Apostila montada pelo professor
José Gonçalo dos Santos

Contato: jose.goncalo.santos@gmail.com

Conceitos de Banco de Dados

1 Introdução

Antes de começarmos a falar sobre as formas de acesso a dados e definirmos Banco de Dados (BD), algumas considerações devem ser feitas. Como, por exemplo, o que representa um BD em uma instituição. Podemos dizer que o BD é o coração da instituição. Devido a isso, qualquer dano causado ao BD pode atingir a "saúde" da instituição. Quando falamos em BD, devemos ter em mente os seguintes termos: dados, informações e SGBD - Sistemas de Gerenciamento de Banco de Dados. Mas, qual a relação desses termos com Banco de Dados?

Para responder à questão anterior de forma simples, dizemos que um banco de dados guarda dados; esses dados podem ser manipulados por um SGBD de maneira que permita que os usuários os acesse de forma que a integridade dos dados seja mantida, e que esses dados sejam transformados em informações.

Como pode ser notado, nos parágrafos anteriores, eu tratei de forma distinta; dados, informações, BD e SGBD. Na verdade, nem sempre essa distinção é feita, há pessoas que tratam o SQL SERVER de forma errônea como BD, o que na verdade é um SGBD; outras dizem que os BDs armazenam informações, o que não uma verdade, porque BD armazena dados, que podem ser transformados em informações, como por exemplo, a Figura 1.1 mostra três tabelas com dados sobre clientes, produtos e compras, mas se olharmos somente a tabela itens, não podemos identificar o cliente que efetuou o pedido. Porém, podemos retirar essa informação através de uma operação de consulta entre as três tabelas e obtermos a informação desejada. Disso, podemos concluir que um BD armazena dados, que podem ser transformados em informações através de inferências sobre eles.

Cliente

CCli	Nome	CPF	Fone
1	Tati	111111111-11	5221020
2	Fabi	222222222-22	6221530
3	Thays	333333333-33	8551315

Pedido

NPed	Data	Valor	Cliente
1	20/09/2004	300,00	2
2	21/09/2004	175,00	1
3	21/09/2004	350,00	3

Itens

NPed	Produto	Qtd	Valor
1	Blusa	1	50,0
1	Saia	1	100,00
1	Vestido	1	150,00
2	Sandália	1	40,00
2	Meias	3	15,00
2	Tênis	1	130,00
3	Saia	1	100,00
3	Blusa	2	100,00
3	Vestido	1	150,00

Figura 1.1: Exemplo de um banco de dados

1.1 Definição de BD

A tecnologia aplicada aos métodos de armazenamento de informações vem crescendo e gerando um impacto cada vez maior no uso de computadores, em qualquer área em que eles podem ser aplicados.

Um "banco de dados" pode ser definido como um conjunto de "dados" devidamente relacionados. Por "dados" podemos compreender com "fatos conhecidos" que podem ser armazenados e que possuem um significado implícito. Porém, o significado do termo "banco de dados" é mais restrito que simplesmente a definição dada acima. Um banco de dados possui as seguintes propriedades:

- Um banco de dados é uma coleção lógica coerente de dados com um significado inerente; uma disposição desordenada dos dados não pode ser referenciada como um banco de dados;
- Um banco de dados é projetado, construído e "povoado" com dados para um propósito específico; um banco de dados possui um conjunto pré-definido de usuários e aplicações;
- Um banco de dados representa algum aspecto de mundo real, o qual é chamado de "mini-mundo"; qualquer alteração efetuada no mini-mundo é automaticamente refletida no banco de dados.

Um banco de dados pode ser criado e mantido por um conjunto de aplicações desenvolvidas especialmente para esta tarefa ou por SGBD. Um SGBD permite aos usuários criarem e manipularem banco de dados de propósito geral. O conjunto formado

por um banco de dados mais as aplicações que o manipulam é chamado de “Sistema de Banco de Dados”.

1.2 Banco de Dados X Abordagem processamento Tradicional de Arquivos

1.2.1 Auto Informação

Uma característica importante da abordagem Banco de Dados é que o SGBD mantém não somente os dados, mas também a forma como são armazenados, contendo uma descrição completa do banco de dados. Estas informações são armazenadas no catálogo do SGBD, o qual contém informações como a estrutura de cada arquivo, o tipo e o formato de armazenamento de cada tipo de dado, restrições, entre outros. A informação armazenada no catálogo é chamada de “Meta dados”. No processamento tradicional de arquivos, o programa que irá manipular os dados é que contém este tipo de informação, ficando limitado a manipular as informações que ele conhece. Utilizando a abordagem banco de dados, a aplicação pode manipular diversas bases de dados diferentes.

1.2.2 Separação entre Programas e Dados

No Processamento tradicional de arquivos, a estrutura dos dados está incorporada ao programa de acesso. Desta forma, qualquer alteração na estrutura de arquivos implica na alteração no código fonte de todos os programas. Já na abordagem bando de dados, a estrutura é alterada apenas no catálogo, não alterando os programas.

A Figura 1.2 ilustra um sistema de banco de dados, como podemos observar, os dados são separados da aplicação e entre os dados e a aplicação existe o SGBD para controlar todo o processo de acesso e manipulação de dados, fazendo com que as regras do BD se tornem independente da aplicação utilizada pelo usuário final.

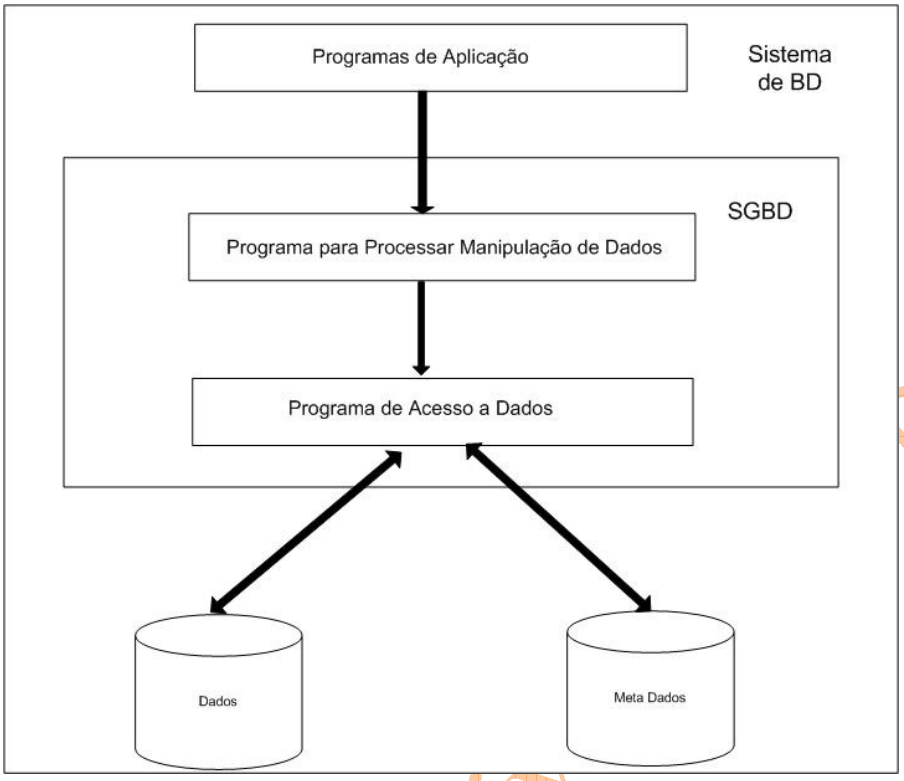

Figura 1.2: Exemplo de um Sistema de Banco de Dados.

1.2.3 Abstração de Dados

O SGBD deve fornecer ao usuário uma "representação conceitual" dos dados, sem fornecer muitos detalhes de como as informações são armazenadas. Um "modelo de dados" é uma abstração de dados que é utilizada para fornecer esta representação conceitual utilizando conceitos lógicos como objetos, suas propriedades e seus relacionamentos. A estrutura detalhada e a organização de cada arquivo são descritas no catálogo.

1.3 Sistemas de Gerenciamento de Banco de Dados – SGBD

Dá-se o nome de Sistema Gerenciador de Banco de Dados (SGBD) ao conjunto de programas responsável pela definição de uma base de dados, proporcionando um ambiente para o acesso da comunidade de usuários às informações armazenadas nos seus diversos arquivos. A Figura 1.3 ilustra este conceito.

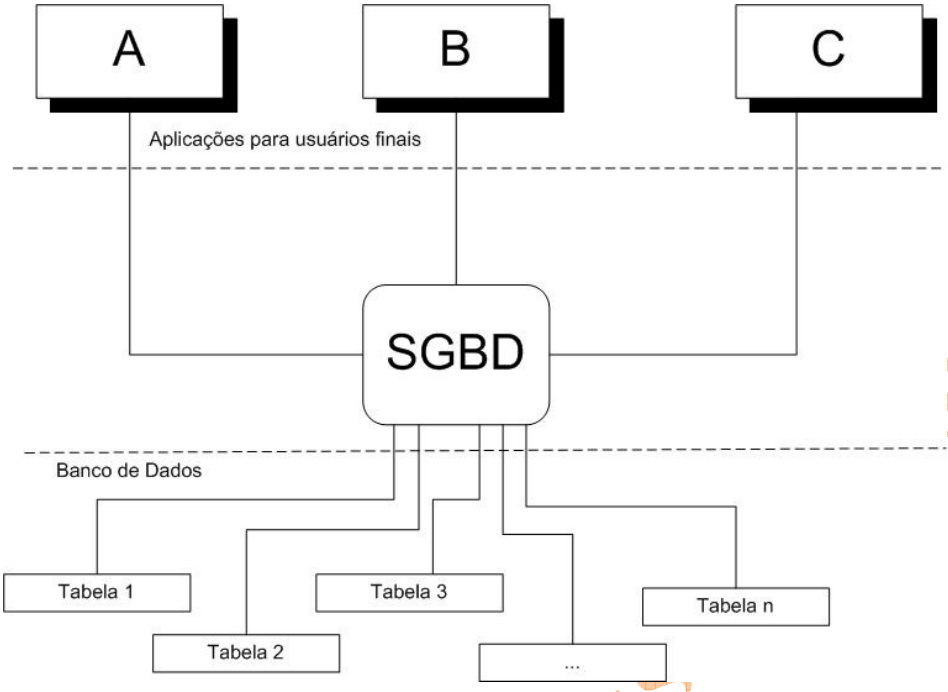

Figura 1.3: Sistema de Gerenciamento de Banco de Dados.

Ao se utilizar um SGBD, certos objetivos fundamentais devem estar implícitos, como a integração das atividades gerenciais e operacionais da empresa; a obtenção de maior segurança em termos do armazenamento dos dados; a facilitação da tarefa de desenvolvimento de sistemas; a melhor adequação da base de dados à realidade do dia-a-dia, entre outros. Para que tudo isso seja possível é necessário que o SGBD obedeça a alguns princípios básicos que são apresentados de forma resumida a seguir:

1. Independência de dados - consiste na capacidade de tornar as características físicas dos dados (como localização, estrutura e tamanho) transparentes para a aplicação. Com isso, os programas aplicativos passam a ter apenas uma visão lógica da base de dados, que não é influenciada por eventuais modificações na forma de armazenamento dos dados. Significa também que devem existir duas sublinguagens incorporadas ao banco de dados: uma Linguagem de Definição de Dados (LDD) e uma Linguagem de Manipulação de Dados (LMD).

2. Compartilhamento de dados - deve ser possível que, num determinado momento, mais de um usuário possa acessar os dados armazenados pelo SGBD. Um SGBD multi-

usuário deve permitir que múltiplos usuários acessem o banco de dados ao mesmo tempo. Este fator é essencial para que múltiplas aplicações integradas possam acessar o banco.

O SGBD multi-usuário deve manter o controle de concorrência para assegurar que o resultado de atualizações seja correto. Um banco de dados multi-usuário deve fornecer recursos para a construção de múltiplas visões dos dados.

3. Garantia de integridade de dados - apesar de permitir que os dados sejam compartilhados, o SGBD deve oferecer serviços que evitem que atualizações concorrentes tornem a base de dados inconsistente.

4. Garantia de segurança de dados - o SGBD deve oferecer meios que resguardem a base de dados nos casos de falha de programa, equipamento ou acidentes. Também deve ser possível que alterações indevidas feitas pelos usuários durante a operação dos aplicativos sejam desfeitas sem prejuízo da coerência dos dados. Considera-se igualmente como um item de segurança a ser proporcionado pelo software gerenciador, o controle dos níveis de acesso dos usuários aos dados do sistema. Um SGBD deve fornecer um subsistema de autorização e segurança, o qual é utilizado pelo DBA para criar "contas" e especificar as restrições destas contas; o controle de restrições se aplica tanto ao acesso aos dados quanto ao uso de softwares inerentes ao SGBD.

5. Relacionamento entre os dados - a implementação dos diversos relacionamentos previstos na base de dados deve ser controlada automaticamente pelo SGBD, externamente à aplicação. Um SGBD deve fornecer recursos para se representar uma grande variedade de relacionamentos entre os dados, bem como, recuperar e atualizar os dados de maneira prática e eficiente.

6. Controle da redundância de dados - devem ser fornecidos meios para que seja possível o controle da repetição de dados pelos diversos arquivos administrados pelo SGBD. Isso é conseguido por meio da centralização da definição dos dados num Dicionário de Dados ou Catálogo, que serve como base para a operação do SGBD. No processamento tradicional de arquivos, cada grupo de usuários deve manter seu próprio

conjunto de arquivos e dados. Desta forma, acaba ocorrendo redundâncias que prejudicam o sistema com problemas como:

- toda vez que for necessário atualizar um arquivo de um grupo, então todos os grupos devem ser atualizados para manter a integridade dos dados no ambiente como um todo;
- a redundância desnecessária de dados levam ao armazenamento excessivo de informações, ocupando espaço que poderia estar sendo utilizado com outras informações.

Estes princípios estão interligados entre se, uma falha em um deles pode refletir em todos, ou quase todos, os outros.

1.3.1 Modelos de SGBDs

Os produtos SGBDs podem ser classificados em 5 categorias principais: os de estrutura hierárquica; os de estrutura de rede; os relacionais; os orientados a objeto e os objeto-relacionais. Cada um desses modelos corresponde a um estágio na evolução do hardware disponível para as aplicações comerciais e no estado da arte na área da teoria de Banco de Dados e da Engenharia de Software e diferem entre si por 4 aspectos principais: a ordem cronológica em que surgiram; os tipos de elos (ligações) utilizados para implementar internamente os relacionamentos entre os dados; as modalidades de relacionamento suportadas e, finalmente, o tipo de linguagem utilizada para implementar as aplicações que interagem com a base de dados. Os modelos hierárquicos e de rede já são considerados ultrapassados; e a abordagem orientada a objetos ainda é uma tecnologia em maturação, embora promissora. Os SGBDs mais utilizados hoje em dia foram concebidos com base em um modelo matemático derivado da Teoria dos Conjuntos e que teve um grande desenvolvimento a partir da década de 70. Esse modelo é chamado Modelo Relacional e os SGBD que suportam tais conceitos são chamados de Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDR). São exemplos de SGBDR: MS-Access, ZIM, MS-SQL Server, Oracle, Sybase, Informix, CA-Ingres, IBM-DB2, Borland-Interbase, entre outros. Os SGBDs objeto-relacionais implementam conceitos orientados a objeto sobre mecanismos relacionais, e ganharam alguma importância nos últimos anos.

1.3.2 Quando não Utilizar um SGBD

Em algumas situações, o uso de um SGBD pode representar uma carga desnecessária aos custos, quando comparado à abordagem processamento de arquivos como por exemplo:

- alto investimento inicial na compra de software e hardware adicionais;
- generalidade que um SGBD fornece na definição e processamento de dados;
- sobrecarga na provisão de controle de segurança, controle de concorrência, recuperação e integração de funções.

Problemas adicionais podem surgir caso os projetistas de banco de dados ou os administradores de banco de dados não elaborem os projetos corretamente ou se as aplicações não são implementadas de forma apropriada. Se o DBA não administrar o banco de dados de forma apropriada, tanto a segurança quanto a integridade dos sistemas podem estar comprometidas. A sobrecarga causada pelo uso de um SGBD e a má administração justificam a utilização da abordagem de processamento tradicional de arquivos em casos como:

- o banco de dados e as aplicações são simples, bem definidas e não se espera mudanças no projeto;
- a necessidade de processamento em tempo real de certas aplicações, que são terrivelmente prejudicadas pela sobrecarga causada pelo uso de um SGBD;
- não haverá múltiplo acesso ao banco de dados.

2 Modelagem de Dados Utilizando o Modelo Entidade Relacionamento

O modelo Entidade-Relacionamento (E-R) é um modelo de dados conceitual de alto nível, cujos conceitos foram projetados para estar o mais próximo possível da visão que o usuário tem dos dados, não se preocupando em representar como estes dados estarão realmente armazenados. O modelo E-R é utilizado, principalmente, durante o processo de projeto de banco de dados.

2.1 Entidades e Atributos

O objeto básico tratado pelo modelo E-R é a "entidade", que pode ser definida como um objeto do mundo real, concreto (automóvel, casa, professor) ou abstrato (duplicata, venda, entre outros) e que possui existência independente. Cada entidade possui um conjunto particular de propriedades (características) que a descreve, chamado atributos.

Um atributo pode ser dividido em diversas sub-partes com significado independente entre si, recebendo o nome de atributo composto, como é o caso do atributo endereço, composto de rua, bairro, cidade, estado e CEP. Um atributo que não pode ser subdividido é chamado de atributo simples ou atômico, como por exemplo, o atributo CPF.

Cabe, aqui, uma observação sobre atributos atômicos, porque para nós, brasileiros, o atributo nome é atômico, já nos Estados Unidos, por exemplo, o atributo nome é composto, porque se divide em *first name, midle name e last name*, portanto, dizer que certos atributos são atômicos ou não, depende da interpretação de cada analista.

Os atributos que podem assumir apenas um determinado valor em uma determinada instância são denominados atributo simplesmente valorado, como é o caso do atributo salário de um empregado. Um atributo que pode assumir diversos valores em uma mesma instância é denominado multi valorado, como por exemplo, telefone de uma pessoa – supondo que ela tenha mais de um telefone e cores de um carro. Um atributo que é gerado a partir de outro é chamado de atributo derivado, são os atributos cujos valores não são armazenados no BD, por não haver necessidade de fazê-lo, como é o caso salário médio dos empregados, que pode ser derivado do atributo salário.

2.2 Tipos Entidades

Um banco de dados costuma conter grupos de objetos que são similares possuindo, os mesmos atributos, porém, cada entidade com seus próprios valores para cada atributo. Estes conjuntos de objetos similares definem um "tipo entidade". Cada tipo de entidade é identificado por seu nome e pelo conjunto de atributos, que definem suas propriedades. A descrição do tipo entidade é chamada de "esquema do tipo entidade", especificando o nome do tipo entidade, o nome de cada um de seus atributos e qualquer restrição que incida sobre as entidades.

2.3 Domínio

Cada atributo simples de um tipo entidade está associado com um conjunto de valores denominado "domínio", o qual especifica o conjunto de valores que podem ser designados para este determinado atributo, para cada entidade.

2.4 Atributos Chave

Uma restrição muito importante em uma entidade de um determinado tipo entidade é a "chave". Um tipo entidade possui um atributo cujos valores são distintos, para cada entidade individual. Este atributo é chamado "atributo chave" e seus valores podem ser utilizados para identificar cada entidade de forma única. Muitas vezes, uma chave pode ser formada pela composição de dois ou mais atributos. Uma entidade pode também ter mais de um atributo chave.

2.3 Tipos e Instância de Relacionamento

Além de conhecer detalhadamente os tipos entidade, é muito comum conhecer também os relacionamentos entre estes tipos entidades.

Um "tipo relacionamento" **R** entre *n* entidades $\mathbf{E_1}$, $\mathbf{E_2}$,... ,$\mathbf{E_n}$, é um conjunto de associações entre entidades deste tipo. Informalmente falando, cada instância de relacionamento $\mathbf{r_1}$ em **R** é uma associação de entidades, onde a associação inclui exatamente uma entidade de cada tipo entidade participante no tipo relacionamento. Isto

significa que, estas entidades estão relacionadas de alguma forma no mini-mundo. A Figura 2.1 mostra um exemplo entre dois tipos entidade, empregado e departamento, e o relacionamento entre eles, trabalha para. Repare que para cada relacionamento, participam apenas uma entidade de cada tipo entidade, porém, uma entidade pode participar de mais do que um relacionamento.

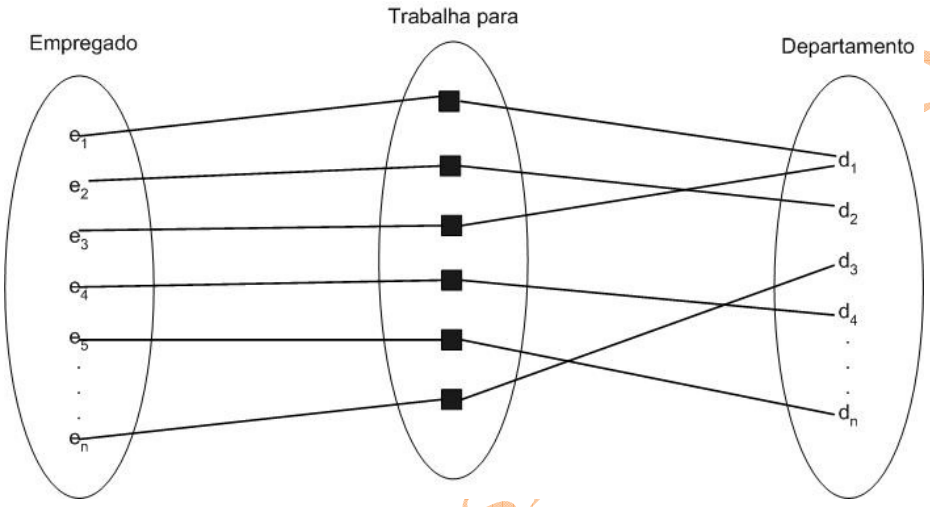

Figura 2.1: Exemplo de um relacionamento.

2.4 Grau de um Relacionamento

O "grau" de um tipo relacionamento é o número de tipos entidade que participam do tipo relacionamento. No exemplo da Figura 2.1, temos um relacionamento binário. Existem três tipos básicos de grau de relacionamento: binário, ternário e n-ário que se referm à existência de duas, três ou mais entidades envolvidas no fato que o relacionamento representa. O grau de um relacionamento é ilimitado, porém, a partir do grau 3 (ternário), a compreensão e a dificuldade de se desenvolver a relação corretamente se tornam extremamente complexas.

Podemos ter também relacionamento binário recursivo, que poderia ser tratado como relacionamento unário visto que apenas um tipo entidade participa do relacionamento, construído por meio de conexões com a mesma entidade, acontecendo como se fossem duas coisas diferentes se relacionando em nossa abstração, duas ocorrências da entidade se relacionando, se associando.

2.6 Relacionamentos como Atributos

Algumas vezes é conveniente pensar em um relacionamento como um atributo. Considere o exemplo da Figura 2.1. Podemos pensar departamento como sendo um atributo da entidade empregado, ou empregador. Como um atributo multivalorado da entidade departamento. Se a existência de uma entidade não for muito bem definida, talvez seja mais interessante para a coesão do modelo lógico, que ela seja representada como um atributo.

2.7 Nomes de Papéis e Relacionamentos Recursivos

Cada tipo entidade que participa de um tipo relacionamento desempenha um **papel** particular no relacionamento. O nome do papel representa: o papel que uma entidade de um tipo entidade participante desempenha no relacionamento. No exemplo da Figura 2.1, temos o papel empregado ou trabalhador para o tipo entidade, EMPREGADO e o papel departamento ou empregador para entidade DEPARTAMENTO. Nomes de papéis não são necessariamente importantes quando todas as entidades participantes desempenham papéis diferentes. Algumas vezes, o papel torna-se essencial para distinguir o significado de cada participação. Isto é muito comum em relacionamentos recursivos. A Figura 2.2 ilustra um exemplo de relacionamento recursivo.

No exemplo da Figura 2.2, temos relacionamento entre o tipo entidade EMPREGADO, onde um empregado pode supervisionar outro empregado e vice-versa.

2.8 Restrições em um tipo relacionamento

Geralmente, os tipos relacionamentos sofrem certas restrições que limitam as possíveis ligações das combinações das entidades participantes. Estas restrições são derivadas de restrições impostas pelo estado destas entidades no mini-mundo. A Figura 2.3 mostra um exemplo de restrição em um relacionamento recursivo.

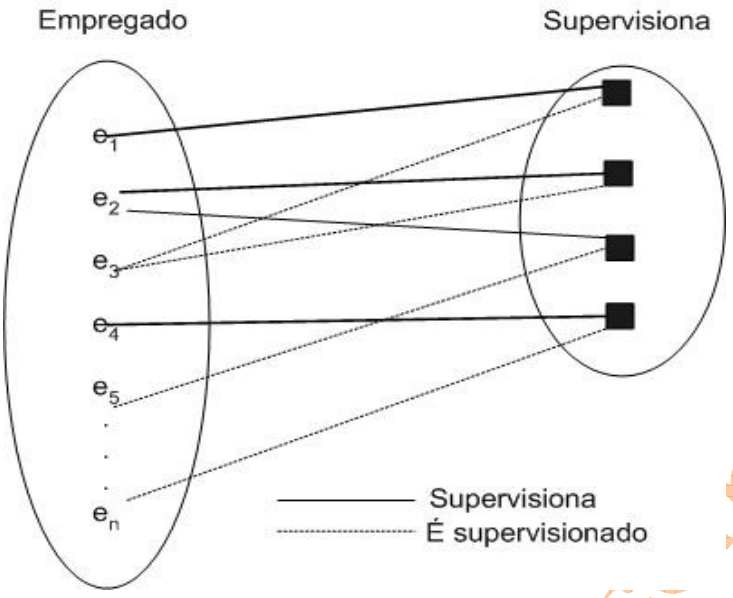

Figura 2.2: Exemplo de relacionamento recursivo.

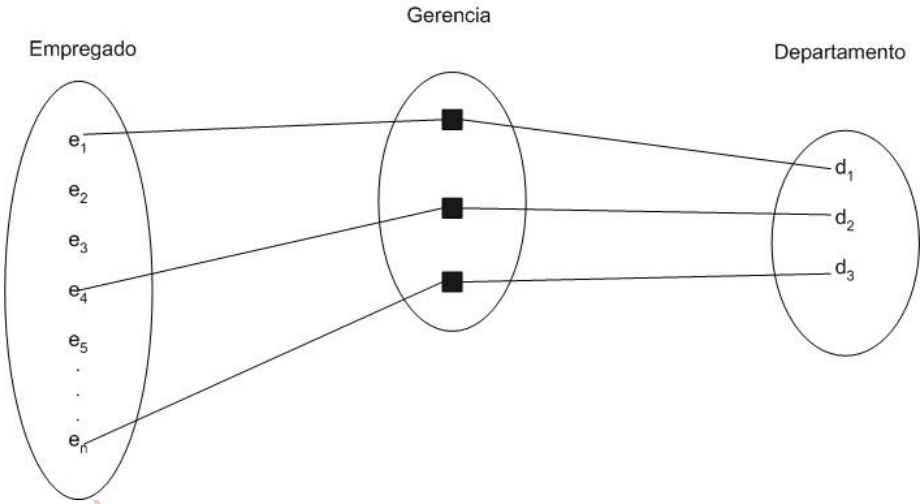

Figura 2.3: Relacionamento empregado gerencia departamento.

No exemplo da Figura 2.3, temos a seguinte situação: um empregado pode gerenciar apenas um departamento, enquanto que um departamento pode ser gerenciado por apenas um empregado. A este tipo de restrição, nós chamamos **cardinalidade.** A cardinalidade indica o número de relacionamentos, do qual uma entidade pode participar. Ela pode ser: 1:1; 1:N**;** N:N. No exemplo da Figura 2.3, a cardinalidade é 1:1, pois cada entidade empregado pode gerenciar apenas um departamento e vice-versa; no exemplo da Figura

2.1, no relacionamento EMPREGADO Trabalha para DEPARTAMENTO, O relacionamento é 1:N, pois um empregado pode trabalhar em apenas um departamento, enquanto um departamento pode possuir vários empregados; na Figura 2.4 temos um exemplo de um relacionamento com cardinalidade N:N.

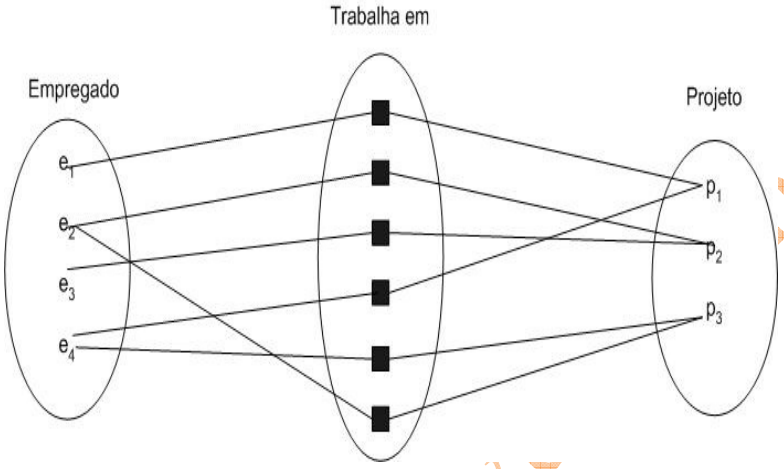

Figura 2.4: Relacionamento N:N.

No exemplo da Figura 2.4, temos que um empregado pode trabalhar em vários projetos, enquanto que um projeto pode ter vários empregados trabalhando.

Outra restrição muito importante é a participação**.** Ela define a existência de uma entidade através do relacionamento, podendo ser parcial ou total, veja o exemplo da Figura 2.3. A participação do empregado é parcial, pois nem todo empregado gerencia um departamento, porém a participação do departamento neste relacionamento é total**,** já que todo departamento precisa ser gerenciado por um empregado. Desta forma, todas ocorrências do tipo entidade DEPARTAMENTO precisam participar do relacionamento, mas nem todas ocorrências do tipo entidade EMPREGADO, precisam participar do relacionamento.

Já no exemplo da Figura 2.1, ambas as participações são totais, pois todo empregado precisa trabalhar em um departamento e todo departamento tem que ter empregados trabalhando nele.

Estas restrições são chamadas de restrições estruturais**.** Algumas vezes, torna-se necessário armazenar um atributo no tipo relacionamento, veja exemplo da figura 2.3. Pode-se querer saber em que dia o empregado passou a gerenciar o departamento, é difícil estabelecer a qual tipo entidade pertence atributo, pois isto é definido apenas pela

existência do relacionamento. Quando temos relacionamentos com cardinalidade 1:1, podemos colocar o atributo em uma das entidades, de preferência, em uma cujo tipo entidade tenha participação total. Neste caso, o atributo poderia ir para o tipo entidade departamento. Isto porque, nem todo empregado participará do relacionamento. Caso a cardinalidade seja 1:N, então podemos colocar o atributo no tipo entidade com participação N. Porém, se a cardinalidade for N:M então o atributo deverá ficar no tipo relação, como seria o caso do relacionamento ilustrado pela Figura 2.4, caso queiramos armazenar quantas horas cada empregado trabalhou em cada projeto, então este deverá ser um atributo do relacionamento, "trabalha em".

2.9 Tipos Entidades Fracas

Alguns tipos entidade podem não ter um atributo chave por si só, isto implica que não poderemos distinguir algumas entidades, porque as combinações dos valores de seus atributos podem ser idênticas. Estes tipos entidades são chamados entidades fracas**.** Elas precisam estar relacionadas com uma entidade pertencente ao tipo entidade proprietária, entidade forte. Este relacionamento é denominado relacionamento identificador**.** Veja o exemplo da Figura 2.5.

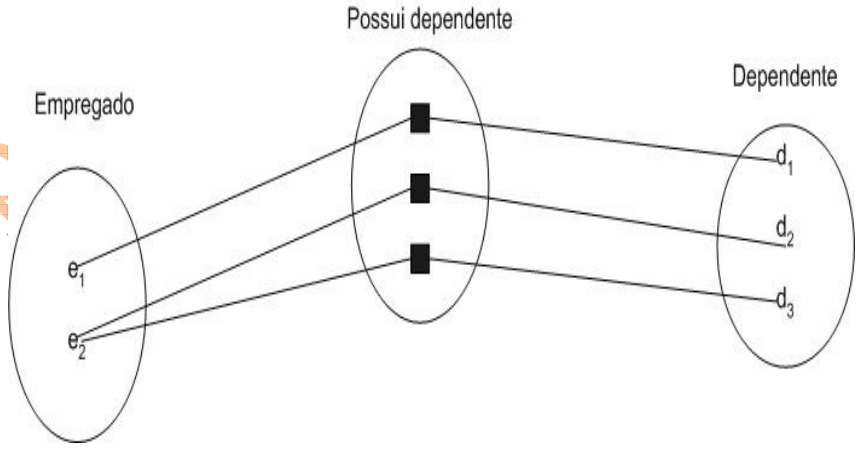

Figura 2.5: Relacionamento com entidade fraca.

O tipo entidade DEPENDENTE é uma entidade fraca, pois não possui um atributo chave, identificador. O tipo entidade EMPREGADO não é uma entidade fraca, pois

possui um atributo para identificação – atributo chave, o número do CPF de um empregado. Porém, um dependente de cinco anos de idade não possui, necessariamente, um documento. Desta forma, esta entidade é um tipo entidade fraca. Um tipo entidade fraca possui uma chave parcial, que juntamente com a chave primária da entidade proprietária, forma uma chave primária composta. Neste exemplo, a chave primária do empregado é o CPF. A chave parcial do dependente é o seu nome, pois dois irmãos não podem ter o mesmo nome, falando logicamente. Desta forma, a chave primária dês entidade fica sendo o CPF do pai, ou da mãe, mais o nome do dependente.

Todos os exemplos vistos anteriormente foram relacionamentos binários, ou seja, entre dois tipos entidades diferentes ou recursivos. Todavia, o modelo entidade-relacionamento não se restringe apenas a relacionamentos binários. O número de entidades que participam de um tipo relacionamento é irrestrito e armazenam muito mais informações do que diversos relacionamentos binários. Considere o seguinte exemplo: um fornecedor pode fornecer uma ou mais peças para um ou mais projeto. Se criarmos dois relacionamentos binários, não teremos estas informações de forma completa como se criarmos um relacionamento ternário, como ilustra a Figura 2.6.

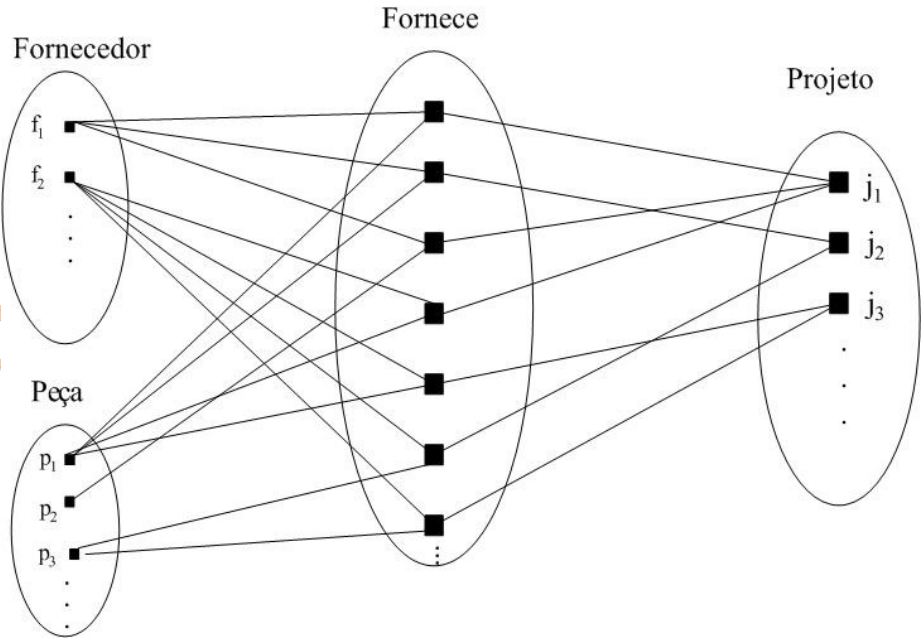

Figura 2.6: Exemplo de relacionamento ternário.

2.10 Diagrama Entidade Relacionamento

O Diagrama Entidade Relacionamento é composto por um conjunto de objetos gráficos que visa representar todos os objetos do modelo Entidade Relacionamento tais como: atributos, atributos chaves, relacionamentos, restrições estruturais, entre outros.

O diagrama ER fornece uma visão lógica do banco de dados, fornecendo um conceito mais generalizado de como estão estruturados os dados de um sistema. Os objetos que compõem o diagrama ER estão listados a seguir, na figura 2.7.

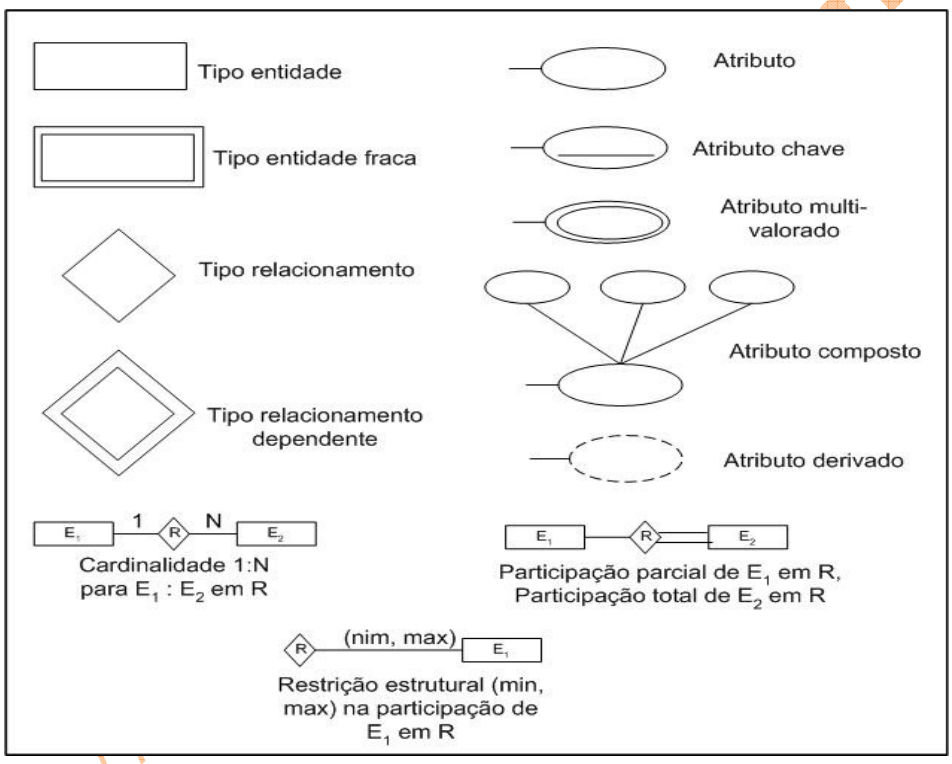

Figura 2.7: Componentes do diagrama entidade-relacionamento.

2.11 Extensões do Modelo Entidade Relacionamento

Os conceitos do modelo Entidade Relacionamento discutidos anteriormente são suficientes para representar conceitual e logicamente à maioria as aplicações de banco de dados, porém, com o surgimento de novas aplicações, surgiu também a necessidade de novas semânticas para a modelagem de informações mais complexas. O modelo Entidade Relacionamento Extendido (ERE) visa fornecer esta semântica para permitir a

representação de informações complexas. É importante frisar que embora o modelo ERE trate classes e subclasses, ele não possui a mesma semântica de um modelo orientado a objetos.

O modelo ERE engloba todos os conceitos do modelo E-R mais os conceitos de subclasse, superclasse, generalização, especialização, e o conceito de herança de atributos.

2.11.1 Subtipos, Supertipos e Especialização

O primeiro conceito do modelo ERE que será abordado é o de subclasse de um tipo entidade. Como visto anteriormente, um tipo entidade é utilizado para representar um conjunto de objetos do mesmo tipo. Em muitos casos, um tipo entidade possui diversos subgrupos adicionais de objetos que são significativas e precisam ser representadas explicitamente devido ao seu significado à aplicação de banco de dados. Para melhor entendimento desses conceitos, consideremos o seguinte exemplo:

Para um banco de dados da empresa "MERE" temos o tipo entidade empregado, o qual possui as seguintes características: nome, RG, CPF, número funcional, endereço completo (rua, número, complemento, CEP, bairro, cidade), sexo, data de nascimento e telefone (DDD e número); caso o(a) funcionário(a) seja um(a) engenheiro(a), secretário(a), então deseja-se armazenar as seguintes informações: número do CREA e especialidade (civil, mecânico, eletro/eletrônico); caso seja um(a) secretário(a), então deseja-se armazenar as seguintes informações: qualificação (bi ou tri língue) e os idiomas no qual possui fluência verbal e escrita.

Se as informações número do CREA, especialidade, tipo e idiomas forem representados diretamente no tipo entidade, empregado, estará representando informações de um conjunto limitado de objetos empregados, para todos os funcionários da empresa. Neste caso, podemos criar dois subtipos do tipo entidade, empregado: engenheiro e secretária**,** as quais irão conter as informações acima citadas. Além disto, engenheiro e secretária podem ter relacionamentos específicos.

Uma entidade não pode existir meramente como componente de um subtipo. Antes de ser componente de um subtipo, uma entidade deve ser componente de um supertipo. Isto leva ao conceito de herança de atributos**;** ou seja, o subtipo herda todos os atributos

do supertipo, porque um objeto do subtipo apresenta as mesmas características de um objeto da entidade do supertipo. Um subtipo pode herdar atributos de supertipos diferentes. A Figura 2.8 ilustra o exemplo do banco de dados da empresa "MERE", usando supertipo e subtipo.

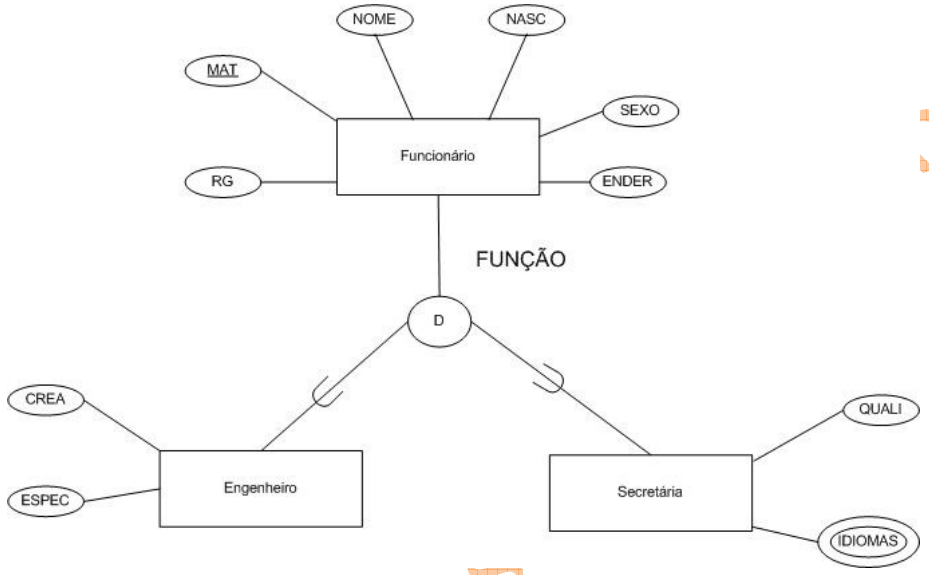

Figura 2.8: Representação de supertipos e subtipos.

2.11.2 Especialização

Especialização é o processo de definição de um conjunto de classes de um tipo entidade; este tipo é chamado de supertipo da especialização. O conjunto de subtipos forma-se baseado em alguma característica que distingue as entidades dentre outras. No exemplo da Figura 2.8, temos uma especialização, a qual pode chamar de função. Vejamos agora outro exemplo, ilustrado pela Figura 2.9, temos a entidade, funcionário e duas especializações.

Como visto anteriormente, um subtipo pode ter relacionamentos específicos com outras entidades ou com a própria entidade que é o seu supertipo. A figura 2.10 ilustra isto.

O processo de especialização permite:

- definir um conjunto de subtipos de um tipo entidade;
- associar atributos específicos adicionais para cada subtipo;
- estabelecer tipos relacionados específicos entre subtipos e outros tipos entidades.

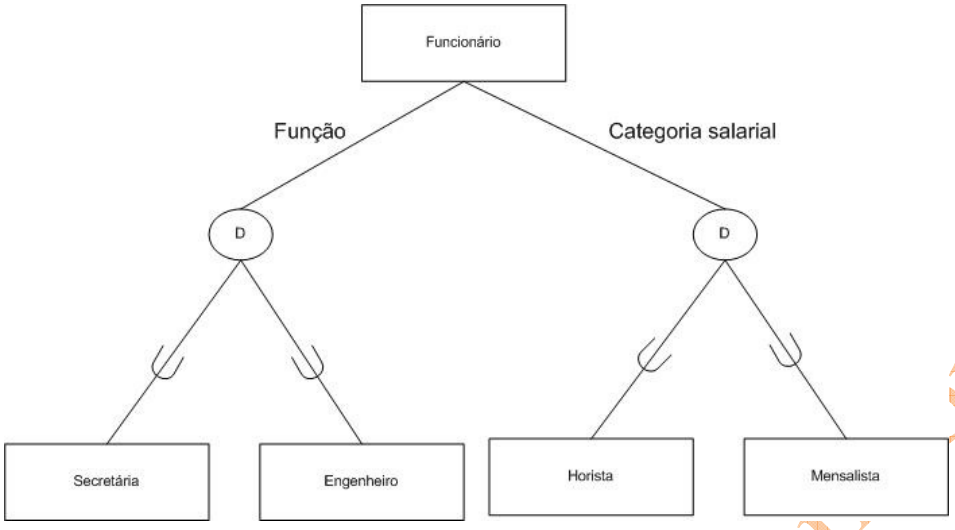

Figura 2.9: Representação de duas especializações para funcionário.

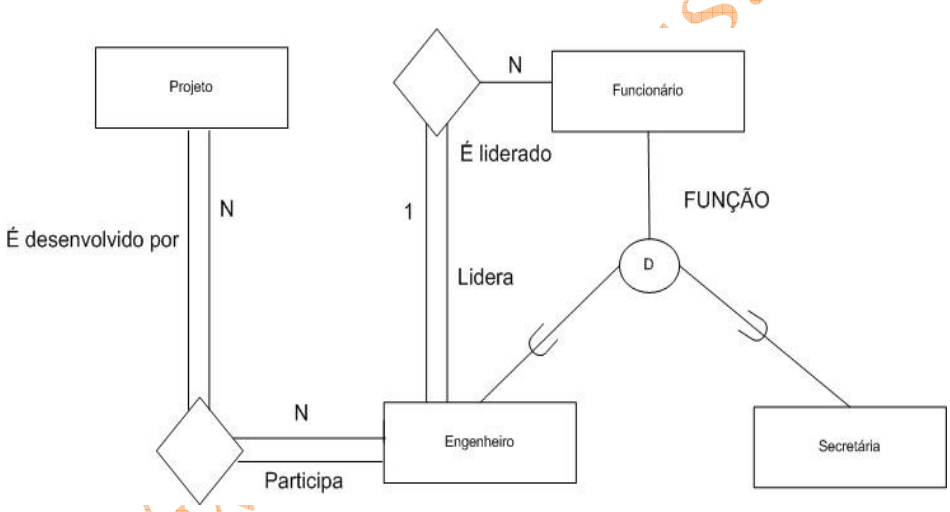

Figura 2.10: Relacionamento entre subtipos e entidades.

2.11.3 Generalização

A generalização pode ser pensada como um processo de abstração reverso ao da especialização, no qual são suprimidas as diferenças entre diversos tipos entidades, identificando suas características comuns e generalizando estas entidades em um supertipo. A figura 2.11 mostra um exemplo de generalização. Como pode ser visto, temos nome e RG comuns às entidades engenheiro e secretária, o que nos leva a uma generalização conforme ilustrada pela Figura 2.12.

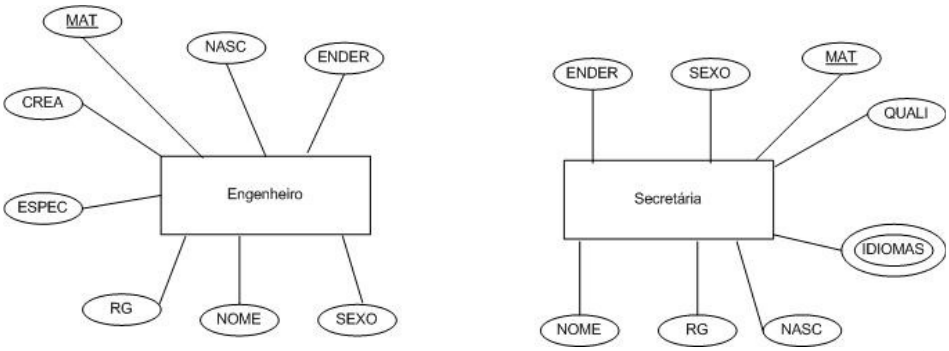

Figura 2.11: Tipos entidades engenheiro e secretária.

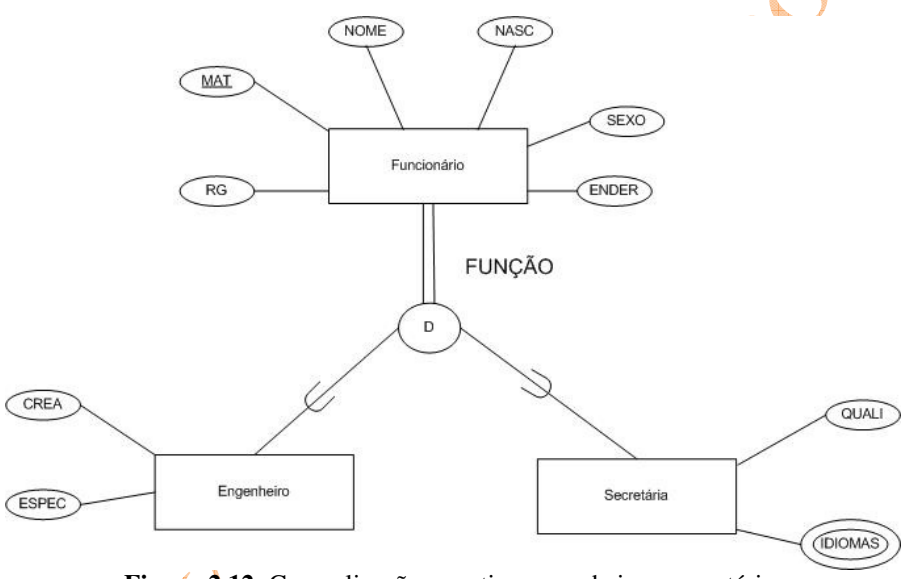

Figura 2.12: Generalização para tipo engenheiro e secretária.

É importante destacar que existe diferença semântica entre a especialização e a generalização. Na especialização, podemos notar que a ligação entre o supertipo e os subtipos é feita através de um traço simples, indicando participação parcial por parte do supertipo. Analisando o exemplo da Figura 2.8, é observado que um funcionário não é obrigado a ser um engenheiro ou uma secretária. Na generalização, podemos notar que a ligação entre o supertipo e os subtipos é feita através de um traço duplo, indicando participação total por parte do supertipo. Analisando o exemplo da Figura 2.12, observa-se que um funcionário é obrigado a ser um engenheiro ou uma secretária.

A letra d dentro do círculo que especifica uma especialização ou uma generalização significa disjunção**.** Uma disjunção em uma especialização ou generalização indica que uma entidade do tipo entidade que representa a superclasse, pode assumir apenas um papel dentro dela. Analisando o exemplo da Figura 2.9, temos duas especializações para o supertipo funcionário, as quais são restringidas através de uma disjunção. Neste caso, um funcionário pode ser um engenheiro ou uma secretária e pode ser horista ou mensalista.

Além da disjunção podemos ter um "overlap", representado pela letra o**.** No caso do "overlap", uma entidade de um supertipo pode ser membro de mais que um subtipo em uma especialização ou generalização. Analisando a generalização ilustrada pela Figura 2.13, podemos dizer que uma peça fabricada em uma tornearia pode ser manufaturada ou torneada ou ainda, pode ser manufaturada e torneada ao mesmo tempo.

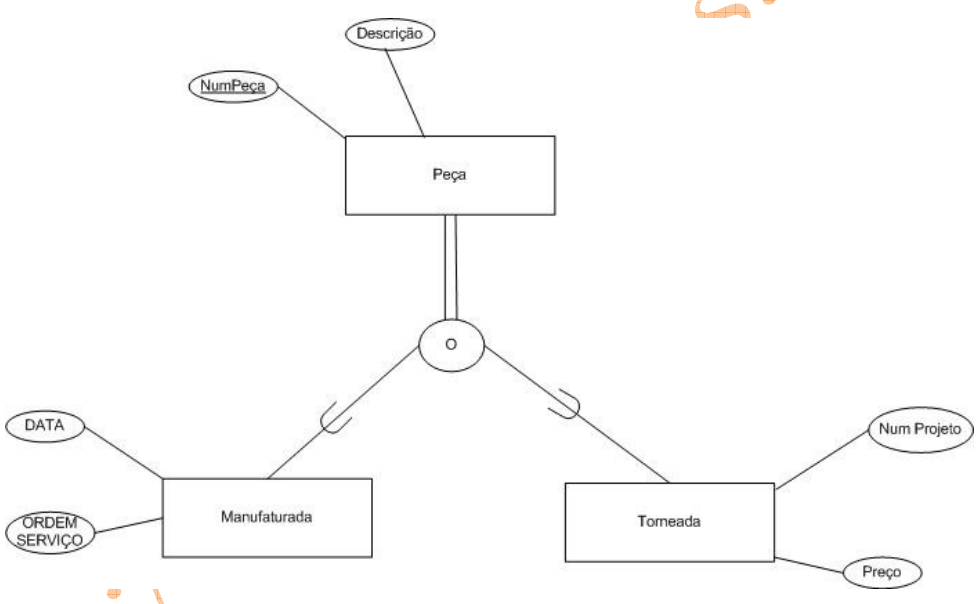

Figura 2.13: Generalização com overlap.

3 O Modelo Relacional

O modelo relacional foi criado por Codd em1970 e tem por finalidade representar os dados como uma coleção de relações, onde cada relação é representada por uma tabela, ou falando de uma forma mais direta, um arquivo. Porém, um arquivo é mais restrito que uma tabela. Toda tabela pode ser considerada um arquivo, porém, nem todo arquivo pode ser considerado uma tabela.

Quando uma relação é pensada como uma tabela de valores, cada linha nesta tabela representa uma coleção de dados relacionados. Estes valores podem ser interpretados como fatos descrevendo uma instância de uma entidade ou de um relacionamento. O nome da tabela e das colunas são utilizados para facilitar a interpretação dos valores armazenados em cada linha. Todos os valores em uma coluna são necessariamente do mesmo tipo.

Na terminologia do modelo relacional, cada tabela é chamada de relação; uma linha de uma tabela é chamada de tupla; o nome de cada coluna é chamado de atributo; o tipo de dado que descreve cada coluna é chamado de domínio**.**

3.1 Domínio de uma relação

Um domínio D é um conjunto de valores atômicos, sendo que por atômico, podemos compreender que cada valor do domínio é indivisível. Durante a especificação do domínio é importante destacar o tipo, o tamanho e a faixa do atributo que está sendo especificado. A Tabela 3.1, mostra um exemplo de domínios de atributo.

Tabela 3.1: Domínio de atributos.

Atributo	**Tipo**	**Tamanho**	**Faixa**
CPF	Numérico	11,0	00000000000-99999999999
Nome	Caractere	45	a-z, A-Z
Salário	Numérico	6,2	000000,00-999999,99

3.2 Esquema de uma Relação

Um esquema de relação R, denotado por $R(A_1,A_2, ..., A_n)$, onde cada atributo A_i é o nome do papel desempenhado por uma domínio D no esquema relação R**,** onde D é o chamado domínio de A_i e é denotado por um dom(A_i). O grau de uma relação R é o número de atributos presentes em seu esquema de relação.

3.3 Instância de um Esquema

A instância r de um esquema relação denotado por r(R) é um conjunto de n-tuplas $r=[t_1,t_2,..., t_n]$ onde os valores de $[t_1,t_2,..., t_n]$ devem estar contidos no domínio D. O valor nulo também pode fazer parte do domínio de um atributo e representa um valor não conhecido para uma determinada tupla.

3.4 Atributo Chave de uma Relação

Uma relação pode ser definida como um conjunto de tuplas distintas. Isto implica que a combinação dos valores dos atributos em uma tupla não pode se repetir na mesma tabela. Existirá sempre um subconjunto de atributos em uma tabela que garantem que não haverá valores repetidos para as suas diversas tuplas, garantindo que $t_1[SC] \neq t_2[SC]$. *SC* é chamada de superchave de um esquema de relação. Toda relação possui ao menos uma superchave – o conjunto de todos os seus atributos. Uma chave C de um esquema de relação R com a propriedade adicional que removendo qualquer atributo A de K, resta ainda um conjunto de atributos K' que não é uma superchave de R**.** Uma chave é uma superchave da qual não se pode extrair atributos. Por exemplo, o conjunto: (RA, Nome, Endereço) é uma superchave para estudante, porém, não é uma chave pois se tirarmos o campo Endereço continuaremos a ter uma superchave. Já o conjunto (Nome da Revista, Volume, Nº da Revista) é uma superchave e uma chave, pois se retirarmos qualquer um dos atributos, deixaremos de ter uma superchave, ou seja, (Nome da Revista, Volume) não identifica uma única tupla. Em outras palavras, uma superchave é uma chave composta**,** ou seja, uma chave formada por mais que um atributo. Veja o exemplo apresentado na Tabela 3.2.

Tabela 3.2: Tabela com os dados de dependentes de empregados de uma empresa.

CPF Responsável	Nome Dependente	Data Nascimento	Relação	Sexo
111111111-11	Karla	10/08/1991	Filha	Feminino
222222222-22	Maria	07/03/1996	Filha	Feminino
333333333-33	Mônica	14/02/1976	Cônjuge	Feminino
444444444-44	João	10/02/95	Filho	Masculino
555555555-55	Silvia	01/05/90	Filha	Feminino

Quando uma relação possui mais que uma chave (não confundir com chave composta) – como, por exemplo, RG e CPF para empregados – cada uma destas chaves são chamadas de chaves candidatas. Uma destas chaves candidatas devem ser escolhida como chave primária**.**

Uma chave estrangeira CE de uma tabela R_1 em R_2 ou vice-versa, especifica um relacionamento entre as tabelas R_1 e R_2**.** As tabelas 3.3 e 3.4 são exemplos de tabelas relacionadas, a Tabela 3.4 tem o atributo NumDept que é uma chave estrangeira, pois ela é a responsável pela ligação entre empregado e departamento. Pode-se observar na Tabela 3.4, que existem duas chaves candidatas, CPF e RG, sendo que a chave escolhida como primária, é o atributo CPF.

Tabela 3.3: Tabela de Departamentos.

Nome	NumDept
Contabilidade	1
Engenharia Civil	2
Engenharia Elétrica	3

Tabela 3.4: Tabela de empregados.

Nome	RG	CPF	NumDept	RG Supervisor	Salário
João Luiz	10101010	111111111-11	1	**NULO**	3.000,00
Fernando	20202020	222222222-22	2	10101010	2.500,00
Ricardo	30303030	333333333-33	3	10101010	2.300,00
Jorge	40404040	444444444-44	3	20202020	4.200,00
Renato	50505050	555555555-55	1	20202020	1.300,00

3.5 Normalização

A Normalização é um processo formal passo a passo que examina os atributos de uma entidade, com o objetivo de evitar anomalias observadas na inclusão, exclusão e alteração de Tuplas específicas.

Esse processo causa a simplificação dos atributos dentro da respectiva tupla, eliminando grupos repetitivos, dependências parciais de chaves concatenadas, dependências transitivas, dados redundantes e dependências multivaloradas.

Para se atingir esse estágio, é necessário que as tuplas sejam analisadas de forma a verificar se seus atributos apresentam relações não-normalizadas, submetendo-os aos conceitos subseqüentes de primeira, segunda, terceira, quarta e quinta forma normal.

3.5.1 Primeira Forma Normal (1FN)

Uma tupla está na primeira forma normal se e somente se o relacionamento entre a chave primária e os atributos não-chaves deve ser unívoco. Aplicar a primeira forma normal significa retirar os elementos repetitivos. Seja o exemplo abaixo, de uma tupla não normalizada:

NotaFiscal(nr_NF, cd_NatOperacao, cd_Transportadora, dt_Emissao, cd_Cliente, nm_Cliente, nm_Rua, nm_Cidade, cd_UF, nr_CNPJ, nr_IE, nr_Telefone, cd_Produto, cd_Unidade, qt_Vendida, nm_produto, cd_Tributacao, vl_Unitario, vl_TotalItem, vl_TotalNF).

Após aplicar a primeira forma normal, obtém-se as seguintes tuplas:

NotaFiscal(nr_NF, cd_NatOperacao, cd_Transportadora, dt_Emissao, cd_Cliente, nm_Cliente, nm_Rua, nm_Cidade, cd_UF, nr_CNPJ, nr_IE, nr_Telefone, vl_TotalNF)

NotaFiscalItem(nr_NF, cd_Produto, cd_Unidade, qt_Vendida, nm_produto, cd_Tributacao, vl_Unitario, vl_TotalItem)

3.5.2 Segunda Forma Normal (2FN)

Consiste em retirarmos das estruturas de dados que possuem chaves compostas, todos os dados que são funcionalmente dependentes de somente alguma parte dessa chave. Elimina-se a dependência parcial, ou seja, os atributos não-chaves devem depender integralmente a chave primária composta. Uma tupla está na segunda forma normal, se estiver na primeira forma normal e não possuir campos que sejam funcionalmente dependentes de parte da chave. Seja o exemplo abaixo:

NotaFiscal (nr_NF, cd_NatOperacao, cd_Transportadora, dt_Emissao, cd_Cliente, nm_Cliente, nm_Rua, nm_Cidade, cd_UF, nr_CNPJ, nr_IE, nr_Telefone, vl_TotalNF)

NotaFiscalItem (nr_NF, cd_Produto, qt_Vendida, vl_Unitario, vl_TotalItem)

Produto (cd_Produto, nm_produto, cd_Unidade, cd_Tributacao)

3.5.3 Terceira Forma Normal (3FN)

A terceira forma normal determina que não deve existir atributos com dependência funcional transitiva em uma tabela, porque podem provocar anomalias de inclusão, manutenção e deleção. A aplicação da terceira forma normal consiste em retirar das estruturas, os campos que são funcionalmente dependentes de outros campos que não são chaves. Uma tupla está na terceira forma normal se ela estiver na segunda e não possuir campos dependentes de outros campos não chaves. Seja o exemplo abaixo:

NotaFiscal(nr_NF, cd_NatOperacao, cd_Transportadora, dt_Emissao, cd_Cliente, nm_Cliente, nm_Rua, nm_Cidade, cd_UF, nr_CNPJ, nr_IE, nr_Telefone, vl_TotalNF)

Podemos observar na tupla acima que os dados do cliente depende funcionalmente da coluna código do cliente, a qual não faz parte da chave primária da tupla, logo tem dependência funcional transitiva. Após aplicar a 3FN, obtém-se as seguintes tuplas:

NotaFiscal(nr_NF, cd_NatOperacao, cd_Transportadora, dt_Emissao, cd_Cliente, vl_TotalNF)

Cliente(cd_Cliente, nm_Cliente, nm_Rua, nm_Cidade, cd_UF, nr_CNPJ, nr_IE, nr_Telefone)

Existem outras formas normais que podem estudadas em livros especializados de banco de dados, mas geralmente a aplicação das três primeiras formas normais são suficientes para a criação de um modelo de banco de dados.

4 ÁLGEBRA RELACIONAL

A álgebra relacional é uma linguagem de consulta procedural. Ela consiste em um conjunto de operações que tornam uma ou duas relações como entrada e produzem uma nova relação como resultado.

As operações fundamentais na álgebra relacional são: selecionar, projetar, renomear, (unárias) - produto cartesiano, união e diferença de conjuntos (binárias).

Além das operações fundamentais, existem outras operações: interseção de conjuntos, ligação natural, dentre outras, que são definidas em termos das operações fundamentais.

4.1 Operação Selecionar

Seleciona tuplas que satisfazem um dado predicado (condição), descartando as outras. Usamos a letra minúscula grega sigma σ para representar a seleção. O predicado aparece subscrito em s. A relação argumento aparece entre parênteses seguindo o σ.

A forma geral de uma seleção é:

σ <condições> (RELAÇÃO)

As comparações são permitidas usando =, $\neq$,<, $\leq$, > e $\geq$ e os conectivos e (^) e ou ($\vee$) e que envolvam apenas os atributos existentes na RELAÇÃO.

Exemplo 1

Selecione as tuplas da relação empréstimo onde o código da agência é 0662.

σ agencia-cod=0662(EMPRESTIMO)

Exemplo 2

Encontrar todas as tuplas onde a quantia emprestada seja maior que 1200.

σ quantia >1200(EMPRESTIMO)

4.2 Operação projetar

A operação projetar é uma operação unária que retorna sua relação argumento, com certas colunas deixadas de fora. A projeção é representada pela letra grega (π).

A forma geral é:

π <atributos da relação> (RELAÇÃO)

Exemplo 1

Mostre o código dos clientes e das agências nas quais eles tomaram empréstimos.

π agencia-cod,cliente-cod(EMPRESTIMO)

Exemplo 2

Mostre os clientes que moram em "Aracruz".

Devemos fazer uma seleção de todos os clientes que moram em Aracruz, em seguida projetar o código destes clientes.

π cliente-nome (π cliente-cidade="Aracruz" (CLIENTE))

Observe que neste caso, se houver clientes com o mesmo nome, apenas um nome aparecerá na tabela resposta!

4.3 Operação produto cartesiano

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

A forma geral é:

RELAÇÃO1 X RELAÇÃO2

Exemplo 1

Para selecionarmos todos os nomes dos clientes que possuam empréstimo na agência cujo código é 0662, escrevemos:

π cliente-nome (σ agencia-cod=0662 ^ Emprestimo.cliente-cod = CLIENTE. cliente-cod (EMPRESTIMO X CLIENTE))

Deve-se tomar cuidado com ambigüidades nos nomes dos atributos.

Exemplo 2

Esquema LANCHE DO PEDRÃO:

CLIENTE = (codigo, nome)

FIADO = (codigo, valor)

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

π codigo,nome,valor (σ CLIENTE.codigo=FIADO.codigo (Cliente x Fiado))

Supondo as tabelas CLIENTE e FIADO abaixo, podemos representar das etapas desta consulta conforme Figura 4.1.

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

$\sigma_{CLIENTE.codigo=FIADO.codigo}$ (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

Figura 4.1: Ilustração de uma consulta usando produto cartesiano.

4.4 Operação Renomear

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. A forma geral é:

ρ <novo nome> (RELAÇÃO2)

Outra forma de renomear uma RELAÇÃO é atribui-la a uma outra. Isto é feito com o símbolo $\leftarrow$.

RELAÇÃO2 $\leftarrow$ RELAÇÃO1

Exemplo

Encontre todos os clientes que moram na mesma rua e cidade que João.

Podemos obter a rua e a cidade de João da seguinte forma:

t $\leftarrow$ π rua, cidade (σ cliente_nome="João" (CLIENTE))

Entretanto, para encontrar outros clientes com esta rua e cidade, devemos referir-nos à relação Cliente pela segunda vez. Perceba que se for inserida novamente uma relação clientes na consulta gerará ambiguidade. Por isso, devemos renomeá-la.

ρ CLIENTE2 (CLIENTE)

Obtemos então:

π rua, cidade (σ CLIENTE.cidade = CLIENTE2.cidade ^ CLIENTE.rua = CLIENTE2.rua (t X ρ CLIENTE2 (CLIENTE))

O resultado desta consulta é ilustrado pela Figura 4.2.

CLIENTE

clinte-cod	cliente-nome	rua	cidade
1	Maria	Rua 1	Cariacica
2	João	Rua 2	Vitória
3	Ana	Rua 3	Cariacica
4	José	Rua 2	Vitória

CLIENTE X CLIENTE2

clinte-cod	cliente-nome	rua	cidade	cliente2-cod	cliente2-nome	rua	Cidade
1	Maria	Rua 1	Cariacica	1	Maria	Rua 1	Cariacica
1	Maria	Rua 1	Cariacica	2	João	Rua 2	Vitória
1	Maria	Rua 1	Cariacica	3	Ana	Rua 3	Cariacica
1	Maria	Rua 1	Cariacica	4	José	Rua 2	Vitória
2	João	Rua 2	Vitória	1	Maria	Rua 1	Cariacica
2	João	Rua 2	Vitória	2	João	Rua 2	Vitória
2	João	Rua 2	Vitória	3	Ana	Rua 3	Cariacica
2	João	Rua 2	Vitória	4	José	Rua 2	Vitória
3	Ana	Rua 3	Cariacica	1	Maria	Rua 1	Cariacica
3	Ana	Rua 3	Cariacica	2	João	Rua 2	Vitória
3	Ana	Rua 3	Cariacica	3	Ana	Rua 3	Cariacica
3	Ana	Rua 3	Cariacica	4	José	Rua 2	Vitória
4	José	Rua 2	Vitória	1	Maria	Rua 1	Cariacica
4	José	Rua 2	Vitória	2	João	Rua 2	Vitória
4	José	Rua 2	Vitória	3	Ana	Rua 3	Cariacica
4	José	Rua 2	Vitória	4	José	Rua 2	Vitória

Figura 4.2: Resultado da consulta usando renoneação.

4.5 Operação União (binária)

A operação binária união é representada, como na teoria dos conjuntos, pelo símbolo $\cup$.

A forma geral é:

RELAÇÃO1 $\cup$ RELAÇÃO2

Suponha que quiséssemos saber todas as pessoas que possuem CONTA ou EMPRÉSTIMO numa determinada agência. Com os recursos que temos até agora, não seria possível conseguirmos tal informação. Nessa situação, deveríamos fazer a união de todos que possuem conta com todos que possuem empréstimos nessa agência.

Exemplo

Selecionar todos os clientes que possuam conta ou empréstimos ou ambos na agência 051.

π cliente-nome (σ agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod(CONTA X CLIENTE))

$\cup$

π cliente-nome (σ agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod(EMPRESTIMO X CLIENTE))

Uma vez que as relações são conjuntos, as linhas duplicadas são eliminadas. Para uma operação União r $\cup$ s ser válida, necessitamos que duas condições devem ser cumpridas:

- As relações r e s precisam ter a mesma paridade. Isto é, elas precisam ter o mesmo número de atributos;
- Os domínios do i-ésimo atributo de r e do i-ésimo atributo de s devem ser os mesmos.

4.6 A operação diferença de conjuntos

A operação diferença de conjuntos permite-nos encontrar tuplas que estão em uma relação e não em outra. A expressão r - s resulta em uma relação que contém todas as tuplas que estão em r e não em s.

A forma geral é:

RELAÇÃO1 **-** RELAÇÃO2

Exemplo 1

Encontrar os clientes que possuem uma conta mas não possuem um empréstimo na agência 051.

π cliente-nome (σ agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod (CONTA X CLIENTE))

-

π cliente-nome (σ agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod (EMPRESTIMO X CLIENTE))

Observação: A operação diferença não é comutativa. Em geral r – s $\neq$ s – r.

Exemplo 2

Suponha o seguinte esquema:

Esquema LANCHE DO PEDRÃO:

CLIENTE = (codigo, nome)

FIADO = (codigo, valor)

Mostre a conta de fiado mais alta.

MENORES $\leftarrow$ π valor (σ valor < AUX.valor (FIADO X ρ AUX (FIADO))

MAIOR $\leftarrow$ π valor (FIADO) - MENORES

O resultado é ilustrado pela Figura 4.3.

CLIENTE

codigo	nome
1	Chico
2	Ana

FIADO

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

FIADO X ρ_{AUX} (FIADO)

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

Figura 4.3: Resultado da seleção da conta de fiado mais alta.

4.7 Operação interseção de conjuntos

É representado pelo símbolo $\cap$.

A forma geral é:

RELAÇÃO1 $\cap$ RELAÇÃO2

Exemplo

Encontrar todos os clientes com um empréstimo e uma conta na agência "051".

π cliente-nome (σ agencia-cod="051" ^ CONTA.cliente-cod = CLIENTE.cliente-cod(CONTA X CLIENTE))

$\cap$

π cliente-nome (σ agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod(EMPRESTIMO X CLIENTE))

A operação Interseção de conjuntos pode ser expressa em função das operações fundamentais da seguinte forma: r $\cap$ s = r - (r – s).

4.8 Operação Ligação natural

A ligação natural é uma operação binária que permite combinar certas seleções e um produto cartesiano em uma única operação. É representada pelo símbolo |X|.

A operação ligação natural forma um produto cartesiano de seus dois argumentos, faz uma seleção forçando uma equidade sobre os atributos que aparecem em ambos os esquemas relação.

A forma geral é:

RELAÇÃO1 |x|(atributoA, atributoB) RELAÇÃO2

Que equivale a:

σ atributoA = atributoB (RELAÇÃO1 X RELAÇÃO2)

Exemplo 1

Encontre todos os cliente que tem empréstimos e a cidade em que vivem.

π cliente-nome, cidade (EMPRESTIMO |X| CLIENTE)

Exemplo 2

Encontre todos os cliente que tem empréstimos e que moram em “Vitória”.

π cliente-nome, cidade (σ cidade=”Vitória” (EMPRESTIMO |X| CLIENTE))

Exemplo 3

Encontre os nomes de todos os clientes que têm conta nas agências situadas em “Vitória”

π cliente-nome, numer-conta (σ agencia-cidade=”Vitória” (CLIENTE |X| CONTA |X| AGENCIA))

4.9 Operação Divisão

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

Exemplo

Suponha que desejamos encontrar todos os clientes que têm uma conta em todas as agências localizadas em "Vitória". Podemos obter todas as agências de Vitória através da expressão:

r1 $\leftarrow$ π agencia-cod (σ cidade="Vitória" (AGENCIA))

Podemos encontrar todos os pares cliente-nome, agencia-cod nos quais um cliente possui uma conta em uma agência escrevendo:

r2 $\leftarrow$ π cliente-nome, agencia-cod (CONTA |X| CLIENTE)

Agora precisamos encontrar clientes que apareçam em r2 com cada nome de agência em r1.

Escrevemos esta consulta da seguinte forma:

π cliente-nome, agencia-cod (CONTA |X| CLIENTE) $\div$ π agencia-cod (σ cidade="Vitória" (AGENCIA))

4.10 Operação de inserção

Esta operação é usada para inserir dados em uma relação.

RELAÇÃO1 $\leftarrow$ RELAÇÃO1 $\cup$ RELAÇÃO2

Os valores dos atributos da RELAÇÃO1 e RELAÇÃO2 devem ter domínios equivalentes.

Exemplo

Podemos acrescentar uma nova conta em CONTA da seguinte forma:

CONTA $\leftarrow$ CONTA $\cup$ {(51, 980987, 1, 1200)}

4.11 Remoção

A remoção é expressa da seguinte forma:

RELAÇÃO1 $\leftarrow$ RELAÇÃO1 - RELAÇÃO2

Exemplo 1

Excluir todas as contas do cliente de código 01

CONTA = CONTA - (σ cliente-cod = "01" (CONTA))

Exemplo 2

Excluir todas as contas de “joão”

T ← (σ cliente-nome=”joão” (CONTA |X| CLIENTE)

CONTA = CONTA - (σ cliente-cod = “01” (T))

Exemplo 3

Suponha que a agência 051 esteja fazendo uma “promoção” e resolva abrir uma conta automaticamente para todos os clientes que possuem empréstimo, mas não possui uma conta nesta agência. O número da nova conta será igual ao número do empréstimo (ou de um dos empréstimos caso haja mais de um).

EMPRESTIMO-051 ← σ agencia-cod = 051 (EMPRESTIMO)

CONTA-051 ← σ agencia-cod = 051 (CONTA)

Os clientes que possuem um empréstimo mas não possuem uma conta na agência 051:

CLIENTE-COD ← π cliente-cod(EMPRESTIMO-051 X CLIENTE) –

π cliente-cod (CONTA-051 |X| CLIENTE)

Os novos clientes serão os que não possuem conta, com o número do empréstimo, com um saldo de 0,00 (supomos que o banco seja sério e não quer dar dinheiro a ninguém J).

CLIEENTES-NOVOS ← (π cliente-cod,agencia-cod,emprestimo-numero

(CLIENTE-COD |X| EMPRESTIMO-051)) x {(0,00)}

A resposta será então: CONTA ← CONTA $\cup$ CLIENTES-NOVOS

4.12 Atualização

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar todos os valores da tupla. Se fizermos estas mudanças usando remoção e inserção podemos não ser capazes de reter os valores que não queremos mudar. Nestes casos usamos o operador atualização, representado pela letra grega δ.

A atualização é uma operação fundamental da álgebra relacional δ atributo ← valor (RELAÇÃO) onde atributo deve ser um atributo de RELAÇÃO e valor deve ter um domínio compatível.

Exemplo 1

Suponha que o rendimento das contas tenha sido de 5% (aumente o saldo de todos em 5%).

δ saldo $\leftarrow$ saldo * 1.05 (CONTAS)

Exemplo 2

Suponhamos que contas com saldo superior a 10.000 recebam 6% de juros e as demais apenas 5%.

δ saldo $\leftarrow$ saldo * 1.06 (σ saldo > 10000 (CONTAS))

δ saldo $\leftarrow$ saldo * 1.05 (σ saldo <= 10000 (CONTAS))

5. Linguagem SQL

Quando os Bancos de Dados Relacionais estavam sendo desenvolvidos, foram criadas linguagens destinadas à sua manipulação. A versão original foi desenvolvida no Laboratório de Pesquisa da IBM. 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). Em 1986 o *American National Standard Institute* (ANSI), publicou um padrão SQL. A SQL estabeleceu-se como linguagem padrão de Banco de Dados Relacional. A IBM publicou o seu próprio SQL standard, o *Systems Aplication Arquitecture Database Interface* (SAA-SQL) em 1987.

Em 1989, tanto ANSI quanto ISO publicaram padrões substitutos (ANSI X3.135-1989 e ISO/IEC 9075:1989) que aumentaram a linguagem e acrescentaram uma capacidade opcional de integridade referencial, permitindo que projetistas de bancos de dados pudessem criar relacionamentos entre dados em diferentes partes do banco de dados. A versão em uso do padrão ANSI / ISSO SQL é o padrão SQL-92 (ANSI X3.135-1992) mas algumas aplicações atuais dão suporte apenas ao padrão SQL-89.

Desde 1993 há trabalhos sendo desenvolvidos para atualizar o padrão de modo que este atenda às características das últimas versões de bancos de dados relacionais lançadas no mercado. A principal inovação da nova versão (chamada provisoriamente de SQL3) é o suporte à orientação a objetos.

Algumas características importantes da linguagem SQL são:

- permitir trabalhar com várias tabelas;
- permitir utilizar o resultado de uma instrução SQL em outra instrução SQL (sub-queries);
- não precisar especificar o método de acesso ao dado;

- ser de fácil aprendizado;
- poder ser utilizada dentro de outras linguagens como C, COBOL, Pascal, Delphi, entre outros (SQL embutida).

SQL engloba vários tipos de comandos:

- Comandos de manipulação de dados, chamados de DML - Data Manipulation Language. São comandos que nos permitem consultar, incluir, excluir ou alterar dados de tabelas.
- Comandos de definição de dados, chamados de DDL - Data Definition Language (por exemplo: Create, Alter e Drop). São comandos que nos permitem manipular a estrutura do banco de dados, criando cada uma de suas tabelas com seus atributos, chaves, entre outros.
- Comandos de controle, chamados de DCL - Data Control Language. São comandos que nos permitem exercer controle sobre parâmetros do banco que influenciam principalmente na performance.

5.1 Estrutura Básica de Expressão SQL

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

A cláusula **select** (selecionar) corresponde à operação projeção da álgebra relacional. É usada para listar os atributos desejados no resultado de uma consulta.

A cláusula **from** (origem) corresponde à operação produto cartesiano da álgebra relacional. Ela lista as relações a serem examinadas na avaliação da expressão.

A cláusula **where** (onde) corresponde ao predicado de seleção da álgebra relacional. Consiste em um predicado envolvendo atributos de relações que aparecem na cláusula from.

Uma típica consulta SQL segue a seguinte ordem:

```
Select a1, a2, ..., an – 3ª
From T1, T2, …, Tn – 1ª
Where P – 2ª
```

Cada a_i representa um atributo e cada T_i é uma relação e P é um predicado.

Por exemplo, se desejamos selecionar em um esquema **banco** todos os clientes de Santarém, o conjunto de instruções é o seguinte:

```
Select Cliente_Nome
From Cliente
Where Cidade = 'Santarém'
```

Ou se quisermos selecionar Nome e RG dos empregados do departamento 5 da tabela de empregados, temos o seguinte conjunto de instruções:

```
Select Nome, RG
From Empregados
Where Departamento = 5
```

O resultado seria o seguinte:

Nome	RG
João	0101010101
Maria	0202020202
Marta	0303030303

Em SQL também é permitido o uso de condições múltiplas. Veja o exemplo a seguir:

```
select nome, rg, salario
from EMPREGADO
where depto = 2 AND salario > 2500.00;
```

que fornece o seguinte resultado:

Nome	RG	Salário
João	0101010101	3200,00

O operador * dentro do especificador *select* seleciona todos os atributos de uma tabela, enquanto que a exclusão do especificador *where* faz com que todas as tuplas de uma tabela sejam selecionadas. Desta forma, a expressão:

```
select *
from empregado;
```

gera o seguinte resultado:

nome	rg	cpf	depto	rg_supervisor	salario
João Luiz	10101010	11111111	1		1 3.000,00
Fernanda	20202020	22222222	1	10101010	2.500,00
Manoel	30303030	33333333	3		5.000,25
Karla	40404040	44444444	2	20202020	2.400,35

5.2 Cláusulas Distinct e All

Diferente de álgebra relacional, a operação *select* em SQL permite a geração de tuplas duplicadas como resultado de uma expressão. Para evitar isto, devemos utilizar a cláusula **distinct**. Por exemplo, sejam as seguintes consultas no esquema EMPRESA.

```
select depto
from EMPREGADO;
```

```
select distinct depto
from EMPREGADO;
```

que geram respectivamente os seguintes resultados:

Depto
1
1
2
2
3
3
4
4
5
5
5

Depto
1
2
3
4
5

A cláusula All é o default para o select ou seja: Select All indica para obter todas as tuplas. Logo, esta cláusula não precisa ser colocada (a não ser, talvez por motivos de documentação).

5.3 Predicados e ligações

A SQL não têm uma representação da operação ligação natural. No entanto, uma vez que a ligação natural é definida em termos de um produto cartesiano, uma seleção e uma projeção, é relativamente simples escrever uma expressão SQL para uma ligação natural. Por exemplo, encontre os nomes e cidades de clientes que possuam empréstimos em alguma agência.

```
Select distinct cliente_nome, cidade
From Cliente, Emprestimo
Where Cliente.cliente_cod=Emprestimo.cliente_cod
```

A SQL inclui os conectores **and**, **or** e **not**; caracteres especiais: (,), ., :, _, %<, >, <= , >= , = , <>, +, - ,* e /; e o operador para comparação: **between**, como mostra o exemplo a seguir.

Selecionar todas as contas que possuam saldo entre 10000 e 20000.

```
Select conta_numero
From CONTA
Where saldo >= 10000 and saldo <= 20000
```

que equivale à consulta:

```
Select conta_numero
From CONTA
Where saldo between 10000 and 20000
```

A SQL inclui também um operador para comparações de cadeias de caracteres, o **like**. Ele é usado em conjunto com dois caracteres especiais:

- Por cento (**%**). Substitui qualquer subcadeia de caracteres.
- Sublinhado (_). Substitui qualquer caractere.

Exemplo 1: encontre os nomes de todos os clientes cujas ruas incluem a subcadeia 'na'

```
Select distinct cliente_nome
From CLIENTE
Where rua like '%na%'
```

Exemplo 2: encontre os nomes de todos os clientes cujas ruas finalizem com a subcadeia 'na', seguido de um caractere.

```
Select distinct cliente_nome
From CLIENTE
Where rua like '%na_'
```

Para que o padrão possa incluir os caracteres especiais (isto é, % , _ , etc...), a SQL permite a especificação de um caractere de escape. O caractere de escape é usado imediatamente antes de um caractere especial para indicar que o caractere especial deverá ser tratado como um caractere normal. Definimos o caractere de escape para uma comparação **like** usando a palavra-chave **escape**. Para ilustrar, considere os padrões seguintes que utilizam uma barra invertida como caractere de escape.

- Like ' ab\%cd%' escape '\' substitui todas as cadeias começando com ' ab%cd';
- Like ' ab_cd%' escape '\' substitui todas as cadeias começando com ' ab_cd'.

A procura por não-substituições em vez de substituições dá-se através do operador **not like**.

5.4 VARIÁVEIS TUPLAS (RENOMEAÇÃO)

Seja o Exemplo: no esquema EMPRESA, selecione o número do departamento que controla projetos localizados em Rio Claro;

```
select t1.numero_depto
from departamento_projeto as t1, projeto as t2
```

```
where t1.numero_projeto = t2.numero;
```

Na expressão SQL acima, *t1* e *t2* são chamados "*alias*" (apelidos) e representam a mesma tabela a qual estão referenciando. Um "alias" é muito importante quando há redundância nos nomes das colunas de duas ou mais tabelas que estão envolvidas em uma expressão. Ao invés de utilizar o "alias", é possível utilizar o nome da tabela, mas isto pode ficar cansativo em consultas muito complexas além do que, impossibilitaria a utilização da mesma tabela mais que uma vez em uma expressão SQL. A palavra chave **as** é opcional. Por exemplo, no esquema EMPRESA, selecione o nome e o RG de todos os funcionários que são supervisores.

```
select distinct e1.nome as "Nome do Supervisor", e1.rg as
"RG do Supervisor"
from empregado e1, empregado e2
where e1.rg = e2.rg_supervisor;
```

que gera o seguinte resultado:

Nome do Supervisor	RG do Supervisor
João	0101010101
Ana	0202020202
Maria	0303030303
Manoel	0404040404

Seja o exemplo, encontre o nome e a cidade de todos os clientes com uma conta em qualquer agência.

```
Select distinct C.cliente_nome, C.cidade
from CLIENTE C, CONTA S
where C.cliente_cod = S.cliente_cod
```

5.4 OPERAÇÕES DE CONJUNTOS

A SQL inclui as operações de conjunto **union**, **intersect** e **minus** que operam em relações e correspondem às operações $\cup$, $\cap$ e **–** da álgebra relacional.

Uma vez que as relações são conjuntos, na união destas, as linhas duplicadas são eliminadas. Para que uma operação $R \cup S$, $R \cap S$ ou R - S seja válida, necessitamos que duas condições sejam cumpridas:

- As relações R e S devem ter o mesmo número de atributos;
- Os domínios do i-ésimo atributo de R e do i-ésimo atributo de S devem ser os mesmos.

Observação: Nem todos os interpretadores SQL suportam todas as operações de conjunto. Embora a operação union seja relativamente comum, são raros os que suportam intersect ou minus.

Exemplo 1:

Mostrar o nome dos clientes que possuem conta, empréstimo ou ambos na agência de código ‘ 051’:

Empréstimo na agência ‘ 051’:

```
Select distinct cliente_nome
From CLIENTE, EMPRESTIMO
Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
agencia_cod = ‘051’
```

Conta na agência ‘ 051’:

Select distinct cliente_nome
From CLIENTE, CONTA
Where CLIENTE.cliente_cod= CONTA.cliente_cod and
CONTA.agencia_cod = ‘051’

Fazendo a união dos dois:

```
(Select distinct cliente_nome
From CLIENTE, EMPRESTIMO
Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
agencia_cod = '051' )
Union
(Select distinct cliente_nome
From CLIENTE, CONTA
Where CLIENTE.cliente_cod= CONTA.cliente_cod and
CONTA.agencia_cod = '051' )
```

Exemplo 2:

Achar todos os clientes que possuam uma conta e um empréstimo na agência de código ' 051'.

```
(Select distinct cliente_nome
From CLIENTE, EMPRESTIMO
Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
agencia_cod = '051' )
intersect
(Select distinct cliente_nome
From CLIENTE, CONTA
Where CLIENTE.cliente_cod= CONTA.cliente_cod and
CONTA.agencia_cod = '051' )
```

Exemplo 3:

Achar todos os clientes que possuem uma conta mas não possuem um empréstimo na agência de código '051'.

```
(Select distinct cliente_nome
From CLIENTE, EMPRESTIMO
Where CLIENTE.cliente_cod=Emprestimos.cliente_cod and
```

```
EMPRESTIMO.agencia_cod = '051' )
minus
(Select distinct cliente_nome
From CLIENTE, CONTA
Where CLIENTE.cliente_cod= CONTA.cliente_cod and
CONTA.agencia_cod = '051' )
```

5.5 ORDENANDO A EXIBIÇÃO DE TUPLAS (ORDER BY)

A cláusula **order by** ocasiona o aparecimento de tuplas no resultado de uma consulta em uma ordem determinada. Para listar em ordem alfabética todos os clientes do banco, fazemos:

```
Select distinct cliente_nome
From CLIENTE
Order by cliente_nome
```

Como padrão, SQL lista tuplas na ordem ascendente. Para especificar a ordem de classificação, podemos especificar **asc** para ordem ascendente e **desc** para descendente. Podemos ordenar uma relação por mais de um elemento, como se segue:

```
Select *
From EMPRESTIMO
Order by quantia desc, agencia_cod asc
```

Para colocar as tuplas (linhas) em ordem é realizada uma operação de *sort*. Esta operação é relativamente custosa e portanto só deve ser usada quando realmente necessário.

5.6 MEMBROS DE CONJUNTOS

O conectivo **in** testa os membros de conjunto, onde o conjunto é uma coleção de valores produzidos por uma cláusula select. Da mesma forma, pode ser usada a expressão **not in**.

Exemplo 1:

Selecione todas as agências com código 1, 2 ou 3.

```
select *
from agencia
where agencia_cod in (1,2,3)
```

Exemplo 2:

Selecione o nome de todos os funcionários que trabalham em projetos localizados em Rio Claro.

```
select e1.nome, e1.rg, e1.depto
from empregado e1, empregado_projeto e2
where e1.rg = e2.rg_empregado and
e2.numero_projeto in
(select numero
from projeto
where localizacao = 'Rio Claro');
```

Exemplo 3:

Encontre todos os clientes que possuem uma conta e um empréstimo na agência 'Princesa Isabel'.

```
Select distinct cliente_nome
From CLIENTE
Where CLIENTE.cliente_cod in
(select cliente_cod
from CONTA, AGENCIA
```

```
where CONTA.agencia_cod = AGENCIA.agencia_cod and
agencia_nome = 'Princesa Isabel')
and CLIENTE.cliente_cod in
(select cliente_cod
from EMPRESTIMO, AGENCIA
where EMPRESTIMO.agencia_cod= AGENCIA.agencia_cod and
agencia_nome = 'Princesa Isabel')
```

Exemplo 4:

Encontre todas as agências que possuem ativos maiores que alguma agência de Vitória.

```
select distinct t.agencia_nome
from AGENCIA t, AGENCIA s
where t.ativos > s.ativos and s.cidade = 'Vitória'
```

Observação: Uma vez que isto é uma comparação "maior que", não podemos escrever a expressão usando a construção **in**.

A SQL oferece o operador **some** (equivalente ao operador **any**), usado para construir a consulta anterior. São aceitos pela linguagem: **>some, <some, >=some, <=some, =some**

```
select agencia_nome
from AGENCIA
where ativos > some
(select ativos
from AGENCIA
where agencia_cidade = 'Vitória')
```

Como o operador **some**, o operador **all** pode ser usado como: **>all, <all, >=all, <=all, =all e <>all**. A construção **> all** corresponde a frase "maior que todos".

Exemplo 5:

Encontrar as agências que possuem ativos maiores do que todas as agências de Vitória.

```
select agencia_nome
from AGENCIA
where ativos > all
(select ativos
from AGENCIA
where agencia_cidade = 'Vitória')
```

A SQL possui um recurso para testar se uma subconsulta tem alguma tupla em seus resultados. A construção **exists** retorna **true** se o argumento subconsulta está não-vazio. Podemos usar também a expressão **not exists**.

Exemplo 1:

No esquema EMPRESA, liste o nome dos gerentes que têm ao menos um dependente.

```
Select nome
from EMPREGADO
where exists
(select *
from DEPENDENTE
where DEPENDENTE.rg_responsavel = EMPREGADO.rg)
and exists
(select *
from DEPARTAMENTO
where DEPARTAMENTO.rg_gerente = EMPREGADO.rg)
```

Exemplo 2:

Usando a construção **exists**, encontre todos os clientes que possuem uma conta e um empréstimo na agência 'Princesa Isabel'.

```
Select cliente_nome
from CLIENTE
where exists
(select *
from CONTA, AGENCIA
where CONTA.cliente_cod= CLIENTE.cliente_cod and
AGENCIA.agencia_cod = CONTA.agencia_cod and
agencia_nome = 'Princesa Isabel')
and exists
(select *
from EMPRESTIMO, AGENCIA
where EMPRESTIMO.cliente_cod= CLIENTE.cliente_cod and
AGENCIA.agencia_cod = EMPRESTIMO.agencia_cod and
agencia_nome = 'Princesa Isabel')
```

5.7 FUNÇÕES AGREGADAS

A SQL oferece a habilidade para computar funções em grupos de tuplas usando a cláusula **group by**. O(s) atributo(s) 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. A SQL inclui funções para computar:

Média: **avg** Mínimo: **min** Máximo: **max** Soma: **sum** Contar: **count**

Exemplo 1:

Encontre o saldo médio de conta em cada agência.

```
Select agencia_nome, avg(saldo)
From CONTA, AGENCIA
where CONTA.agencia_cod = AGENCIA.agencia_cod
Group by agencia_nome
```

Exemplo 2:

Encontre o número de depositantes de cada agência.

```
Select agencia_nome, count(distinct cliente_nome)
From CONTA, AGENCIA
where CONTA.agencia_cod = AGENCIA.agencia_cod
Group by agencia_nome
```

Observação: Note que nesta última consulta é importante a existência da cláusula distinct, pois um cliente pode ter mais de uma conta em uma agência, e deverá ser contado uma única vez.

Exemplo 3:

Encontre o maior saldo de cada agência.

```
Select agencia_nome, max(saldo)
From CONTA, AGENCIA
Where CONTA.agencia_cod= AGENCIA.agencias_cod
Group by agencia_nome
```

Ás vezes, é útil definir uma condição que se aplique a grupos em vez de tuplas. Por exemplo, poderíamos estar interessados apenas em agências nas quais a média dos saldos é maior que 1200. Esta condição será aplicada a cada grupo e não à tuplas simples e é definida através da cláusula **having**. Expressamos esta consulta em SQL assim:

```
Select agencia_nome, avg(saldo)
From CONTA, AGENCIA
Where CONTA.agencia_cod= AGENCIA.agencias_cod
Group by agencia_nome Having avg(saldo)>1200
```

Às vezes, desejamos tratar a relação inteira como um grupo simples. Nesses casos, não usamos a cláusula **group by**, por exemplo, encontre a média de saldos de todas as contas:

```
Select avg(saldo)
From CONTA
```

5.8 MODIFICANDO O BANCO DE DADOS

4.8.1. Remoção

Podemos remover somente tuplas inteiras, não podemos remover valores apenas em atributos particulares.

Sintaxe:

```
Delete From R
Where P
```

onde R representa uma relação e P um predicado.

Note que o comando **delete** opera em apenas uma relação. O predicado da cláusula **where** pode ser tão complexo como o predicado **where** do comando **select**.

Exemplo 1:

Remover todas as tuplas de empréstimo.

```
Delete From EMPRESTIMO
```

Exemplo 2:

Remover todos os registros da conta de 'João'.

```
Delete From CONTA
where cliente_cod in
(select cliente_cod
from CLIENTE
where cliente_nome = 'João')
```

Exemplo 3:

Remover todos os empréstimos com números entre 1300 e 1500.

```
Delete From EMPRESTIMO
where emprestimo_numero between 1300 and 1500
```

Exemplo 4:

Remova todas as contas de agências localizadas em 'Vitória'.

```
Delete From CONTA
where agencia_cod in
(select agencia_cod
from AGENCIA
where agencia_cidade='Vitoria')
```

5.8.2 Inserção

Para inserir um dado em uma relação, ou especificamos uma tupla para ser inserida escrevemos uma consulta cujo resultado seja um conjunto de tuplas a serem inseridas. Os valores dos atributos para tuplas inseridas precisam necessariamente ser membros do mesmo domínio do atributo, por exemplo, inserir uma nova conta para João (código = 1), número 9000, na agência de código=2 cujo valor seja 1200.

```
insert into CONTA
values (2,9000,1,1200)
```

Na inserção acima é considerando a ordem na qual dos atributos correspondentes estão listados no esquema relação. Caso o usuário não se lembre da ordem destes atributos, pode fazer o mesmo comando da seguinte forma:

```
insert into CONTA (agencia_cod, conta_numero,
cliente_cod, saldo)
values (2,9000,1,1200)
```

Podemos querer também inserir tuplas baseadas no resultado de uma consulta. Por exemplo, inserir todos os clientes que possuam empréstimos na agência 'Princesa Isabel' na relação CONTA com um saldo de 200. O número da nova conta é o número do empréstimo * 10000.

```
insert into CONTA (agencia_cod, conta_numero,
cliente_cod, saldo)
select AGENCIA.agencia_cod, emprestimo_numero*10000,
cliente_cod, 200
from EMPRESTIMO, AGENCIA
where EMPRESTIMO.agencia_cod= AGENCIA.agencia_cod and
agencia_nome = 'Princesa Isabel'
```

5.8.3 Atualizações

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar todos os valores na tupla.

Para isso, o comando **update** pode ser usado.

Suponha que esteja sendo feito o pagamento de juros, e que em todos saldos sejam acrescentados em 5%.

Escrevemos

```
update CONTA
set saldo = saldo * 1,05
```

Suponha que todas as contas com saldo superiores a 10000 recebam aumento de 6% e as demais de 5%.

```
Update CONTA
set saldo = saldo * 1,06
where saldo >10000
Update CONTA
set saldo = saldo * 1,05
where saldo<=10000
```

A cláusula **where** pode conter uma série de comandos select aninhados. Considere, por exemplo, que todas as contas de pessoas que possuem empréstimos no banco terão acréscimo de 1%.

```
Update CONTA
set saldo = saldo * 1,01
where cliente_cod in
(select cliente_cod
from EMPRESTIMO )
```

Exemplo 1:

No esquema EMPRESA, atualize o salário de todos os empregados que trabalham no departamento 2 para R$ 3.000,00.

```
update empregado
set salario = 3000
where depto = 2;
```

Exemplo 2:

No esquema BANCO, atualize o valor dos ativos. Os ativos são os valores dos saldos das contas da agência.

```
update agencia
set ativos =
(select sum(saldo)
from conta
where conta.agencia_cod = agencia.agencia_cod)
```

5.8.4 Valores Nulos

É possível dar valores a apenas alguns atributos do esquema para tuplas inseridas em uma dada relação. Os atributos restantes são designados como nulos. Considere a requisição:

```
insert into CLIENTE (cliente_cod, cliente_nome, rua,
cidade)values (123,'Andrea',null,null)
```

A palavra chave **null** pode ser usada em um predicado para testar se um valor é nulo. Assim, para achar todos os clientes que possuem valores nulos para rua, escrevemos:

```
select distinct cliente_nome
from CLIENTE
where rua is null
```

O predicado **is not null** testa a ausência de um valor nulo.

5.9 DEFINIÇÃO DE DADOS

O conjunto de relações de um Banco de Dados precisa ser especificado ao sistema por meio de uma linguagem de definição de dados - DDL. A SQL DDL permite a especificação não apenas de um conjunto de relações, mas também de informações sobre cada relação, incluindo:

- Esquema para cada relação;
- Domínio de valores associados a cada atributo;
- Conjunto de índices a ser mantido para cada relação;
- Restrições de integridade;
- A estrutura física de armazenamento de cada relação no disco.

Uma relação SQL é definida usando o comando **create table**:

```
create table <nome_tabela> (
<nome_coluna1> <tipo_coluna1> <NOT NULL>,
<nome_coluna2> <tipo_coluna2> <NOT NULL>,
...
<nome_colunan> <tipo_colunan> <NOT NULL>);
```

A restrição **not null** indica que o atributo deve ser obrigatoriamente preenchido; se não for especificado, então o *default* é que o atributo possa assumir o valor nulo. Por exemplo, criar a tabela EMPREGADO do esquema EMPRESA, teríamos o seguinte comando:

```
create table EMPREGADO
(nome char (30) NOT NULL,
rg integer NOT NULL,
cic integer,
depto integer NOT NULL,
rg_supervisor integer,
salario, decimal (7,2)NOT NULL)
```

O comando **create table** inclui opções para especificar certas restrições de integridade, conforme veremos. A relação criada acima está inicialmente vazia. O comando insert poderá ser usado para carregar os dados para uma relação.

Para remover uma tabela de banco de dados SQL, usamos o comando **drop table**. O comando **drop table** remove todas as informações sobre a relação retirada do banco de dados:

```
drop table <nome_tabela>;
```

Por exemplo, eliminar a tabela EMPREGADO do esquema EMPRESA, teríamos o seguinte comando:

```
drop table EMPREGADOS;
```

Observe que neste caso, a chave da tabela EMPREGADO (rg) é utilizada como chave estrangeira ou como chave primária composta em diversos tabelas que devem ser devidamente corrigidas. Este processo não é assim tão simples pois, como vemos neste caso a exclusão da tabela EMPREGADO implica na alteração do projeto físico de diversas tabelas. Isto acaba implicando na construção de uma nova base de dados.

O comando **alter table** é usado para alterar a estrutura de uma relação existente. Permite que o usuário faça a inclusão de novos atributos em uma tabela. A forma geral para o comando **alter table** é a seguinte:

```
alter table <tabela> <add,drop,modify> <coluna>
<tipo_coluna>;
```

onde *add*, adiciona uma coluna; *drop*, remove uma coluna; e *modify*, modifica algo em uma tabela.

No caso do comando **alter table**, a restrição NOT NULL não é permitida pois assim que se insere um novo atributo na tabela, o valor para o mesmo em todas as tuplas da tabela receberão o valor NULL. Não é permitido eliminar algum atributo de uma relação já definida. Assim, caso você desejar eliminar uma chave primária referenciada em outra tabela como chave estrangeira, ao invés de obter a eliminação do campo, obterá apenas um erro.

5.10 VISÕES (VIEWS)

Uma view em SQL é uma tabela que é derivada de outras tabelas ou de outras views. Uma view não necessariamente existe em forma física; é considerada uma tabela virtual (em contraste com as tabelas cujas tuplas são efetivamente armazenadas no banco de dados).

Uma view é definida usando o comando **create view**. Para definir uma visão, precisamos dar a ela um nome e definir a consulta que a processa.

A forma geral é:

```
create view <nomevisao> as <expressão de consulta>
```

onde, <expressão de consulta> é qualquer consulta SQL válida. Como exemplo, considere a visão consistindo em nomes de agências e de clientes.

```
Create view TOTOS_CLIENTES as
(select agencia_nome,cliente_nome
from CLIENTE, CONTA, AGENCIA
where CLIENTE.cliente_cod = CONTA.cliente_cod and
CONTA.agencia_cod = AGENCIA.agencia_cod)
```

```
union
(select agencia_nome,cliente_nome
from CLIENTE, EMPRESTIMO, AGENCIA
where CLIENTE.cliente_cod = EMPRESTIMO.cliente_cod and
EMPRESTIMO.agencia_cod = AGENCIA.agencia_cod)
```

Nomes de visões podem aparecer em qualquer lugar onde nome de relação (tabela) possa aparecer. Usando a visão TOTOS_CLIENTES, podemos achar todos os clientes da agência 'Princesa Isabel', escrevendo:

```
select cliente_nome
from TOTOS_CLIENTES
where agencia_nome = 'Princesa Isabel'
```

Uma modificação é permitida através de uma visão apenas se a visão em questão está definida em termos de uma relação do atual banco de dados relacional. Por exemplo, selecionando tuplas de empréstimos.

```
Create view emprestimo_info as
(select agencia_cod, emprestimo_numero, cliente_cod
from EMPRESTIMO)
```

Uma vez que a SQL permite a um nome de visão aparecer em qualquer lugar em que um nome de ralação aparece, podemos escrever:

```
insert into emprestimo_info
values (1,40,7)
```

Esta inserção é representada por uma inserção na relação EMPRESTIMO, uma vez que é a relação a partir do qual a visão emprestimo_info foi construída. Devemos, entretanto, ter algum valor para *quantia*. Este valor é um valor nulo. Assim, o **insert** acima resulta na inserção da tupla: (1,40,7,null) na relação EMPRESTIMO.

Da mesma forma, poderíamos usar os comandos *update,* e *delete*.

Para apagar uma visão, usamos o comando:

```
drop view <nomevisão>
```

Por exemplo, apagar a visão emprestimo_info:

```
drop view emprestimo_info
```

5.11 RESTRIÇÕES DE INTEGRIDADE

As restrições de integridade fornecem meios para assegurar que mudanças feitas no banco de dados por usuários autorizados não resultem na inconsistência dos dados.

Há vários tipos de restrições de integridade que seriam cabíveis a bancos de dados. Entretanto as regras de integridade são limitadas às que podem ser verificadas com um mínimo de tempo de processamento.

5.11.1 Restrições de domínios

Um valor de domínio pode ser associado a qualquer atributo. Restrições de domínio são as mais elementares formas de restrições de integridade. Elas são facilmente verificadas pelo sistema sempre que um novo item de dado é incorporado ao banco de dados.

O princípio que está por traz do domínio de atributos é similar aos dos tipos em linguagem de programação.

De fato é possível criar domínios em SQL através do comando **create domain**.

A forma geral é:

```
CREATE DOMAIN nome_do_domínio tipo_original
[ [ NOT ] NULL ]
[ DEFAULT valor_default ]
[ CHECK ( condições ) ]
```

Por exemplo, cria o tipo_codigo como um número de 3 algarismos com valores permitidos de 100 a 800:

```
create domain tipo_codigo numeric(3)
not null
check ((tipo_codigo >= 100) and (tipo_codigo <= 800))
```

A cláusula **check** da SQL-92 permite modos poderosos de restrições de domínio. Ela pode ser usada também no momento de criação da tabela. Por exemplo:

```
create table dept
(deptno integer(2) not null,
dname char(12),
loc char(12),
check (dptono >= 100))
```

5.11.2 Restrições de integridade referencial

Muitas vezes, desejamos assegurar que um valor que aparece em uma relação para um dado conjunto de atributos, apareça também para um certo conjunto de atributos em outra relação. Isto é chamado de *Integridade Referencial*.

No relacionamento da Figura 4.1, considerando $\mathbf{K_1}$ como sendo *chave primária* de $\mathbf{E_1}$ e $\mathbf{K_2}$ como sendo *chave primária* de $\mathbf{E_2}$, temos em **R** tuplas que serão identificadas inclusive por $\mathbf{K_1}$ e $\mathbf{K_2}$ (na forma de chaves estrangeiras). Desta forma não poderá existir um elemento $\mathbf{K_1}$ em **R** que não faça parte de $\mathbf{E_1}$, tão pouco um $\mathbf{K_2}$ em **R** que não faça parte de $\mathbf{E_2}$.

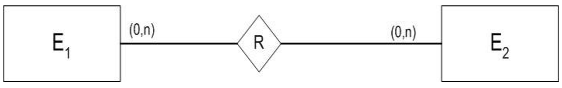

Figura 4.1: Exemplo de integridade referencial.

As modificações no banco de dados podem causar violações de integridade referencial. Considerações devem ser feitas ao inserir, remover e atualizar tuplas.

A SQL permite a especificação de chaves primárias, candidatas e estrangeiras como parte da instrução **create table**.

- A cláusula **primary key** da instrução *create table* inclui uma lista de atributos que compreende a *chave primária*;
- A cláusula **unique key** da instrução *create table* inclui uma lista de atributos que compreende a *chave candidata*;
- A cláusula **foreign key** da instrução *create table* inclui uma lista de atributos que compreende a *chave estrangeira* e o nome da relação referida pela chave estrangeira.

Seja o exemplo, criar as relações cliente, agencia e conta para o esquema do banco.

```
Create table CLIENTE
(cliente_cod integer not null,
cliente_nome char(30),
rua char(30),
cidade char(30),
primary key (cliente_cod));
Create table AGENCIA
(agencia_cod integer not null,
agencia_nome char(30),
agencia_cidade char(30),
fundos decimal (7,2),
primary key (agencia_cod),
check (fundos >= 0));
Create table CONTA
(agencia_cod int,
conta_numero char(10) not null,
cliente_cod int not null,
saldo decimal (7,2),
primary key (cliente_cod,conta_numero),
foreign key (cliente_cod) references clientes,
foreign key (agencia_cod) references agencias,
check (fundos >= 0));
```

Notem que os atributos chaves precisam ter a especificação de **not null**.

Quando uma regra de integridade referencial é violada, o procedimento normal é rejeitar a ação que ocasionou essa violação. Entretanto é possível criar ações para modificação das tabelas associadas onde houve (ou haveria) a quebra de integridade referencial.

Isto é feito através das cláusulas **on delete cascade** e **on update cascade** associadas à clausula **foreign key**. Por exemplo:

```
Create table CONTA
(...
foreign key (agencia_cod) references agencia)
on delete cascade on update cascade,
...)
```

Neste exemplo, se a remoção de uma tupla da tabela agência resultar na violação da regra de integridade, o problema é resolvido removendo as tuplas associadas com esta agência da tabela conta. De forma semelhante, se o código de uma agência for alterado, as contas desta agência serão também alteradas.

5.11.3 Asserções

Uma asserção (afirmação) é um predicado expressando uma condição que desejamos que o banco de dados sempre satisfaça. Quando uma assertiva é criada, o sistema testa sua validade. Se a afirmação é válida, então qualquer modificação posterior no banco de dados será permitida apenas quando a asserção não for violada.

Restrições de domínio e regras de integridade referencial são formas especiais de asserções.

O alto custo de testar e manter afirmações tem levado a maioria de desenvolvedores de sistemas a omitir o suporte para afirmações gerais. A proposta geral para a linguagem SQL inclui uma construção de proposta geral chamada instrução **create assertion** para a expressão de restrição de integridade. Uma afirmação pertencente a uma única relação toma a forma:

```
create assertion <nome da asserção> check <predicado>
```

Exemplo 1:

Definir uma restrição de integridade que não permita saldos negativos.

create assert saldo_restricao
check (not exists (**select** * **from** CONTA **where** saldo < 0))

Exemplo 2:

Só permitir a inserção de saldos maiores que quantias emprestadas para aquele cliente.

```
create assert saldo_restricao2
check (not exists (select * from conta
where saldo <
(select max(quantia)
from EMPRESTIMO
where EMPRESTIMO.cliente_cod = CONTA.cliente_cod))
```

6 Arquitetura de Camadas

Está na moda falar agora em "*arquitetura de camadas*" no desenvolvimento de aplicativos que acessam banco de dados. Vejamos o que isto quer dizer.

6.1 Uma camada (single-tier)

Inicialmente os aplicativos eram monolíticos: não havia uma separação clara entre a linguagem de programação e o acesso ao banco de dados. Por exemplo:

- XBase (dBase, Clipper,etc...)
- COBOL/VSAM
- Tabelas Paradox (Delphi)

Nestas situações, o código de acesso a banco de dados era em boa medida inseparável do código do resto do aplicativo. Para compartilhamento de dados em diversos computadores, um ou mais arquivos localizados no servidor são lidos e atualizados por vários aplicativos ao mesmo tempo. É claro que existe a necessidade de efetuar certo "policiamento" para evitar, por exemplo, que duas ou mais cópias do aplicativo, rodando em estações diferentes, atualizem um mesmo dado ao mesmo tempo.

Os aplicativos devem então efetuar "travamentos" ("locks") do todo ou de partes dos arquivos em momentos críticos para evitar perda de informações.

6.1.1 Problemas do Single-Tier

Os problemas com arquitetura tradicional são: integridade, segurança e às vezes, desempenho. Vejamos como um aplicativo tradicional funciona.

O servidor de arquivos tradicional funciona simplesmente como um "disco remoto", no qual as estações podem ler e escrever dados em arquivos. A informação que circula na rede é parecida com "me dê os primeiros 1000 bytes do arquivo empregados.dbf ou "escreva estes 500 bytes no arquivo emp.ntx a partir da posição 4200". Todo acesso com tabelas DBF ou Paradox deve ser feito desta forma pelos aplicativos.

Para os aplicativos poderem ler e alterar os dados é necessário dar aos usuários da rede, previlégio de escrita e leitura nos diretórios de dados. Isto quer dizer que toda a segurança tem que ser implementada pelo aplicativo, pois a rede está "fora da jogada".

Pouca coisa impede alguém de copiar os dados em disquete e levá-los embora ou de examinar campos "internos" com um aplicativo como o DBU ou Database Desktop trazido de casa em disquete.

Durante a operação do sistema, muitas coisas ruins podem ocorrer. Uma estação pode repentinamente "cair" (travou, foi desligada etc.) e deixar as tabelas em um estado instável.

Uma atualização, mesmo em um Único registro, é uma operação complexa, que depende da atualização de vários arquivos de dados e índices, caso, esta operação seja interrompida no meio, teremos problemas de integridade de dados ou índices, um problema bastante conhecido dos desenvolvedores Clipper.

Vamos supor que o aplicativo precisa "localizar o funcionário cujo código é 1000". Para fazer isto com tabelas DBF/DB, o aplicativo precisa saber onde, dentro do arquivo de dados, está o registro do funcionário 1000. Aqui entram em cena, os índices: o aplicativo efetua várias leituras no arquivo de índice (NDX/NTX/DB etc.) para saber onde está o registro procurado e só então lê os dados da tabela DBF/DB. É importante notar que neste processo foram lidos dezenas, talvez centenas de kilobytes do arquivo índice. Toda esta informação foi obrigada a circular pela rede local. Não seria melhor se pudéssemos simplesmente pedir "me dê o registro do funcionário, cujo código é 1000", deixando todas as complexas operações de busca para serem realizadas dentro do servidor? Pois é exatamente o que ocorre em um servidor SQL: enviamos o comando "select * from empregados where código = 1000" e recebemos de volta apenas os dados. Isto é muito eficiente do ponto de vista de aproveitar a capacidade da rede.

6.2 Duas Camadas (Two-tier)

Este é o Client/Server "tradicional". Existe uma divisão clara entre o aplicativo e o acesso ao bando de dados. Para resolver os problemas acima, foram criados os servidores de bancos de dados.

A idéia é criar um programa ("servidor de banco de dados") que é o *único* com permissão para acessar os arquivos de dados. Todas as consultas e atualizações devem ser

pedidas para este programa. Isto resolve diversos problemas, pois o servidor de banco de dados pode:

- Efetuar verificações adicionais de segurança (validar usuários, por exemplo)
- Garantir que as atualizações são realmente efetuadas completamente ou então descartadas. Isto é possível, porque o servidor deve possuir "no-break" e rodar apenas softwares altamente confiáveis, ao contrário das estações.

Existem diversos servidores de bancos de dados, alguns deles simples e que não usam a linguagem SQL como o ADS ou o BTrieve, este tipo de servidor pode ser uma boa solução intermediária, especialmente o ADS, que é capaz de usar tabelas DBF com grande confiabilidade.

Normalmente, no entanto, quando se fala em servidores de bancos de dados está se falando em "servidores SQL", como o Oracle, Informix, Sybase, Interbase e MS SQL Server. Todos estes produtos são bons, funcionam e fazem o que prometem.

6.3 Três camadas (three-tier) / Múltiplas camadas (n-camadas)

Com o passar do tempo, descobriu-se alguns problemas com o Client/Server "puro": a "lógica de negócios" encontra-se misturada com o código da "interface com o usuário". Foi idealizada devido à ascensão da internet.

Vejamos um exemplo, tomemos um sistema de controle de materiais/estoque. Cada vez que for feito um pedido de material, várias operações devem ser feitas:

- Verificação de disponibilidade de estoque;
- Verificação, se, aquele usuário pode pedir determinado material e naquela quantidade;
- Baixa do estoque;
- Verificação de estoque mínimo e eventual criação de um pedido de compra;
- Lançamento do custo em conta adequada ("centro de custo").

As operações acima podem ser chamadas de uma "transação", que implementa parte da "lógica de negócios" do aplicativo. Note que ela é independente tanto do banco de dados como da interface com o usuário.

Se a transação for executada de apenas um *único* programa, não existe nenhum problema. O código que implementa a transação estará dentro do executável do aplicativo.

O problema é que a transação acima pode ser invocada de uma série de aplicativos, por exemplo:

- Pedido interno de material de escritório;
- Pedido de material para produção;
- Venda pela equipe interna;
- Venda através da WEB.

É claro que não desejamos escrever a transação mais de uma vez. Se todos os programas acima forem escritos em uma linguagem como C, Pascal ou Basic, poderíamos colocar a transação em um arquivo-fonte, chamá-la de vários lugares. Infelizmente, alguns problemas ainda restam:

- Caso a transação mude, devemos recopilar e redistribuir todos os aplicativos que usam aquela transação. Manter a consistência em uma grande organização, pode ao ser fácil;
- Alguns aplicativos podem ser escritos em várias linguagens e ferramentas, como Delphi, Visual Basic, C++ etc. Isto é especialmente provável no caso de aplicativos WEB;
- A transação em si pode ser bastante pesada. Roda-la em um micro relativamente lento, no "cliente", com acesso a uma linha de comunicação lenta, pode não ser o ideal.

6.4 Solução em múltiplas camadas

A solução para os problemas acima é dividir o aplicativo em três camadas (Figura 5.1):

- Interface com o usuário.
- Lógica de negócios (transações).
- Acesso a banco de dados (servidor de banco de dados).

Este esquema é uma evolução do Client/Server tradicional. Nele, separamos as transações da interface com o usuário. Estas transações são colocadas fisicamente em outro executável (normalmente uma DLL especial com extensão OCX). Este executável pode ou não ser rodado na mesma máquina que o aplicativo cliente.

Nos modelos mais usualmente propostos, estes executáveis contendo as transações são escritos segundo o padrão de objetos da Microsoft, o COM - Component Object Model. Utilizando o padrão DCOM, eles podem ser executados em qualquer máquina da rede.

Este esquema é bastante novo e as ferramentas necessárias à sua implantação estão em sua infância. Para desenvolver um aplicativo verdadeiramente "3-tier" hoje em dia, o trabalho adicional ainda será muito grande. Na maioria dos casos, vale mais a pena esperar um pouco, até que as ferramentas melhorem e aí alterar os aplicativos. Além disto, alguns aplicativos simplesmente não necessitam de toda esta complexidade adicional.

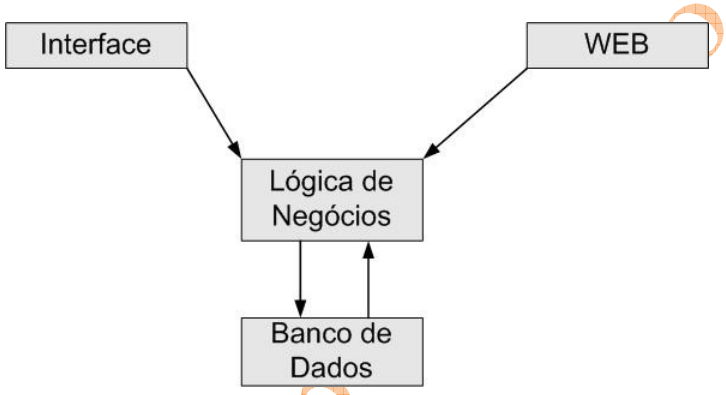

Figura 5.1: Representação da arquitetura de 3 camadas.

7. Servidores SQL

Os servidores SQL estão no mercado há uns 20 anos e neste tempo eles foram adicionando diversos recursos que vão muito além de simplesmente acessar as tabelas de dados. Um programador Clipper pode achar esta afirmação estranha, pois "o que pode haver de tão importante além de acessar os dados?" A resposta é: muita coisa, se, você realmente utilizar todos os recursos do servidor SQL o seu aplicativo será mais poderoso, confiável e bem mais fácil de desenvolver.

Bibliografia

BATISTI, Júlio. **SQL server 2000 Administração e Desenvolvimento**: Curso Completo. Rio de Janeiro, 2001.

DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. 7. ed. Rio de Janeiro. Ed. Campus, 2000.

GARCIA – MOLINA, Hector. **Implementação de sistemas de banco de dados**. Rio de janeiro. Campus, 2001.

HEUSER, Carlos Alberto. **Projeto de Banco de Dados.** 2. ed. Porto Alegre. Sagra-Luzzatto, 1999.

KORTH, Henry F. ε SILBERSCHATZ, Abraham, **Sistema de Bancos de Dados**. 3. Ed. São Paulo. Makron Books, 1999.

MACHADO, Felipe N. R. **Banco de Dados**: Projeto e Implementação. São Paulo: Érica, 2004.

O' NEIL, Patrick. **Database - - principles, programming e performance.** 2.ed. USA. Morgan Kaufmann Publishers, 2001.

RAMALHO, José Antônio. **Sql Server 7 – Iniciação e Referência.** São Paulo. Makron Books, 1999.

RAPCHAN, Francisco. **Banco de Dados**. Documento on-line. Disponível na WWW <www.geocities.com/chicorapchan > Acesso em: Agosto/2004.