Apostila de Banco de Dados I

V1.0 2013A

Sumário

1. Introdução

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 os mesmos podem ser aplicados.

Um “banco de dados” pode ser definido como um conjunto de “dados” devidamente relacionados. Por “dados” podemos compreender como “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 populado 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 do mundo real, o qual é chamado de “mini-mundo” ; qualquer alteração efetuada no mini-mundo é automaticamente refletida no banco de dados.

Dado é qualquer elemento identificado em sua forma bruta que, por si só, não conduz a uma compressão de determinado fato ou situação.

Informações são dados tratados. Representação estruturada e formatada do dado. O resultado do processamento de dados são as informações. As informações tem significado, podem ser tomadas decisões ou fazer afirmações considerando as informações.

O **conhecimento** vai além de informações, pois ele além de ter um significado tem uma aplicação. Segundo a Wikipedia, conhecimento é: Conhecimento é o ato ou efeito de abstrair idéia ou noção de alguma coisa, como por exemplo: conhecimento das leis; conhecimento de um fato (obter informação); conhecimento de um documento; termo de recibo ou nota em que se declara o aceite de um produto ou serviço; saber, instrução ou cabedal científico (homem com grande conhecimento).

Chiavenato (2004) faz uma comparação:

Dados	**Informação**	**Conhecimento**
• Um conjunto de fatos a respeito do mundo; • São geralmente quantificados; • São facilmente capturados e arquivados em computadores; • Não permitem julgamentos ou significados; • Não constituem base para a ação	• Um conjunto de dados conjugados que possuem relevância e propósito; • Pode ser transformada pela análise humana e julgamento; • Pode ser arquivada em documento ou em arquivos virtuais; • Constitui base para a ação.	• Um conjunto organizado e estruturado de informações a respeito do mundo; • Requer intervenção humana e inteligente; • Proporciona comparações; • Permite deduções; • Proporciona implicações; • Está relacionado com o que as outras pessoas pensam.

Os fatos conhecidos são dados que podem ser armazenados. Dados armazenados com estrutura definem uma informação. A informação que possui um significado e aplicação dependendo do contexto se transforma em conhecimento.

Um banco de dados pode ser criado e mantido por um conjunto de aplicações desenvolvidas especialmente para esta tarefa ou por um “Sistema Gerenciador de Banco de Dados” (SGBD). Um SGBD permite aos usuários criarem e manipularem bancos de dados de propósito geral. O objetivo principal do

SGBD é proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação e armazenamento das informações no banco de dados.

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

Os banco de dados são amplamente usados:

- Banco: para informações de clientes, contas, empréstimos e todas as transações bancárias.
- Linhas aéreas: para reservas, e informações de horários. As linhas aéreas foram umas das primeiras a usar bancos de dados de maneira geograficamente distribuída.
- Universidades: para informações de alunos, registros de cursos e notas.
- Transações de cartão de crédito: para compras com cartões de crédito e geração de faturas mensais.
- Telecomunicação: para manter registros de chamadas realizadas, gerar cobranças mensais, manter saldos de cartões de chamada pré-pagos e armazenar informações sobre as redes de comunicações.
- Finanças: para armazenar informações sobre valores mobiliários, vendas e compras de instrumentos financeiros como ações e títulos; também para armazenar dados de mercado em tempo real a fim de permitir negócios on-line por clientes e transações automatizadas pela empresa.
- Vendas: para informações de clientes, produtos, compra.
- Revendedores on-line: para dados de vendas descritos aqui, além de acompanhamento de pedidos, geração de lista de recomendações personalizadas e manutenção de avaliações de produto-online.
- Indústria: para gerenciamento da cadeia de suprimento e para controlar a produção de itens nas fábricas, estoques de itens em armazéns e lojas, além de itens.
- Recursos humanos: para informações sobre funcionários, salários, descontos em folha de pagamento, benefícios e para geração de contra-cheques.

1.1. Abordagem Banco de Dados X Abordagem Processamento Tradicional de Arquivos

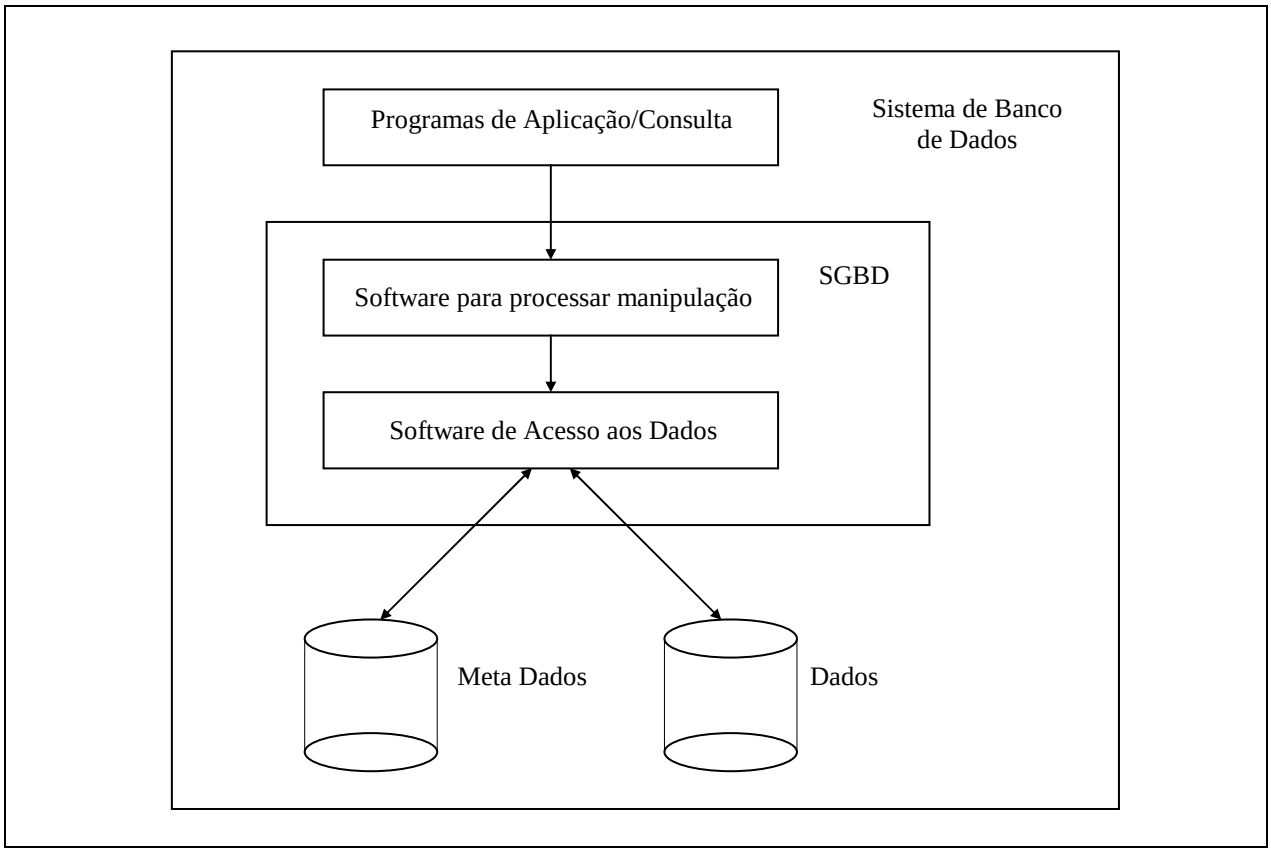
1.1.1. Auto Informação

Uma característica importante da abordagem Banco de Dados é que o SGBD mantém não somente os dados mas também a forma como os mesmos 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, etc. A informação armazenada no catálogo é chamada de “Meta Dados”. No processamento tradicional de arquivos, o programa que irá manipular os dados deve conter este tipo de informação, ficando limitado a manipular as informações que o mesmo conhece. Utilizando a abordagem banco de dados, a aplicação pode manipular diversas bases de dados diferentes.

1.1.2. 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 banco de dados, a estrutura é alterada apenas no catálogo, não alterando os programas.

Figura 1 - Um ambiente de Sistema de Banco de Dados



1.1.3. 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.

1.1.4. Múltiplas Visões de Dados

Como um conjunto de informações pode ser utilizado por um conjunto diferenciado de usuários, é importante que estes usuários possam ter “visões” diferentes da base de dados. Uma “visão” é definida como um subconjunto de uma base de dados, formando deste modo, um conjunto “virtual” de informações.

1.2. Usuários

Para um grande banco de dados, existe um grande número de pessoas envolvidas, desde o projeto, uso até manutenção.

1.2.1. Administrador de Banco de Dados (DBA)

Em um ambiente de banco de dados, o recurso primário é o banco de dados por si só e o recurso secundário o SGBD e os softwares relacionados. A administração destes recursos cabe ao Administrador de Banco de Dados, dentre as funções de um DBA[1] destacamos:

- **Definição do esquema**: O DBA cria o esquema de banco de dados original escrevendo um conjunto de definições que são transformadas pelo compilador DDL em um conjunto de tabelas armazenadas de modo permanente no dicionário de dados.
- **Definição da estrutura de dados e métodos de acesso**. O DBA cria estrutura de dados e métodos de acesso apropriados escrevendo um conjunto de definições, as quais são traduzidas pelo compilador de armazenamento de dados e pelo compilador de linguagem de definição de dados.
- **Esquema e modificações na organização física**. Os programadores realizam relativamente poucas alterações no esquema do banco de dados ou na descrição da organização física de armazenamento por meio de um conjunto de definições que serão usadas ou pelo compilador DDL ou pelo compilador de armazenamento de dados e definição de dados, gerando modificações na tabela apropriada, interna ao sistema.
- **Concessão de autorização de acesso ao sistema**. O fornecimento de diferentes tipos de autorização no acesso aos dados permite que o administrador de dados regule o acesso dos diversos usuários às diferentes partes do sistema.
- **Manutenção de Rotina**. Exemplos da manutenção de rotina são:
    - Realizar backups periódicos do banco de dados, sejam em fitas ou em servidores remotos, para previnir perda de dados no caso de acidentes, como incêndio, inundação, etc.
    - Garantir que haja suficiente espaço livre em disco para operações normais e aumentar o espaço em disco conforme o necessário.
    - Monitorar tarefas sendo executas no banco de dados e assegurar que o desempenho não seja comprometido por tarefas muito onerosas submetidas por alguns usuários.

O Projetista de Banco de Dados é responsável pela identificação dos dados que devem ser armazenados no banco de dados, escolhendo a estrutura correta para representar e armazenar dados. Muitas vezes, os projetistas de banco de dados atuam como "staff" do DBA, assumindo outras responsabilidades após a construção do banco de dados. É função do projetista também avaliar as necessidades de cada grupo de usuários para definir as visões que serão necessárias, integrando-as, fazendo com que o banco de dados seja capaz de atender a todas as necessidades dos usuários.

1.2.2. Usuários Finais

A meta básica de um sistema de banco de dados é proporcionar um ambiente para recuperação de informações e para o armazenamento de novas informações no banco de dados. Há quatro tipos de usuários de sistemas de banco de dados, diferenciados por suas expectativas de interação com o sistema.

- **Usuários Sofisticados**: são usuários que estão familiarizados com o SGBD e realizam consultas complexas, interagem com o sistema sem escrever programas.
- **Usuários Especialistas**: são usuários sofisticados que escrevem aplicações especializadas de bancos de dados que não podem ser classificadas como aplicações tradicionais em processamento de dados. Ex. sistemas especialistas, sistemas de base de conhecimento, etc..

---

[1] Do inglês **D**ata**B**ase **A**dministrator.

- **Usuários Navegantes**: são usuários que interagem com o sistema chamando um dos programas aplicativos já escritos. Acessam o banco de dados casualmente.
- **Analistas de Sistemas e Programadores de Aplicações**: analistas determinam os requisitos dos usuários finais e desenvolvem especificações para transações que atendam estes requisitos, e os programadores implementam estas especificações como programas, testando, depurando, documentando e dando manutenção no mesmo. É importante que, tanto analistas quanto programadores, estejam a par dos recursos oferecidos pelo SGBD.

1.3. Vantagens e desvantagens do uso de um SGBD

1.3.1. Controle de Redundância

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.

1.3.2. Compartilhamento de Dados

Um SGBD multiusuário 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 multiusuário deve manter o controle de concorrência para assegurar que o resultado de atualizações sejam corretos. Um banco de dados multiusuários deve fornecer recursos para a construção de múltiplas visões.

1.3.3. Restrição a Acesso não Autorizado

Um SGBD deve fornece 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.

1.3.4. Representação de Relacionamentos Complexos entre Dados

Um banco de dados pode incluir uma variedade de dados que estão inter-relacionados de várias formas. 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.

1.3.5. Tolerância a Falhas

Um SGBD deve fornecer recursos para recuperação de falhas tanto de software quanto de hardware.

1.4. 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 tradicional 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 ser comprometidas. A sobrecarga causada pelo uso de um SGBD e a má administração justificam a utilização da abordagem 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.

1.5. História dos Sistemas de Banco de Dados

O processamento de dados tem impulsionado o crescimento dos computadores desde os primeiros dias dos computadores comerciais. Na verdade, a automação das tarefas de processamento de dados já existia antes mesmo dos computadores. Cartões perfurados, inventados por Herman Hollerith, foram usados no início do século XX para registrar dados do censo dos Estados Unidos, e sistemas mecânicos foram usados para processar os cartões perfurado e tabular os resultados. Mais tarde, os cartões perfurados passaram a ser amplamente usados como um meio de inserir dados em computadores.

As técnicas de armazenamento e processamento de dados evoluíram ao longo dos anos.

- **Década de 1950 e início da década de 1960:**
  - Processamento de dados usando fitas magnéticas para armazenamento
    - Fitas fornecem apenas acesso seqüencial
  - Cartões perfurados para entrada
- **Final da década de 1960 e década de 1970:**
  - Discos rígidos permitem acesso direto aos dados
  - Modelos de dados de rede e hierárquico em largo uso
  - Ted Codd define o modelo de dados relacional
    - Ganharia o ACM Turing Award por este trabalho
    - IBM Research inicia o protótipo do System
    - UC Berkeley inicia o protótipo do Ingres
  - Processamento de transação de alto desempenho (para a época)

- **Década de 1980:**
  - Protótipos relacionais de pesquisa evoluem para sistemas comerciais
    - SQL se torna o padrão do setor
  - Sistemas de banco de dados paralelos e distribuídos

- Sistemas de banco de dados orientados a objeto

- **Década de 1990:**
  - Grandes aplicações de suporte a decisão e exploração de dados
  - Grandes data warehouses de vários terabytes
  - Surgimento do comércio Web
- **Década de 2000:**
  - Padrões XML e XQuery
  - Administração de banco de dados automatizada

2. Conceitos de SGBD

2.1. Visão de Dados

Um SGBD é uma coleção de arquivos e programas inter-relacionados que permitem ao usuário o acesso para consultas e alterações desses dados. O maior benefício de um banco de dados é proporcionar ao usuário uma visão abstrata dos dados. Isto é, o sistema acaba por ocultar determinados detalhes sobre a forma de armazenamento e manutenção de dados.

2.2. Abstração de Dados

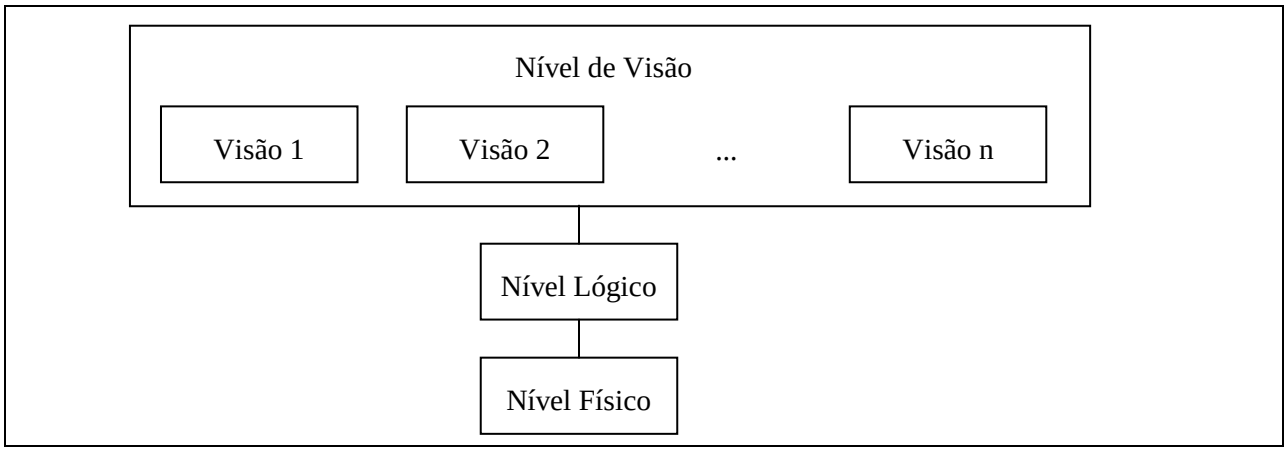
Para que se possa usar um sistema, ele precisa ser eficiente na recuperação das informações. Esta eficiência está relacionada à forma pela qual foram projetadas as complexas estruturas de representação desses dados no banco de dados. Já que muitos dos usuários dos sistemas de banco de dados não são treinados em computação, os técnicos em desenvolvimento de sistemas omitem essa complexidade desses usuários por meio dos níveis de abstração, de modo a facilitar a interação dos usuários com o sistema. A divisão a seguir é proposta pelo Study Group on Data Base Management Systems (chamada arquitetura ANSI/SPARC).

Nível Físico ou interno, é o mais baixo nível de abstração que descreve como esses dados estão de fato armazenados. No nível físico, estruturas de dados complexas de nível baixo são descritas em detalhes.

Nível Lógico ou conceitual, este nível médio de abstração descreve quais dados estão armazenados no banco de dados e quais o inter-relacionamentos entre eles. Assim, o banco de dados como um todo é descrito em termos de um número relativamente pequeno estruturas simples. Embora a implementação dessas estruturas simples no nível lógico possa envolver estruturas complexas no nível físico, o usuário do nível lógico não necessariamente precisa estar familiarizado com essa complexidade. O nível lógico de abstração é utilizado pelos administradores de banco de dados que precisam decidir quais informações devem pertencer ao banco de dados. Geralmente usuários individuais são levados em consideração neste nível.

Nível de Visão ou externo, o mais alto nível de abstração descreve apenas parte do banco de dados. A despeito das estruturas simples do nível lógico, alguma complexidade permanece devido ao tamanho dos bancos de dados. Muitos dos usuários de banco de dados não precisam conhecer todas as suas informações. Pelo contrário, os usuários normalmente utilizam apenas parte do banco de dados. Assim, para que estas interações sejam simplificadas, um nível visão é definido. Grupos de usuários são levados em consideração para construir as visões. O sistema pode proporcionar diversas visões do mesmo banco de dados.

Figura 2 - Os três níveis de abstração de dados



2.3. Instâncias e Esquema

Em qualquer modelo de dados utilizado, é importante distinguir a “descrição” do banco de dados do “banco de dados” por si próprio. A descrição de um banco de dados é chamada de “esquema de um banco de dados” e é especificada durante o projeto do banco de dados. Geralmente, poucas mudanças ocorrem no esquema do banco de dados.

Os dados armazenados em um banco de dados em um determinado instante do tempo formam um conjunto chamado de “instância do banco de dados”. A instância altera toda vez que uma alteração no banco de dados é feita.

O SGBD é responsável por garantir que toda instância do banco de dados satisfaça ao esquema do banco de dados, respeitando sua estrutura e suas restrições. Analogias com conceitos de linguagem de programação, como tipos de dados, variáveis e valores são úteis aqui. Por exemplo, vamos definir um registro de cliente, note que na declaração do seu tipo, não definimos qualquer variável.

```
Tipo
      Cliente = Registro
            nome_cliente : String;
            seguro_social : String;
            rua_cliente : String;
            cidade_cliente : String;
      Fim_registro;
```

Para declarar `cliente1` corresponde agora a uma área de memória que contém um registro do tipo cliente.

```
Variáveis
      cliente1 = Cliente;
```

Um esquema de banco de dados corresponde à definição do tipo em uma linguagem de programação. Uma variável de um dado tipo tem um valor em particular em dado instante. Assim, esse valor corresponde a uma instância do esquema do banco de dados.

Os sistemas de banco de dados apresentam diversos esquemas. No nível mais baixo há o esquema físico; no nível intermediário, o esquema lógico; e no nível mais alto, os subesquemas. Em geral, os sistemas de banco de dados dão suporte a um esquema físico, um esquema lógico e vários subesquemas.

2.4. Independência de Dados

A “independência de dados” pode ser definida como a capacidade de se alterar um esquema em um nível em um banco de dados sem ter que alterar um nível superior (Figura 2). Existem dois tipos de independência de dados:

- **Independência de dados física**: é a capacidade de alterar o esquema físico sem que com isso, qualquer programa de aplicação precise ser reescrito. Modificações no nível físico são necessárias, ocasionalmente, para aprimorar o desempenho.
- **Independência de dados lógica:** é a capacidade de alterar o esquema lógico sem que com isso, qualquer programa de aplicação precise ser reescrito. Modificações no nível lógico são necessárias sempre que uma estrutura lógica do banco de dados é alterada.

A independência de dados lógica é mais difícil de ser alcançada que a independência física, uma vez que os programas de aplicação são mais fortemente dependentes da estrutura lógica do que de seu acesso.

O Conceito de independência de dados é de várias formas similar ao conceito de tipo abstrato de dados empregado nas linguagens modernas de programação. Ambos os conceitos omitem os detalhes de implementação do usuário, permitindo que ele se concentre em sua estrutura geral em vez de se concentrar nos detalhes tratados no nível mais baixo.

2.5. Modelo de Dados

Sob a estrutura do banco de dados está o modelo de dados; um conjunto de ferramentas conceituais usadas para a descrição de dados, relacionamentos entre dados, semântica de dados e regras de consistência. Os mais comuns são modelos lógicos com base em objetos e modelos lógicos com base em registros.

2.5.1. Modelo Lógico com Base em Objetos

Os modelos lógicos com base em objetos são usados na descrição de dados no nível lógico e de visões. São caracterizados por dispor de recursos de estruturação bem mais flexíveis e por viabilizar a especificação explícita das restrições dos dados. Existem vários modelos nessa categoria, e outros ainda estão por surgir. Alguns amplamente conhecidos, como:

- Modelo Entidade-Relacionamento
- Modelo Orientado a Objeto

2.5.1.1. Modelo Entidade-Relacionamento

O modelo de dados entidade-relacionamento(ER) tem por base a percepção do mundo real como um conjunto de objetos básicos, chamados entidades, e dos relacionamento entres eles. Uma entidade é uma “coisa” ou um “objeto” do mundo real que pode ser identificado por outros objetos.

2.5.1.2. Modelo Orientado a Objeto

Como o modelo ER, o modelo orientado a objetos tem por base um conjunto de objetos. Um objeto contém valores armazenados em variáveis instâncias dentro do objeto. Um objeto também contém conjuntos de códigos que operam o objeto. Esses conjuntos de códigos são chamados operações.

2.5.2. Modelo Lógicos com Base em Registros

Modelos lógicos com base em registros são usados para descrever os dados no nível lógico e de visão. Em contraste com os modelos com base em objetos, este tipo de modelo é usado tanto para especificar a estrutura lógica do banco de dados quanto para implementar uma descrição de alto nível.

Os modelos com base em registros são assim chamados porque o banco de dados é estruturado por meio de registros de formato fixo de todos os tipos. Cada registro define um número fixo de campos ou atributos, e cada campo possui normalmente tamanho fixo.

Os três modelos de dados com base em registro mais comumente utilizados são o relacional, o de rede e o hierárquico.

2.5.2.1. Modelo Relacional

O modelo relacional usa um conjunto de tabelas para representar tanto os dados como a relação entre eles. Cada tabela possui múltiplas colunas e cada uma possui um nome único. As tabelas abaixo apresentam um exemplo de banco de dados relacional em duas tabelas: uma mostrando os clientes do banco e outra suas contas.

Figura 3 - Um exemplo de Banco de Dados Relacional

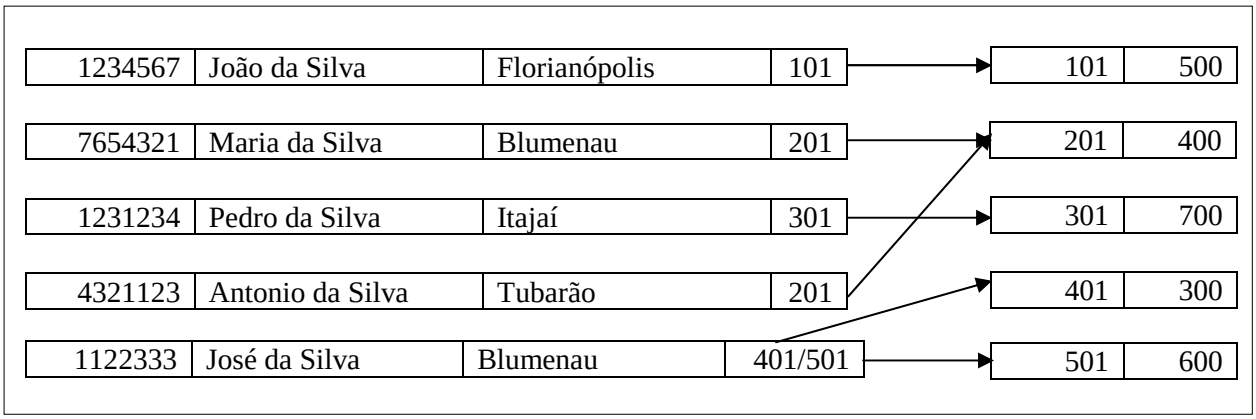
RG_Cliente	Nome_Cliente	Cidade_Cliente	Número_Conta
1234567	João da Silva	Florianópolis	101
7654321	Maria da Silva	Blumenau	201
1231234	Pedro da Silva	Itajaí	301
4321123	Antônio da Silva	Tubarão	201
1122333	José da Silva	Blumenau	401
1122333	José da Silva	Blumenau	501

Número_Conta	Saldo
101	500
201	400
301	700
401	300
501	600

2.5.2.2. Modelo de Rede

Os dados no modelo de rede são representados por um conjunto de registros e as relações entre estes registros são representadas por links(ligações), as quais podem ser vistas por ponteiros. Os registros são organizados no banco de dados por um conjunto arbitrário de gráficos.

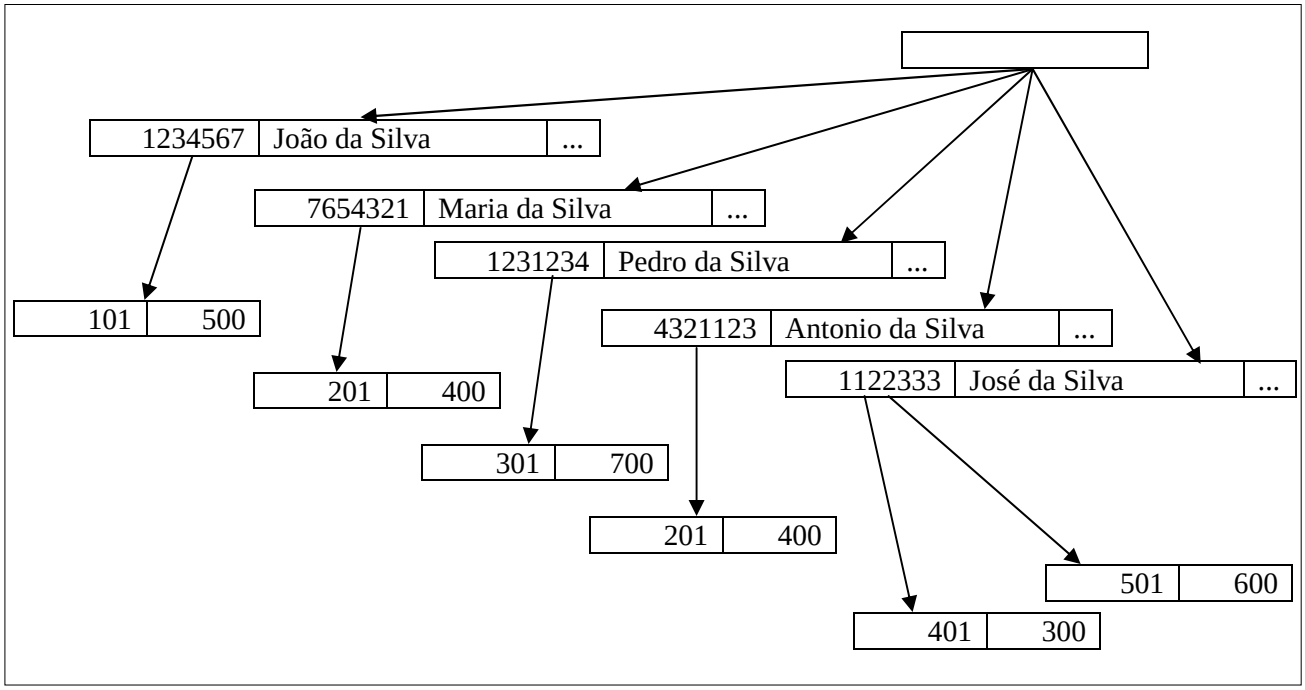
Figura 4 - Um exemplo de Banco de em Rede



2.5.2.3. Modelo Hierárquico

O modelo hierárquico é similar ao modelo em rede, pois os dados e suas relações são representados, respectivamente, por registros e links. A diferença é que no modelo hierárquico os registros estão organizados em árvores em vez de em gráficos arbitrários.

Figura 5 - Um exemplo de Banco de Dados Hierárquico



2.6. Linguagem de Banco de Dados

Um sistema de banco de dados proporciona dois tipos de linguagens: uma específica para os esquemas do banco de dados e outra para expressar consultas e atualizações.

2.6.1. Linguagem de Definição de Dados

Para a definição dos esquemas lógico ou físico pode-se utilizar uma linguagem chamada **DDL** (Data Definition Language - Linguagem de Definição de Dados). O **SGBD** possui um compilador **DDL** que permite a execução das declarações para identificar as descrições dos esquemas e para armazená-las em tabelas que constituem um arquivo especial chamado dicionário de dados ou diretório de dados.

Um dicionário de dados é um arquivo de **metadados** – isto é, dados a respeito de dados. Em sistema de banco de dados, esse arquivo ou diretório é consultado antes que o dados real seja modificado.

Em um **SGBD** em que a separação entre os níveis lógico e físico são bem claras, é utilizado uma outra linguagem, a **SDL** (Storage Definition Language - Linguagem de Definição de Armazenamento) para a especificação do nível físico. A especificação do esquema conceitual fica por conta da **DDL**.

Em um **SGBD** que utiliza a arquitetura três esquemas, é necessária a utilização de mais uma linguagem para a definição de visões, a **VDL** (Vision Definition Language - Linguagem de Definição de Visões).

2.6.2. Linguagem de Manipulação de Dados

Uma vez que o esquema esteja compilado e o banco de dados esteja populado, usa-se uma linguagem para fazer a manipulação dos dados, a **DML** (Data Manipulation Language - Linguagem de Manipulação de Dados).

Por manipulação entendemos:

- A recuperação das informações armazenadas no banco de dados;
- Inserção de novas informações no banco de dados;
- A remoção das informações no banco de dados;
- A modificação das informações no banco de dados;

No nível físico, precisamos definir algoritmos que permitam o acesso eficiente aos dados. Nos níveis mais altos de abstração, enfatizamos a facilidade de uso. O Objeto é proporcionar uma interação eficiente entre homens e sistema.

2.7. Os Módulos Componentes de um SGBD

Um sistema de banco de dados é particionado em módulos que lidam com cada uma das responsabilidades do sistema geral. Os componentes funcionais de um sistema de banco de dados podem ser divididos, de modo amplo, nos componentes do gerenciador de armazenamento e processador de consultas.

O gerenciador de armazenamento é importante porque os bancos de dados normalmente exigem uma grande quantidade de espaço de armazenamento. Os bancos de dados corporativos variam de centenas de gigabytes a – para os maiores banco de dados – terabytes de dados. Um gigabyte possui 1000 megabytes (ou 1 bilhão de bytes) e um terabyte possui 1 milhão de megabytes (ou 1 trilhão de bytes). Como a memória principal dos computadores não pode armazenar tanta quantidade de dados, as

informações são armazenadas em discos. Os dados são movidos entre o armazenamento em disco e a memória principal conforme o necessário. Uma vez que o movimento de dados para o disco é lento, em comparação à velocidade da CPU, é fundamental que o sistema de banco de dados estruture os dados de modo a minimizar a necessidade de mover dados entre disco e memória principal.

O processador de consulta é importante porque ajuda o sistema de banco de dados a simplificar o acesso aos dados. As visões de alto nível ajudam a alcançar esses objetivos; com ela, os usuários do sistema não são desnecessariamente afligidos com os detalhes físicos da implementação do sistema. Entretanto, o rápido processamento das atualizações e consultas é importante. É função do sistema de banco de dados é traduzir atualizações e consultas escritas em uma linguagem não procedural, no nível lógico, em uma seqüência eficiente de operações no nível físico.

2.7.1. Gerenciador de Armazenamento

Um gerenciador de Armazenamento é um módulo de programa que fornece a interface entre os dados de baixo nível armazenados no banco de dados e os programas de aplicação e consultas submetidos ao sistema. O gerenciador de armazenamento é responsável pela interação com o gerenciador de arquivos. Os dados brutos são armazenados no disco usando o sistema de arquivos, que normalmente é fornecido por um sistema operacional convencional. O gerenciador de armazenamento traduz as várias instruções DML em comandos de sistema de arquivo de baixo nível. Portanto, o gerenciador de armazenamento é responsável por armazenar, recuperar e atualizar dados no banco de dados.

Os componentes do gerenciador de armazenamento incluem:

- **Gerenciador de autorização e integridade**, que testa a satisfação das restrições de integridade e verifica a autoridade dos usuários para acessar dados.
- **Gerenciador de transações**, que garante que o banco de dados permaneça em estado consistente (correto) a despeito de falhas no sistema e que transações concorrentes serão executadas sem conflitos em seus procedimentos.
- **Gerenciador de arquivos**, que controla a alocação de espaço no armazenamento de disco e as estruturas de dados usadas para representar informações armazenadas no disco.
- **Gerenciador de buffer**, responsável por buscar dados do armazenamento de disco para a memória principal e decidir que dados colocar em **cachê** na memória principal. O gerenciador de buffer é uma parte crítica do sistema de banco de dados, já que permite que o banco de dados manipule tamanhos de dados que sejam muito maiores do que o tamanho da memória.

O gerenciador de armazenamento implementa estruturas de dados como parte da implementação do sistema físico:

- **Arquivo de Dados**, que armazena o próprio banco de dados.
- **Dicionário de Dados**, que armazena os **metadados** relativos à estrutura do banco de dados, em especial o esquema de banco de dados.
- **Índices**, que proporcionam acesso rápido aos itens de dados que são associados a valores determinados.
- **Estatísticas de dados** armazenam informações estatísticas relativas aos dados contidos no banco de dados.

2.7.1.1. Gerenciamento de Transação

Muitas vezes, várias operações no banco de dados forma uma única unidade lógica de trabalho. Um exemplo é uma transferência de fundos, em que uma conta (digamos A) é debitada e outra conta (digamos B) é creditada. Obviamente, é fundamental que tanto o débito quanto o crédito ocorram, ou

então que nenhuma ocorra. Ou seja, a transferência de fundos precisa acontecer em sua totalidade ou não acontecer. Essa propriedade "tudo ou nada" é chamada **atomicidade**. Além disso, é essencial que a execução da transferência de fundos preserve a consistência do banco de dados. Isto é, o valor da soma de A+B precisa ser preservado. Essa propriedade de exatidão é chamada **consistência**. Finalmente, após a execução bem-sucedida de uma transferência de fundos, os novos valores das contas A e B precisa persistir, apesar da possibilidade de falha do sistema. Essa propriedade de persistência é chamada de **durabilidade**.

Como temos acesso de diversos usuários ao banco de dados, diversas operações de transferência de fundos podem estar acontecendo concorrentemente. O resultado final das operações é igual ao obtido com a execução isolada de cada uma delas. A observância da propriedade de **isolamento** das transações pelos SGBDs impede a ocorrência dos problemas de acesso a dados.

Uma transação é um conjunto de operações que realiza uma única função lógica em uma aplicação de banco de dados. Cada transação é uma unidade da atomicidade, consistente, isolada e durável.

Garantir as propriedades da atomicidade, consistência, isolamento e durabilidade[2] é função do próprio sistema de banco de dados – especificamente, do componente de gerenciamento de transação.

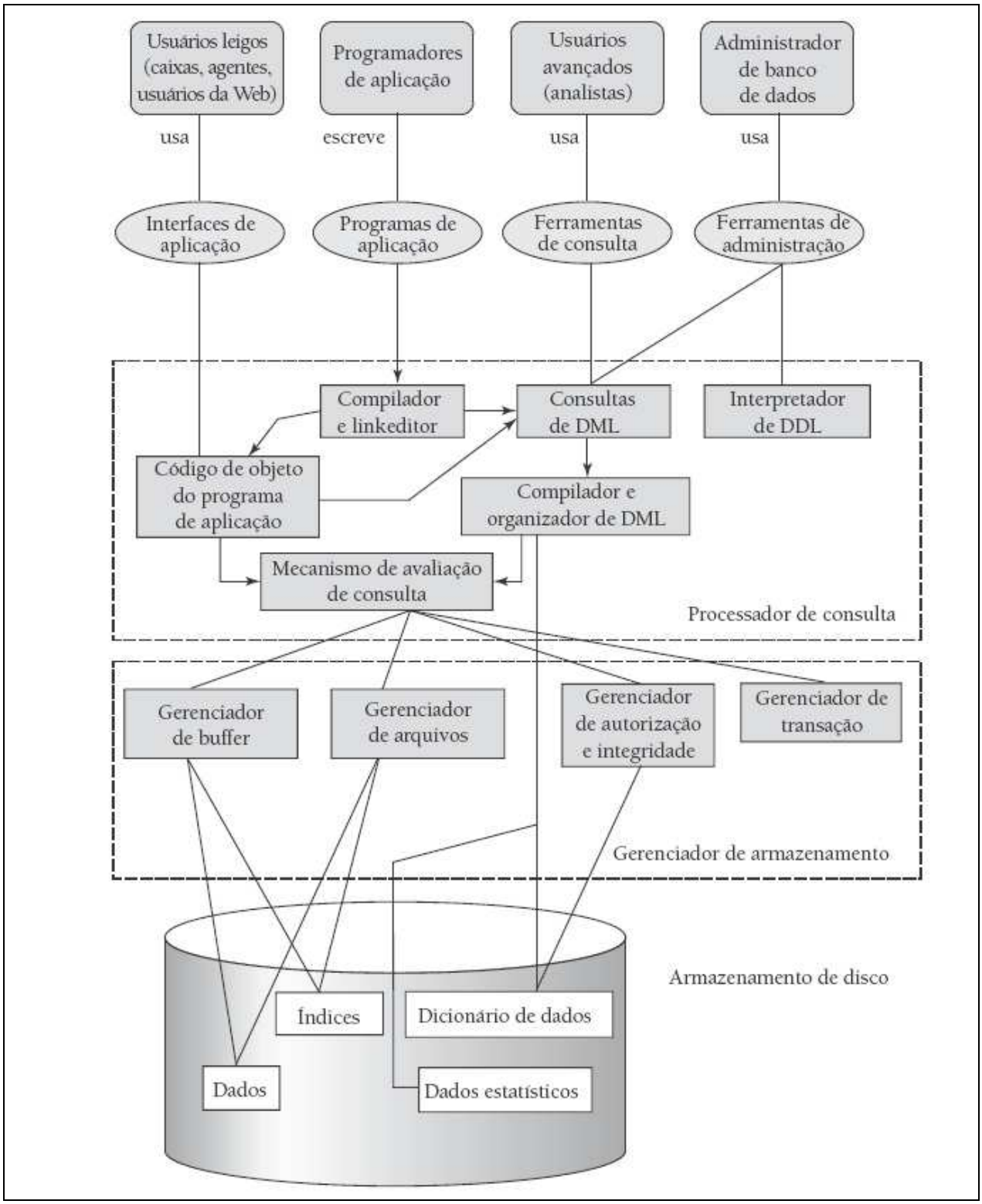
2.7.2. Processador de Consultas

Os componentes do processador de consultas incluem:

- **Interpretador DML**, que interpreta os comandos DDL e registra as definições no dicionário de dados.
- **Compilador de DML**, que traduz instruções DML em uma linguagem de consulta para um plano de avaliação consistindo em instruções de baixo nível que o mecanismo de avaliação de consulta entende. Uma consulta normalmente pode ser traduzida em qualquer um de vários planos de avaliação que produzem todos o mesmo resultado. O compilador de DML também realiza otimização de consulta; ou seja, ele seleciona o plano de avaliação de menor custo dentre as alternativas.
- **Mecanismo de avaliação de consulta,** que executa instruções de baixo nível geradas pelo compilador de DML.

[2] Estas propriedades também são chamadas de propriedades ACID.

Figura 6 - Estrutura de um Sistema Gerenciador de Banco de Dados



2.8. Classificação dos SGBDs

O principal critério para se classificar um SGBD é o modelo de dados no qual é baseado. A grande maioria dos SGBDs contemporâneos são baseados no modelo relacional, alguns em modelos conceituais e alguns em modelos orientados a objetos. Outras classificações são:

- **Usuários**: um SGBD pode ser mono-usuário, comumente utilizado em computadores pessoais ou multi-usuários, utilizado em estações de trabalho, mini-computadores e máquinas de grande porte;

- **Localização**: um SGBD pode ser localizado ou distribuído; se ele for localizado, então todos os dados estarão em uma máquina (ou em um único disco) ou distribuído, onde os dados estarão distribuídos por diversas máquinas (ou diversos discos);

- **Ambiente**: ambiente homogêneo é o ambiente composto por um único SGBD e um ambiente heterogêneo é o ambiente compostos por diferentes SGBDs.

2.9. Arquiteturas de Banco de Dados

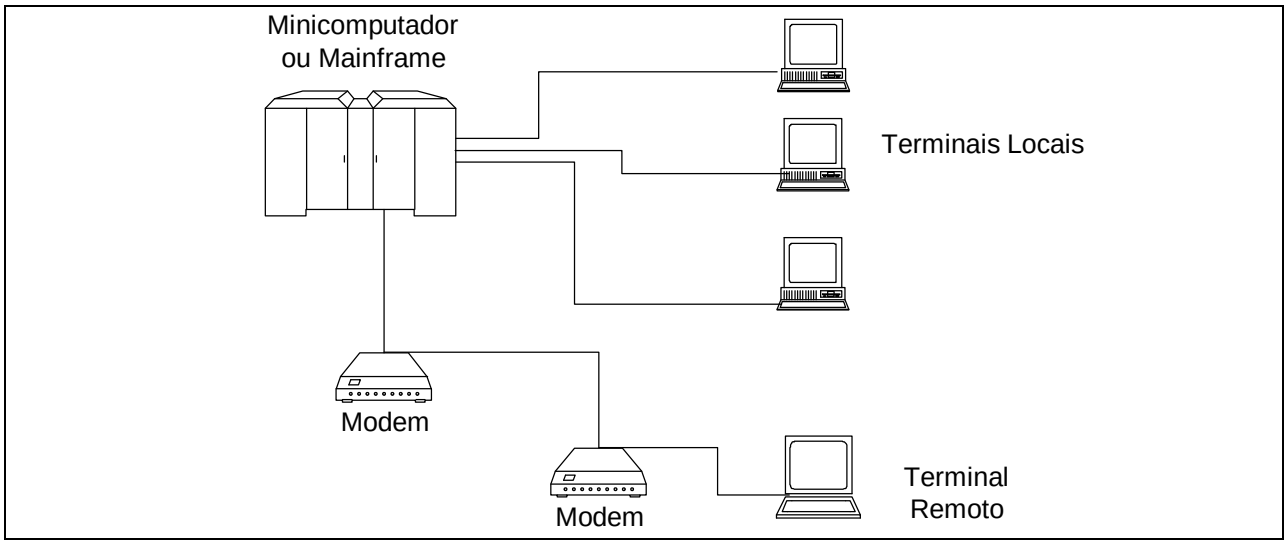
O tipo de sistema computador no qual rodam os bancos de dados pode ser dividido em quatro categorias ou plataformas: Centralizada, PC, Cliente/Servidor, Distribuído e Paralelo. Os quatro diferem, principalmente no local onde realmente ocorre o processamento dos dados. A arquitetura do próprio SGBD não determina, necessariamente, o tipo de sistema computador no qual o banco de dados precisa rodar; contudo, certas arquiteturas são mais convenientes (ou mais comuns) para algumas plataformas do que para outras.

2.9.1. Plataformas Centralizadas

Em um sistema centralizado, todos os programas rodam em um computador "hospedeiro" principal, incluindo o SGBD, os aplicativos que fazem acesso ao banco de dados e as facilidades de comunicação que enviam e recebem dados dos terminais dos usuários.

Os usuários têm acesso ao banco de dados através de terminais conectados localmente ou discados (remotos), conforme aparece na Figura 7.

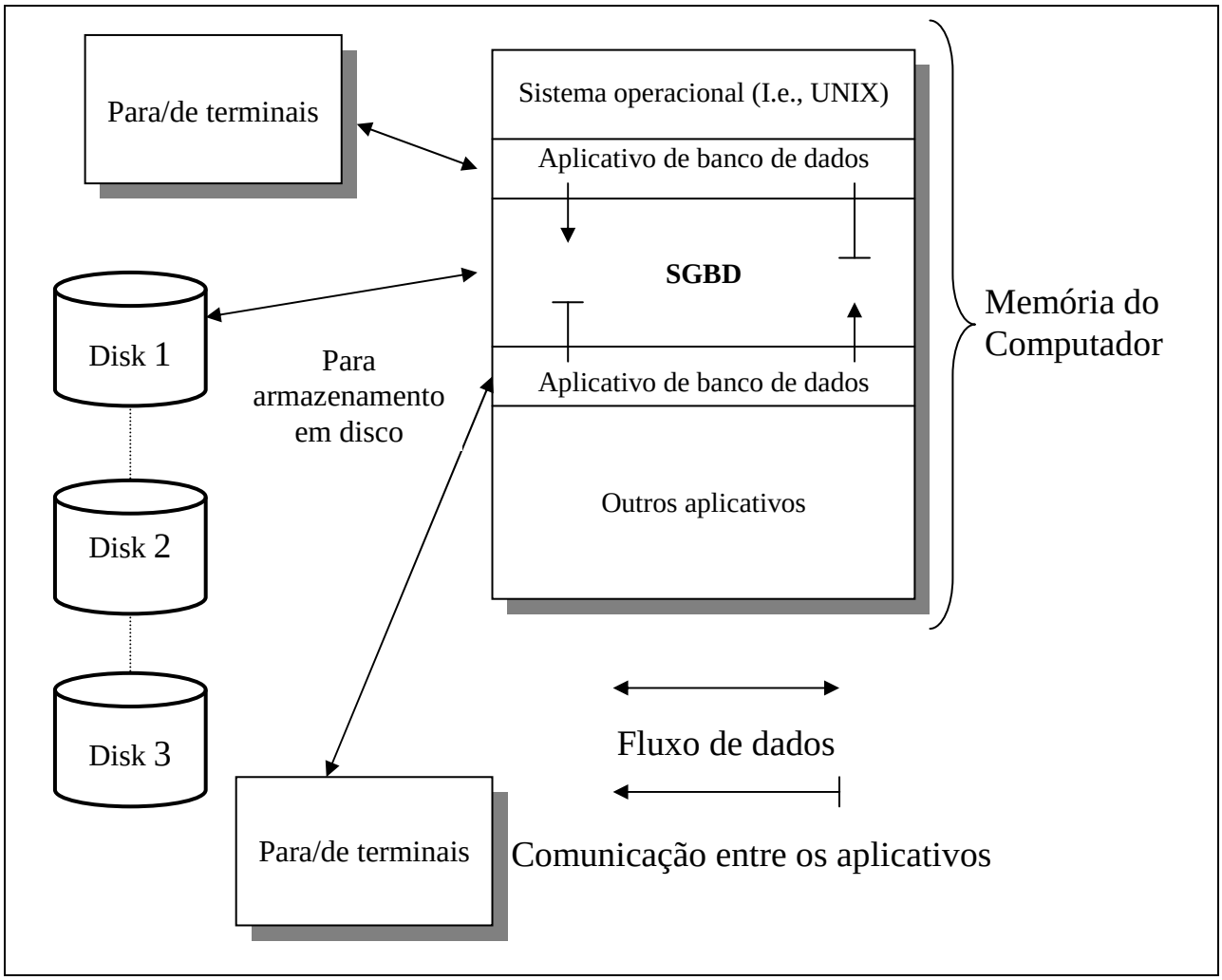
Figura 7 – Arquitetura Centralizada



Geralmente os terminais são "mudos", tendo pouco ou nenhum poder de processamento e consistem somente de uma tela, um teclado e do hardware para se comunicar com o hospedeiro. O advento dos microprocessadores levou ao desenvolvimento de terminais mais inteligentes, onde o terminal compartilha um pouco da responsabilidade de manipular o desenho da tela e a entrada do usuário. Embora os sistemas de mainframe e de minicomputador sejam as plataformas principais para sistemas de banco de dados de grandes empresas, os baseados em PC também se podem se comunicar com sistemas centralizados através de combinações de hardware/software que emulam(imitam) os tipos de terminais utilizados com um hospedeiro em particular.

Todo o processamento de dados de um sistema centralizado acontece no computador hospedeiro e o SGBD deve estar rodando antes que qualquer aplicativo possa ter acesso ao banco de dados. Quando um usuário liga um terminal, normalmente vê uma tela de log-in; o usuário introduz um ID de conexão e uma password, a fim de ter acesso aos aplicativos do hospedeiro. Quando o aplicativo de banco de dados é inicializado, ele envia a informação de tela apropriada para o terminal e responde com ações diferentes, baseadas nos toques de tecla dados pelo usuário. O aplicativo e SGBD, ambos rodando no mesmo hospedeiro, se comunicam pela área de memória compartilhada ou de tarefa do aplicativo, que são gerenciadas pelo sistema operacional do hospedeiro. O SGBD é responsável pela movimentação dos dados nos sistemas de armazenamento de disco, usando os serviços fornecidos pelo sistema operacional. A Figura 8 apresenta um modo possível de interação desses aplicativos: os aplicativos se comunicam com os usuários pelos terminais e com o SGBD; o SGBD se comunica com os dispositivos de armazenamento (que podem ser discos rígidos, mas não estão limitados a isso) e com os aplicativos.

Figura 8 - Compartilhamento da Memória em Sistemas Centralizados



O SGBD que roda no sistema hospedeiro pode ser baseado em qualquer um dos quatro modelos. Contudo, os modelos hierárquico e relacional são os mais comuns. Nos mainframes, o SGBD é normalmente baseado no IMS da IBM, que é um banco de dados hierárquico. Recentemente, entretanto, mais e mais mainframes estão rodando SGBDs baseados no modelo relacional, principalmente o DB2 da IBM.

As principais vantagens de um sistema centralizado são a segurança centralizada e a capacidade de manipular enormes quantidades de dados em dispositivos de armazenamento. Os sistemas centralizados também podem suportar vários usuários simultaneamente; é comum, para um banco de dados em um mainframe IBM, suportar até 1000 usuários de uma vez. As desvantagens geralmente estão relacionadas aos custos de aquisição e manipulação desses sistemas. Os grandes sistemas de mainframe e de minicomputadores exigem facilidades de suporte específicas, como o habitual centro de dados com pisos elevados, sistemas de refrigeração e grandes sistemas de controle climático. Normalmente é necessário um staff de operadores e programadores altamente treinados para manter o sistema ativo e funcionando, com consideráveis custos adicionais de pessoal. Finalmente, o preço de aquisição de hardware de grandes sistemas centralizados freqüentemente atinge milhões de dólares e a manutenção é também onerosa.

Atualmente, as empresas têm optado cada vez mais por minicomputadores dimensionados para departamento, como o Micro Vax da DEC e o AS/400 da IBM, pois não custam tanto e são bem suportados como sistemas centralizados, e , geralmente, não exigem um ambiente especial. Esses sistemas são mais convenientes para pequenas empresas com poucos usuários (não mais do que 200) ou

para aplicativos de banco de dados que só interessam a um único departamento de uma grande empresa (isto é um minicomputador que roda aplicativos de engenharia só pode interessar ao departamento de projetos). Os computadores menores também pode ser colocado em rede com outros minicomputadores e mainframes, para que todos os computadores possam compartilhar dados.

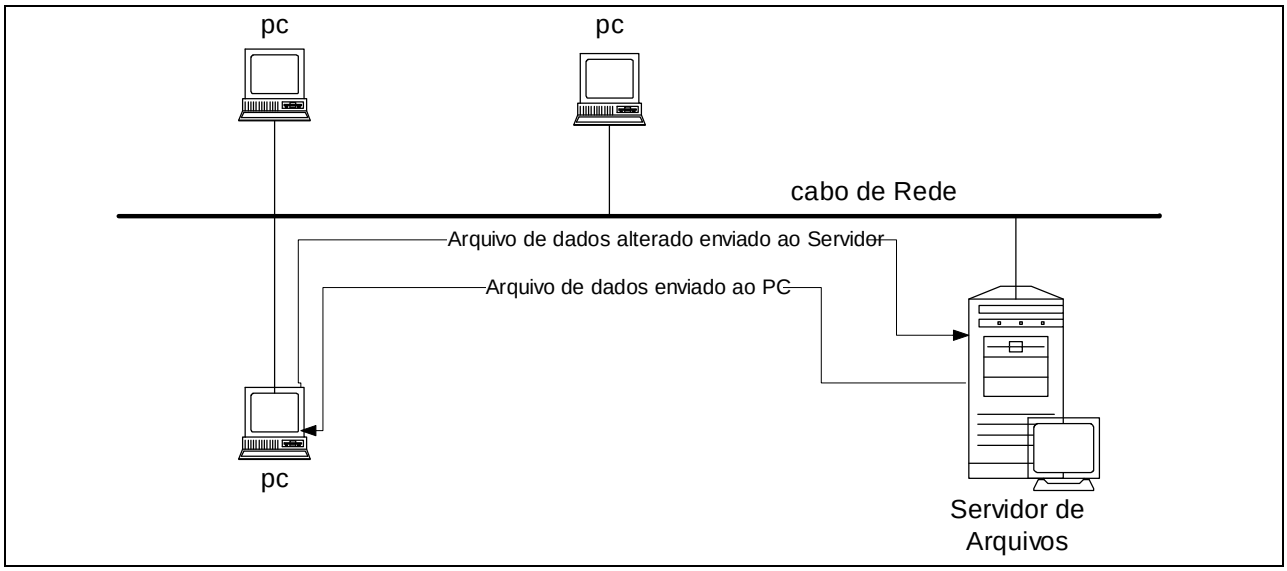
2.9.2. Sistemas de Computador Pessoal

Os computadores pessoais (PCs) apareceram no final dos anos 70 e revolucionaram a maneira de ver e utilizar computadores. Um dos primeiros sistemas operacionais de sucesso para PCs foi c CP/M(Control Program for Microcomputers), da Digital. O primeiro SGBD baseado em PC de sucesso, dBase II, da Ashtan-Tate, rodava sob CP/M. Quando a IBM lançou o primeiro PC baseado no MS-DOS, em 1981, a Ashton-Tate portou o dBASE para o novo sistema operacional. Desde então, o dBASE gerou versões mais novas, compatíveis e parecidas e SGBDs competitivos que têm provado, à comunidade de processamento de dados, que os PCs podem executar muitas das tarefas dos grandes sistemas.

Quando um SGBD roda em um PC, este atua como computador hospedeiro e terminal. Ao contrário dos sistemas maiores, as funções do SGBD e do aplicativo de banco de dados são combinadas em um único aplicativo. Os aplicativos de banco de dados em um PC manipulam a entrada do usuário, a saída da tela e o acesso aos dados do disco. Combinar essas diferentes funções em uma unidade, dá ao SGBD muito poder, flexibilidade e velocidade; contudo, normalmente ao custo da diminuição da segurança e da integridade dos dados. Os PCs originaram-se como sistemas stand-alone, mas, recentemente, muitos têm sido conectados em redes locais (LANs). Em uma LAN, os dados, e normalmente os aplicativos do usuário, residem no servidor de arquivo um PC que roda um sistema operacional de rede (NOS) especial, como o NetWare da Novell ou o LAN Manager da Microsoft. O servidor de arquivo gerencia o acesso aos dados compartilhados pelos usuários da rede em seus discos rígidos e, freqüentemente, da acesso a outros recursos compartilhados, como impressoras .

Embora uma LAN permita aos usuários de bancos de dados baseados em PC, compartilhar arquivos de dados comuns, ela não muda o funcionamento do SGBD significativamente. Todo o processamento de dados real ainda é executado no PC que roda o aplicativo de banco de dados. O servidor de arquivo somente procura em seus discos os dados necessários para o usuário e envia esses dados para o PC, através do cabo da rede. Os dados são, então, processados pelo SGBD que está rodando no PC e quaisquer mudanças no banco de dados exige, do PC, o envio de todo o arquivo de dados de volta ao servidor de arquivo, para ser novamente armazenado no disco. Essa troca esta mostrada na Figura 9. Embora o acesso de vários usuários a dados compartilhados seja uma vantagem, existe uma desvantagem significativa, de um SGBD baseado em LAN, relativa à rapidez ou ao poder do servidor de arquivo, terem seu desempenho limitado pelo poder do PC que está rodando o SGBD real. Quando vários usuários estão tendo acesso ao banco de dados, os mesmos arquivos precisam ser enviados do servidor para cada PC que está tendo acesso a eles. Esse tráfego ampliado pode diminuir a velocidade da rede.

Figura 9 - Sistemas de Computador Pessoal



A única melhoria necessária para um SGBD multiusuário, em relação a um monousuário, é a capacidade de manipular, simultaneamente, as alterações dos dados realizados por vários usuários. Normalmente isso é feito por algum tipo de esquema de bloqueio, no qual o registro ou o arquivo de dados que um usuário está atualizando ou alterando, é bloqueado para evitar que os outros usuários também o alterem. A maioria dos SGBDs, baseada em LAN, disponível hoje em dia, são simplesmente versões multiusuários de sistemas de banco de dados stand-alone comuns, contudo, os tipos de esquemas de bloqueio variam bastante e podem afetar significativamente o desempenho de um banco de dados multiusuário.

A maioria dos SGBDs baseada em PC é projetada no modelo relacional, mas o fato de o SGBD não estar separado do aplicativo de banco de dados significa que muitos (senão a maioria) dos princípios relacionais não estão implementados. Os componentes ausentes mais notáveis são os que tratam da integridade dos dados. A maior parte dos bancos de PC permite acesso direto aos arquivos de dados, fora do SGBD que os criou. Isso cria uma situação, na qual podem ser feitas alterações nos arquivos, violadores Das regras pelas quais o aplicativo assegura a integridade dos dados. Tal violação pode até tornar ilegível o arquivo de dados para o DBMS. Por essa razão os bancos de dados de PC baseados num modelo relacional, são descritos mais precisamente, como semi-relacionais. Alguns dos bancos de dados de PC semi-relacionais mais comuns, disponíveis hoje em dia, incluem o R:Base da Microrim, o dBASE IV da Borland (a Borland adquiriu a Ashton-Tate no final de 1991) e seus muitos "clones", como o FoxPro da Microsoft, o Paradox da Borland, o DataEase da DataEase International e o Advanced Revelation, da Revelation Technologies.

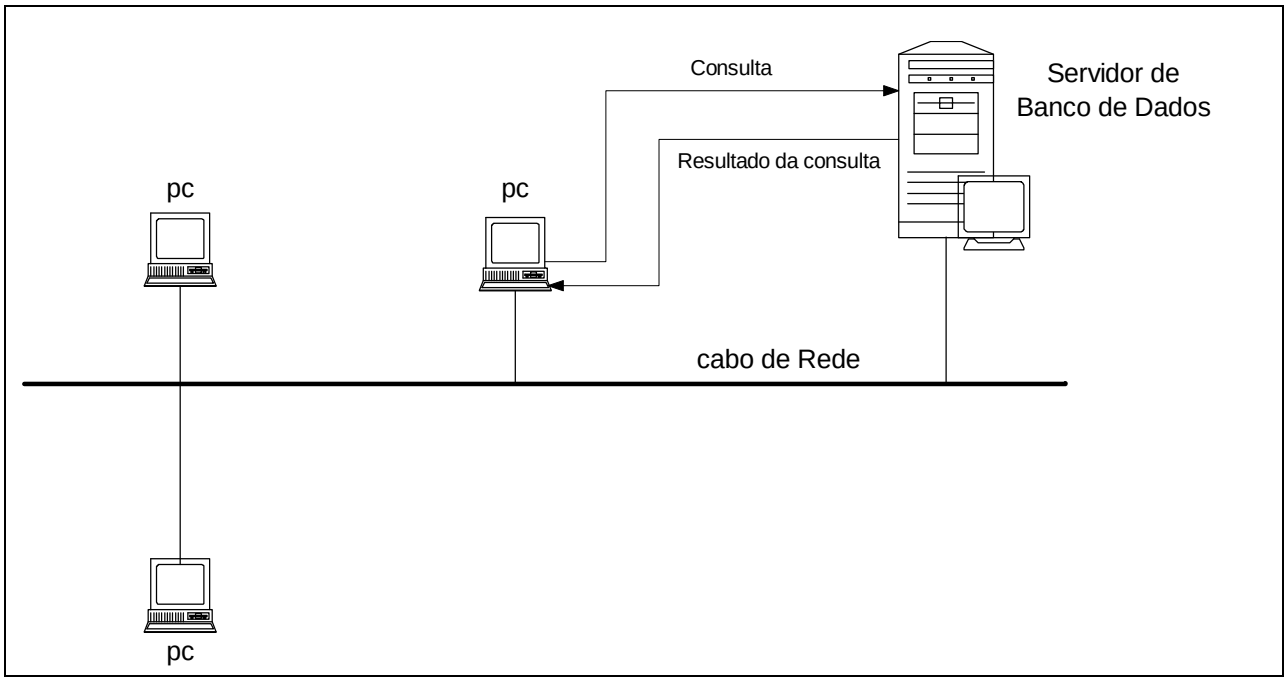
Conforme mencionado anteriormente, os bancos de dados de PC mais limitados, normalmente são baseados no modelo do sistema de gerenciamento de arquivo. Também existem SGBDs, baseados em PC, derivados do modelo em rede como o DataFlex, da Data Access Corporation e o db- Vista III, da Raima Corporation.

A maioria dos sistemas de banco de dados multiusuário baseado em PC, manipula o mesmo número de usuários dos sistemas centralizados menores. Entretanto, os problemas decorrentes da manipulação de várias transações simultâneas, do aumento no tráfego da rede e do limite do poder de processamento dos PCs que rodam o SGBD, provocam o aumento da complexibilidade e a degradação no desempenho, à medida que o número de usuários se multiplica. A solução desenvolvida para essas limitações é o sistema de banco de dados Cliente/Servidor.

2.9.3. Bancos de Dados Cliente/Servidor

Na sua forma mais simples, um banco de dados Cliente/Servidor (C/S) divide o processamento de dados em dois sistemas: o PC cliente, que roda o aplicativo de banco de dados, e o servidor, que roda totalmente ou parte do SGBD real. O servidor de arquivo da rede local continua a oferecer recursos compartilhados, como espaço em disco para os aplicativos e impressoras. O servidor de banco de dados pode rodar no mesmo PC do servidor de arquivo ou (como é mais comum) em seu próprio PC. O aplicativo de banco de dados do PC cliente, chamado *sistema front-end*, manipula toda a tela e o processamento de entrada/saída do usuário. O sistema *back-end* do servidor de banco de dados manipula o processamento dos dados e o acesso ao disco. Por exemplo, um usuário do front-end gera um pedido (consulta) de dados do servidor de banco de dados, e o aplicativo front-end envia, para o servidor, o pedido pela rede. O servidor de banco de dados executa a pesquisa real e retorna somente os dados que respondem a pergunta do usuário, conforme aparece na Figura 10.

Figura 10 - Sistemas Cliente / Servidor



A vantagem imediata de um sistema C/S é óbvia: dividir o processamento entre dois sistemas reduz a quantidade do tráfego de dados no cabo da rede.

Em um dos casos tipicamente confusos sobre o significado de um mesmo termo que às vezes encontramos no campo da computação, a definição de Cliente/Servidor é aparentemente o contrário dos sistemas baseados em UNIX, rodando a interface gráfica X-Windows. A divisão no processamento é a mesma do sistema C/S baseado em PC, mas o front-end é chamado servidor no X-Windows, pois fornece os serviços de apresentação e de interface do usuário. O sistema back-end, no qual roda o SGBD, é referido como cliente dos serviços fornecidos pelo sistema front-end.

O número de sistemas C/S está aumentando rapidamente-novos sistemas estão sendo projetados e divulgados quase mensalmente. Embora os sistemas clientes normalmente rodem em PC, o sevidor do banco de dados pode rodar de um PC a um mainframe. Mais e mais aplicativos de front-end estão aparecendo, incluindo desde os que ampliam o escopo dos SGBDs baseados em PC tradicionais, até os servidores de banco de dados.

A maior desvantagem dos sistemas de bancos de dados descritos até aqui é que eles exigem o armazenamento dos dados em um único sistema. Isso pode ser um problema para empresas grandes que

precisam suportar usuários do banco de dados espalhados em uma área geográfica extensa ou que precisem compartilhar parte de seus dados departamentais com outros departamentos ou com um hospedeiro central. É necessário um modo de distribuir os dados entre os vários hospedeiros ou localidades, o que levou ao desenvolvimento dos sistemas de processamento distribuído.

2.9.4. Sistemas de Processamento Distribuído

Ozsu e Tamer(2001) definem um banco de dados distribuído como uma coleção de vários bancos de dados logicamente inter-relacionados, distribuídos por uma rede de computadores. Um *sistema de gerenciamento de banco de dados distribuído* (SGBD Distribuído) é definido então como o sistema de software que permite o gerenciamento de banco de dados distribuído e que torna a distribuição transparente para os usuários. Algumas vezes, a expressão "*sistema de banco de dados distribuídos*" (SBDD) é empregada para se referir em conjunto ao banco de dados distribuído e ao SGBD distribuído. As duas expressões importantes nessas definições são "logicamente inter-relacionados" e "distribuídos por uma rede de computadores". Elas ajudam a eliminar certos casos que as vezes são aceitos como representações de um SBDD.

A distribuição física dos arquivos do banco gera problemas que não existem em um sistema onde os arquivos residem em um mesmo computador. Com isto sistemas multiprocessados não são distribuídos, são ditos multiprocessados. Estes sistemas compartilham algum tipo de memória, seja a memória primária (*memória compartilhada*, estreitamente acoplado, Figura 11) ou memória secundária (*disco compartilhado*, livremente acoplado, Figura 12).

Figura 11 – Multiprocessador de memória compartilhada

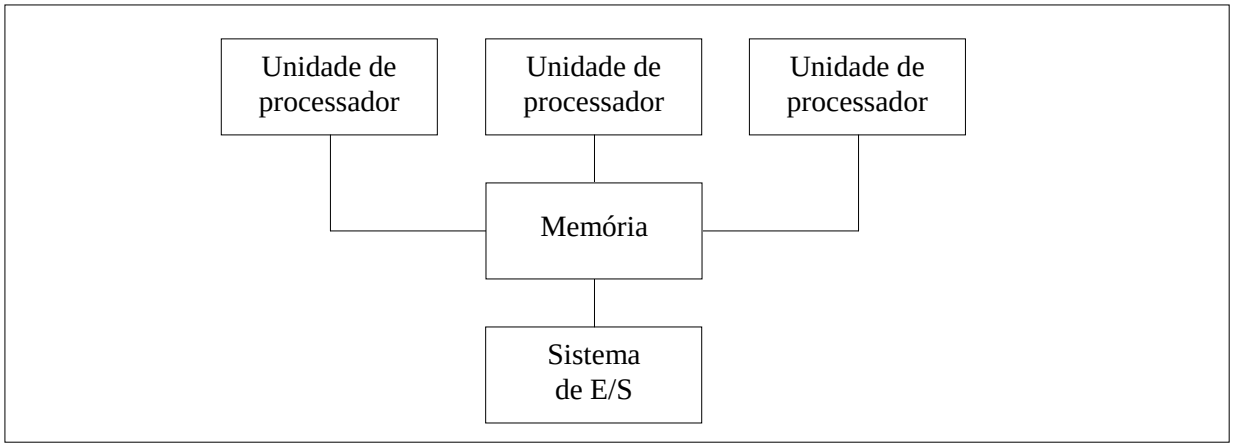
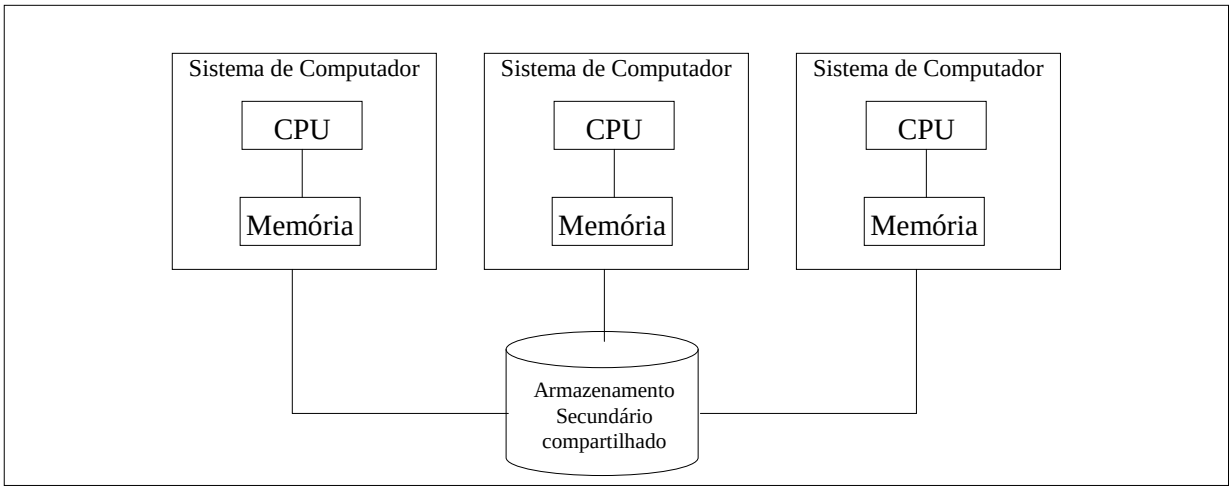
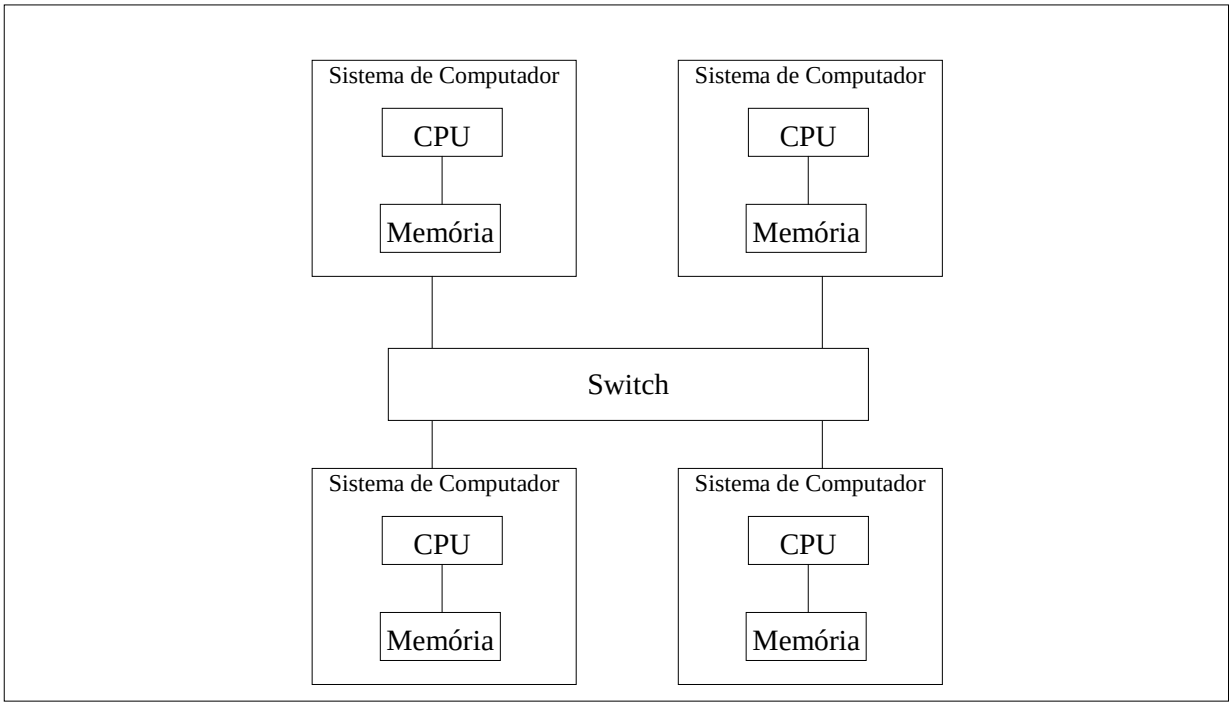


Figura 12 – Multiprocessador de disco compartilhado



Outra distinção comumente feita neste contexto ocorre entre as arquiteturas de *tudo compartilhado* e *nada compartilhado*. O primeiro modelo arquitetônico permite que cada processador tenha acesso a tudo(memórias primárias e secundárias, e ainda periféricos) no sistema . Arquitetura nada compartilhada é aquela em que cada processador tem sua própria memória primária e secundária (Figura 13).

Figura 13 – Sistema de multiprocessador de nada compartilhado



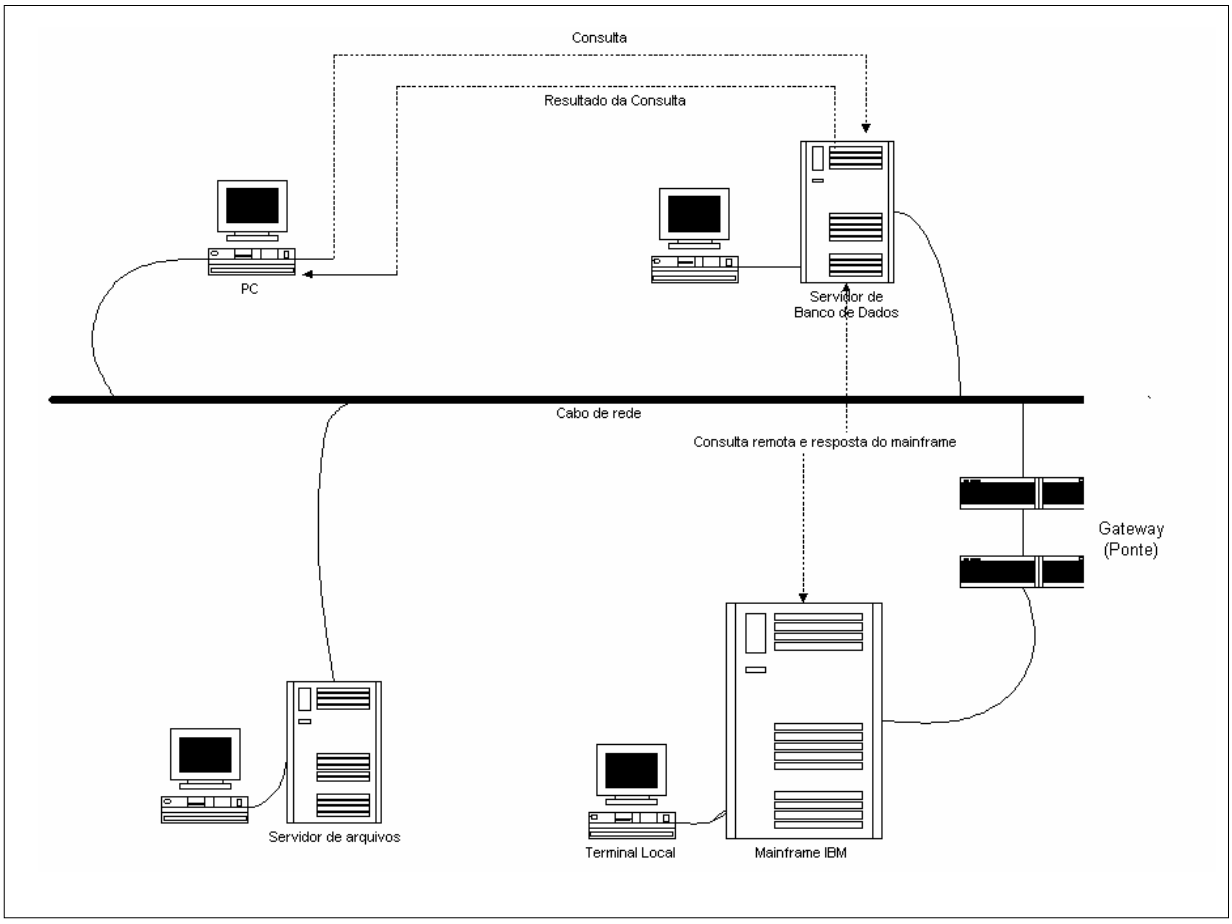
Uma forma simples de processamento distribuído já existe a alguns anos. Nessa forma limitada, os dados são compartilhados entre vários sistemas hospedeiros, através de atualizações enviadas pelas conexões diretas (na mesma rede) ou por conexões remotas, via telefone ou via linhas de dados dedicadas. Um aplicativo rodando em um ou mais dos hospedeiros, extras a parte dos dados alterados durante um período de tempo definido pelo programador e, então, transmite os dados para um hospedeiro centralizado ou para outros hospedeiros do circuito distribuído. Os outros bancos de dados são, então, atualizados para que todos os sistemas estejam sincronizados. Esse tipo de processamento de dados distribuído normalmente ocorre entre computadores departamentais ou entre LANs e sistemas hospedeiros. Após o dia de trabalho, os dados vão para um grande microcomputador central ou para um hospedeiro mainframe.

Embora esse sistema seja ideal para compartilhar parte dos dados entre diferentes hospedeiros, ele não responde ao problema do acesso, pelo usuário, aos dados não armazenados em seus hospedeiros locais. Os usuários devem mudar suas conexões para os diferentes hospedeiros, a fim de ter acesso aos vários bancos de dados, lembrando-se, entretanto, de onde está cada banco. Combinar os dados dos bancos de dados existentes em hospedeiros, também apresenta alguns sérios desafios para os usuários e para os programadores. Há ainda, o problema dos dados duplicados; embora os sistemas de armazenamento em disco tenham diminuído de preço através dos anos, fornecer vários sistemas de disco para armazenar os mesmos dados pode ficar caro. Manter todos os conjuntos de dados duplicados em sincronismo aumenta a complexidade do sistema.

A solução para esses problemas está emergindo da tecnologia do acesso "sem costura" a dados, denominada *processamento distribuído*. No sistema de processamento distribuído o usuário pede dados do hospedeiro local; se este informar que não possui os dados, sai pela rede procurando o sistema que os tenha. Em seguida, retorna os dados ao usuário, sem que este saiba que foram trazidos de um sistema desconhecido exceto, talvez, por um ligeiro atraso na obtenção dos dados. A ilustra uma forma de sistema de processamento distribuído.

Primeiramente, o usuário cria e envia uma busca de dados para o servidor do banco de dados local. O servidor, então, envia, para o mainframe (possivelmente através de um gateway ou de um sistema de ponte que une as duas redes), o pedido dos dados que não possui. Ele responde à consulta. Finalmente, o servidor do banco de dados local combina esse resultado com os dados encontrados em seu próprio disco e retorna a informação ao usuário.

Figura 14 - Sistemas Distribuídos



O ideal é que esse sistema distribuído também possa funcionar de outro modo: os usuários de terminal conectado diretamente ao mainframe podem ter acesso aos dados existentes nos servidores de

arquivos remotos. O projeto e a implementação dos sistemas de processamento distribuído é um campo muito novo. Muitas partes ainda não estão no lugar e as soluções existentes nem sempre são compatíveis uma com as outras.

2.9.5. Sistemas de Processamento Paralelo

Sistemas paralelos melhoram as velocidades de processamento e E/S usando várias CPUs e discos em paralelo. As máquinas paralelas estão se tornando cada vez mais comuns, tornando o estudo de sistemas de banco de dados paralelos cada vez mais importante. A força motriz por trás dos sistemas de banco de dados paralelos é a demanda de aplicações que precisam consultar bancos de dados extremamente grandes (da ordem de terabytes) ou que tenham de processar um número extremamente grande de transações por segundo (da ordem de milhares por segundo). Os sistemas de banco de dados centralizados e cliente-servidor não são poderosos o suficiente para lidar com tais aplicações.

No processamento paralelo, muitas operações são realizadas simultaneamente, ao contrário do processamento serial, em que as etapas computacionais são realizadas sequencialmente. Uma máquina paralela com granularidade grossa (coarse grain) consiste em um pequeno número de processadores poderosos; uma máquina maciçamente paralela ou paralela com granularidade fina(fine grain) utiliza milhares de processadores menores. A maior parte das máquinas de alto nível hoje em dia fornece algum grau de paralelismo com granularidade grossa; máquinas com dois ou quatros processadores são comuns. Computadores maciçamente paralelos podem ser distinguidos de máquinas paralelas de granularidade grossa pelo grau muito maior de paralelismo que eles administram. Computadores paralelos, com centenas de CPUs, e discos, estão disponíveis comercialmente.

Existem duas medidas principais de desempenho de um sistema de banco de dados: (1) throughput, o número de tarefas que podem ser completadas em determinado intervalo de tempo, e (2) tempo de resposta, a quantidade de tempo necessária para completar uma única tarefa desde o momento em que ela foi submetida. Um sistema que processa um grande número de transações pequenas pode melhorar o throughput processando muitas transações em paralelo. Um sistema que processa grandes transações pode melhorar o tempo de resposta e também o throughput realizando subtarefas de cada transação em paralelo.

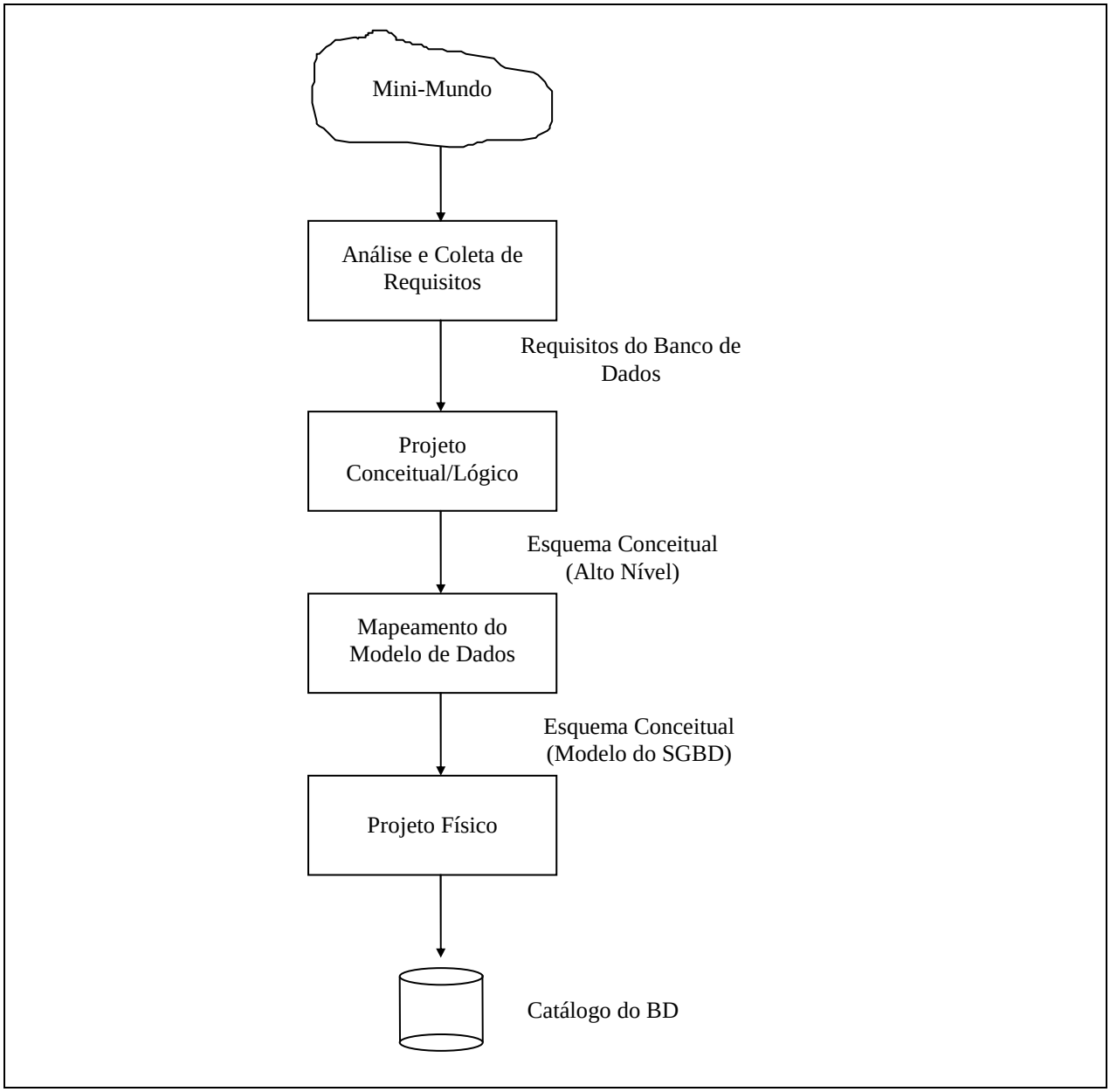
3. Modelagem de Dados Utilizando o Modelo Entidade Relacionamento (ER)

O modelo Entidade-Relacionamento 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 ER é utilizado principalmente durante o processo de projeto de banco de dados.

3.1. Modelo de Dados Conceitual de Alto Nível

A Figura 15 faz uma descrição simplificada do processo de projeto de um banco de dados.

Figura 15 - Fases do Projeto de um Banco de Dados



3.2. Entidades e Atributos

O objeto básico tratado pelo modelo ER é a "**entidade**", que pode ser definida como um objeto do mundo real, concreto ou abstrato e que possui existência independente. Cada entidade possui um conjunto particular de propriedades 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**". Um atributo que não pode ser subdividido é chamado de "atributo simples" ou "atômico".

Os **atributos** que podem assumir apenas um determinado valor em uma determinada instância é denominado "**atributo simplesmente valorado**", enquanto que um atributo que pode assumir diversos valores em uma mesma instância é denominado "**multivalorado**".

Um atributo que é gerado a partir de outro atributo é chamado de "**atributo derivado**".

3.3. Tipos de Entidade, Conjunto de Entidade, Atributo Chave e Conjunto de Valores

Um banco de dados costuma conter grupos de entidades que são similares, possuindo os mesmos atributos, porém, cada entidade com seus próprios valores para cada atributo. Este conjunto (ou coleção) de entidades similares define um "**tipo de entidade**". Cada tipo 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. A coleção de todas as entidades de um determinado tipo entidade no banco de dados em qualquer ponto do tempo é chamada de "**conjunto de entidades**"; geralmente nos referimos ao conjunto de entidades utilizando o mesmo nome do tipo de entidade.

Uma restrição muito importante em uma entidade de um determinado tipo entidade é a "**chave**" ou "**restrição de unicidade**" nos atributos. 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.

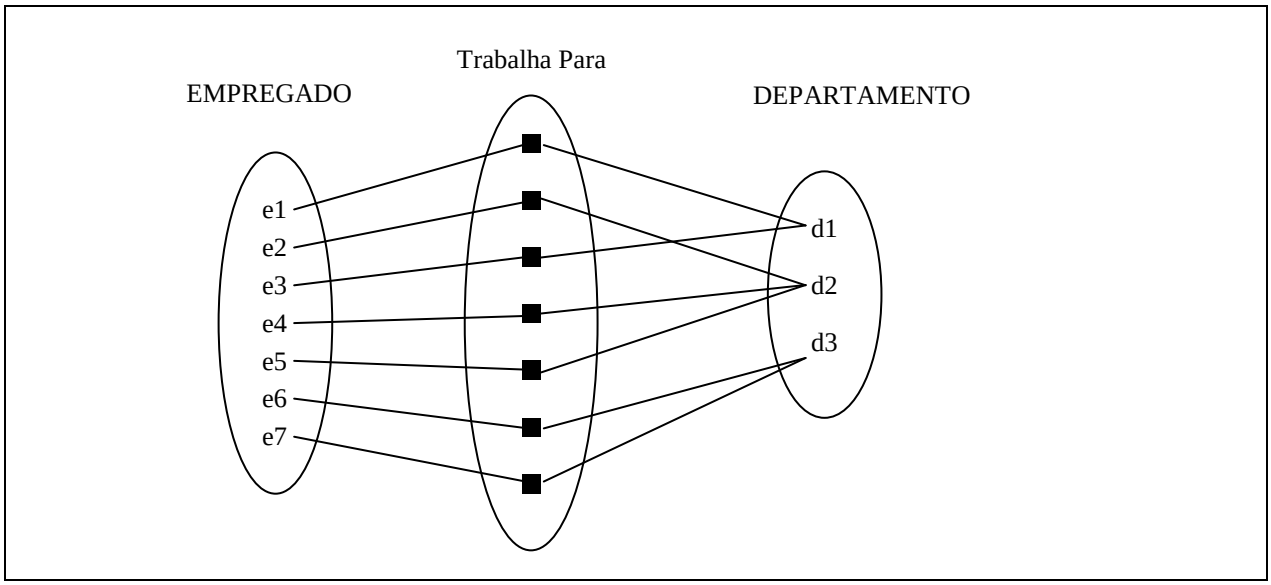
Cada atributo simples de um tipo entidade está associado com um "**conjunto de valores**" denominado "**domínio**", o qual especifica o conjunto de valores (ou domínio de valores) que podem ser designados para este determinado atributo para cada entidade.

3.4. Tipos, Conjuntos e Instâncias de Relacionamento

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

Um "**tipo de relacionamento**" **R** entre *n* entidades **E1, E2, ..., En**, é um conjunto de associações(ou um **conjunto de relacionamento**s) entre entidades deste tipo. Informalmente falando, cada **instância de relacionamento r1** 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 16 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 16 - Exemplo de um Relacionamento



3.5. 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 16, temos um relacionamento binário. 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.

3.6. Outras Características de um Relacionamento

3.6.1. Relacionamentos como Atributos

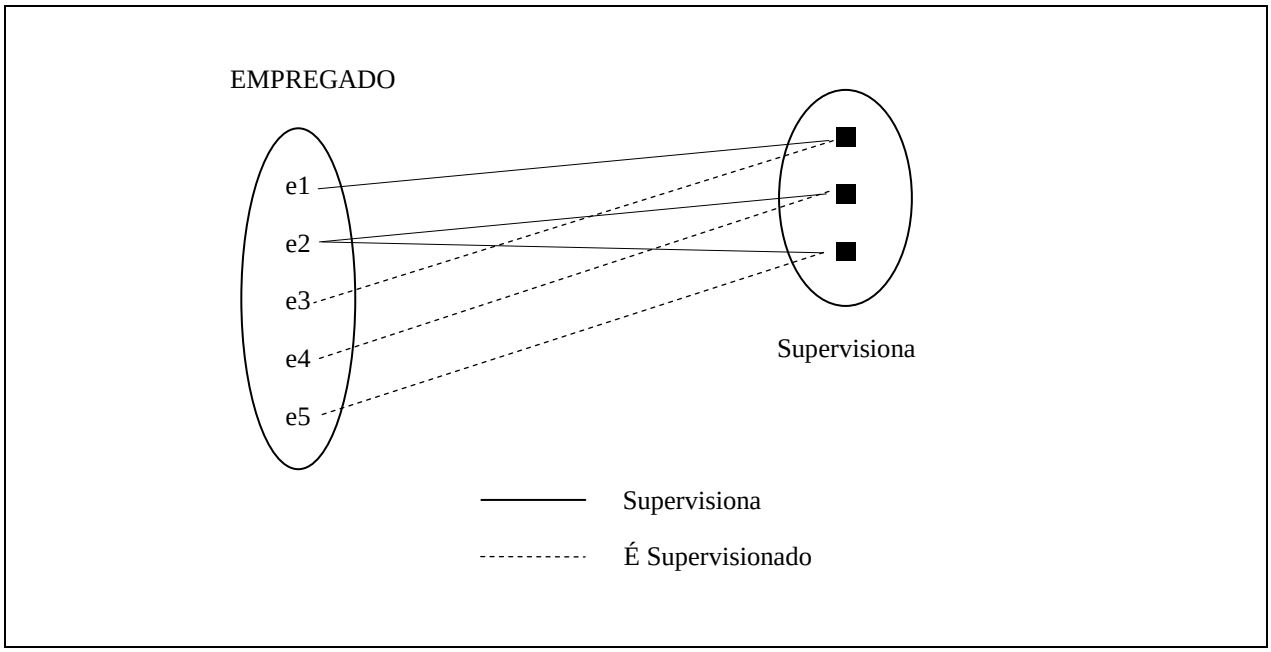
Algumas vezes é conveniente pensar em um relacionamento como um atributo. Considere o exemplo da Figura 16. Podemos pensar departamento como sendo um atributo da entidade empregado, ou empregado, como um atributo multivalorado da entidade departamento. Se uma entidade não possuir existência muito bem definida, talvez seja mais interessante para a coesividade do modelo lógico que ela seja representada como um atributo.

3.6.2. 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 16, nós temos o papel empregado ou trabalhador para o tipo entidade EMPREGADO e o papel departamento ou empregador para a 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**" (ou **auto-relacionamento**).

Um relacionamento recursivo é um relacionamento entre entidades do mesmo tipo entidade. Veja o exemplo da Figura 17.

Figura 17 - Um Relacionamento Recursivo

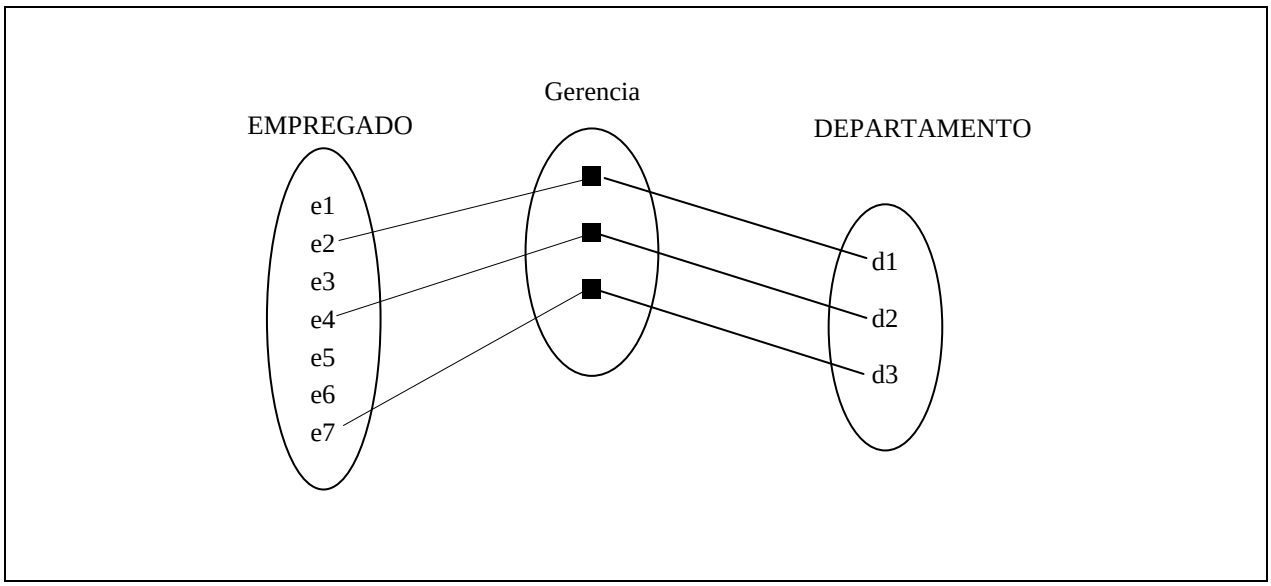


No exemplo, temos um relacionamento entre o tipo entidade EMPREGADO, onde um empregado pode supervisionar outro empregado e um empregado pode ser supervisionado por outro empregado.

3.6.3. Restrições em Tipos Relacionamentos

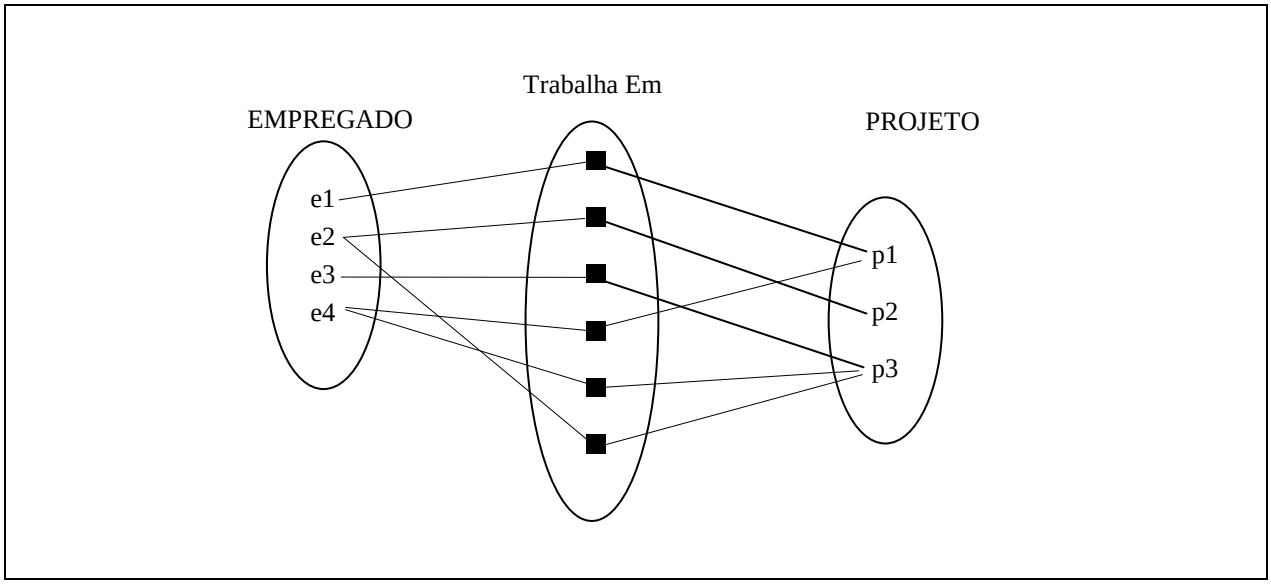
Geralmente, os tipos relacionamentos sofrem certas restrições que limitam as possíveis combinações das entidades participantes. Estas restrições são derivadas de restrições impostas pelo estado destas entidades no mini-mundo. Veja o exemplo da Figura 18.

Figura 18 - Relacionamento EMPREGADO gerencia DEPARTAMENTO



No exemplo da Figura 18, 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 dos quais uma entidade pode participar. A cardinalidade pode ser: 1:1, 1:N, M:N. No exemplo da Figura 18, a cardinalidade é 1:1, pois cada entidade empregado pode gerenciar apenas um departamento e um departamento pode ser gerenciado por apenas um empregado. No exemplo da Figura 16, no relacionamento EMPREGADO Trabalha Para DEPARTAMENTO, o relacionamento é 1:N, pois um empregado pode trabalhar em apenas um departamento, enquanto que um departamento pode possuir vários empregados. Na Figura 19 temos um exemplo de um relacionamento com cardinalidade N:M.

Figura 19 - Relacionamento N:M



No exemplo da Figura 19, nós temos que um empregado pode trabalhar em vários projetos enquanto que um projeto pode ter vários empregados trabalhando.

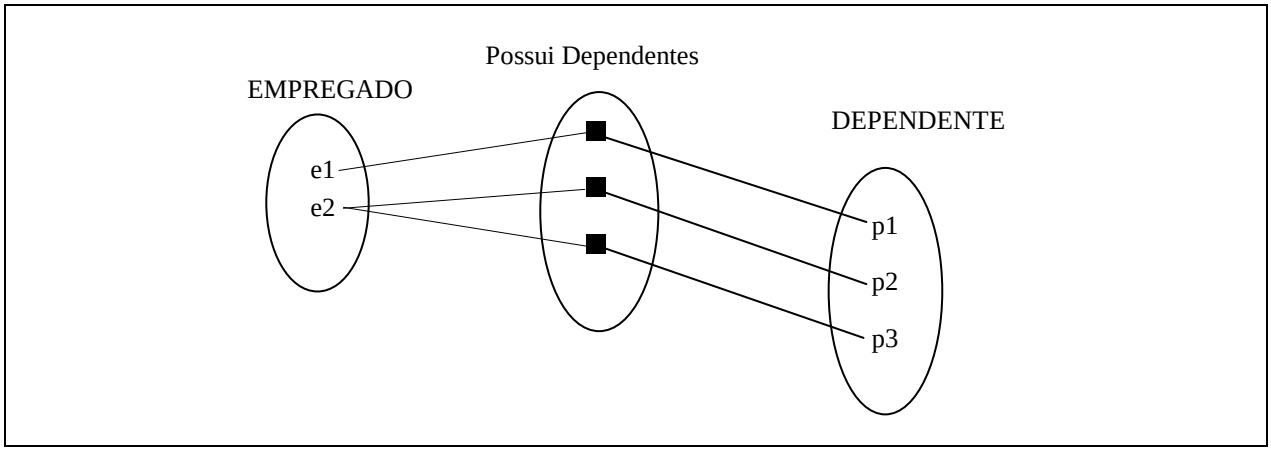
Outra restrição muito importante é a **participação**. A participação define a existência de uma entidade através do relacionamento, podendo ser **parcial** ou **total**. Veja o exemplo da Figura 18. A participação do empregado é **parcial**, pois nem todo empregado gerencia um departamento, porém a participação do departamento neste relacionamento é **total** pois todo departamento precisa ser gerenciado por um empregado. Desta forma, **todas** as entidades do tipo entidade DEPARTAMENTO precisam participar do relacionamento, mas **nem todas** as entidade do tipo entidade EMPREGADO precisam participar do relacionamento. Já no exemplo da figura 5, 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 o exemplo da Figura 18. Eu posso querer saber em que dia o empregado passou a gerenciar o departamento. É difícil estabelecer a qual tipo entidade pertence atributo, pois o mesmo 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. No 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 mesmo ficar no tipo relação. Veja o exemplo da . Caso queiramos armazenar quantas horas cada empregado trabalhou em cada projeto, então este deverá ser um atributo do relacionamento.

3.6.4. Tipos de Entidades Fracas

Alguns tipos entidade podem não ter um atributo chave por si só. Isto implica que não poderemos distinguir algumas entidades por que as combinações dos valores de seus atributos podem ser idênticas. Estes tipos entidade são chamados **tipos de entidades fracas**. Ao contrário, tipos de entidades regulares que efetivamente possuem um atributo chave são às vezes chamados **tipos de entidade fortes**. As entidades deste tipo precisam estar relacionadas com uma entidade pertencente ao **tipo de entidade proprietária**[3] ou **identificadora**. Este relacionamento é chamado de **relacionamento identificador**. Veja o exemplo da Figura 20. Uma entidade fraca sempre possui uma restrição de participação total(dependência de existência) com relação a seu relacionamento identificador, porque uma entidade fraca não pode ser identificada sem uma entidade proprietária.

Figura 20 - Relacionamento com uma Entidade Fraca (Dependente)



O tipo de entidade DEPENDENTE é uma entidade fraca pois não possui um método de identificar uma entidade única. O EMPREGADO não é uma entidade fraca pois possui um atributo para identificação (atributo chave). O número do RG de um empregado identifica um único empregado. Porém, um dependente de 5 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 RG. A chave parcial do DEPENDENTE é o seu nome, pois dois irmãos não podem ter o mesmo nome. Desta forma, a chave primária desta entidade fica sendo o RG do pai ou mãe mais o nome do dependente.

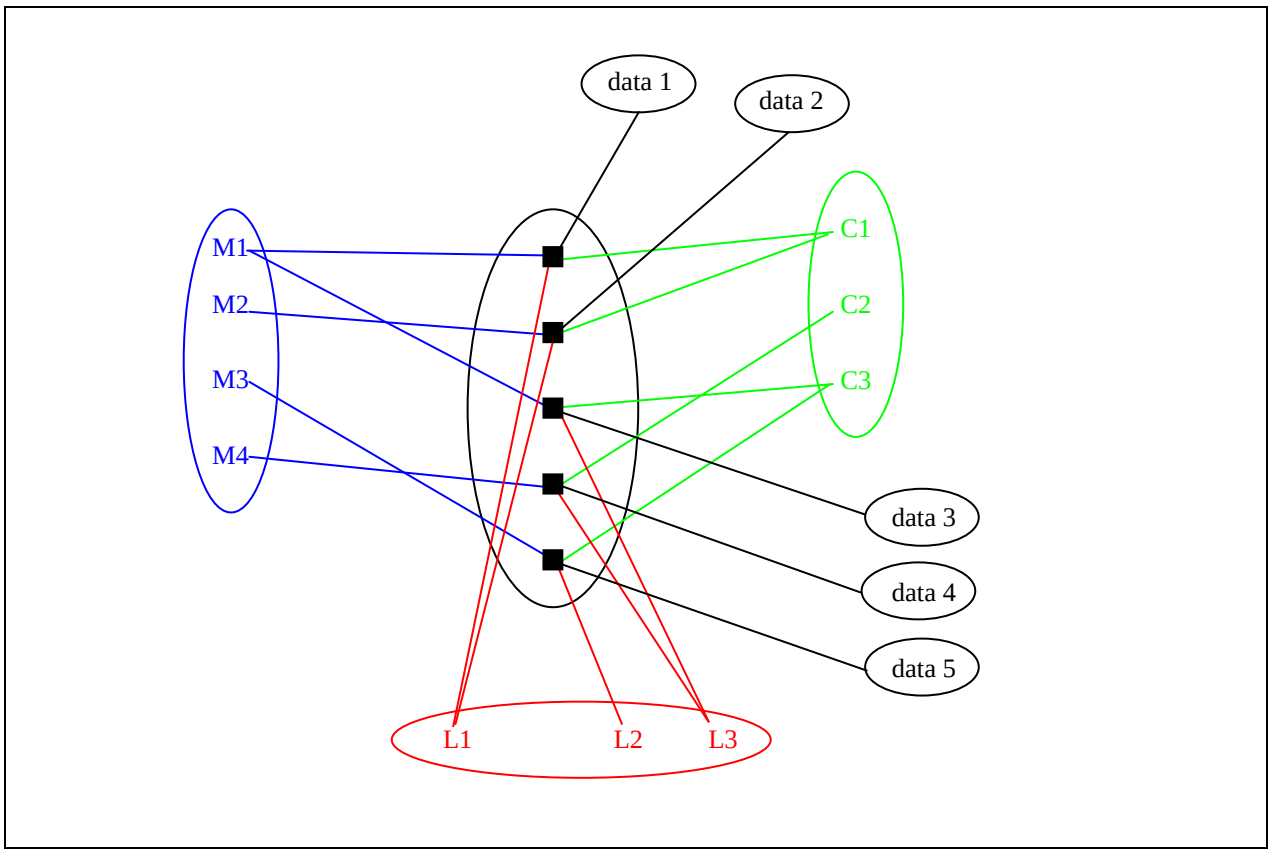
[3] O tipo de entidade proprietária às vezes é chamado de tipo de entidade pai ou tipo de entidade dominante.

Todos os exemplos vistos acima foram para relacionamentos binários, ou seja, entre dois tipos entidades diferentes ou recursivos. Porém, o modelo entidade relacionamento não se restringe apenas à 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 motorista pode efetuar uma viagem para uma localidade dirigindo um determinado caminhão em uma determinada data.*

Se efetuarmos três relacionamentos binários, não teremos estas informações de forma completa como se criássemos um relacionamento ternário. Veja o resultado como fica no exemplo da Figura 21.

Figura 21 - Relacionamento Ternário



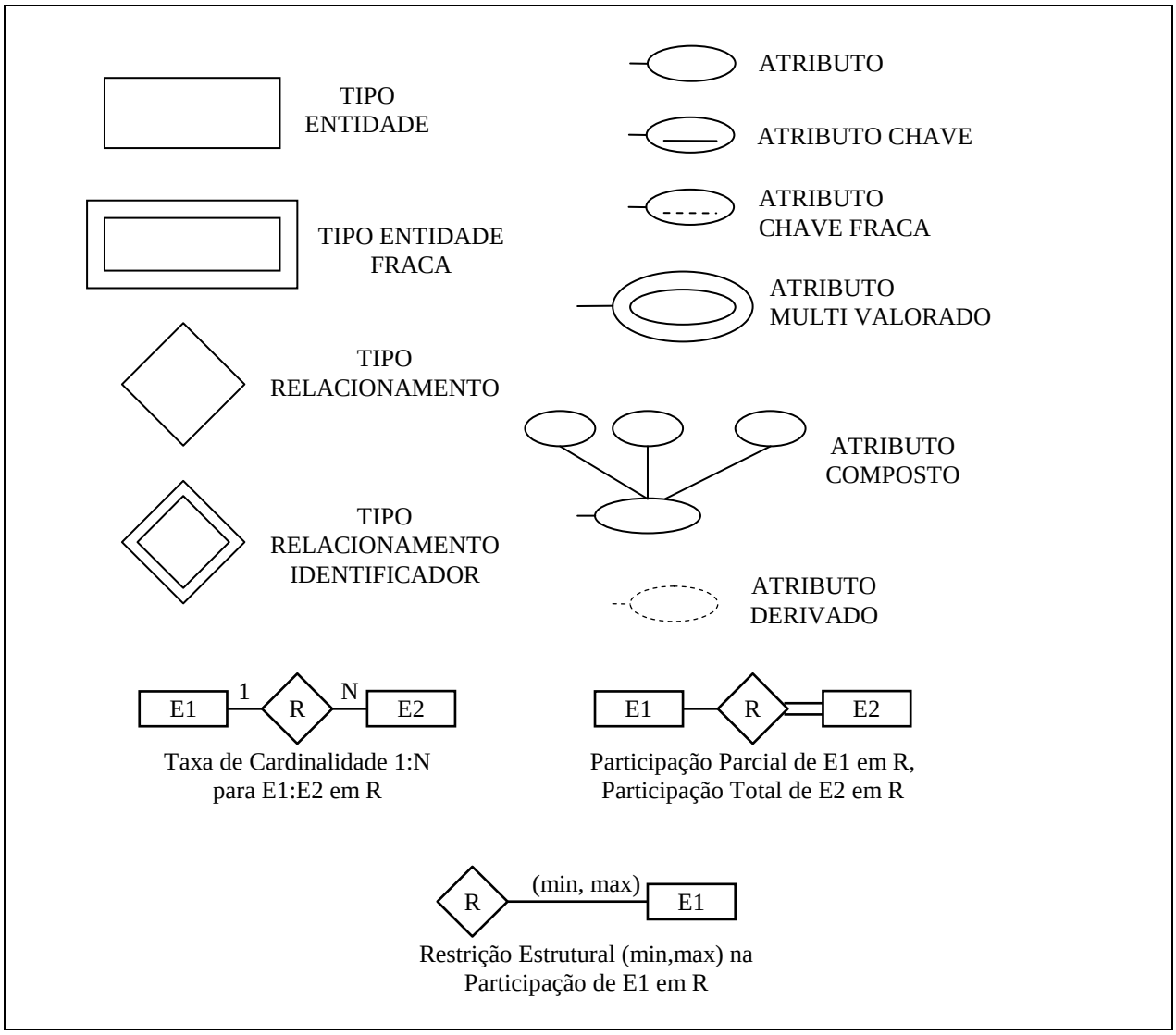
3.7. Diagrama Entidade Relacionamento

O Diagrama Entidade Relacionamento[4] é composto por um conjunto de objetos gráficos que visa representar todos os objetos do modelo Entidade Relacionamento tais como entidades, atributos, atributos chaves, relacionamentos, restrições estruturais, etc.

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 22.

Figura 22 – Resumo das notações para Diagramas ER



[4] Será utilizado uma forma próxima do diagrama ER proposto por Peter Pin-Shan Chen em 1976. Sua home page http://www.csc.lsu.edu/~chen/chen.html

3.7.1. Dicas para Construção de Diagramas ER

3.7.1.1. Substantivo

A presença de um **substantivo** usualmente indica uma **entidade**.

Qualquer substantivo que desempenha um certo papel no contexto do nosso negócio é definitivamente uma entidade. Assim, a decisão sobre o que se torna uma entidade e o que se torna um atributo pode depender dos processos de negócio.

3.7.1.2. Verbo

A presença de um **verbo** é uma forte indicação de um **relacionamento.**

Há duas exceções notáveis para verbos:

- O verbo "**tem/ter"** indica que uma entidade tem um atributo, ou que uma entidade agrega outras entidades.

- A expressão "**é um**" indica uma classificação.

3.7.1.3. Adjetivo

Um **adjetivo**, que é uma qualidade, é uma forte indicação de um **atributo.**

Em muitos casos a distinção entre um atributo e uma entidade pode ser feita utilizando uma regra simples: um atributo pode pertencer a uma entidade, mas uma entidade não pode pertencer a um atributo.

Um adjetivo que pertence a uma única entidade é um candidato a se tornar um atributo. Ao contrário, um adjetivo que tem outras conexões é um candidato a ser modelado como entidade. Por exemplo: um cliente tem nome, mas um nome não tem cliente.

3.7.1.4. Advérbio

Um **advérbio** temporal, qualificando o verbo, é uma indicação de um atributo do **relacionamento.**

3.7.1.5. Gramática

Gramática segundo Pasquale Cipro Neto ver referência Neto, 1998.

Substantivo é a palavra que nomeia os seres. O conceito de seres deve incluir os nomes de pessoas, de lugares, de instituições, de grupos, de indivíduos e de entes de natureza espiritual mitológica.

Verbo é a palavra que se flexiona em número (singular/plural), pessoa (primeira, segunda, terceira), modo (indicativo, subjuntivo, imperativo), tempo (presente, pretérito, futuro) e voz (ativa, passiva, reflexiva). Pode indicar ação (fazer, copiar), estado (ser, ficar), fenômeno natural (chover, anoitecer), ocorrência (acontecer, suceder), desejo (aspirar, almejar) e outros processos.

Adjetivo é a palavra que caracteriza o substantivo, atribuindo-lhe qualidades (ou defeitos) e modos de ser, ou indicando-lhe o aspecto ou o estado.

Advérbio é a palavra que caracteriza o processo verbal, exprimindo circunstâncias em que esse processo se desenvolve.

3.8. Modelo Entidade Relacionamento Extendido

Os conceitos do modelo Entidade Relacionamento discutidos anteriormente são suficientes para representar logicamente a maioria das aplicações de banco de dados. Porém, com o surgimento de novas aplicações, surgiu também à 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 **ER** mais os conceitos de **subclasse**, **superclasse**, **generalização** e **especialização** e o conceito de **herança de atributos**.

3.8.1. Subclasses, Superclasses e Herança

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 entidades do mesmo tipo. Em muitos casos, um tipo entidade possui diversos subgrupos adicionais de entidades que são significativas e precisam ser representadas explicitamente devido ao seu significado à aplicação de banco de dados. Leve em consideração o seguinte exemplo:

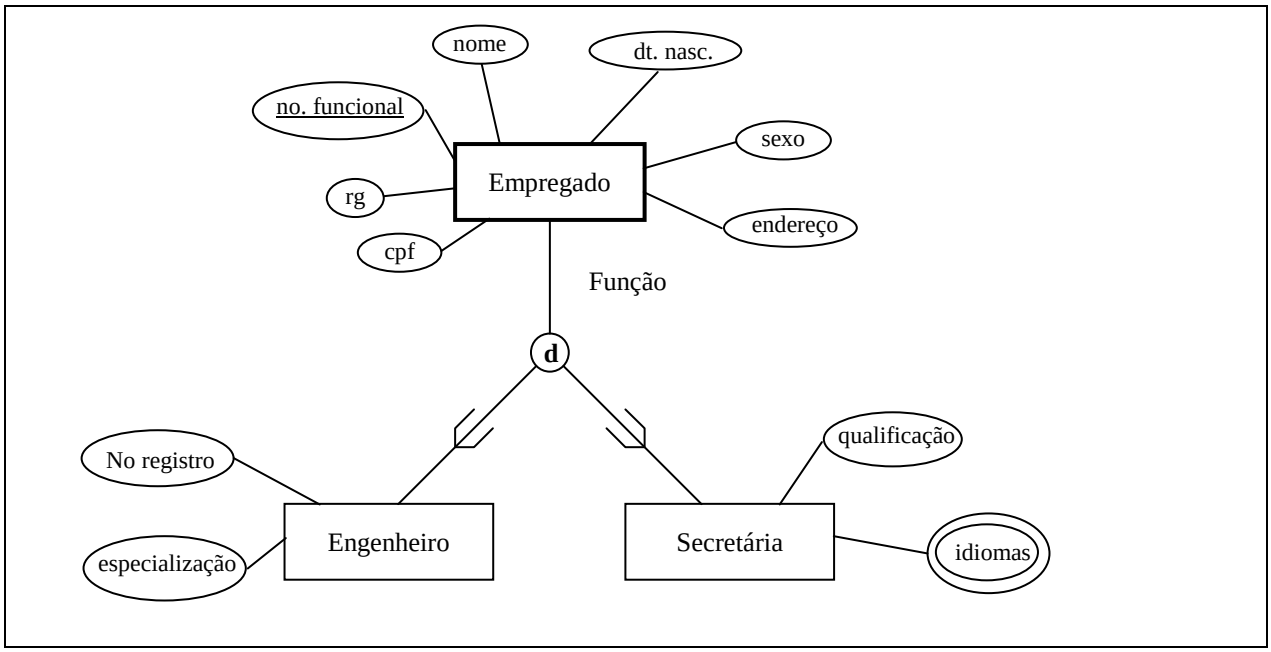
> *Para um banco de dados de uma empresa temos o tipo entidade empregado, o qual possui as seguintes características: nome, rg, cpf, número funcional, endereço completo (rua, número, complemento, cep, bairro, cidade), sexo, data de nascimento e telefone (ddd e número); caso o(a) funcionário(a) seja um(a) engenheiro(a), então deseja-se armazenar as seguintes informações: número do CREA e especialidade (Civil, Mecânico, Elétro/Eletrônico); caso o(a) funcionário(a) seja um(a) secretário(a), então deseja-se armazenar as seguinte 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 representadas diretamente no tipo de entidade *empregado* estaremos representando informações de um conjunto limitados de entidades *empregado* para os todos os funcionários da empresa. Neste caso, podemos criar duas subclasses do tipo de 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 uma subclasse. Antes de ser componente de uma subclasse, uma entidade deve ser componente de uma superclasse. Isto leva ao conceito de **herança de tipo (atributo)**; ou seja, a subclasse herda todos os atributos da superclasse. Isto porque a entidade de subclasse representa as mesmas características de uma mesma entidade da superclasse. Uma subclasse pode herdar atributos de superclasses diferentes.

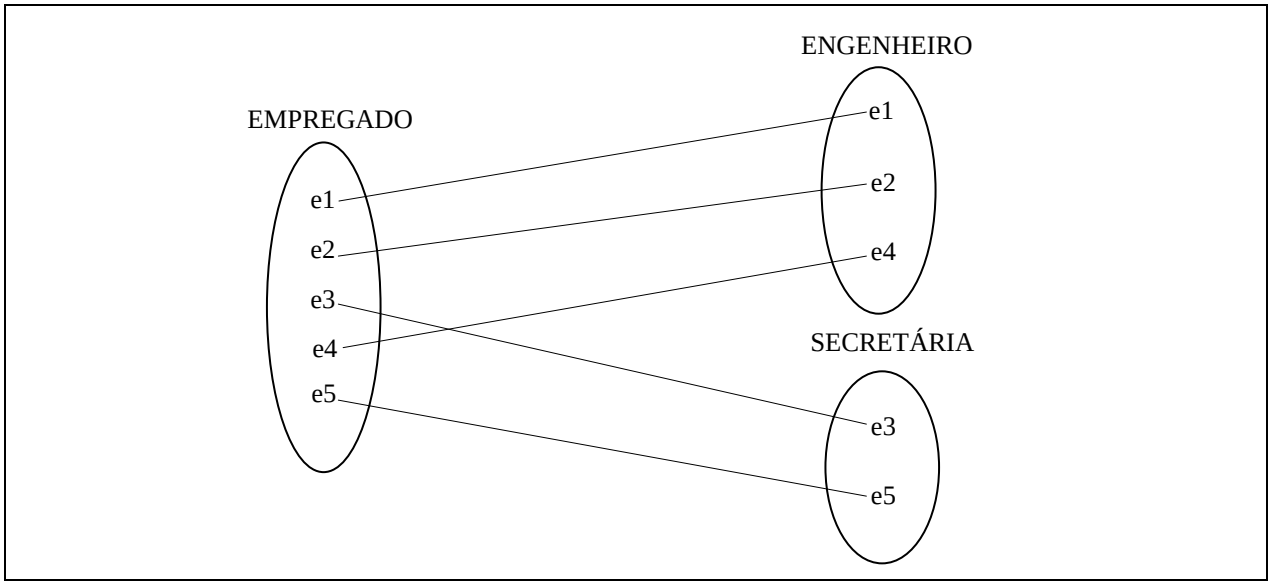
A Figura 23 mostra a representação diagramática do exemplo acima.

Figura 23 - Representação de Superclasse e Subclasses



A Figura 24 mostra algumas instâncias de entidades que pertencem às subclasses da especialização {ENGENHEIRO, SECRETÁRIA}. Repare novamente que uma entidade que pertence a uma subclasse representa a mesma entidade do mundo real que a entidade a ela ligada na superclasse EMPREGADO, embora a mesma entidade seja apresentada duas vezes; por exemplo e1 é mostrada tanto em EMPREGADO como em ENGENHEIRO.

Figura 24 – Algumas instâncias da especialização de EMPREGADO

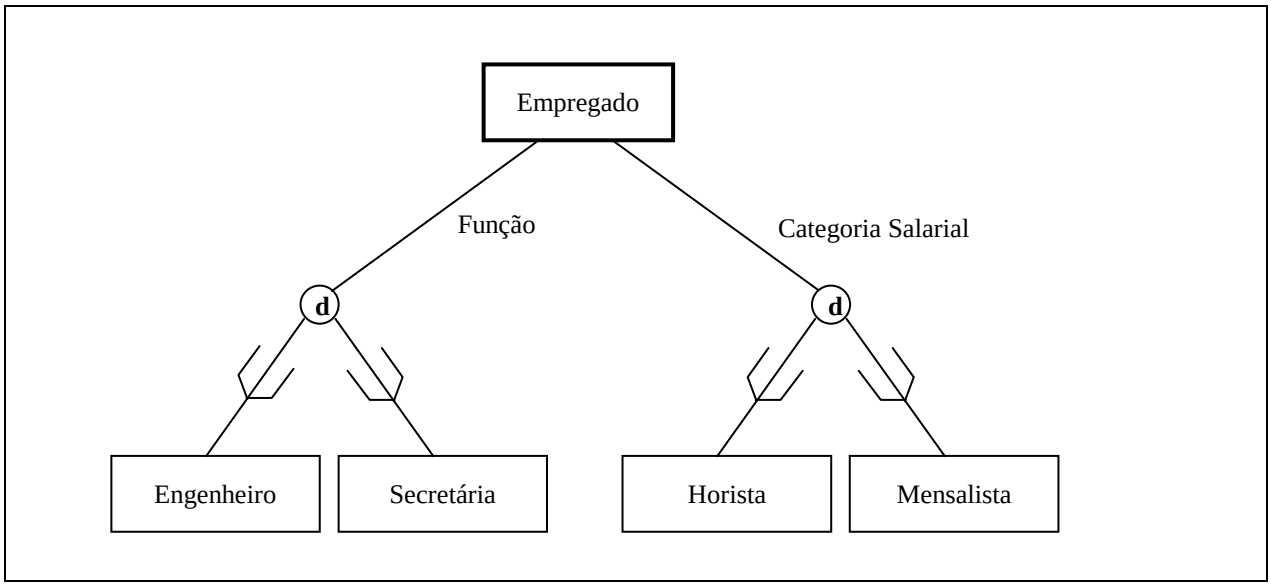


3.8.2. Especialização

Especialização é o processo de definição de um conjunto de classes de um tipo entidade; este tipo entidade é chamado de superclasse da especialização. O conjunto de subclasses é formado baseado em alguma característica que distingue as entidades entre si.

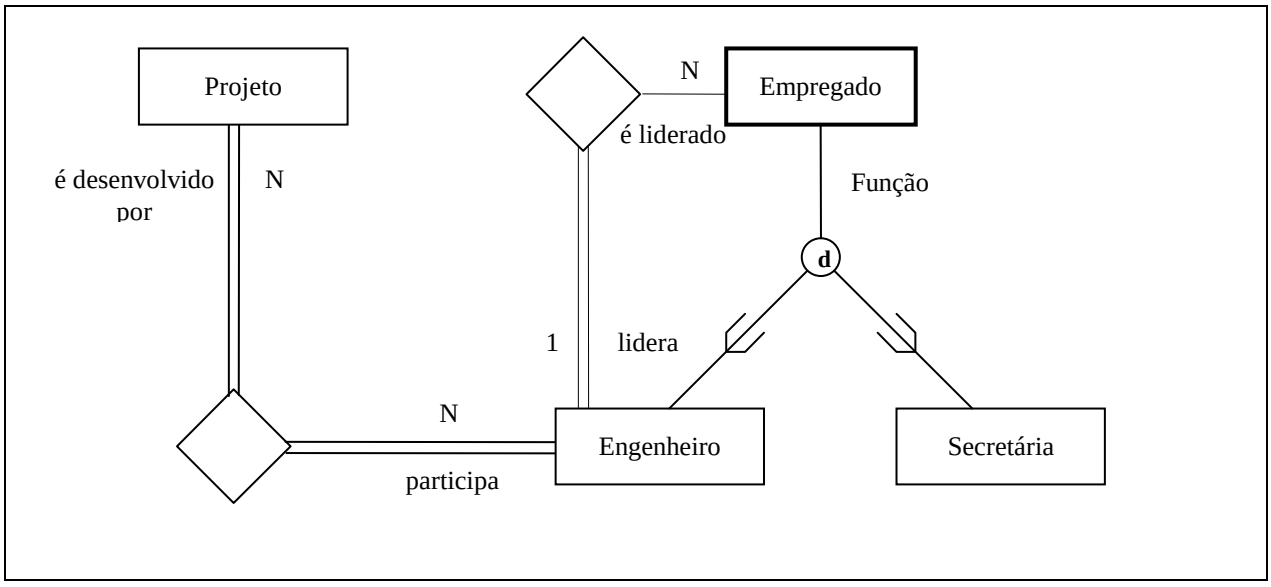
No exemplo da Figura 23, temos uma especialização, a qual podemos chamar de *função*. Veja agora no exemplo da Figura 25, temos a entidade empregado e duas especializações.

Figura 25 - Duas Especializações para Empregado: Função e Categoria Salarial



Como visto anteriormente, uma subclasse pode ter relacionamentos específicos com outras entidades ou com a própria entidade que é a sua superclasse. Veja o exemplo da Figura 26.

Figura 26 - Relacionamentos entre Subclasses e Entidades



O processo de *especialização* nos permite:

- definir um conjunto de subclasses de um tipo entidade;
- associar atributos específicos adicionais para cada subclasse;
- estabelecer tipos relacionamentos específicos entre subclasses e outros tipos entidades.

3.8.3. 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 uma superclasse.

Figura 27 - Tipos Entidades Engenheiro e Secretária

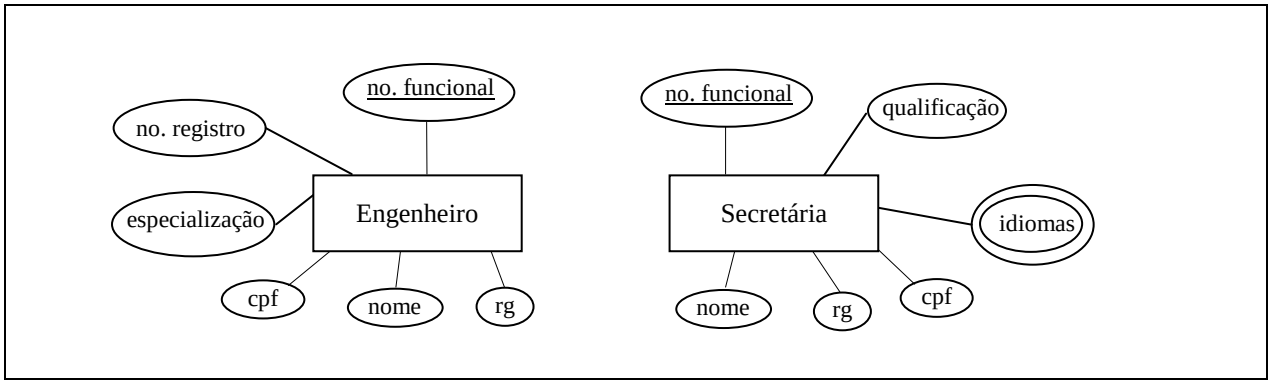
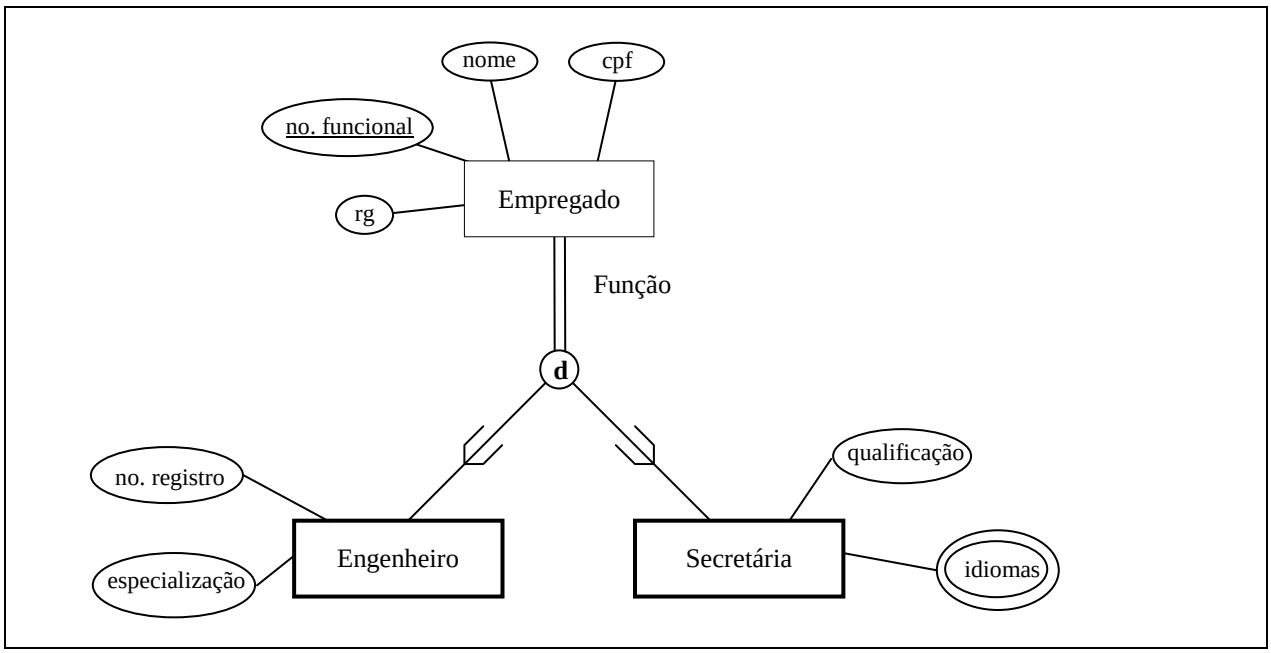


Figura 28 - Generalização Empregado para os Tipos Entidades Engenheiro e Secretária

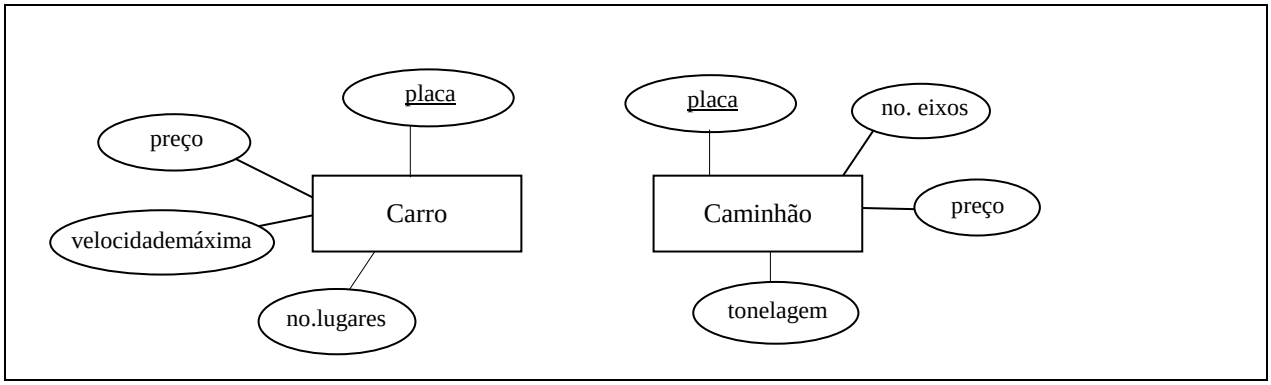


É importante destacar que existe diferença semântica entre a *especialização* e a *generalização*. Na *especialização*, podemos notar que a ligação entre a superclasse e as subclasses é feita através de um traço simples, indicando **participação parcial** por parte da superclasse. Analisando o exemplo da Figura 23, é observado que um empregado **não** é obrigado a ser um *engenheiro* ou uma *secretária*. Na generalização, podemos notar que a ligação entre a superclasse e as subclasses é feita através de um traço duplo, indicando **participação total** por parte da superclasse. Analisando o exemplo da Figura 28, é observado que um empregado **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 da mesma. Analisando o exemplo da Figura 25. Temos duas especializações para a superclasse *Empregado*, as quais são restringidas através de uma disjunção. Neste caso, um empregado pode ser um *engenheiro* ou uma *secretária* e o mesmo pode ser *horista* ou *mensalista*.

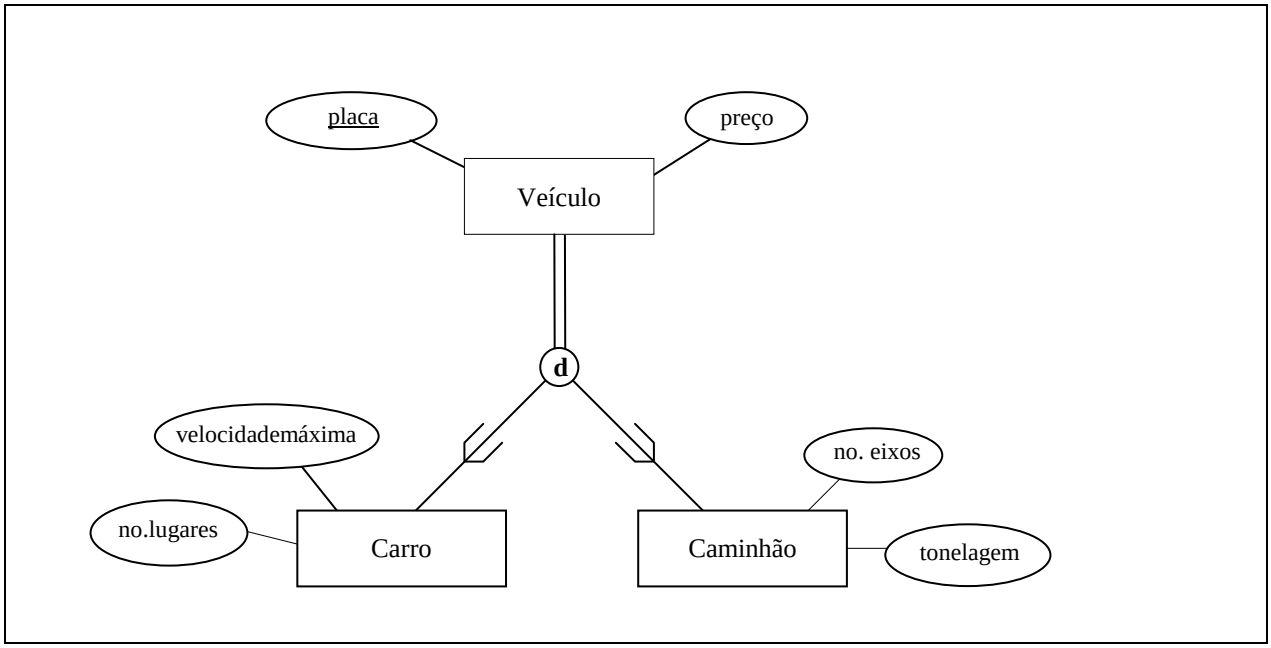
Abaixo outro exemplo de uma generalização entre *Carro* e *Caminhão*. A Figura 29 mostra as os tipos entidade e seus atributos.

Figura 29 - Tipos Entidades Carro e Caminhão



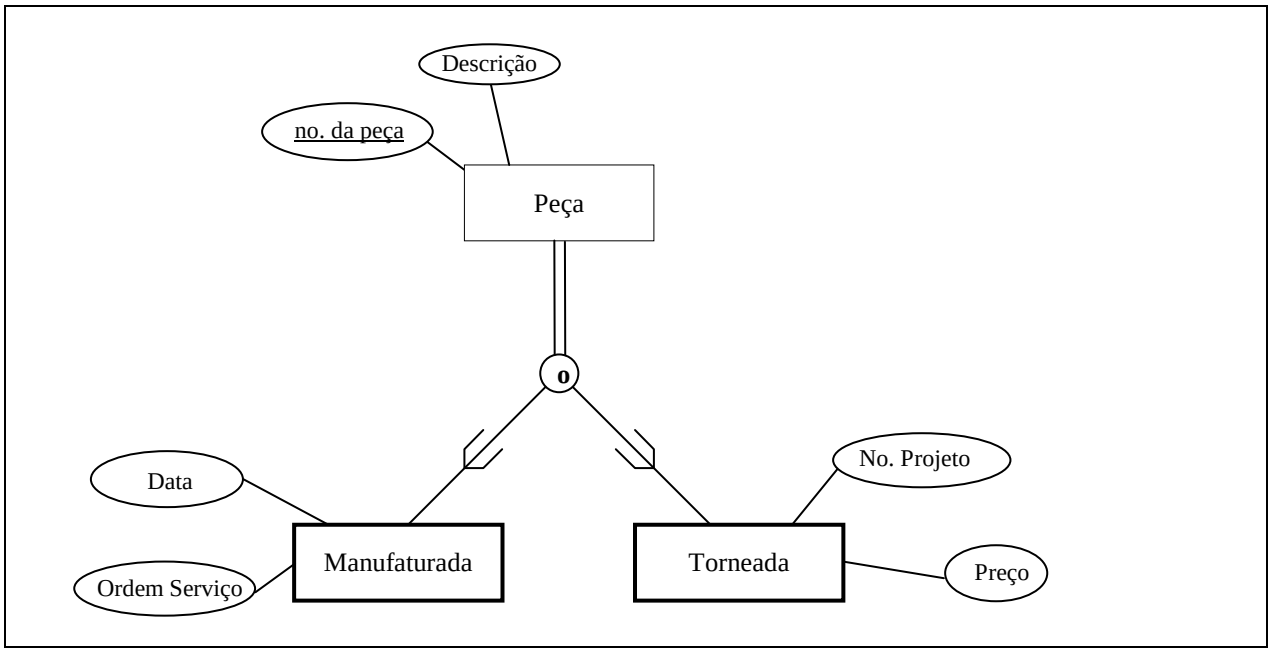
O tipo entidade *Veículo (*Figura 30*)* surge como a generalização entre *Carro* e *Caminhão*.

Figura 30 - Generalização Veículo para os Tipos Entidades Carro e Caminhão



Além da *disjunção* podemos ter um "**overlap**", representado pela letra **o**. No caso do "overlap", uma entidade de uma superclasse pode ser membro de mais que uma subclasse em uma especialização ou generalização. Analise a generalização no exemplo da Figura 31. Suponha que uma peça fabricada em uma tornearia pode ser *manufaturada* ou *torneada* ou ainda, pode ter sido *manufaturada* e *torneada*.

Figura 31 - Uma Generalização com "Overlap"



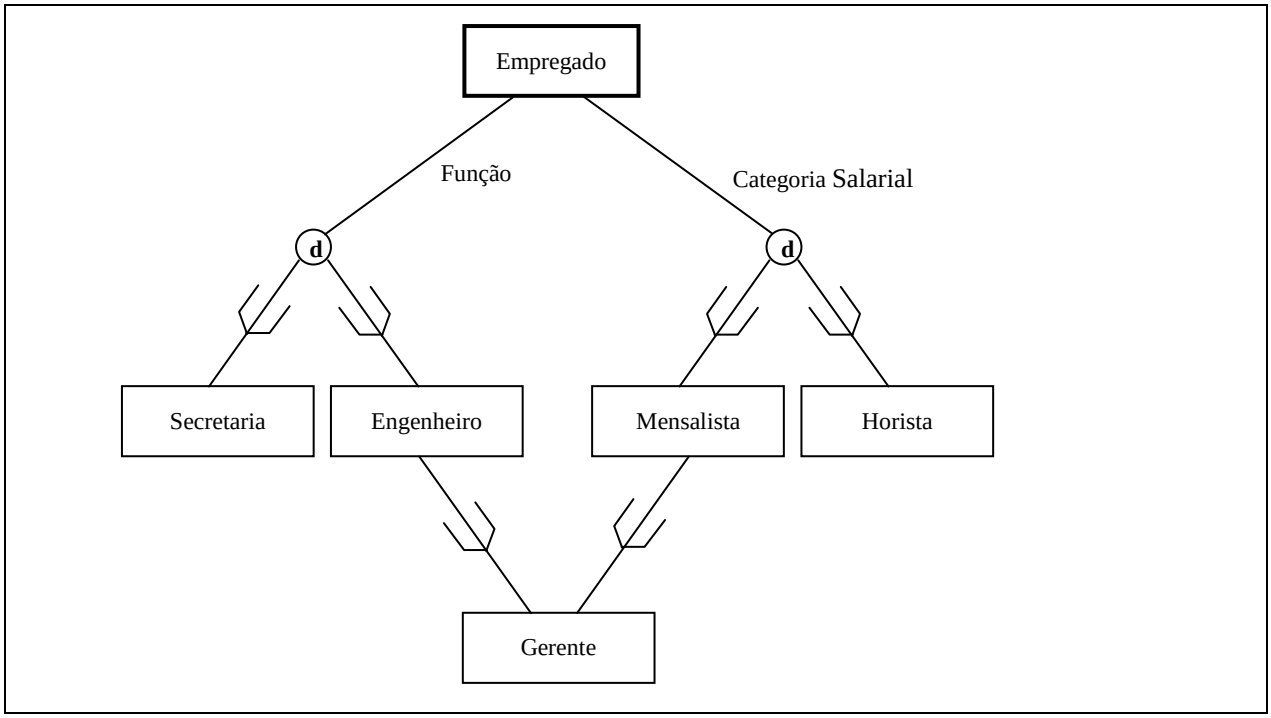
3.8.4. "Lattice" ou Múltipla Herança

Uma subclasse pode ser definida através de um "lattice", ou múltipla herança, ou seja, ela pode ter diversas superclasses, herdando características de todas. Leve em consideração o seguinte exemplo:

> *Uma construtora possui diversos funcionários, os quais podem ser engenheiros ou secretárias. Um funcionário pode também ser assalariado ou horista. Todo gerente de departamento da construtora deve ser um engenheiro e assalariado.*

A Figura 32 exibe o modelo lógico da expressão acima.

Figura 32 - Um "Lattice" com a Subclasse Gerente Compartilhada



Neste caso então, um gerente será um funcionário que além de possuir as características próprias de *Gerente*, herdará as características de *Engenheiro* e de *Mensalista*.

3.8.5. Notações Diagramáticas Alternativas para Diagramas ER

A Figura 33 mostra uma série de diferentes notações diagramáticas para representar conceitos de modelos ER e ERE. Infelizmente, não existe notação padronizada: diferentes projetistas de bancos de dados preferem diferentes notações. De modo semelhante, várias ferramentas CASE(Computer-Aided Software Engineering) e metodologias de OOA (Object-Oriented Analysis) utilizam várias notações. Algumas notações são associadas a modelos que têm conceitos adicionais e restrições além daquelas dos modelos ER e ERE, enquanto outros modelos têm menor quantidade de conceitos e restrições. A notação que utilizamos é bastante próxima da notação original para diagramas ER, que ainda é amplamente utilizada.

A Figura 33(a) mostra diferentes notações para exibir tipos/classes de entidades, atributos e relacionamentos. Anteriormente utilizamos os símbolos marcados (i) da Figura 33(a) – ou seja, triângulo, oval e losango. Observe que o símbolo (ii) para tipos/classes de entidades, o símbolo (ii) para atributos e o símbolo (ii) para relacionamentos são semelhantes, mas são utilizados por diferentes metodologias para representar três diferentes conceitos. O símbolo para linha reta (iii) para representar relacionamentos é utilizado por várias ferramentas e metodologias.

A Figura 33(b) mostra algumas notações para se anexarem atributos a tipos de entidades. Utilizamos a notação (i). A notação (ii) utiliza a terceira notação (iii) para atributos da Figura 33(a). As duas últimas notações na Figura 33(b) – (iii) e (iv) – são populares em metodologias de OOA e em algumas ferramentas CASE. Em particular, a última notação exibe ambos os atributos e os métodos de uma classe, separados por uma linha horizontal.

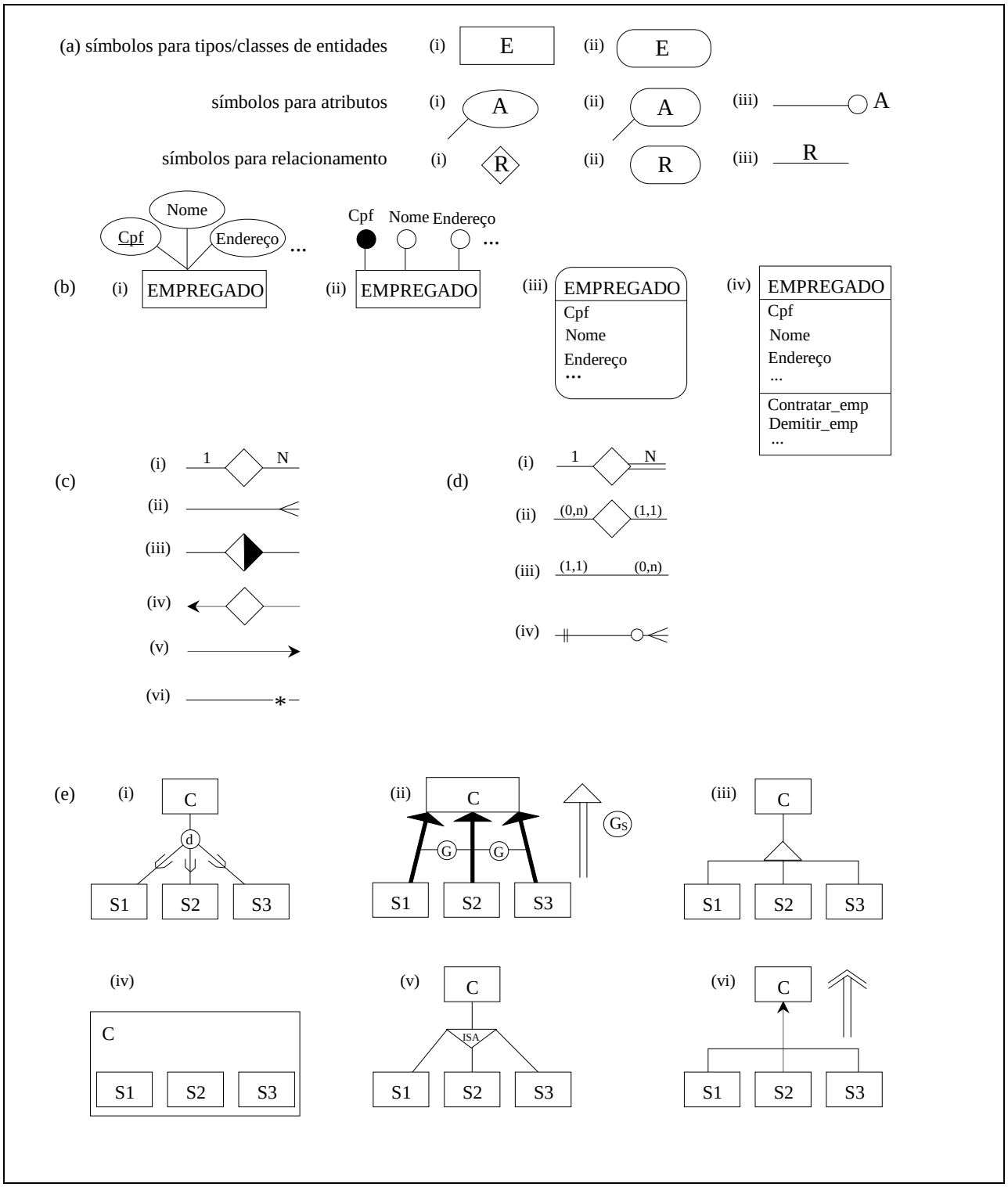
A Figura 33(c) mostra várias notações para representar a taxa de cardinalidade de relacionamento binários. Utilizamos a notação (i) anteriormente. A notação (ii) – conhecida como notação de pés de galinha (*chicken feet*) – é bastante popular. A notação (iv) utiliza a seta como uma referência funcional (do lado N para o lado 1) e se assemelha à nossa notação para chaves estrangeiras no modelo relacional; a notação (v) – utilizada em *diagramas de Bachman* – utiliza a seta na direção inversa (do lado 1 para o lado N). Para um relacionamento do tipo 1:1, (ii) utiliza linha reta sem quaisquer pés de galinha; (iii) torna brancas as duas metades do losango e (iv) coloca pontas de setas em ambos os lados. Para um relacionamento do tipo N:M, (ii) utiliza pés de galinha em ambas as extremidades da linha; (iii) torna negras ambas as metades do losango; e (iv) não exibe nenhuma ponta de seta.

A Figura 33(d) mostra diversas variações para exibir restrições (min, max), que são utilizadas para exibir a taxa de cardinalidade e a participação total/parcial. A notação (ii) é a notação alternativa que utilizamos anteriormente. Lembre-se de que nossa notação especifica a restrição de que cada entidade deve participar em pelo menos (min) e no máximo (max) instâncias de relacionamentos. Portanto, para um relacionamento do tipo 1:1, ambos os valores max são iguais a 1; e para relacionamentos do tipo N:M, ambos os valores max são iguais a n. Um valor mim maior que 0 (zero) especifica participação total (dependência de existência). Em metodologias que utilizam uma linha reta para exibir relacionamentos, é comum *reverter o posicionamento* das restrições (min, max), conforme mostramos em (iii). Uma outra técnica popular – que segue o mesmo posicionamento de (iii) – é exibir *min* como o (a letra “o” ou círculo, que representa zero) ou como | (barra vertical, que representa 1) e exibir max como | (barra vertical, que representa 1) ou como pés de galinha (que representam n), como mostramos em (iv).

Figura 33(e) mostra algumas notações para exibir especializações/generalização. Utilizamos a notação (i) anteriormente, onde a letra **d** dos círculos especifica que as subclasses (S1, S2 e S3) são disjuntas e a letra **o** especifica subclasses que se sobrepõem. A notação (ii) utiliza G (para generalização) para especificar disjunto e Gs para especificar sobreposto; algumas notações utilizam uma seta densa

(grossa), enquanto outros utilizam a seta vazia(mostrado ao lado). A notação (iii) utiliza um triângulo apontando em direção a uma superclasse e a notação (v) utiliza um triângulo apontando em direção às subclasses mais a palavra ISA (is a, é um); também é possível utilizar ambas as notações na mesma metodologia, como (iii) indicando generalização e (v) indicando especialização. A notação (iv) coloca os quadrados representando subclasses dentro do quadrado representando a superclasse. Dentre as notações baseadas em (iv), algumas utilizam uma seta com uma linha única enquanto outras utilizam uma seta com linhas duplas (exibida ao lado).

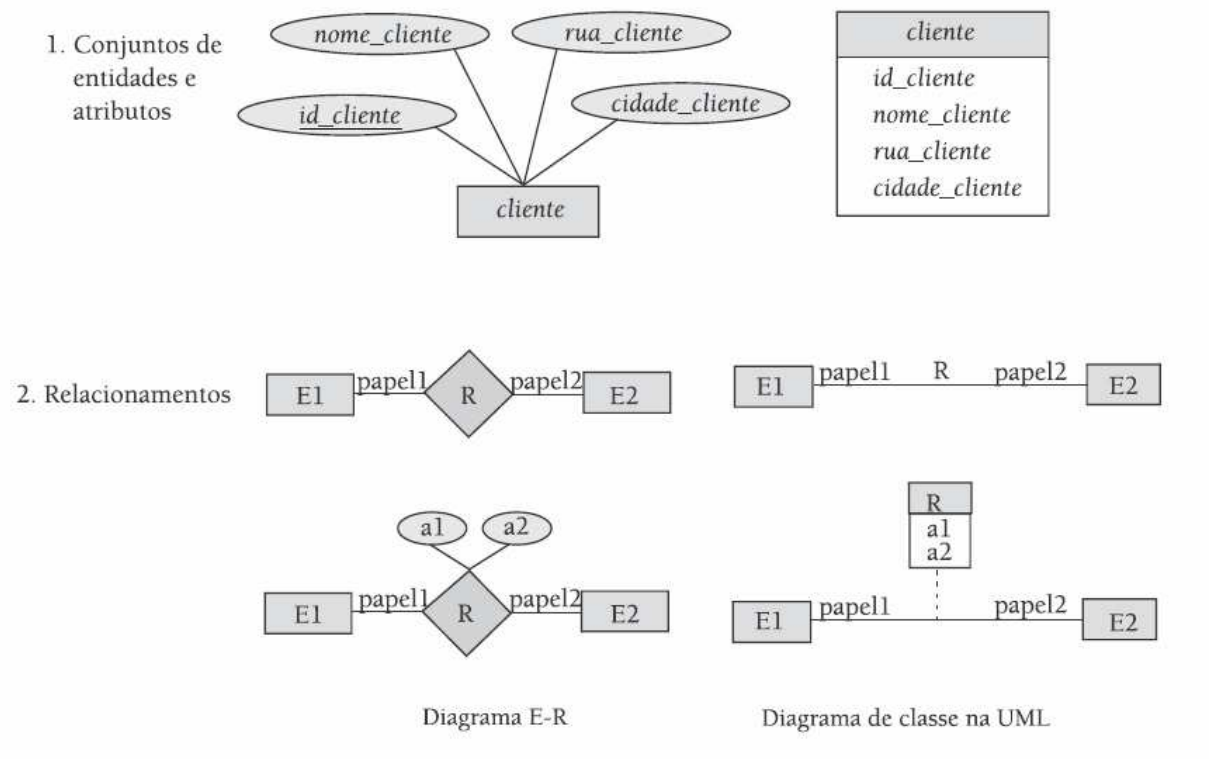
Figura 33 - Notações Alternativas



3.8.6. Modelagem Conceitual de Objetos Utilizando Diagramas de Classes da UML

As metodologias de modelagem de objetos, como a UML(Unified Modeling Language) são padrão de mercado atualmente. Embora a UML seja apresentada para o projeto do software, uma importante parte do projeto do software envolve projetar o banco de dados que serão acessados pelos módulos do software. Dessa forma, uma importante parte dessa metodologia – por exemplo, diagrama de classe[5] – é semelhante, sob vários aspectos, aos diagramas ERE. Infelizmente, a terminologia é geralmente diferente.

Figura 34 - Símbolos usados na notação do diagrama de classe UML



Os conjuntos de entidades são mostrados como retângulos e, diferente do diagrama ER, os atributos são mostrados dentro do retângulo e não como elipses separadas.

Os conjuntos de relacionamento binários são representados na UML simplesmente desenhando uma linha conectando os conjuntos de entidades. O nome do conjunto de relacionamento é escrito adjacente à linha.

A função desempenhada por um conjunto de entidades em um conjunto de relacionamento também pode ser especificada escrevendo o nome da função na linha, adjacente ao conjunto de entidades.

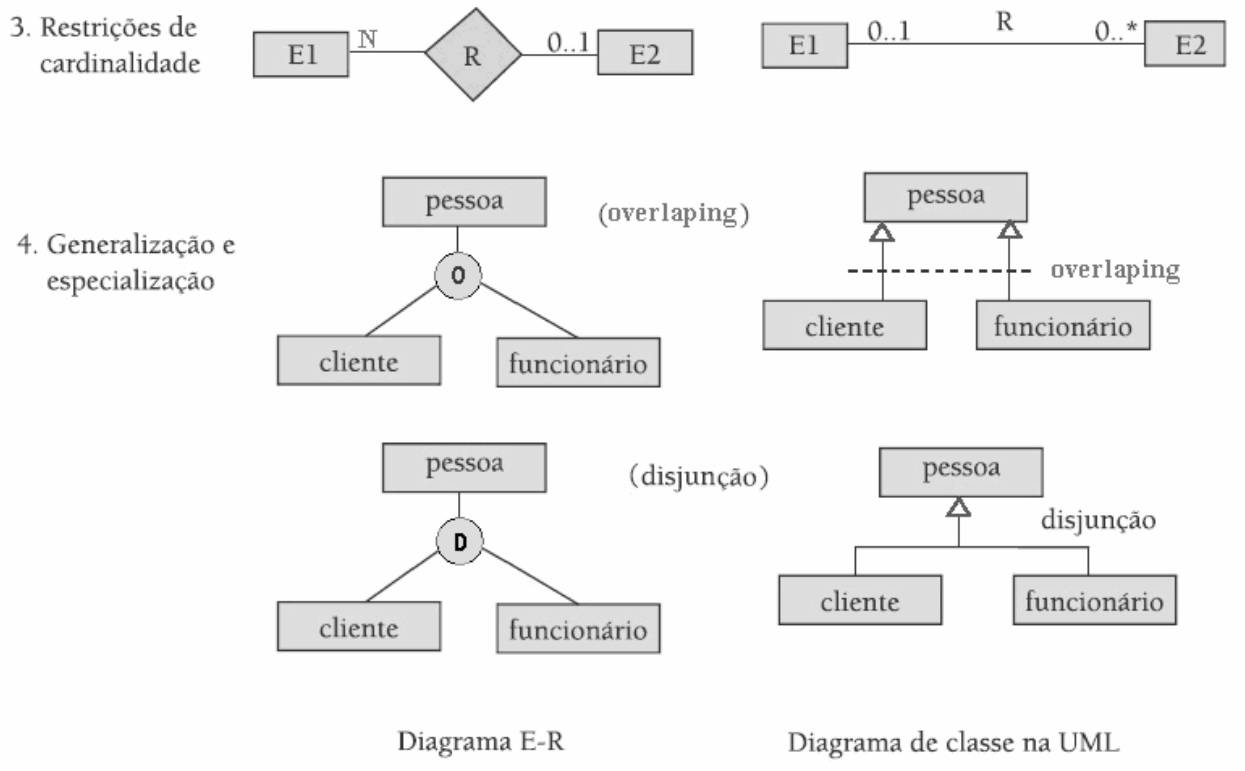
Alternativamente, o nome do conjunto de relacionamento pode ser escrito em um retângulo, juntamente com atributos do conjunto de relacionamento, e o retângulo é conectado usando uma linha pontilhada até a linha representando o conjunto de relacionamento.

Os relacionamentos não binários são desenhados usando losangos, exatamente como nos diagramas ER.

---

[5] Uma classe é similar a um tipo de entidade exceto pelo fato de ela pode possuir operações.

Figura 35 - Símbolos usados na notação do diagrama de classe UML



As restrições de cardinalidade são especificadas na forma l..h, onde l indica o mínimo e h, o número máximo de relacionamentos em que um conjunto de entidades pode participar.

Atenção: O posicionamento das restrições é exatamente o inverso do posicionamento das restrições nos diagramas ER.

A restrição N (muitos) no lado E2 e a restrição 1 no lado E1 significam que cada entidade E1 pode participar em muitos relacionamentos; ou seja, o relacionamento é muitos-para-um de E2 para E1.

Valores únicos como 1 ou N podem ser escritos nas bordas; o valor único 1 em uma borda é tratado como equivalente a 1..1, enquanto N é equivalente a 0..*.

4. O Modelo Relacional

O **modelo relacional** foi criado por Edgar Frank "Ted" Codd[6] em 1970 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.

Codd definiu 12 regras para que um banco de dados, para que seja considerado "totalmente relacional". As doze regras estão baseadas na regra zero, que determina o seguinte: "Qualquer sistema considerado, ou que deseja ser, um sistema gerenciador de banco de dados relacionam deve ser capaz de gerenciar, por completo, bases de dados através de sua capacidade relacional". Essa regra determina que um SGBDR não permite exceções quanto ao modelo relacional de gerenciamento de bases de dados. Algumas vezes as regras se tornam uma barreira e nem todos os SGBDs relacionais fornecem suporte a elas. As 12 regras são as seguintes:

- **Regra 1** - Representação de valores em tabelas: Todas informações do banco de dados relacional são representadas de forma explícita no nível lógico e exatamente em apenas uma forma - por valores em tabelas. As informações devem ser apresentadas como relações (tabelas formadas por linhas e colunas) e o vínculo de dados entre as tabelas deve ser estabelecido por meio de valores de campos comuns. Isso se aplica tanto aos dados quanto aos metadados.
- **Regra 2** – Acesso Garantido: Cada um e qualquer valor atômico em um banco de dados relacional possuem garantia de ser logicamente acessado pela combinação do nome da tabela, do valor da chave primária e do nome da coluna.
- **Regra 3** – Tratamento sistemático de nulos: Valores nulos devem ser suportados de forma sistemática e independente do tipo de dado para representar informações inexistentes e / ou inaplicáveis, ou seja valores nulos devem ter um tratamento diferente de "valores em branco".
- **Regra 4** – Dicionário de dados ativo baseado no modelo relacional: A descrição do banco de dados é representada no nível lógico da mesma forma que os dados ordinários, permitindo que usuários autorizados utilizem a mesma linguagem relacional aplicada aos dados regulares.
- **Regra 5** – Linguagem Detalhada: Um sistema relacional pode suportar várias linguagens e várias formas de recuperação de informações. Entretanto, deve haver pelo menos uma linguagem, com uma sintaxe bem definida e expressa por conjuntos de caracteres, que suporte de forma compreensiva todos os seguintes itens: definição de dados, definição de views, manipulação de dados (interativa e embutida em programas), restrições de integridade, autorizações e limites de transações.
- **Regra 6** – Atualização de Visões: Todas as visões ("views") que são teoricamente atualizáveis devem também ser atualizáveis pelo sistema.
- **Regra 7** – Atualização de alto nível: A capacidade de manipular um conjunto de dados (relação) por um simples comando deve-se estender às operações de inclusão, alteração ou exclusão de dados.
- **Regra 8** – Independência Física: Programas de aplicação permanecem logicamente inalterados quando ocorrem mudanças no método de acesso ou na forma de armazenamento físico.
- **Regra 9** – Independência Lógica: Mudanças nas relações e nas views provocam pouco ou nenhum impacto nas aplicações.
- **Regra 10** – Independência de Integridade: As aplicações não são afetadas quando ocorrem mudanças nas restrições de integridade.
- **Regra 11** – Independência de Distribuição: As aplicações não são logicamente afetadas quando ocorrem mudanças geográficas dos dados. Devem permanecer inalterados quando são distribuídos em meios ou máquinas diferentes.

---

[6] http://en.wikipedia.org/wiki/Edgar_F._Codd

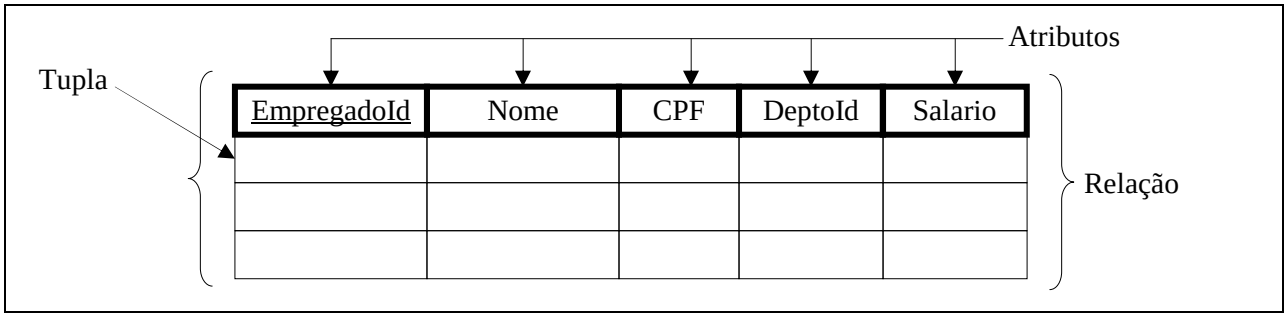
• **Regra 12** – Não Subversão: Se um sistema possui uma linguagem de baixo nível, essa linguagem não pode ser usada para subverter as regras de integridades e restrições definidas no nível mais alto.

Além dessas doze regras básicas, o modelo relacional também define nove regras estruturais que tratam da definição de chaves primárias, chaves estrangeiras, views, tabelas etc.; dezoito regras de manipulação que definem as operações de "join", "union", "division" etc.; e três regras de integridade.

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 desta tabela são utilizados para facilitar a interpretação dos valores armazenados em cada linha da tabela. 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**.

Figura 36 – Relação Empregado



4.1. Domínios, Tuplas, Atributos e Relações

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. Por exemplo:

Coluna	**Tipo**	**Tamanho**	**Faixa**
RG	Numérico	10,0	03000000-25999999
Nome	Caracter	30	a-z, A-Z
Salario	Numérico	5,2	00100,00-12999,99

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

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.

4.2. 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 diversas tuplas da mesma, garantindo que $t1[SC] \neq t2[SC]$.

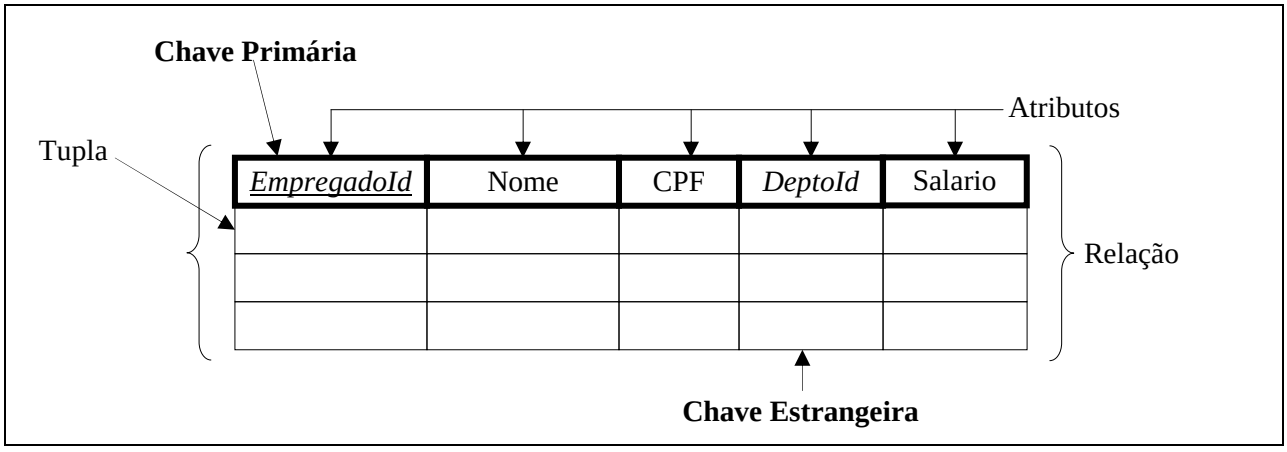
SC é chamada de **superchave** de um esquema de relação. Toda relação possui ao menos uma superchave - o conjunto de todos os seus atributos. Uma **chave C** de um esquema de relação **R** é uma superchave de **R** com a propriedade adicional que removendo qualquer atributo **A** de **K**, resta ainda 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: *(RegistroAcademico, 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 qualquer um dos atributos que retirarmos, 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 abaixo:

Relação DEPENDENTE				
EmpregadoId	Nome	DataNascimento	Relacao	Sexo
10101010	Jorge	27/12/86	Filho	Masculino
10101010	Luiz	18/11/79	Filho	Masculino
20202020	Fernanda	14/02/69	Conjuge	Feminino
20202020	Angelo	10/02/95	Filho	Masculino
30303030	Fernanda	01/05/90	Filho	Feminino

Quando uma relação possui mais que uma chave (não confundir com chave composta) - como por exemplo EmpregadoId e CPF para empregados - cada uma destas chaves são chamadas de **chaves candidatas**. Uma destas chaves candidatas deve ser escolhida como **chave primária**(Figura 37).

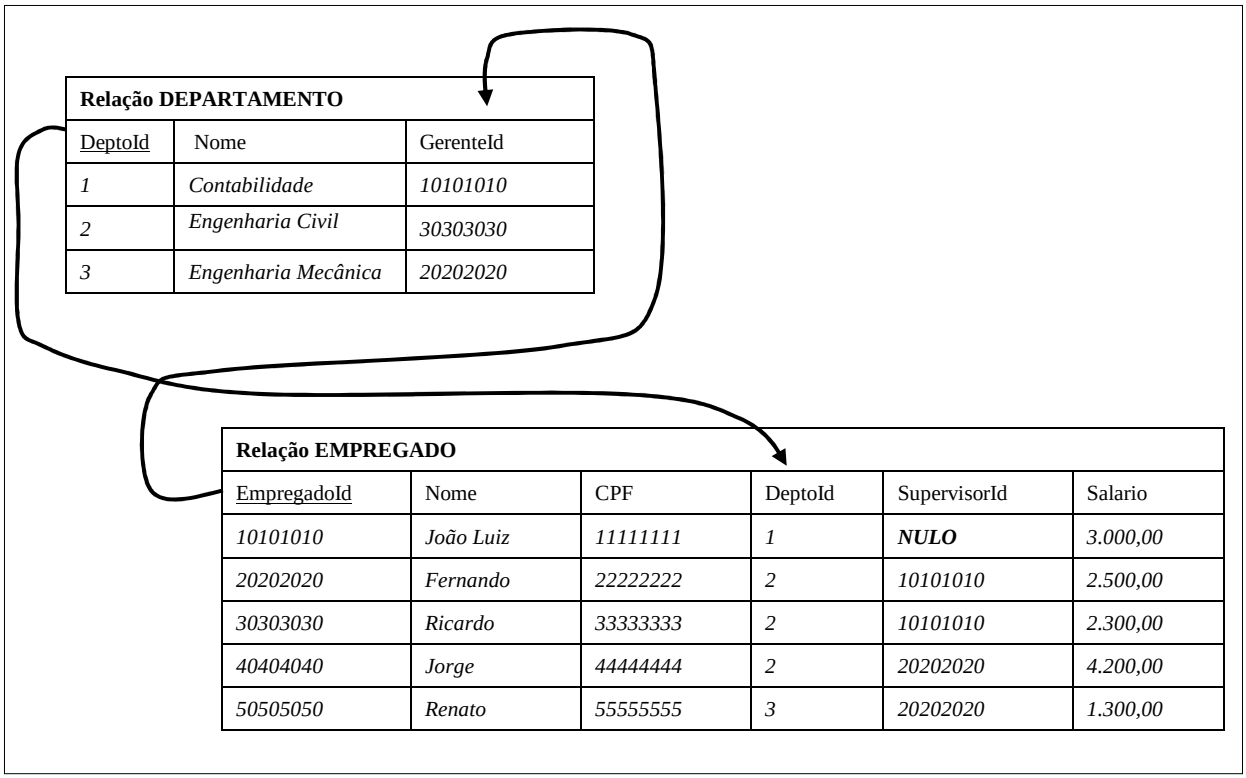
Figura 37 – Relação Empregado e suas chaves



4.3. Chave Estrangeira

Uma **chave estrangeira CE** de uma tabela $\mathbf{R_1}$ em $\mathbf{R_2}$ ou vice-versa, especifica um relacionamento entre as tabelas $\mathbf{R_1}$ e $\mathbf{R_2}$.

Figura 38 - Exemplo de Chave Estrangeira



Relação DEPARTAMENTO

DeptoId	Nome	GerenteId
1	*Contabilidade*	*10101010*
2	*Engenharia Civil*	*30303030*
3	*Engenharia Mecânica*	*20202020*

Relação EMPREGADO

EmpregadoId	Nome	CPF	DeptoId	SupervisorId	Salario
10101010	*João Luiz*	*11111111*	*1*	***NULO***	*3.000,00*
20202020	*Fernando*	*22222222*	*2*	*10101010*	*2.500,00*
30303030	*Ricardo*	*33333333*	*2*	*10101010*	*2.300,00*
40404040	*Jorge*	*44444444*	*2*	*20202020*	*4.200,00*
50505050	*Renato*	*55555555*	*3*	*20202020*	*1.300,00*

4.4. Restrições de Integridade

As definições anteriores ajudam no entendimento das restrições de integridade. A maioria das aplicações de banco de dados possui certas restrições de integridade que devem sustentar os dados. Um SGBD deve fornecer capacidades para definir e impor essas restrições. Existem três tipos de integridade.

4.4.1 Integridade de Domínio

Neste tipo de restrição de integridade, as validações ocorrem em cada campo da tabela, como por exemplo se um campo deve aceitar valores nulo ou apenas uma faixa de valores, por exemplo: Temos um campo Sexo, que só deve aceitar valores do tipo M ou F e, independente da aplicação, não deverá aceitar outros valores.

4.3.2 Integridade de Entidade

Já na restrição de integridade de entidade, temos validações um nível acima da integridade de domínio, onde define-se quais campos de nossa tabela são chaves primárias ou únicas. Um exemplo de chave única é o campo de CPF que nunca pode se repetir. Obs.: Geralmente chaves únicas são chamadas de chaves candidatas, pois são candidatas a chave primária, mas não são uma.

4.3.3 Integridade de Referência

A restrição de integridade de referencia ou referencial, cria uma ligação entre colunas tipo chave primária ou única de outras tabelas, as famosas chaves estrangeiras. Não se pode criar uma referência (relacionamento) de uma coluna se esta não for uma chave primária de outra tabela. Este tipo de integridade é fundamental para verificar se um dado será inserido de forma correta e, como é uma chave estrangeira, se este dado realmente existe.

4.3. Mapeamento do Modelo Entidade Relacionamento para o Modelo Relacional

Os dados mo diagrama entidade e relacionamento são de alto nível e precisam ser trazidos para o modelo relacional. Existem regras para esta transformação.

O mapeamento do modelo entidade relacionamento para o Modelo Relacional segue oito passos básicos a saber:

1. Para cada entidade **E** no modelo ER é criada uma tabela $\mathbf{T_1}$ no Modelo Relacional que inclua todos os atributos simples de **E**; para cada atributo composto, são inseridos apenas os componentes simples de cada um; um dos atributos chaves de **E** deve ser escolhida como a chave primária de $\mathbf{T_1}$;

2. Para cada entidade fraca **EF** com entidade proprietária **E** no modelo ER, é criada uma tabela $\mathbf{T_1}$ no Modelo Relacional incluindo todos os atributos simples de **EF**; para cada atributo composto , são inseridos apenas os componentes simples de cada um; a chave primária desta relação $\mathbf{T_1}$ será composta pela chave parcial da entidade fraca **EF** mais a chave primária da entidade proprietária **E**;

3. Para cada relacionamento regular com cardinalidade 1:1 entre entidades $\mathbf{E_1}$ e $\mathbf{E_2}$ que geraram as tabelas $\mathbf{T_1}$ e $\mathbf{T_2}$ respectivamente, devemos escolher a chave primária de uma das relações ($\mathbf{T_1}$, $\mathbf{T_2}$)e inserí-la como chave estrangeira na outra relação; se um dos lados do relacionamento tiver participação total e outro parcial, então é interessante que a chave do lado com participação **parcial** seja inserido como chave estrangeira no lado que tem participação **total**;

4. Para cada relacionamento regular com cardinalidade 1:N entre entidades $\mathbf{E_1}$ e $\mathbf{E_2}$ respectivamente e que geraram as tabelas $\mathbf{T_1}$ e $\mathbf{T_2}$ respectivamente, deve-se inserir a chave primária de $\mathbf{T_1}$ como chave estrangeira em $\mathbf{T_2}$;

5. Para cada relacionamento regular com cardinalidade N:M entre entidades $\mathbf{E_1}$ e $\mathbf{E_2}$, cria-se uma nova tabela $\mathbf{T_1}$, contendo todos os atributos do relacionamento mais o atributo chave de $\mathbf{E_1}$ e o atributo chave de $\mathbf{E_2}$; a chave primária de $\mathbf{T_1}$ será composta pelos atributos chave de $\mathbf{E_1}$ e $\mathbf{E_2}$;

6. Para cada atributo multivalorado $\mathbf{A_1}$, cria-se uma tabela $\mathbf{T_1}$, contendo o atributo multivalorado $\mathbf{A_1}$, mais o atributo chave **C** da tabela que representa a entidade ou relacionamento que contém $\mathbf{A_1}$; a chave primária de $\mathbf{T_1}$ será composta por $\mathbf{A_1}$ mais **C**; se $\mathbf{A_1}$ for composto, então a tabela $\mathbf{T_1}$ deverá conter todos os atributos de $\mathbf{A_1}$;

7. Para cada relacionamento n-ário, n > 2, cria-se uma tabela $\mathbf{T_1}$, contendo todos os atributos do relacionamento; a chave primária de $\mathbf{T_1}$ será composta pelos atributos chaves das entidades participantes do relacionamento;

8. Converta cada especialização com *m* subclasses $\mathbf{\{S_1, S_2, ..., S_m\}}$ e superclasse **SC**, onde os atributos de **SC** são $\mathbf{\{c, a_1, a_2, ..., a_n\}}$ onde **c** é a chave primária de **SC**, em tabelas utilizando uma das seguintes opções:

   8.1. Crie uma tabela **T** para **SC** com os atributos $\mathbf{A(T) = \{c, a_1, a_2, ..., a_n\}}$ e chave $\mathbf{C(T) = c}$; crie uma tabela $\mathbf{T_i}$ para cada subclasse $\mathbf{S_i}$, $1 \leq i \leq m$, com os atributos $\mathbf{A(T_i) = \{c\} \cup A(S_i)}$, onde $\mathbf{C(T) = c}$;

   8.2. Crie uma tabela $\mathbf{T_i}$ para cada subclasse $\mathbf{S_i}$, $1 \leq i \leq m$, com os atributos $\mathbf{A(T_i) = A(S_i) \cup \{c, a_1, a_2, ..., a_n\}}$ e $\mathbf{C(T_i) = c}$;

   8.3. Crie uma tabela **T** com os atributos $\mathbf{A(T) = \{c, a_1, a_2, ..., a_n\} \cup A(S_1) \cup ... \cup A(S_m) \cup \{t\}}$ e $\mathbf{C(T) = c}$, onde **t** é um atributo **tipo** que indica a subclasse à qual cada tupla pertence, caso isto venha a ocorrer;

   8.4. Crie uma tabela **T** com atributos $\mathbf{A(T) = \{c, a_1, a_2, ..., a_n\} \cup A(S_1) \cup ... \cup A(S_m) \cup \{t_1, t_2, ..., t_m\}}$ e $\mathbf{C(T) = c}$; esta opção é para generalizações com “overlapping”, e cada $\mathbf{t_i}$, $1 \leq i \leq m$, é

um atributo “booleano” indicando se a tupla pertence ou não à subclasse $\mathbf{S_i}$; embora funcional, esta opção pode gerar uma quantidade muito grande de valores nulos;

Figura 39 - Mapeamento para o Modelo Relacional

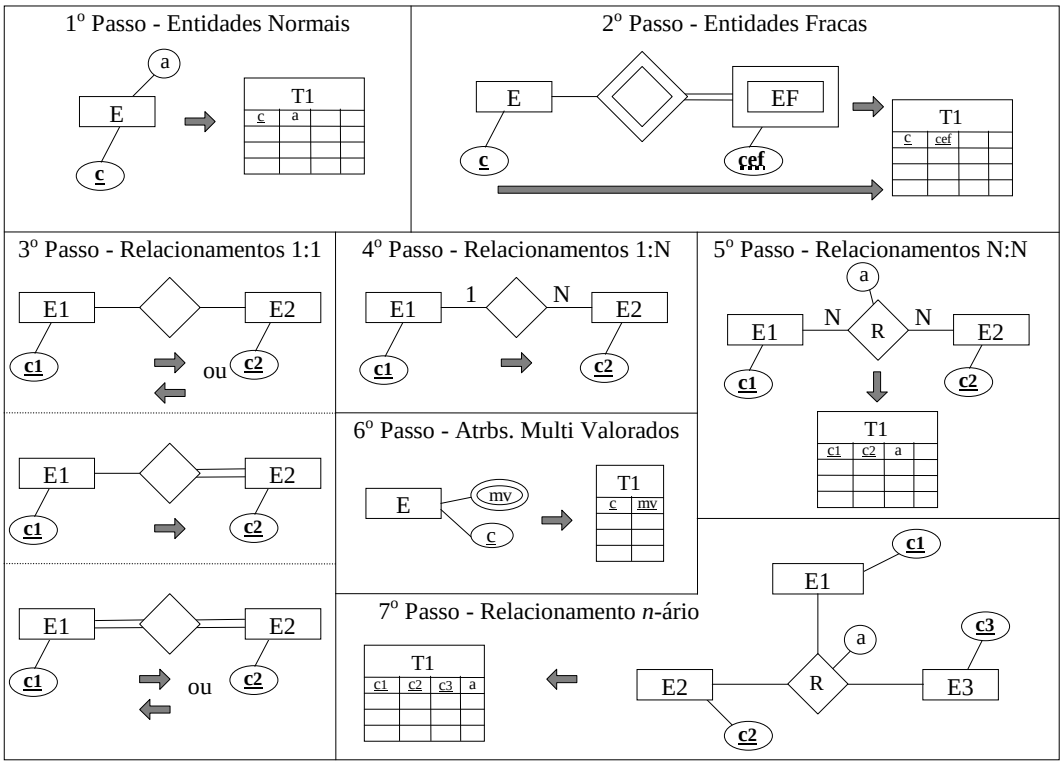
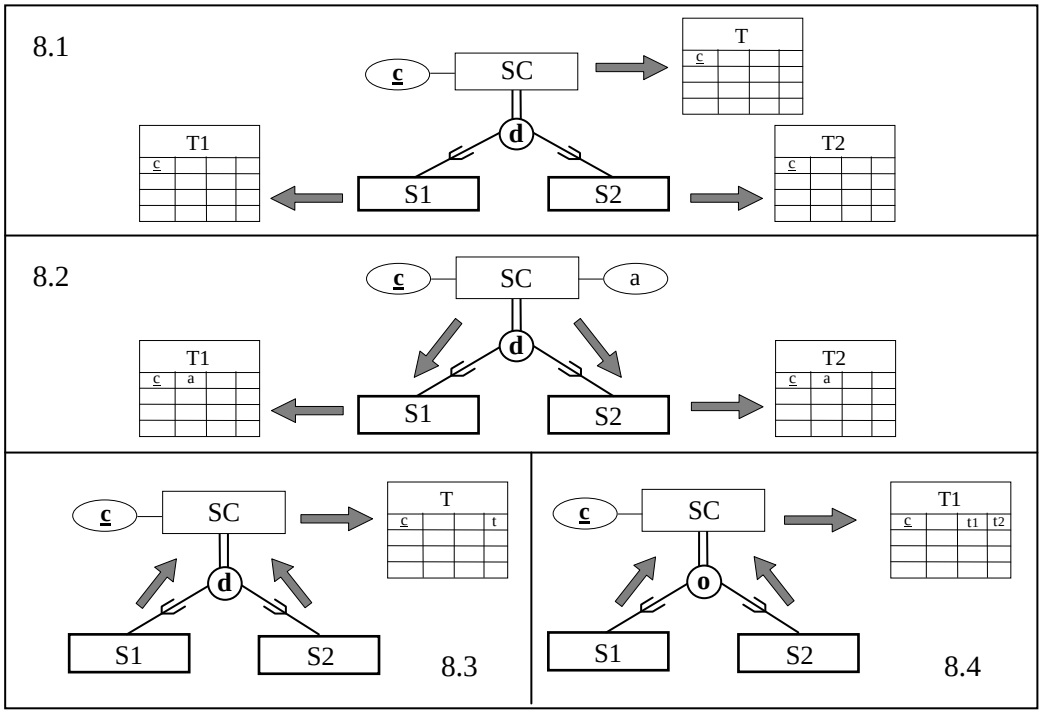


Figura 40 - Mapeamento para o Modelo Relacional com Subclasses



4.4. Dicionário de Dados

O dicionário de dados é um repositório de dados sobre os dados do software. Ele deverá conter a definição dos elementos que do Modelo de Dados, como Entidades e Atributos.

Existem muitos esquemas de notação comuns usadas pelos analistas de sistemas. O que está mostrado a seguir está entre os mais comuns, e usa alguns símbolos simples:

Símbolo	Significado
=	É composto de
+	E
[ ]	Seleção (Escolha uma das opções alternativas)
{ }	Iteração (indica ocorrência repetida, “zero ou mais ocorrências de”)
( )	Opcional (pode estar presente ou ausente)
|	separa opções alternativas na construção [ ] (Seleção)
* *	Comentário
@	Identificador (campo chave) de uma entidade.

Para definir completamente um elemento de dados, definições podem incluir:

- O *significado* do elemento de dados no contexto desta aplicação. Isso é normalmente apresentando como um comentário.
- A *composição* do elemento de dados, se ele for composto por componentes elementares significativos.
- Os *valores* que o elemento de dados poderá assumir, se ele for um elemento de dados elementar que não pode ser decomposto.

Elementos de dados elementares são aqueles para os quais não existe decomposição significativa no contexto da aplicação. Isto é, muitas vezes, uma questão de interpretação e que deve ser explorada cuidadosamente com o usuário.

Para exemplificar, podemos definir os esquemas cliente e produto como:

caracter_alfabético = [A-Z | a-z] * uma letra do alfabeto *
caracter_alfanumérico = [A-Z | a-z | 0-9 | ‘ | - | |] * um número, uma letra ou um sinal de pontuação *
caracter_numérico = [0-9] * um número decimal *
Cliente = @ClienteId + nome_cliente + endereço + limite_credito + sexo + data_nascimento * um cliente da empresa XYZ *
ClienteId = numero_chave * identificação de um cliente *
data = “DD/MM/YYYY” * uma data valida no formato dois dígitos numéricos para o dia, dois para o mês e quatro para o ano*
data_nascimento = data * data de nascimento do cliente, data superior a 01/01/1910 *
endereço = 1{caracter_alfanumérico}50 * endereço de um cliente para contato *
limite_credito = valor_monetário * valor do credito que será concedido a um cliente para pedidos não previamente pagos * * valor maior que zero *
nome_cliente = título_cortesia + primeiro_nome + (nome_intermediário) + último_nome * nome do cliente* * preenchimento obrigatório *
nome_intermediário = 1{caracter_alfabético}20
nome_produto = 1{caracter_alfabético}40
numero_chave = 1{caracter_numérico}12 * valor numérico para um campo chave *
preço = valor_monetário
primeiro_nome = 1{caracter_alfabético}20
Produto = @ProdutoId + nome_produto + preço + unidade * um produto da empresa XYZ *
ProdutoId = numero_chave * identificação de um produto *

sexo = [“M” | “F”] * Sexo M para Masculino e F para Feminino *
título_cortesia = [“Sr.” | “Sra.” | “Srta.”| “Sras.” | “Dr.” | “Professor”]
último_nome = 1{caracter_alfabético}20
unidade = [“UN” | “DZ” | “CX” | “KG” | “GR”] * unidade do produto*
valor_monetário = 0{caracter_numérico}9, 0{caracter_numérico}2* valor numérico com casas decimais*

No livro Análise Estruturada Moderna Edward Yourdon ano 1992 capitulo 10 e Anexo F página 732 você encontrará mais detalhes e um exemplo completo de dicionário de dados.

4.5. Dependência Funcional

4.5.1. Dependência Funcional

Uma **dependência funcional (DF)** é uma restrição entre dois conjuntos de atributos de uma base de dados. Suponha que o esquema de uma base de dados **R** possua *n* atributos A_1, A_2, ..., A_n; pense em **R** = **{ A_1, A_2, ... , A_n }** como a representação universal da base de dados. Uma dependência funcional, representada por **X → Y** entre dois conjuntos de atributos **X** e **Y** que são subconjuntos de **R** especificam uma restrição nas tuplas que podem compor uma instância relação *r* de **R**. A restrição estabelece que para qualquer par de tuplas t_1 e t_2 em *r* de forma que t_1.**[X]** = t_2.**[X]**, é obrigado a existir t_1.**[Y]** = t_2.**[Y]**. Isto significa que os valores do componente **Y** em uma tupla em *r* **depende de**, ou **é determinada pelos** valores do componente **X**. Para **X → Y** lê-se: ***Y** é funcionalmente dependente de **X***, ou ***X** infere sobre **Y***. Os atributos **X** e **Y** podem ser compostos.

Para a base de dados do apêndice A temos como exemplo as seguintes dependências funcionais:

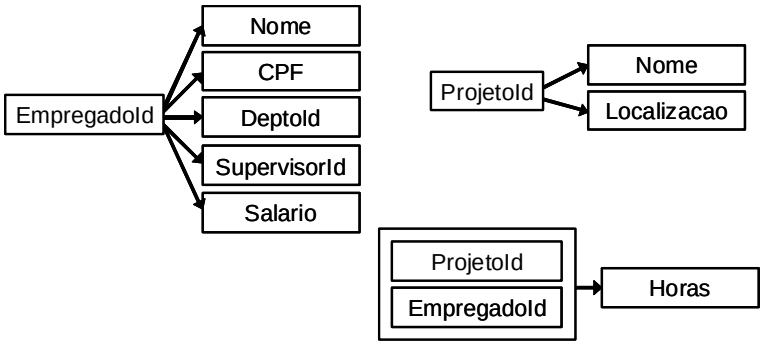
1. **EmpregadoId → { Nome, CPF, DeptoId, SupervisorId, Salario }**
2. **ProjetoId → { Nome, Localizacao }**
3. **{ EmpregadoId, ProjetoId } → Horas**

além de outras.

A dependência 1 implica que o número de um **EmpregadoId** define de forma única o nome do empregado e o CNPF do empregado. A dependência 2 implica que o **ProjetoId** define de forma única o nome do projeto e sua localização e a dependência 3 implica que o **EmpregadoId** mais o **ProjetoId** define de forma única o número de horas que o empregado trabalhou no projeto. A especificação das inferências deve ser elaborada pelo projetista de banco de dados em conjunto com o analista de sistemas, pois os mesmos deverão ter conhecimento da semântica da base de dados.

As dependências podem ser representadas também através de diagramas de dependências. A figura abaixo representa o exemplo anterior.

Figura 41 - Diagrama de Dependências



4.5.1.1. Dependência Funcional Multivalorada

Formalmente, uma dependência funcional multivalorada (DMV) X→→Y especificada no esquema de relação R, em que tanto X como Y são subconjuntos de R, especifica a seguinte restrição em qualquer estado da relação *r* de R: se duas tuplas t_1 e t_2, existirem em *r* de modo que t_1[X] = t_2[X], então duas tuplas t_3 e t_4 também deveriam existir em r com a seguintes propriedades(t_1,t_2,t_3,t_4 não são

necessariamente distintas), onde utilizamos Z(é a abreviação para os atributos remanescentes em R depois que os atributos em (X∪Y) são removidos de R) para representar (R – (X ∪ Y)):

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$
- $t_3[X] = t_1[X]$ e $t_4[X] = t_2[X]$
- $t_3[X] = t_2[X]$ e $t_4[X] = t_1[X]$

Sempre que X →→ Y se mantém, dizemos que X **multidetermina** Y. Devido à simetria na definição, sempre que X →→ Y se mantém em R, também o faz X →→ Z. Portanto, X →→ Y implica X→→ Z, e por isso é às vezes escrito como X →→ Y|Z.

A definição formal especifica que, dado um determinado valor de X, o conjunto de valores Y, determinado por esse valor de X é completamente determinado por X sozinho e *não depende* dos valores dos atributos remanescentes Z de R. Portanto sempre que existem duas tuplas que têm valores distintos de Y mas com o mesmo valor de X, esses valores de Y devem ser repetidos em tuplas separadas com *todo valor distinto de* Z que ocorra com o mesmo valor de X. Isso informalmente corresponde a Y sendo um atributo multivalorado das entidades representadas por tuplas em R.

4.5.2. Normalização

O processo de **normalização** pode ser visto como o processo no qual são eliminados esquemas de relações (tabelas) não satisfatórios, decompondo-os, através da separação de seus atributos em esquemas de relações menos complexas, mas que satisfaçam as propriedades desejadas.

O processo de normalização como foi proposto inicialmente por Ted Codd conduz um esquema de relação através de uma bateria de testes para certificar se o mesmo está na **1ª**, **2ª** e **3ª Formas Normais**. Estas três formas normais partem do principio de que um conjunto de dependências funcionais é dado para cada relação, e que cada relação tem uma chave primária especificada. Por isto são chamadas de Formas Normais Baseadas em Chaves Primárias.

Quando definimos cuidadosamente um Diagrama ER, identificando todas as entidades corretamente, os esquemas de relação gerados do diagrama ER não precisam de muito mais normalização.Entretanto, pode haver dependências funcionais entre os atributos de uma entidade. Por exemplo:Suponha que uma entidade ***funcionário*** tenha atributos *número_depto* e *endereço_depto* e que exista uma dependência funciona *número_depto*→*endereço_depto*. Poderíamos então normalizar a relação gerada de funcionário.

A maioria dos exemplos dessas dependências surge de um projeto ruim de diagrama ER. Nesse exemplo, se tivéssemos projetado o diagrama ER corretamente, teríamos criado uma entidade *depto* com o atributo *endereço_depto* e uma relação entre ***funcionário*** e ***depto***. Da mesma forma, uma relação envolvendo mais de duas entidades pode não estar em uma forma normal desejável. Como a maioria dos relacionamentos é binária, esses casos são relativamente raros.(Na verdade, algumas variantes do diagrama ER realmente dificultam ou impossibilitam especificar relações não binárias).

As dependências funcionais podem ajudar a detectar um mau projeto ER. Se as relações geradas não estiverem na forma normal desejada, o problema pode de ser corrigido no diagrama E-R. Ou seja, a normalização pode ser deixada por conta da intuição do projetista durante a modelagem ER, e pode ser feita formalmente nas relações geradas do modelo ER.

Inicialmente Frank "Ted" Codd propôs três formas normais, mais tarde uma 3FN mais sólida chamada de formal normal de Boyce-Codd (FNBC) foi proposta. Todas estas formas normais são baseadas nas dependências funcionais entre atributos de uma relação. Posteriormente, foram propostos uma **quarta forma normal (4FN)** e uma **quinta forma normal (5FN)**, baseadas nos conceitos de dependências multivaloradas e dependências de junção respectivamente.

Figura 42 - Restrições das Formas Normais

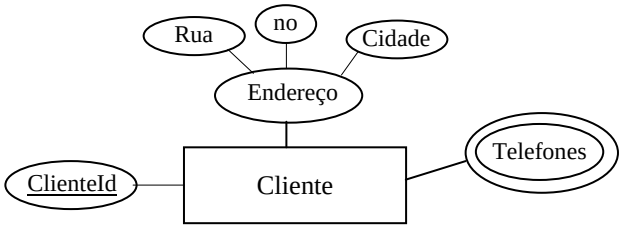
Como conseqüência do processo de normalização temos: problemas com anomalias e inconsistências diminuem, relações são simplificadas e estrutura mais regulares, aumento da integridade dos dados e eventual queda de performance.

4.5.2.1. 1ª Forma Normal

A **1ª Forma Normal** prega que todos os atributos de uma tabela devem ser **atômicos** (indivisíveis), ou seja, não são permitidos atributos multivalorados, atributos compostos ou atributos multivalorados compostos. Leve em consideração o esquema a seguir:

- **CLIENTE**

1. ClienteId
2. { Telefone }
3. Endereço: (Rua, Número, Cidade)



gerando a tabela resultante:

{ ClienteId } → { {Telefone}, Endereço (Rua Número, Cidade)}

		Telefone 1	Endereço		
Cliente	ClienteId	Telefone *n*	Rua	No	Cidade

sendo que a mesma não está na 1ª Forma Normal pois seus atributos não são atômicos. Para que a tabela acima fique na 1ª Forma Normal temos que eliminar os atributos não atômicos, gerando as seguintes tabelas como resultado:

{ ClienteId } → { Rua Número, Cidade}

{ ClienteId, Telefone } → { ø }

Cliente	ClienteId	Rua	Número	Cidade

Cliente_Telefone	ClienteId	Telefone

4.5.2.2. 2ª Forma Normal

A **2ª Forma Normal** prega o conceito da **dependência funcional total**. Uma dependência funcional **X → Y** é **total** se removemos um atributo **A** qualquer do componente **X** e desta forma, a dependência funcional deixa de existir. A dependência funcional **X → Y** é uma **dependência funcional parcial** se existir um atributo **A** qualquer do componente **X** que pode ser removido e a dependência funcional **X → Y** não deixa de existir.

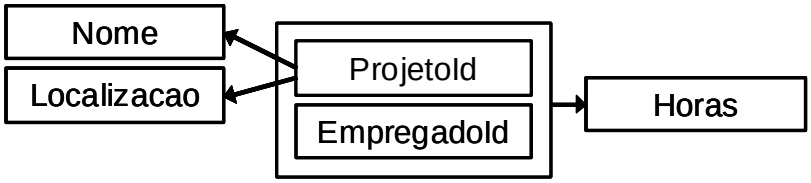
Veja a dependência funcional 3 do ítem **4.5.1. Dependência Funcional**:

{ EmpregadoId , ProjetoId } → Horas (2FN)

é uma dependência funcional total, pois se removermos o atributo **EmpregadoId** ou o atributo **ProjetoId**, a dependência funcional deixa de existir.

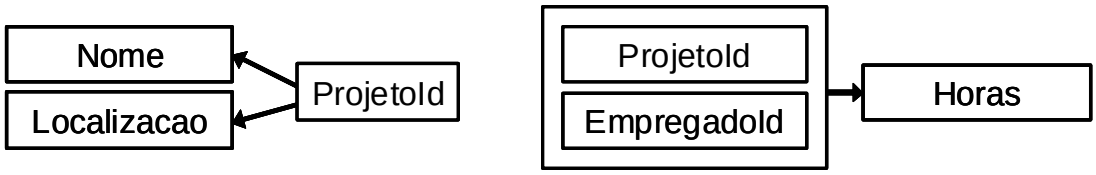
Uma tabela **T** está na 2ª Forma Normal se estiver na 1ª Forma Normal e todo atributo que não compõem a chave primária **C** for totalmente funcionalmente dependente da chave primária **C**. Se uma tabela não está na 2ª Forma Normal à mesma pode ser normalizada gerando outras tabelas cujos atributos que não façam parte da chave primária sejam totalmente funcionalmente dependente da mesma, ficando a tabela na 2ª Forma Normal.

{ EmpregadoId , ProjetoId } → {Horas, Nome, Localização}



Aplicando a 2a Forma normal:

{ EmpregadoId , ProjetoId } → Horas (2FN)



Problemas resolvidos com esta normalização

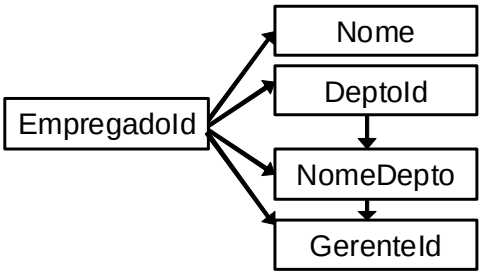
- Inserção: projeto só pode ser inserido quando existir empregados com suas respectivas horas.
- Remoção: ao se remover algum projeto, necessário remover também a informação das horas a ele associado.
- Atualização: a chave **ProjetoId** se repete em Projeto e Horas Projeto.

4.5.2.3. 3ª Forma Normal

A **3ª Forma Normal** prega o conceito de **dependência transitiva**. Considere a relação T composta pelos atributos X, Z e Y. Uma dependência funcional **X → Y** em uma tabela **T** é uma dependência transitiva se existir um conjunto de atributos **Z** que não é um subconjunto de chaves de **T** e as dependências **X → Z**, **Z → Y**, são válidas sendo que Y não pertence nem a Z e X. Considere a seguinte tabela como exemplo:

EmpregadoId → { Nome , *DeptoId*, NomeDepto, GerenteId}

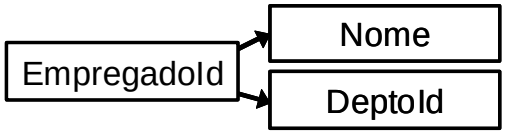
Empregado	EmpregadoId	Nome	DeptoId	NomeDepto	GerenteId

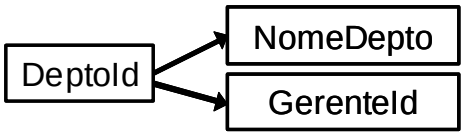


onde temos a seguinte dependência transitiva:

Empregado → { Nome , *DeptoId*}

***DeptoId* → { Nome, GerenteId}**





Outro Exemplo:

NotaFiscal	NumeroNF	Total	ClienteId	NomeCliente	Item	ProdutoId	NomeProduto	QtdeVendida

onde temos a seguinte dependência transitiva:

NumeroNF → { ClienteId, Total } (1FN)

ClienteId → { Nome } (1FN)

{NumeroNF, Item} → {*ProdutoId*, QtdeVendida} (2FN)

***ProdutoId* → { Nome} (3FN)**

Uma tabela está na 3ª Forma Normal se estiver na 2ª Forma Normal e não houver dependência transitiva entre atributos não chave.

Problemas resolvidos com esta normalização

- Inserção: não se pode inserir um departamento até que haja um empregado neste departamento.
- Remoção: ao se remover o ultimo empregado de um departamento remove-se os dados do departamento.
- Atualização: quando o gerente de um departamento muda necessário atualizar diversas tuplas.

4.5.3. Resumo das Formas

A Tabela 1 abaixo informalmente resume as três formas normais em chaves primárias, os testes utilizado em cada caso e a normalização ou "remédio" correspondente para se alcançar a forma normal.

Tabela 1 - Resumo das Formas Normais

Forma Normal	Teste	Solução(Normalização)
Primeira (**1FN**)	A relação não deve ter qualquer atributo não atômico nem relações agrupadas (multivalores).	Forme novas relações para cada atributo não atômico ou relação agrupada (multivalorada).
Segunda (**2FN**)	Para relações nas quais a chave primária contém múltiplos atributos, nenhum atributo não chave deve ser funcionalmente dependente de uma parte da chave primária.	Decomponha e monte uma relação para cada chave parcial com seu(s) atributo(s) dependente(s). Certifique-se de manter uma relação com a chave primária original e quaisquer atributos que sejam completamente dependentes dela em termos funcionais.
Terceira (**3FN**)	A relação não deve ter um atributo não chave funcionalmente determinado por um outro atributo não chave (ou por um conjunto de atributos não chave). Ou seja, não deve haver dependências transitivas de um atributo não chave na chave primária.	Decomponha e monte uma relação que inclua o(s) atributo(s) não chave que funcionalmente determine(m) outros atributos não chave.

4.5.4. Outras Formas

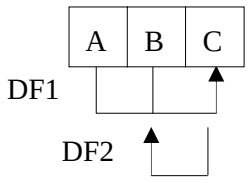
4.5.4.1. Forma Normal de Boyce-Codd

A forma normal de Boyce-Codd(FNBC) foi proposta como uma forma mais simples da 3FN, mas foi considerada mais restrita do que ela, porque toda relação na FNBC também está na 3FN; entretanto, uma relação na 3FN não está necessariamente na FNBC.

Um esquema de relação R está na FNBC se, sempre que uma dependência funcional não-trivial X→A se mantiver em R, X for uma superchave de R. A única diferença entre as definições da FNBC e da 3FN é que a condição da 3FN que diz que permite que A seja principal, está ausente da FNBC.

Na prática, a maioria dos esquemas de relação que estão na 3FN também estão na FNBC. Somente se X→A se mantiver em um esquema de relação R, não sendo X uma superchave e sendo A um atributo principal, R estará na 3FN mas não na FNBC.

A relação abaixo não esta na terceira forma normal.



DF1 : {aluno, disciplina}→instrutor

DF2 : instrutor→disciplina

Note que (aluno, disciplina) é uma chave candidata para essa relação e que as dependências mostradas seguem o padrão da figura acima. Portanto essa relação está na 3FN, mas não está na FNBC.

A decomposição desse esquema em dois esquemas não é simples porque pode ser decomposto em um de três pares possíveis:

1. (aluno, instrutor) e (aluno, disciplina)
2. (disciplina, instrutor) e (disciplina, aluno)
3. (instrutor, disciplina) e (instrutor, aluno)

Todas as três decomposições “perdem” a dependência DF1. A mais desejável dessas três decomposições é a terceira, porque não irá gerar tuplas inválidas após uma junção. Em geral uma relação que não esteja na FNBC, deve ser decomposta de modo a satisfazer essa propriedade (de ser sem perda de junção), mesmo que isso signifique abster-se da preservação de todas as dependências funcionais.

LECIONA		
Aluno	**Disciplina**	**Instrutor**
Naiana	Banco de Dados	João
Fernando	Banco de Dados	Pedro
Fernando	Sistemas Operacionais	Maria
Fernando	Teoria	Marco
Guilherme	Sistemas Operacionais	José
Guilherme	Banco de Dados	João
Márcia	Banco de Dados	Antonio
Zuleide	Banco de Dados	Pedro

4.5.4.2. Quarta Forma Normal

Agora a definição da quarta forma normal(4FN), que é violada quando uma relação que tem dependências multivaloradas indesejáveis, e portanto pode ser utilizada para identificar e decompor relações. Um esquema de relação R está na 4FN com relação a um conjunto de dependências F (que inclui dependências funcionais e dependências multivaloradas) se, para toda dependência multivalorada não-trivial $X \rightarrow\rightarrow Y$ em F^+, X for uma superchave de R.

Por exemplo, considere a relação EMPREGADO mostrada na Figura 43. Uma tupla nessa relação EMPREGADO representa o fato de que um empregado cujo nome seja NomeEmpregado trabalha no projeto cujo nome é NomeProjeto e tem um dependente cujo nome é NomeDependente. Um empregado pode trabalhar em diversos projetos e pode ter diversos dependentes, e os projetos e dependentes do empregado são independentes um do outro. Em um diagrama ER, cada qual seria representado como um atributo multivalorado ou como um tipo de entidade fraca.

Figura 43 - Quarta Forma Normal

Relação EMPREGADO		
NomeEmpregado	NomeProjeto	NomeDependente
João	X	Pedro
João	Y	Maria
João	X	Maria
João	Y	Pedro

Relação EMPREGADO_PROJETOS	
NomeEmpregado	NomeProjeto
Joao	X
Joao	Y

Relação DEPENDENTES	
NomeEmpregado	NomeDependente
Joao	Pedro
Joao	Maria

A relação EMPREGADO da Figura 43 não está na 4FN porque nas DMVs não triviais NomeEmpregado→→NomeProjeto e NomeEmpregado→→NomeDependente, NomeEmpregado não é a superchave de EMPREGADO. Decompomos EMPREGADO em EMPREGADO_PROJETOS e DEPENDENTES. Tanto EMPREGADO_PROJETOS como DEPENDENTES são DMVs triviais. Nenhuma outra DMV não trivial se mantém em EMPREGADO_PROJETOS e DEPENDENTES.

4.5.4.3. Quinta Forma Normal

Também chamada *forma normal de junção de projeto*. Um esquema de relação R está na quinta forma normal (5FN) (ou forma normal de projeção de junção[FNPJ]) com respeito a um conjunto F de dependências funcionais, multivaloradas e de junção se, para toda dependência de junção não trivial $DJ(R_1,R_2,...,R_n)$ em F^+ (ou seja, implicada por F), todo R_i for uma superchave de R.

Figura 44 - Quinta Forma Normal

Relação FORNECIMENTO

NomeFornecedor	NomePeca	NomeProjeto
João	Quadrado	X
João	Círculo	Y
Antonio	Quadrado	Y
Wilson	Círculo	Z
Antonio	Triângulo	X
Antonio	Quadrado	X
João	Quadrado	Y

Relação R1

NomeFornecedor	NomePeca
Joao	Quadrado
Joao	Círculo
Antonio	Quadrado
Wilson	Círculo
Antonio	Triângulo

Relação R2

NomeFornecedor	NomeProjeto
João	X
João	Y
Antonio	Y
Wilson	Z
Antonio	X

Relação R3

NomePeca	NomeProjeto
Quadrado	X
Círculo	Y
Quadrado	Y
Círculo	Z
Triângulo	X

Para exemplo de uma DJ, considere a relação toda-chave FORNECIMENTO da Figura 44. Suponha que sempre se mantenha a seguinte restrição adicional: sempre que um fornecedor *f* fornece a peça *p*, e um projeto *j* utiliza a peça *p*, e o fornecedor *f* fornece pelo menos uma peça para o projeto *j*, então o fornecedor *f* também estará fornecendo a peça *p* para o projeto *j*. Essa restrição pode ser reafirmada de outras maneiras, e especificada uma dependência de junção DJ(R1,R2,R3) entre as três projeções R1(NomeFornecedor,NomePeca), R2(NomeFornecedor,NomeProjeto) e R3(NomePeca, NomeProjeto) de FORNECIMENTO. Se essa restrição se mantém, as tuplas abaixo da linha pontilhada na Figura 44 deve existir em qualquer estado válido (legal) da relação FORNECIMENTO que também contém as tuplas acima da linha pontilhada.

A Figura 44 mostra como a relação FORNECIMENTO *com a dependência de junção* é decomposta em três relações R1, R2 e R3, que estão cada qual na 5FN. Note que aplicar a JUNÇÃO NATURAL a quaisquer duas dessas relações produz tuplas inválidas, mas com uma JUNÇÃO NATURAL aplicada as três conjuntamente não o fazem.

Na prática, descobrir DJs em banco de dados com centenas de atributos só é possível com um elevado grau de intuição sobre os dados por parte do projetista. Portanto, a prática atual de projetos de banco de dados dedica pouca atenção a elas.

5. SQL

5.1 Histórico

Certamente o SQL tem representado o padrão para linguagens de banco de dados relacionais. Existem diversas versões de SQL. A versão original foi desenvolvida pela IBM no Laboratório de Pesquisa de San José. Essa linguagem, originalmente chamada de Sequel, foi implementada como parte do projeto do Sistema R no início dos anos 70. Desde então, a linguagem Sequel foi evoluindo e seu nome foi mudado para SQL(Structured Query Language – Linguagem de Consulta Estruturada). Inúmeros produtos dão suporte atualmente para a linguagem SQL. O SQL se estabeleceu claramente como a linguagem padrão de banco de dados relaicona.

Em 1986 a American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) publicaram um padrão SQL, chamado SQL-86. Em 1989, o ANSI publicou um padrão estendido para a linguagem: SQL-89. A próxima versão do padrão foi a SQL-92(SQL 2), seguida das versões SQL:1999(SQL 3), SQL:2003, SQL:2006 e SQL:2008. A versão mais recente é a SQL:2011.

5.2. Conceitos

SQL é um conjunto de declarações que é utilizado para acessar os dados utilizando gerenciadores de banco de dados.

SQL pode ser utilizada para todas as atividades relativas a um banco de dados podendo ser utilizada pelo administrador de sistemas, pelo DBA, por programadores, sistemas de suporte à tomada de decisões e outros usuários finais.

SQL é uma combinação de construtores em Álgebra Relacional.

5.3. Partes

Linguagem de Definição de Dados (Data-Definition Language – DDL). A DDL do SQL fornece comandos para definir esquemas de relações, excluir relações e modificar esquemas.

Linguagem de Manipulação de Dados Interativa (Data-Manipulaiton Language – DML). A DML da SQL inclui uma linguagem consulta baseada em álgebra relacional e no cálculo relacional de tupla. Ela também inclui comandos para inserir, excluir e modificar tuplas no banco de dados.

Integridade. A DDL SQL inclui comandos para especificar restrições de integridade às quais os dados armazenados no banco de dados precisam satisfazer. As atualizações que violam as restrições de integridade são proibidas.

Definições de visões. A DDL SQL inclui comandos para definir visões.

Controle de transações. A SQL inclui comandos para especificar o início e o fim de transações.

SQL embutida (Embedded DML) e **SQL Dinâmica**. A SQL embutida e a dinâmica definem como as instruções SQL podem ser incorporados dentro das linguagens de programação de finalidade geral como C, C++, Java, Cobol, Pascal e Fortran.

Autorização A DDL SQL inclui comandos para especificação de diretos de acesso para relações e visões.

6. SQL DDL

6.1. Linguagem de Definição de Dados

O conjunto de Relações (Tabelas) em um banco de dados deve ser especificado para o sistema por meio de uma linguagem de definição de dados(DDL).

A SQL DDL permite não só a especificação de um conjunto de relações, como também informações acerca de cada uma das relações, incluindo:

- Esquema de cada relação.
- Domínio dos valores associados a cada atributo.
- As regras de integridade.
- Conjunto de índices para manutenção de cada relação.
- Informações sobre segurança e autoridade sobre cada relação.
- A estrutura de armazenamento físico de cada relação no disco.

6.2. Esquema Base

Para DDL iremos considerar uma empresa na área bancária que possui as seguintes relações:

```
Cidade = (nome_cidade, estado_cidade)
Agencia = (nome_agencia, cidade_agencia, fundos
Cliente = (nome_cliente, rua_cliente, cidade_cliente, salario_cliente, estado_civil)
Emprestimo = (numero_emprestimo, nome_agencia,total)
Devedor = (nome_cliente, numero_emprestimo)
Conta = (numero_conta, nome_agencia, saldo, rg_funcionario_gerente)
Depositante = (nome_cliente, numero_conta)
Departamento= (codigo_departamento, nome_departamento, local_departamento, total_salario)
Funcionario = (rg_funcionario, nome_funcionario, cpf_funcionario, salario_funcionario, rg_supervisor, codigo_departamento)
```

6.3. Tipos de Domínios em SQL

O padrão SQL aceita diversos tipos de domínios internos, incluindo:

char(n) – é uma cadeia de caracteres de tamanho fixo, com o tamanho n definido pelo usuário. Pode ser usada a sua forma completa **character** no lugar de **char**.

varchar(n) – é uma cadeia de caracteres de tamanho variável, com o tamanho n máximo definido pelo usuário. A forma completa, **character varying**, é equivalente.

int – é um inteiro (um subconjunto finito de inteiros que depende do equipamento). A forma completa, **integer** é equivalente.

smallint é um inteiro pequeno (um subconjunto do domínio dos tipos inteiros que depende do equipamento).

numeric(p,d) – é um número de ponto fixo cuja precisão é definida pelo usuário. O número consiste de p dígitos (mais o sinal), e d dos p dígitos estão a direita da vírgula decimal. Assim, numeric(3,1) permite que 44,5 seja armazenado exatamente, mas nem 444,4 nem 0,32 podem ser armazenados exatamente em um campo deste tipo.

real, double precision - são números de ponto flutuante e ponto flutuante de precisão dupla cuja precisão depende do equipamento.

float(n) - é um número de ponto flutuante com precisão definida pelo usuário em pelo menos n dígitos.

date – é uma data de calendário contendo um ano (com quatro dígitos), mês e dia do mês.

time – a hora do dia, em horas, minutos e segundos.

timestamp – uma combinação de **date** e **time**.

clob – objetos grandes para dados de caractere.
blob – objetos grandes para dados binários. As letras lob significam "**Large OBject**".

A SQL permite que a declaração de domínio de um atributo inclua a especificação de **not null**, proibindo, assim, a inserção de valores nulos para este tipo de atributo.

5.1.3.1. Definição de Dominíos

A SQL permite definir domínios usando a cláusula **create domain**, como mostra o exemplo:

```
CREATE DOMAIN nome_pessoa CHAR(20);
```

Podemos então usar o domínio de nome em nome_pessoa para definir o tipo de um atributo com um domínio exato embutido.

A cláusula check da SQL permite modos poderos de restrições de domínios que a maioria dos sistemas de tipos de linguagens de programação não permite. Especificamente a cláusula check permite ao projeto do esquema determinar um predicado que deva ser satisfeito por qualquer valor designado a uma variável cujo tipo seja o domínio. Por exemplo uma cláusula check pode garantir que o domínio relativo ao salário de um funcionário contenha somente valores maiores que 0.

```
CREATE DOMAIN salario_funcionario NUMERIC(10,2)
       CONSTRAINT ck_teste_salario_funcionario CHECK (VALUE > 0);
```

A cláusula check pode ser usada para restringir os valores nulos em um domínio:

```
CREATE DOMAIN salario_cliente NUMERIC(10,2)
       CONSTRAINT teste_nulo_salario_cliente check (VALUE NOT NULL);
```

Como outro exemplo, o domínio pode ser restrito a determinado conjunto de valores por meio do uso da clúsula in:

```
CREATE DOMAIN estado_civil CHAR(20) constraint ck_teste_estado_civil
   CHECK (VALUE IN ('Casado','Solteiro','Separado'));
```

A cláusula **constraint** é opcional e é usada para dar um nome a restrição. O nome é usado para indicar quais restrições foram violadas em determinada atualização.

A cláusula check pode ser mais complexa, dado que são permitidas subconsultas que se refiram a outras relações na codição check.

Por exemplo, a seguinte codição check poderia ser especificada na relação deposito:

```
CHECK (nome_agencia IN (SELECT nome_agencia FROM agencia));
```

6.4. Definição de Esquema em SQL

Definimos uma relação SQL usando o comando **create table**:

```
CREATE TABLE r (A1 D1,
                A2 D2,
                ...,
                An Dn,
                <regras de integridade 1>,
                <regras de integridade 2>,
                ...,
                <regras de integridade n>);
```

em que r é o nome da relação, cada Ai é o nome de um atributo no esquema da relação r e Di é o tipo de domínio dos valores no domínio dos atributos Ai. É possível definir valores default para os atributos, esta especificação e feita depois da especificação do domínio do atributo pela para ***default*** e o valor desejado.

As principais regras de integridade são:

```
PRIMARY KEY (Aj1, Aj2, ..., Ajm)

CHECK (P)

FOREIGN KEY(Aj1, Aj2, ..., Ajm) REFERENCES r
```

A especificação **PRIMARY KEY** diz que os atributos Aj1, Aj2, ..., Ajm formam a chave primária da relação. É necessário que os atributos de chave primária sejam não nulos e únicos; isto é, nenhuma tupla pode ter um valor nulo para a chave primária, e nenhum par de tuplas na relação pode ser igual em todos os atributos de chave primária.

A cláusula **CHECK** especifica um predicado P que precisa ser satisfeito por todas as tuplas em uma relação. No exemplo abaixo a cláusula check foi usada para simular um tipo enumerado, especificando que estado civil pode ser atribuído.

A cláusula **FOREIGN KEY** inclui a relação dos atributos que constituem a chave estrangeira (Aj1, Aj2, ..., Ajm) quanto o nome da relação à qual a chave estrangeira faz referência(r).

Todos os atributos de uma chave primária são declarados implicitamente como **NOT NULL**.

Definição do banco de dados de exemplo:

```
CREATE TABLE Cliente (
     cpf_cliente VARCHAR(11),
     nome_cliente VARCHAR(20) NOT NULL,
     rua_cliente VARCHAR(30),
     cidade_cliente VARCHAR(40),
     estado_civil VARCHAR(20),
     rg_cliente VARCHAR(7),
     idade_cliente INT,
     data_cadastro DATE,
     PRIMARY KEY (cpf_cliente),
     CHECK (estado_civil IN ('Casado','Solteiro','Viúvo')
);

CREATE TABLE Agencia (
     codigo_agencia INT,
     nome_agencia char(15) NOT NULL,
     cidade_agencia VARCHAR(40),
     fundos NUMERIC(10,2) DEFAULT 0.0,
     PRIMARY KEY(codigo_agencia),
     CHECK (fundos >=20)
);

CREATE TABLE Emprestimo(
     numero_emprestimo INT NOT NULL,
     codigo_agencia INT NOT NULL,
     total  NUMERIC(10,2),
     PRIMARY KEY(numero_emprestimo),
     FOREIGN KEY (codigo_agencia) REFERENCES agencia,
     CHECK (total > 0)
);

CREATE TABLE Devedor(
     cpf_cliente VARCHAR(11) NOT NULL,
     numero_emprestimo INT NOT NULL,
     PRIMARY KEY (cpf_cliente, numero_emprestimo),
     FOREIGN KEY (cpf_cliente) REFERENCES cliente,
     FOREIGN KEY (numero_emprestimo) REFERENCES empréstimo
);
```

```
CREATE TABLE Conta(
     numero_conta INT,
     codigo_agencia INT NOT NULL,
     cpf_cliente VARCHAR(11) NOT NULL,
     saldo NUMERIC(10,2) DEFAULT 0.0,
     PRIMARY KEY (numero_conta),
     FOREIGN KEY (codigo_agencia) REFERENCES agencia,
     FOREIGN KEY (cpf_cliente) REFERENCES cliente
);

CREATE TABLE Depositante(
     cpf_cliente VARCHAR(11),
     numero_conta INT,
     PRIMARY KEY (cpf_cliente, numero_conta),
     FOREIGN KEY (cpf_cliente) REFERENCES cliente,
     FOREIGN KEY (numero_conta) REFERENCES conta
);
```

6.4.2. Esvaziar Tabelas

Para apagar o conteúdo de uma tabela sem remover o esquema utiliza um dos comandos abaixo.

```
TRUNCATE TABLE <Nome_Tabela>;
```

ou

```
DELTE FROM <Nome_Tabela>;
COMMIT;
```

Onde <Nome_Tabela> é a tabela que se deseja esvaziar o conteúdo sem perder o esquema no banco de dados.

6.4.3. Remover Tabelas

O comando drop table remove todas as informações de uma relação do banco de dados. Tanto os dados como o esquema é excluído.

```
DROP TABLE <Nome_Tabela>;
```

Onde <Nome_Tabela> é a tabela que se deseja excluir do banco de dados.

6.4.4. Adicionar Atributos

Usamos o comando alter table em SQL para adicionar atributos a uma relação existente. Todas as tuplas da relação recebem valores nulos para seu novo atributo.

```
ALTER TABLE <Nome_Tabela> ADD <Nome_Atributo> <Tipo_Atributo>;
```

Onde <Nome_Tabela> é o nome da relação existente, <Nome_Atributo> é o nome do novo atributo que será adicionado e <Tipo_Atributo> é seu domínio.

6.4.5. Alterar Atributos

Usamos o comando alter table em SQL também para alterar atributos a uma relação existente.

```
ALTER TABLE <Nome_Tabela> MODIFY <Nome_Atributo> <Tipo_Atributo>;
```

Onde <Nome_Tabela> é o nome da relação existente, <Nome_Atributo> é o nome do atributo que será alterado e <Tipo_Atributo> é seu novo domínio. O atributo não deve ter valores a ser que seja literal e se deseja alterar o tamanho.

6.4.6. Renomear Atributos

Usamos o comando alter table em SQL também para renomear atributos de uma relação existente.

```
ALTER TABLE <Nome_Tabela> RENAME COLUMN <Nome_Origem> TO <Nome_Destino>;
```

Onde <Nome_Tabela> é o nome da relação existente, <Nome_Origem> é o nome do atributo que será alterado e <Nome_Destino> é seu novo nome.

6.4.7. Excluir Atributos

Podemos excluir atributos de uma relação usando o comando

```
ALTER TABLE <Nome_Tabela> DROP COLUMN <Nome_Atributo>;
```

Onde <Nome_Tabela> é a tabela de onde se deseja apagar o atributo <Nome_Atributo>.

6.4.8. Renomear Tabelas

Permite da um novo nome a uma tabela.

```
RENAME <Nome_Origem> TO <Nome_Destino>;
```

Onde <Nome_Origem> é a tabela que deseja renomear e <Nome_Destino> o novo nome para a tabela. O comando rename também funciona para visões, seqüências e sinônimos.

6.5. Integridade

No SQL todas as regras de integridade de dados e entidade são definidas por objetos chamados CONSTRAINT. Que podem ser definidos quando da criação da tabela ou posteriori via comando ALTER TABLE.

Os constraints suportados são:

```
NOT NULL
UNIQUE
PRIMARY KEY
FOREIGN KEY
CHECK
```

6.5.1. Definição de Constraints

6.5.1.1. CONSTRAINTS IN-LINE

Exemplo:

```
CREATE TABLE Cliente (
     cpf_cliente VARCHAR(11) CONSTRAINT PK_cliente PRIMARY KEY,
     nome_cliente VARCHAR(20) not null,
     rua_cliente VARCHAR(30),
     cidade_cliente VARCHAR(40),
     estado_civil VARCHAR(20),
     rg_cliente VARCHAR(7),
     idade_cliente INT,
     data_cadastro DATE);
```

6.5.1.2. CONSTRAINTS OUT-OF-LINE

Exemplo:

```
CREATE TABLE Cliente (
  cpf_cliente VARCHAR(11),
  nome_cliente VARCHAR(20) NOT NULL,
```

```
  rua_cliente VARCHAR(30),
  cidade_cliente VARCHAR(40),
  estado_civil VARCHAR(20),
  rg_cliente VARCHAR(7),
  idade_cliente INT,
  data_cadastro DATE,
  CONSTRAINT PK_cliente PRIMARY KEY (cpf_cliente));
```

Nota: Quando o constraint for definido sem nome, o oracle define um nome para o mesmo - `sys_c00n` - onde n é um número seqüencial crescente.

6.5.2. Constraints

6.5.2.1. NOT NULL CONSTRAINT

A especificação not null proíbe a inserção de um valor nulo para esse atributo. Qualquer modificação de banco de dados que causaria a inserção de um nulo em um atributo declarado para ser not null gera um erro.

Exemplo de uma tabela que precisa que todos os campos sejam preenchidos:

```
CREATE TABLE cliente (
  cpf_cliente VARCHAR(11) CONSTRAINT NN_cliente_cpf NOT NULL,
  nome_cliente VARCHAR(20) CONSTRAINT NN_cliente_nome NOT NULL,
  rua_cliente VARCHAR(30) CONSTRAINT NN_cliente_rua NOT NULL,
  cidade_cliente VARCHAR(40) CONSTRAINT NN_cliente_cidade NOT NULL,
  estado_civil VARCHAR(20) CONSTRAINT NN_cliente_estado_civil NOT NULL,
  rg_cliente VARCHAR(7) CONSTRAINT NN_cliente_rg NOT NULL,
  data_cadastro DATE CONSTRAINT NN_cliente_data_cadastro NOT NULL);
```

6.5.2.2. UNIQUE CONSTRAINT

Atributos de uma declaração que seja **unique** (isto é, atributos de uma chave candidata) têm a garantia de unicidade deste valor para todas as tuplas. Mesmo a tabela já possuindo chave primária pode-se garantir a unicidade do atributo.

Exemplo de uma tabela que possui um atributo como chave primária e um outro com uma chave candidata ativada como única.

```
CREATE TABLE cliente (
  cpf_cliente VARCHAR(11),
  nome_cliente VARCHAR(20) CONSTRAINT NN_cliente_nome NOT NULL,
  rua_cliente VARCHAR(30),
  cidade_cliente VARCHAR(40),
  estado_civil VARCHAR(20),
  rg_cliente VARCHAR(7),
  data_cadastro DATE,
  CONSTRAINT UK_cliente_rg UNIQUE (rg_cliente));
```

6.5.2.3. PRIMARY KEY CONSTRAINT

Valor único que identifica cada linha da tabela.

Exemplo:

```
CREATE TABLE cliente (
  cpf_cliente VARCHAR(11),
  nome_cliente VARCHAR(20) CONSTRAINT NN_cliente_nome NOT NULL,
  rua_cliente VARCHAR(30),
  cidade_cliente VARCHAR(40),
  estado_civil VARCHAR(20),
  rg_cliente VARCHAR(7),
  data_cadastro DATE,
```

```
  CONSTRAINT PK_cliente PRIMARY KEY(cpf_cliente),
  CONSTRAINT UK_cliente_rg UNIQUE (rg_cliente));
```

6.5.2.4. FOREIGN KEY CONSTRAINT

Deve estar associada a uma **primary key** ou **unique** definida anteriormente.
Pode assumir valor nulo ou igual ao da chave referenciada.
Não existe limite para um número de **foreign keys**.
Garante a consistência com a primary key referenciada.
Pode fazer referência a própria tabela.
Não pode ser criada para views, synonyns e remote table.

Exemplo out-of-line:

```
CONSTRAINT FK_conta_cliente
     FOREIGN KEY (cpf_cliente)
          REFERENCES cliente(cpf_cliente)
```

ou

```
CONSTRAINT FK_conta_cliente
     FOREIGN KEY (cpf_cliente) REFERENCES cliente
```

Exemplo in-line:

```
CREATE TABLE Conta (
     ...
     cpf_cliente VARCHAR(11) CONSTRAINT FK_conta_cliente
          REFERENCES cliente,
     ...);
```

6.5.2.5. CHECK CONSTRAINT

As validações de colunas são feitas utilizando o CHECK CONSTAINT.

Exemplo:

```
CREATE TABLE cliente(
  cpf_cliente VARCHAR(11),
  nome_cliente VARCHAR(20) CONSTRAINT NN_cliente_nome NOT NULL,
  rua_cliente VARCHAR(30),
  cidade_cliente VARCHAR(40),
  estado_civil VARCHAR(20),
  rg_cliente VARCHAR(7),
  data_cadastro DATE,
  idade_cliente INT,
  CONSTRAINT PK_cliente PRIMARY KEY(cpf_cliente),
  CONSTRAINT UK_cliente_rg UNIQUE (rg_cliente),
  CONSTRAINT CK_cliente_estado_civil
    CHECK (estado_civil IN ('solteiro','casado','viúvo')),
  CONSTRAINT CK_cliente_idade
    CHECK (idade_cliente BETWEEN 1 AND 250);
```

6.5.2.6. DEFAULT SPECIFICATION

Podemos atribuir valores default para colunas, visando facilitar a inserção de dados

Exemplo:

```
CREATE TABLE cliente (
  cpf_cliente VARCHAR(11),
  nome_cliente VARCHAR(20) CONSTRAINT NN_cliente_nome NOT NULL,
  rua_cliente VARCHAR(30),
  cidade_cliente VARCHAR(40),
  estado_civil VARCHAR(20) DEFAULT 'solteiro',
```

```
  rg_cliente VARCHAR(7),
  data_cadastro DATE DEFAULT SYSDATE,
  idade_cliente INT,
  CONSTRAINT PK_cliente PRIMARY KEY(cpf_cliente),
  CONSTRAINT UK_cliente_rg UNIQUE (rg_cliente),
  CONSTRAINT CK_cliente_estado_civil
       CHECK (estado_civil IN ('solteiro','casado','viúvo')),
  CONSTRAINT CK_cliente_idade CHECK (idade_cliente BETWEEN 1 AND 250));
```

6.5.3. Padronização Nomes de Constraints

Para facilitar a identificação da origem de uma constraint é interessante padronizar seus nomes. Abaixo uma sugestão para os mesmo:

Primary Key

```
  PK_<Nome da Tabela>
  Ex.: PK_CLIENTE, PK_PRODUTO, PK_PEDIDO
```

Foreign Key

```
  FK_<Tabela_Origem>_<Tabela_Destino>
  Ex.: FK_PEDIDO_CLIENTE, PK_ITEM_PRODUTO
```

Check

```
  CK_<Nome_Tabela>_<Nome_Atributo>
  Ex.: CK_CLIENTE_IDADE, CK_CLIENTE_SALARIO
```

Not Null

```
  NN_<Nome_Tabela>_<Nome_Atributo>
  Ex.: NN_CLIENTE_NOME, CK_CLIENTE_CPF
```

Unique

```
  UK_<Nome_Tabela>_<Nome_Atributo>
  Ex.: UK_CLIENTE_CPF, UK_CLIENTE_RG, UK_USUARIO_LOGIN
```

6.5.4. Deleção em Cascata

Opção a ser utilizada quando da definição do constraint foreign key, para que quando deletamos registros da tabela pai os registros da tabela filho sejam automaticamente deletados.

Exemplo:

```
CREATE TABLE depositante(
     ..............................
     nome_cliente VARCHAR(20) CONSTRAINT fk_depositante_cliente
          REFERENCES cliente ON DELETE CASCADE
     ........................);
```

6.5.5. Atualização em Cascata[7]

Opção a ser utilizada quando da definição do constraint foreign key, para que quando atualizados registros da tabela pai os registros da tabela filho sejam automaticamente atualizados. Pode ser utilizado junto com deleção em cascata.

Exemplo:

```
CREATE TABLE depositante(
     ..............................
     nome_cliente VARCHAR(20) CONSTRAINT fk_depositante_cliente
          REFERENCES cliente ON DELETE CASCADE ON UPDATE CASCADE
```

[7] Não está disponível no oracle.

```
     ........................);
```

6.5.6. Adicionando uma Foreign Key para a própria tabela

Se você tiver um relacionamento recursivo onde existe um atributo que é uma chave estrangeira relacionado com a própria tabela não existe forma de criar a constraint da chave estrangeira ao mesmo tempo em que cria a tabela. É necessário primeiro criar a tabela com a chave primaria e depois adicionar a chave estrangeira.

Exemplo de uma tabela com relacionamento recursivo.

```
CREATE TABLE Funcionario(
     rg_funcionario INT ,
     nome_funcionario VARCHAR(20),
     cpf_funcionario VARCHAR(11),
     salario_funcionario NUMERIC(9,2),
     rg_supervisor INT,
     CONSTRAINT pk_funcionario PRIMARY KEY(rg_funcionario));

ALTER TABLE Funcionario add CONSTRAINT Fk_funcionario
     FOREIGN KEY (rg_supervisor) REFERENCES Funcionario;
```

6.5.7. Adicionar Constraints

Permite adicionar uma constraint a uma tabela.

```
ALTER TABLE <Nome_Tabela> ADD CONSTRAINT <Nome_Constraint> <Especificacao>;
```

Onde <Nome_Tabela> é a tabela na qual onde se deseja adicionar a constraint, <Nome_Constraint> o nome da constraint e <Especificacao> a definição da constraint.

Exemplo:

```
ALTER TABLE Funcionario ADD CONSTRAINT uk_cpf Unique(cpf_funcionario);
```

6.5.8. Excluir Constraint

Permite excluir uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> DROP CONSTRAINT <Nome_Constraint>;
```

Onde <Nome_Tabela> é a tabela que possui a constraint e <Nome_Constraint> o nome da constraint se deseja excluir.

Exemplo:

```
ALTER TABLE Funcionario DROP CONSTRAINT uk_cpf;
```

6.5.9. Renomear Constraint

Permite renomear uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> RENAME CONSTRAINT <Nome_Antigo> TO <Nome_Novo>;
```

Onde <Nome_Tabela> é a tabela que possui a constraint que deseja renomear e <Nome_Antigo> o nome da constraint se deseja renomear e <Nome_Novo> o novo nome da constraint.

Exemplo:

```
ALTER TABLE Funcionario RENAME CONSTRAINT SYS_C000000 TO PK_Funcionario;
```

6.5.10. Ativar Constraints

Permite ativar uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> ENABLE CONSTRAINT <Nome_Constraint>;
```

Onde <Nome_Tabela> é a tabela que possui a constraint e <Nome_Constraint> o nome da constraint se deseja ativar.

Exemplo:

```
ALTER TABLE Funcionario ENABLE CONSTRAINT uk_cpf;
```

6.5.11. Desativar Constraints

Permite desativar uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> DISABLE CONSTRAINT <Nome_Constraint>;
```

Onde <Nome_Tabela> é a tabela que possui a constraint e <Nome_Constraint> o nome da constraint se deseja desativar.

Exemplo:

```
ALTER TABLE Funcionario DISABLE CONSTRAINT uk_cpf;
```

6.6. Dicionário de Dados

O Dicionário de dados não é refletido somente pelos domínios dos atributos em um SGBD. Os comentários realizados no dicionário de dados para as tabelas e atributos podem ser armazenados no banco de dados nas suas respectivas tabelas e atributos.

Os comentários criados de tabelas e colunas são armazenados no dicionário de dados do Oracle em:

- ALL_COL_COMMENTS
- **USER_COL_COMMENTS**
- ALL_TAB_COMMENTS
- **USER_TAB_COMMENTS**

Utilize ALL para ver os comentários de todas as tabelas e USER para os comentários criados somente pelo usuário em suas tabelas e atributos.

6.6.1. Comentando Tabelas

Para adicionar um comentário a uma tabela.

```
COMMENT ON TABLE <Nome_Tabela> IS '<Comentario>';
```

Onde <Nome_Tabela> é a tabela que irá receber o comentário <Comentario>.

Exemplo:

```
COMMENT ON TABLE Funcionario IS 'Tabela que armazena os dados do funcionario';
```

6.6.2. Visualizando o comentário de Tabelas

Para visualizar o comentário de uma tabela.

```
SELECT COMMENTS FROM USER_TAB_COMMENTS WHERE TABLE_NAME = '<Nome_Tabela>';
```

Onde <Nome_Tabela> é a tabela que será consultada.

Exemplo:

```
SELECT COMMENTS
```

```
FROM USER_TAB_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário e sua tabela:

```
SELECT TABLE_NAME, COMMENTS
FROM USER_TAB_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário de todas as tabelas:

```
SELECT TABLE_NAME, COMMENTS
FROM USER_TAB_COMMENTS;
```

6.6.3. Comentando Atributos

Para adicionar um comentário a um atributo de uma tabela.

```
COMMENT ON COLUMN <Nome_Tabela>.<Nome_Atributo> IS '<Comentario>';
```

Onde <Nome_Tabela> e a tabela que possui o atributo <Nome_Atributo> que irá receber o comentário <Comentario>.

Exemplo:

```
COMMENT ON COLUMN Funcionario.RG_FUNCIONARIO IS 'Campo chave primaria da tabela funcionario';
```

6.6.4. Visualizando o comentário de Atributos

Para visualizar os comentários de atributos de uma tabela.

```
SELECT COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = '<Nome_Tabela>';
```

Onde <Nome_Tabela> é a tabela que será consultada.

Exemplo:

```
SELECT COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário e o atributo e sua tabela:

```
SELECT TABLE_NAME, COLUMN_NAME, COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

7. SQL DML

7.1. Linguagem de Manipulação de Dados

Uma vez que o esquema esteja compilado e o banco de dados esteja populado, usa-se uma linguagem para fazer a manipulação dos dados, a DML (Data Manipulation Language - Linguagem de Manipulação de Dados).

Por manipulação entendemos:

- A recuperação das informações armazenadas no banco de dados;
- Inserção de novas informações no banco de dados;
- A remoção das informações no banco de dados;
- A modificação das informações no banco de dados

7.2. Esquema Base

Para DML iremos considerar uma empresa na área bancária que possui as seguintes relações:

```
Cidade = (nome_cidade, estado_cidade)
Agencia = (nome_agencia, cidade_agencia, fundos)
Cliente = (nome_cliente, rua_cliente, cidade_cliente, salario_cliente)
Emprestimo = (numero_emprestimo, nome_agencia,total)
Devedor = (nome_cliente, numero_emprestimo)
Conta = (numero_conta, nome_agencia, saldo, rg_funcionario_gerente)
Depositante = (nome_cliente, numero_conta)
Departamento=  (codigo_departamento,  nome_departamento,  local_departamento,
total_salario)
Funcionario    =    (rg_funcionario,    nome_funcionario,    cpf_funcionario,
salario_funcionario, rg_supervisor, codigo_departamento)
```

7.3. Estruturas Básicas

A estrutura básica consiste de três cláusulas: select, from e where.

Select – corresponde a operação de projeção da Álgebra relacional. Ela é usada para relacionar os atributos desejados no resultado de uma consulta.

From – Corresponde a operação de produto cartesiano da Álgebra Relacional. Ela associa as relações que serão pesquisadas durante a evolução de uma expressão.

Where – Corresponde a seleção do predicado da Álgebra Relacional. Ela consiste em um predicado envolvendo atributos da relação que aparece na cláusula from.

Uma consulta típica em SQL tem a seguinte forma:

```
SELECT A1, A2, .., An
FROM R1, R2, ..., Rn
WHERE P;
```

Cada Ai representa um atributo em cara Ri uma relação. P é um predicado. A consulta é equivalente à seguinte expressão em álgebra relacional:

$\pi_{A1, A2, ..,An} (\sigma_{p} (R1 \times R2 \times ... Rn))$

7.3.1. Cláusula Select

Exemplo: Encontre os nomes de todas as agências da relação empréstimo.

```
SELECT nome_agencia
```

```
FROM emprestimo;
```

No casos em que desejamos forçar a eliminação da duplicidade, podemos inserir a palavra chave distinct depois de select.

```
SELECT distinct nome_agencia
FROM emprestimo;
```

No SQL permite usar a palavra chave all para especificar explecitamente que as duplicidades não serão eliminadas:

O asterisco '*' pode ser usado para denotar 'todos os atributos', assim para exibir todos os atributos de empréstimo:

```
SELECT *
FROM emprestimo;
```

A cláusula SELECT pode conter expressões aritméticas envolvendo os operadores +, -, / e * e operandos constantes ou atributos das tuplas:

```
SELECT nome_agencia, numero_emprestimo, total * 100
FROM emprestimo;
```

7.3.2. A cláusula where

Considere a consulta 'encontre todos os números de empréstimos feitos na agência 'Central' com totais emprestados acima de 1200 dólares'. Esta consulta pode ser escrita em SQL como:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE nome_agencia = 'Central' AND total > 1200;
```

A SQL usa conectores lógicos **and**, **or** e **not** na cláusula where. Os operandos dos conectivos lógicos podem ser expressões envolvendo operadores de comparação **<, <=, >, >=,** = e !=. Para a diferença pode ser utilizando ainda o operador <>.

A SQL possui o operador de comparação **between** para simplificar a cláusula where que especifica que um valor pode ser menor ou igual a algum valor e maior ou igual a algum outro valor. Se desejarmos encontrar os números de empréstimos cujos montantes estejam entre 90 mil dólares e 100 mil dólares, podemos usar a comparação **between** escrevendo:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total BETWEEN 90000 AND 100000;
```

em vez de

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total >=90000 AND total <= 100000;
```

Similarmente podemos usar o operador de comparação **not between.**

7.3.3. A cláusula from

A cláusula from por si só define um produto cartesiano das relações da cláusula. Uma vez que a junção natural é definida em termos de produto cartesiano, uma seleção é uma projeção é um meio relativamente simples de escrever a expressão SQL para uma junção natural.

Escrevemos a expressão em álgebra relacional:

$$\pi_{\text{nome\_cliente, numero\_emprestimo}} (\text{devedor} \bowtie \text{emprestimo})$$

ou com produto cartesiano

```
                (σ                                (devedor X emprestimo)))
(π                devedor.numero_emprestimo=emprestimo.numero_emprestimo
  nome_cliente, numero_emprestimo
```

para a consulta ‘para todos os clientes que tenham um empréstimo em um banco, encontre seus nomes e números de empréstimos’. Em SQL, essa consulta pode ser escrita como:

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

Note que a SQL usa a notação nome_relação.nome_atributo, como na álgebra relacional para evitar ambigüidades nos casos em que um atributo aparecer no esquema mais de uma relação.

O SQL incorpora extensões para realizar junções naturais e junções externas na clausula from.

7.3.4. A operação Rename

A SQL proporciona um mecanismo para rebatizar tanto relações quanto atributos, usando a cláusula **as**, da seguinte forma:

```
nome_antigo AS nome_novo
```

Considere novamente a consulta usada anteriormente:

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

Por exemplo se desejarmos que o nome do atributo nome_cliente seja substituido pelo nome nome_devedor, podemos reescrever a consulta como:

```
SELECT distinct nome_cliente AS nome_devedor, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

A cláusula **as** pode aparecer tanto na cláusula **select** quanto na cláusula **from**.

7.3.5. Variáveis Tuplas

Variáveis tuplas são definidas na cláusula **from** por meio do uso da cláusula **as**. Para ilustrar, reescrevemos a consulta ‘para todos os funcionários encontre o nome do seu supervisor’, assim:

```
SELECT distinct F.nome_funcionario, S.nome_funcionario
FROM funcionario AS F, funcionario AS S
WHERE F.rg_funcionario = S.rg_supervisor;
```

Variáveis tuplas são úteis para comparação de duas tuplas de mesma relação.

7.3.6. Operações em Strings

As operações em strings mais usadas são as checagens para verificação de coincidências de pares, usando o operador **like**. Indicaremos esses pares por meio do uso de dois caracteres especiais:

Porcentagem (%) : o caracter % compara qualquer substring.

Sublinhado (_) : o caracter _ compara qualquer caracter.

Comparações desse tipo são sensíveis ao tamanho das letras; isto é, minúsculas não são iguais a maiúsculas, e vice-versa. Para ilustrar considere os seguintes exemplos;

'**Pedro%**' corresponde a qualquer string que comece com ‘Pedro’

'**%inh%**' corresponde a qualquer string que possua uma substring ‘inh’, por exemplo ‘huguinho’, ‘zezinho’ e ‘luizinho’

'_ _ _' corresponde a qualquer string com exatamente três caracteres

'_ _ _**%**' corresponde a qualquer string com pelo menos três caracteres

Pares são expressões em SQL usando o operador de comparação **like**. Considere a consulta ‘encontre os nomes de todos os clientes possuam a substring ‘Silva’ ‘. Esta consulta pode ser escrita assim:

```
SELECT nome_cliente
FROM cliente
WHERE nome_cliente LIKE '%Silva%';
```

A SQL permite-nos pesquisar diferenças em vez de coincidências, por meio do uso do operador de comparação **not like**.

7.3.7. Precedência dos Operadores

Como qualquer linguagem existe uma ordem na precedência dos operadores:

Ordem de Avaliação	Operadores
1	Operadores Aritméticos
2	Operador de Concatenação
3	Condições de Comparação
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Condição lógica NOT
7	Condição lógica AND
8	Condição lógica OR

Para modificar a ordem de avaliação utilize parênteses.

7.3.8. Ordenação e Apresentação de Tuplas

A SQL oferece ao usuário algum controle sobre a ordenação por meio da qual as tuplas de uma relação serão apresentadas. A cláusula order by faz com que as tuplas do resultado de uma consulta apareçam em uma determinada ordem.

Para listar em ordem alfabética todos os clientes que tenham um empréstimo na agência Centro, escrevemos:

```
SELECT distinct nome_cliente
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo
AND nome_agencia = 'Centro'
ORDER BY nome_cliente;
```

Por default, a cláusula **order by** relaciona os itens em ordem ascendente. Para a especificação da forma de ordenação, devemos indicar **desc** para ordem descendente e **asc** para ordem ascedente. A ordenação pode ser realizada por diversos atributos.

Suponha que desejamos listar a relação cliente inteira por ordem ascendente de nome de cidade. Se diversos cliente residirem na mesma cidade, queremos que seja realizada uma segunda ordenação descendente pelo nome do cliente. Expressamos essa consulta da seguinte forma:

```
SELECT cidade_cliente, nome_cliente
FROM cliente
ORDER BY cidade_cliente ASC, nome_cliente DESC;
```

Para completar uma solicitação de order by, a SQL precisa realizar uma sort (intercalação). Uma vez que a intercalação de um grande número de tuplas pode ser custosa, é desejável usá-la somente quando realmente necessário, ou utilizar em um campo indexado.

7.4. Composição de Relações

Além de fornecer o mecanismo básico do produto cartesiano para a composição das tuplas de duas relações o SQL oferece diversos outros mecanismos para composição de relações como as junções condicionais e as junções naturais, assim como várias formas de junções externas. Essas operações adicionais são usadas tipicamente como expressões de subconsultas na clausula from.

Um produto cartesiano é formado nas seguintes condições:

- Uma condição de junção(join) é omitida
- Uma condição de junção(join) é inválida
- Todas as linhas da primeira tabela são multiplicadas com todas as linhas da segunda tabela

Para impedir a formação de um produto cartesiano, sempre inclua uma condição de restrição válida na cláusula WHERE.

7.4.1. Tipos de Junções e Condições

Operações de junção tomam duas relações e têm como resultado outra relação. Embora as expressões para junção externa sejam normalmente usadas na cláusula from, elas podem ser usadas em qualquer lugar onde se usa uma relação.

As junções podem ser de quatro tipos:

- EquiJoin = Junção Interna
- Non-EquiJoin = Não Junção Interna
- OuterJoin = Junção Externa
- SelfJoin = Auto Junção(interna)

Cada uma das variantes das operações de junção em SQL consiste em um tipo de junção e em uma condição de junção. As condições de junção definem quais tuplas das duas relações apresentam correspondência e quais atributos são apresentados no resultado de uma junção. O tipo de junção define como as tuplas em cada relação que não possuam nenhuma correspondência (baseado na condição de junção) com as tuplas da outra relação devem ser tratadas. Abaixo estão alguns dos tipos de junções permitidos e as condições de junção. O primeiro tipo é a junção interna (INNER JOIN) e os outros tipos são de junções externas(OUTER JOIN). As três condições são NATURAL, ON e USING.

Claúsulas de Junção
Inner join => join
Left outer join => left join
Right outer join => right join
Full outer join => full join

Condições de junção
Natural
On <predicado>
Using (A1, A2, ..., An)

O uso de uma condição de junção é obrigatório para junções externas, mas opcional para junções internas (se for omitido, o resultado será um produto cartesiano). Sintaticamente a palavra-chave NATURAL aparece antes do tipo da junção, e as condições ON e USING apareçam no final de uma expressão de junção. As palavras chaves INNER e OUTER são opcionais, uma vez que os nomes dos demais tipos de junções nos permitem deduzir se trata de uma junção interna ou externa.

O significado da condição NATURAL, em termos de correspondência de tuplas das duas relações é bastante rígido. A ordem dos atributos no resultado de uma junção natural é a seguinte. Os atributos da junção (isto é, os atributos comuns a ambas as relações) aparecem primeiro, conforme a ordem em que aparecem na relação do lado esquerdo. Depois vêm todos os atributos para os quais não há correspondência aos do lado esquerdo e, finalmente, todos os atributos sem correspondência aos da relação do lado direito.

O tipo de junção RIGHT OUTER JOIN e simétrico a LEFT OUTER JOIN. As tuplas da relação do lado direito que não correspondem a nenhuma tupla da relação do lado esquerdo são preenchidos com nulos e adicionados ao resultado da junção externa direita.

A expressão seguinte é um exemplo da combinação dos tipos de junção NATURAL e do RIGHT OUTER JOIN.

```
Emprestimo NATURAL RIGHT OUTER JOIN devedor
```

Os atributos do resultado são definidos por um tipo de junção, que é uma junção natural; então numero_emprestimo aparece somente uma vez.

A condição de junção USING(A1, A2, ..., An) é similar à condição de junção natural exceto pelo fato de que seus atributos de junção são os atributos A1, A2, ..., An, em vez de todos os atributos comuns a ambas as relações. Os atributos A1, A2, ..., An devem ser somente os atributos comuns a ambas as relações e eles aparecem apenas uma vez no resultado da junção.

O tipo FULL OUTER JOIN é uma combinação dos tipos de junções externas à esquerda e à direita. Depois que o resultado de uma junção interna é processado, as tuplas da relação do lado esquerda que não correspondem a nenhuma das tuplas do lado direito são preenchidas com valores nulos, e depois adicionados ao resultado. Similarmente, as tuplas da relação do lado direito que não coincidem com nenhuma das tuplas da relação do lado esquerdo são também preenchidas com nulos e adicionados ao resultado.

Por exemplo, o resultado da expressão:

```
Emprestimo FULL OUTER JOIN devedor USING(numero_empresitmo)
```

é mostrado abaixo.

nome_agencia	numero_emprestimo	total	nome_cliente
…	…	…	…

7.4.2. Junção Interna

Junção Interna(Inner Join) é quando tuplas são incluídas no resultado de uma consulta somente se existir uma correspondente na outra relação.

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 INNER JOIN tabela2 ON tabela1.atributo1 = tabela2.atributo1;
```

Se o nome dos atributos for igual a cláusula ON e desnecessária, para isto usa-se o a Junção Natural com cláusula NATURAL.

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 NATURAL INNER JOIN tabela2;
```

A cláusula INNER é opcional, uma vez que os nomes dos demais tipos de junções nos permite deduzir quando se trata de outro tipo junção. Sendo assim consulta anterior pode ser rescrita desta forma:

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 NATURAL JOIN tabela2;
```

7.4.3. Junção de Mais de duas tabelas

É possível realizar a junção de mais de duas tabelas utilize parentes para separar cada par de tabelas unidas.

```
SELECT tabela1.atributo1, tabela2.atributo2, tabela3.atributo3
FROM (tabela1 INNER JOIN tabela2
            ON tabela1.atributo1 = tabela2.atributo1) INNER JOIN tabela3
                        ON tabela2.atributo3 = tabela3.atributo3;
```

Ou com produto cartesiano:

```
SELECT tabela1.atributo1, tabela2.atributo2, tabela3.atributo3
FROM tabela1, tabela2, tabela3
WHERE tabela1.atributo1 = tabela2.atributo1
     AND tabela2.atributo3 = tabela3.atributo3;
```

7.4.4. Junção Externa

Junção externa é quando tuplas são incluídas no resultado sem que exista uma tupla correspondente na outra relação. Utiliza-se junção externa para listar tuplas que não usualmente se reúnem numa condição join.

Podem ser de três tipos:

RIGHT OUTER JOIN = Junção Externa a Direita
LEFT OUTER JOIN = Junção Externa a Esquerda
FULL OUTER JOIN = Junção Externa Total (Junção Externa a Esquerda + Junção Externa a Direita)

O operador de junção externa é o sinal de adição (+)

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE  R1.A1(+) = R2.A1;

ou

SELECT R1.A1, R2.A2
FROM R1, R2
WHERE  R1.A1 = R2.A2(+);
```

ou OUTER JOIN tanto para a esquerda(LEFT) como a direita(RIGHT)

```
SELECT R1.A1, R2.A2
FROM R1 RIGHT OUTER JOIN R2 ON R1.A1= R2.A2;

ou

SELECT R1.A1, R2.A2
FROM R1 LEFT OUTER JOIN R2 ON R1.A1 = R2.A2;
```

7.4.5. Auto Junção

Se for necessário realizar a junção da tabela com ela mesma, obriga-se renomear uma das tabelas ou as duas para não ocorrer ambigüidade de nomes.

```
SELECT tabela1.atributo1, tabela1.atributo2
FROM tabela1 t1 INNER JOIN tabela1 t2 ON t1.atributo1 = t2.atributo2;
```

ou com produto cartesiano:

```
SELECT tabela1.atributo1, tabela1.atributo2
FROM tabela1 t1, tabela1 t2
WHERE t1.atributo1 = t2.atributo2
```

7.5. Modificações no Banco de Dados

7.5.1. Remoção

Um pedido para remoção de dados é expresso muitas vezes do mesmo modo que uma consulta. Podemos remover somente tuplas inteiras; não podemos excluir valores de um atributo em particular. Em SQL, a remoção é expressa por:

```
DELETE FROM r
WHERE P;
```

em que **P** representa um predicado e r, uma relação. O comando **delete** encontra primeiro todas as tuplas **t** em **r** para as quais **P(t)** é verdadeira e então remove-las de **r**. A cláusula **where** pode ser omitida nos casos de remoção de todas as de **r**.

O comando **delete** opera somente uma relação.

Exemplos:

Remova todos os registros de contas do cliente Smith.

```
DELETE FROM depositante
WHERE nome_cliente = 'JOAO';
```

Remova todos os empréstimos com total entre 1.300 e 1.500 dólares.

```
DELETE FROM emprestimo
WHERE total BETWEEN 1300 AND 15000;
```

7.5.2. Inserção

Para inserir dados em relação podemos especificar uma tupla a ser inserida ou escrever uma consulta cujo resultado é um conjunto de tuplas a inserir. A inserção é expressa por:

Remova todos os registros de contas do cliente Smith.

```
INSERT into r(A1, A2,..., AN)
VALUES (V1, V2,...,VN);
```

Em que **r** é a relação **Ai** representa os atributos as serem inseridos e **Vi** os valores contidos nos atributos **Ai**.

Para inserir a informação que existe uma conta 25 na agência de Agencia13 e que ela tem um saldo de 1.99 dólares. Escrevemos:

```
INSERT into conta
VALUES(25,'AGENCIA13',1.99);
```

O seguinte comando é idêntico ao apresentado anteriormente:

```
INSERT into conta(NUMERO_CONTA,NOME_AGENCIA,SALDO)
VALUES(25,'AGENCIA13',1.99);
```

Na inserção também podemos utilizar a palavra-chave null para que se construir uma única expressão de insert.

No exemplo a seguir somente o nome do cliente é inserido.

```
INSERT INTO
       cliente(nome_cliente, rua_cliente, cidade_cliente, salario_cliente)
Values (‘João da Silva’,null,null,null);
```

Para preencher os outros atributos de cliente basta trocar null pelo valor desejado.

7.5.3. Atualização

Em determinadas situações, podemos querer modificar valores das tuplas sem, no entanto alterar todos os valores. Para esse fim, o comando **update** pode ser usado.

```
UPDATE r
SET A1 = V1, A2 = V2, ... NA = VN
WHERE p;
```

em r é a relação, Ai representa o atributo a ser atualizado e Vi o valor a ser atualizado no atributo Ai, e P um predicado.

Suponha que pagamento da taxa de juros anual esteja sendo efetuado e todos os saldos deverão ser atualizados de 5 por cento. Escrevemos:

```
UPDATE conta
SET saldo = saldo * 1,05;
```

O comando anterior é aplicado uma vez para cada tupla de conta.

Suponha agora que constas com saldo superior a 10 mil dólares recebam 6 por cento e juros, embora todas as outras recebam 5 por cento. Escrevemos

```
UPDATE conta
SET saldo = saldo * 1,06
WHERE saldo > 10000;
```

7.6. Transação

Transação é uma unidade atômica de trabalho que atua sobre um banco de dados. Uma transação pode ser constituída por uma ou mais operações de acesso à base de dados. Todas as operações devem ser bem-sucedidas, caso contrário os efeitos da transação devem ser revertidos.

Uma transação inicia com o primeiro comando SQL emitido para a base de dados e finaliza com os seguintes eventos:

- COMMIT ou ROLLBACK

- Comandos DDL executam commit automático
- Saída do Usuário
- Queda do Sistema

7.6.1. Commit

Uma transação bem-sucedida termina quando um comando **COMMIT** é executado. O comando **COMMIT** finaliza e efetiva todas as alterações feitas na base de dados durante a transação.

Grava uma transação no banco de dados.

Sintaxe:

```
COMMIT;
```

Exemplo:

Alteração de dados

```
SQL> UPDATE funcionario
2  SET codigo_departamento = 3
3  WHERE  rg_funcionario = '1112';
1 linha atualizada.
```

Commit nos dados

```
SQL> COMMIT;
Commit completado.
```

7.6.2 RollBack

Se uma transação aborta antes de o comando **COMMIT** ser executado, a transação deve ser desfeita, isto é, todas as mudanças feitas durante a transação devem ser desconsideradas. O processo de recuperação automática que permite desfazer as alterações feitas contra a base é chamado **ROLLBACK**. O **ROLLBACK** retorna a situação dos objetos da base alterados na transação à mesma situação em que se encontravam no início da transação.

O **ROLLBACK** reverte os efeitos de uma transação como se ela nunca tivesse existido.

Restaura uma transação. Recupera o banco de dados para a posição que esta após o último comando **COMMIT** ser executado.

Sintaxe:

```
ROLLBACK;
```

Exemplo:

```
SQL> DELETE FROM funcionario;
14 linhas apagadas.

SQL> ROLLBACK;
Rollback completado.
```

8. Bibliografia

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML.** 1. ed. Rio de Janeiro: Campus, 2002. 286 p.

CHEN, P. **The Entity-Relationship Model - Toward a Unified View of Data**. TODS, 1976.

CHIAVENATO, I. **Introdução à Teoria Geral da Administração.** 7. ed. Rio de Janeiro: Campus, 2004. 634 p.

DATE, C. J. **Introdução a Sistemas de Banco de Dados.** 7. ed. Rio de Janeiro: Campus, 2000. 803 p.

DAUM, B. **Modelagem de Objetos de negócio com XML:** abordagem com base em XML Schema. Rio de Janeiro: Elsevier, 2004.

ELMASRI, S. N.; NAVATHE, B.S.. **Sistemas de Banco de Dados:** Fundamentos e Aplicações. 3. ed. Rio de Janeiro: Livros Técnicos e Científicos, 2002. 837 p.

KROENKE, D.M. **Banco de Dados:** Fundamentos, Projeto e Implementações. 6. ed. Rio de Janeiro: Livros Técnicos e Científicos, 1998. 382 p.

NETO, P. C., INFANTE, U. **Gramática da língua portuguesa.** São Paulo: Scipione, 1988.

OZSU, M.T.; VALDURIEZ, P. **Princípios de sistemas de banco de dados distribuídos.** 1. ed. Rio de Janeiro: Campus, 2001.

SILBERSCHATZ, A. ; KORTH, H.F. ; SUDARSHAN, S. **Sistema de Banco de Dados. 5**. ed. Rio de Janeiro: Elsevier, 2006.

YOURDON, E. **Análise Estrutura Moderna.** 3. ed. Rio de Janeiro: Campus, 1992. 836 p.

Apêndice A - Exemplo de um Banco de Dados

Relação EMPREGADO					
EmpregadoId	Nome	CPF	DeptoId	SupervidorId	Salario
10101010	*João Luiz*	*11111111*	*1*	***NULO***	*3.000,00*
20202020	*Fernando*	*22222222*	*2*	*10101010*	*2.500,00*
30303030	*Ricardo*	*33333333*	*2*	*10101010*	*2.300,00*
40404040	*Jorge*	*44444444*	*2*	*20202020*	*4.200,00*
50505050	*Renato*	*55555555*	*3*	*20202020*	*1.300,00*

Relação DEPTO		
DeptoId	Nome	GerenteId
1	*Contabilidade*	*10101010*
2	*Engenharia Civil*	*30303030*
3	*Engenharia Mecânica*	*20202020*

Relação PROJETO		
ProjetoId	Nome	Localizacao
5	*Financeiro 1*	*São Paulo*
10	*Motor 3*	*Rio Claro*
20	*Prédio Central*	*Campinas*

Relação DEPENDENTE				
EmpregadoId	Nome	DtNascimento	Relacao	Sexo
10101010	Jorge	27/12/86	Filho	Masculino
10101010	Luiz	18/11/79	Filho	Masculino
20202020	Fernanda	14/02/69	Conjuge	Feminino
20202020	Angelo	10/02/95	Filho	Masculino
30303030	Adreia	01/05/90	Filho	Feminino

Relação DEPTO_PROJ	
DeptoId	ProjetoId
2	*5*
3	*10*
2	*20*

Relação EMP_PROJ		
EmpregadoId	ProjetoId	Horas
20202020	*5*	*10*
20202020	*10*	*25*
30303030	*5*	*35*
40404040	*20*	*50*
50505050	*20*	*35*