1.1 O Modelo Relacional

Um banco de dados relacional consiste em uma coleção de *tabelas*, cada qual designada por um nome único. Cada tabela é uma estrutura representada no modelo entidade-relacionamento E-R, este último é baseado na percepção do mundo real que consiste em um conjunto de objetos básicos chamados *entidades* e nos *relacionamentos* entre esses objetos. Ele foi desenvolvido para facilitar o projeto de banco de dados permitindo a especificação de um esquema de empresa. Tal esquema representa a estrutura lógica geral do banco de dados.

1.1.1 Entidades e Conjuntos de Entidades

Uma Entidade é um objeto que existe e é distinguível dos outros objetos. Por exemplo, João Alberto com o RG 890123456 é uma entidade, visto que isto identifica unicamente uma pessoa particular no universo. Uma entidade pode ser concreta, como uma pessoa ou livro, ou abstrata, como um feriado ou um conceito.

Um conjunto de Entidades é um conjunto de entidades do mesmo tipo. O conjunto das pessoas com conta em um banco, por exemplo, pode ser definido como o conjunto de entidades Cliente.

Conjuntos de Entidades não precisam ser disjuntos. Por exemplo, é possível definir o conjunto de todos os funcionários de um banco e o conjunto de todos os clientes do mesmo banco. Uma entidade pessoa pode ser uma entidade funcionário, uma entidade cliente, ambas ou nenhuma delas.

Uma Entidade é representada por um conjunto de atributos. Os possíveis atributos de uma entidade cliente podem ser: nome-cliente, rg-cliente, rua-cliente e cidade-cliente. Para cada atributo, existe um conjunto de valores permitidos, chamados domínio daquele atributo. O domínio do atributo nome-cliente pode ser o conjunto de todas as cadeias de texto com um certo tamanho. Assim, o domínio do atributo número-conta pode ser o conjunto de todos os inteiros positivos.

Formalmente, um atributo é uma função que mapeia um conjunto de entidades em um domínio. Portanto, toda entidade é descrita por um conjunto de pares (atributo, valor de dado), um par para cada atributo do conjunto de entidades. Uma entidade cliente é descrita pelo conjunto {(nome,Harris), (rg,890123456), (rua,Padre Rodolfo), (cidade,Pindamonhangaba)}, que significa que a entidade descreve uma pessoa chamada Harris com documento 890123456 que mora na rua Padre Rodolfo em Pindamonhangaba.

O conceito de um conjunto de Entidades corresponde à noção de definição de tipo usada em linguagens de programação. Uma variável de determinado tipo tem um valor particular em um determinado instante de tempo. Uma variável em linguagens de programação corresponde ao conceito de uma entidade no modelo E-R.

Um banco de dados inclui uma coleção de conjuntos de entidades, cada qual contendo qualquer número de entidades do mesmo tipo.

1.2 Estrutura Básica

Considere uma tabela chamada DEPOSITO. Ela tem quatro atributos: nome-agência, número-conta, nome-cliente e saldo. Para cada atributo, existe um conjunto de valores permitidos, chamado *domínio* daquele atributo. Para o atributo nome-agência, por exemplo, o domínio é o conjunto de todos os nomes de agências. Seja D_1 este conjunto, D_2 o conjunto de todos os números de contas, D_3 o conjunto de todos os nomes de

clientes e D_4 o conjunto de todos os saldos. Sendo assim qualquer linha de depósito deve consistir em uma quádrupla (v_1, v_2, v_3, v_4), onde v_1 é um nome de agência (isto é, v_1 está no domínio de D_1), v_2 é um número de conta, v_3 é um nome de cliente e v_4 é um saldo. Geralmente, depósito conterá apenas um subconjunto de todas as linhas possíveis. Conseqüentemente, depósito é um subconjunto de

$$D_1 \times D_2 \times D_3 \times D_4$$

Geralmente, uma tabela de n colunas precisa ser um subconjunto de

$$D_1 \times D_2 \times \ldots D_{n-1} \times D_n$$

Os matemáticos definem uma relação como um subconjunto de um produto cartesiano de uma lista de domínios. Isso corresponde quase exatamente à definição de tabela. A única diferença é que associamos nomes a atributos, ao passo que matematicamente são usados “nomes” numéricos, usando o inteiro 1 para representar o atributo cujo domínio aparece primeiro na lista de domínios, 2 para o segundo e assim por diante. Devido ao fato de as tabelas serem essencialmente relações, podemos usar os termos relação e tupla no lugar de tabela e linha.

1.3 Esquema de Banco de Dados

Quando falamos sobre um banco de dados, precisamos diferenciar entre o esquema e a instância de um banco de dados, que é o dado dentro do banco de dados em um determinado instante.
O conceito do esquema relacional corresponde à noção de definição de tipo das linguagens de programação. Uma variável de um dado tipo tem um valor particular em um instante determinado. Assim, uma variável corresponde ao conceito de uma instância de uma relação.
É conveniente atribuir um nome a um esquema relacional. Usaremos aqui como exemplo uma notação que usa nomes com letras minúsculas para relações e nomes começando com maiúsculas para os esquemas das mesmas. Seguindo esta notação podemos usar Esquema-depósito para representar o esquema relacional para a relação depósito.

Esquema-depósito = (nome-agência, número-conta, nome-cliente, saldo)

Ressaltando o fato de depósito ser uma relação num esquema Depósito dado por depósito (Esquema-depósito)

Geralmente, um esquema relacional é uma lista de atributos e de seus domínios correspondentes, e quando desejamos defini-los utilizamos a notação abaixo, a fim de definir o esquema relacional para a relação depósito.

(nome-agência:string[1], número-conta:integer[2], nome-cliente:string, saldo:Float[3])

[1] String é uma cadeia de caracteres (letras e números) utilizada para armazenamento de dados.

[2] Integer é utilizado para números não fracionados positivos ou não com um domínio de até 1 bilhão.

Como outro exemplo, considere a relação cliente, cujo esquema é:

Esquema-cliente = (nome-cliente, rua, cidade-cliente)

Note que nome-cliente aparece em ambos os esquemas. Isto não é uma coincidência. Ao contrário, o uso de atributos comuns é uma forma de relacionar tuplas de relações distintas. Por exemplo, suponha que desejamos achar as cidades onde moram os clientes que têm conta em determinada agência. Primeiro encontramos os clientes desta agência na relação depósito, então para cada cliente, examinamos a relação cliente para encontrar a cidade onde ele vive. Usando a terminologia do modelo entidade-relacionamento, diremos que o atributo nome-cliente representa o mesmo conjunto em ambas as relações.
Talvez pareça que no nosso exemplo, teríamos apenas um esquema relacional ao invés de diversos. Pode ser mais fácil pensar em apenas um esquema do que em vários. Suponha que utilizemos apenas uma relação para o nosso esquema:

Esquema-conta-info = (nome-agência, número-conta, nome-cliente, saldo, rua, cidade-cliente)

Observe que, se um cliente pode ter várias contas, precisamos listar seu endereço para cada conta. Ou seja, precisamos repetir certas informações várias vezes. Esta repetição é dispendiosa e pode ser evitada usando duas relações, como em nosso exemplo. Fora que, imagine se o cliente mudou de endereço, o trabalho que dá ficar atualizando cada conta dele, ou mesmo em uma determinada falha do sistema algumas contas são atualizadas com o novo dado e outras não.

1.4 Restrições de mapeamento

Um esquema relacional pode definir certas restrições com as quais o conteúdo do banco de dados tem de estar de acordo. Uma restrição muito importante são as chamadas cardinalidades[4] do mapeamento, que expressam o número de entidades às quais outra entidade pode ser associada via um conjunto de relacionamentos.
Estas cardinalidades são muito úteis na descrição de conjuntos de relacionamentos binários, embora ocasionalmente contribuam para a descrição de relacionamentos que envolvam mais de dois conjuntos de entidades (ou tabelas).
Para um conjunto de relacionamento binário R entre os conjuntos A e B, a cardinalidade pode ser uma das seguintes:

- **um-para-um:** Uma entidade A está associada no máximo à uma entidade de B, e uma entidade B está associada no máximo à uma entidade de A.
- **um-para-muitos:** Uma entidade A está associada à qualquer número de entidades de B, e uma entidade B, entretanto, está associada no máximo à uma entidade A.
- **muitos-para-um:** Uma entidade A associada no máximo à uma entidade de B, e uma entidade B,

[3] Float ou número de ponto flutuante é utilizado para armazenamento de dados fracionados.

[4] Podemos chamar esta característica de multiplicidade também.

entretanto, pode estar associada à qualquer número de entidades de A.

- **muitos-para-muitos:** Uma entidade A está associada à qualquer número de entidades de B, assim como uma entidade B está associada à qualquer número de entidades de A.

A cardinalidade do mapeamento é obviamente dependente do mundo real que está sendo modelado pelo conjunto de relacionamentos.

Para ilustrar, considere o relacionamento ContaCliente. Se, em um banco, uma conta pode pertencer à apenas um cliente, e um cliente pode ter várias contas, então este relacionamento é um-para-muitos de conta para cliente. Agora se a conta for conjunta, por exemplo, este relacionamento muda de muitos para muitos.

A dependência de existência forma outra importante classe de restrições. Especificamente, se a dependência da entidade x depende da existência da entidade y, então x é dito dependente da existência de y. Operacionalmente, isto significa que, se y for eliminado x também o será. A entidade y é chamada de dominante, e x é chamada de subordinada.

Para ilustrar, considere os conjuntos conta e transação. Formamos um conjunto de relacionamentos log entre dois conjuntos, o qual especifica que, para uma conta em particular, pode haver diversas transações. Este conjunto de relacionamentos é um-para-muitos de conta para transação. Toda entidade de transação deve estar associada a uma entidade de conta. Caso uma entidade de conta seja eliminada, todas as entidades de transação relacionadas a esta conta deverão ser eliminadas também (ou irão ficar perdidas no banco de dados). Em comparação as transações podem ser eliminadas sem prejudicar qualquer conta.

1.5 Chaves

É importante poder especificar como entidades e relacionamentos são identificados. Conceitualmente, entidades e relacionamentos individuais são distintos, mas numa perspectiva de banco de dados a diferença entre eles precisa ser expressa em termos de seus atributos. A *superchave* permite-nos fazer tais distinções. Uma superchave é um conjunto formado por um ou mais atributos que, tomado coletivamente, permite-nos identificar unicamente uma entidade no conjunto de entidades. Por exemplo, o atributo rg do conjunto de entidades cliente é suficiente para distinguir uma entidade cliente das outras. Desta forma, rg é uma superchave. De forma semelhante, a combinação nome-cliente e rg é uma superchave, já o atributo cliente por si só não é uma superchave, pois diversas pessoas podem ter o mesmo nome.

O conceito de superchave não é suficiente para nossos propósitos, uma vez que, como vimos anteriormente, uma superchave pode conter atributos aspúrios. Se K é uma superchave, então também o é qualquer subconjunto de K. Freqüentemente, procuramos superchaves que não tenham nenhum subconjunto próprio que seja uma superchave. Tais superchaves mínimas são chamadas de chaves *candidatas*.

É possível que diversos conjuntos distintos de atributos possam servir como uma chave candidata. Suponha que uma combinação de nome-cliente e rua seja suficiente para distinguir entre membros do conjunto de entidades cliente. Então ambos, {rg} e {nome-cliente, rua}, são chaves candidatas. Embora os atributos rg e nome-cliente juntos possam distinguir entidades de cliente, sua combinação não forma uma chave candidata, uma vez que rg sozinho já o seja.

Utilizamos o termo chave *primária* para denotar uma chave candidata que é escolhida pelo projetista de banco de dados como meio principal de identificar uma entidade dentro de um conjunto das mesmas.

É possível que um conjunto de entidades não contenha atributos suficientes para formar uma chave primária. Tal conjunto é nomeado como conjunto de entidades fraco. Da mesma forma que um conjunto que a possui denomina-se conjunto de entidades forte. Para ilustrar, considere o conjunto de entidade que possui os

atributos número-transação, data e valor, embora cada entidade transação seja distinta, transações de contas diferentes podem conter o mesmo número-transação. Assim, este conjunto não contém uma chave primária e é, portanto, fraco. Para que um conjunto destes tenha significado, ele deve fazer parte de um conjunto de relacionamentos um-para-muitos. Este conjunto de relacionamentos não deve ter atributos descritivos, uma vez que qualquer atributo requerido pode estar associado ao conjunto de entidades fraco.

Os conceitos de conjunto entidade forte e fraco estão relacionados às dependências de existência introduzidas na seção anterior. Um membro de uma entidade forte é por definição uma entidade dominante, enquanto que um membro de uma entidade fraca é uma entidade subordinada.

Embora um conjunto de entidades fraco não tenha uma chave primária, precisamos todavia de um meio de distinção entre todas essas entidades do conjunto que dependa de uma entidade forte particular. O discriminador de um conjunto de entidades fraco é um conjunto de atributos que permita que esta seja distinção feita. Por exemplo, o discriminador do nosso exemplo anterior é o atributo número-transação, uma vez que para cada conta um número de transação univocamente identifica uma única transação.

A chave primária de um conjunto de entidades fraco é formada pela chave primária do conjunto de entidades forte do qual dependente sua existência, mais seu discriminador. No caso do conjunto de entidades transação nossa chave primária é {número-conta, número-transação}, onde número-conta identifica a entidade dominante de uma transação e número-transação distingue entidades transação dentro da mesma conta.

A chave primária de um conjunto de entidades permite-nos distinguir as várias entidades dentro deste. Precisamos de um mecanismo similar para distinguir os vários relacionamentos dentro de um conjunto de relacionamentos (banco de dados). Para isso, precisamos explicar primeiro como os relacionamentos individuais são descritos. Uma vez que isso seja feito, podemos explicar como uma chave primária para um conjunto de relacionamentos é definida.

Seja R um relacionamento envolvendo os conjuntos de entidades E_1, E_2, ..., E_n. Digamos que a chave primária (E_i) designe o conjunto de atributos que formam a chave primária do conjunto de entidades E. Assuma que os nomes dos atributos de todas as chaves primárias sejam únicos. Suponha que R não tenha atributos. Então os atributos que descrevem relacionamentos individuais do conjunto R, representado por atributo (R), são:

chave-primária(E_1) U ... U chave-primária(E_n) U {a_1, a_2, ..., a_n}

Para ilustrar, considere o conjunto de relacionamentos ContaCliente, que envolve os seguintes conjuntos de entidades:

- cliente, com chave primária rg.
- conta, com chave primária número-conta.

Uma vez que o conjunto de relacionamentos tem o atributo data, o conjunto atributo (ContaCliente) consiste em três atributos: rg, número-conta e data.

Podemos explicar agora o que constitui a chave primária do conjunto de relacionamentos R. A composição da chave primária depende da cardinalidade do mapeamento e da estrutura dos atributos associados ao conjunto de relacionamentos R.

Considere agora o conjunto ContaCliente, se o mesmo for muitos para muitos, então sua chave primária é {rg, número-conta}, já se for um para muitos de cliente para conta então sua chave primária é {rg}.

1.6 Linguagens de Consulta

Uma linguagem de consulta é uma linguagem na qual um usuário requisita informações do banco de dados. Essas linguagens são tipicamente de mais alto nível do que as linguagens de programação. Elas podem ser classificadas como procedurais e não-procedurais. Numa linguagem procedural, o usuário instrui o sistema a executar uma seqüência de operações no banco de dados a fim de computar o resultado desejado. Numa linguagem não-procedural, o usuário descreve a informação desejada sem fornecer um procedimento específico para obter tal informação.

Neste ponto, devemos referir-nos apenas às consultas. Uma linguagem completa de manipulação inclui não apenas uma linguagem de consulta, mas também uma linguagem para modificação do banco de dados. Tais linguagens incluem comandos para inserir e apagar tuplas assim como comandos para modificar partes de uma tupla existente. Examinaremos modificações no banco de dados depois de completar nossa discussão sobre a linguagem de consulta.

1.7 Álgebra Relacional

A álgebra relacional é uma linguagem de consulta procedural. Ela consiste em um conjunto de operações que usam uma ou mais relações como entrada e produzem uma nova relação como seu resultado. As operações fundamentais na álgebra relacional são: selecionar, projetar, produto cartesiano, re-nomear, união e diferença de conjuntos, junção natural, divisão e atribuição. Estas operações serão definidas em termos de operações fundamentais.

1.7.1 Selecionar

Esta operação seleciona as tuplas que satisfazem um dado predicado. Usamos a letra grega sigma (σ) para representar a seleção. O predicado aparece como subscrito de σ. A relação argumento é dada entre parênteses seguindo a letra σ. Assim, para selecionar as tuplas da relação empréstimo onde agência é “Cambuci” escrevemos:

$$\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{empréstimo})$$

O resultado desta operação pode ser semelhante à:

nome-agência	número-empréstimo	nome-cliente	quantia
Cambuci	15	João	1500
Cambuci	27	Maria	2500

Podemos encontrar todas as tuplas nas quais a quantia emprestada é maior que $ 1.200 escrevendo:

$$\sigma_{\text{quantia}>1200}(\text{empréstimo})$$

Geralmente, permitimos comparações usando no predicado da seleção =, ≠, <, =, >, =. Além disso, diversos predicados podem ser combinados em um predicado maior usando os conectivos e (Λ) e ou (V). Assim para encontrar aquelas tuplas pertencentes a empréstimos maiores que $1200 feitos pela agência Cambuci temos:

$$\sigma_{\text{nome-agência}=\text{“Cambuci”} \Lambda \text{quantia}>1200}(\text{empréstimo})$$

O predicado de seleção pode incluir comparações entre dois atributos. Para ilustrar considere o esquema:

Esquema-usuário = (nome-cliente, nome-gerente)

Ilustrando podemos encontrar todos os clientes que têm o mesmo nome do gerente escrevendo:

$$\sigma_{\text{nome-cliente}=\text{nome-gerente}}(\text{usuário})$$

1.7.2 Projetar

No exemplo anterior, obtivemos a relação (nome-cliente, nome-gerente) na qual t[nome-cliente] = t[nome-gerente] para todas as tuplas t. Pode parecer redundante listar duas vezes o nome de pessoas. Poderíamos preferir uma relação apenas com os nomes dos clientes. A operação projetar permite-nos produzir esta relação, ela é uma operação unária que nos dá como resultado sua relação argumento com certos atributos deixados de fora. Ela é representada pela letra grega pi (π). Listamos os atributos desejados nos resultados como subscritos de π. O argumento de relação segue a letra π entre parênteses.

Suponha que desejemos uma relação mostrando clientes e nomes das agências das quais eles tomaram empréstimos. Escrevemos:

$$\pi_{\text{nome-agência, nome-cliente}}(\text{empréstimo})$$

Agora voltando à consulta anterior podemos escrever:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-cliente}=\text{nome-gerente}}(\text{usuário}))$$

Note que, em vez de dar o nome da relação como argumento da operação demos uma expressão cujo cálculo dá como resultado uma outra relação.

1.7.3 Produto Cartesiano

As Operações discutidas até o momento permitem extrair informações de uma tabela de cada vez. Ainda não fomos capazes de combinar informações a partir de duas relações. Uma operação que nos permite esta tarefa é o *produto cartesiano*, representado por uma cruz (x). Trata-se de uma operação binária. Devemos usar a

notação infixa[5] para operações binárias e então escrever o produto cartesiano das relações r_1 e r_2 como r_1 x r_2. Lembremos que uma relação é definida como um subconjunto cartesiano de um conjunto de domínios. A partir desta definição, devemos ter alguma intuição sobre a definição da operação cartesiana da álgebra relacional.
Suponha que desejamos listar os clientes que fizeram empréstimo na agência Cambuci, assim como seus respectivos endereços. Precisamos, neste caso, de informações contidas na relação cliente e empréstimo. O resultado r seria empréstimo x cliente o que seria representado por:

r = (cliente.nome-cliente, cliente.rua, cliente.cidade-cliente, empréstimo.nome-cliente, empréstimo.nome-agência, empréstimo.quantia).

Ou seja, todos os atributos das relações ligados por um atributo em comum, no caso nome-cliente. Se torna necessário adicionar o nome da relação para distinguir cliente.nome-cliente de empréstimo.nome-cliente. Para os atributos que aparecem em apenas um dos esquemas pode-se suprimir o prefixo nome da relação já que não resultará em nenhuma ambigüidade, podendo ser escrito assim:

r = (cliente.nome-cliente, rua, cidade-cliente, empréstimo.nome-cliente, nome-agência, quantia)

Agora que conhecemos o esquema relacional resultante, quais tuplas aparecem em r ? Como pode-se suspeitar, mas não tão obviamente, construímos uma tupla de r a partir de cada par de tuplas possíveis, uma para relação cliente e outra da relação empréstimo, ou matematicamente um produto cartesiano mesmo. Assim, r pode ser uma relação enorme, mas que não condiz à nossa necessidade. Lembra-se “clientes que fizeram empréstimo na agência Cambuci, assim como seus respectivos endereços”, considerando a relação r = cliente x empréstimo. Se escrevermos:

$$\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{cliente X empréstimo})$$

Então, teremos uma relação com os empréstimos pertencentes à agência Cambuci. Entretanto, a coluna cliente.nome-cliente pode conter clientes que não sejam da agência Cambuci. Se, você não entende como, recorde-se que um produto cartesiano toma todos os possíveis pares de uma tupla cliente com uma tupla empréstimo. Note que a coluna empréstimo.nome-cliente contém apenas clientes da agência Cambuci.
Uma vez que a operação cartesiana associa qualquer tupla de empréstimo com qualquer outra de cliente, sabemos que algumas tuplas em cliente x empréstimo têm nossa informação correta. Isto ocorre nos casos em que empréstimo.nome-cliente é igual à cliente.nome-cliente. Assim se escrevemos:

$$\sigma_{\text{cliente.nome-cliente}=\text{empréstimo.nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{cliente X empréstimo}))$$

Teremos somente aquelas tuplas de cliente x empréstimo cuja a agência é Cambuci e contém os dados dos clientes da mesma.

[5] Notação “infixa” é usual, isto é, operando - operador - operando, nessa ordem.

1.7.4 Re-nomear

Toda vez que necessitamos repetir a mesma relação mais de uma vez em uma determinada consulta surge a necessidade de contornar a ambigüidade desta consulta. Ilustrando, considere a consulta "Dê os nomes dos clientes que moram na mesma rua e cidade de João". Podemos obter a rua e cidade onde mora o João escrevendo:

$$\pi_{\text{rua, cidade-cliente}}(\sigma_{\text{nome-cliente}=\text{“João”}}(\text{cliente}))$$

Entretanto, para encontrarmos outros clientes com esta rua e cidade, devemos relacionar cliente uma segunda vez:

$$\sigma_{P}(\text{cliente} \times \pi_{\text{nome-cliente}}(\sigma_{\text{nome-cliente}=\text{“João”}}(\text{cliente})))$$

Onde P é um predicado de seleção que requer que os valores de cidade-cliente e rua sejam iguais. Para especificar, por exemplo, a qual valor de rua nos referimos, não podemos usar cliente.rua, uma vez que os valores de rua são tomados da relação cliente. Dificuldade semelhante iremos encontrar para cidade-cliente. Este problema é resolvido usando o operador re-nomear, representado por ρ. A Expressão:

$$\rho_{x}^{(r)}$$

Dá como resultado a relação r como o nome x. Devemos usar isto para re-nomear uma referência à relação cliente e então referenciar duas vezes sem ambigüidades. Na consulta seguinte, usamos o nome cliente2 como um segundo nome para a relação cliente. Referimo-nos à cliente2 quando computamos a rua e a cidade de João.

$$\pi_{\text{cliente.nome-cliente}}(\sigma_{\text{cliente2.rua} = \text{cliente.rua} \wedge \text{cliente2.cidade-cliente} = \text{cliente.cidade-cliente}}(\text{cliente} \times \pi_{\text{nome-cliente}}(\sigma_{\text{nome-cliente}=\text{“João”}}\rho_{\text{cliente2}}^{(\text{cliente})})))$$

1.7.5 União

Vamos considerar a consulta "Dê todos os clientes da agência Cambuci". Isto é, localize todos que possuem conta e/ou empréstimo. Para responder esta consulta precisamos dos dados da relação empréstimo e da relação depósito. Sabemos como encontrar todos os clientes com empréstimo:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{empréstimo}))$$

Sabemos também como encontrar todos os clientes que possuem conta nesta agência:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{depósito}))$$

Para responder a consulta, utilizamos a operação binária União, representada, como na teoria de conjuntos, por U. Assim, a expressão da consulta que precisamos fica:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{empréstimo}))$$
$$\cup\ \pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{depósito}))$$

1.7.6 Diferença

A operação Diferença, representada por -, permite-nos encontrar tuplas que estão em uma relação, mas não estão em outra. A expressão r - s resulta em uma relação contendo as tuplas que estão em r, mas não em s.
Podemos, por exemplo, encontrar os clientes que têm conta na agência Cambuci, mas não tem empréstimo escrevendo:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{depósito}))$$
$$-\ \pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{empréstimo}))$$

1.8 Definição formal de álgebra relacional

As operações que vimos permitem-nos dar uma definição completa de uma expressão relacional. Uma expressão básica em álgebra relacional consiste em uma das expressões que seguem:

- Uma relação no banco de dados.
- Uma relação constante[6].

Uma expressão geral é construída a partir de sub-expressões menores. Sejam E_1 e E_2, expressões da álgebra relacional. Então, as seguintes são expressões relacionais:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, onde P é um predicado sobre os atributos de E_1
- $\pi_S(E_1)$, onde S é uma lista de alguns atributos de E_1
- $\rho_X(E_1)$, onde X é o novo nome para a relação E_1

1.9 Operações Adicionais

As operações fundamentais são suficientes para expressar qualquer consulta da álgebra relacional. Entretanto, se nos restringirmos apenas a elas, algumas consultas comuns são longas demais para serem expressas. Portanto, definimos algumas operações adicionais que não adicionam nenhuma potência à álgebra, mas simplificam consultas comuns.

[6] Relação constante é uma relação dada pelo usuário do banco de dados apenas na formulação da consulta; portanto, ela não pertence ao banco de dados. Note-se que a definição dada é do tipo indutivo: define expressões usando sub-expressões.

1.9.1 Intersecção

A primeira operação adicional que devemos definir é a intersecção de conjuntos (n). Suponha que desejemos encontrar todos os clientes com empréstimo e depósito na agência Cambuci. Usando a intersecção podemos escrever:

$$\pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{empréstimo}))$$
$$\cap\ \pi_{\text{nome-cliente}}(\sigma_{\text{nome-agência}=\text{“Cambuci”}}(\text{depósito}))$$

Observe que qualquer expressão utilizando intersecção pode ser re-escrita substituindo a operação intersecção por um par de diferenças de conjuntos como segue:

$$r \cap s = r - (r - s)$$

Assim, a intersecção não é uma operação fundamental nem acrescenta nenhuma potência à álgebra relacional. É simplesmente mais conveniente escrever r n s do que r - (r - s).

1.9.2 Junção Natural

É freqüentemente desejável simplificar certas consultas que requerem um produto cartesiano. Tipicamente, uma consulta cartesiana envolve uma operação de seleção no resultado desse produto. Considere a consulta: “Dê os clientes que têm um empréstimo no banco e as cidades em que vivem”. Primeiro, formamos o produto cartesiano das relações empréstimo e cliente, e então selecionamos aquelas tuplas que pertencem a um único nome-cliente. Para isso, escrevemos:

$$\pi_{\text{empréstimo.nome-cliente, cidade-cliente}}$$
$$(\sigma_{\text{cliente.nome-cliente}=\text{empréstimo.nome-cliente}}(\text{cliente X empréstimo}))$$

A junção natural é uma operação binária que permite combinar certas seleções de um produto cartesiano em uma operação. É representada pelo símbolo “Junção”. A operação Junção Natural forma um produto cartesiano de seus dois argumentos, faz uma seleção forçando uma igualdade sobre os atributos que aparecem em ambos os esquemas relação e finalmente remove colunas duplicadas. Sendo assim nossa consulta fica:

$$\pi_{\text{empréstimo.nome-cliente, cidade-cliente}}\ (\text{cliente}\ \ \text{empréstimo})$$

1.9.3 Divisão

A operação Divisão, representada por ÷, serve para consultas que incluem a frase “para todos”. Suponha que desejemos encontrar todos os clientes que têm uma conta em todas as agências localizadas em São Paulo. Podemos encontrar as agências pela expressão:

$$r_1 = \pi_{\text{nome-agência}}(\sigma_{\text{cidade-agência}=\text{“São Paulo”}}(\text{agências}))$$

Podemos encontrar os clientes que possuem conta em alguma agência escrevendo:

$$r_2 = \pi_{\text{nome-cliente, nome-agência}}(\text{depósito})$$

Agora precisamos encontrar clientes que aparecem em r_2 com cada nome de agência em r_1. A consulta pode ser respondida escrevendo:

$$\pi_{\text{nome-cliente, nome-agência}}(\text{depósito}) \div \pi_{\text{nome-agência}}(\sigma_{\text{cidade-agência}=\text{“São Paulo”}}(\text{agências}))$$

1.10 SQL

Existem inúmeras versões de SQL. A versão original foi desenvolvida no Laboratório de Pesquisa da IBM San Jose (hoje o centro de pesquisa Almaden). Esta linguagem originalmente chamada Sequel, foi implementada como parte do projeto System R no início dos anos 70. A linguagem evoluiu desde então, e seu nome foi mudado para SQL (Structured Query Language)[7]. Numerosos produtos suportam a linguagem SQL. Embora as versões do produto SQL difiram em diversos detalhes de linguagem, as diferenças são, em sua maioria, secundárias.
Em 1986, o American National Standard Institute (ANSI) publicou um padrão para o SQL. A IBM publicou seu próprio padrão, o System Aplication Architecture (SAA-SQL).

1.11 Estrutura Básica

A estrutura básica de uma expressão SQL, consiste em três cláusulas: **select**, **from** e **where**.

- **Select** corresponde à operação projeção da álgebra relacional. Sendo usada para listar os atributos desejados no resultado de uma consulta.
- **From** corresponde à operação produto cartesiano. Ela lista as relações a serem examinadas na avaliação da expressão.
- **Where** corresponde ao predicado da seleção da álgebra relacional. Consiste em um predicado envolvendo atributos das relações constantes na cláusula **From**.

Uma típica consulta SQL tem a forma:

select $a_1, a_2, ..., a_n$
from $r_1, r_2, ..., r_n$
where P

Cada a_i representa um atributo e cada r_i é uma relação. P é um predicado. Esta consulta é equivalente à expressão da álgebra relacional.

$$\pi_{a_1, a_2, ..., a_n}(\sigma_P(r1 \times r2 \times ... rn))$$

[7] “SEQUEL” era a abreviatura de “Structured English Query Language”; a IBM fez muito bem em eliminar o english como se verá na sintaxe do SQL.

Caso a cláusula **where** seja omitida, o predicado P é verdadeiro. A lista a_1, a_2, ..., a_n de atributos pode ser substituída por um asterisco (*) para selecionar todos os atributos de todas as relações presentes na cláusula **from**.

A expressão SQL forma o produto cartesiano das relações mencionadas na cláusula **from**, executa uma seleção da álgebra relacional usando o predicado da cláusula **where**, e então, projeta o resultado para os atributos da cláusula **select**.
O resultado de uma consulta SQL é, obviamente, uma relação. Vamos considerar uma consulta simples usando nosso exemplo bancário, "Dê os nomes de todas as agências na relação depósito".

select nome-agência **from** depósito

1.12 Operações de conjuntos e tuplas duplicadas

Linguagens de consulta são baseadas na noção matemática de uma relação como sendo um conjunto. Assim, nunca aparecem tuplas duplicadas em relações. Todavia, SQL permite duplicações em relações. A consulta anterior listará então nome-agência uma para cada tupla na qual ela aparece na relação depósito.
Nesses casos onde queremos forçar a eliminação de duplicações, inserimos a palavra **distinct** depois de **select**. Podemos re-escrever a consulta anterior se quisermos as duplicações removidas como:

select distinct nome-agência **from** depósito

Note-se também que é permitido o uso da palavra **all** para especificar explicitamente que as duplicações não serão removidas. Uma vez que a preservação das duplicações é padrão, o uso da palavra **all** torna-se desnecessário se quisermos as duplicações:

select all nome-agência **from** depósito

1.13 Operações com conjuntos

A SQL inclui as operações **union**, **intersect**, e **minus**[8], que operam sobre relações e correspondem às operações U, n e - da álgebra relacional.

(**select** nome-cliente **from** depósito)
union
(**select** nome-cliente **from** empréstimo)

Como padrão a operação **union** elimina as tuplas duplicadas. Para reter duplicações torna-se necessário escrever **union all** no lugar de **union**.

[8] O Interbase e muitos outros bancos de dados implementam apenas a operação **union.**

1.14 Predicados e Junções

SQL não tem uma representação para junção natural. No entanto, uma vez que a junção natural é definida em termos de produtos cartesianos é simples escrever uma expressão SQL para uma junção natural.

Lembre-se da expressão:

$\pi_{\text{empréstimo.nome-cliente, cidade-cliente}} (\text{cliente} \quad \text{empréstimo})$

Em SQL podemos escrever esta expressão como:

select distinct cliente.nome-cliente, cidade-cliente
from empréstimo, cliente
where empréstimo.nome-cliente=cliente.nome-cliente

Vamos estender a consulta e considerar um caso mais complicado. Suponha que queremos apenas os clientes da agência “Cambuci’, podemos reescreve-lá assim:

select distinct cliente.nome-cliente, cidade-cliente
from empréstimo, cliente
where empréstimo.nome-cliente=cliente.nome-cliente and nome-agência=“Cambuci”

Em SQL usam-se os conectivos lógicos **and**, **or**, e **not** em vez dos símbolos matemáticos. Isto permite o uso de expressões aritméticas como operandos para os operadores de comparação. Uma expressão aritmética pode envolver qualquer dos operadores +, - , * e / em constantes ou valores de tuplas. Muitas implementações de SQL permitem funções aritméticas especiais para tipos particulares. Por exemplo, a SAA-SQL, da IBM, inclui numerosas funções para o tipo de dado data.
SQL inclui um operador de comparação **between** para simplificar as cláusulas **where** que especificam que um valor seja “maior que” ou “igual a” algum valor e “menor que” ou “igual a” algum outro valor. Se desejamos achar os números de contas com saldos entre 90.000 e 100.000, inclusive, podemos escrever:

select numero-conta **from** depósito
where saldo **between** 90000 **and** 100000

em vez de:

select número-conta **from** depósito
where saldo >= 90000 **and** saldo <=100000

SQL inclui também um operador de comparação de cadeias de caracteres. Os padrões são descritos usando dois caracteres especiais:

- % (porcentagem) - equivale a qualquer sub-cadeia.
- _ (sublinhado) - equivale a qualquer caractere.

Padrões de caracteres são sensíveis à forma, ou seja, caracteres maiúsculos e minúsculos não se equivalem. Para ilustrar considere os exemplos:

- "Pedro%" equivale a qualquer cadeia começando com "Pedro".
- "%Santos%" equivale a qualquer cadeia que contenha "Santos" em qualquer posição.
- "___" equivale a qualquer cadeia com 3 caracteres.
- "___%" equivale a qualquer cadeia com pelo menos 3 caracteres.

Padrões de caracteres são expressos em SQL usando o operador de comparação **like**. Considerando o exemplo:

select nome-cliente **from** clientes
where nome-cliente **like** "João%"

Para que os padrões de caracteres possam incluir os caracteres especiais (% e _), a SQL permite a especificação de um caractere de **escape**. Este é utilizado imediatamente antes de um caractere especial para indicar que este é um caractere normal. Considere o exemplo:

like "ab\%cd%" **escape** "\" equivale à todas as cadeias começando com "ab%cd"
like "ab\\cd%" **escape** "\" equivale à todas as cadeias começando com "ab\cd"

1.15 Pertinência a Conjuntos

SQL usa operações do cálculo relacional que permitem testar a pertinência de tuplas a uma relação usando o conectivo **in**, onde o conjunto pode ser uma coleção de valores produzidos por uma sub-consulta.

select nome-cliente **from** empréstimo
where nome-agência = "Cambuci"
  and nome-cliente **in** (**select** nome-cliente
    from depósito **where** nome-agência = "Cambuci")

É possível testar uma pertinência a uma relação arbitrária. A SQL usa a notação $<a_1, a_2, .., a_n>$ para representar uma tupla de n elementos. Por exemplo:

select nome-cliente **from** empréstimo
where nome-agência = "Cambuci"
  and <nome-agência,nome-cliente> **in** (**select** nome-agência, nome-cliente **from** depósito **where** nome-agência = "Cambuci")

1.16 Comparação de Conjuntos

Fomos capazes de usar a pertinência de conjuntos na consulta anterior, mas considere a consulta "Dê os nomes das agências que possuem ativos maiores alguma agência de SP". Podemos escrever:

select T.nome-agência **from** agência T, agência S
where T.ativos > S.ativos **and** S.cidade-agência = “SP”

SQL oferece, no entanto, um estilo alternativo para esta consulta. A frase “maior que algum” é representada pela palavra **some**. Esta consulta pode ser re-escrita assim:

select nome-agência **from** agência
where ativos > **some** (**select** ativos **from** agência
where cidade-agência = “SP”)

SQL permite comparações **< some, <= some, = some, >= some** e **> some**. A palavra reservada **any** é sinônimo de **some** em SQL. É possível também utilizar a expressão “maior que todos” com a palavra reservada **all**.

1.17 Teste de Relações Vazias

SQL inclui um recurso para testar se uma sub-consulta tem alguma tupla em seus resultados. A construção **Exists** retorna se o argumento sub-consulta é não vazio. Utilizando **Exists** podemos escrever a consulta “Dê todos os clientes que possuem uma conta e um empréstimo na agência Cambuci”.

select nome-cliente **from** cliente
where exists (select * **from** depósito
where depósito.nome-cliente = cliente.nome-cliente
and nome-agência = “Cambuci”)
and exists (select * **from** empréstimo
where empréstimo.nome-cliente = cliente.nome-cliente
and nome-agência = “Cambuci”)

1.18 Ordenação da Exibição de Tuplas

SQL oferece algum controle sobre a ordem na qual as tuplas são exibidas. A cláusula **order by** ocasiona o aparecimento de tuplas no resultado de uma consulta em uma ordem determinada.

select nome-cliente **from** empréstimo
order by nome-cliente

Como padrão, SQL lista itens na ordem ascendente. Para especificar a ordem de classificação podemos especificar **desc** para ordem descendente ou **asc** para ordem ascendente. Além do mais a ordenação pode ser feita com múltiplos atributos separados por vírgula, ou pode ser especificada também com a posição do atributo no resultado da consulta ao invés do seu nome:

select saldo, nome-cliente **from** empréstimo
order by saldo **desc**, nome-cliente **asc**

ou

select saldo, nome-cliente **from** empréstimo
order by 1 **desc**, 2 **asc**

1.19 Funções de Agregação

SQL oferece a possibilidade de computar funções em grupos de tuplas usando a cláusula **group by**. O atributo ou os atributos dados na cláusula **group by** são usados para formar grupos. Tuplas com o mesmo valor em todos os atributos na cláusula **group by** são colocados em um grupo. SQL inclui funções para computar:

- média : **avg**
- mínimo : **min**
- máximo : **max**
- soma : **sum**
- quantidade : **count**

Para ilustrar considere a consulta "Dê o saldo médio de conta em todas as agências":

select nome-agência, **avg(**saldo**) as** saldo
from depósito **group by** nome-agência

Às vezes, é útil definir uma condição que se aplique ao resultado do grupo em vez de cada tupla. Por exemplo, podemos estar interessados apenas em agências em que a média de saldo seja maior que $ 1200. Para expressar esta consulta devemos utilizar a cláusula **having** desta forma:

select nome-agência, **avg(**saldo**) as** saldo
from depósito **group by** nome-agência
having avg(saldo**)** > 1200

1.20 Junções

Algumas versões de SQL permite-nos, ainda, utilizar um meio especial de produto cartesiano utilizando-se as cláusulas **inner join, left join, right join** e **full join** e o predicado de ligação **on** na seguinte forma:

tabela1 **inner join** tabela2 **on** tabela1.campo = tabela2.campo

Ficando assim uma de nossas consultas anteriores assim

select distinct cliente.nome-cliente, cidade-cliente
from empréstimo **inner join** cliente **on** empréstimo.nome-cliente=cliente.nome-cliente
where nome-agência="Cambuci"

A diferença entre **inner**, **left**, **right** e **full** diz respeito à cardinalidade das relações:

- **inner** traz as tuplas que existem nas duas relações
- **left** traz as tuplas que existem na relação à esquerda
- **right** traz as tuplas que existem na relação à direita
- **full** traz todas as tuplas

1.21 Remoção

Uma requisição de remoção é expressa da mesma forma que uma consulta, sendo expressa por

delete from r **where** P

Note que o comando **delete** opera em apenas uma relação, se desejamos remover tuplas de várias relações executamos um comando para cada relação.

1.22 Inserção

Usada para inserirmos dados em uma relação, ou especificamos uma tupla para ser inserida ou escrevemos uma consulta cujo resultado seja um conjunto de tuplas para ser inserida. Obviamente os valores dos atributos para tuplas inseridas precisam ser membros do mesmo domínio do atributo. Da mesma forma as tuplas inseridas devem ser do mesmo tipo.
A instrução **insert** mais simples é uma requisição para inserir uma tupla como no exemplo:

insert into depósito (nome-agência, conta, nome-cliente, valor)
values (“Centro”, 1234, “José”, 1500)

Ou paralelamente através de uma consulta:

insert into depósito (nome-agência, conta, nome-cliente, valor)
select nome-agência, conta, nome-cliente, valor **from** empréstimo

1.23 Atualização

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar todos os valores na tupla. Para isto usamos a instrução **update**.

update depósito **set** saldo = saldo * 1.05

1.24 Visões

Uma visão do usuário é definida em SQL usando o comando **create view**. Para definir uma visão devemos dar um nome a ela e definir uma consulta que a processe.

create view todos-clientes **(**nome-agência, nome-cliente)
as (select nome-agência, nome-cliente **from** depósito
union select nome-agência, nome-cliente **from** empréstimo)