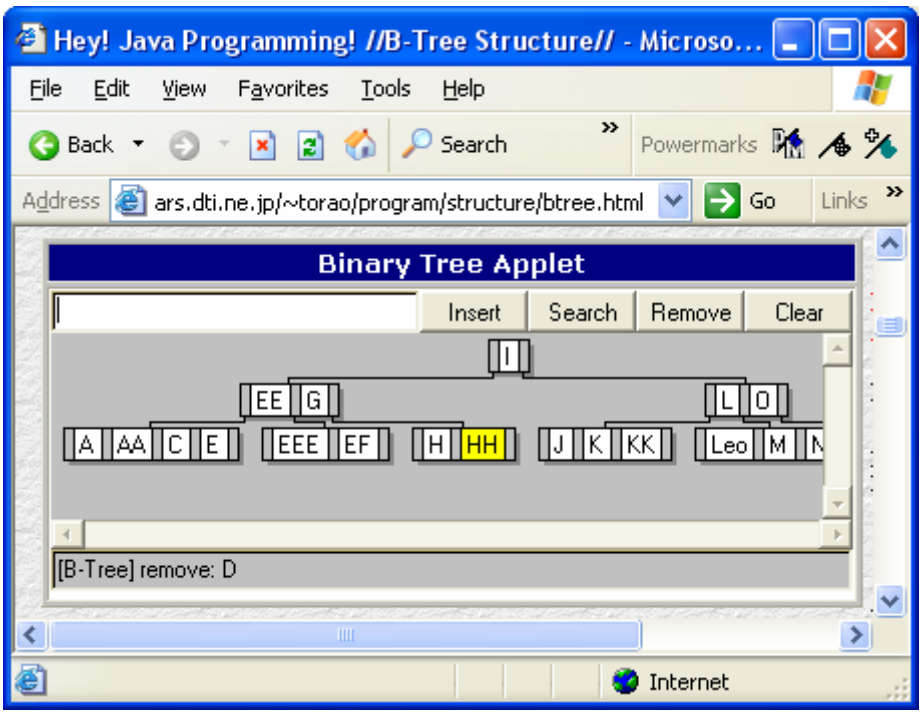
Nesta situação, duas operações são possíveis: **concatenação** OU **redistribuição** de chaves. Tenha em mente que estas operações devem re-colocar a árvore B correta perante a sua definição.

Duas páginas são consideradas *adjacentes* quando possuem o mesmo pai (são irmãs) e os ponteiros que as ligam ao pai são consecutivos. Assim, a página que ficou somente com a chave “E” tem duas páginas *adjacentes* (a página [“A” , “AA”] e a página [“EEE”, “EF”].

Quando a página afetada pela remoção (P) tem uma página adjacente (Q) tal que o total de chaves em P adicionado ao total de chaves em Q NÃO ULTRAPASSA 2d chaves, as páginas P e Q são concatenadas.

No nosso exemplo a página P só contém a chave “E”, e a página adjacente Q contém as chaves [“A”, “AA”]. A concatenação de páginas leva a seguinte configuração [“A”, “AA”, “E”]. Para a ávore ficar correta, falta descermos uma chave de um nó interno. A chave escolhida no nó interno para “descer” é a chave que fica entre as páginas adjacentes (“C”) neste exemplo.

A nossa árvore B fica com a seguinte configuração agora.



Continuando, imagine que agora queremos remover a chave “C”. A remoção é imediata (simples – construa a figura!).

Nova situação surge quando vamos remover “EEE”. A página P, afetada pela remoção vai conter só uma chave “EF” e sua página adjacente a esquerda vai conter três [A, AA, E]. O total de chaves das páginas adjacentes NÃO É MENOR QUE 2d. Neste exemplo, o total de chaves é exatamente igual a 2d (4) e não existe vantagem em concatenar nós, pois não poderemos descer com a chave do meio (“EE”).

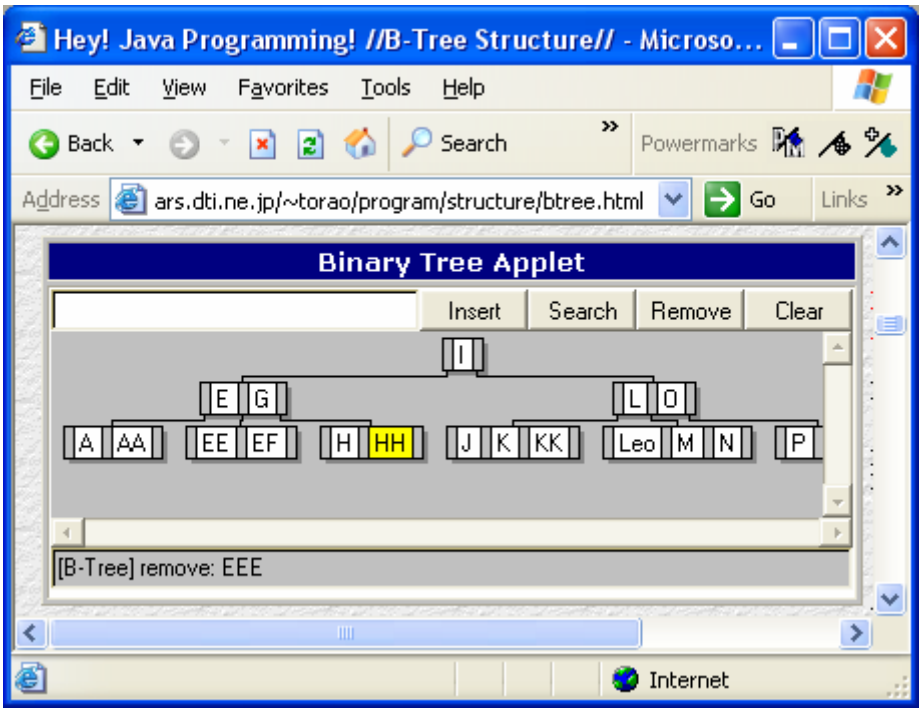
Nestas situações, surge a REDISTRIBUIÇÃO. Neste processo, imagine que uma concatenação foi efetuada:

- Criou-se uma página com as chaves [A, AA, E, EE, F].
- A, AA, E pertenciam a página adjacente.
- F pertencia a página que sofreu a remoção.
- EE foi a chave do nó interno que “desceu”.

A página do primeiro tópico consiste de cinco chaves (violando a regra). Então faça a cisão, ou seja, d primeiros para um lado, d últimos para o outro e a central pra cima. Assim, mantemos duas páginas [A, AA] e [EE, F] e subimos a nova chave central “E”.

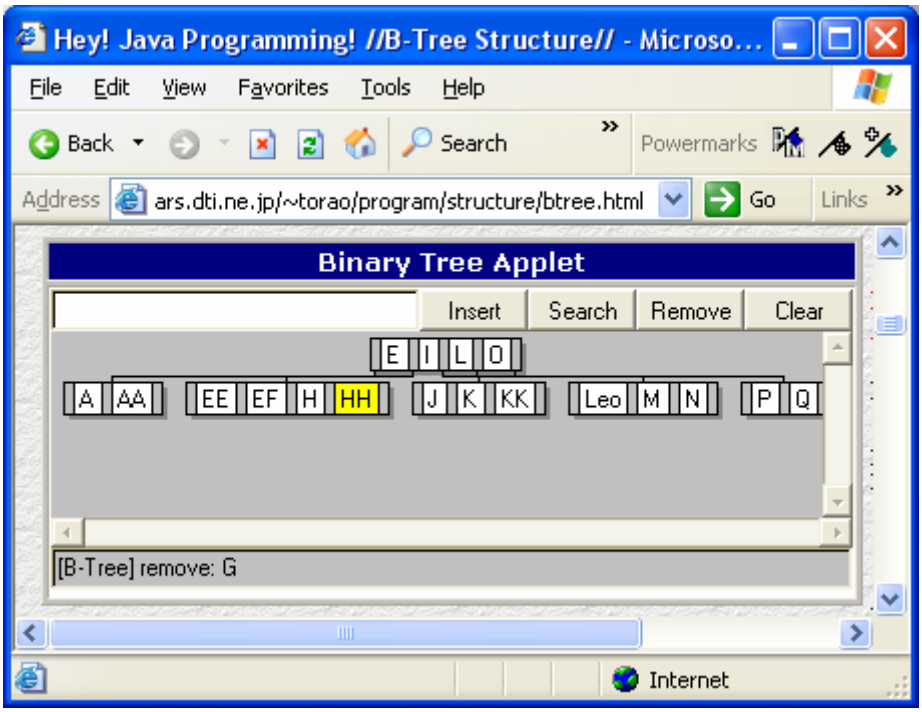
A redistribuição de chaves é uma concatenação seguida de uma cisão de páginas.

Temos a árvore:



Finalmente, como a cisão, a concatenação de chaves é propagável para níveis superiores da árvore, gerando diminuição de altura caso a raiz seja atingida.

Por exemplo, se apagamos “G”, “H” toma o seu lugar no nó interno, mas a folha fica com um número menor de chaves que o permitido. Assim, a página vai sofrer uma concatenação com a sua adjacente a esquerda, resultando que a chave “G” vai “descer”, deixando o nó interno com uma chave somente “E”. O nó interno vai propagar a concatenação para o seu adjacente e a raiz da árvore descerá de nível.



Literatura complementar:

“Estruturas de Dados e Seus Algoritmos” 2ª Edição

Jayme Luiz Szwarcfiter

“File Organization and Processing” ← maravilhoso livro

Alan L. Tharp

Nota: a árvore B que usei nestas notas de aula encontra-se em http://www.mars.dti.ne.jp/~torao/program/structure/btree.html

Outros programas legais (incluindo árvore B – bTree)

http://www.dcc.unicamp.br/~rezende/ASTRAL/

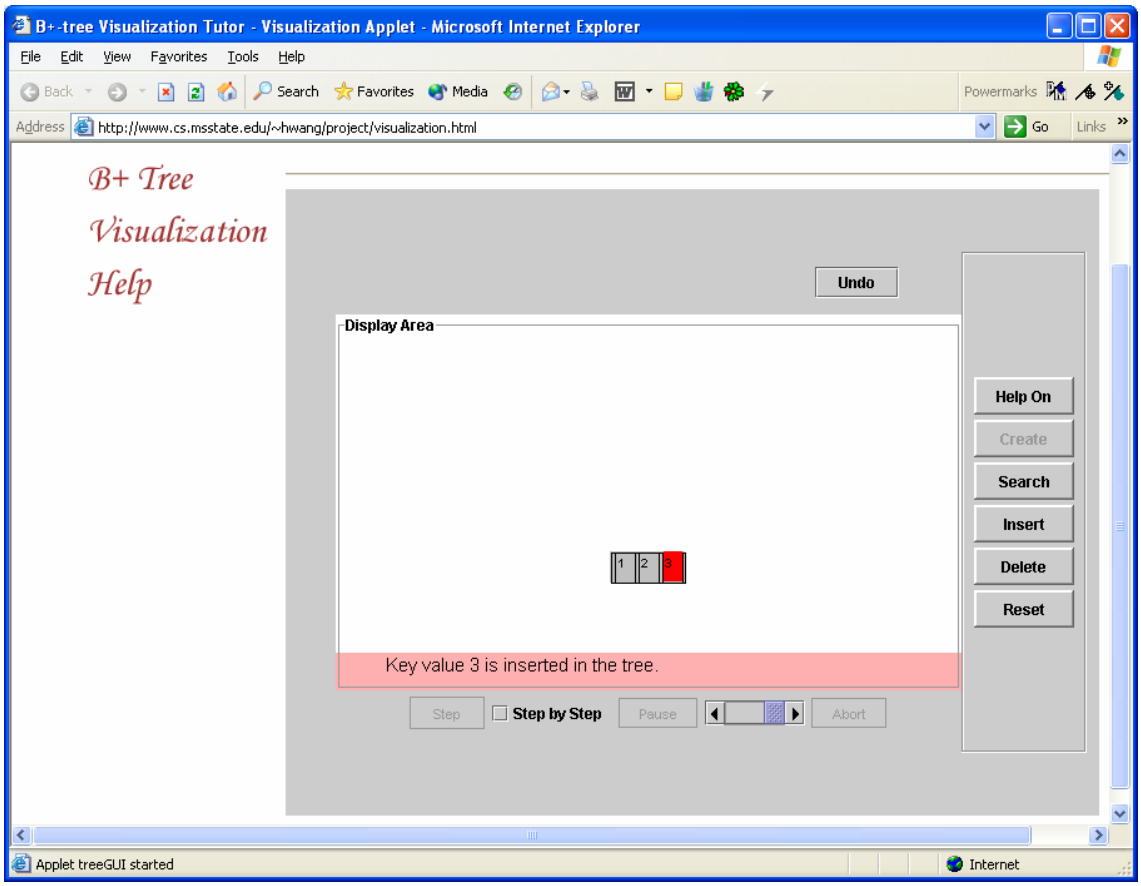
Que tal treinar?

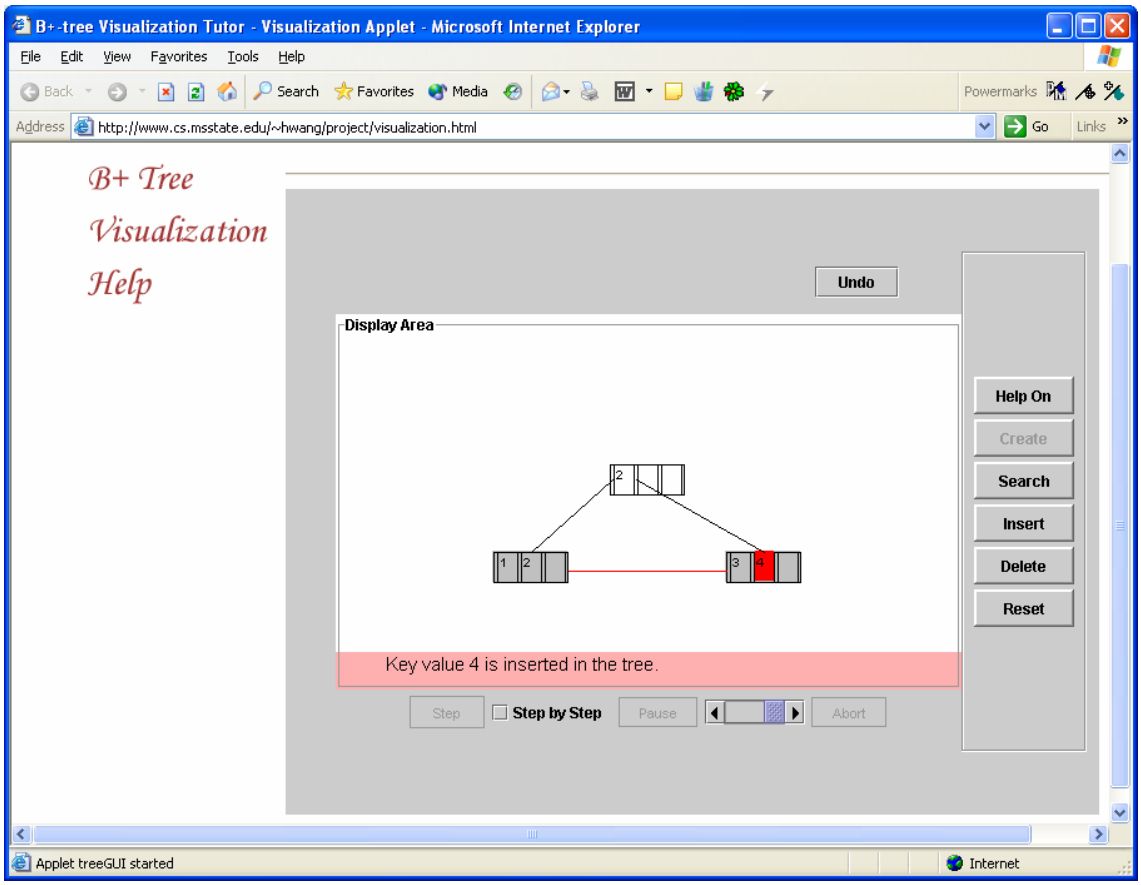
3.4.2 Árvores B+

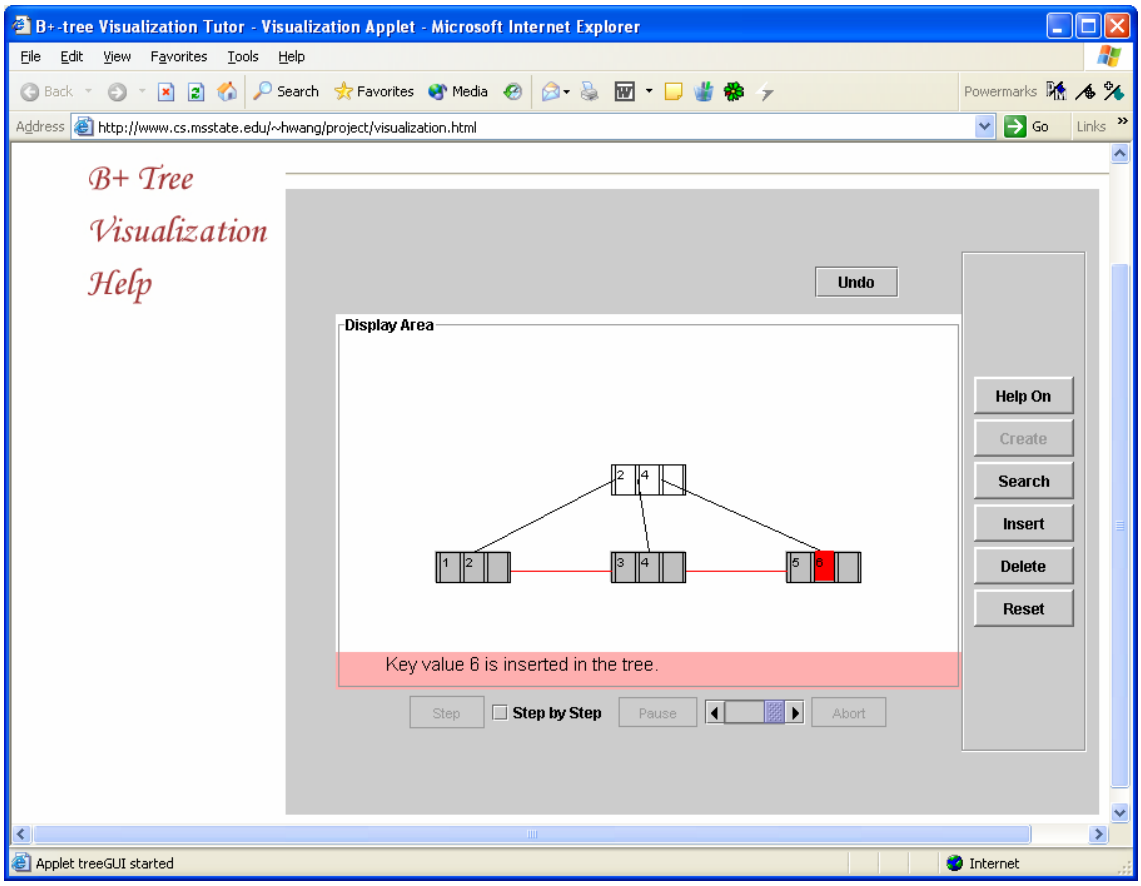
Embora revolucionária a árvore B ainda sofre com alguns problemas quando existe a necessidade de acesso seqüencial-indexado. Imagine que vc precisa encontrar as chaves maiores do que “H”. Na árvore acima, em dois acessos a disco, vc a encontra. Mas onde estão as próximas? Na verdade estão espalhadas entre nós internos e folhas.

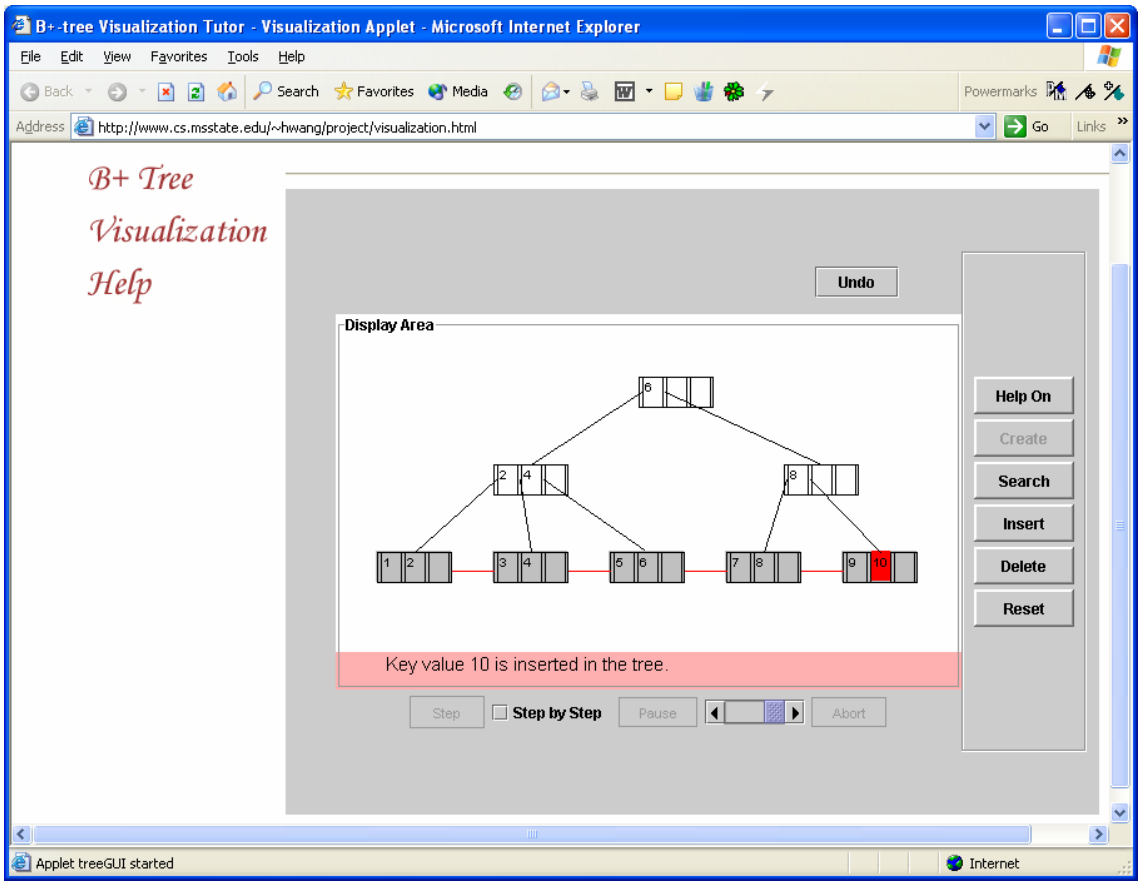
As árvores B+ são uma variante das árvores B em que **TODAS AS CHAVES ENCONTRAM-SE NAS FOLHAS**. Os nós internos apenas **REPLICAM** chaves para guiar uma busca até a chave existente na folha. **TODAS AS FOLHAS SÃO INTERLIGADAS POR PONTEIROS (sequence-sets) (**o que corrige o "problema" da busca seqüencial a partir de determinada chave).

Observe a árvore B+ com no máximo 3 chaves por nó:




B+-tree Visualization Tutor - Visualization Applet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.cs.msstate.edu/~hwang/project/visualization.html
B+ Tree
Visualization
Help
Undo
Display Area
Help On
Create
Search
Insert
Delete
Reset
Key value 4 is inserted in the tree.
Step
Step by Step
Pause
Abort
Applet treeGUI started
Internet


B+-tree Visualization Tutor - Visualization Applet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.cs.msstate.edu/~hwang/project/visualization.html
B+ Tree
Visualization
Help
Undo
Display Area
Help On
Create
Search
Insert
Delete
Reset
Key value 6 is inserted in the tree.
Step
Step by Step
Pause
Abort
Applet treeGUI started
Internet



Os nós internos da árvore B+ se comportam como os da árvore B.

Animação disponível em: http://www.cs.msstate.edu/~hwang/project/visualization.html

3.5 Boas Práticas para Indexação

The attributes whose values are required in equality or range conditions (selection operation) and those that are keys or that participate in join conditions (join operation) require access paths. Remember SELECTIVITY.

The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins. On the other hand, during insert, delete, or update operations, existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions.

The physical design decisions for indexing fall into the following categories:

1. *Whether to index an attribute:* The attribute must be a key, or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join. One factor in favor of setting up many indexes is

that some queries can be processed by just scanning the indexes without retrieving any data.

2. *What attribute or attributes to index on:* An index can be constructed on one or multiple attributes. If multiple attributes from one relation are involved together in several queries, (for example, `(garment_style_#, color)` in a garment inventory database), a multiattribute index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For example, the above index assumes that queries would be based on an ordering of colors within a `garment_style_#` rather than vice versa.
3. *Whether to set up a clustered index:* At most one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. In most RDBMSs, this is specified by the keyword CLUSTER. (If the attribute is a key, a primary index is created, whereas a clustering index is created if the attribute is not a key.) If a table requires several indexes, the decision about which one should be a clustered index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering. If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on. If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should *not* be clustered, since the main benefit of clustering is achieved when retrieving the records themselves.
4. *Whether to use a hash index over a tree index:* In general, RDBMSs use B^+-trees for indexing. However, ISAM and hash indexes are also provided in some systems. B^+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s).
5. *Whether to use dynamic hashing for the file:* For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes discussed later would be suitable. Currently, they are not offered by most commercial RDBMSs.

Capítulo 4 - Processamento de Transações

4.1 Conceito de Transação

Uma transação é uma <u>unidade lógica</u> de processamento do banco de dados, que inclui uma ou mais operações de acesso ao banco de dados. Estas operações podem incluir inclusões, remoções, alterações e leituras. As operações do banco de dados que formam uma transação podem ser embutidas dentro de um programa de aplicação ou podem ser especificadas de maneira interativa através de uma linguagem de consulta, como o SQL, por exemplo.

Uma transação é iniciada com um comando "iniciar a transação" (*begin transaction* ou *begin work*) e terminada com um comando "terminar a transação" (*end transaction* ou *commit transaction*). As operações que formam uma transação podem ser desfeitas (não confirmadas) se o comando "desfazer a transação" (*rollback transaction*) for executado.

Para entendimento dos conceitos de transação, vamos imaginar que o banco de dados seja formado por um conjunto de "**itens de dados nomeados**". O tamanho de um item de dados se chama "**granularidade**" e pode ser o atributo de uma tupla, a tupla em si, um bloco de disco com diversas tuplas, e até mesmo uma tabela inteira. Os conceitos relacionados ao processamento de transações são independentes da granularidade dos itens de dados analisados.

Seguindo esta analogia, é possível estabelecer dois tipos de operações básicas no banco de dados:

Read_item(X) → lê o valor do item do banco de dados "X"

Write_item(X) → escreve o valor do item do banco de dados "X"

A Figura 9 abaixo, representa duas transações simplificadas: T1 e T2.

T1	T2

read_item(X);
X:=X-N;
write_item(X);
read_item(Y);
Y:=Y+N;
write_item(Y);
	read_item(X);
	X:=X+M;
	write_item(X);

Figura 9 - Transações T1 e T2

O "**conjunto de leitura**" (read-set) de uma transação é o conjunto de todos os itens de dados que a transação lê. O "**conjunto de gravação**" (write-set), é o conjunto de todos os itens de dados que a transação modifica no banco de dados.

Exercício: qual o read-set e o write-set de T1 e de T2?

Quando pensamos em um ambiente multi-usuário e multi-programado, transações podem ser submetidas por vários usuários e executadas de maneira concorrente pelo SGBD, acessando os mesmos itens de dados. Se estas execuções não

forem controladas, podemos ter problemas e tornarmos nosso banco de dados inconsistente. Bem-vindo ao mundo do "controle de concorrência" e da "recuperação de falhas".

4.2 Por que Controlar a Concorrência?

Olhando para a Figura 9, imagine que o item de dados "X" representa o número de assentos reservados em um determinado vôo "V1" e que "Y" representa o número de assentos reservados em um outro vôo "V2".

Imagine que T1 e T2 são submetidas ao mesmo tempo por dois usuários diferentes. T1 transfere "N" assentos reservados do vôo "V1" para o vôo "V2". T2 reserva "M" assentos no vôo "V1".

4.2.1 Problema 1 – Perda de Atualização

T1	T2

read_item(X);
X:=X-N;
 	
write_item(X);
read_item(Y);
 	
Y:=Y+N;
write_item(Y);
	
	read_item(X);
	X:=X+M;
	
	write_item(X);

Observamos claramente que o item de dados "X" atualizado em T1 foi perdido com a atualização posterior de T2. T2 lê o valor de "X" **ANTES** de T1 atualizá-lo com o novo valor. Confira com valores numéricos.

4.2.2 Problema 2 – Atualização Temporária (Leitura Suja – *dirty read*)

T1	T2

read_item(X);
X:=X-N;
write_item(X);
 	
read_item(Y);
<<FALHA NA TRANSAÇÃO>>
	
	read_item(X);
	X:=X+M;
	write_item(X);

Neste caso, com a falha da transação T1, o valor **ORIGINAL** de "X" é retornado ao banco de dados (sem a subtração do valor "N"). O problema se dá porque "T2" já havia lido o valor temporário de "X" e efetuou as contas com um valor incorreto. Confira com valores numéricos.

4.2.3 Problema 3 – Agregação Incorreta

T1	T3
	Sum:=0; Read_item(A); Sum:=Sum + A; (…)

read_item(X);
X:=X-N;
write_item(X);	

	Read_item(X); Sum:=Sum+X; Read_item(Y); Sum:=Sum+Y;

read_item(Y);
Y:=Y+N;
write_item(Y);	

Neste cenário, temos uma terceira transação "T3" que está calculando o total de assentos reservados em todos os vôos através de uma função de agregação "SUM". Em T1, o write-set é [X,Y], mas T3 utiliza o valor correto de X e o valor incorreto de Y, pois a transação T1 ainda não corrigiu o valor de Y.

4.3 Por que Mecanismos de Recuperação?

Sempre que uma transação é submetida ao SGBD, este deve garantir que: (1) todas as operações da transação se completam com sucesso e seu efeito é registrado permanentemente no banco de dados, **OU** (2) a transação não tem absolutamente NENHUM efeito no banco de dados, nem em quaisquer outras transações.

Como uma transação é composta de diversas etapas, as primeiras etapas podem ser realizadas com sucesso até que uma determinada etapa **falhe**. Neste caso, o SGBD deverá automaticamente realizar o comando "desfazer transação" (*rollback*).

Tipos de falhas mais comuns:

1. Falha do Computador ("crash do sistema") → erro de hardware, software ou de rede ocorrido durante a execução da transação. Erros de hardware são geralmente relacionados como a memória principal ou a CPU.
2. Erro Lógico da Transação → overflow, divisão por zero, violação de restrições de integridade, aborto espontâneo pelo usuário etc.
3. Imposição do Controle de Concorrência → o próprio sistema pode detectar um impasse (*deadlock*) e desfazer automaticamente uma das transações.

4.4 Propriedades Desejáveis de uma Transação

Propriedades ACID.

Atomicidade → uma transação é uma unidade atômica de processamento; é realizada integralmente ou não realizada de jeito algum.

Consistência Preservada → uma transação é preservadora de consistência se a sua plena execução levar o banco de dados de um estado consistente para outro estado também consistente.

Isolamento → uma transação deve parecer como se estivesse sendo executada isoladamente de outras transações. Ou seja, a execução de uma transação não deve sofrer interferência de quaisquer outras transações que estejam sendo executadas concorrentemente.

Durabilidade ou Persistência → as alterações aplicadas ao banco de dados por meio de uma transação confirmada devem persistir no banco de dados. Estas modificações não podem ser perdidas em função de nenhuma falha.

4.5 Transações na linguagem SQL

De forma geral, os ambientes interativos mantém uma transação aberta até que sejam executados os comandos “commit” ou “rollback”. Neste momento, uma nova transação é aberta.

Os diversos SGBDs do mercado têm a opção de configurar o “isolamento” da transação, permitindo que leituras sujas sejam suportáveis, por exemplo. O nível de isolamento default na maioria dos produtos é “read commited”.

Capítulo 5 - Técnicas de Controle de Concorrência