Resumo Teórico // MAC0350

UML

• Significa *Unified Modelling Language*.

• Serve o propósito de visualizar, especificar (ou documentar) e construir software orientado a objetos.

• Facilita a comunicação entre membros da equipe, pois cada símbolo gráfico utilizado tem uma semântica bem definida.

• O UML oferece uma forma padrão de se desenhar as “plantas” (como em arquitetura) de um sistema, permitindo incluir tanto aspectos abstratos (como funcionalidades de um sistema), como aspectos concretos de um sistema (como classes, esquemas de bancos de dados, entre outras coisas).

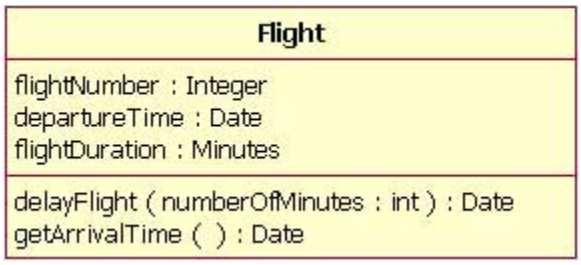
• Existem alguns tipos diferentes de diagramas UML. Os principais tipos são de **estrutura**, e de **comportamento**. Começaremos estudando diagramas de estrutura, que possuem os seguintes subtipos:

- Diagrama de classes;
- Diagrama de pacotes;
- Diagrama de objetos;
- Diagrama de componentes;
- Diagrama de implantação.

• Começemos estudando os diagramas de classes. Estes servem, de forma geral, para modelar o vocabulário do sistema, em termos de quais abstrações fazem parte do sistema, e quais caem fora de seu domínio; modelar as colaborações/interações entre elementos do

sistema; e, por fim, a modelagem lógica dos dados manipulados pelo sistema.

• Os diagramas possuem notação com semântica bem definida. Estudemos os diversos símbolos pertencentes a esta notação.



Esta é a representação de uma classe. Seu nome é dado na primeira divisória, seguido por seus atributos, e finalmente suas operações (ou métodos). Se alguma componente aparece em *ítalico*, esta componente é *abstrata*. Uma quarta divisória, contendo as responsabilidades de uma classe, pode aparecer, mas todas as divisórias com exceção do nome da classe são opcionais.

Com relação aos atributos e métodos, podemos fornecer especificações de acesso com a seguinte notação: + representa *public*, - representa *private*, e # representa *protected*.

Relacionamentos representam conexões entre classes, e podem ter naturezas diferentes:

• Relacionamentos de dependência são representados por segmentos vazados e direcionados, em que uma classe usa outra (a classe que "usa" aponta para a classe que é "usada"). Mudanças na implementação na classe "usada" podem causar efeitos na classe que a usa.

• Relacionamentos do tipo generalização são representados por segmentos cheios e direcionados, em que a classe superior (posicionalmente) é uma generalização das outras ligadas.

• Relacionamentos do tipo associação, são representados por segmentos cheios e não-direcionados, que representam objetos e classes que estão interconectados. Pode também ser chamado de ligação (*link*).

Associações podem ser complementadas com diversos ***ornamentos***

para associações (que aparecem escritos acima dos segmentos). Os ornamentos possuem diversos tipos, que são os seguintes:

• O nome da associação, que descreve a natureza da relação, e pode ser direcionada;

• Os papéis, que são escritos logo ao lado de cada objeto.

• A multiplicidade, que representam relações quantitativas (do tipo *has_many*, entre outras). São representadas logo ao lado dos objetos, numéricamente.

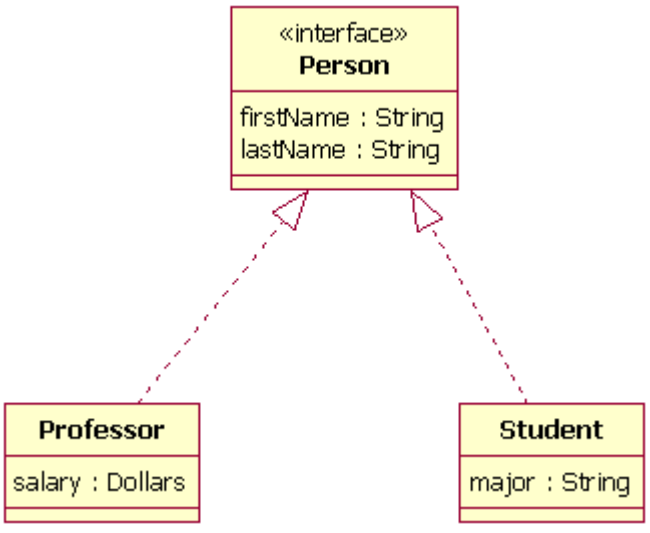
• A agregação, representada por um *diamante vazio*, que representa uma relação "todo/parte" ou "possui um", na qual a classe "maior" tem o diamante conectado diretamente em si.

• A composição, representada por um *diamante cheio*, que é um tipo especial de agregação em que as partes são inseparáveis do todo.

• Existem outros tipos de relações também, como classes de associação, que são classes que possuem as propriedades de classes e de associações.

• Interfaces são coleções de atributos e/ou operações que possuem um nome. São utilizadas para especificar serviços sem ditar suas implementações. Uma interface pode participar de todos os relacionamentos (generalizações, associações e dependências). Uma realização é a relação entre uma interface e a classe que a implementa.

• Além de ornamentos para associações, existem ornamentos genéricos, utilizados para estender a linguagem "oficial". Eles adicionam algum texto ou elemento gráfico ao diagrama, e podem ser dos seguintes tipos:



• Estereótipos, que são uma extensão do vocabulário de UML, que permite a criação de um tipo básico novo (como interface, por exemplo) que é específico ao problema que está sendo resolvido. Existem cerca de 50 estereótipos padrões já estabelecidos, como *become*, *enumeration* ou *utility*.

• Valores rotulados (*tagged values*), que permitem a especificação de propriedades de elementos de um modelo.

• Restrições, que especificam condições que devem ser seguidas pelo sistema.

• Os elementos de um diagrama de pacotes são agrupados dentro de pacotes. Pacotes são nomeados para organizar elementos de um sistema, e cada elemento pode pertencer a um único pacote. Cada elemento dentro de um pacote deve ter nome único.

• Uma derivação dos diagramas de classe são os diagramas de objetos, que representam um conjunto de objetos e suas relações em um determinado instante da execução do sistema.

• Também podemos representar componentes (em um diagrama de componentes), que são partes do sistema que podem ser substituídas e que oferecem uma implementação de um conjunto de interfaces, e nós, que representam elementos físicos capazes de oferecer recursos computacionais.

• Os diagramas mencionados acima são drasticamente diferentes de um diagrama de classes. É pertinente checar as referências abaixo para melhor visualização de seus formatos.

• Outras referências:

http://creately.com/blog/diagrams/uml-diagram-types-examples/

https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html

https://www.tutorialspoint.com/uml/index.htm

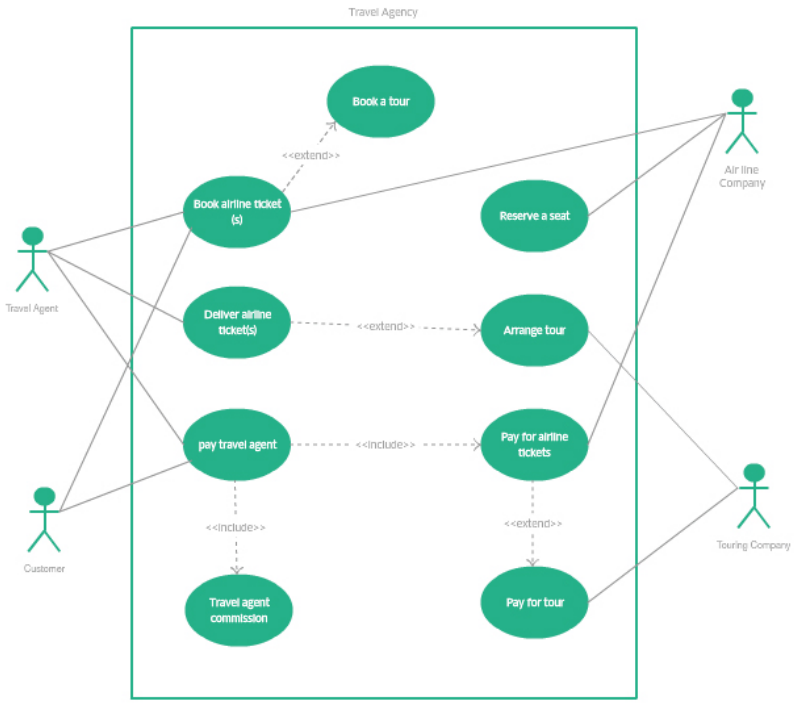
• Agora, passemos a falar sobre diagramas de **comportamento**. Os diagramas de comportamento (ou diagramas comportamentais) são

utilizados principalmente para visualizar, documentar e especificar aspectos dinâmicos de um sistema. Os diagramas de comportamento podem ser dos seguintes tipos:

- Diagramas de casos de uso;
- Diagramas de sequência;
- Diagramas de colaboração;
- Diagrama de estados;
- Diagrama de atividades.

• Comecemos por diagarams de casos de uso. Um caso de uso é uma sequência de ações, incluindo variantes, que um sistema realiza a fim de gerar um resultado observável de interesse para um ator. Um ator é um papel que um usuário desempenha quando participa de um caso de uso. Por exemplo, um ator pode ser um administrador, que compra materiais, gera relatórios de compra e atualiza estoques.
Neste tipo de diagrama, existem fluxos de eventos, que descrevem casos de uso. O fluxo de eventos principal descreve o caso em que corre tudo bem, mas podem existir fluxos de eventos excepcionais, que cobrem as variações que podem ocorrem quando diferentes coisas dão errado, ou quando algo incomum acontece.

•



Para a organização dos diagramas de casos de uso, existem as seguintes componentes adicionais:

- Pacotes;
- Generalização;
- Inclusão;
- Extensão.

• Pacotes são análogos aos pacotes presentes em diagramas estruturais, uma vez que possuem a função de agrupar ações de naturezas similares, ou relacionadas a uma mesma função de um ator.

• As generalizações, também, são análogas às generalizações dos diagramas estruturais de classes.

• As inclusões são dadas pelo estereótipo *«include»*, que indica que um caso inclui o outro, e permite fatorar comportamento comum a vários casos.

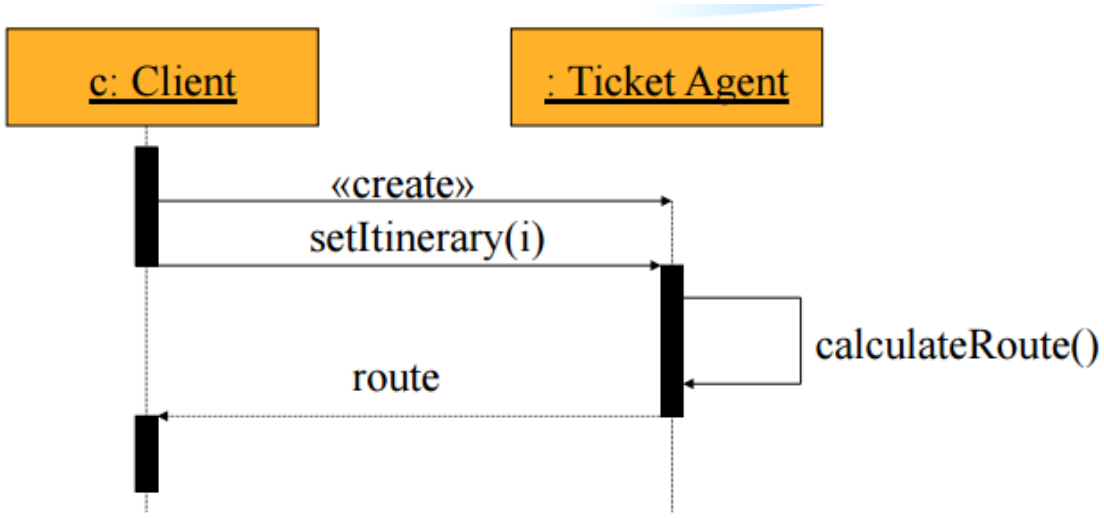
• Similarmente, o estereótipo *«extend»* é utilizado para indicar que um caso estende o outro. É útil para fatorar comportamentos incomuns.

• Por fim, objetos podem enviar interações e mensagens entre si. Uma interação é um comportamento composto da troca de um conjunto de

mensagens entre um grupo de objetos a fim de atingir um determinado objetivo. Uma mensagem, por sua vez, é uma comunicação entre objetos que resulta na transmissão de informação com o intuito de que alguma atividade seja realizada.

• Em seguida, tratemos de diagramas de sequência. O diagrama de sequência é um diagrama de interações, que enfatiza a ordem temporal das mensagens, e tenta retratar o fluxo dos objetos e ações de um sistema. Comecemos explicitando os tipos de símbolos representados neste tipo de diagrama, e seus significados.
Antes de tudo, vale ressaltar que estes diagramas geralmente são lidos de cima para baixo, começando da esquerda para a direita.
Uma linha de vida é uma linha tracejada e vertical, que representa o tempo de vida de um objeto. Um foco de controle é um retângulo fino vertical sobreposto à linha de vida que mostra o período durante o qual um objeto está realizando uma ação.

• Em seguida, tratemos de diagramas de colaboração. Diagramas de



colaboração enfatizam a organização de objetos que participam de uma certa interação. Nestes diagramas, um caminho é uma ligação entre objetos, possivelmente com um estereótipo («local», «global», entre outros). Números de sequência indicam a ordem temporal das mensagens em um ou mais níveis.

• Diagramas de estados, por suas vezes, são diagramas mais simples, que representam os possíveis estados que um sistema pode assumir, e as possíveis transições entre os estados. Tanto os estados quanto as

transições são simplesmente escritas de forma textual, e ligadas por segmentos orientados.

• Por fim, temos os diagramas de atividades, que são um tipo específico de diagrama de estados. São úteis para modelar fluxo de trabalho (*workflow*) e processos de negócios, e representam as atividades que afetam um sistema e os fluxos que levam de uma atividade a outra.

• Novamente, cada tipo de diagrama difere drasticamente em termos visuais um do outro (com exceção dos últimos dois, que são similares entre si), então é pertinente checar as referências anteriormente citadas para ter uma noção melhor de suas aparências.

Testes Automatizados e Test-Driven Development (TDD)

• Com o aumento da complexidade dos sistemas durante os anos, a necessidade da criação de testes automatizados tornou-se cada vez mais evidente, uma vez que, quão mais complexos os sistemas, mais provável é que os seres humanos que os desenvolvem cometam erros.

• Os testes são úteis pois detectam a presença de erros em um sistema, em suas diversas partes. Os testes automatizados são particularmente relevantes neste sentido pois, para sistemas grandes, a execução de testes automatizados para as diversas componentes de um sistema pode acusar erros durante o desenvolvimento de cada componente. Isto é importante, pois o desenvolvimento de uma componente pode afetar (negativamente) outra, e isto pode não ser percebido se não forem feitos testes nas diversas componentes.

• Antigamente, os desenvolvedores terminavam o código, faziam alguns testes não-planejados e específicos (*ad-hoc*), enviavam o produto para o Controle de Qualidade (*Quality Assurance*), e estes eram responsáveis por testar manualmente o software.
Atualmente, testes fazem parte de todas as iterações de métodos ágeis. Os desenvolvedores testam seu próprio código, com o auxílio de ferramentas de teste e processos altamente automatizados. Por outro lado, o Controle de Qualidade e grupos de teste focam seus esforços em melhorar a qualidade das ferramentas de teste em si.

• Neste contexto de testes, existem algumas formas de tratar o

desenvolvimento com testes. Alguns exemplos disso são o *Behavior Driven Development* (*BDD*) e o *Test Driven Development*.

• No *BDD*, os desenvolvedores escrevem *user stories*, que basicamente são descrições de *features* a serem implementadas, com alguns cenários de usos. Então, a partir destas *user stories*, são escritos *acceptance tests*, que são testes (em código) dos cenários mencionados acima.

• Em *TDD*, por outro lado, nós não temos o passo das *user stories*, e escrevemos todos os testes (*unit tests* e *functional tests*) para uma certa parte do código, **antes mesmo de escrever o código**. Ou seja, nós escrevemos testes para o código que eventualmente desejamos ter.

• *Unit Tests* (Testes Unitários) são testes curtos, focados em testar uma única funcionalidade em um único objeto ou módulo, geralmente.

• Os *unit tests* devem ser ***FIRST***, ou seja, **F**ast, **I**ndependent, **R**epeatable, **S**elf-checking e **T**imely.
O conjunto de testes precisa ser rapido, pois eles serão rodados com frequência.
Os testes precisam ser independentes uns dos outros, para que você possa rodar qualquer conjunto de testes a qualquer momento.
É necessário poder rodar os testes incontáveis vezes obtendo resultados iguais, para ajudar a isolar *bugs* e habilitar um processo automatizado.
Os testes precisam checar se eles obtiveram sucesso, ou falharam, sem checagem humana de *output*.
Por fim, os testes precisam ser escritos ou no mesmo momento que o código, ou antes mesmo do código (se for *TDD*, principalmente).

• Em *Test-First Development*, o desenvolvimento de código se inicia pensando em alguma funcionalidade do código. Em seguida, fazemos um teste para esta funcionalidade, que naturalmente falha; e então, escrevemos o código mais simples possível que faça com que o teste passe. Por fim, refatoramos o código, para deixá-lo mais simples e de melhor qualidade. Então, continuamos, pensando na funcionalidade seguinte.
Uma característica importante deste processo é que, no inicio de cada iteração do processo, **o código sempre se encontra funcional**.

• Podemos perceber que *TDD* se utiliza de técnicas similares de *debugging* clássico, mas de forma mais produtiva. Além disso, escrever testes pode parecer consumir mais tempo inicialmente, mas ele economiza tempo no geral (especialmente quando o trabalho é em maior escala, com uma equipe grande de desenvolvedores).

• Note: o termo *seam* (costura) refere-se a um local em que dois lugares distintos de um software se encontram, em que algo pode ser injetado.

• Recomenda-se utilizar *it* como placeholders em testes, e *pending* para denotar testes que sabe-se ser necessários no futuro.

• A utilização de testes automáticos permite que mudanças sejam feitas com maior segurança, aumentando a vida útil do *software*.

• Antigamente, a prática comum era escrever funções de testes para cada módulo do sistema no próprio módulo. O problema com esta prática é que estas funções se misturavam com o código do próprio sistema, afetando a legibilidade do código em sua totalidade. Por isso, surgiram os *frameworks* (ou **arcabouços**) que auxiliam e padronizam a escrita de testes automatizados, facilitando o isolamento do código de teste do código da aplicação.

• Note: *IDE* significa *Integrated Development Environment*.

• *Acceptance tests* podem ser tomados também como uma forma de verificar se o que foi implementado atende corretamente ao que o cliente esperava, ou seja, eles validam o sistema do ponto de vista do cliente. *Acceptance tests* requerem não apenas a chamada de métodos e procedimentos, mas também a simulação das ações de um usuário interagindo com o programa, isto é, um clique do mouse, uma tecla pressionada, uma opção selecionada, entre outras ações. Por isso é fundamental a utilização de arcabouços que consigam abstrair as ações de usuário encapsulando o funcionamento interno das interfaces para facilitar a escrita e manutenção dos testes de aceitação.

• *Acceptance tests* são naturalmente mais complexos que outros tipos de teste (como o *unit test*), pois envolve todas as camadas do sistema, dependendo da modelagem correta da base de dados, do funcionamento correto dos módulos internos do sistema, da integração entre eles e da interação do usuário com a interface.

• Outras referências:

https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/

https://codeutopia.net/blog/2013/07/28/why-use-user-story-based-testing-tools-like-cucumber-instead-of-other-tddbdd-tools/

Beleza de Código

• O código "belo" tem vários benefícios, incluindo tornar trabalhos em equipe mais agradáveis, redução do número de *bugs*, melhor manutenibilidade e maior produtividade. Podemos dizer que um código belo traz, no geral, qualidade para o projeto.

• O termo "beleza" é naturalmente um termo envolto em subjetividade, mas podemos ter uma ideia geral do que se refere, em termos de desenvolvimento de *software*. Em desenvolvimento de *software*, a beleza encontra-se em estruturas coesas e adequadas, e uma estética comum sendo respeitada ao longo de um projeto (entre todos os membros de uma equipe). Tratando-se de código, especificamente, um código belo, segundo Rebecca Wirfs-Brock, possui equilíbrio, eficiência, expressividade e precisão (o código executa exatamente o que se esperava dele).

• Outro conceito que se aproxima do conceito de código belo é o de código limpo. Este também é um conceito abstrato, portanto seguem algumas definições dadas a este conceito: "código limpo é conciso, e faz uma coisa bem, de forma direta"; "código limpo é facilmente legível, e sua simplicidade de leitura faz com que seja pouco provável que se escondam *bugs* dentro dele"; "código limpo é aquele que você olha, e você sabe que não existe nenhuma mudança trivial que possa deixá-lo melhor".

• Ou seja, podemos deduzir que código limpo, em geral, refere-se a um código minimalista, direto, que é facil de ler, e executa bem a tarefa que lhe pertence.

• Algumas práticas podem ajudar o código a ficar mais limpo e belo. Vamos explicitar algumas a seguir.

• Primeiramente, a escolha de nomes expressivos é fundamental, pois torna o código mais fácil de ler, considerando que o leitor não precisará procurar atribuir um significado a uma variável, pois esta

terá um nome que expressará seu significado.
Algumas diretrizes são: utilizar nomes claros e legíveis, e prefira nomes curtos, desde que isto não prejudique a clareza dos nomes; use nomes que possam ser usados em buscar, e use os padrões da linguagem, se houverem (como camelCase, entre outros). Por fim, procure comunicar bem o que você quer dizer, visando não confundir o leitor. Por exemplo, evite utilizar diferentes palavras para um mesmo conceito (*get*, *fetch*, *retrieve*), e evite piadas ou metáforas nos nomes das variáveis.

• Outro aspecto específico, mas igualmente importante, em que reside o potencial de beleza de código, são as **funções**.

• O que a função faz deve ser óbvio para o leitor (pois isto permite entender o problema), e a implementação da solução deve ser simples (pois isto permite entender a solução).

• Para a utilidade da função ser óbvia para o leitor, é ideal que uma função contenha apenas uma abstração (ou seja, ela faz apenas uma e apenas uma coisa), e que sejam evitados efeitos colaterais (por exemplo, uma função deve ou modificar um objeto, ou retornar um valor, e nunca os dois ao mesmo tempo).

• Para o propósito de ter funções como acima, foi criado uma diretriz de criação de código chamada ***DRY*** (***Don't Repeat Yourself***). Código *DRY* implica em código que não se repete. Se um trecho de código aparece em mais de um lugar, há uma abstração implícita; logo, se este trecho se encontra dentro de uma função, a função faz mais de uma coisa, e isto é o oposto do que foi dito anteriormente.
Funções idealmente não conseguem ser divididas em seções, ou seja, não é possível extrair uma sub-função de uma função que esteja escrita de forma ideal.

• “The primary cost of abstraction is indirection. In my experience, you should only abstract when the added abstraction clarifies things more than the added indirection confuses them.” – Bryan Edds

• Para os casos em que a abstração se torna excessiva, uma solução é a **refatoração** de código.

• Agora, em termos de implementação simples de funções, é ideal que as funções sejam pequenas, contenham apenas um nível de abstração, e evitar estruturas aninhadas ("Blocos de if's/while's/else's devem ser

diretos no que fazem (provavelmente, apenas chamar uma função)").

• Por fim, sobre funções, é ideal que seu número de argumentos seja pequeno. Muitos argumentos dificultam testes, e normalmente funções com vários argumentos possuem mais de uma camada de abstração (especialmente se os argumentos forem do tipo "*flag*").

• Por fim, para fechar o assunto de beleza de código, falemos de **comentários**.

• Os comentários, em verdade, podem ser vistos como uma forma de compensar nossa incapacidade de expressão através do código.

• Comentários podem ser ruins, pois comentários nem sempre acompanham a evolução do código, e eles acabam tornando-se falsos. Logo, a verdade encontra-se apenas no código em si.

• Existem alguns tipos de bons comentários, no entanto. Comentários sobre aspectos legais que influenciam o código; comentários informativos sobre algum conhecimento específico envolvido no código (exemplo: *RegExp*); esclarecimento de decisões de design; esclarecimentos sobre bibliotecas de terceiros; comentários que frisam a importância de um determinado elemento; e comentários do tipo *to-do*.

Bancos de Dados

• Começamos, então, nossos estudos de Bancos de Dados. Toda empresa ou negócio que cresce, atualmente, precisa lidar com uma grande quantidade de dados. Os dados precisam ser precisos e atualizados, então o surgimento da necessidade de ferramentas de gerenciamento destes dados complexos foi apenas natural.

• A solução para este problema foi encontrada no conceito de Sistema Gerenciador de Banco de Dados, ou SGBD.

• No início da computação, programas gravavam seus dados em disco, segundo suas próprias estruturas. Se vários programas precisassem compartilhar os dados de um mesmo arquivo, todos estes programas teriam que conhecer e manipular as mesmas estruturas. Havia um

claro problema: era extremamente difícil garantir a unicidade das estruturas de dados entre os diversos programas.

• Para evitar este problema, criou-se um sistema intermediário, que conhece a estrutura de dados dos arquivos, fornece apenas os dados que cada programa precisa, e armazena os dados de cada programa. Com a evolução da complexidade e aumento do uso deste sistema intermediário, a coleção de arquivos que contém os dados pertinentes a um sistema passou a se chamar de Banco de Dados, e o sistema intermediário que o administra passou a se chamar Sistema Gerenciador de Banco de Dados.

• Com o tempo, surgiram padrões para descrever as estruturas de dados: os **modelos de dados**. A descrição de um banco de dados, segundo um modelo de dados, é chamada de meta dados.

• Atualmente, um banco de dados é uma coleção de dados coerente e logicamente relacionados, com algum significado associado, e seu objetivo final é, geralmente, representar algum aspecto do mundo real.

• As grandes vantagens da implementação e uso de bancos de dados são: eliminação de redundâncias, facilidade de manutenção e facilidade e repidez ao realizar consultas.

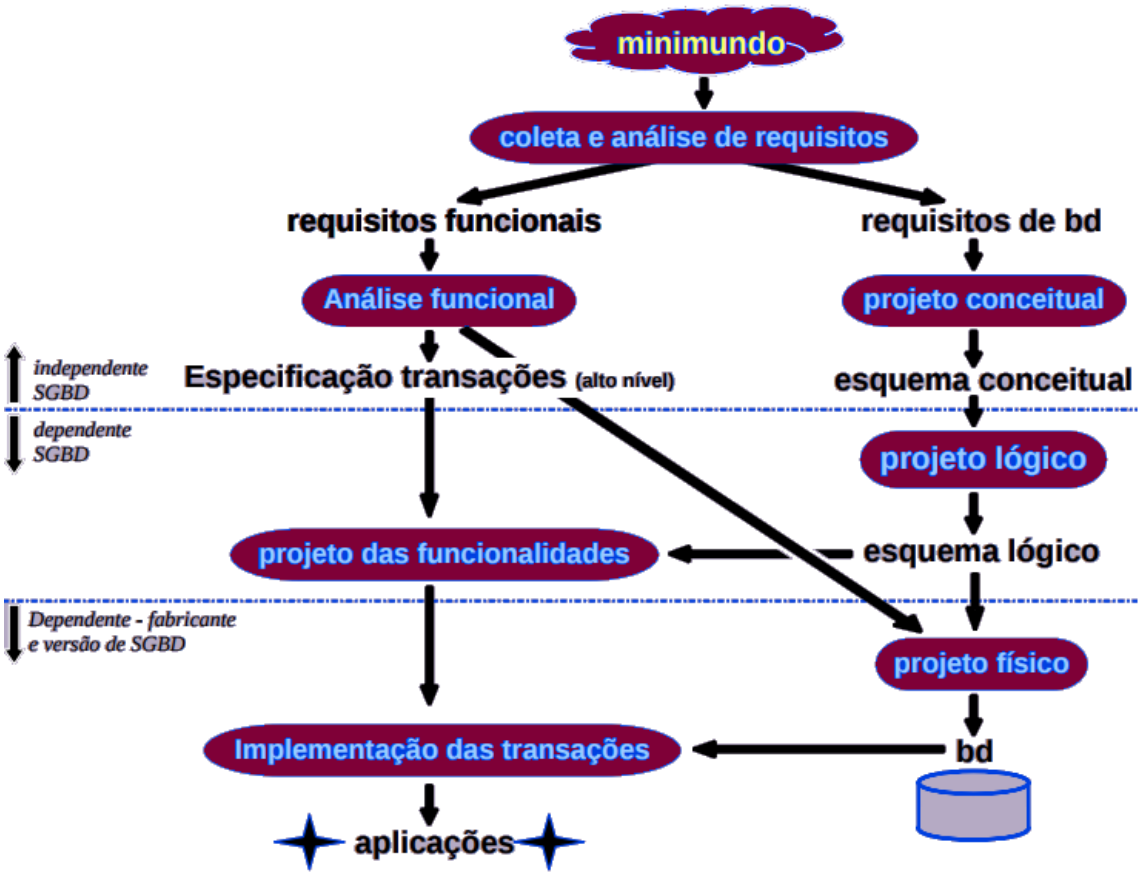
• Atualmente, os únicos dois motivos para não se usar bancos de dados são: se a aplicação é demasiadamente simples e estável, ou se os requisitos de tempo-real não puderem ser atendidos.

• Existem várias formas de implementar bancos de dados. Mais que isso, o banco de dados possui uma camada de abstração que define como dados reais serão representados em nível de máquina.
Uma das formas de encarar esta abstração é o chamado **Three-Schema Approach** (ou Arquitetura Three-Schema). Esta arquitetura define que o sistema de informação (ou banco de dados) possui três níveis diferentes de visão: a visão interna, a conceitual (intermediária) e a externa. A visão interna refere-se à capacidade de representar a estrutura de armazenamento físico dos dados; a visão conceitual refere-se à capacidade de abstração dos dados; e, finalmente, a visão externa refere-se a como os dados são representados nas aplicações (pode ocorrer de múltiplas formas).

• Este tipo de arquitatura geraria uma independência entre os níveis,

ou seja, a existência de um nível conceitual de dados permitiria alterar o esquema conceitual sem ter que fazer alterações nos outros dois esquemas, por exemplo.

• A **modelagem de dados** tornou-se especialmente importante nos dias atuais, pois, como a presença de softwares expandiu-se para as mais diversas áreas de negócios, é necessária uma forma de traduzir requerimentos, expressos por indivíduos não familiares com as estruturas físicas de um banco de dados, para o formato físico de bancos de dados.



• Existem modelos para diferentes níves de abstração de dados. Existem modelos conceituais, lógicos e físicos. Estudaremos, futuramente, alguns dos diferentes modelos de dados que existem. Por ora, detalhemos como os modelos se relacionam com os diferentes níveis de dados.

• É útil observar a seguinte definição, retirada da Wikipedia (https://pt.wikipedia.org/wiki/Modelagem_de_dados), complementada

pelos conhecimentos adquiridos em aula:

Os modelos de dados são ferramentas que permitem demonstrar como serão construídas as estruturas de dados que darão suporte aos processos de negócio, como esses dados estarão organizados e quais os relacionamentos que pretendemos estabelecer entre eles.

A abordagem que se dispensa ao assunto normalmente atende a três perspectivas:

• **Modelagem Conceitual**: é uma representação com alto nível de abstração, e modela da forma mais natural possível os fatos do mundo real, sua propriedades e relacionamentos. Por ser exclusivamente um modelo conceitual, podemos dizer que ele é um modelo intependente do banco de dados. O modelo conceitual é um diagrama em blocos que demonstra todas as relações entre as entidades, suas especializações, seus atributos e auto-relações. Um exemplo deste tipo de modelagem de dados é o modelo entidade-relacionamento;

• **Modelagem Lógica**: agrega mais alguns detalhes de implementação. Também são chamados de modelos de banco de dados. O modelo lógico mostra as ligações entre as tabelas de banco de dados, as chaves primárias, os componentes de cada uma, etc. Um exemplo deste tipo de modelagem de dados é o modelo relacional (**tabelas**). Estes modelos apóiam as especificações dos dados do modelo (ou seja, além dos dados, temos especificados seus domínios e restrições), e a manipulação dos dados do modelo;

• **Modelagem Física**: demonstra como os dados são fisicamente armazenados. Inclui a análise das características e recursos necessários para armazenamento e manipulação das estruturas de dados (estrutura de armazenamento, endereçamento, acesso e alocação física). Um exemplo de modelagem física seria uma sequência de comandos executados em SQL, a fim de criar as tabelas, estruturas e ligações projetadas até então e finalmente criar o banco de dados.

Matriz de Classificação de SGBDs*

• A seguinte síntese de conceitos é pertinentes:

• **Banco de dados (BD)**: conjunto de dados integrados que por objetivo atender a uma comunidade de usuários.

• **Modelo de dados**: descrição formal das estruturas de dados para representação de um BD; com suas respectivas restrições e linguagem para criação e manipulação de dados.

• **Sistema Gerenciador de banco de dados (SGBD)**: software que incorpora as funções de definição, recuperação e alteração de dados em um BD.

• **Modelagem de dados**: é a ação de representar/abstrair dados do minimundo com o objetivo de criar projetos conceituais e lógicos de um BD. Alguns autores incluem os projetos físicos como parte da modelagem de dados, pelo fato de que as otimizações são oriundas de análises do comportamento dinâmico do BD.

• **Projeto conceitual BD**: ação que produz o esquema de dados abstratos que descreve a estrutura de um BD de forma independente de um SGBD (esquema conceitual).

• **Projeto lógico BD**: ação que produz o esquema lógico de dados que representa a estrutura de dados de um BD em acordo com o modelo de dados

subjacente a um SGBD.

• **Projeto físico BD**: ação que produz o esquema físico de dados a partir do esquema de lógico de dados com a adição das estratégias de otimização para manipulação das estruturas de dados. As estratégias de otimização são dependentes dos fabricantes dos SGBDs e de suas versões.

• Comecemos a estudar, então, o **Modelo Entidade-Relacionamento** (**MER**). Este modelo de dados é considerado um modelo de alto nível, criado com o objetivo de representar a semântica associada aos dados do "minimundo". Este modelo é utilizado na fase de projeto conceitual, onde o esquema conceitual do banco de dados da aplicação é concebido.
Seus conceitos são intuitivos, permitindo que projetistas de banco de dado capturem os conceitos associados aos dados da aplicação, sem a interferência da tecnologia específica de implementação do banco de dados.

• O esquema conceitual criado utilizando-se o **MER** é chamado de **Diagrama Entidade-Relacionamento** (**DER**). Perceba que o MER é a parte conceitual de modelagem do processo, que o projetista do banco de dados precisa saber e aplicar, para obter o DER.

• O objeto mais elementar que é possível representar neste tipo de modelagem é chamado de **entidade**. Uma entidade é algo do mundo real, que possui uma existência independente. Cada entidade possui propriedades particulares, que são chamadas de **atributos**. Uma entidade em particular terá um (ou mais) **valor**(es) para cada um de seus atributos.
Por exemplo, uma pessoa (entidade) pode possuir cor de olhos (atributo) azuis (valor).

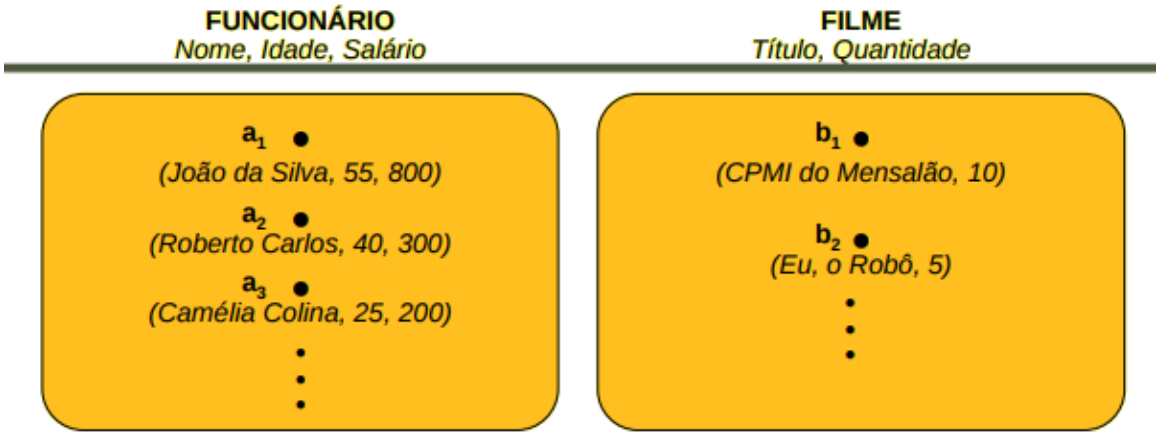
• Alguns atributos podem ser divididos em sub-partes com significados independentes. Por exemplo, um endereço (atributo) possui o Estado, cidade e rua como sub-atributos.

• Muitos atributos possuem apenas um único valor, e estes são chamados de univalorados. No entanto, é possível que existam atributos com um conjunto de valores, neste caso chamados de multivalorados. Por exemplo, uma pessoa pode possuir múltiplos números de telefone).

• Existem atributos com valores mais complexos. Um destes casos são os atributos derivados, os quais possuem valores que necessitam de algum tipo de processamento (utilizando informações obtidas do próprio banco de dados) para serem obtidos.
Por exemplo, podemos ter um atributo idade, com seus valores sendo expressos por: data_atual-data_de_nascimento (ou seja, as datas estão separadas por um hífen).

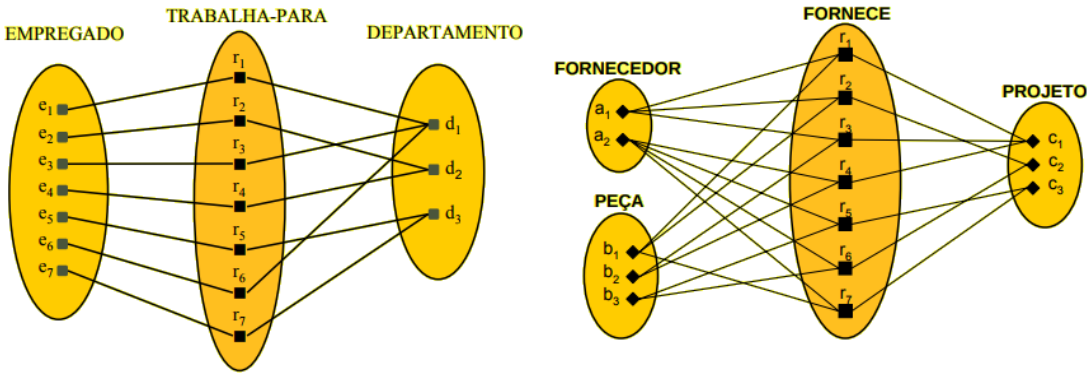
• Algumas vezes pode acontecer de um atributo não possuir valor. Nesses casos, atribui-se um valor nulo (*null*) para esse atributo. O valor *null* geralmente é utilizado quando ocorrem situações em que um valor não é aplicável, ou quando o valor é desconhecido.

• Entidades que possuem a mesma "estrutura" e a mesma semântica são representadas como **tipo de entidade**. Segue um diagrama exemplificando:



• Uma restrição importante sobre entidades de um tipo de entidade é a restrição de **atributo-chave**. Todo tipo de entidade deve ter um atributo-chave, seja ele simples ou composto, e os valores de um atributo-chave devem ser distintos. Esta unicidade deve valer para quaisquer extensões deste tipo de entidade.
Os atributos-chave são as *keys* (ou seja, *primary keys*, *foreign keys*, entre outras).

• Um **relacionamento** é uma associação entre uma ou mais entidades. Assim como existem tipos de entidades, existem **tipos de relacionamentos**. O **grau de um tipo de relacionamento** é igual ao número do tipos de entidades envolvidas no tipo de relacionamento em questão.



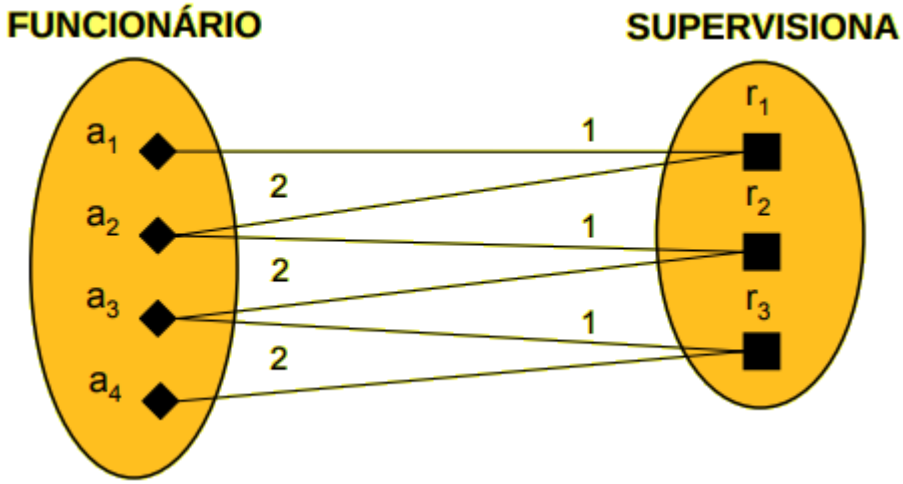
• Os relacionamentos podem ser representados como atributos. Por exemplo: o Tipo de Relacionamento EMPREGADO TRABALHA_PARA DEPARTAMENTO pode ser pensado como:

- EMPREGADO possuindo um atributo DEPARTAMENTO ou;
- DEPARTAMENTO possuindo um atributo EMPREGADO (multivalorado).

• Cada tipo de entidade que participa de um tipo de relacionamento possui um **papel** específico. O papel é apenas um nome que auxilia na representação semântica. A escolha deste nome nem sempre é simples.

• Existem casos, no entanto, em que a indicação do papel é necessária, como no caso de tipos de relacionamento recursivos (exemplificado a seguir). Outros casos em que a indicação do papel é necessária envolvem tipos de relacionamento em que a semântica não fique clara, ou seja ambígua.

• O



que torna a semântica dos modelos de bancos de dados realmente expressivas, no entanto, são as diversas **restrições** que podem ser impostas entre as entidades e os tipos de entidades. Estudemos

algumas delas, a seguir.

• A **razão de cardinalidade** é uma restrição que especifica a quantidade de instâncias de relacionamentos em que uma entidade pode participar (1:1, 1:N, N:N).

• A **participação** especifica se a existência de uma entidade depende dela estar relacionada com outra entidade através de um relacionamento. A participação pode ser total (dependência existencial) ou parcial.
Para exemplificar os tipos de participação: imagine o caso EMPREGADO TRABALHA_PARA DEPARTAMENTO. O empregado pode existir somente se trabalhar para algum departamento (caracterizando participação total), enquanto o departamento pode existir, mesmo sem possuir nenhum empregado (caracterizando participação parcial).

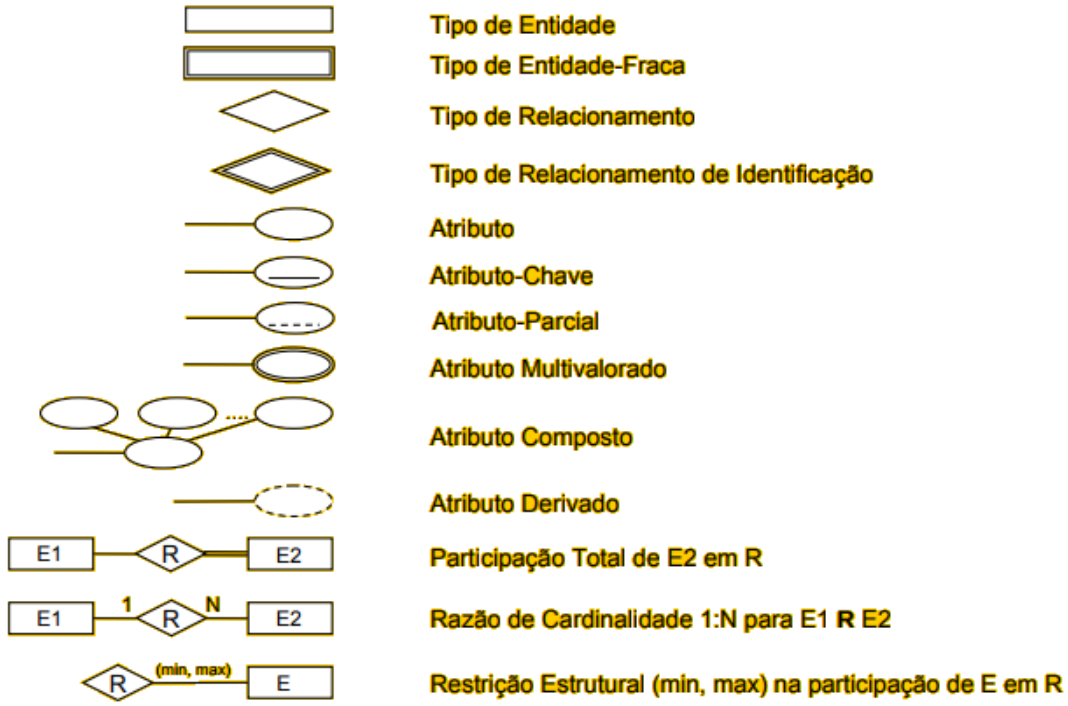
Este exemplo visual ilustra tanto a razão de cardinalidade quanto o exemplo de participação anteriormente definido.

• Uma **restrição estrutural** é o resultado da combinação das outras duas restrições apresentadas anteriormente. A restrição estrutural gera um intervalo numérico fechado, baseado na lógica imposta pelas restrições anteriores.
Por exemplo, no exemplo anterior, a restrição estrutural de EMPREGADO é (1,1), ou seja, ele pode participar no mínimo em 1 relacionamento, e no máximo em 1 relacionamento. Por outro lado, a restrição estrutural de DEPARTAMENTO é (0, N).

• Os tipos de relacionamentos também podem possuir atributos. Lembre-se: tipos de relacionamentos são puramente abstratos, e representam relações entre tipos de entidades, estes podendo ser representados concretamente por tabelas (como veremos futuramente), então por mais que digamos que os atributos estejam nos tipos de relacionamento, em verdade estamos adicionando mais um campo em uma das tabelas que se relacionam.
Por exemplo, considerando o tipo de relacionamento anteriormente ilustrado, podemos dizer que o tipo de relacionamento TRABALHA_PARA possui o atributo DataInicio (que representa quando um empregado começou a trabalhar em um departamento), desde que "coloquemos estes atributos no tipo de entidade que está do lado N do relacionamento".

• Por fim, possuímos os chamados **tipos de entidade-fraca**, que são tipos de entidade que não possuem atributos-chave. Desta forma, estes só podem ser identificadas através da associação com uma outra entidade. Podemos entender estes como uma outra forma de representar atributos, normalmente multivalorados.
Um tipo de entidade-fraca sempre tem restrição de participação total (dependência existencial) com respeito ao seu tipo de relacionamento de identificação, porque não é possível identificar uma entidade-fraca sem a correspondente entidade proprietária.
Um tipo de entidade-fraca tem uma chave-parcial, que é um conjunto de atributos que pode univocamente identificar entidades-fracas relacionadas à mesma entidade proprietária.

• Segue um resumo da notação utilizada nos Diagramas Entidade-Relacionamento, para possível referência:



• Passemos, portanto, a discutir o **Modelo de Dados Relacional**.

• O Modelo Relacional (**MR**) é um modelo de dados lógico utilizado para desenvolver projetos lógicos de bancos de dados. Os SGBDs que utilizam o MR são denominados SGBD Relacionais. O MR apresenta os dados do BD como **relações** (não confundir com as relações apresentadas anteriormente; a palavra relação é apresentada no sentido de "lista" ou rol de informações).

• Note que Modelo de Dados Relacional e Modelo Relacional são o mesmo conceito. Logo, as siglas **MR** e **MDR** são equivalentes.

• Cada relação pode ser entendida como uma **tabela**, ou um simples arquivo de registros. As relações possuem **atributos**, os quais possuem **valores**. Os valores de atributos são indivisíveis (atômicos). O conjunto de atributos de uma relação é chamado de **relação esquema**. Cada atributo possui um **domínio**, que consiste dos possíveis valores que o atributo pode possuir, e uma descrição dos mesmos. O **grau de uma relação** é o número de atributos que uma relação possui. Cada linha de uma relação (tabela) nos fornece uma **tupla** de valores relacionados.

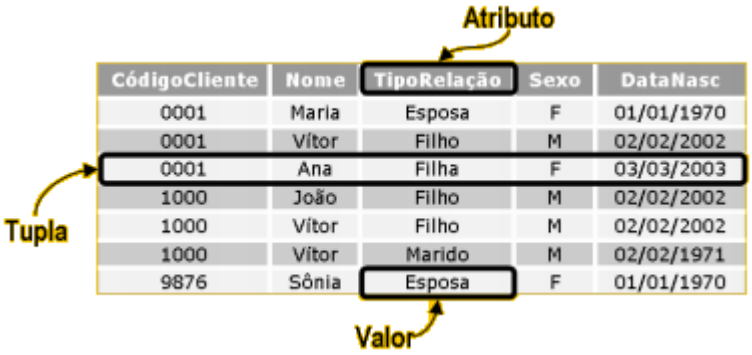
• Existe a chamada **notação relacional**, que auxilia na representação de diversas partes de uma relação:

• $R(A_1, A_2, ..., A_n)$ representa a relação esquema de grau "n", com A_m sendo o m-ésimo atributo da relação.

• $t = <v_1, v_2, ..., v_n>$ representa a tupla t em uma relação t(R), e v_m representa o valor do m-ésimo atributo da relação R.

• $t[A_m]$ representa o mesmo que v_m, ou seja, o valor do m-ésimo atributo da Relação R (que t participa). Podemos representar um conjunto de valores de forma similar ($t[A_u, A_w, ..., A_z]$).

• Podemos perceber rapidamente a utilidade da capacidade de selecionar *subsets* de valores, baseados em atributos desejados.



CódigoCliente	Nome	TipoRelação	Sexo	DataNasc
0001	Maria	Esposa	F	01/01/1970
0001	Vitor	Filho	M	02/02/2002
0001	Ana	Filha	F	03/03/2003
1000	João	Filho	M	02/02/2002
1000	Vitor	Filho	M	02/02/2002
1000	Vitor	Marido	M	02/02/1971
9876	Sônia	Esposa	F	01/01/1970

Na figura acima, por exemplo, t = <0001, Ana, Filha, F, 03/03/2003> é uma tupla.

Outro exemplo: t[CódigoCliente] = 0001, e t[Nome, Sexo] = Ana, Filha.

• As relações possuem atributos denominados **atributos-chaves**. Uma **superchave** é um subconjunto de atributos de uma relação cujos valores são distintos (ou seja, pode constituir de um ou mais atributos). Uma **chave** é uma superchave mínima, uma **chave-candidata** também é uma chave em uma relação, e uma **chave-primária** é uma das chaves escolhidas dentre as chaves-candidatas de uma relação.

• Para a relação ilustrada acima, por exemplo, existe a superchave trivial que é representada por SC_a = { CodigoCliente, Nome, TipoRelação, Sexo, DataNasc }. SC_b = { CodigoCliente, Nome } é uma superchave mínima, pois não podemos retirar nenhum de seus atributos sem remover a propriedade de superchave do conjunto resultante, e portanto SC_b é uma chave.

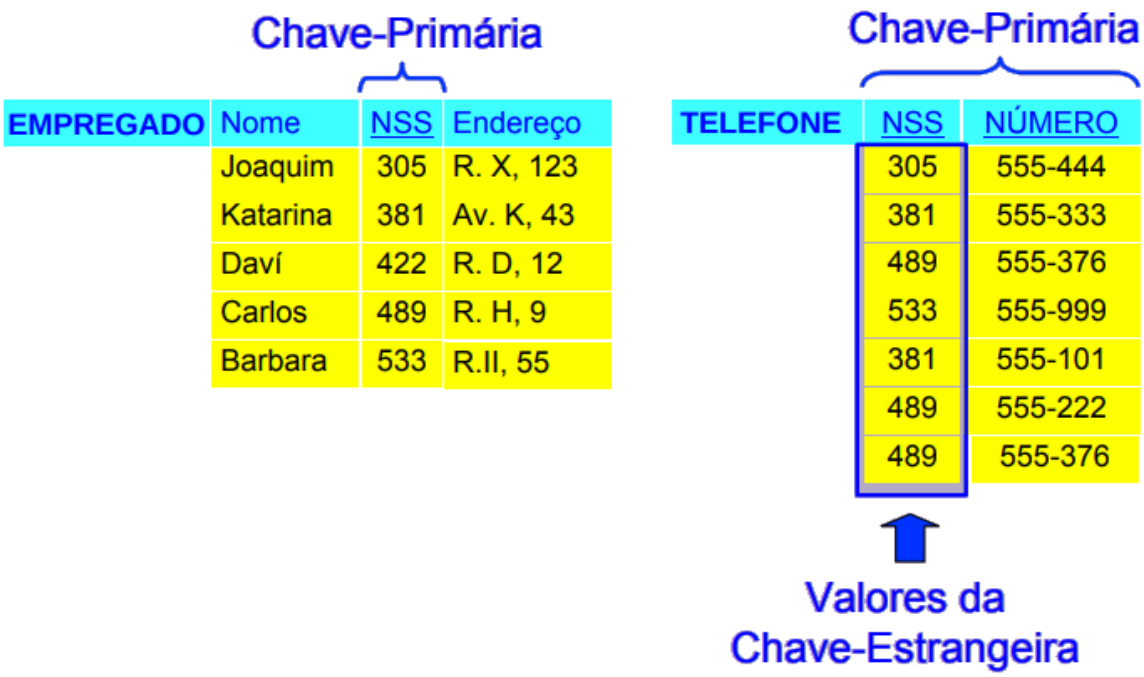
• No caso de uma relação esquema possuir mais de uma chave possível, todas as chaves são chamadas de chaves-candidatas. A chave-primária é a escolhida, dentre as chaves-candidatas, para

identificar de forma única, tuplas de uma relação. A chave-primária é indicada na relação esquema sublinhando-se os seus atributos.

• O **esquema de um BD relacional** é o conjunto de todos os esquemas de relações presentes neste Banco de Dados.

• As **restrições de integridade** são regras que restringem os valores que podem ser armazenados nas relações. Um SGBD relacional deve garantir as seguintes restrições:

• Restrição de Chave: os valores das chavescandidatas devem ser únicos em todas as tuplas de uma relação.
• Restrição de Entidade: chaves-primárias não podem ter valores nulos.
• Restrição de Integridade Referencial: Usada para manter a consistência entre tuplas. Estabelece que um valor de atributo, que faz referência a uma outra tupla, deve-se referir a uma tupla existente.



Exemplo de restrição de integridade referencial.

• É comum, em projetos lógicos de BD, realizar a modelagem dos dados através de um modelo de dados de alto nível. O produto deste processo é o esquema do BD (neste caso específico, o esquema do BD é relacional).

O modelo de dados de alto-nível normalmente adotado é o Modelo Entidade-Relacional, e o esquema do BD é especificado no Modelo Relacional.

• Nos slides sobre este tema (https://www.ime.usp.br/~jef/bd03.pdf), existe um exemplo da criação de um esquema de BD no Modelo Relacional, a partir de um modelo de dados Entidade-Relacional. Ele mostra como criar tabelas a partir de entidades do modelo de dados, e como discernir seus atributos, entre outras informações pertinentes.

• Passemos, agora, a falar do **Modelo Entidade-Relacionamento Extendido** (**MER-X**). O MER-X é uma extensão do MER, o qual adiciona a abstração de agregação, e a abstração de generalização/especialização.

• Falemos, primeiro, da **abstração de agregação**. Esta abstração é a que permite construir objetos compostos a partir de objetos componentes. Ou seja, os elementos de modelagem podem ser associados formando outros elementos, que representam essa associação.

• No MER-X, essa associação pode ocorrer de duas maneiras distintas:

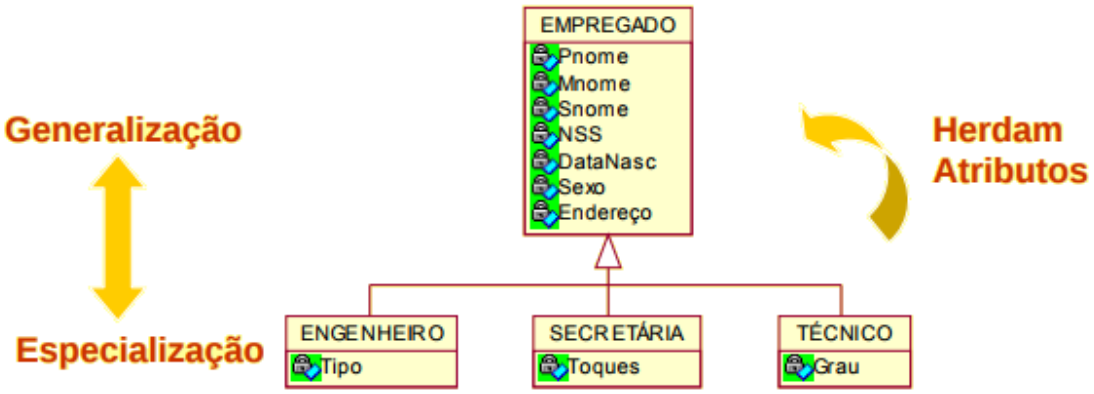
- Agregando Atributos a Tipos de Entidades (gerado as Classes, que vimos em UML) e aos Tipos de Relacionamento (gerando as Classes Associativas);
- Combinar Tipos de Entidades relacionadas por meio de um Tipo de Relacionamento e compor um Tipo de Entidade Agregada (de nível abstrato mais alto).

• Nos slides sobre este tema (https://www.ime.usp.br/~jef/bd04.pdf), podemos ter alguns exemplos deste tipo de associação. Podemos ver que a grande utilidade deste tipo de operação é que permite que tenhamos uma tradução direta de nosso Diagrama Entidade-Relacional para um diagrama UML.

• A abstração de **especialização/generalização**, por sua vez, é utilizada com propósitos similares a abstração de agregação. Como vimos anteriormente, os tipos de entidades representam classes. Desta forma, podemos pensar no conceito ja visto de hierarquia de classes. Como o nome sugere, esta abstração será utilizada (e é utilizada, com nomes similares) em UML para apresentar o conceito

de hierarquia entre classes.

• Podemos ver, novamente nos slides sobre este tema, que podemos ter subclasses e superclasses, e a transição entre elas envolve a generalização/especialização. As subclasses herdam atributos de suas superclasses, o que torna o diagrama mais compreensivo e compacto.



• Podemos ver nos slides que existem formas mais complexas de se trabalhar com este tipo de abstração, como, por exemplo, herança múltipla de classes, entre outros conceitos.

• É possível transformar um mapeamento DER-X para o MDR. Podemos seguiros mesmos passos do conjuntos de slides anterior, com algumas diferenças, explicitadas neste slide.

• Agora que abordamos as principais formas de modelagem conceitual e lógica, passemos para um assunto mais abstrato: a **Álgebra Relacional**.

• Como o nome sugere, a Álgerbra Relacional contempla um conjunto de operações que permitem especificar consultas sobre relações.

• As operações são divididas em dois grupos:

  • Operações da Teoria de Conjuntos;
  • Operações desenvolvidas especificamente para Bancos de Dados Relacionais.

• A seguir, falemos primeiramente das operações desenvolvidas especificamente para Bancos de Dados Relacionais.

• Começemos estudando o operador **SELECT**. O operador SELECT é representado pelo símbolo **σ**, e é utilizado para selecionar, segundo

alguma condição, tuplas de uma relação.
Por exemplo: selecionar os empregados que trabalham para o departamento 4 é dada pela sintaxe $\sigma_{NDEP = 4}$ (EMPREGADO).

• Para especificar as condições, podemos nos utilizar dos seguintes recursos: valores constantes; nomes de atributos; operadores relacionais (=, <, ≤, >, ≥, ≠); operadores lógicos (AND, OR, NOT).

• Exemplo de uma consulta mais complexa: Selecionar os empregados que trabalham no departamento 4 e ganham mais de 2500 ou aqueles que trabalham no departamento 5 e ganham mais que 3000, dado pela expressão

$$\sigma_{(NDEP = 4 \text{ AND } SALÁRIO > 2500) \text{ OR } (NDEP = 5 \text{ AND } SALÁRIO > 3000)} (EMPREGADO)$$

• O operador SELECT é um operador unário, ou seja, ele seleciona tuplas de apenas uma relação. O grau da relação resultante é o mesmo da relação original. Além disso, o operador SELECT é comutativo, ou seja, ele podemos trocar operações SELECT em cascata pela conjuntiva "AND".
Exemplo: $\sigma_{<cond\ 1>}(\sigma_{<cond\ 2>}(R)) = \sigma_{<cond\ 1> AND <cond\ 2>}(R)$

• Passemos agora para o operador **PROJECT** (com o significado de "projetar"). O operador PROJECT é representado pelo símbolo **π** (pi). Enquanto o operador SELECT seleciona tuplas de uma relação, o operador PROJECT seleciona colunas de uma relação.
Exemplo: $\pi_{SNOME, PNOME, SALÁRIO}$ (EMPREGADO)

• O operador PROJECT remove quaisquer tuplas duplicadas da relação resultante. Isto implica que o número de tuplas resultante sempre será menor ou igual ao número de tuplas da relação original. Note que o operador PROJECT não é comutativo.
Além disso, caso a lista de atributos de uma operação PROJECT contenha a lista de atributos de outra operação, que será realizada em cascata após a primeira operação, esta operação conjunta será igual a realizar apenas a primeira consulta.

• É possível combinar os operadores em uma única expressão para realizar uma consulta mais complexa.

• Podemos criar relações (tabelas) intermediárias para explicitar a sequencia de operações, e aumentar a clareza desta. Além disso, ao criarmos uma nova relação, podemos renomear os atributos desta.

Exemplo:

$$DEP5\_EMPS \leftarrow \sigma_{NDEP=5}(EMPREGADO)$$
$$RESULT_{(NOME,\ SOBRENOME,\ SALÁRIO)} \leftarrow \pi_{PNOME,\ SNOME,\ SALÁRIO}(DEP5\_EMPS)$$

• Agora, passemos a falar das operações de teoria dos conjuntos. Os operadores de teoria dos conjuntos aplicam-se ao modelo relacional pois uma relação é como um conjunto de tuplas.

• Os operadores são os seguintes:

- R∪S - União (todas as tuplas de R e todas de S);
- R∩S - Intersecção (todas as tuplas comuns a R e S);
- R–S - Diferença (todas as tuplas de R que não estão em S);
- R x S – Produto Cartesiano (combinação das tuplas de R com as de S).

• Todas estas operações, com exceção do Produto Cartesiano, requerem que R e S estejam compatíveis no momento da operação (ou seja, possuam os mesmos atributos). As três primeiras operações, de fato, são as mais intuitivas, enquanto a última pode ser um pouco mais complicada. Um exemplo bom se encontra no 21º slide do assunto em questão (https://www.ime.usp.br/~jef/bd05.pdf).

• Agora, falemos do operador **JOIN**. O operador join é representado pelo símbolo ⋈. Este operador é um dos mais úteis da álgebra relacional, e é geralmente utilizado para combinar informações de duas ou mais relações. O JOIN pode ser definido como um produto cartesiano seguido por uma seleção.
Exemplo:

$$EMP\_DEP \leftarrow EMP\_NOMES\ X\ DEPENDENTE$$
$$DEP\_ATUAL \leftarrow \sigma_{NSS=ENSS}(EMP\_DEP)$$

=

$$DEP\_ATUAL \leftarrow EMP\_NOMES \bowtie_{NSS=ENSS} DEPENDENTE$$

• É comum encontrar JOIN que tenham somente comparações de igualdade. Quando isso ocorre, o JOIN é chamado de EQUIJOIN. É notável que, no resultado de um EQUIJOIN haverá, sempre, um ou mais pares de atributos com valores idênticos.

• Devido a tal duplicidade ser desnecessária, uma nova operação foi criada: NATURAL JOIN. O NATURAL JOIN (representado pelo símbolo *), é um EQUIJOIN seguido da remoção de atributos desnecessários. A

forma geral desse operador é:

$$Q \leftarrow R *_{(lista1),\ (lista2)} S, \text{ onde:}$$

lista1 especifica os atributos de R, e lista2 os atributos de S. Na relação resultante, os atributos da lista2 não irão aparecer. Pode-se continuar a especificar o sinal de igualdade na condição, apesar de ser desnecessário.
Por exemplo, podemos escrever:

$$PROJ\_DEPT \leftarrow PROJETO *_{DNUM\ =\ DNÚMERO} DEPARTAMENTO$$

ou

$$PROJ\_DEPT \leftarrow PROJETO *_{(DNUM),\ (DNÚMERO)} DEPARTAMENTO$$

• Em seguida, falemos do operador **DIVISION**. O operador DIVISION é representado pelo símbolo ÷, e é útil para um tipo específico de consulta que ocorre com frequência.
Por exemplo: recuperar os nomes de empregados que trabalham em todos os projetos em que John Smith trabalha.

• É importante perceber que tanto os diversos tipos de JOIN quanto o DIVISION podem ser escritos como uma combinação dos operadores básicos vistos anteriormente.
A divisão, por sua vez, pode ser escrita em termos dos operadores básicos: π, X e –.

• Sejam duas instâncias de relação A(x, y) e B(y). O resultado de A÷B contém todos os valores x de A que não são desqualificados. Note que um valor x é desqualificado se, ao anexar um valor y de B, resultar em tuplas <x, y> que não estão em A.
A seguinte expressão permite realizar justamente isso:

$$XDESQUALIFICADO \leftarrow \pi_x ((\pi_x (A) \times B) - A)$$

Agora, basta tirar de A as tuplas desqualificadas, resultando na divisão:

$$A \div B \leftarrow \pi_x (A) - XDESQUALIFICADO, \text{ portanto}$$

$$A \div B \leftarrow \pi_x (A) - \pi_x ((\pi_x (A) \times B) - A)$$

• Agora, falemos de **funções de agregação**. Funções de agregação recebem como conjunto de entrada um conjunto de tuplas, e devolvem um único valor. Algumas funções de agregação são as seguintes: SUM, AVERAGE, MAXIMUM, MINIMUM. Funções de agregação são escritas utilizando-se do operador FUNCTION (expresso pelo símbolo I).
Por exemplo: recuperar para cada departamento, o número de empregado e sua média salarial, é feito pela expressão:

$$R_{(DNO,\ NRO\_EMPS,\ MÉDIA)} \leftarrow NDEP\ I_{COUNT\ NSS,\ AVERAGE\ SALÁRIO}\ (EMPREGADO)$$

Note que, neste caso, NDEP é o atributo de agruamento. Se nenhum atributo de agrupamento for especificado, as funções de agregação irão ser aplicadas para todas as tuplas da relação.

• Na álgebra relacional, é impossível definir a clausura recursiva. A clausura recursiva ocorre quando existem relacionamentos recursivos. Porém, como ela não pode ser definida, não a daremos muita atenção.

• Os operadores JOIN vistos até agora, nos quais apenas tuplas que satisfazem a condição de junção são mantidas no resultado, são conhecidas como junções internas (**INNER JOINS**).

• Junções externas (**OUTER JOINS**) podem ser utilizadas quando queremos manter todas as tuplas de R, S ou de ambas no resultado do JOIN, independentemente de existirem tuplas correspondentes na outra relação.
Por exemplo: obter a lista de nomes de todos os empregados e o nome dos departamentos que gerenciam. Se gerenciarem algum departamento. Se não gerenciarem nenhum departamento, indicar com um valor null, é dada pela consulta:

$$TEMP \leftarrow (EMPREGADO =\!\triangleright\!\triangleleft_{NSS=GERNSS}\ DEPARTAMENTO)$$
$$RESULTADO \leftarrow \pi_{PNOME,\ MNOME,\ SNOME,\ DNOME}\ (TEMP)$$

• O operador visto na consulta acima representa um **LEFT OUTER JOIN**. O operador LEFT OUTER JOIN mantém todas as tuplas da primeira relação no resultado. O operador **RIGHT OUTER JOIN**

mantém todas as tuplas da segunda relação no resultado. Um terceiro operador, **FULL OUTER JOIN**, indicada por ⟹⟸ , mantém todas as tuplas em ambas as relações, preenchendo quando necessário as tuplas não casadas.

• Agora que finalizamos nossa abordagem de Álgebra Relacional, tratemos do **Cálculo Relacional de Tuplas** (**CRT**). Em verdade, o CRT é apresentado como uma alternativa à Álgebra Relacional. Enquanto a Álgebra Relacional é procedimental, o CRT é declarativo. O CRT permite descrever um conjunto de respostas sem explicitar como elas serão computadas.

• O CRT influenciou fortemente as linguagens de consulta comerciais, tais como a SQL. Uma linguagem de consulta L é considerada relacionalmente completa se L expressar qualquer consulta que possa ser realizada em CRT.

• Uma consulta em CRT tem o seguinte formato:

$$\{ t \mid P(t)\}$$

{ t | P(t) } representa o conjunto de todas as tuplas t, tal que o predicado P é verdadeiro para t. Neste caso, t é uma variável de tuplas, P é uma expressão condicional, e t.A ou t[A] denota o valor do atributo A da tupla t.

• O seguinte exemplo é uma consulta em CRT:

$$\{ t \mid EMPREGADO(t) \text{ AND } t.SALARIO > 5000 \}$$

Note que EMPREGADO(t) é equivalente a dizer que t ∈ EMPREGADO.

• Podemos especificar, no CRT, os atributos que desejamos saber, antes da condição para selecionar tuplas.
Por exemplo:

$$\{ t.PNOME , t.SNOME \mid EMPREGADO(t) \text{ AND } t.SALARIO > 5000 \}$$

• Uma expressão geral do CRT é da forma:

$$\{ t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid P(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m}) \}$$

• Denominamos $t_1, \ldots, t_{n+m}$ como variáveis de tuplas, A_i como um atributo correspondente à tupla associada, e P como uma condição ou **fórmula**.

• Uma fórmula é feita por **átomos**, que podem ser: $R(t_i)$, onde R é a relação e t_i é uma variável de tupla; $t_i.A$ op $t_j.B$, onde op ∈ {=, <, ≤, >, ≥, ≠ }; $t_i.A$ op c ou c op $t_i.A$, onde c é uma constante.

• Cada átomo resulta em um valor TRUE ou FALSE. Para átomos da forma R(t), se t ∈ R, então temos TRUE, e caso contrário, temos FALSE. Uma fórmula pode ser composta por um ou mais átomos conectados pelos operadores lógicos AND, OR e NOT. A implicação (⇒) e a dupla implicação (⇔) também podem ser usadas. Lembre-se:

$$X \Rightarrow Y \equiv (\text{NOT } X) \text{ OR } Y$$
$$X \Leftrightarrow Y \equiv (X \Rightarrow Y) \text{ AND } (Y \Rightarrow X)$$

• As fórmulas também podem incorporar os seguintes quantificadores: o quantificador universal (**para todo**, ∀) e o quantificador existencial (**existe**, ∃).
As tuplas t_1 e t_2 , nas cláusulas $\forall t_1$ ou $\exists t_2$, são variáveis de tupla vinculadas. Se t não for vinculada, então será livre.

• É possível perceber, neste momento, que as fórmulas respeitam as mesmas regras de **Lógica de Primeira Ordem**.

• Neste contexto, temos que todo átomo é uma fórmula. Se F_1 e F_2 são fórmulas, então F_1 AND F_2, F_1 OR F_2, NOT(F_1) e NOT(F_2) são fórmulas. Se F é fórmula, então (∃t)(F(t)), também será. A fórmula (∃t)(F(t)) será TRUE se F for TRUE para pelo menos uma tupla t.
Se F é fórmula, então (∀t)(F(t)), também será. A fórmula (∀t)(F(t)) será TRUE se F for TRUE para todas as tuplas t no universo.

• As seguintes transformações também são válidas:

- $F_1 \Rightarrow F_2 \equiv \text{NOT } F_1 \text{ OR } F_2$
- F_1 AND $F_2 \equiv$ NOT (NOT F_1 OR NOT F_2)
- $(\forall t)\ (F(t)) \equiv \text{NOT } (\exists t)\ (\text{NOT } F(t))$
- $(\exists t)\ (F(t)) \equiv \text{NOT } (\forall t)\ (\text{NOT } F(t))$
- $(\forall t)\ (F_1(t) \text{ AND } F_2(t)) \equiv \text{NOT } (\exists t)\ (\text{NOT } (F_1(t)) \text{ OR NOT } (F_2(t)))$
- $(\forall t)\ (F_1(t) \text{ OR } F_2(t)) \equiv \text{NOT } (\exists t)\ (\text{NOT } (F_1(t)) \text{ AND NOT } (F_2(t)))$
- $(\exists t)\ (F_1(t) \text{ AND } F_2(t)) \equiv \text{NOT } (\forall t)\ (\text{NOT } (F_1(t)) \text{ OR NOT } (F_2(t)))$
- $(\exists t)\ (F_1(t) \text{ OR } F_2(t)) \equiv \text{NOT } (\forall t)\ (\text{NOT } (F_1(t)) \text{ AND NOT } (F_2(t)))$
- $(\forall t)\ (F(t)) \Rightarrow (\exists t)\ (F(t))$
- $\text{NOT } (\exists t)\ (F(t)) \Rightarrow \text{NOT } (\forall t)\ (F(t))$

• Nos slides são dados alguns exemplos de operações de Álgebra Relacional, e as operações equivalentes em CRT. Não serão

explicitadas todos neste resumo, mas seguem alguns exemplos.

• Um exemplo de projeção:

$$\pi_{\text{SNOME, PNOME, SALÁRIO}}\,(\text{EMPREGADO})$$
$$=$$
$$\{\ t.\text{PNOME}\ ,\ t.\text{SNOME}\ ,\ t.\text{ENDERECO}\ |\ \text{EMPREGADO}(\ t\ )\ \}$$

• Um exemplo de seleção:

$$\sigma_{\text{sexo='F'}}\,(\text{EMPREGADO})$$
$$=$$
$$\{\ t\ |\ \text{EMPREGADO}(t)\ \text{AND}\ t.\text{SEXO} = \text{'F'})\ \}$$

• Um exemplo de JOIN: nós queremos recuperar o nome e o endereço de todos os empregados que trabalham para o departamento 'Pesquisa'.

$$\text{DEP} \leftarrow \sigma_{\text{DNOME = 'Pesquisa'}}\,(\text{DEPARTAMENTO})$$
$$\text{EMPDEP} \leftarrow (\text{DEP} \rhd\lhd_{\text{DNÚMERO = NDEP}}\,\text{EMPREGADO})$$
$$\text{RESULT} \leftarrow \pi_{\text{PNOME, SNOME, ENDEREÇO}}\,(\text{EMPDEP})$$

$$=$$

$$\{\ t.\text{PNOME}\ ,\ t.\text{SNOME}\ ,\ t.\text{ENDERECO}\ |\ \text{EMPREGADO}(t)\ \text{AND}$$

$$(\exists d)\ (\ \text{DEPARTAMENTO}(d)\ \text{AND}$$

$$d.\text{DNOME} = \text{'Pesquisa'}\ \text{AND}\ d.\text{DNUMERO} = t.\text{DNO}\ )\ \}$$

• Uma expressão em CRT pode gerar uma infinidade de relações. Por exemplo, a expressão

$$\{t\ |\ \text{NOT}\ (\text{R}(t))\}$$

pode gerar uma infinidade de tuplas que não estão em R.
Uma **expressão segura** é uma expressão que garante a produção de um número finito de tuplas como resultado. A expressão acima, por exemplo, não é uma expressão segura.

• Em seguida, estudemos outro tipo de Cálculo Relacional: o **Cálculo Relacional de Domínio** (**CRD**). O CRD é uma linguagem de consulta não-procedimental, com capacidade expressiva equivalente ao CRT. Agora, diferentemente do CRT, o CRD utiliza-se de variáveis de domínio, ao invés de tuplas. O CRD influenciou fortemente algumas

linguagens de consulta comerciais, como a QBE.

• As variáveis de domínio, envolvidas neste tipo de cálculo, representam os atributos de uma relação.
Por exemplo, queremos recuperar a data de aniversário e endereço do empregado cujo nome é "John B. Smith":

$$\{ uv \mid (\exists q)(\exists r)(\exists s)(\text{EMPREGADO } (qrstuvwxyz) \text{ AND } q=\text{'John'} \text{ AND } r=\text{'B'} \text{ AND } s=\text{'Smith'}) \}$$

sendo "q" a coluna (o atributo) PNOME, "r" a coluna MNOME, e "s" a coluna SNOME, "u" a coluna DATANASC e "v".

• Para formar uma relação de grau n, especifica-se n variáveis de domínio.

$$\{ x_1 , x_2, \ldots, x_n \mid P (x_1 , x_2 , \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}) \}$$

Temos que $x_1 , x_2, \ldots, x_n$ representam as variáveis livres de domínio, e P é um predicado.

• Um predicado é uma fórmula atômica composta por:

- Uma fórmula atômica $R(x_1 , x_2, \ldots, x_n)$, em que R é o nome de uma relação de grau "n", e cada um dos x_i, com $1 \leq i \leq n$, é uma variável de domínio;
- Uma fórmula atômica x op y, em que op $\in \{=, <, >, \leq, \geq, \neq\}$ e x e y são variáveis de domínio;
- Uma fórmula atômica x op c (ou c op x), em que c é uma constante, e x é uma variável de domínio.

• Assim como no CRT, as fórmulas são avaliadas em valores-verdade. $R(x_1 , x_2, \ldots, x_n)$, será TRUE apenas se houver valores de domínio correspondentes a uma tupla de R. Temos que (x op y) ou (x op c) ou (c op x) será TRUE caso as variáveis de domínio tenham valores que satisfaçam.

• Os quantificadores vistos anteriormente em CRT podem ser utilizados também em CRD.

• São dados alguns exemplos de uso nos slides (https://www.ime.usp.br/~jef/bd07.pdf).
Um outro exemplo relevante é o seguinte: selecionar o nome e o

endereço dos empregados que trabalham para o departamento de ‘Informática’.

{ qsv | (∃z) (∃l) (∃m) (EMPREGADO (qrstuvwxyz)

AND DEPARTAMENTO (lmno) AND

l = ‘Pesquisa’ AND m = z) }

• O conceito de expressões seguras também existe em CRD. Uma expressão é dita segura se atende os seguintes requisitos: todos os valores que aparecem nas tuplas da expressão são valores dentro do domínio da mesma; todas as fórmulas (∃x)(P(x)) são verdadeira se, e somente se, existir um valor x no domínio de P tal que P(x) seja verdadeiro; todas as fórmulas (∀x)(P(x)) são verdadeiras se, e somente se, P(x) for verdadeiro para todos os valores de x dentro do domínio de P.
As proposições acima garantem que possamos testar todas as fórmulas “existe um” e “para todo” sem a necessidade de testar todas as suas infinitas possibilidades de ocorrência.

• Enfim, estudemos uma das linguagens mais utilizadas para manipulação e consulta a bancos de dados: a **SQL**.

• SQL, ou Structured Query Language, é uma linguagem declarativa (ou seja, não procedimental). Para recordar: uma linguagem declarativa refere-se a uma linguagem com a utilidade de descrever as suas estruturas, e não como elas serão utilizadas.

• Desenvolvida pela IBM, é um padrão industrial que atinge grande parte do mercado de SGBDs. É uma linguagem simples, concisa, e com grande poder de consulta.

• A linguagem SQL possui tanto aspectos de **DDL** (Data Definition Language), com seus comandos CREATE, ALTER e DROP, quanto aspectos de **DML** (Data Manipulation Language), com seus comandos INSERT, UPDATE e DELETE).
Em verdade, todos os comandos em SQL são divididos nestas duas grandes categorias: o aspecto DDL permite a especificação do esquema base de dados, enquanto o aspecto DML
permite a inserção, remoção, alteração e consultas nas instâncias da base de dados.

• O SQL também possui as seguintes capacidades: criação de visões (views); criação de especificações de segurança e autorizações; controle de transações; regras para interação com linguagens de programação, entre outros recursos.

• As grandes vantagens do SQL provêm do fato de que o SQL baseia-se no Modelo de Dados Relacional. No entanto, o vocabulário do SQL diverge do vocabulário original do Modelo Relacional. Por exemplo: uma relação é chamada de **tabela**; uma tupla é chamada de **linha**; um atributo é chamado de **coluna**. E, acima de tudo, uma das maiores divergências com relação ao Modelo Relacional refere-se ao fato que a linguagem SQL admite que tabelas possuam linhas idênticas.

• Agora, vamos tratar com mais detalhes alguns comandos da DLL.

• O **CREATE SCHEMA** é um comando utilizado para criar esquemas de aplicações. O **esquema** permite agrupar as tabelas, restrições, visões, domínios e outros construtores (como concessão de autoridade) que descrevem o esquema.
Um exemplo:

CREATE SCHEMA COMPANHIA AUTHORIZATION MAC0426

• O **CREATE DOMAIN** é utilizado para definir domínios de atributos. O formato geral de utilização do comando é:

CREATE DOMAIN nome AS tipo [<restrições de coluna>]

É facilitada, desta forma, a redefinição de tipos de dados de um domínio utilizados por muitos atributos de um esquema, além de melhorar a legibilidade do esquema.
Um exemplo concreto:

CREATE DOMAIN TIPO_NSS AS CHAR(9)

Podemos definir um novo domínio com a especificação de uma restrição sobre o tipo de dados.
Por exemplo:

CREATE DOMAIN TIPO_DEPNUM AS INTEGER
CHECK (TIPO_DEPNUM > 0 AND TIPO_DEPNUM < 21);

• O **CREATE TABLE** é um comando que cria uma tabela (tabela base), e define suas colunas e restrições. Uma forma geral e simples

de utilização do comando é:

CREATE TABLE [esquema].tabela (

atrib1 tipo [<restrições da coluna 1>],

atrib2 tipo [<restrições da coluna 2>],

....

atribn tipo [<restrições da coluna n>],

<restrições da tabela>

);

Primeiramente, percebemos que cada atributo (coluna) da tabela possui um tipo associado. Alguns dos principais tipos de dados em SQL são os seguintes: INTEGER e SMALLINT; DECIMAL [(*precision*, *scale*)], em que *precision* é o número total de dígitos, e *scale* é o número de dígitos decimais; DOUBLE PRECISION, FLOAT e REAL; CHAR (n), que consiste de uma junção de caracteres de tamanho fixo "n"; VARCHAR (n), que consiste de uma junção de caracteres de tamanho variável, até "n"; BLOB, um *Binary Large Object*; DATE, TIME e TIMESTAMP.
As restrições de coluna são: NOT NULL; DEFAULT *valor*; CHECK (*condição*).
As restrições de tabela são: PRIMARY KEY (<atributos da chave primária>); UNIQUE (<atributos chave candidata>);
FOREIGN KEY (<atributos chave estrangeira>
REFERENCES tabelaRef [(<chave primária>)] [<**ações**>].
Como podemos ver, existem algumas ações relacionadas a *foreign keys* (chaves estrangeiras). Estas são as seguintes: ON DELETE e ON UPDATE (condições para ativar uma determinada ação); CASCADE, SET NULL e SET DEFAULT (ações em si); CHECK (*condição*).

• Podemos dizer, então, que a forma geral de criação de tabelas pode ser dada da seguinte forma:

CREATE TABLE [esquema].tabela (

```
    atrib1 tipo [(tamanho)] [NOT NULL | DEFAULT valor] [CHECK  (condição)],
    atrib2 tipo [(tamanho)] [NOT NULL | DEFAULT valor] [CHECK  (condição)],
    ...
    [CONSTRAINT nome da restrição]
        PRIMARY KEY (<atributos chave primária>),
    [CONSTRAINT nome da restrição]
        UNIQUE (< atributos chave candidata>),
    [CONSTRAINT nome da restrição]
        FOREIGN KEY (<atributos chave estrangeira>)
            REFERENCES tabelaRef [(<chave primária>)]
                [ON DELETE CASCADE | SET NULL | SET DEFAULT]
                [ON UPDATE CASCADE | SET NULL | SET DEFAULT],
    [CONSTRAINT nome da restrição]
        CHECK (condição)
);
```

Exemplos concretos de criação de tabelas são dadas nos slides (https://www.ime.usp.br/~jef/bd08.pdf).

• O comando **ALTER TABLE** é utilizado para incluir/alterar/remover definições de colunas e restrições. A forma geral de utilização do comando é:

ALTER TABLE tabela <ação>;

As ações possíveis são as seguintes:

• ADD novoAtrib tipo [<restrições de coluna>]. O valor do novo atributo nas linhas (tuplas) será *null*, desde que não seja especificada nenhuma cláusula *default*. Assim, a cláusula NOT NULL não pode ser aplicada. Um exemplo:
ALTER TABLE COMPANHIA.EMPREGADO ADD FUNCAO VARCHAR(12);

• ADD [CONSTRAINT nome] <restrição de tabela>;

• DROP atributo [CASCADE | RESTRICT]. A ação **CASCADE** faz com que todas as visões e restrições (constrains) que referenciam o atributo sejam removidas automaticamente. A ação **RESTRICT** faz com que o atributo só é removido se não houver nenhuma visão ou restrição que o referencie;

• DROP CONSTRAINT nome.

• O comando **DROP TABLE** é utilizado para excluir uma tabela do banco de dados. A forma geral de utilização do comando é:

DROP TABLE tabela [CASCADE | RESTRICT];

• O comando **DROP SCHEMA** é análogo ao comando anterior, mas é utilizado para esquemas. A forma geral de utilização deste comando é:

DROP SCHEMA esquema [CASCADE | RESTRICT];

• Em seguida, vamos tratar com mais detalhes alguns comandos da DML.

• O comando **INSERT** é utilizado para inserir uma ou mais linhas (tuplas) em uma tabela. As formas gerais de utilização são as seguintes (primeiro uma inserção simples, e depois uma inserção de múltiplas linhas):

INSERT INTO tabela [(atrib1,atrib2,...)] VALUES (valor1, valor2,...)

INSERT INTO tabela [(atrib1,atrib2,...)] <comando SELECT>

Os exemplos concretos existentes nos slides são bastante úteis.

• O comando **UPDATE** modifica o valor de um atributo em uma ou mais tuplas da tabela. A forma geral de utilização deste comando é a seguinte:

```
UPDATE tabela SET
        atributo1 = <valor ou expressão>,
        atributo2 = <valor ou expressão>,
        ...
WHERE <condição de localização>;
```

• O comando **DELETE** é utilizado para remover uma ou mais tuplas da tabela. A forma geral de utilização deste comando é a seguinte:

DELETE FROM tabela1 [FROM tabela2]
WHERE <condição de localização>;

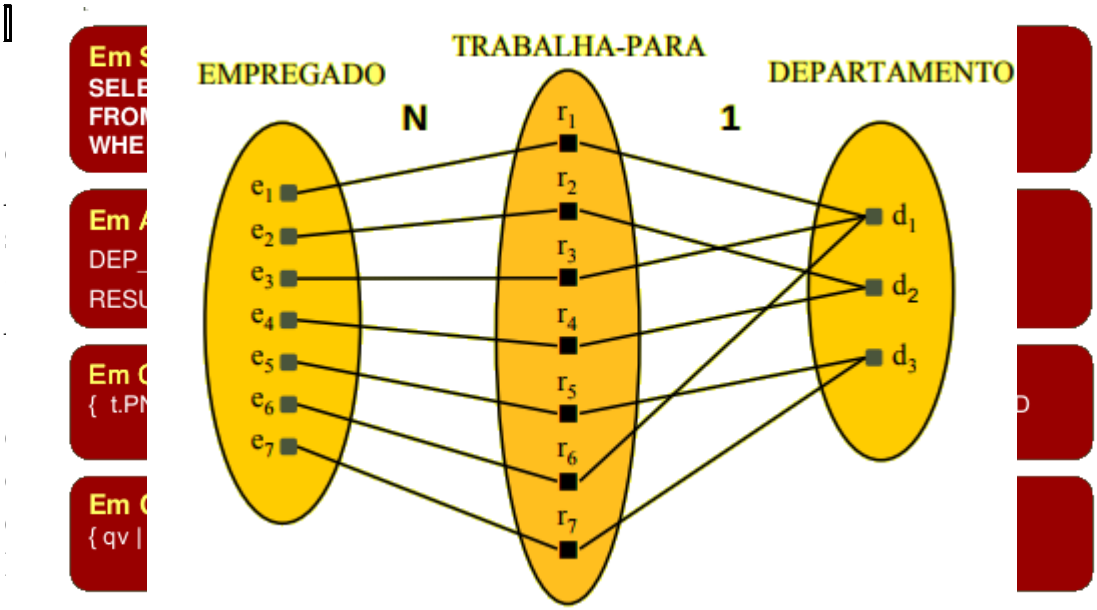
• O comando **SELECT** é utilizado para realizar consultas. Sua forma geral é a seguinte:

SELECT [DISTINCT | ALL] <lista de atributos>

FROM <lista de tabelas>
[WHERE <condições>]
[GROUP BY atributo]
[HAVING <condições>]
[ORDER BY atributo [ASC | DESC]];

O comando SELECT é possívelmente o mais útil, pois permite que especifiquemos com precisão um conjunto que desejamos obter. O comando SELECT gera uma tabela como resultado da consulta (esta sendo temporária, se não for guardada de alguma forma).
O comando SELECT possui algumas especificidades, como visto acima. A lista de atributos pode ser substituída por "*", que sinaliza a presença de todos os atributos. A palavra ALL é o *default* e indica que linhas (tuplas) duplicadas serão mantidas. Alternativamente, podemos explicitar a palavra DISTINCT para remover linhas duplicadas. A palavra FROM precede de onde os dados necessários serão retirados, enquanto a palavra WHERE precede as condições de seleção dos resultados.

• Considere o seguinte exemplo: desejamos obter o nome e o endereço dos empregados que trabalham para o departamento de ‘Pesquisa'. Observe as diferentes formas de se realizar esta consulta:



SELECT EMPREGADO.PNOME, PROJETO.PNOME
FROM PROJETO, DEPARTAMENTO, EMPREGADO
WHERE DNUM=DNUMERO AND GERNSS=SSN AND
PLOCALIZACAO='Stafford‘;

• Aliases são úteis para melhorar a clareza de consultas, e podem ser utilizadas para realizar consultas que referenciam duas vezes a mesma tabela. Aliases são utilizados de duas formas distintas, uma delas mais clara, pois se utiliza da palavra-chave **AS**.
Por exemplo:

```
SELECT E.PNOME, E.SNOME, S.PNOME, S.SNOME
FROM EMPREGADO E S
WHERE E.NSSSUPER=S.NSS;
```

=

```
SELECT E.PNOME, E.SNOME, S.PNOME, S.SNOME
FROM EMPREGADO AS E, EMPREGADO AS S
WHERE E.NSSSUPER=S.NSS
```

• A forma com que as colunas (os atributos) estão sendo escritos acima (*tabela.coluna*) é chamado de referenciar os atributos. Isto serve tanto para eliminar ambiguidades, quanto para deixar consultas mais claras.
Exemplo:

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;
```

• A seguir, trataremos de alguns aspectos mais pontuais da linguagem SQL. Note que aspectos pontuais que sejam mencionados a partir desde ponto nos slides, mas que tenham sido mencionados alguma vez anteriormente neste resumo, não serão repetidos.

• A ausência da clausula WHERE implica a ausência de condições, e equivale à expressão WHERE TRUE.
Exemplo:

```
SELECT NSS
FROM EMPREGADO;
```

Devolverá todas as linhas da tabela empregado, monstrando apenas a coluna NSS. Se mais de uma relação é especificada na cláusula FROM e não existir nenhuma condição de junção, então o resultado será o

produto cartesiano.
Exemplo:

```
SELECT NSS, DNOME
FROM EMPREGADO, DEPARTAMENTO;
```

É extremamente importante não negligenciar a especificação de qualquer condição de seleção ou de junção na cláusula WHERE; sob a pena de gerar resultados incorretos e volumosos.

• A linguagem SQL incorpora algumas das operações de Teoria de Conjuntos. A mais notável delas é **UNION**, porém a intersecção e subtração são incorporadas em algumas versões de SQL. As relações resultantes dessas operações são sempre conjunto de tuplas; tuplas duplicadas são eliminadas do resultado. Como é de se esperar, o conjunto de operações aplicam-se somente às relações que são compatíveis na união.
A SQL também possui operações sobre multiconjuntos (conjuntos que permitem repetição de elementos). Estas são denotadas por UNION ALL, **EXCEPT** ALL, e **INTERSECT** ALL.

• Uma consulta SELECT completa, chamada de consulta aninhada, pode ser especificada dentro da cláusula WHERE de uma outra consulta, chamada consulta externa.
Por exemplo:

```
SELECT PNAME, SNAME, ENDERECO
FROM EMPREGADO
WHERE DNUM IN ( SELECT DNUMERO
                FROM DEPARTAMENTO
                WHERE DNOME=‘Pesquisa' );
```

A consulta externa seleciona tuplas de empregados se o valor de seu DNUM pertencer ao resultado da consulta aninhada. Note que o operador **IN** é equivalente ao operador "pertence" da Teoria de Conjuntos.

• Existem formas mais complexas de realizar consultas aninhadas, mas essas não são pertinentes para este resumo. De qualquer forma, elas encontram-se presentes nos slides referentes ao assunto de SQL para consulta, se necessário.

• É possível utilizar um conjunto de valores explicitamente enumerado

na cláusula WHERE ao invés de utilizar uma consulta aninhada.
Exemplo:

```
SELECT DISTINCT ENSS
FROM TRABALHA-PARA
WHERE PNO IN (1, 2, 3);
```

• É possível realizar a operação de divisão, vista também na Álgebra Relacional, em SQL. Mais detalhes encontram-se nos slides sobre este assunto.

• A SQL permite que consultas verifiquem se um valor é nulo utilizando **IS** ou **IS NOT**.
Exemplo:

```
SELECT PNOME, SNOME
FROM EMPREGADO
WHERE NSSSUPER IS NULL
```

• É importante notar que todos os tipos de JOIN vistos anteriormente encontram-se presentes em SQL. As operações JOIN podem ser utilizadas tanto depois de uma consulta (após o FROM), como durante o processo de geração da consulta (ao lado do FROM). Vejamos os seguintes exemplos:

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

```
SELECT PNOME, SNOME, ENDERECO
FROM EMPREGADO JOIN DEPARTAMENTO ON DNUMERO=DNUM
WHERE DNOME=‘Pesquisa';
```

Todos os tipos de JOIN são extremamente úteis, e podem ser estudados em mais detalhes no link:

https://www.w3schools.com/sql/sql_join.asp

• A SQL possui as seguintes funções agregadas: COUNT, SUM, MAX, MIN e AVG.
Exemplo:

```
SELECT MAX(SALARIO), MIN(SALARIO), AVG(SALARIO)
FROM EMPREGADO;
```

• Como na extensão da Álgebra Relacional, a SQL tem uma cláusula **GROUP BY** para especificar os atributos de agrupamento, que devem aparecer na cláusula SELECT. Existe um bom exemplo no link mencionado acima.

• Algumas vezes queremos recuperar os valores das funções agregadas que satisfaçam a certas condições. A cláusula **HAVING** é usada para especificar essa condição.
Por exemplo:

```
SELECT PNUMERO, PNOME, COUNT(*)
FROM PROJETO, TRABALHA-PARA
WHERE PNUMERO=PNO
GROUP BY PNUMERO, PNOME
HAVING COUNT(*) > 2;
```

• É possível realizar a comparação de substrings e operações aritméticas em SQL. Ambas operações são extremamente úteis, e seus detalhes encontram-se nos slides sobre SQL.

• A cláusula **ORDER BY** é usada para ordenar tuplas resultantes de uma consulta com base nos valores de alguns atributos.
Exemplo:

```
SELECT D.DNOME, E.SNOME, E.PNOME, P.PNOME
FROM DEPARTAMENTO D, EMPREGADO E, TRABALHA-EM W, PROJETO P
WHERE D.DNUMERO=E.DNUM AND E.NSS=W.ENSS AND W.PNO=P.PNUMERO
ORDER BY D.DNOME, E.SNOME;
```

• A ordem padrão é ascendente, mas podemos utilizar a palavra-chave **DESC** para especificar que queremos a ordem descendente. A palavra-chave **ASC** pode ser usada para especificar explicitamente a ordem ascendente.
Exemplo:

```
SELECT D.DNOME, E.SNOME, E.PNOME, P.PNOME
FROM DEPARTAMENTO D, EMPREGADO E, TRABALHA-EM W, PROJETO P
WHERE D.DNUMERO=E.DNUM AND E.NSS=W.ENSS AND W.PNO=P.PNUMERO
ORDER BY D.DNOME ASC, E.SNOME ASC;
```

• Por fim, falemos de **Dependências Funcionais** e **Normalização de Base de Dados Relacionais**.

• **Dependências Funcionais** são restrições ao conjunto de relações válidas. Elas permitem expressar determinados fatos em banco de dados relativos ao empreendimento que se deseja modelar, e também servem para medir formalmente a qualidade do projeto relacional.

• Pode-se dizer, também, que as Dependências Formais são restrições que são derivadas do significado e do inter-relacionamento dos dados de atributos.

• A dependência funcional (DF) é um dos conceitos fundamentais no desenho dos modelos de dados relacionais. A dependência funcional é uma associação que se estabelece entre dois ou mais atributos duma relação e define-se do seguinte modo: Se **A** e **B** são atributos, ou conjuntos de atributos, da relação **R**, diz-se que **B** é funcionalmente dependente de **A** se cada um dos valores de **A** em **R** tem associado a si um e um só valor de **B** em **R**; a DF tem a notação: **A** -› **B**.
Em outras palavras, **A** deve ter um valor único (diferente) para cada **B**.

• O atributo **A**, no exemplo acima, é chamado de **atributo determinante**, enquanto **B** é chamado de **atributo dependente**.

• Escrever X -› Y determina que, se duas tuplas tiverem o mesmo valor para X, elas devem ter o mesmo valor para Y. Ou seja: para quaisquer tuplas t_1 e t_2 de r(R): Se $t_1[X] = t_2[X]$, então $t_1[Y] = t_2[Y]$.
Além disso, se K é uma chave de R, então K determina funcionalmente todos os atributos de R (uma vez que nós nunca teremos duas tuplas distintas com $t_1[K] = t_2[K]$).
Observe os seguinte exemplos:

- O número do seguro social determina o nome do empregado, logo:

NSS -› ENOME

- O número do projeto determina o nome do projeto e a sua localização, logo:

PNUMERO -› { PNOME, PLOCALIZACAO }

• Existem um conjunto de regras de inferência para este tipo de relação, que podem ser conferidas nos slides referentes a este assunto (https://www.ime.usp.br/~jef/bd09.pdf).

• Normalização de Banco de Dados é um conjunto de regras que visa, principalmente, a organização de um projeto de banco de dados para

reduzir a redundância de dados, aumentar a integridade de dados e o desempenho. Para normalizar o banco de dados, deve-se examinar as colunas (atributos) de uma entidade e as relações entre entidades (tabelas), com o objetivo de se evitar anomalias observadas na inclusão, exclusão e alteração de registros.

• Pode-se dizer que o objetivo principal da normalização de um banco de dados é decompor relações consideradas "ruins" ou inadequadas de acordo com padrões de bancos de dados estabelecidos. As **Formas Normais** são, então, indicativos do nível de qualidade de uma relação. Estudemos algumas delas em seguida.

• Antes de tudo, é necessário definir o conceito de **atributo primo**. Um atributo primo (ou primário) é um atributo que é membro de alguma chave-candidata. Alternativamente, um atributo não-primo é o oposto de um atributo primo.
É interessante notar que, se uma tabela tiver suas chaves com tamanho 1 (ou seja, apenas um atributo por chave-candidata), um atributo não-primo também é um atributo não-chave, e vice-versa.

• **Primeira Forma Normal** (ou **1FN**). Nesta forma os atributos precisam ser atômicos, o que significa que as tabelas não podem ter atributos possuindo mais de um valor (atributos multivalorados).

Exemplo: CLIENTE = {ID + ENDEREÇO + *TELEFONES*}. Porém, uma pessoa poderá ter mais de um número de telefone, sendo assim o atributo "TELEFONES" é multivalorado. Para normalizar, é necessário:

- Identificar a chave primária e também a coluna que possui dados repetidos (nesse exemplo "TELEFONES") e removê-los;
- Construir uma outra tabela com o atributo em questão, no caso "TELEFONES". Mas não se esquecendo de fazer uma relação entre as duas tabelas: CLIENTE = {ID + ENDEREÇO} e TELEFONE (nova tabela) = {CLIENTE_ID (chave estrangeira) + TELEFONE}.

• **Segunda Forma Normal** (ou **2FN**). Primeiramente, para estar na 2FN é preciso estar também na 1FN. A 2FN define que os atributos normais, ou seja, os não-primos, devem depender unicamente da chave primária da tabela. Assim, as colunas da tabela que não são dependentes dessa chave devem ser removidas da tabela principal e uma nova tabela deve ser criada se utilizando desses dados.

Por exemplo: PROFESSOR_CURSO = {ID_PROF + ID_CURSO + SALARIO + *DESCRICAO_CURSO*} Como podemos observar, o atributo "DESCRICAO_CURSO" não depende unicamente da chave primária "ID_PROF", mas sim somente da chave "ID_CURSO". Para normalizar, é necessário:

- Identificar os dados não dependentes da chave primária (nesse exemplo "DESCRICAO_CURSO") e removê-los;
- Construir uma nova tabela com os dados em questão: PROFESSOR_CURSO = {ID_PROF + ID_CURSO + SALARIO} e CURSOS (nova tabela) = {ID_CURSO + DESCRICAO_CURSO}.

• Terceira Forma Normal (ou 3FN). Assim como para estar na 2FN é preciso estar na 1FN, para estar na 3FN é preciso estar também na 2FN. A 3FN define que todos os atributos dessa tabela devem ser **funcionalmente independentes** uns dos outros, ao mesmo tempo que devem ser dependentes exclusivamente da chave primária da tabela. A 3NF foi projetada para melhorar o desempenho de processamento dos banco de dados e minimizar os custos de armazenamento.

Por exemplo: FUNCIONARIO = {ID + NOME + VALOR_SALARIO + *VALOR_FGTS*}. Como sabemos o valor do FGTS é proporcional ao salário, logo o atributo normal "VALOR_FGTS" é dependente do também atributo normal "VALOR_SALARIO". Para normalizar, é necessário:

- Identificar os dados dependentes de outros (nesse exemplo "VALOR_FGTS");
- Removê-los da tabela. Esses atributos poderiam ser definitivamente excluídos -- e deixando para a camada de negócio a responsabilidade pelo seu cálculo -- ou até ser movidos para uma nova tabela e referenciar a principal ("FUNCIONARIO").

Exclarecimentos Adicionais

• O professor deixou algumas questões como indicativos do que pode cair na prova. Tendo em vista este fato, o foco do estudo voltou-se mais para os temas envolvidos nestas questões.
Alguns tópicos que foram abordados com pouco detalhe, por exemplo,

receberão atenção adicional nesta seção.

• Utilizaremos a referência do site:

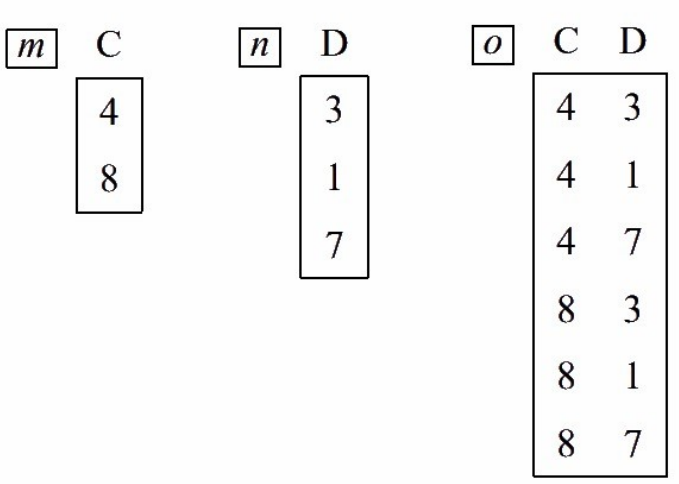
https://www2.cs.arizona.edu/~mccann/research/divpresentation.pdf

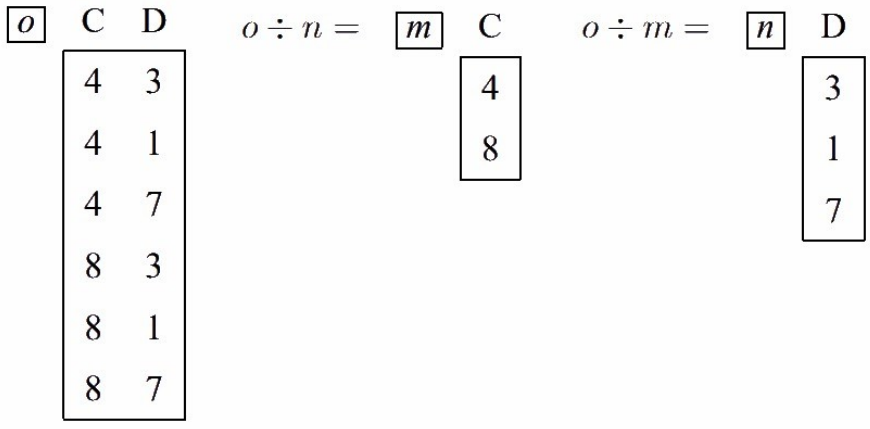
para facilitarmos a explicação do conceito de divisão relacional.

• A divisão relacional é definida usando os operadores básicos: π, X e –. O processo da divisão é baseado em encontrar valores que <u>não</u> são parte da resposta final.
É normalmente difícil expressar a divisão em SQL.

• A divisão identifica os valores de atributos de uma relação que sejam pareados com <u>todos</u> os valores de uma outra relação.

• A divisão aritimética está para a multiplicação, assim como a divisão relacional está para o Produto Cartesiano.
Observe o seguinte exemplo:





• Fugindo um pouco do tom formal, existe um formato para resolver exercícios que envolvem divisão. Observe a forma de realizar divisões relacionais exibido acima, primeiramente, e perceba que ela é apenas um Produto Cartesiano invertido.
Os exercícios que pedem para encontrar "todos" de algo que atendam a uma especificação são exercícios cuja resolução envolve o uso de divisão relacional.
O atributo do qual você deseja "todos" estará em uma tabela, e o requerimento será feito sobre um atributo de outra tabela. Haverá também um atributo que liga as duas tabelas.
Para resolver o exercício, separe primeiramente em uma tabela o atributo desejado e o atributo de ligação, e em outra o atributo de ligação, com a restrição já aplicada sobre ele. Então, efetue a divisão do segundo pelo primeiro para obter a resposta desejada.

• O segundo exercício possui o seguinte enunciado: "*Por que as regras de mapeamento de Modelo Conceitual para Modelo Lógico garantem, pelo menos, a Terceira Forma Normal?*".
Primeiramente, lembremos que a 3FN (Terceira Forma Normal) implica termos as seguintes restrições em nossas tabelas: os atributos são atômicos, e todos os artributos são funcionalmente independentes uns dos outros, ao mesmo tempo que devem depender exclusivamente da chave primária da tabela (isto implica em haver apenas uma chave na tabela: a chave primária em si).
Agora, os passos para o mapeamento de Modelo Conceitual (Modelo Entidade-Relacional) para Modelo Lógico (Modelo Relacional), garantem o 3FN pois, primeiramente, quebram todos os atributos multivalorados em atributos simples, criando novas relações quando necessário, ao mesmo tempo que garante apenas uma chave por relação, criando relações novas quando necessário.

• O terceiro exercício possui o seguinte enunciado: "*Uma chave pode ser uma superchave? E uma superchave, pode ser uma chave? Justifique dando evidências.*". A primeira parte da pergunta é trivial. Uma chave é, por definição, uma superchave mínima. Uma superchave, por outro lado, só será uma chave se for mínima.

• O quarto exercício possui o seguinte enunciado: "*Dada uma relação*

ternária com cardinalidade variando de 1-1-1 até M-N-P, analise as depedendências funcionais.".

• Para saber se uma dependência pode ser mantida, basta observar os valores do atributo determinante, e ver a quais valores do atributo dependente eles se ligam. Se um valor do atributo determinante "levar" a dois "lugares" diferentes, então a dependência é inválida. Ou seja, se um atributo determinante possuir dois valores iguais, e os valores correspondentes no atributo dependente forem diferentes entre si, então a dependência é inválida.

• Para ilustrar o dito acima, observe o exercício 4, da página 87 da apostila (https://www.ime.usp.br/~jef/apostila.pdf). Neste exercício, vemos que a dependência A -› B é inválida, pois o valor "10", em A, leva tanto a "b1" quanto a "b2", em B. Por outro lado, a relação B -› C é completamente válida pois, por mais que tenhamos valores repetidos em B ("b1" e "b3"), eles levam sempre para os mesmos valores em C ("b1" leva sempre a "c1", e "b3" leva sempre a "c4").

• Note que a resposta do item "b" do exercício 4, mencionado acima, é afirmativa. Por mais que um único atributo não possa ser chave, por que todos possuem elementos repetidos, a combinação de atributos {A, B} pode ser um chave (ou seja, é uma chave-candidata). Isso se da pois não existem pares de elementos iguais, e sabemos que esta é uma superchave mínima pois dissemos que nenhuma única coluna pode ser uma superchave.
A solução deste exercício foi destacada para lembrar do conceito de superchave e chave-candidata.