

Módulo III - Projeto

Boas Práticas com Bancos de Dados

- 1) Boas práticas em SQL para desenvolvedores - 1**
- 2) Estudo de Índices em Geral e no PostgreSQL - 9**
- 3) Modelagem de Dados - 56**
- 4) Projeto de Bancos de Dados (Arquitetura) - 93**

1) Boas práticas em SQL para desenvolvedores

Escrito por juliano mmartins em dezembro 19, 2007

Boas práticas em SQL para desenvolvedores Quem nunca ouviu alguém reclamar: "o sistema está lento hoje!!!"? Nestes relatos de degradação de desempenho, frequentemente levantamos que esta degradação é decorrente de instruções SQL mal estruturadas ou ainda banco de dados mal planejado, o que, num efeito cascata, só é sentido conforme o sistema vai sendo utilizado, as tabelas sendo povoadas, etc. O SGDB começa a exigir muito processamento, memória e a gerar gargalos na rede, causando assim, efeitos no desempenho da aplicação e na rede!

Outro fato é que atualmente, boa parte dos desenvolvedores desenvolvem código SQL sem ter muito conhecimento sobre fundamentos de banco de dados. Tal falta de conhecimento gera a produção de código ineficiente e com baixa performance. É comum ver equipes de desenvolvimento que não tem um DBA, ficando assim, o desenvolvedor com a tarefa de criar um banco de dados.

Com estes problemas em mente, resolvi criar este post com algumas dicas para que os desenvolvedores tenham algum conteúdo básico e rápido e melhorem suas instruções SQL e a criação de banco de dados.

Tentarei ser o mais genérico possível para conseguir cobrir os bancos de dados mais utilizados atualmente (DB2, Oracle, MySQL, Postgres, etc), porém, algumas dicas podem não ser aplicáveis a todos os bancos.

É importante lembrar que praticamente todas as dicas são "debatíveis" em diferentes cenários, portanto, fiquem a vontade para comentar.

Vamos lá:

- 1- Normalize seu banco de dados. Isso quer dizer basicamente, divida tabelas grandes em tabelas menores e remova redundância, ou seja, que dados estejam duplicados sem real necessidade.
2. Em instruções select, evite usar "*". Seja restritivo, traga somente os campos realmente necessários, isso alivia a memória do servidor, diminui tráfego na rede, etc. Algumas pessoas defendem que também não devem ser criados determinados campos, por exemplo, você tem A + B

e pretende guardar C onde $C = A + B$. Ao invés de criar uma coluna para armazenar C, passe a utilizar "select (A+B) AS C from tabela". Esse pensamento pode não ser necessariamente válido para Dws. Vamos supor que você tem uma tabela enorme com dados sobre salário por ano. Você pode armazenar o percentual de ajuste e o valor ajustado, fazendo aí uma "desnormalização" para que o comando que vai recuperar os valores do salário não "frite" a CPU forçando-a a fazer muitas contas e perdendo muito desempenho.

3. Existe muito debate sobre essa: Não utilize seu banco de dados para armazenar imagens, ao invés disso, armazene a URL. Vale lembrar que os bancos de dados atuais estão cada vez mais aprimorados na manipulação de imagens, portanto, aqui abre-se espaço para uma enorme discussão, benchmarck, etc.

4. Para obter maior performance, utilize chaves primárias numéricas ou ainda campos pequenos nas chaves.

5. Utilizando-se stored procedures e functions ao invés de escrever código no seu programa, vai garantir maior desempenho e segurança para seu sistema como um todo.

6. Utilize o conceito de transações. Vários problemas podem ocorrer, por exemplo, a rede cair. Aprenda sobre commit e rollback.

7. Use sempre o tipo de dados correto para armazenar os dados. Por exemplo, não armazene sexo, que vai ser M ou F em um campo Varchar, use apenas 1 caractere: CHAR(1).

8. Evite o uso de cursores, eles consomem muito tempo já que "navegam" registro por registro.

9. Otimize a clausula WHERE: Simples exemplos são o uso de ">" e ">=". Se você quer retornar todas as pessoas de uma tabela que tem idade "> 3", use no where ">=4", dessa forma o banco não fará o scan das páginas até encontrar o 3. Esse princípio é válido desde que você tenha um índice na idade.

10. Quando possível, crie instruções SQL idênticas, pois no momento da execução de uma instrução, o banco compila a mesma e a preserva em memória, na próxima execução, não vai precisar compilar novamente. Uma ótima técnica para fazer isso é utilizar variáveis nas suas instruções ao invés de passar parametros para o banco.

11. Utilize os mecanismos do banco de dados para persistência: Primary Key, Foreign Key, etc são feitos e otimizados para isso.

12. Quando possível, trave (lock) uma tabela para executar alguma operação que vai demandar muito acesso a esta tabela, por exemplo, se você vai alterar a estrutura de uma tabela grande ou importar dados neste tabela (falando-se de tabelas realmente grandes).

13. Sempre utilize o nome das colunas em instruções SELECT, INSERT, UPDATE evitando utilizar "*".

14. Evite utilizar o operador "LIKE", ele pode facilmente fazer o desempenho de um banco de dados ruir!

15- Utilize EXISTS ao invés de COUNT para verificar se existe um determinado registro em uma tabela. É comum ver desenvolvedores fazendo um "select count(X) from Y" para verificar se o COUNT é maior que 0. Utilizando-se EXISTS, o sgbd vai parar no primeiro registro encontrado, se utilizar count, o banco vai varrer toda a tabela.

16- Em joins de tipos de dados diferentes, o SGBD vai ter que converter o tipo hierarquicamente inferior para o outro tipo a fim de efetuar a comparação, e assim, não vai utilizar um índice caso exista.

17. Sobre índices:

- * Não crie índices em campos que são alterados constantemente, pois o banco vai ter que atualizar toda sua estrutura de índices em qualquer update feito no campo.

- * Prefira criar os índices em chaves primárias e estrangeiras, e em suas queries, utilize estes índices.

- * Não tenha muitos índices em seu banco, só o necessário: uma breve explicação sobre o motivo disso, é que o banco de dados mantém toda uma estrutura para gerenciar os índices, então, quanto mais índices, mais tempo/processamento o SGBD vai utilizar para a manutenção dos mesmos.

- * No momento de importação/importação de uma base de dados, não exporte/importe índices. Isso vai consumir mais tempo/processamento. Também pode-se não fazer backup de índices.

- * Não crie índices em colunas que possuem pouca variação de valores.

18. Entenda os fundamentos de banco de dados. Uma ótima leitura é o livro "Sistema de Banco de Dados", de Abraham Silberschatz, Henry F. Korth e S. Sudarshan lançado no Brasil pela Editora Elsevier.

Práticas Recomendadas - Mauro Pichiliani

Recentemente eu estava revisando o conteúdo do M.O.C. do curso 2073A (Programming a Microsoft SQL Server 2000 Database) e notei que a cada final de capítulo (chamado de módulos) são apresentadas várias práticas recomendadas pela Microsoft para quem utiliza o SQL Server. Como estas práticas são importantes e recomendadas pela própria Microsoft, esta semana resolvi reproduzir algumas destas idéias, de acordo com cada módulo.

Módulo 2: Overview Of Programming SQL Server

- * Mantenha a lógica de negócio no Servidor , como Stored Procedures
- * Use sintaxe SQL ANSI
- * Escolha uma convenção de nomes apropriada
- * Salve instruções como Scripts and comente-os bem
- * Formate sua instruções Transact-SQL para ser legível a outros

Módulo 3: Creating and Managing Databases

- * Faça um Backup do banco de dados Master
- * Especifique um tamanho máximo do arquivo
- * Especifique um incremento de autocrescimento grande
- * Mude o grupo de arquivos (filegroup) padrão

Módulo 4: Creating Data Types and Tables

- * Especifique tipos de dados corretos e o de tamanhos apropriados
- * Sempre especifique características de coluna no CREATE TABLE

- * Gere Scripts para recriar Bancos de Dados e Objetos de Bancos de Dados

Módulo 5: Implementing Data Integrity

- * Use constraints por quê elas são compatível com o padrão ANSI
- * Use integridade referencial em cascata ao invés de Triggers

Módulo 6: Planning Indexes

- * Crie índices em colunas que fazem join em tabelas
- * Use índices para reforçar unicidade
- * Evite chaves cluster grandes
- * Considere a utilização de índices clustered para suportar ordenação e pesquisas por intervalo
- * Crie índices que suportam argumentos de pesquisa

Módulo 7: Creating and Maintaining Indexes

- * Use a opção FILLFACTOR para otimizar o desempenho
- * Use a opção DROP_EXISTING para a manutenção dos índices
- * Execute DBCC SHOWCONTIG para medir a fragmentação
- * Permita ao SQL Server criar e atualizar estatísticas automaticamente
- * Considere a criação de estatísticas em colunas não indexadas para permitir planos de execução mais eficientes

Módulo 8: Implementing Views

- * Utiliza uma convenção de nomes padrão
- * dbo deve ser o dono de todas as views
- * Verifique a dependência de objetos antes de se apagar um objeto
- * Nunca apague as linhas na tabela syscomments
- * Cuidadosamente avalie a criação de views baseada em views

Módulo 9: Implementing Stored Procedures

- * Verifique os parâmetros de entrada (input)
- * Faça cada Stored Procedure efetuar uma única tarefa
- * Valide dados antes de iniciar uma transação
- * Use as mesmas configurações de conexão para todas as Stored Procedures
- * Use WITH ENCRYPTION para esconder o texto de Stored Procedures

Módulo 10: Implementing User-defined Functions

- * Utilize funções escalares complexas em pequenos conjuntos de resultados
- * Utilize multi-statement functions ao invés de Stored Procedures que retornam tabelas
- * Use in-line functions para criar views parametrizadas
- * Use in-line functions para filtrar views indexadas

Módulo 11: Implementing Triggers

- * Use Triggers somente quando necessário
- * Mantenha as instruções de definição de triggers o mais simples possível
- * Inclua instruções de checagem de término na definição de triggers recursivos
- * Procure minimizar o uso da instrução ROLLBACK em triggers

Módulo 12: Programming Across Multiple Servers

- * Use servidores linkados para acessos de dados remotos freqüentes
- * Use consultar Ad Hoc para acessos de dados remotos não tão freqüentes
- * Configure servidores linkados para executar Stored Procedures remotamente ou para executar consultas distribuídas
- * Restrinja acesso a servidores linkados
- * Evite duplicar contas de login em servidores diferentes
- * Selecione a coluna apropriada para definir a partição

Módulo 13: Optimizing Query Performance

- * Utilize o Query Governor para prevenir consultas de longa duração de consumir recursos do sistema

- * Possua um bom conhecimento dos dados e como as consultas acessam os dados
- * Crie índices que cobrem as consultas usadas mais freqüentemente
- * Estabeleça estratégias de indexação para uma consulta ou múltiplas consultas
- * Evite sobre-escrever o otimizador de consultas

Módulo 14: Analyzing Queries

- * Defina um índice em uma coluna de alta seletividade
- * Garanta que índices úteis existem para todas as colunas referenciadas no operador OR
- * Minimize o uso de joins HASH

Módulo 15: Managing Transaction and Locks

- * Mantenha transações curtas
- * Faça transações de modo a evitar deadlocks
- * Use os padrões do SQL Server para lock
- * Seja cuidadoso ao utilizar opções de lock

Documentação do DBA

Olá pessoal. Na coluna desta semana falarei um pouco sobre os principais documentos com os quais um DBA deve lidar. Estes documentos são úteis não apenas para ajudar a documentar aspectos técnicos, mas também para organizar e gerenciar o trabalho do DBA (*Database Administrator*).

Ao contrário do que alguns podem pensar, a documentação não é apenas um trabalho desnecessário que toma tempo e torna tudo mais lento. Ela é um item necessário e obrigatório em qualquer projeto, servindo a vários propósitos.

Em geral, é difícil encontrar profissionais, em particular DBAs, que trabalhem com muita documentação. Isso se deve tanto à questão cultural como a questão que envolve a disponibilidade de tempo para criar e manter a documentação atualizada. De qualquer maneira, neste artigo apresento os 10 principais documentos utilizados por quem trabalha tanto como DBA como desenvolvedor.

1) MER (Modelo Entidade Relacionamento)

O MER (Modelo Entidade Relacionamento), também conhecido apenas como modelo de dados ou diagrama de entidade-relacionamento, é a principal documentação de um banco de dados. Neste diagrama são relacionadas as principais entidades (tabelas) e seus relacionamentos, além de alguns detalhes a respeito das entidades, como nome das colunas nas tabelas, tipos de dados e *constraints*. Geralmente utiliza-se uma ferramenta CASE para modelagem deste diagrama, sendo o ER-Win um dos softwares mais utilizados para elaborar este diagrama. Independente do software utilizado é imprescindível que o DBA conte com ao menos um MER para cada uma das bases que ele administra.

2) Padrões de Variáveis (Tabelas, colunas etc) e Documentação

Uma das boas práticas de desenvolvimento é contar com padrões. Saber colocar nomes significativos e explicativos em funções, variáveis, classes etc está se tornando cada vez mais um pré-requisito para quem trabalha com desenvolvimento. No que tange ao banco de dados, a ideia não é diferente: possuir um padrão de nomes para tabelas, colunas, *constraints* e objetos de bancos de dados é muito importante. Aqui a ideia não é apenas possuir um padrão e utilizá-lo largamente, mas possuir um padrão eficaz que seja fácil de ser utilizado, além de fazer sentido no contexto de banco de dados. Para formalizar a adoção de um padrão, recomenda-se montar um documento explicando todos os detalhes deste padrão como, por exemplo, se o padrão é baseado na notação húngara, se os nomes devem ser em português ou se há um limite no tamanho da coluna.

3) Capacity Plan

A necessidade de prever recursos de hardware é uma das grandes responsabilidades de um DBA. Além de economizar recursos, a previsão mostra que há um controle não apenas para os recursos que estão sendo utilizados, mas também para os recursos que podem ser necessários no futuro. Para ajudar nesta previsão, o DBA deve ser responsável pela elaboração de um documento chamado *Capacity Plan*, ou Plano de Capacidade. Este documento deve listar as necessidades de espaço de armazenamento, utilização de CPU, largura de banda e outros requisitos técnicos que possam impactar o banco de dados. Com certeza, este é um documento que envolve muitos aspectos e deve ser elaborado com cuidado. Por questões práticas, muitas vezes é necessário fazer uso de estimativas e tendências ao invés de contar com informações precisas. Deste modo, o documento não precisa estar 100% correto, mas deve conter uma boa base e previsão dos principais recursos computacionais relacionados ao banco de dados.

4) Dicionário de dados

O Dicionário de dados é um documento que complementa o MER. Este documento deve conter mais detalhes a respeito das tabelas e seus relacionamentos. Por exemplo, além de listar todas as colunas de uma tabela, o documento deve fornecer também uma pequena descrição do significado desta coluna, quais são os valores possíveis, a quantidade típica de valores armazenados e quais constraints agem sobre esta coluna. Além das informações sobre colunas, este documento apresenta o nome dos objetos que dependem da tabela, como *stored procedure*, *triggers*, *views*, funções etc, e suas respectivas funções, além dos parâmetros necessários e o que é retornado. É importante notar que este documento deve sempre estar alinhado e atualizado com a base de dados atual, para evitar desencontros e desentendimentos.

5) Política de segurança

O documento contendo a política de segurança é um documento não-técnico que envolve os procedimentos, responsabilidades e atribuições relacionadas tanto à segurança das informações como do acesso à elas. Geralmente este documento contém uma política de usuários e senhas, que especificam várias regras, como as definidas abaixo:

- Troca de senha a cada três meses;
- Desabilitar as contas padrão;
- Forçar senhas com letras, números e caracteres especiais que tenham um tamanho mínimo de 10 posições;

Outras políticas gerais de senha, como o cancelamento após um algumas tentativas e horários definidos para certos usuários, também deve constar neste documento, sempre tendo em mente a utilização de sistemas e bancos de dados.

6) N.D.A (*non-disclosure agreement*) - Compromisso de sigilo

Imaginem a seguinte situação:

Somos responsáveis por uma base de dados que deve ser integrada com um sistema externo à empresa. Para discutir os detalhes desta integração, uma reunião é marcada com a equipe externa à empresa que desenvolve o sistema. Durante esta reunião são apresentadas informações sigilosas da empresa que trabalhamos, com o objetivo de discutir os aspectos da integração.

Vamos supor que na situação apresentada acima os profissionais da equipe externa hajam da má fé e utilizem as informações fornecidas para seu próprio benefício, seja comercialmente ou não. Este tipo de situação pode gerar diversos problemas, podendo chegar ao ponto onde a equipe que agiu de

má fé ser acusada de roubo.

Para e proteger de situações como estas, é comum fazer uso de um documento chamado NDA (*non-disclosure agreement*), também conhecido como compromisso de sigilo. Este é o tipo de documento que protege todo mundo: tanto quem assina como quem solicita a assinatura. Em termos práticos, que assina compromete-se a não revelar nenhum detalhe da informação que lhe vai ser comunicada sob pena de ser alvo de procedimento legal.

7) S.L.A. (*Service Level Agreement*) - Acordo de nível de service (ANS)

O SLA, também conhecido como Acordo de nível de serviço - ANS, é um acordo entre a área prestadora de serviços e seus clientes. Este acordo deve deixar claro quais serviços estão sendo oferecidos (serviços específicos) e o nível de cada serviço (horas de funcionamento, *downtime*, horário do suporte etc). Geralmente este acordo é colocado na forma de um contrato que deve ser assinado na contratação do serviço. Para banco de dados, em particular, pode-se utilizar um SLA interno, onde o DBA se compromete a dar algum tipo de retorno (*feedback*) ao solicitante. Notem que este retorno não quer dizer, necessariamente, a resolução do problema ou o conserto, mas sim que o DBA está ciente da solicitação.

8) Diagrama de arquitetura

Atualmente é comum encontrar nas empresas diversos ambientes de bancos de dados. Estes ambientes são separados de acordo com a sua finalidade, isto é, seu principal objetivo. Por exemplo, é comum encontrar ambientes de desenvolvimento, onde os programadores/analistas executam diversos testes durante o processo de desenvolvimento, e ambientes de programação, onde os usuários finais trabalham com os dados reais dos sistemas.

Para documentar e organizar o gerenciamento destes ambientes, o DBA deve elaborar um diagrama de arquitetura, que indica, de forma gráfica, quais servidores pertencem ao ambiente de desenvolvimento e ao ambiente de produção, como eles estão localizados em relação aos usuários com informações sobre link, rede, zonas desmilitarizadas (DMZ), *firewalls*, roteadores, etc. Este tipo de diagrama contém informações relacionadas à estrutura arquitetural dos ambientes e é extremamente útil para quem não conhece a organização física e lógica dos componentes da rede e dos servidores. É importante lembrar que este documento pode ser flexível, ou seja, pode incluir detalhes específicos, como endereços I.P. e senhas, ou apresentar uma visão de alto nível, onde apenas os principais servidores são apresentados.

9) Estratégia de Backup

Todo DBA profissional deve possuir uma estratégia de backup adequada. Esta estratégia deve ser montada de acordo com as necessidades de recuperação e disponibilidade do sistema, ou seja, toda a estratégia de backup vai depender do quanto de *downtime* e tempo de recuperação é aceitável.

É importante documentar a estratégia de backup utilizada, tanto para oficializar este tipo de tarefa como para conscientizar os usuários a respeito do que pode ser recuperado, quando, sob quais condições e a qualidade do que foi recuperado. Geralmente este documento contém todos os bancos de dados envolvidos na estratégia, como o backup será realizado, qual a periodicidade, qual é o procedimento para recuperação, quem são os responsáveis e os recursos envolvidos. Por fim, é importante atualizar este documento conforme as necessidades de disponibilidade e recuperação mudam de acordo com o volume de informações manipuladas pelos sistemas.

10) Procedimento para controle de chamados ou O.S. (ordem de serviço)

Este último item não é exatamente um documento, mas sim um procedimento que deve ser adotado

para o controle de solicitações de serviço (ou chamados) ao DBA. Este tipo de controle evita problemas de comunicação entre quem solicitou e que realiza uma tarefa. Deixar este controle apenas a cargo do envio de e-mails é um primeiro passo, mas investir em um sistema para controlar o acesso às pessoas é algo fundamental, uma vez que este procedimento está diretamente relacionado com as regras definidas no SLA. Obviamente, este tipo de controle deve ser utilizado de forma sensata, pois existem diversos tipos de solicitações que podem exigir tratamentos diferentes, como apenas uma olhada no estado de um servidor. Mais uma vez, a idéia aqui é estabelecer um mecanismo de controle tanto para quem solicita a tarefa como para quem a executa.

Outros tipos de documentos são necessários para quem trabalha com desenvolvimento de sistemas. Atas de reunião, manuais de implementação de sistemas e *help-online* são apenas alguns exemplos disto. Em geral, o DBA é o profissional que menos tem que lidar com este tipo de documentação. Apesar disso, é importante que o profissional que trabalha com banco de dados, e também com desenvolvimento de sistemas, tenha consciência que esta documentação não necessariamente é burocracia, mas sim que ela é um artefato extremamente importante para todos os profissionais envolvidos.

Mauro Pichiliani no iMasters.

2) Estudo de Índices em Geral e no PostgreSQL

CREATE INDEX

Nome

CREATE INDEX -- cria um índice

Sinopse

```
CREATE [ UNIQUE ] INDEX nome_do_índice ON tabela [ USING método ]  
    ( { coluna | ( expressão ) } [ classe_de_operadores ] [, ... ] )  
    [ TABLESPACE espaço_de_tabelas ]  
    [ WHERE predicado ]
```

Descrição

O comando CREATE INDEX constrói o índice nome_do_índice na tabela especificada. Os índices são utilizados, principalmente, para melhorar o desempenho do banco de dados (embora a utilização não apropriada possa resultar em uma degradação de desempenho). [\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#)

Os campos chave para o índice são especificados como nomes de coluna ou, também, como expressões escritas entre parênteses. Podem ser especificados vários campos, se o método de índice suportar índices multicolunas.

O campo de um índice pode ser uma expressão computada a partir dos valores de uma ou mais colunas da linha da tabela. Esta funcionalidade pode ser utilizada para obter acesso rápido aos dados baseado em alguma transformação dos dados básicos. Por exemplo, um índice computado como upper(col) permite a cláusula WHERE upper(col) = 'JIM' utilizar um índice.

O PostgreSQL fornece os métodos de índice B-tree, R-tree, hash e GiST. O método de índice B-tree é uma implementação das *B-trees* de alta concorrência de Lehman-Yao [\[5\]](#). O método de índice R-tree implementa R-trees padrão utilizando o algoritmo de divisão quadrática (*quadratic split*) de Guttman [\[6\]](#). O método de índice hash é uma implementação do hash linear de Litwin [\[7\]](#) [\[8\]](#). Os usuários também podem definir seus próprios métodos de índice, mas é muito complicado.

Quando a cláusula WHERE está presente, é criado um *índice parcial*. Um índice parcial é um índice contendo entradas para apenas uma parte da tabela, geralmente uma parte mais útil para indexar do que o restante da tabela. Por exemplo, havendo uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados ocupam uma pequena parte da tabela, mas que é bastante usada, o desempenho pode ser melhorado criando um índice apenas para esta parte da tabela. Outra aplicação possível é utilizar a cláusula WHERE junto com UNIQUE para impor a unicidade de um subconjunto dos dados da tabela. Para obter informações adicionais deve ser consultada a [Seção 11.7](#).

A expressão utilizada na cláusula WHERE pode referenciar apenas as colunas da tabela subjacente, mas pode usar todas as colunas, e não apenas as que estão sendo indexadas. Atualmente não são permitidas subconsultas e expressões de agregação na cláusula WHERE. As mesmas restrições se aplicam aos campos do índice que são expressões.

Todas as funções e operadores utilizados na definição do índice devem ser "imutáveis" (*immutable*), ou seja, seus resultados devem depender somente de seus argumentos, e nunca de uma influência externa (como o conteúdo de outra tabela ou a hora atual). Esta restrição garante que o comportamento do índice é bem definido. Para utilizar uma função definida pelo usuário na expressão do índice ou na cláusula WHERE, a função deve ser marcada como IMMUTABLE na sua criação.

Parâmetros

UNIQUE

Faz o sistema verificar valores duplicados na tabela quando o índice é criado, se existirem dados, e toda vez que forem adicionados dados. A tentativa de inserir ou de atualizar dados que resultem em uma entrada duplicada gera um erro.

nome_do_índice

O nome do índice a ser criado. O nome do esquema não pode ser incluído aqui; o índice é sempre criado no mesmo esquema da tabela que este pertence.

tabela

O nome (opcionalmente qualificado pelo esquema) da tabela a ser indexada.

método

O nome do método de índice a ser utilizado. Pode ser escolhido entre btree, hash, rtree e gist. O método padrão é btree.

coluna

O nome de uma coluna da tabela.

expressão

Uma expressão baseada em uma ou mais colunas da tabela. Geralmente a expressão deve ser escrita entre parênteses, conforme mostrado na sintaxe. Entretanto, os parênteses podem ser omitidos se a expressão tiver a forma de uma chamada de função.

classe_de_operadores

O nome de uma classe de operadores. Veja os detalhes abaixo.

espaço_de_tabelas

O espaço de tabelas onde o índice será criado. Se não for especificado, será utilizado o [default_tablespace](#), ou o espaço de tabelas padrão do banco de dados se default_tablespace for uma cadeia de caracteres vazia.

predicado

A expressão de restrição para o índice parcial.

Observações

Consulte o [Capítulo 11](#) para obter informações sobre quando os índices podem ser utilizados, quando não são utilizados, e em quais situações particulares podem ser úteis.

Atualmente somente os métodos de índice B-tree e GiST suportam índices com mais de uma coluna. Por padrão podem ser especificadas até 32 colunas (este limite pode ser alterado na construção do PostgreSQL). Também atualmente somente B-tree suporta índices únicos.

Pode ser especificada uma *classe de operadores* para cada coluna de um índice. A classe de operadores identifica os operadores a serem utilizados pelo índice para esta coluna. Por exemplo, um índice B-tree sobre inteiros de quatro bytes usaria a classe `int4_ops`; esta classe de operadores inclui funções de comparação para inteiros de quatro bytes. Na prática, a classe de operadores padrão para o tipo de dado da coluna é normalmente suficiente. O ponto principal em haver classes de operadores é que, para alguns tipos de dado, pode haver mais de uma ordenação que faça sentido. Por exemplo, pode-se desejar classificar o tipo de dado do número complexo tanto pelo valor absoluto quanto pela parte real, o que pode ser feito definindo duas classes de operadores para o tipo de dado e, então, selecionando a classe apropriada na construção do índice. Mais informações sobre classes de operadores estão na [Seção 11.6](#) e na [Seção 31.14](#).

Para remover um índice deve ser utilizado o comando [DROP INDEX](#).

Por padrão não é utilizado índice para a cláusula `IS NULL`. A melhor forma para utilizar índice nestes casos é a criação de um índice parcial usando o predicado `IS NULL`.

Exemplos

Para criar um índice B-tree para a coluna `titulo` na tabela `filmes`:

```
CREATE UNIQUE INDEX unq_titulo ON filmes (titulo);
```

Para criar um índice para a coluna `codigo` da tabela `filmes` e fazer o índice residir no espaço de tabelas `espaco_indices`:

```
CREATE INDEX idx_codigo ON filmes(codigo) TABLESPACE espaco_indices;
```

Compatibilidade

O comando `CREATE INDEX` é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

Consulte também

[ALTER INDEX](#), [DROP INDEX](#)

Notas

- [1] Oracle — O comando CREATE INDEX é utilizado para criar um índice em: 1) Uma ou mais colunas de uma tabela, de uma tabela particionada, de uma tabela organizada pelo índice, de um agrupamento (*cluster* = objeto de esquema que contém dados de uma ou mais tabelas, todas tendo uma ou mais colunas em comum. O banco de dados Oracle armazena todas as linhas de todas as tabelas que compartilham a mesma chave de agrupamento juntas); 2) Um ou mais atributos de objeto tipado escalar de uma tabela ou de um agrupamento; 3) Uma tabela de armazenamento de tabela aninhada para indexar uma coluna de tabela aninhada. [Oracle® Database SQL Reference 10g Release 1 \(10.1\) Part Number B10759-01](#) (N. do T.)
- [2] Oracle — A *tabela organizada pelo índice* é um tipo especial de tabela que armazena as linhas da tabela dentro de segmento de índice. A tabela organizada pelo índice também pode ter um segmento de estouro (*overflow*) para armazenar as linhas que não cabem no segmento de índice original. [Hands-On Oracle Database 10g Express Edition for Linux — Steve Bobrowski](#), pág. 350 (N. do T.)
- [3] SQL Server — O comando CREATE INDEX cria um índice relacional em uma tabela ou visão especificada, ou um índice XML em uma tabela especificada. O índice pode ser criado antes de existir dado na tabela. Podem ser criados índices para tabelas e visões em outros bancos de dados especificando um nome de banco de dados qualificado. O argumento CLUSTERED cria um índice em que a ordem lógica dos valores da chave determina a ordem física das linhas correspondentes da tabela. A criação de um índice agrupado (*clustered*) único em uma visão melhora o desempenho, porque a visão é armazenada no banco de dados da mesma maneira que a tabela com um índice agrupado é armazenada. Podem ser criados índices em colunas calculadas. No SQL Server 2005 as colunas calculadas podem ter a propriedade PERSISTED. Isto significa que o Mecanismo de Banco de Dados armazena os valores calculados, e os atualiza quando as outras colunas das quais a coluna calculada depende são atualizadas. O Mecanismo de Banco de Dados utiliza os valores persistentes quando cria o índice para a coluna, e quando o índice é referenciado em uma consulta. [SQL Server 2005 Books Online — CREATE INDEX \(Transact-SQL\)](#) (N. do T.)
- [4] DB2 — O comando CREATE INDEX é utilizado para: Definir um índice em uma tabela do DB2 (o índice pode ser definido em dados XML ou dados relacionais); Criar uma especificação de índice (metadados que indicam ao otimizador que a tabela de origem possui um índice). A cláusula CLUSTER especifica que o índice é o índice agrupador da tabela. O fator de agrupamento de um índice agrupador é mantido ou melhorado dinamicamente à medida que os dados são inseridos na tabela associada, tentando inserir as novas linhas fisicamente próximas das linhas para as quais os valores chave deste índice estão no mesmo intervalo. [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)
- [5] Lehman, Yao 81 - Philip L. Lehman , s. Bing Yao, Efficient locking for concurrent operations on B-trees, ACM Transactions on Database Systems (TODS), v.6 n.4, p.650-670, Dec. 1981 (N. do T.)
- [6] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD

Conference 1984. (N. do T.)

[7] Litwin, W. Linear hashing: A new tool for file and table addressing. In Proceedings of the 6th Conference on Very Large Databases, (New York, 1980}, 212-223. (N. do T.)

[8] Witold Litwin: [Linear Hashing: A new Tool for File and Table Addressing](#) - Summary by: Steve Gribble and Armando Fox. (N. do T.)

Fonte: <http://pgdocptbr.sourceforge.net/pg80/sql-createindex.html>

Através da Criação de uma Tabela também podemos criar um índice
(é o que ocorre com mais frequência)

CREATE TABLE -- cria uma tabela

Sinopse

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
    { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ]
  [ restrição_de_coluna [ ... ] ]
    | restrição_de_tabela
    | LIKE tabela_ancestral [ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
)
[ INHERITS ( tabela_ancestral [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE espaço_de_tabelas ]
```

onde restrição_de_coluna é:

```
[ CONSTRAINT nome_da_restrição ]
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  PRIMARY KEY [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  CHECK (expressão) |
  REFERENCES tabela_referenciada [ ( coluna_referenciada ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE ação ] [ ON UPDATE ação ] }
  [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

e restrição_de_tabela é:

```
[ CONSTRAINT nome_da_restrição ]
{ UNIQUE ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE
espaço_de_tabelas ] |
  PRIMARY KEY ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE
espaço_de_tabelas ] |
  CHECK ( expressão ) |
  FOREIGN KEY ( nome_da_coluna [, ... ] )
    REFERENCES tabela_referenciada [ ( coluna_referenciada [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE ação ] [ ON UPDATE
```

```
ação ] }
```

```
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

CREATE INDEX

Name

CREATE INDEX -- define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ...] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Obs.: É bom observar a sintaxe das versões dos manuais existente. Veja que na versão 8.0 do manual em português, não existe o parâmetro CONCURRENTLY.

Description

CREATE INDEX constructs an index *name* on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

PostgreSQL provides the index methods B-tree, hash, GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the WHERE clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See [Section 11.8](#) for more discussion.

The expression used in the WHERE clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are

expressions.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or `WHERE` clause, remember to mark the function immutable when you create it.

Parameters

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

CONCURRENTLY

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done. There are several caveats to be aware of when using this option — see [Building Indexes Concurrently](#).

name

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

table

The name (possibly schema-qualified) of the table to be indexed.

method

The name of the index method to be used. Choices are `btree`, `hash`, `gist`, and `gin`. The default method is `btree`.

column

The name of a column of the table.

expression

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

opclass

The name of an operator class. See below for details.

ASC

Specifies ascending sort order (which is the default).

DESC

Specifies descending sort order.

NULLS FIRST

Specifies that nulls sort before non-nulls. This is the default when DESC is specified.

NULLS LAST

Specifies that nulls sort after non-nulls. This is the default when DESC is not specified.

storage_parameter

The name of an index-method-specific storage parameter. See below for details.

tablespace

The tablespace in which to create the index. If not specified, [default_tablespace](#) is consulted, or [temp_tablespaces](#) for indexes on temporary tables.

predicate

The constraint expression for a partial index.

Index Storage Parameters

The WITH clause can specify *storage parameters* for indexes. Each index method can have its own set of allowed storage parameters. The built-in index methods all accept a single parameter:

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency. B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

Building Indexes Concurrently

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the `CONCURRENTLY` option of `CREATE INDEX`. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially use the index to terminate. Thus this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation might slow other operations.

In a concurrent index build, the index is actually entered into the system catalogs in one transaction, then the two table scans occur in a second and third transaction. If a problem arises while scanning the table, such as a uniqueness violation in a unique index, the `CREATE INDEX` command will fail but leave behind an "invalid" index. This index will be ignored for querying purposes because it might be incomplete; however it will still consume update overhead. The `psql \d` command will mark such an index as `INVALID`:

```
postgres=# \d tab
           Table "public.tab"
  Column | Type   | Modifiers
-----+-----+-----
   col   | integer |
Indexes:
    "idx" btree (col) INVALID
```

The recommended recovery method in such cases is to drop the index and try again to perform `CREATE INDEX CONCURRENTLY`. (Another possibility is to rebuild the index with `REINDEX`. However, since `REINDEX` does not support concurrent builds, this option is unlikely to seem attractive.)

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the "invalid" index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur in parallel, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular `CREATE INDEX` command can be performed within a transaction block, but `CREATE INDEX`

CONCURRENTLY cannot.

Notes

See [Chapter 11](#) for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Currently, only the B-tree and GiST index methods support multicolumn indexes. Up to 32 fields can be specified by default. (This limit can be altered when building PostgreSQL.) Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. More information about operator classes is in [Section 11.9](#) and in [Section 34.14](#).

For index methods that support ordered scans (currently, only B-tree), the optional clauses `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` can be specified to reverse the normal sort direction of the index. Since an ordered index can be scanned either forward or backward, it is not normally useful to create a single-column `DESC` index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed-ordering query, such as `SELECT ... ORDER BY x ASC, y DESC`. The `NULLS` options are useful if you need to support "nulls sort low" behavior, rather than the default "nulls sort high", in queries that depend on indexes to avoid sorting steps.

Use [***DROP INDEX***](#) to remove an index.

Prior releases of PostgreSQL also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`, to simplify conversion of old databases to GiST.

Examples

To create a B-tree index on the column `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

To create an index with non-default sort ordering of nulls:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

To create an index without locking out writes to the table:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

[*ALTER INDEX*](#), [*DROP INDEX*](#)

Fonte: <http://www.postgresql.org/docs/8.3/static/sql-createindex.html>

Índices parciais

O *índice parcial* é um índice construído sobre um subconjunto da tabela; o subconjunto é definido por uma expressão condicional (chamada de *predicado* [1] do índice parcial). O índice contém entradas apenas para as linhas da tabela que satisfazem o predicado. [2]

O principal motivo para criar índices parciais é evitar a indexação de valores freqüentes. Como um comando procurando por um valor freqüente (um que apareça em mais que uma pequena percentagem de linhas da tabela) não utiliza o índice de qualquer forma, não faz sentido manter estas linhas no índice. Isto reduz o tamanho do índice, acelerando as consultas que utilizam este índice. Também acelera muitas operações de atualização da tabela, porque o índice não precisa ser atualizado em todos os casos. O [Exemplo 11-1](#) mostra uma aplicação possível desta idéia.

Exemplo 11-1. Definir um índice parcial excluindo valores freqüentes

Suponha que os registros de acesso ao servidor *Web* são armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente sobre acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Assumindo que exista uma tabela como esta:

```
CREATE TABLE tbl_registro_acesso (  
    url          varchar,  
    ip_cliente   inet,  
    ...  
);
```

Para criar um índice parcial adequado ao exemplo acima, deve ser utilizado um comando como:

```
CREATE INDEX idx_registro_acesso_ip_cliente ON tbl_registro_acesso (ip_cliente)
    WHERE NOT (ip_cliente > inet '192.168.100.0' AND ip_cliente < inet
'192.168.100.255');
```

Uma consulta típica que pode utilizar este índice é:

```
SELECT * FROM tbl_registro_acesso WHERE url = '/index.html' AND ip_cliente =
inet '212.78.10.32';
```

Uma consulta típica que não pode utilizar este índice é:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Deve ser observado que este tipo de índice parcial requer que os valores comuns sejam determinados a priori. Se a distribuição dos valores for inerente (devido à natureza da aplicação) e estática (não muda com o tempo) não é difícil, mas se os valores freqüentes se devem meramente à carga de dados coincidentes, pode ser necessário bastante trabalho de manutenção.

Outra possibilidade é excluir do índice os valores para os quais o perfil típico das consultas não tenha interesse, conforme mostrado no [Exemplo 11-2](#). Isto resulta nas mesmas vantagens mostradas acima, mas impede o acesso aos valores "que não interessam" por meio deste índice, mesmo se a varredura do índice for vantajosa neste caso. Obviamente, definir índice parcial para este tipo de cenário requer muito cuidado e experimentação.

Exemplo 11-2. Definir um índice parcial excluindo valores que não interessam

Se existir uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados representam uma pequena parte da tabela, mas são os mais acessados, é possível melhorar o desempenho criando um índice somente para os pedidos não faturados. O comando para criar o índice deve ser parecido com este:

```
CREATE INDEX idx_pedidos_nao_faturados ON pedidos (num_pedido)
    WHERE faturado is not true;
```

Uma possível consulta utilizando este índice é

```
SELECT * FROM pedidos WHERE faturado is not true AND num_pedido < 10000;
```

Entretanto, o índice também pode ser utilizado em consultas não envolvendo num_pedido como, por exemplo,

```
SELECT * FROM pedidos WHERE faturado is not true AND valor > 5000.00;
```

Embora não seja tão eficiente quanto seria um índice parcial na coluna valor, porque o sistema precisa percorrer o índice por inteiro, mesmo assim, havendo poucos pedidos não faturados, a utilização do índice parcial para localizar apenas os pedidos não faturados pode ser vantajosa.

Deve ser observado que a consulta abaixo não pode utilizar este índice:

```
SELECT * FROM pedidos WHERE num_pedido = 3501;
```

O pedido número 3501 pode estar entre os pedidos faturados e os não faturados.

O [Exemplo 11-2](#) também ilustra que a coluna indexada e a coluna utilizada no predicado não precisam corresponder. O PostgreSQL suporta índices parciais com predicados arbitrários, desde que somente estejam envolvidas colunas da tabela indexada. Entretanto, deve-se ter em mente que o

predicado deve corresponder às condições utilizadas nos comandos que supostamente vão se beneficiar do índice. Para ser preciso, o índice parcial somente pode ser utilizado em um comando se o sistema puder reconhecer que a condição WHERE do comando implica matematicamente no predicado do índice. O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer expressões equivalentes matematicamente escritas de forma diferente (Não seria apenas extremamente difícil criar este provador de teoremas geral, como este provavelmente também seria muito lento para poder ser usado na prática). O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, " $x < 1$ " implica " $x < 2$ "; senão, a condição do predicado deve corresponder exatamente a uma parte da condição WHERE da consulta, ou o índice não será reconhecido como utilizável.

Um terceiro uso possível para índices parciais não requer que o índice seja utilizado em nenhum comando. A idéia é criar um índice único sobre um subconjunto da tabela, como no [Exemplo 11-3](#), impondo a unicidade das linhas que satisfazem o predicado do índice, sem restringir as que não fazem parte.

Exemplo 11-3. Definir um índice único parcial

Suponha que exista uma tabela contendo perguntas e respostas. Deseja-se garantir que exista apenas uma resposta "correta" para uma dada pergunta, mas que possa haver qualquer número de respostas "incorretas". Abaixo está mostrada a forma de fazer:

```
CREATE TABLE tbl_teste
(
    pergunta    text,
    resposta    text,
    correto     bool
    ...
);

CREATE UNIQUE INDEX unq_resposta_correta ON tbl_teste (pergunta, correto)
WHERE correto;
```

Esta forma é particularmente eficiente quando existem poucas respostas corretas, e muitas incorretas.

Finalizando, também pode ser utilizado um índice parcial para mudar a escolha do plano de comando feito pelo sistema. Pode ocorrer que conjuntos de dados com distribuições peculiares façam o sistema utilizar um índice quando na realidade não deveria. Neste caso, o índice pode ser definido de modo que não esteja disponível para o comando com problema. Normalmente, o PostgreSQL realiza escolhas razoáveis com relação à utilização dos índices (por exemplo, evita-os ao buscar valores com muitas ocorrências, desta maneira o primeiro exemplo realmente economiza apenas o tamanho do índice, mas não é necessário para evitar a utilização do índice), e a escolha de um plano grosseiramente incorreto é motivo para um relatório de erro.

Deve-se ter em mente que a criação de um índice parcial indica que você sabe pelo menos tanto quanto o planejador de comandos sabe. Em particular, você sabe quando um índice poderá ser vantajoso. A formação deste conhecimento requer experiência e compreensão sobre como os índices funcionam no PostgreSQL. Na maioria dos casos, a vantagem de um índice parcial sobre um índice regular não é muita.

Podem ser obtidas informações adicionais sobre índices parciais em [The case for partial indexes](#) , [Partial indexing in POSTGRES: research project](#) e [Generalized Partial Indexes](#) .

Notas

- [1] predicado — especifica uma condição que pode ser avaliada para obter um resultado booleano. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [2] Os sistemas gerenciadores de banco de dados SQL Server 2000, Oracle 10g e DB2 8.1 não possuem suporte a índices parciais. [Comparison of relational database management systems](#) (N. do T.)

Examinar a utilização do índice

Embora no PostgreSQL os índices não necessitem de manutenção e ajuste, ainda assim é importante verificar quais índices são utilizados realmente pelos comandos executados no ambiente de produção. O exame da utilização de um índice por um determinado comando é feito por meio do comando [EXPLAIN](#); sua aplicação para esta finalidade está ilustrada na [Seção 13.1](#). Também é possível coletar estatísticas gerais sobre a utilização dos índices por um servidor em operação da maneira descrita na [Seção 23.2](#).

É difícil formular um procedimento genérico para determinar quais índices devem ser definidos. Existem vários casos típicos que foram mostrados nos exemplos das seções anteriores. Muita verificação experimental é necessária na maioria dos casos. O restante desta seção dá algumas dicas.

- O comando [ANALYZE](#) sempre deve ser executado primeiro. Este comando coleta estatísticas sobre a distribuição dos valores na tabela. Esta informação é necessária para estimar o número de linhas retornadas pela consulta, que é uma necessidade do planejador para atribuir custos dentro da realidade para cada plano de comando possível. Na ausência de estatísticas reais, são assumidos alguns valores padrão, quase sempre imprecisos. O exame da utilização do índice pelo aplicativo sem a execução prévia do comando ANALYZE é, portanto, uma causa perdida.
- Devem ser usados dados reais para a verificação experimental. O uso de dados de teste para definir índices diz quais índices são necessários para os dados de teste, e nada além disso. É especialmente fatal utilizar conjuntos de dados de teste muito pequenos. Enquanto selecionar 1.000 de cada 100.000 linhas pode ser um candidato para um índice, selecionar 1 de cada 100 linhas dificilmente será, porque as 100 linhas provavelmente cabem dentro de uma única página do disco, e não existe nenhum plano melhor que uma busca seqüencial em uma página do disco. Também deve ser tomado cuidado ao produzir os dados de teste, geralmente não disponíveis quando o aplicativo ainda não se encontra em produção. Valores muito semelhantes, completamente aleatórios, ou inseridos ordenadamente, distorcem as estatísticas em relação à distribuição que os dados reais devem ter.
- Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desativar vários tipos de planos (descritos no [Seção 16.4](#)). Por exemplo, desativar varreduras seqüenciais (`enable_seqscan`) e junções de laço-aninhado (`enable_nestloop`), que são os planos mais básicos, forcem o sistema a utilizar

um plano diferente. Se o sistema ainda assim escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponde ao índice (Qual tipo de consulta pode utilizar qual tipo de índice é explicado nas seções anteriores).

- Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: ou o sistema está correto e realmente a utilização do índice não é apropriada, ou a estimativa de custo dos planos de comando não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando `EXPLAIN ANALYZE` pode ser útil neste caso.
- Se for descoberto que as estimativas de custo estão erradas existem, novamente, duas possibilidades. O custo total é calculado a partir do custo por linha de cada nó do plano vezes a seletividade estimada do nó do plano. Os custos dos nós do plano podem ser ajustados usando parâmetros em tempo de execução (descritos no [Seção 16.4](#)). A estimativa imprecisa da seletividade é devida a estatísticas insuficientes. É possível melhorar esta situação ajustando os parâmetros de captura de estatísticas (consulte o comando [ALTER TABLE](#)).

Se não for obtido sucesso no ajuste dos custos para ficarem mais apropriados, então pode ser necessário o recurso de forçar a utilização do índice explicitamente. Pode-se, também, desejar fazer contato com os desenvolvedores do PostgreSQL para examinar este problema.

Índices com várias colunas

Pode ser definido um índice contendo mais de uma coluna. Por exemplo, se existir uma tabela como:

```
CREATE TABLE teste2 (  
    principal int,  
    secundario int,  
    nome      varchar  
);
```

(Digamos que seja armazenado no banco de dados o diretório /dev...) e freqüentemente sejam feitas consultas como

```
SELECT nome  
FROM   teste2  
WHERE  principal = constante AND secundario = constante;
```

então é apropriado definir um índice contendo as colunas principal e secundario como, por exemplo,

```
CREATE INDEX idx_teste2_princ_sec ON teste2 (principal, secundario);
```

Atualmente, somente as implementações de B-tree e GiST suportam índices com várias colunas. Podem ser especificadas até 32 colunas (Este limite pode ser alterado durante a geração do PostgreSQL; consulte o arquivo `pg_config_manual.h`).

O planejador de comandos pode utilizar um índice com várias colunas, para comandos envolvendo a coluna mais à esquerda na definição do índice mais qualquer número de colunas listadas à sua direita, sem omissões. Por exemplo, um índice contendo (a, b, c) pode ser utilizado em comandos

envolvendo todas as colunas a, b e c, ou em comandos envolvendo a e b, ou em comandos envolvendo apenas a, mas não em outras combinações (Em um comando envolvendo a e c, o planejador pode decidir utilizar o índice apenas para a, tratando c como uma coluna comum não indexada). Obviamente, cada coluna deve ser usada com os operadores apropriados para o tipo do índice; as cláusulas envolvendo outros operadores não são consideradas.

Os índices com várias colunas só podem ser utilizados se as cláusulas envolvendo as colunas indexadas forem ligadas por AND. Por exemplo,

```
SELECT nome
  FROM teste2
 WHERE principal = constante OR secundario = constante;
```

não pode utilizar o índice `idx_teste2_princ_sec` definido acima para procurar pelas duas colunas (Entretanto, pode ser utilizado para procurar apenas a coluna principal).

Os índices com várias colunas devem ser usados com moderação. Na maioria das vezes, um índice contendo apenas uma coluna é suficiente, economizando espaço e tempo. Um índice com mais de três colunas é quase certo não ser útil, a menos que a utilização da tabela seja muito peculiar.

Índices em expressões

Uma coluna do índice não precisa ser apenas uma coluna da tabela subjacente, pode ser uma função ou uma expressão escalar computada a partir de uma ou mais colunas da tabela. Esta funcionalidade é útil para obter acesso rápido às tabelas com base em resultados de cálculos. [1]

Por exemplo, uma forma habitual de fazer comparações não diferenciando letras maiúsculas de minúsculas é utilizar a função `lower`:

```
SELECT * FROM teste1 WHERE lower(col1) = 'valor';

-- Para não diferenciar maiúsculas e minúsculas, acentuadas ou não (N. do T.)

SELECT * FROM teste1 WHERE lower(to_ascii(col1)) = 'valor';
```

Esta consulta pode utilizar um índice, caso algum tenha sido definido sobre o resultado da operação `lower(col1)`:

```
CREATE INDEX idx_teste1_lower_col1 ON teste1 (lower(col1));

-- Para incluir as letras acentuadas (N. do T.)

CREATE INDEX idx_teste1_lower_ascii_col1 ON teste1 (lower(to_ascii(col1)));
```

Se o índice for declarado como `UNIQUE`, este impede a criação de linhas cujos valores de `col1` diferem apenas em letras maiúsculas e minúsculas, assim como a criação de linhas cujos valores de `col1` são realmente idênticos. Portanto, podem ser utilizados índices em expressões para impor restrições que não podem ser definidas como restrições simples de unicidade.

Como outro exemplo, quando são feitas habitualmente consultas do tipo

```
SELECT * FROM pessoas WHERE (primeiro_nome || ' ' || ultimo_nome) = 'Manoel
Silva';
```

então vale a pena criar um índice como:

```
CREATE INDEX idx_pessoas_nome ON pessoas ((primeiro_nome || ' ' ||  
ultimo_nome));
```

A sintaxe do comando CREATE INDEX normalmente requer que se escreva parênteses em torno da expressão do índice, conforme mostrado no segundo exemplo. Os parênteses podem ser omitidos quando a expressão for apenas uma chamada de função, como no primeiro exemplo.

É relativamente dispendioso manter expressões de índice, uma vez que a expressão derivada deve ser computada para cada linha inserida, ou sempre que for atualizada. Portanto, devem ser utilizadas somente quando as consultas que usam o índice são muito freqüentes.

Notas

- [1] O sistema de gerenciamento de banco de dados Oracle 10g também permite usar função e expressão escalar na coluna do índice, mas o SQL Server 2000 e o DB2 8.1 não permitem. [Comparison of relational database management systems](#) (N. do T.)

Remoção dos índices

Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados usando o COPY e, depois, criar os índices necessários para a tabela. Criar um índice sobre dados pré-existentes é mais rápido que atualizá-lo de forma incremental durante a carga de cada linha.

Para aumentar uma tabela existente, pode-se remover o índice, carregar a tabela e, depois, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não tiver sido criado novamente.

ALTER INDEX

Nome

ALTER INDEX -- altera a definição de um índice

Sinopse

```
ALTER INDEX nome  
    ação [, ... ]  
ALTER INDEX nome  
    RENAME TO novo_nome
```

onde ação é um entre:

```
OWNER TO novo_dono  
SET TABLESPACE nome_do_espaco_de_indices
```

Descrição

O comando ALTER INDEX altera a definição de um índice existente. Existem diversas subformas: OWNER

Esta forma torna o usuário especificado o dono do índice. Somente pode ser utilizado por um superusuário.

SET TABLESPACE

Esta forma altera o espaço de tabelas do índice para o espaço de tabelas especificado, e move os arquivos de dados associados ao índice para o novo espaço de tabelas. Consulte também [CREATE TABLESPACE](#).

RENAME

A forma RENAME muda o nome do índice. Não há efeito sobre os dados armazenados.

Todas as ações, com exceção de RENAME, podem ser combinadas em uma lista de alterações múltiplas a serem aplicadas em paralelo.

Parâmetros

nome

O nome (opcionalmente qualificado pelo esquema) de um índice existente.

novo_nome

O novo nome do índice.

novo_dono

O nome de usuário do novo dono do índice.

nome_do_espaco_de_tabelas

O nome do espaço de tabelas para o qual o índice será movido.

Observações

Estas operações também podem ser feitas utilizando [ALTER TABLE](#). O comando ALTER INDEX é, na verdade, apenas um sinônimo para as formas de ALTER TABLE que se aplicam aos índices.

Não é permitido alterar qualquer parte de um índice dos catálogos do sistema.

Exemplos

Para mudar o nome de um índice existente:

```
ALTER INDEX distribuidores RENAME TO fornecedores;
```

Para mover um índice para outro espaço de tabelas:

```
ALTER INDEX distribuidores SET TABLESPACE espaco_de_tabelas_rapido;
```

Compatibilidade

O comando ALTER INDEX é uma extensão do PostgreSQL.

Mudar Dono de Índice

Para mudar o dono de uma tabela, índice, sequência ou visão deve ser utilizado o comando [ALTER TABLE](#).

Índices únicos

Os índices também podem ser utilizados para impor a unicidade do valor de uma coluna, ou a unicidade dos valores combinados de mais de uma coluna.

```
CREATE UNIQUE INDEX nome ON tabela (coluna [, ...]);
```

Atualmente, somente os índices B-tree poder ser declarados como únicos.

Quando o índice é declarado como único, não pode existir na tabela mais de uma linha com valores indexados iguais. Os valores nulos não são considerados iguais. Um índice único com várias colunas rejeita apenas os casos onde todas as colunas indexadas são iguais em duas linhas.

O PostgreSQL cria, automaticamente, um índice único quando é definida na tabela uma restrição de unicidade ou uma chave primária. O índice abrange as colunas que compõem a chave primária ou as colunas únicas (um índice com várias colunas, se for apropriado), sendo este o mecanismo que impõe a restrição.

Nota: A forma preferida para adicionar restrição de unicidade a uma tabela é por meio do comando ALTER TABLE ... ADD CONSTRAINT. A utilização de índices para impor restrições de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente. Entretanto, deve-se ter em mente que não é necessário criar índices em colunas únicas manualmente; caso se faça, simplesmente se duplicará o índice criado automaticamente.

DROP INDEX

Nome

DROP INDEX -- remove um índice

Sinopse

```
DROP INDEX nome [, ...] [ CASCADE | RESTRICT ]
```

Descrição

O comando DROP INDEX remove do sistema de banco de dados um índice existente. Para executar este comando é necessário ser o dono do índice.

Parâmetros

nome

O nome (opcionalmente qualificado pelo esquema) do índice a ser removido.

CASCADE

Remove automaticamente os objetos que dependem do índice.

RESTRICT

Recusa remover o índice se existirem objetos que dependem do mesmo. Este é o padrão.

Exemplos

O comando a seguir remove o índice idx_titulo:

```
DROP INDEX idx_titulo;
```

Compatibilidade

O comando DROP INDEX é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

Índices – de Daniel Oslei

Olá pessoal. Desculpem pela demora na edição de um novo artigo para a Coluna PostgreSQL iMasters.

No primeiro artigo, anunciei que a Coluna PostgreSQL não tratará apenas de assuntos específicos desse SGBD, mas também de conceitos de bancos de dados, e é isso que ocorrerá neste primeiro artigo sobre índices. PostgreSQL ficará um pouco de lado, para que tratemos de uma breve introdução sobre o que são índices. No próximo artigo trataremos mais especificamente os índices no PostgreSQL.

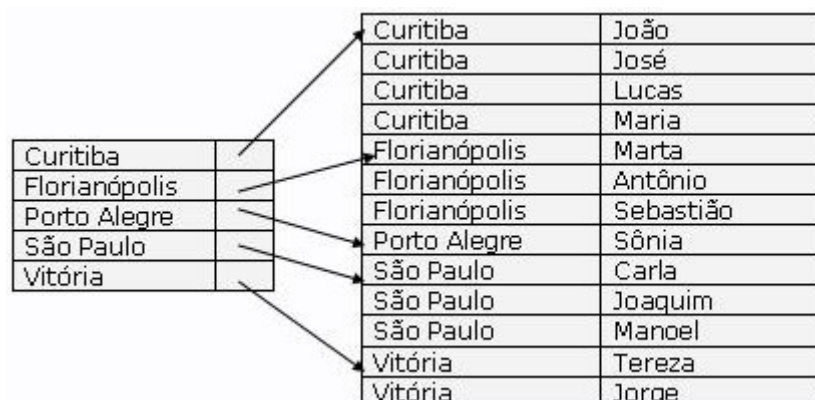
Em grande parte das consultas que são feitas a uma base de dados, fazem referência a apenas uma pequena proporção dos registros de um arquivo. Por exemplo, numa base de dados de uma grande empresa podem ser feitas muito mais consultas para se saber os clientes que moram na cidade de Curitiba do que consultas para saber quem são os clientes de Porto Alegre. E dentro das pesquisas dos clientes curitibanos, podem haver muito mais pesquisas em relação a quem mora em um determinado bairro do que em outros. Esses são exemplos bem simples de como determinadas consultas são feitas repetidamente e várias vezes ao dia. Mas imaginem se o SGBD para fazer essas consultas tenha que analisar registro por registro para poder retornar um resultado. Se a quantidade de registros for pouca, isso será imperceptível. Mas se pelo contrário, existirem um enorme número de linhas em várias tabelas, com várias chaves estrangeiras, começaram a surgir graves problemas de desempenho no sistema. Por isso, é necessária a criação de estruturas para que as consultas sejam executadas na melhor performance possível.

Muitas vezes, quando consultamos um livro, não podemos lê-lo todo para encontrarmos o que procuramos. Se for deste modo há um tempo muito grande sendo desperdiçado. Para isso que existe nos livros os índices, no qual podemos encontrar com mais facilidade o que desejamos. Da mesma maneira, os bancos de dados utilizam índices, para que não só consultas, mas inserções, exclusões e atualizações sejam feitas com mais agilidade.

Habitualmente, os índices são utilizados para melhorar o desempenho dos bancos de dados. Um índice permite ao servidor de banco de dados encontrar e trazer linhas específicas com muito mais rapidez do que faria sem o índice. Mas os índices também produzem trabalho adicional para o sistema de banco de dados como um todo, portanto, deve-se adquirir bons conhecimentos sobre o assunto para o seu devido uso.

Os índices podem beneficiar as atualizações e as exclusões com condição de procura. Eles também podem ser utilizados em consultas com junção. Portanto, um índice definido em uma coluna que faça parte da condição de junção pode acelerar, significativamente, a consulta.

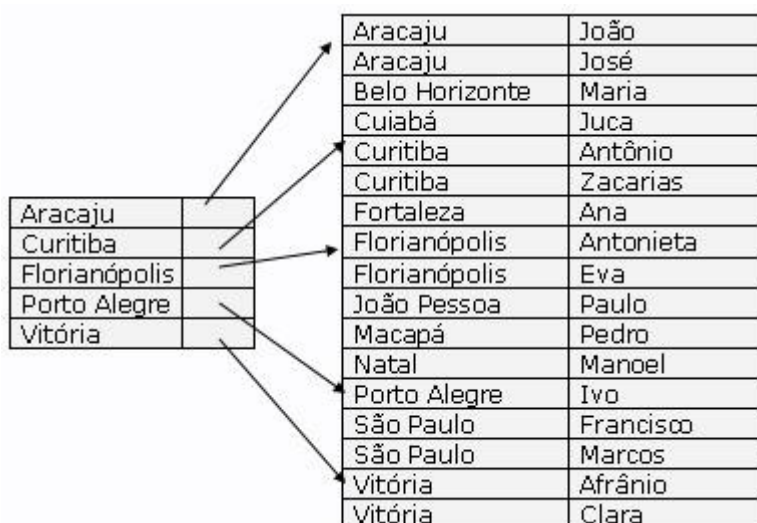
Um dos mais antigos esquemas de índice utilizados em sistema de banco de dados é chamado de arquivo indexado seqüencialmente, que são projetados para aplicações que requerem tanto o processamento seqüencial de um arquivo inteiro quanto o acesso aleatório a registros individuais. Num exemplo um pouco mais avançado, mostramos na imagem abaixo uma tabela na qual os registros são indexados pelo nome da cidade em que moram os clientes. Para encontrarmos os clientes de uma determinada cidade, encontramos a cidade na primeira tabela e seguimos para onde o ponteiro correspondente está apontando, lendo seqüencialmente até encontrar uma cidade diferente da solicitada:



Índices Densos

Reparem que neste exemplo, para cada cidade existe um registro de índice (ou entrada de índice), mesmo que haja apenas um registro para determinada cidade. Este tipo de colocação de índices é chamado de índices densos. Existe uma outra forma de índice conhecida como índices esparsos.

Veja o exemplo abaixo:

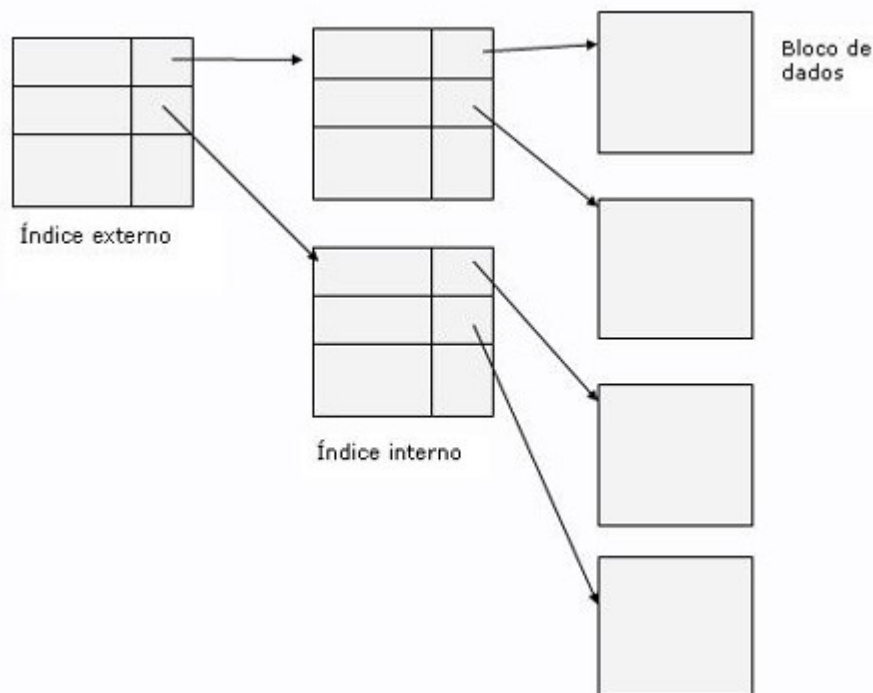


Índices Esparsos

Neste exemplo acima, são criados registros de índices para apenas alguns dos valores. Para localizar um registro, encontramos a entrada de índice com o maior valor de chave de procura que seja menor ou igual ao valor de chave de procura que estamos procurando. Iniciamos no registro apontando para a entrada de índice e seguimos os ponteiros no arquivo até encontrarmos o registro desejado. **Os índices densos são preferíveis comparados aos índices esparsos, devido a possibilidade de encontrarmos com mais agilidade o desejado.** A vantagem dos índices esparsos é o fato de ocuparem pouco espaço em disco e menos trabalho na manutenção em inserções e exclusões.

Mesmo assim, se pensarmos em grandes fontes de armazenamento de dados, essa forma de índices tornaria o desempenho do banco extremamente baixo. Se um índice for suficientemente pequeno para ser mantido na memória principal, o tempo de busca para encontrar uma entrada será baixo. Entretanto, se o índice for tão grande que tenha de ser mantido em disco, a busca de uma entrada exigirá diversas leituras de blocos. Para solucionar este problema, o índice deve ser tratado como qualquer outro arquivo seqüencial, e construímos um índice esparsos no índice primário:

Cada vez que ocorre uma inserção ou remoção de dados, os índices devem ser atualizados. Quando é inserida uma informação, se o índice for denso, é feita uma procura com o dado chave para o índice, se caso não for encontrado, esse valor é incluído no índice. Se o índice for esparsos e armazenar uma entrada para cada bloco, não é necessário fazer nenhuma alteração no índice, exceto se um novo bloco tenha sido criado, então o primeiro valor de procura que aparece no novo bloco é inserido. Quando um registro é removido, se o registro for o único para o valor chave de procura, então esse valor chave é excluído do índice. Com índices esparsos, removemos uma valor de chave

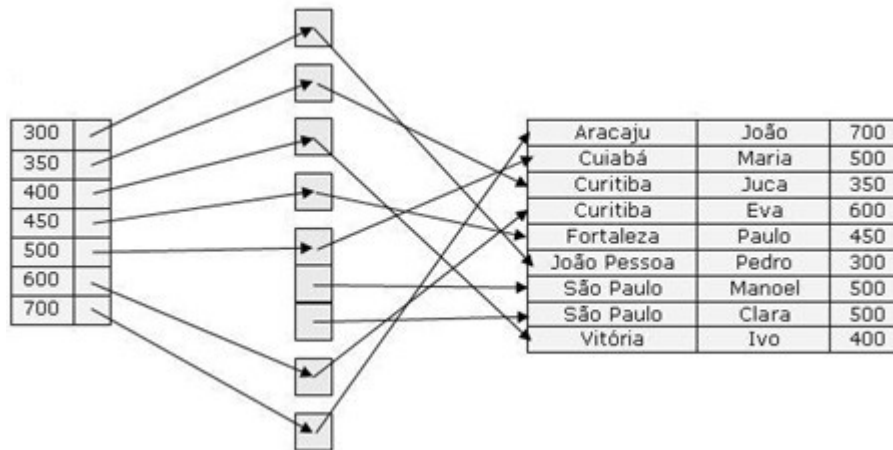


e substituímos sua entrada (se houver) no índice pelo próximo valor de chave de procura (na ordem da chave de procura). Se o próximo valor de chave de procura já tiver uma entrada de índice, a entrada é apagada em vez de ser substituída.

São chamados de índices primários os índices que pertencem a uma chave primária ou a que definam a seqüência dos registros. Existem também os índices secundários, que normalmente pertencem a chaves candidatas. Os índices secundários são muito semelhantes aos índices densos, a não ser pelo fato de os registros apontados por valores sucessivos no índice não estão armazenados seqüencialmente, pois, os registros estão dispostos de tal forma a satisfazer a ordenação do índice primário, o que resulta no fato de os índices secundários terem que possuir ponteiros para todos os registros. Uma leitura seqüencial na ordem do índice primário é satisfatório, pois, os registros estão

fisicamente armazenados na mesma ordem do índice primário, o que não acontece com os índices secundários. Como a ordem do índice primário e do secundário diferem, provavelmente surgiriam complicações com a leitura através da ordem do índice secundário.

Com índices secundários, os ponteiros não apontam diretamente para o arquivo com registros, mas para um *bucket* que contém ponteiros para o arquivo. Veja o exemplo abaixo:



O que foi mostrado acima, é apenas o princípio de como os índices são feitos, para que servem e como funcionam. Existem algoritmos muito mais avançados e complexos do que os já citados, e ainda existem muitos estudos em cima deste assunto, sempre na tentativa de fazer com que os índices sejam mais eficientes, exigindo menos processamento, otimização do espaço em disco e exijam menos manutenção.

O PostgreSQL atualmente implementa quatro tipos de índices: B-Tree, R-Tree, GiST e Hash. Cada um com o seu grau de eficiência, podendo ser recomendado algum deles para uma determinada aplicação e outros para outros tipos de aplicações. Veremos na parte II de nosso artigo, como funciona cada um deles e para o que são recomendados. Na parte III, veremos detalhadamente como são usados.

Índices - Parte 02

Como visto na coluna passada os índices são objetos de vital importância para o bom desempenho do banco de dados. Também foi visto que o PostgreSQL usa índices para melhor responder as requisições solicitadas a ele. Mas tem um porém, o PostgreSQL não cria por conta própria os índices, quem deve criar os índices é o usuário proprietário da tabela, exceto quando é criada uma chave primária, na qual toda chave primária é um índice. E para isso ele já deve ter em mente qual das colunas será mais usada em cláusulas WHERE durante a existência da tabela. Índices são criados usando o comando **CREATE INDEX**, veja o exemplo:

```
CREATE INDEX nome_indice ON tabela_nome (coluna_nome);
```

Ao criar a chave primária devemos ter em mente qual o campo a ser utilizado na cláusula WHERE.

No exemplo citado acima, foi criado um índice *nome_indice* na coluna *coluna_nome* da tabela *tabela_nome*. Quando o índice é criado com sucesso, é retornado para o prompt a palavra **CREATE** (é aconselhável utilizar nomes significativos para os índices).

Quando se cria um índice, este índice está relacionado a uma determinada coluna, portanto, não pode auxiliar no acesso de informações em outras colunas porque os índices são classificados de acordo com a coluna correspondente, eis o motivo para o qual deve-se saber muito bem a conceituação do assunto para criar índices que tragam realmente melhora de desempenho. É claro que você pode criar vários índices dentro de uma mesma tabela, mas um índice que seja raramente usado é um desperdício de espaço em disco e também vários índices requerem que para cada atualização num registro seja também feita uma atualização em cada índice.

Por padrão quando criamos um índice e não especificamos qual o tipo que queremos usar, o PostgreSQL utiliza o B-Tree, ou seja, nas situações mais comuns. Mas podem ocorrer vezes em que o PostgreSQL escolha outro tipo, para isso, ele analisa o caso e leva em consideração se a coluna indexada está envolvida numa comparação envolvendo determinados operadores:

| Tipo | Operadores |
|--------|------------------------------|
| B-Tree | <, <=, =, >=, > |
| R-Tree | <<, &<, &>, >>, @, ~=, && |
| Hash | = |

No momento esta definição pode ser que fique um pouco obscura, mas por enquanto não vamos nos preocupar com isso. Em artigo próximo veremos com mais detalhes para o que são e como são utilizados os operadores e classes de operadores, vamos concentrar nossos raciocínios apenas em índices.

Criar Índice com Tipo não Padrão

Mas se desejarmos escolher o tipo de índice, devemos acrescentar ao final o comando USING, logo após especificando o tipo desejado:

```
CREATE INDEX nome ON tabela USING HASH (coluna);
CREATE INDEX nome ON tabela USING BTREE (coluna);
CREATE INDEX nome ON tabela USING RTREE (coluna);
CREATE INDEX nome ON tabela USING GIST (coluna);
```

Normalmente gera-se muita confusão no momento de escolher que índice utilizar, já que cada um tem uma finalidade. Na tabela seguinte há uma tentativa de conceituar brevemente cada um dos quatro índices:

É comum precisarmos utilizar consultas que usam na cláusula WHERE condições de pesquisas que dependem muito de duas colunas, então é necessário fazer com que o índice utilize duas colunas. Isso é totalmente possível no PostgreSQL, desde que os índices sejam do tipo B-tree ou GiST e não ultrapassem o limite de 32 colunas (o valor de 32 colunas pode ser alterado apenas durante a geração do PostgreSQL, alterando o arquivo pg_config.h em versões 7.3 ou pg_config_manual.h em versões 7.4). Quando se utiliza mais de uma coluna, o índice é organizado de acordo com a primeira coluna especificada na declaração, sendo o segundo utilizado quando a primeira coluna possuir vários valores iguais, portanto a segunda coluna surgiria como uma segunda opção de classificação.

| Índice | Definição |
|--------|---|
| B-Tree | São árvores de pesquisa balanceadas desenvolvidas para trabalharem em discos magnéticos ou qualquer outro dispositivo de armazenamento de acesso direto em memória secundária. O índice <i>B-tree</i> é uma implementação das árvores B de alta concorrência propostas por Lehman e Yao. |
| R-Tree | Também conhecido como árvores R, utiliza o algoritmo de partição quadrática de Guttman, sendo utilizada para indexar estrutura de dados multidimensionais, cuja implementação está limitada a dados com até 8Kbytes, sendo bastante limitada para dados geográficos reais. Utilizado normalmente com dados do tipo box, circle, point e outros. |
| Hash | O índice <i>hash</i> é uma implementação das dispersões lineares de Litwin. Na própria documentação do PostgreSQL está presente a seguinte nota: "Os testes mostram que os índices <i>hash</i> do PostgreSQL têm desempenho semelhante ou mais lento que os índices <i>B-tree</i> , e que o tamanho e o tempo de construção dos índices <i>hash</i> são muito piores. Os índices <i>hash</i> também possuem um fraco desempenho sob alta concorrência. Por estas razões, a utilização dos índices <i>hash</i> é desestimulada"**. . |
| GiST | Generalized Index Search Trees (Árvores de Procura de Índice Generalizadas), para mais informações visite http://www.sai.msu.su/~megeera/postgres/gist/doc/intro.shtml |

CREATE INDEX coluna1_coluna2_tabela_idx ON tabela (coluna1,coluna2);

Também é importante salientar que quando utilizado várias colunas num índice, o otimizador de consultas pode utilizar todas as colunas especificadas ou apenas uma ou algumas, de acordo como ele decidir. Isto vai depender se as colunas são consecutivas. Por exemplo, um índice incluindo (*col_1*, *col_2*, *col_3*) pode ser utilizado em consultas envolvendo *col_1*, *col_2* e *col_3*, ou em consultas envolvendo *col_1* e *col_2*, ou em consultas envolvendo apenas *col_1*, mas não em outras combinações. (Em uma consulta envolvendo *col_1* e *col_3*, o otimizador pode decidir utilizar um índice para *col_1* apenas, tratando *col_3* como uma coluna comum não indexada).

```
CREATE TABLE clientes (id serial, nome varchar(50), ano_nasc int, valor_devido float)
CREATE INDEX ano_valor_clientes_idx ON clientes USING BTREE (ano_nasc, valor_devido);
SELECT nome FROM clientes WHERE ano_nasc > 1970 AND valor_devido < 2000;
SELECT nome FROM clientes WHERE ano_nasc > 1970 OR valor_devido < 2000;
```

No exemplo acima é criada uma tabela de importância desprezível, na qual é criado índices para as duas últimas colunas. Supondo que as tabelas já estão povoadas, são feitas duas consultas, ambas utilizando os mesmos critérios de consultas, exceto por uma utilizar o AND e outra o OR. No

entanto apenas a primeira consulta utiliza o índice. Por definição, o PostgreSQL utiliza apenas o índice com mais de uma coluna quando as colunas estão unidas numa cláusula WHERE por AND, em outros casos o índice vai ser utilizado apenas na coluna que foi definida por primeiro na criação do índice. No exemplo anterior, na segunda consulta o índice `ano_valor_clientes_idx` só será usado na procura dos clientes nascidos após 1970, pelo fato de `ano_nasc` ser a coluna principal do índice.

Os índices com mais de uma coluna devem ser evitados. Dependendo da análise há casos em que é melhor haver duas colunas do que uma. No entanto os especialistas afirmam que o uso de três ou mais colunas é praticamente inviável.*

* *Ou inevitável? (Observação de Ribamar FS).*

Neste artigo foi apenas frisado a criação de índices no PostgreSQL, o que na realidade é muito pouco para a sua compreensão. No próximo artigo continuaremos a falar do assunto, mostrando mais detalhes para que se possa tirar o melhor proveito do uso da indexação.

* Para se visualizar os índices criados na base de dados utilize o comando 'di' sem as aspas.

** Hashing: valor de identificação produzido através da execução de uma operação numérica, denominada função de hashing, em um item de dado. O valor identifica de forma exclusiva o item de dado, mas exige um espaço de armazenamento bem menor. Por isso, o computador pode localizar mais rapidamente os valores de hashing do que os itens de dado, que são mais extensos. Uma tabela de hashing associa cada valor a um item de dado exclusivo. Webster's New World Dicionário de Informática, Brian Pfaffenberger, Editora Campus, 1999.

Índices - Parte 3

Olá pessoal. Nesta série de artigos sobre índice, vimos, no primeiro artigo, conceitos básicos sobre o assunto, sem detalhar os índices no PostgreSQL. No segundo artigo tivemos a oportunidade de aprendermos como se cria índices e quais as possibilidades de índices que temos para usar. Neste terceiro artigo avançaremos mais um pouco, mostrando mais alguns detalhes importantes na sua criação. Se acaso desejar ver os dois primeiros artigos, acesse o links abaixo:

- [Parte 1](#);
- [Parte 2](#);

Índices únicos

Os índices não precisam ser necessariamente usados como forma de aceleração nas consultas, mas também como uma maneira de restringir que dados se repitam em uma determinada coluna. Portanto, ele funciona de forma semelhante a uma chave primária, não permitindo que, por exemplo, num campo onde se digita o cpf de um usuário sejam inseridos duas vezes o mesmo cpf. Para criar um índice que tenha esta função acrescenta-se a cláusula UNIQUE após a palavra CREATE:

CREATE UNIQUE INDEX nome_indice ON tabela (coluna)

Em casos em que são acrescentados valores nulos para a coluna, o PostgreSQL não restringe a sua inserção, podendo assim, existir vários valores nulos nesta determinada coluna da tabela. Outro detalhe importante é o de usar duas ou mais colunas com índices únicos, nestes casos apenas será impedida a inserção de dados quando todos os novos dados pertencentes ao índice estão coincidindo com uma outra tupla já existente. Num caso em que haja três colunas num índice, e se acaso for

inserir um registro que repita os dados das duas primeiras colunas, mas não da terceira, o registro será inserido sem nenhum problema.

B-Tree é o único tipo de índice que aceita restrição de unicidade.

O PostgreSQL oferece esta possibilidade, mas se for usar restrição em colunas para que sejam dados únicos, é mais aconselhável utilizar as CONSTRAINTs, a utilização de índices para impor a restrição de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente. Entretanto, se deve ter em mente que não há necessidade de criar índices em colunas únicas manualmente; se isto for feito, simplesmente será duplicado o índice criado automaticamente.

```
CREATE TABLE usuario  
(  
  cpf int,  
  nome varchar,  
  CONSTRAINT constraint_exemplo UNIQUE (cpf)  
);
```

Utilização de índices em expressões

Os índices não precisam necessariamente ficarem atrelados aos dados contidos na base de dados. Eles podem, de acordo com a necessidade, estar baseados em funções do PostgreSQL. Esta funcionalidade é útil para obter acesso rápido às tabelas baseado no resultado de cálculos.

Por exemplo, são utilizadas freqüentemente as funções lower e upper. A primeira função tem como finalidade converter os caracteres de uma string totalmente para minúsculas, e a segunda para maiúsculas. Podemos fazer comparações da seguinte forma:

```
SELECT * FROM tabela WHERE lower(coluna) = 'valor';
```

ou

```
SELECT * FROM teste1 WHERE upper(coluna) = 'VALOR';
```

Para que haja uma melhora de performance nestes tipos de consultas, podemos utilizar índices que já utilizem uma determinada função para a sua organização:

```
CREATE INDEX idx_tabela_lower_coluna ON (lower(coluna));
```

ou

```
CREATE INDEX idx_tabela_upper_coluna ON (upper(coluna));
```

Outro exemplo que podemos dar está relacionado ao fato de muitas vezes quando são inseridos dados em um determinado campo, eles podem vir com caracteres de espaço no início ou no fim da string. Para isso podem ser criados índices que automaticamente se adaptem aos registros da coluna, sem considerar os caracteres de espaço que estão sobrando. Neste caso, teremos que utilizar a função ltrim, que remove os caracteres que desejarmos no início de uma string:

```
CREATE INDEX idx_testando_ltrim_coluna  
ON tabela  
USING btree  
(ltrim(coluna));
```

Um outro exemplo pode ser utilizado para quem habitualmente faz consultas como:


```
SELECT * FROM pessoas WHERE (nome || ' ' || sobrenome) = 'Juca Silva';
```

Então vale a pena criar um índice como:

```
CREATE INDEX idx_pessoas_nome ON pessoas ((nome || ' ' || sobrenome));
```

Se o índice for declarado como UNIQUE, este impede a criação de linhas cujos valores da coluna diferem apenas em maiúsculas e minúsculas, assim bem como linhas cujos valores da coluna são realmente idênticos. Portanto, os índices em expressões podem ser utilizados para impor restrições que não podem ser definidas como restrições simples de unicidade.

Índices parciais

Os índices parciais, como o próprio nome nos diz, não cobre uma coluna na sua totalidade, cobre apenas um subconjunto, obedecendo a uma determinada condição. Para isso, o índice conterá apenas entradas que satisfaçam a condição indicada na sua criação. O principal objetivo dos índices parciais é evitar que valores muito comuns para aquela coluna sejam indexados. A vantagem da indexação parcial, é que não será necessário atualizar o índice a cada vez que ocorrer uma atualização na tabela, apenas quando envolverem valores específicos para o índice, o que também implicará na redução do tamanho do índice e maior agilidade em consultas. Abaixo serão apresentados dois exemplos de possíveis utilizações de índices parciais, o segundo foi extraído da própria documentação do PostgreSQL:

a) Dentro de uma livraria existem uma grande quantidade de livros, sendo que existem livros escritos em português, inglês, alemão, francês e italiano. Do total destes livros, 8% são em português, 80% em inglês, 2% em alemão, 6% em francês e 4% em italiano. Mas 70% das buscas são procurando livros escritos em português, italiano, francês ou alemão e 30% em inglês. Portanto, há uma considerável quantidade de procura para livros nos idiomas que não seja o inglês.

Podemos criar um índice para o campo idioma envolvendo a coluna toda. Mas, no entanto, se as consultas são a maior parte para os idiomas em português, italiano, francês e alemão, vamos criar um índice apenas para os quatro idiomas mais usados e que possuem menos registros no banco.

Considerando que os nomes dos idiomas estão numa tabelas a parte, e inclui-se apenas o código do idioma na tabela principal, a tabela idioma possui os seguintes valores:

| Tabela idioma | |
|---------------|-----------|
| codigo | idioma |
| 1 | Inglês |
| 2 | Português |
| 3 | Alemão |
| 4 | Francês |
| 5 | Italiano |

Usando as informações citadas acima, podemos criar um índice parcial da seguinte forma:

```
CREATE INDEX idx_livro_idioma ON livro (idioma) WHERE idioma != 1;
```

Consultas que usarem como condição ser de qualquer idioma exceto inglês, utilizará o índice parcial.

b) Suponha que as informações sobre o acesso ao servidor Web são armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente sobre o acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Para criar um índice parcial adequado ao exemplo acima, deve ser utilizado um comando como este:

```
CREATE INDEX idx_registro_acesso_ip_cliente  
ON tbl_registro_acesso (ip_cliente)  
WHERE NOT (ip_cliente > inet '192.168.100.0'  
AND ip_cliente < inet '192.168.100.255');
```

Tipicamente, uma consulta que poderia utilizar este índice seria:

```
SELECT * FROM tbl_registro_acesso WHERE url =  
'http://conteudo.imasters.com.br/1959/index.html' AND ip_cliente = inet '212.78.10.32';
```

Uma consulta que não poderia utilizar este índice seria:

```
SELECT * FROM tbl_registro_acesso WHERE ip_cliente = inet '192.168.100.23';
```

Deve-se levar em consideração que os exemplos dados anteriormente, são meramente explicativos ao que se refere os índices parciais, sem seguir ou levar em consideração conceitos de normalização de banco de dados. O PostgreSQL suporta índice parcial com predicados arbitrários, desde que somente colunas da tabela sendo indexada estejam envolvidas. Entretanto, deve-se ter em mente que a condição colocada na criação do índice deve corresponder às condições utilizadas nas consultas que supostamente vão se beneficiar do índice. O índice parcial apenas será utilizado em uma consulta se o sistema puder reconhecer que a condição WHERE da consulta implica matematicamente no predicado do índice.

O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer predicados matematicamente equivalentes escritos de formas diferentes. O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, “ $x < 1$ ” implica “ $x < 2$ ”; senão, a condição do predicado deve corresponder exatamente à condição WHERE da consulta, ou o índice não será reconhecido como utilizável. Normalmente, o PostgreSQL realiza escolhas racionais sobre a utilização dos índices, por exemplo, evita-os ao buscar valores com muita ocorrência, de tal forma que o segundo exemplo realmente economiza apenas o tamanho do índice, não sendo necessário para evitar a utilização do índice.

No próximo artigo veremos o que mais interessa sobre os índices, que são dicas para que os índices realmente tenham um significado positivo no seu banco de dados. Por enquanto é isso pessoal, um grande abraço a todos e até mais.

Índices - Dicas de desempenho

Olá comunidade iMasters! Depois de três artigos mostrando conceitos de índices no PostgreSQL, enfim chegamos à última parte. E nesse artigo teremos como objetivo obter idéias mais avançadas na criação de índice, tentando conduzir sempre ao melhor desempenho possível.

Para que esse objetivo seja conquistado, dependerá do responsável pela administração do banco de dados fazer uma análise correta da situação que se passa com auxílio de algumas estatísticas sobre o banco. Como obter estas estatísticas ficará para ser aprofundado num próximo artigo, mas desde já comecemos a se acostumar com algumas idéias interessantes. Colocarei em forma de itens uma breve seqüência de dicas:

- É importante utilizar índices em chaves estrangeiras, já que estes são muito utilizados em **joins**. O índice será útil quando a tabela que possui uma chave estrangeira tentar acessar dados na tabela que dá suporte à esta chave. Também será útil no caso em que for excluir determinada linha de uma tabela que possui uma chave estrangeira, terá que ser feita uma leitura na tabela de onde vem os dados para a chave estrangeira para excluir os dados nesta segunda tabela no caso em que foi definido **ON DELETE CASCADE**, ou para não permitir a deleção na primeira tabela se caso foi determinado **ON DELETE RESTRICT**.
- As expressões de índice são relativamente dispendiosas de serem mantidas, uma vez que a expressão derivada deve ser computada para cada linha ao ser inserida ou sempre que for atualizada. Portanto, devem ser utilizadas somente quando as consultas que usam o índice são muito frequentes.
- Deve-se ter muito cuidado em comparações de proporcionalidade envolvendo números relativamente pequenos. Enquanto selecionar 1.000 de cada 100.000 linhas (1% do total) pode ser um candidato para um índice, selecionar 1 de cada 100 linhas, que também corresponde a 1% do total, dificilmente será, porque as 100 linhas provavelmente cabem dentro de uma única página do disco, não havendo nenhum plano melhor que uma busca seqüencial em uma página do disco.
- Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desativar vários tipos de planos. Por exemplo, desativar varreduras seqüenciais (`enable_seqscan`) e junções de laço-aninhado (`nested-loop joins`) (`enable_nestloop`), que são os planos mais básicos, forcem o sistema a utilizar um plano diferente. Se o sistema ainda assim escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponder ao índice.
- Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: ou o sistema está correto e a utilização do índice não é apropriada, ou a estimativa de custo dos planos de comando não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando **EXPLAIN ANALYZE** pode ser útil neste caso.
- Quando achar necessário que determinadas informações tenham que ser únicas em uma determinada coluna, evite que a aplicação que utiliza o banco de dados faça isto. Haverá mais vantagens se o SGBD se responsabilizar pela unicidade dos dados.
- Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados usando o **COPY** e, depois, criar todos os índices necessários para a tabela. Criar um índice sobre dados pré-existent é mais rápido que atualizar de forma incremental durante a carga de cada linha. Para carregar uma tabela existente, pode-se remover o índice, carregar a tabela

e, depois, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não tiver sido criado novamente.

- Utilizando o comando abaixo, poderá se obter informações sobre os índices contidos no banco de dados, como por exemplo, o número total de varreduras que utilizaram um determinado índice e o número de linhas lidas com aquele índice.

`SELECT * FROM pg_stat_all_indexes`

- Em determinados casos, em que haja várias consultas que utilizem os comandos ORDER BY, GROUP BY e DISTINCT é aconselhável criar um índice para a coluna que está sendo utilizada nestas consultas. Isso se deve ao fato de que cada vez que ocorre isto, o SGBD dispara um SORT para a ordenação dos dados, o que pode corromper desempenho. Havendo índices para este caso, os dados já poderão estar ordenados e no final das contas economizando alguns milissegundos em processamento.

Brevemente será lançado um artigo para esclarecimento da utilização de informações estatísticas que o PostgreSQL possibilita para que se possa melhor planejar o caminho que o SGBD terá que percorrer e também para que se possa mais facilmente encontrar pontos que causem problemas no seu desempenho.

Dica:

Como já havia escrito na outra mensagem ...

Se você tiver MUITOS NULLS na tabela ou se o Índice não foi criado corretamente, ele não será usado mesmo.

Normalmente , se 30% dos valores da hash forem NULLS, o planejador não usará o índice, pq é mais rápido fazer scan.

Outra possibilidade é criar índice composto condicional (tsac AND ISNULL(tpgt)) e utilizar a funcao na comparação !

Thiago Risso na pgbr-geral.

Índices

- Tabelas pequenas não necessitam
- Como os índices têm seu custo devemos evitar muitos índices numa tabela
- Campos em cláusulas WHERE, em grandes tabelas e em consultas muito utilizadas são fortemente recomendados e geralmente melhoram muito o desempenho.
- Campos numéricos são mais indicados para índice
- Com índice vale o ditado popular: "Nem oito nem oitenta".

Teoria sobre Índices e uso no PostgreSQL:

Uma boa indicação é a série de tutoriais do Daniel Oslei, no site iMasters (<http://www.imasters.com.br>).

Os SGBDs utilizam índices para que consultas de recuperação, inserção, alteração e

exclusão sejam feitas com melhor performance.

Criar um índice:

```
create index nome on tabela(campo);
```

Existem vários tipos de índice e o mais popular é o B-Tree. O tipo B-Tree é o único tipo que aceita restrição de unicidade.

Criar um índice de um tipo específico:

```
create index nome on tabela using rtree (campo);
```

Índice em dois campos:

```
create index idx_c1c2 on clientes(campo1, campo2);
```

Criando Índices com Funções (funcionais):

```
create index idx_minusculas on clientes(lower(campo));  
create index idx_minusculas on clientes(upper(campo));
```

Índices com Expressões:

```
create index idx_concatena on clientes((nome || ' ' || sobrenome));
```

Índices Parciais:

```
create index idx_livro_idioma on livros(idioma) where idioma != 1;  
create index idx_acesso on clientes(ip_cliente)  
  where not (ip_cliente > inet '10.3.20.5'  
  and ip_cliente < inet '10.5.20.6');
```

Informações sobre todos os índices do banco atual:

```
select * from pg_stat_all_indexes;
```

Dicas

- Sempre crie chaves primárias para as tabelas
- Sempre crie chaves estrangeiras quando indicado
- Não criar índices:
 - em tabelas com muitos nulos
 - em tabelas muito pequenas

Da FAQ do PostgreSQL:

Indexes are not used by every query. Indexes are used only if the table is larger than a minimum size, and the query selects only a small percentage of the rows in the table. This is because the random disk access caused by an index scan can be slower than a straight read through the table, or sequential scan.

To determine if an index should be used, PostgreSQL must have statistics about the table. These statistics are collected using VACUUM ANALYZE, or simply ANALYZE. Using statistics, the optimizer

knows how many rows are in the table, and can better determine if indexes should be used. Statistics are also valuable in determining optimal join order and join methods. Statistics collection should be performed periodically as the contents of the table change.

Indexes are normally not used for ORDER BY or to perform joins. A sequential scan followed by an explicit sort is usually faster than an index scan of a large table. However, LIMIT combined with ORDER BY often will use an index because only a small portion of the table is returned.

If you believe the optimizer is incorrect in choosing a sequential scan, use SET enable_seqscan TO 'off' and run query again to see if an index scan is indeed faster.

When using wild-card operators such as LIKE or ~, indexes can only be used in certain circumstances:

- The beginning of the search string must be anchored to the start of the string, i.e.
 - LIKE patterns must not start with %.
 - ~ (regular expression) patterns must start with ^.
- The search string can not start with a character class, e.g. [a-e].
- Case-insensitive searches such as ILIKE and ~* do not utilize indexes. Instead, use expression indexes, which are described in section 4.8.
- The default C locale must be used during initdb because it is not possible to know the next-greatest character in a non-C locale. You can create a special text_pattern_ops index for such cases that work only for LIKE indexing. It is also possible to use full text indexing for word searches.

How do I perform regular expression searches and case-insensitive regular expression searches? How do I use an index for case-insensitive searches?

The ~ operator does regular expression matching, and ~* does case-insensitive regular expression matching. The case-insensitive variant of LIKE is called ILIKE.

Case-insensitive equality comparisons are normally expressed as:

```
SELECT *  
FROM tab  
WHERE lower(col) = 'abc';
```

This will not use an standard index. However, if you create an expression index, it will be used:

```
CREATE INDEX tabindex ON tab (lower(col));
```

If the above index is created as UNIQUE, though the column can store upper and lowercase characters, it can not have identical values that differ only in case. To force a particular case to be stored in the column, use a CHECK constraint or a trigger.

Links dos originais no iMasters:

<http://imasters.uol.com.br/artigo/1897/postgresql/indices/>
http://imasters.uol.com.br/artigo/1922/postgresql/indices_-_parte_02/
http://imasters.uol.com.br/artigo/1959/postgresql/indices_-_parte_3/
http://imasters.uol.com.br/artigo/1977/postgresql/indices_-_dicas_de_desempenho/

Arquitetura Interna de Banco de Dados - Índices

Autor: Prof. Vandor Roberto Vilardi Rissoli

Nos arquivos sequencias o compromisso de manter os registros fisicamente ordenados pelo valor da chave de ordenação, acarreta uma série de cuidados.

A operação de inserção, por exemplo, pode conduzir à necessidade do uso de áreas de extensão, além de periódicas reorganizações do BD.

Com o objetivo de conseguir um acesso eficiente aos registros, pode ser acrescentada uma nova estrutura para uso do BD, sendo ela definida como ÍNDICE.

O uso de índice é conveniente aos processos que precisam de um desempenho mais ágil, mesmo para os registros que não se encontrem ordenados fisicamente.

Exemplo:

Para uma consulta que deseje encontrar todas as contas somente da agência de Brasília têm-se que:

- ☐ refere-se a uma porção (ou fração) de todos os registros de contas;
- ☐ é ineficiente para o sistema ler todos os registros para localizar somente os que são desta agência;
- ☐ seria eficiente um acesso direto a esses registros;

Assim torna-se interessante acrescentar uma nova estrutura que permita essa forma de acesso.

O atributo, ou o conjunto de atributos, usado para procurar um registro em um arquivo é chamado de chave de procura ou chave de acesso.

Esta definição de chave difere das definições de chaves estudadas até agora (candidata, primária, ...). Por meio das chaves de procura será possível acrescentar em um arquivo várias chaves de acesso, sendo elas disponibilizadas de acordo com a necessidade existente na situação.

Geralmente, é interessante ter mais que um índice para um arquivo.

Exemplo:

- Imagine o catálogo de livros em uma biblioteca. Caso você deseje encontrar um livro de um autor específico, esta pesquisa será realizada sobre o catálogo de autores.
- O resultado desta pesquisa fornecerá um cartão que identificará onde encontrar o livro.
- Para ajudar na pesquisa sobre o catálogo, a biblioteca mantém os cartões em ordem alfabética.
- Com isso não será necessário pesquisar todos os cartões do catálogo para encontrar o cartão desejado.

- Em uma biblioteca são diversos os catálogos de procura (autor, título, assunto, entre outros).

Essa nova estrutura adicional consiste na definição de índices para os arquivos de dados, definindo os arquivos indexados.

nos arquivos indexados podem existir tantos índices quantas forem as chaves de acesso aos registros;

um índice irá consistir de uma entrada para cada registro do arquivo, sendo que as entradas encontram-se ordenadas pelo valor da chave de acesso;

cada índice é formado pelo valor de um atributo chave do registro e pela sua localização física no interior do arquivo;

a estrutura com a tabela de índices é também armazenada e mantida em disco.

Ainda assim, existirão grandes arquivos de índices que não seriam manipulados tão eficientemente (como no exemplo anterior “biblioteca”).

Por isso, algumas técnicas mais sofisticadas de índices são adotadas.

Dois tipos básicos de índices:

Ordenados: baseiam-se na ordenação dos valores

Hash: baseiam-se na distribuição uniforme dos valores determinados por uma função (função de hash)

Não existe uma técnica melhor, sendo cada técnica mais adequada para aplicações específicas.

Cada técnica deve ser avaliada sobre alguns fatores:

- tipos de acesso: encontrar registros com um atributo determinado ou dentro de uma faixa de valores
- tempo de acesso: tempo gasto para encontrar um item de dados ou um conjunto de itens
- tempo de inserção: tempo gasto para incluir um novo item de dados, incluindo o tempo de localização do local correto e a atualização da estrutura de índice
- tempo de exclusão: tempo gasto para excluir um item de dados, sendo incluído o tempo de localização do registro, além do tempo de atualização do índice
- sobrecarga de espaço: espaço adicional ocupado pelo índice, compensando este sacrifício pela melhoria no desempenho

ÍNDICES ORDENADOS

Para conseguir acesso aleatório rápido sobre os registro de um arquivo, pode-se usar uma

estrutura de índice.

Cada índice está associado a uma chave de procura (armazena a chave de procura de forma ordenada e associa a ela os registros que possuem aquela chave);

Os registros em um arquivo indexado podem ser armazenados (eles próprios) em alguma ordem;

Um arquivo pode ter diversos índices, com diferentes chaves de procura, estando ele ordenado sequencialmente pelo seu índice primário;

Normalmente, os índices primários em um arquivo são as chaves primárias, embora isso nem sempre ocorra

Índice Primário

Este índice consiste em um arquivo ordenado cujos registros são de tamanho fixo, contendo dois campos:

- 1º- mesmo tipo do campo chave no arquivo de dados
- 2º- ponteiro para o bloco do disco

O índice primário é um índice seletivo, pois possui entradas para um subconjunto de registros (bloco), ao invés de possuir uma entrada para cada registro do arquivo de dados.

Como o índice primário mantém ordenado os registros no arquivo, os processos de inserção e remoção são um grande “problema” para este tipo de estrutura.

ÍNDICE ORDENADO

Há dois tipos de índices ordenados (denso e esparso):

Índice Denso

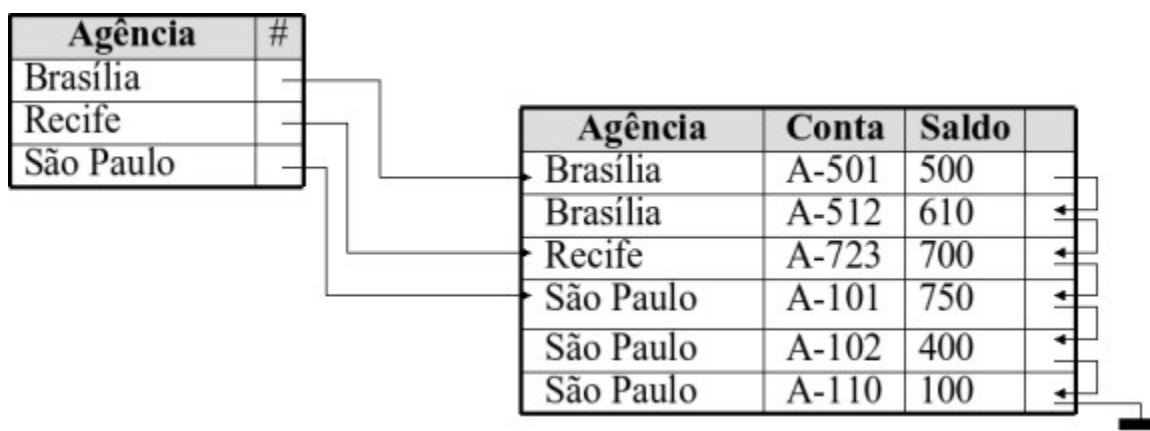
Um registro de índice aparece para cada valor distinto da chave de procura no arquivo;

O registro contém o valor da chave de procura e um ponteiro para o primeiro registro de dados com esse valor;

Alguns autores usam esta expressão para identificar quando um registro de índice aparece para cada registro no arquivo de dados.

Exemplo:

Índice denso para as agências bancárias.



Índice Esparso

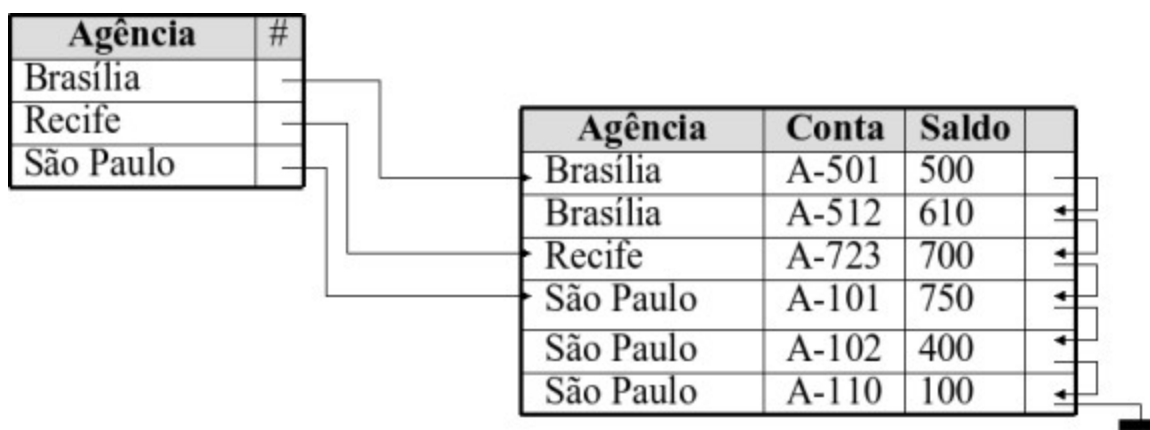
Um registro de índice é criado apenas para alguns dos valores do arquivo de dados;

Assim como os índices densos, o registro contém o valor da chave de procura e um ponteiro para o primeiro registro de dados com esse valor;

Localiza-se a entrada do índice com o maior valor da chave de procura que seja menor ou igual ao valor da chave de procura que se deseja. Inicia-se no registro apontado e segue-se pelos ponteiros até localizar o registro procurado.

Exemplo:

Índice esparso para as agências bancárias.



Uma nomenclatura também utilizada é a de índice de cluster, onde tem-se os registros de um arquivo, fisicamente ordenados por um campo não chave, podendo serem distintos ou não.

Índice Secundário

Consiste de um arquivo ordenado que não usa o mesmo campo de ordenação como índice.

São os índices cujas chaves de procura especificam uma ordem diferente da ordem sequencial do arquivo, podendo seus valores serem distintos para todos os registros ou não.

Os índices secundários melhoram o desempenho das consultas que usam chaves diferentes da chave de procura do índice primário, mas impõem uma sobrecarga significativa na atualização do BD.

O projetista do BD decide quais índices secundários são desejáveis baseado na estimativa da frequência de consultas e atualizações.

ÍNDICES DE NÍVEIS MÚLTIPLOS

Arquivo de índice muito grande para ser eficiente;

Registros de índices são menores que os registros de dados;

Índices grandes são armazenados como arquivos seqüenciais em disco.

☐ Uma busca em um índice grande também é muito onerosa.

Uma possível solução na agilização de grandes Índices

Criação de índice ESPARSO como índice primário;

Realização de busca binária sobre o índice mais externo (encontrar maior valor da chave de procura que seja menor ou igual ao valor desejado);

O ponteiro indica o bloco do índice interno que será “varrido” até se encontrar o maior valor da chave de procura que seja menor ou igual ao valor desejado;

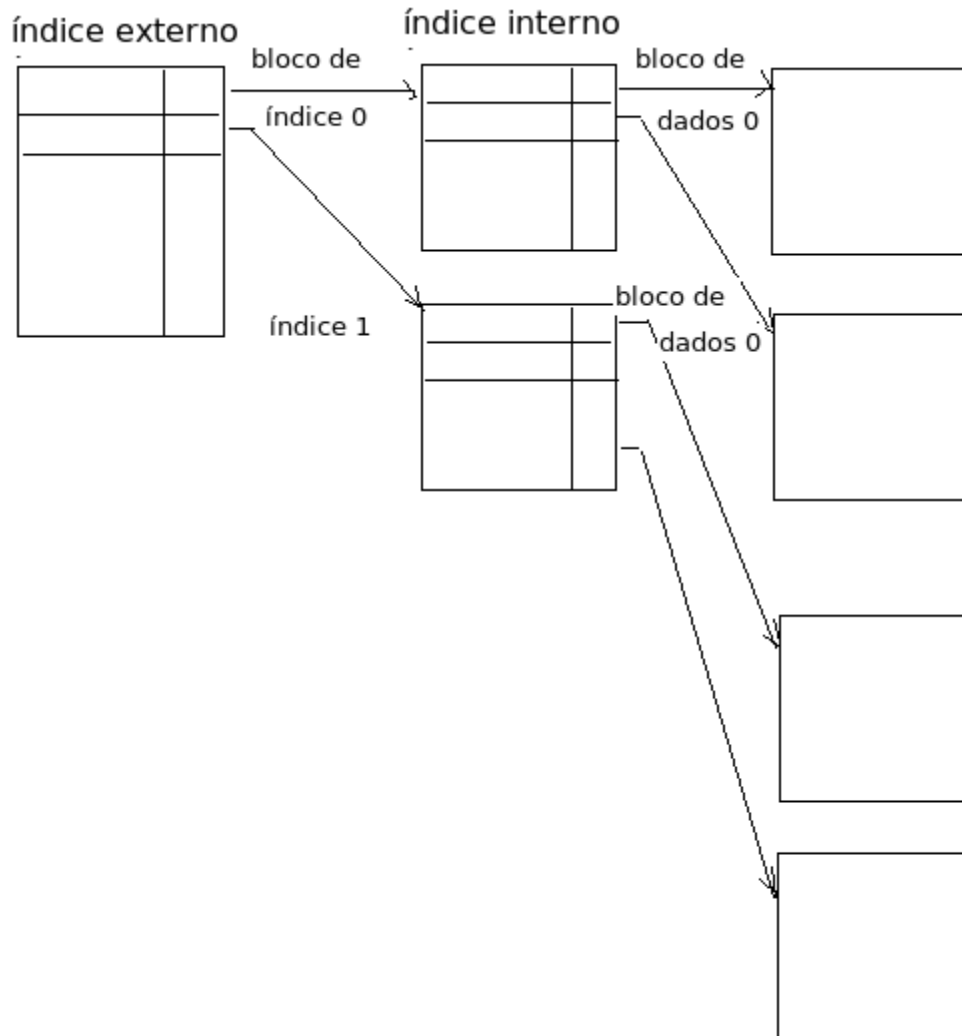
O ponteiro neste registro indica o bloco do arquivo que contém o registro procurado.

Observe a representação a seguir:

Utiliza-se dois níveis de indexação;

- caso o nível mais externo ainda seja muito grande;
- não cabendo total-mente na memória principal;
- pode ser criado um novo nível;

- criar quantos níveis forem necessários.



BUSCA ou PROCURA usando Níveis Múltiplos

Requer um número significativamente menor de operações de E/S que a busca binária;

Cada nível do índice corresponde a uma unidade de armazenamento físico (trilha, cilindro, ou disco);

□ Os níveis de índices múltiplos estão relacionados as estruturas de árvores, como as árvores binárias usadas na indexação de memória.

ATUALIZAÇÃO DE ÍNDICE

Independente do tipo de índice utilizado, ele deve ser atualizado sempre que um registro for inserido ou removido do arquivo de dados.

REMOÇÃO

Localizar o registro;
Remover o registro;
(se houver)
Atualizar o índice;
(denso ou esperso)

INSERÇÃO

Localizar o registro;
(usa chave de procura)
-Incluir os dados (registro)
Atualizar o índice
(denso ou esperso)

ORGANIZAÇÃO DE ARQUIVOS COM HASHING

Para uma localização de dados eficiente, em arquivos sequenciais, é necessário o acesso a uma estrutura de índice ou o uso da busca binária, que acarreta em mais operações de E/S.

Outra alternativa seria a aplicação de técnicas de hashing, onde os arquivos seriam organizados por meio de uma função de hash.

Com o uso destas técnicas, evitaria-se o acesso a uma estrutura de índice;

O hashing também proporciona um meio para construção de índices;

A organização de arquivos hashing oferece:

Acesso direto ao endereço do bloco de disco que contém o registro desejado;

Aplica-se uma função sobre o valor da chave de procura do registro para conseguir-se a identificação do bloco de disco correto;

Utiliza-se do conceito de bucket (balde) para representar uma unidade de armazenamento de um ou mais registros;

Normalmente equivalente a um bloco de disco, porém ele pode denotar um valor maior ou menor que um bloco de disco.

FUNÇÕES HASH

Uma função Hash ideal distribui as chaves armazenadas uniformemente por todos os buckets, de forma que todos os buckets tenham o mesmo número de registros.

No momento do projeto, não se sabe precisamente quais os valores da chave de procura que serão armazenados no arquivo.

Deseja-se que a função de hash atribua os valores da chave de procura aos buckets atentando a uma distribuição:

- UNIFORME

- ALEATÓRIA (o valor de hash não será correlacionado a nenhuma ordem visível externamente – alfabética por exemplo – parecendo ser aleatória)

Observe a escolha de uma função hash sobre o arquivo conta, usando a chave de procura nome da agência.

Decidiu-se a existência de 27 buckets (por estado);

Função simples, mas não distribui uniformemente os dados (SP tem mais conta que MA), além de não ser aleatória;

Neste outro exemplo, aplica-se as características típicas das funções de hash.

Efetuem cálculos sobre a representação binária interna dos caracteres da chave de procura;

Uma função simples calcularia a soma das representações binárias dos caracteres de uma chave de procura, retornando o módulo da soma pelo número de buckets;

CONCLUSÃO SOBRE AS HASH

O emprego destas funções requerem um projeto cuidadoso.

Função Hash “RUIM”: pode resultar em procuras que consumam um tempo proporcional a quantidade de chaves no arquivo;

Função Hash “BOA”:oferece um tempo de procura médio (constante pequena), independente da quantidade de chaves de procura no arquivo;

OVERFLOW DE BUCKET

Até o momento supôs-se que os buckets sempre tem espaço para inserir um novo registro, porém isso pode não acontecer. Quando o bucket não possuir espaço suficiente, diz-se que ocorreu um overflow de bucket.

Este overflow pode acontecer por várias razões, sendo algumas delas apresentadas a seguir:

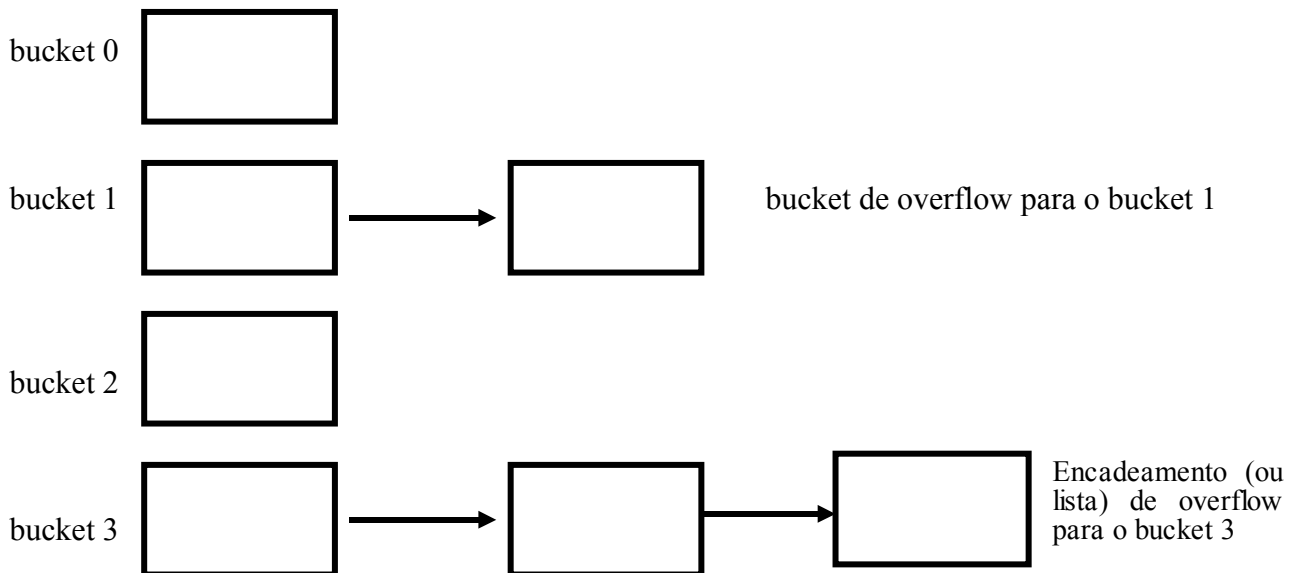
Buckets insuficientes: conhecer o total de registro quando a função de hash for escolhida;

Desequilíbrio (skew): pode acontecer por duas razões:

- registros múltiplos com a mesma chave de procura;
- distribuição não uniforme realizada pela função de hash escolhida.

Baseado em alguns cálculos é possível reduzir a probabilidade de overflow de buckets, processo este conhecido como fator de camuflagem (ou fator de fudge).

Alguns espaços serão desperdiçados no bucket, cerca de 20% ficará vazio, mas o overflow será reduzido. Ainda sim, o overflow de bucket poderá acontecer, por exemplo:



Cuidados com o algoritmo de procura para o overflow:

A função de hash é usada sobre a chave de procura para identificar qual o bucket do registro desejado;

Todos os registros do bucket devem ser analisados, quando a igualdade com a chave de procura acontecer, inclusive nos bucket de overflow;

Existem outras variações na aplicação das técnicas de hashing (hashing aberto, linear probing, ...), porém a técnica descrita anteriormente é preferida para banco de dados (aspectos referentes aos problemas com a remoção).

Desvantagens desta Técnica

A função de hash deve ser escolhida até o momento no qual o sistema será implementado;

Como a função de hash mapeia os valores da chave de procura para um endereço fixo de bucket, ela não poderá ser trocada facilmente, caso o arquivo que esteja sendo indexado aumente muito ou diminua;

ATIVIDADE EM GRUPO

Temas (livro do Korth capítulo 11):

- 1- Arquivos de Índice Árvore B+ (pág.347 – 11.3)
- 2- Arquivos de Índice Árvore B (pág.356 – 11.4)
- 3- Índices Hash (pág.362 – 11.5,2)
- 4- Arquivos Grid (pág.374 – 11.9.1 e 11.9.2)

Atividade (painel integrado):

- desenvolver o estudo dos temas pelo grupo
- discutir e anotar os conteúdos mais relevantes
- trocar os membros de cada grupo que explicaram aos novos companheiros de grupo o material estudado

INDEXAÇÃO ORDENADA X HASHING

Cada esquema representa vantagens e desvantagens para determinadas situações (inclusive para os arquivos heap – arquivos que não são ordenados de nenhuma maneira em particular);

Tipo de consulta também é um fator importante:

Igualdade (... where atributo = valor)

em hash o tempo médio de procura é uma constante independente do tamanho do BD;

Faixa de valores (... where atributo < valor)

na ordenada localiza-se o valor e retorna-se todos os valores seguintes do bucket até atingir o valor limite, enquanto que o hash é aleatório e terá que localizar cada valor;

Material para Consulta e Apoio ao Conteúdo

ELMASRI, R. e NAVATHE, S. B., Fundamentals of Database Systems - livro
Capítulo 6

SILBERSCHATZ, A., KORTH, H. F., Sistemas de Banco de Dados - livro
Capítulo 11

Melhorando o Desempenho de Consultas com Adição de Índice

1) Melhorar o desempenho de consultas, adicionando índices aos campos da cláusula where (em consultas que retornam muitos registros), de tabelas com muitos registros, como exemplo:

Observe que a tabela de ceps não está normalizada, quase todos os campos deveriam pertencer a outras tabelas, como é o caso de tipo, logradouro, bairro, municipio e uf. Como também vários campos deveriam conter índices para agilizar as consultas.

Exemplifiquemos adicionando um índice em uf, mas antes de adicionar mediremos o desempenho e

após também.

2) Criar uma estrutura semelhante à do arquivo .csv que queremos importar:

```
create table cep_full
(
    cep char(8),
    tipo char(72),
    logradouro char(70),
    bairro char(72),
    municipio char(60),
    uf char(2)
);
```

3) Importar para a tabela criada todo o conteúdo do CSV (Comma Separated Values):

Linux:

```
\copy cep_full from /home/ribafs/cep_brasil_unique.csv
```

Windows:

```
\copy cep_full from c:/teste/cep_brasil_unique.csv
```

4) Fazer uma cópia da tabela cep_full:

```
create table cep_full_index as select * from cep_full;
```

5) Adicionar chave primária à tabela criada:

```
alter table cep_full_index add constraint cep_pk primary key(cep);
```

Vamos planejar uma consulta, mas antes atualizemos as estatísticas internas do planejador:

```
analyze cep_full_index;
```

6) Testar um plano de consulta pela UF:

```
explain select count(uf) from cep_full_index where uf='CE';
```

Analyze os resultados. Uma observação importante é o fato de não usar índice, mas uma busca sequencial.

Veja também os valores dos custos, que são relativamente altos.

Observe que o comando explain não executa a consulta, apenas mostra o plano interno de execução da mesma,

por isso seu resultado é instantâneo por maior que seja a tabela.

Vamos então criar um índice para o campo uf para ver o resultado:

```
create index idx_uf on cep_full_index(uf);
```

Agora repetir a execução do plano de consulta anterior para ver como se comporta com índice:

```
explain select count(uf) from cep_full_index where uf='CE';
```

Veja que os custos da função count caíram pela metade enquanto que os custos da varredura, que agora usa índice, caíram de 35.457 para 173, o que representa menos de 0.5%.

Agora faremos um teste aparentemente mais realista:

```
create table t1 as select * from cep_full;
alter table t1 add constraint cep_pk primary key(cep);
analyze t1;
\timing
select count(uf) from t1 where uf='CE';

create index idx_uf on t1(uf);
select count(uf) from t1 where uf='CE';
```

Veja que gastou em torno de 10% de quando não havia índice.

Então, sempre que se deparar com uma consulta em tabelas com grande quantidade de registro não existe em usar índices nos campos do where. Caso ainda tenha alguma dúvida já sabe o que fazer, elabore um plano de execução de duas consultas (uma com e outra sem índice no campo) e veja o que o PostgreSQL fará nos dois casos.

Teoria sobre Índices e uso no PostgreSQL:

Uma boa indicação é a série de tutoriais do Daniel Oslei, no site iMasters (<http://www.imasters.com.br>).

Garantindo desempenho com o operador LIKE
De Rodrigo HJort na SQL Magazine 52

Consultas em campos string que contenham acentos não melhoram com a criação de índices simples.

Para isso devemos criar índices especiais:

A criação de índice comum em campos nem leva a uma busca indexada pelo planejador.

```
create index nome_idx on clientes(nome varchar_pattern_ops);
```

Com o índice acima realmente podemos melhorar o desempenho de consultas por campos acentuados em LATIN1.

Veja que ainda existe um outro porém, para consultas por esse campo (nome) usando LIKE, apenas será

melhorado o desempenho se usarmos uma consulta do tipo:

```
select * from clientes
  where nome LIKE 'JOSÉ CARLOS %';
```

Com o caractere coringa % no final da string de busca.
Caso o caractere esteja no início já não resolverá.

Tem mais: caso desejemos usar alguma função como upper ou lower na busca, estas funções deverão fazer parte do índice para que tenhamos um desempenho melhorado, ou seja, para que o planejador utilize o índice.

Quando o caractere coringa está no início da string de busca devemos encontrar uma forma para que o planejador use o índice e uma delas é criar um índice reverso.

Opção - criar uma função em plperl:

```
create language plperl;

create or replace function reverse(vchar) returns varchar as $$
  return reverse $_[0];
$$ language plperl immutable strict;

select reverse('Joao Brito Cunha');

create index idx_nome2 on clientes (reverse(nome));
create index idx_nome3 on clientes (reverse(nome) varchar_pattern_ops);
analyze clientes;
```

Nos testes o planejador irá utilizar o índice idx_nome3, devido ser LATIN1.

Obs.: Um índice reverso irá ler os nomes do final para o começo, ao contrário do índice normal. "Joao" será lido como "oaoJ".

Caso utilizemos o caractere coringa tanto no início quanto no final da string de busca, então os métodos atuais não ajudarão em termos de desempenho.

3) Modelagem de Dados

Artigos de Mauri Gonçalves no iMasters

Modelagem de Dados no Access

Neste e nos próximos artigos vou falar um pouco sobre um assunto que muitos desconhecem: a Modelagem de Dados.

Primeiramente, o que é modelagem de dados? Para quem não sabe, a modelagem de dados é um processo no qual você "projeta" ou "planeja" a sua base de dados de forma que você possa aproveitar os recursos do Gerenciador de Banco e também para que possa construir um banco de dados consistente, que reaproveite recursos, que exija menos espaço em disco e, sobretudo, que possa ser bem administrado.

Assim, como no processo de software, a modelagem de dados é um processo que possui etapas a serem seguidas, mas que podem ser superadas, dependendo do tipo de banco que se pretende construir. O documento principal da modelagem de dados é o Diagrama de Entidade-Relacionamento - DER (leia-se: dér) ou Modelo de Entidade-Relacionamento (MER). Neste documento, são representadas as entidades e os relacionamentos entre elas. As entidades são os "embriões" das tabelas do banco. Até avançarmos esta fase da modelagem, elas recebem esta nomenclatura.

Esta primeira fase é o que chamamos de Modelagem Lógica. É quando determinamos o fluxo de dados entre as entidades, isto é, como o próprio nome diz, quando determinamos a lógica do banco que iremos contruir.

O relacionamento entre as entidades é um quesito que deve ser especialmente analisado. No modelo lógico, toda entidade deve estar relacionada à outra. Quando sobram entidades sem relacionamento, é sinal de que há algum problema. Podem ser entidades que estão sobrando, ou seja, que na verdade não deveriam existir, ou alguma entidade pode estar relacionada à qual não deveria.

Dependendo do tipo de base de dados que se deseja, pode-se aproveitar ferramentas do próprio SGBD e, desta forma, economizar linhas de código.

Suponha que façamos um controle de bens domésticos. Certamente para este sistema não há previsão de migrar a base de dados para uma plataforma maior, como o SQL Server, ou Oracle, certo? Então por que não aproveitar alguns recursos do SGBD Access para controlar os seus dados? Isto deve ser levado em conta quando se modela um banco. Mas há também casos onde se prevê uma migração ou, quem sabe, se está apenas pensando em um módulo de um sistema.

Neste caso, quanto menos dependência do gerenciador do banco, melhor. Pode-se implementar rotinas no próprio sistema e torná-lo "universal" a qualquer tipo de banco de dados, seja ele proprietário (Access, SQL Server, Oracle) ou livre (MySQL, PostgreSQL).

Em plataformas como a Oracle, há módulos de modelagem de dados próprios, totalmente integrados ao banco de dados. Como sabemos, este não é o caso do Access. Mesmo assim, podemos fazer uma análise, por mais simples que seja o banco de dados, antes de colocar Access para rodar.

Após terminarmos a modelagem lógica, partimos para a modelagem física. Nesta etapa, vamos determinar as tabelas, campos e relacionamentos que efetivamente vamos contruir em nosso banco

de dados. Para isto, vamos repensar o modelo lógico que criamos e adequá-lo para a "realidade". Após esta visão geral, vamos nos aprofundar mais na modelagem voltada para o Access.

Modelagem de Dados 1: Entidades

Olá pessoal! Depois da visão geral ([artigo anterior](#)), vamos por a mão na massa e iniciar o Modelo Lógico do nosso banco de dados. Para que os conceitos fiquem bem visíveis, vou exemplificar para vocês um banco de dados para um sistema de Biblioteca.

O start da modelagem se dá a partir das **ENTIDADES**. Uma entidade é uma representação de um conjunto de informações sobre determinado conceito do sistema. Toda entidade possui **ATRIBUTOS**, que são as informações que referenciam a entidade.

Para exemplificar no sistema de controle de Biblioteca, partimos do conceito principal que é o empréstimo de obras por usuários da biblioteca. A partir deste conceito inicial, vamos ramificando e descobrindo novos conceitos. Podemos iniciar nosso raciocínio da seguinte forma:

"Uma biblioteca possui **Obras literárias** que podem ser tomadas em **empréstimos** pelos **usuários** credenciados."

Podemos rapidamente enxergar um *cadastro de livros*, um *cadastro de usuários* e um *registro de empréstimos*, certo? É essa visão que temos que ter ao modelarmos um banco, isto é, devemos detectar as informações que devemos armazenar.

Para identificar se aquele conceito pode ser uma entidade você deve apenas se perguntar: "Eu desejo armazenar quais informações sobre este conceito ?" Se houverem informações a serem armazenadas, você tem uma **ENTIDADE**. Exemplificando: Eu desejo armazenar os seguintes dados do livro: Título, Autor, Editora, Ano, Edição e Volume. Temos então a entidade Livro.

A cada uma destas informações que armazenamos, damos o nome de ATRIBUTO. Um atributo pode ser uma informação única, bem como pode ser um conjunto de informações. Exemplificando: Sobre empréstimos, eu tenho os seguintes atributos: *Código, livro emprestado, usuário que emprestou, data de empréstimo e data de devolução*. O atributo "livro emprestado" refere-se ao livro, porém sabemos que há informações que devem ser armazenadas sobre livros como vimos antes. Temos aí um exemplo de um atributo que é um conjunto de outros atributos: todos os atributos da entidade Livro, formam um atributo da entidade Empréstimo.

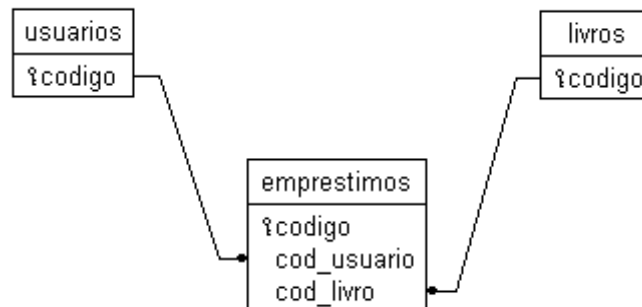
Outra situação é a seguinte: deseja-se armazenar 2 números de telefone para cada usuário. Telefone é um dos atributos da entidade Usuário. Neste caso, não temos referência à nenhuma outra entidade, ou seja, temos mais de uma informação para o mesmo atributo. A este atributo damos o nome de **ATRIBUTO MULTIVALORADO**, pois temos 2 valores para o mesmo atributo em uma mesma entidade. Sabemos que nos bancos de dados não é possível armazenar mais de uma informação no mesmo campo. Por isso, veremos mais à frente uma solução para os atributos multivalorados.

Quando os atributos de uma entidade formam o atributo de outra, poderemos dizer que existem uma referência entre as entidades. Naquele atributo da entidade Empréstimos vamos armazenar apenas uma referência à entidade Livro. O mesmo ocorrerá com relação à entidade Usuário. Pelo simples fato de existir esta referência de uma entidade em outra, temos então o que chamamos de **RELACIONAMENTO**.

Neste começo, temos um rascunho do nosso Diagrama de Entidade-Relacionamento:

Entidades: Usuário, Livro, Empréstimo

Relacionamentos: Usuário - Empréstimo, Livro - Empréstimo



Modelagem de Dados 2 - Os Relacionamentos

Agora que definimos nossas entidades, vamos falar sobre os relacionamentos entre elas.

Na matéria anterior vimos como identificar os relacionamentos entre entidades, porém devemos observar alguns aspectos importantes que sinalizam erros na modelagem:

01. Quando "sobram" entidades sem relacionamentos;

02. Quando ocorrem "ciclos fechados" de relacionamentos;

Exemplo: Usuário relaciona-se com Empréstimo que relaciona-se com Livro que relaciona-se com Usuário que relaciona-se com Empréstimo, etc...

03. Entidades com muitos atributos relacionando-se com entidades com apenas alguns atributos;

04. Muitas entidades relacionando-se à uma mesma entidade;

É importante atentar para esses erros para que não haja acúmulo de inconsistências e que não torne a modelagem um processo problemático. Mas não se preocupe em resolver tudo de uma vez só, pois mais a frente veremos formas de VALIDAR nossa modelagem, de maneira que os erros não passem despercebidos.

Determinados os relacionamentos, temos que verificar o número de referências de uma entidade em outra, ou seja, agora vamos verificar a CARDINALIDADE dos relacionamentos. Vejamos as possibilidades:

- Relacionamento Um-Para-Um (1:1)

Uma instância da entidade A relaciona-se a uma instância da entidade B

- Relacionamento Um-Para-Vários (1:N)

Uma instância da entidade A relaciona-se a várias instâncias da entidade B

- Relacionamento Vários-Para-vários (N:M)

Várias instâncias da entidade A relacionam-se a várias instâncias da entidade B

Estas 3 cardinalidades apresentadas acima são implicitamente, CARDINALIDADES MÁXIMAS.

mas pode-se determinar a CARDINALIDADE MÍNIMA, que pode ser descrita desta forma:

- Relacionamento Um-Para-Vários (0,1:1,N)

Nenhuma ou uma instância da entidade A relaciona-se á uma ou várias instancias da entidade B.

Exemplo de relacionamento Um-Para-Vários com cardinalidade mínima:

Usuario - Empréstimo (1,1:0,N)

- Um usuário pode estar relacionado a nenhum ou a vários empréstimos.
- Um empréstimo deve estar relacionado a somente um usuário.

Conclusão: Pode ser que um usuário nunca faça empréstimos, assim como pode haver usuário que faça vários empréstimos, porém um empréstimo obrigatoriamente tem que ser feito por um único usuário.

Exemplo de relacionamento Vários-Para-Vários com cardinalidade mínima:

Empréstimo - Livro (0,N:1,N)

- Um empréstimo pode estar relacionado a um ou a vários livros.
- Um livro pode estar relacionado a nenhum ou a vários empréstimos.

Conclusão: Pode ser que um livro nunca seja emprestado, assim como pode haver livros que tenham sido emprestado várias vezes, porém um empréstimo deve conter pelo menos um livro ou pode conter vários.

A Cardinalidade Mínima pode ser incluída no Modelo Lógico, mas é pouco utilizada por ser, muitas vezes, redundante e óbvia, mas muito é útil no que refere à expôr a clareza dos relacionamentos entre entidades.

Se você está em dúvida quanto á quem é o lado "Um" e quem é o lado "Vários" no seu relacionamento, use as seguintes dicas:

Relacionamentos 1 para N: no 1 o campo é a PK. No N o campo é a FK.

Relacionamentos 1 para 1: no 1 o campo é a PK. No outro 1 o campo é também uma PK.

Relacionamentos N para N: no primeiro N o campo é FK. No outro N o campo é também uma FK.

- Veja em qual das entidades está o atributo que faz referencia à outra entidade. Esta entidade onde está o atributo é o lado "Vários" a outra é o lado "Um", trata-se então de um relacionamento Um-Para-Vários.

- Se ambas as entidades tiverem atributos que referenciam uma à outra, então ambas possuem a cardinalidade "Varios", isto é, trata-se de um relacionamento Vários-Para-Vários.

Estas 2 regrinhas funcionam como fórmulas. Apenas aplique-as ao seu relacionamento, sem pensar muito. Mesmo que fique um pouco confuso para você, mais pra frente verá que elas são válidas.

Descrição do nosso Diagrama de Entidade-Relacionamento:

Entidades: Usuário, Empréstimo, Livro

Relacionamentos: Usuario-Emprestimo(1:N), Emprestimo-Livro(N-N)

Na próxima matéria mostrarei para vocês o Diagrama de Entidade-Relacionamento propriamente dito e as ferramentas de software que você pode utilizar na modelagem do seu banco de dados.

Modelagem de Dados - Validação do modelo ER

Nesta terceira parte da nossa série, vou falar sobre a validação do Diagrama Entidade-Relacionamento: o DER. Tendo definido as entidades, seus atributos e relacionamentos, vamos fazer uma verificação em nosso modelo ER em busca de falhas. É nesta fase que vamos validar nosso modelo ER e corrigir falhas em relacionamentos e possíveis entidades que deveriam ou não existir.

Nesta parte, também começamos a visualizar o modelo físico do banco de dados, onde as entidades viram tabelas e os atributos viram campos. Abaixo estão descritos os erros mais frequentes ocorridos no modelo:

Associações Incorretas: No modelo ER sempre pensamos nas associações entre entidades não entre atributos. Seria incorreto por exemplo associar ao Livro (voltando ao nosso modelo Biblioteca) o nome do usuário que o tomou emprestado. Não se pode associar o nome do usuário ao livro, e sim o usuário como um todo. Verifique se não há associações entre atributos no seu modelo. Faça sempre associações entre Entidades.

Usar uma entidade como atributo de outra: No modelo lógico, as associações ainda se dão de forma abstrata. É incorreto colocar na entidade Empréstimo o atributo Usuário, sendo que Usuário é uma entidade associada e não um atributo e isso vale para qualquer outra entidade relacionada. Essa regra costuma gerar muita controvérsia porque costumamos confundir o conceito com o modelo físico onde criamos as Chaves - mas este assunto veremos mais à frente.

Usar o número incorreto de entidades em um relacionamento: Verifique sempre os casos de mais de uma entidade associada à mesma entidade. Nestes casos, o que surge é um relacionamento redundante e desnecessário.

Depois de verificados e corrigidos os erros nas associações (ou relacionamentos) devemos verificar se o modelo está completo. Nele devem ser expressadas todas as entidades, com seus atributos e relacionamentos. Para isso, verifique se é possível obter todas as informações desejadas a partir do modelo construído e também se todas as transações de dados podem ser executadas. Se tudo estiver ok, partimos para o próximo passo.

Verificar redundâncias: Um modelo de banco de dados deve ser "mínimo", ou seja, não apresentar nenhum tipo de redundância. Para verificar a redundância nas transações com dados, analise a função de cada relacionamento e reveja os que fizerem operações muito semelhantes ou iguais. Verifique também se há ciclos associativos, por exemplo:

"A é associado a B que é associado a C que é associado a A"

Neste caso, há uma redundância na parte "C que é associado a A", pois C já é associado a A através da entidade B, ou seja, este exemplo representa uma associação desnecessária e pode ser descartada sem nenhum prejuízo para o modelo. O correto seria: "A é associado a B que é associado a C"

Além disso, podem haver atributos redundantes nas entidades. Atributos redundantes são aqueles que podem ter uma nomenclatura diferente porem eles armazenam os mesmo dados, a mesma informação. Verifique a existencia destes atributos e elimine-os do modelo.

Outro item considerável é o aspecto de tempo do banco de dados. Quando construímos modelo ER, pensamos somente na situação momentânea do banco de dados. Para corrigir isso, devemos verificar atributos e relacionamentos que podem ser alterados durante a utilização do banco de dados.

Um exemplo ótimo de atributo, porém fora do nosso caso da Biblioteca, é o Empregado.

Suponha a existência de uma entidade Empregado e o atributo Salario. A partir dessa entidade, pode-se criar uma entidade Salario com o atributo Data. Desta forma, obtém-se um histórico de salários do empregado.

Um exemplo de associação pode ser a associação entre Empregado e Departamento. Adicionando um atributo Data neste relacionamento, teremos um histórico de departamentos por onde o Empregado passou.

Também é importante levar em conta o armazenamento de informações antigas. Para evitar o crescimento demasiado do banco de dados, podem ser eliminadas informações antigas ou podem ser reinseridas no banco de outra forma. Por isso, é importante pensar no caso de remoção de dados da base, que outras informações seriam comprometidas. Deve-se planejar como guardar informações antigas ou que com o passar do tempo não venham mais a ser utilizadas, como, por exemplo, informações que serão usadas somente para cálculos estatísticos ou visões.

Pode ocorrer também a existência de entidades isoladas, isto é, sem nenhuma associação. A ocorrência destas entidades não é de todo incorreto, mas, na prática, elas acabam sendo descartadas, por isso verifique se ela não faz parte de alguma associação incorreta ou que não tenha sido feita.

Lembre-se que por mais simples que seja seu modelo ER, você deve fazer a validação completa. Este é o momento de corrigir erros e falhas de modelagem porque quando identificadas na modelagem física você acaba tendo que voltar a esse passo para fazer a arrumação do problema.

No próximo artigo, começamos a falar sobre a Modelagem Física.

Modelagem de Dados (Parte 04) - Abordagem Relacional

A grande maioria dos Sistemas Gerenciadores de Banco de Dados (ou simplesmente SGBD's) são Relacionais. Os bancos Relacionais são compostos por Tabelas ou Relações. Este termo "Relações" é mais usado na literatura original sobre abordagem relacional (daí a denominação "Relacional"), enquanto que "Tabela" é um termo mais prático e mais usado em produtos comerciais. Para não haver confusão de conceitos, vou usar a terminologia "Tabela".

Uma Tabela é um conjunto não ordenado de linhas ou (tuplas - terminologia original) e cada linha possui campos (atributos). Cada campo é identificado por um nome de campo (ou nome de atributo) e o conjunto de linhas de uma tabela, que possuem o mesmo nome chama-se coluna. Para muitos, nada disso é novidade, mas não nos custa nada lembrar os conceitos.

Modelo lógico: relação, atributo, tupla

Modelo físico: tabela, campo, registro

Algumas características específicas de uma tabela de banco de dados:

- As linhas da tabela não são ordenadas. A recuperação de linhas em uma tabela acontece arbitrária, ou seja, a recuperação será da forma como está no banco e pronto, a não ser que a ordem seja especificada pelo programador na sua consulta.

- Os valores de campo de uma tabela de banco de dados são mono-valorados. Pode-se apenas ter 1

valor em um campo, não sendo possível armazenar "coleções" como Arrays (ou Vetores) por exemplo.

Obs.: No PostgreSQL um campo pode armazenar informações do tipo array.

- A linguagem de consulta ao banco de dados permite o acesso por qualquer critério envolvendo os campos de uma ou mais linhas. Em um arquivo de dados por exemplo é necessário um índice ou esquema de ponteiros para buscar algum valor. Na verdade índices também existem nos bancos de dados, mas isso não é considerado pelo programador nas consultas às tabelas.

Um item primordial quando falamos sobre relacionamento entre linhas de tabelas é a CHAVE. A chave primária é formada por um ou mais campos de uma tabela e serve como identificador de uma linha. Porém a chave primária deve ser sempre mínima, ou seja, ser composta pelo menor número de campos possível (no mínimo 1). Veremos mais adiante o uso de chaves com mais de um campo, que recebem o nome de "chave composta".

Vistos os conceitos básicos de um banco relacional, temos que considerar alguns pontos antes de iniciarmos a transformação de um modelo ER em um modelo físico para o banco de dados:

- O modelo ER não considera implementação com nenhum SGBD. É muito comum, surgirem modificações no esquema lógico para possibilitar a criação do modelo físico e usar os recursos do SGBD.

- Ao construir o seu banco físico, você deve considerar fatores como:

- Diminuir o número de acessos a disco: A cada consulta à tabela, todas os campos da linha são carregados para a memória, mesmo que você utilize apenas 1 deles)

- Evitar junções: Em muitas consultas a banco, são feitas junções de dados em várias linhas de tabelas. Utilize todos os dados necessários de preferência em uma única linha diminuindo o número de junções, pois uma junção acaba envolvendo vários acessos a disco.

- Diminuir o número de chaves primárias: Fatalmente você acabará juntado algumas tabelas que você dividiu durante a modelagem lógico. Isso porque a presença de chaves em locais diferentes, armazenando a mesma informação acaba virando mais espaço em disco utilizado e mais processamento.

- Evitar campos opcionais: Tecnicamente um campo vazio no banco de dados não ocupa espaço em disco, devido às técnicas de compressão de dados existentes nos SGBDs de hoje. Mas o problema surge quando a obrigatoriedade ou não do preenchimento de um campo depende do valor de outro campo.

Este fator acaba sendo resolvido via programação, ou seja, pelo sistema que vai acessar aquele banco e isso deve ser evitado. Algumas medidas de integridade de dados podem ser tomadas já no banco, deixando o mínimo de campos que podem assumir o valor NULL.

A transformação do modelo ER em um modelo relacional segue as seguintes etapas:

- Tradução de Entidades e Atributos
- Tradução de Relacionamentos e respectivos atributos
- Tradução de generalizações/especializações

Inicialmente, a tradução de entidades é razoavelmente óbvia: uma entidade gera uma tabela. Cada atributo da entidade gera uma coluna e os atributos identificadores da entidade tornam-se chave

primária. Essa é uma tradução inicial. No decorrer da transformação entre modelos, algumas tabelas ainda poderão ser fundidas ou divididas.

Mesmo assim, não recomendável apenas transcrever os nomes de atributos como nomes dos campos. Lembre que agora estamos falando de tabela física, de campos que serão acessados via programação ou seja, temos agora uma nova visão da situação. É nessa fase que, por questões de boa prática e organização no processo de modelagem deseja-se que sejam definidos alguns "padrões" para nomes de campos, abreviaturas e sempre primar por nomes curtos para os campos, ou seja das colunas da tabela.

Para deixar mais claro a forma como voce deve fazer isso, vou mostrar um exemplo:

Tem-se a entidade Pessoa com os atributos: Codigo, Nome, Endereço e Data de Nascimento. Código é o atributo identificador. Um exemplo de "tradução" dos campos seria:

Tabela: Pessoa - Campos: CodPessoa, NomePess, EnderecoPess, DataNascPess

Utilizei o seguinte padrão:

- Para a chave primária, utilizei o prefixo "Cod" + o nome da tabela.
- Para os demais campos utilizei o mesmo nome + o sufixo "Pess" em todos eles, identificando-os como sendo da tabela Pessoa.

Mas há uma boa justificativa para esse sufixo "Pess" nos nomes:

Ipoteticamente em uma instrução SQL pode haver uma junção da tabela Pessoa com a tabela Departamento. Departamento também possui um campo Nome. É recomendável não utilizar o mesmo nome de campo em tabelas diferentes para não gerar a seguinte situação:

```
SELECT Pessoa.Nome, Departamento.Nome FROM Pessoa, Departamento (...);
```

Na seleção SQL de campos o mesmo nome, deve-se especificar o nome da tabela de origem, o que pode tornar a cláusula muito longa, visto que consultas muitos mais complexas que essa são feitas com muita frequência.

```
SELECT NomePess, NomeDept FROM Pessoa, Departamento (...);
```

A utilização do sufixo para os mesmos campos de tabelas diferentes, nos poupa o trabalho de incluir o nome da tabela de origem, tornando a cláusula SQL mais limpa e legível, além de tornar o campo reconhecido em qualquer consulta: (Todos os campos com final "Pess" serão reconhecidos como sendo da tabela Pessoa)

Esse exemplo mostra que vários fatores estão envolvidos na transformação para o modelo relacional. No proximo artigos seguiremos com a proxima etapa da tradução: Relacionamentos e Atributos..

Parte 05 (Transformação entre Modelos)

Olá caros leitores! Vamos em frente com a série de artigos sobre Modelagem de Dados. Essa parte da transformação entre modelos é a parte mais interessante da modelagem. Aqui começa "de verdade" a surgir, finalmente, o banco de dados propriamente dito. Em especial a tradução dos relacionamentos (assunto que começo a abordar hoje) é o que deixa mais claro os conceitos vistos na modelagem conceitual, ou modelagem lógica.

Um relacionamento nada mais é do que uma "ligação" entre duas entidades (agora, Tabelas) que juntas criam informações complexas. Como o assunto agora é tabela física do banco de dados, nós temos que construir nossa base de forma "conectada". Para isso nós usamos as famosas CHAVES. As chaves tem variações e diferentes finalidades. De forma rápida vou explicar um pouco sobre cada tipo de chave:

Chave Primária: A chave primária é o que chamávamos na modelagem lógica de "atributo identificador". Geralmente é um campo da tabela que armazena uma informação única, que não se repete em nenhum outro registro daquela mesma tabela. Desta forma ele serve como identificador daquele registro.

Chave Composta: A chave composta é formada pela chave primária e por alguma outra informação que também é única na tabela. Os dois campos juntos, formam uma chave composta. Este tipo de chave é usado com mais frequência em tabelas provenientes de relacionamentos N:N [vários para vários] (veremos mais á frente).

Chave Estrangeira: Esta chave é um campo em uma tabela que armazena o conteúdo da chave primária de outra tabela. Chave estrangeira é sinônimo de relacionamento entre tabelas. Se há relacionamento há chave estrangeira.

A primeira coisa que define a forma como voce vai traduzir o relacionamento é a cardinalidade das entidades (assunto abordado no começo da série).

Cardinalidade 1:N (Um-Para-Vários): Indica a adição de um campo na tabela correspondente ao "lado N". Esse campo será uma chave estrangeira e vai armazenar a chave primária da tabela do "lado 1". Veja um esquema relacional que exemplifica esse tipo de cardinalidade:

Funcionario (CodFuncionario, NomeFunc)

Ramal (CodRamal, CodFuncionario, NumeroRamal)

CodFuncionario referencia Funcionario

Observe: Segundo a notação de Esquema Relacional, campo sublinhado indica Chave Primária. No esquema relacional da tabela Ramal temos dois campos sublinhados (indicando que os dois forma a chave primática). O campo CodFuncionario no entanto é uma chave estrangeira, porque ele armazena um valor que identifica um registro na tabela Funcionario.

Cardinalidade N:N (Vários-Para-Vários): Este tipo de relacionamento gera uma terceira tabela. Neste caso nenhum dos lados do relacionamento vai armazenar chave estrangeira (como no 1:N). O que vai acontecer é que cria-se uma nova tabela, que vai armazenar as chaves de ambos os lados. Veja um esquema relacional que exemplifica esse tipo de cardinalidade:

Funcionario(CodFuncionario, NomeFunc)

Projeto(CodProjeto, TituloProjeto)

Participacao(CodFuncionario,CodProjeto)

CodFuncionario referencia Funcionario

CodProjeto referencia Projeto

Observe: Segundo este esquema relacional, um funcionário pode participar de vários projetos, sendo que em um projeto pode-se ter vários funcionários. Neste caso cria-se mais uma tabela, que vai juntar as duas chaves do relacionamento. No exemplo, cria-se a tabela "Participacao" que indica o funcionário e qual projeto ele participa.

Cardinalidade 1:1 (Um-Para-Um): Esse tipo de cardinalidade em grande maioria dos casos não justifica um relacionamento. Mas pode ser implementado por questões de organização dos dados. Veremos futuramente na parte de Normalização, como resolver relacionamentos 1:1..

No próximo artigo, continuamos a fazer a tradução de relacionamentos . Até breve.

Parte 06 (Generalizações / Especializações)

Caros colegas. Estive ausente da coluna nesses 3 meses por motivos profissionais, mas aqui estou, e darei continuidade à série sobre Modelagem.

No artigo anterior vimos como traduzir os relacionamentos. Para concluir essa etapa, vamos às recomendações:

- Tradução de relacionamentos é orientada sempre pela cardinalidade mínima.
- Se houverem relacionamentos 1:1, verifique se não é melhor fundir as 2 tabelas em uma única.
- A chave primária é colocada sempre na tabela que corresponde ao lado N do relacionamento 1:N.
- Relacionamentos N:N sempre geram uma terceira tabela, com as chaves primárias das 2 tabelas originais.
- Se necessário, escreva o esquema relacional do seu banco de dados, antes de gerar suas tabelas.

A tradução de generalizações/especializações tem alguns aspectos particulares que devemos analisar.

Primeiramente vamos supor uma entidade com especializações:

Entidade: Pessoa - Atributos: Código, Nome, Endereço e Telefone

Entidade: Pessoa Física - Atributos: Todos os atributos de Pessoa, CPF, RG

Entidade: Pessoa Jurídica - Atributos: Todos os atributos de Pessoa, CNPJ, IE, Razão Social

As entidades Pessoa Física e Jurídica são especializações da entidade Pessoa. Como representar isso no banco de dados?

Bem, nesse caso temos 2 alternativas:

1) Criar uma única tabela para todas as especializações e incluir um campo diferenciador: Seria juntar todos os tipos de Pessoa, em uma única tabela e acrescentar mais um campo para identificar a Pessoa. Exemplo:

Pessoa: Código, TipoDePessoa, Nome, Endereço, Telefone, CPF, RG, CNPJ, IE, RazaoSocial

2) Criar uma tabela para cada especialização e definir mais um campo identificador

Pessoa: Código, Nome, Endereço, Telefone

Pessoa_Fisica: CodPessoa, CPF, RG

Pessoa_Juridica: CodPessoa, CNPJ, IE, RazaoSocial

A vantagem da primeira alternativa é que não precisaremos fazer junções da tabela generalizada (Pessoa) com a tabela especializada (Pessoa Física ou Jurídica) quando precisarmos de informações específicas. Outra vantagem é que a chave primária da tabela Pessoa fica armazenada somente 1 vez no banco de dados. A desvantagem é que, ao fazermos uma consulta no banco de dados, a linha inteira (todos os campos) são carregados na memória, mas sabemos que haverá campos em branco, dependendo do tipo de Pessoa cadastrada.

Na segunda alternativa, há a necessidade de fazer junções quando formos obter todas as informações de uma Pessoa. Porém, a vantagem é que teremos somente os dados necessários sem a necessidade de carregar todos os campos na memória, gerando mais acessos ao banco de dados. As chaves primárias de Pessoa são repetidas nas tabelas especializadas e, quando houver atualização das informações de uma pessoa, haverá a necessidade de criar uma instrução para cada tabela especializada.

A escolha de um dos tipos de tradução para generalizações/especializações irá depender do projeto que está sendo construído e dos recursos disponíveis para quem está modelando. Nada impede que as duas alternativas sejam usadas no mesmo projeto de banco de dados, uma alternativa pra cada caso de tabelas generalizadas.

Concluídas as etapas de tradução entre modelos, temos o banco de dados formado, com suas tabelas, campos e relacionamentos.

No próximo artigo falarei sobre como refinar o seu modelo relacional e também sobre a Normalização do seu banco de dados.

Observações:

Nesta série de artigos, eu faço uma análise de alto nível sobre a Modelagem de Dados sem aprofundar-me nos processos de modelagem. Se for do seu interesse, vale a pena pesquisar o conteúdo específico sobre cada etapa que descrevi nessa série.

Se você me enviou e-mails e não obteve resposta, peço a gentileza de reenviar. Faço questão de responder a todas as mensagens, apesar do pouco tempo hábil.

Final (Normalização)

Caros leitores. Estou de volta com a última parte desta série, onde falarei sobre Normalização.

Normalização é um processo baseado nas chamadas formas normais. Uma forma normal é uma regra e deve ser aplicada na construção das tabelas do banco de dados para que estas fiquem bem projetadas. Segundo autores, existem 4 formas normais. Neste artigo vou falar sobre as 3 primeiras, sendo as principais.

Com o banco de dados construído, devem-se aplicar as 3 formas normais em cada tabela, ou grupo de tabelas relacionadas. As formas têm uma ordem e são dependentes, isto é, para se aplicar a segunda norma, deve-se obrigatoriamente ter aplicado a primeira e assim por diante.

Então, vamos às normas:

1 Forma Normal: Verificação de Tabelas Aninhadas.

Para uma tabela estar na primeira forma normal ela não deve conter tabelas aninhadas. Um jeito fácil de verificar esta norma é fazer uma leitura dos campos das tabelas fazendo a pergunta: Este campo depende de qual?.

Vamos exemplificar, com a tabela Venda. Este é o esquema relacional da tabela:

Venda(Codvenda, Cliente, Endereco, Cep, Cidade, Estado, Telefone, Produto, Quantidade, Valorunitario, Valorfinal).

O raciocínio é o seguinte: A tabela Venda, deve armazenar informações da venda. Pois bem, verificando o campo Cliente, sabemos que ele depende de CodVenda, afinal para cada Venda há um cliente. Vendo o campo Endereço, podemos concluir que ele não depende de Codvenda, e sim de Cliente, pois é uma informação referente particularmente ao cliente. Não existe um endereço de venda, existe sim um endereço do cliente para qual se fez a venda. Nisso podemos ver uma tabela aninhada. Os campos entre colchetes, são referentes ao cliente e não á venda.

Venda (Codvenda, [Cliente, Endereço, Cep, Cidade, Estado, Telefone, Produto, Quantidade, Valorunitario, Valorfinal]).

A solução é extrair estes campos para uma nova tabela, adicionar uma chave-primária à nova tabela e relaciona-la com a tabela Venda criando uma chave-estrangeira.

Ficaria desta forma:

Cliente (Codcliente, Nome, Endereço, Cep, Cidade, Estado, Telefone).

Venda (Codvenda, Codcliente, Produto, Quantidade, Valorunitario, Valorfinal).

Agora aplicamos novamente á primeira forma normal as 2 tabelas geradas. Uma situação comum em tabelas de cadastro é o caso Cidade-Estado. Analisando friamente pela forma normal, o Estado na tabela Cliente, depende de Cidade. No entanto Cidade, também depende de Estado, pois no caso de a cidade ser Curitiba o estado sempre deverá ser Paraná, porém se o Estado for Paraná, a cidade também poderá ser Londrina. Isso é o que chamamos de Dependência funcional: é onde aparentemente, uma informação depende da outra. No caso Cidade-Estado a solução é simples:

Extraímos Cidade e Estado, de Cliente e geramos uma nova tabela. Em seguida, o mesmo processo feito anteriormente: adicionar uma chave-primária à nova tabela e relaciona-la criando uma chave-estrangeira na antiga tabela.

Cidade (Codcidade, Nome, Estado).

Cliente (Codcliente, Codcidade, Nome, Endereço, Cep, Telefone).

Venda (Codvenda, Codcliente, Codcidade, Produto, Quantidade, Valorunitario, Valorfinal).

Seguindo com o exemplo, a tabela Cliente encontra-se na 1 forma normal, pois não há mais tabelas aninhadas. Verificando Venda, podemos enxergar mais uma tabela aninhada. Os campos entre colchetes são referente á mesma coisa: Produto de Venda

Venda (Codvenda, Codcliente, Codcidade, [Produto Quantidade, [Valorunitario, Valorfinal]).

Na maioria das situações, produtos têm um valor previamente especificado. O Valorunitário depende de Produto. Já a Quantidade (campo entre Produto e Valorunitario) não depende do produto e sim da Venda.

Cidade (Codcidade, Nome, Estado).

Cliente (Codcliente, Codcidade, Nome, Endereço, Cep, Telefone).

Produto (Codproduto, Nome, Valorunitario).

Venda (Codvenda, Codcliente, Codcidade, Codproduto, Quantidade, Valorfinal).

Passando a tabela Venda pela primeira forma normal, obtivemos 3 tabelas. Vamos á próxima forma

2 Forma Normal: Verificação de Dependências Parciais

Para uma tabela estar na segunda forma normal, além de estar na primeira forma ela não deve conter dependências parciais. Um jeito de verificar esta norma é refazer a leitura dos campos fazendo a pergunta: Este campo depende de toda a chave? Se não, temos uma dependência parcial.

Vimos antes o caso Cidade-Estado que gerava uma dependência funcional. É preciso entender este conceito para que você entenda o que é Dependência Parcial.

Após a normalização da tabela Venda, acabamos com uma chave composta de 4 campos:

Venda (Codvenda, Codcliente, Codcidade, Codproduto, Quantidade, Valorfinal).

A questão agora é verificar se cada campo não-chave depende destas 4 chaves. O raciocínio seria assim:

1. O primeiro campo não-chave é Quantidade.
2. Quantidade depende de Codvenda, pois para cada venda há uma quantidade específica de itens.
3. Quantidade depende de Codvenda e Codcliente, pois para um cliente podem ser feitas várias vendas, com quantidades diferentes.
4. Quantidade não depende de Cidade. Quem depende de Cidade é Cliente. Aqui está uma dependência parcial.
5. Quantidade depende de Codproduto, pois para cada produto da Venda á uma quantidade certa.

Quantidade depende de 3 campos, dos 4 que compõe a chave de Venda. Quem sobrou nessa história foi Codcidade. A tabela Cidade já está ligada com Cliente, que já está ligado com Venda. A chave Codcidade em Venda é redundante, portanto podemos eliminá-la.

Venda (Codvenda, Codcliente, Codproduto, Quantidade, Valorfinal).

O próximo campo não-chave é Valorfinal. Verificando Valorfinal, da mesma forma que Quantidade, ele depende de toda a chave de Venda. Portanto vamos á próxima norma.

3 Forma Normal: Verificação de Dependências Transitivas

Para uma tabela estar na segunda forma normal, além de estar na segunda forma ela não deve conter dependências transitivas. Um jeito de verificar esta norma é refazer a leitura dos campos fazendo a pergunta: Este campo depende de outro que não seja a chave? Se Sim, temos uma dependência transitiva..

No exemplo de Venda, temos um caso de dependência transitiva:

Na tabela Venda, temos Valorfinal. Este campo é o resultado do valor unitário do produto multiplicado pela quantidade, isto é, para um valor final existir ele DEPENDE de valor unitário e quantidade. O Valorunitário está na tabela Produto, relacionada à Venda e Quantidade está na

própria Venda. Valorfinal depende destes 2 campos e eles não são campos-chave, o que nos leva a pensar: Se temos valor unitário e quantidade, porque teremos valor final? O valor final nada mais é que o resultado de um cálculo de dados que já está estão no banco, o que o torna um campo redundante.

Quando for necessário ao sistema obter o valor final, basta selecionar o valor unitário e multiplicar pela quantidade. Não há porque guardar o valor final em outro campo. Aqui a solução é eliminar o campo Valorfinal.

Cidade (Codcidade, Nome, Estado).

Cliente (Codcliente, Codcidade, Nome, Endereco, Cep, Telefone).

Produto (Codproduto, Nome, Valorunitario).

Venda (Codvenda, Codcliente, Codproduto, Quantidade).

Em tese, agora temos todas as tabelas normalizadas. Ainda restou o caso do campo Estado na tabela Cidade, mas eu deixarei para uma outra ocasião, pois o objetivo aqui é mostrar conceitualmente o processo de normalização do banco de dados.

É muito comum, no processo de normalização enxergamos todas as formas normais ao mesmo tempo. Enquanto separamos as tabelas aninhadas, já conseguimos ver as dependências transitivas e logo mais encontramos uma dependência parcial, tudo assim, ao mesmo tempo. Isso é normal. Só tome cuidado, para não deixar nada passar batido.

Reforçando o recado do artigo anterior: tem muito mais a ser visto sobre as etapas que eu mostrei nessa série. Aqui foi só uma demonstração do que se deve levar em conta ao modelar e construir um banco de dados íntegro, correto e que aproveita os dados da forma mais eficiente possível.

Links dos originais:

http://imasters.uol.com.br/artigo/4468/uml/modelagem_de_dados_no_access/

http://imasters.uol.com.br/artigo/4629/bancodedados/modelagem_de_dados_1_entidades/

http://imasters.uol.com.br/artigo/4799/access/modelagem_de_dados_2_-_os_relacionamentos/

http://imasters.uol.com.br/artigo/5111/bancodedados/modelagem_de_dados_-_validacao_do_modelo_er/

http://imasters.uol.com.br/artigo/5265/bancodedados/modelagem_de_dados_parte_04_-_abordagem_relacional/

http://imasters.uol.com.br/artigo/5403/bancodedados/modelagem_de_dados_-_parte_05_transformacao_entre_modelos/

http://imasters.uol.com.br/artigo/6167/bancodedados/modelagem_de_dados_-_parte_06_generalizacoes_especializacoes/

http://imasters.uol.com.br/artigo/7020/bancodedados/modelagem_de_dados_-_final_normalizacao/

Modelagem de dados e administração

Não diagramação e mapeamento

SQL não é relacional e contém muitos limites arbitrárias

...a diferença entre um mau e um bom programador é se considera o código ou as

estruturas de dados mais importantes. Maus programadores preocupam-se com código. Bons programadores preocupam-se com estruturas de dados e seus relacionamentos. Torvalds, Linux.

Mostra-me teus fluxogramas e esconda-me tuas tabelas, e continuarei no escuro. Mostra-me tuas tabelas, e... não precisarei de teus fluxogramas: serão óbvios.
-- Brooks, Frederick Phillips, Jr.: The Mythical Man-Month.

Dicionários de dados são essenciais.
qDiagramas podem, e devem, ser gerados automaticamente.
qFerramentas de modelagem podem unificar modelos de diversas bases.
qGlossários são úteis.
qDicionários de tipos de dados são essenciais.
qDicionário geral de dados dão mais trabalho.
qDiagramas te promovem: AutoDoc!

O Modelo Relacional de Dados - Parte 01 - Júlio Battisti

Introdução

Objetivo: Em uma série de quatro artigos, apresentarei alguns conceitos básicos sobre Bancos de Dados, mais especificamente sobre o Modelo Relacional de Dados.

Para a melhor utilização, ou seja, para uma utilização eficiente de bancos de dados como o Microsoft Access, SQL Server, ORACLE, DB2 ou qualquer outro banco de dados relacional, é importante o conhecimento e correto entendimento dos conceitos apresentados nesta série de artigos. Vou abordar os seguintes Conceitos:

- . Entidades e atributos
- . Chave primária
- . Relacionamentos entre entidades (tabelas)
- . Integridade Referencial
- . Normalização de tabelas
- . Um Problema Proposto
- . Arquitetura do Microsoft Access.

Nota: Os exemplos apresentados utilizarão telas do Microsoft Access. Porém os princípios básicos do modelo relacional aplicam-se a qualquer banco de dados baseado no modelo relacional de dados. Estes bancos de dados são algumas vezes denominados: SGBDR - Sistemas Gerenciadores de Banco de Dados Relacionais.

Entidades e Atributos:

Toda a Informação de um banco de dados relacional é armazenada em Tabelas, que na linguagem do modelo relaciona, também são chamadas de Entidades. Por exemplo, posso ter uma Tabela "Clientes", onde seriam armazenadas informações sobre os diversos clientes.

Sobre cada um dos clientes podem ser armazenadas diversas informações tais como:

- .Nome
- .RG
- .CPF
- .Rua
- .Bairro
- .Telefone
- .CEP
- .Data de Nascimento

Essas diversas características de cada Cliente são os "**Atributos**" da entidade Cliente, também chamados de campos da tabela Cliente.

"O Conjunto de todos os Atributos de um cliente e os valores dos atributos é o que forma o Registro do Cliente".

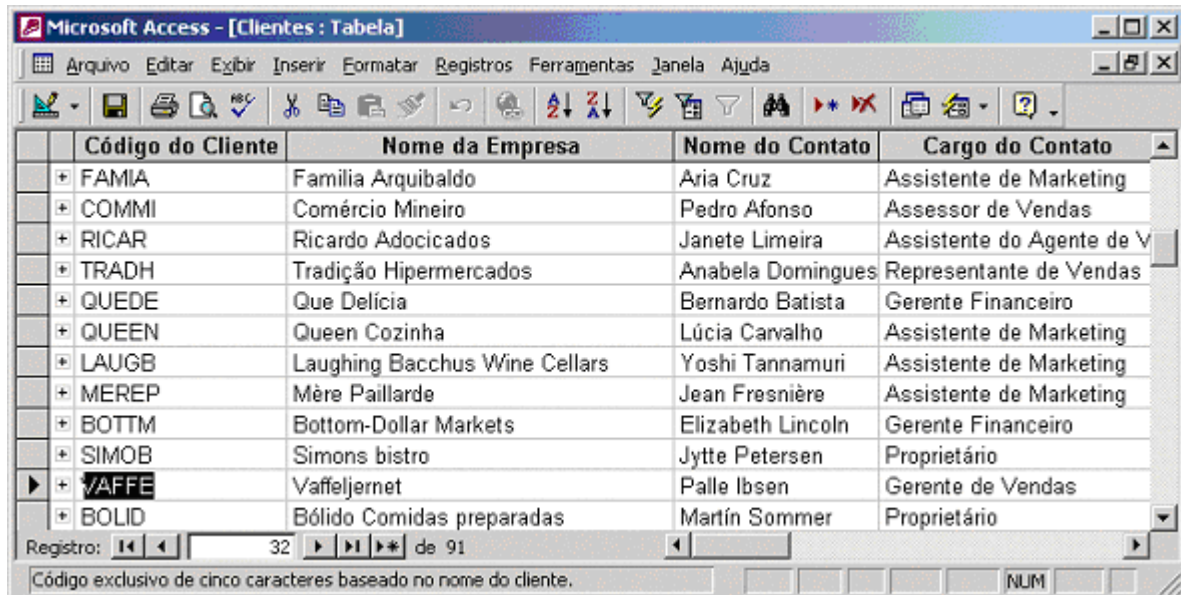
Com isso temos uma Tabela que é constituída por um conjunto de Registros (uma linha completa com informações sobre o cliente) e cada Registro formado por um conjunto de atributos (Nome, Endereço, etc).

Resumindo:

Entidade ou Tabela: Um conjunto de Registros.

Campos ou Atributos: Características Individuais da tabela.

Considere o Exemplo da figura abaixo, mostro uma tabela com cadastro de Clientes com os seus diversos Campos (atributos):



| | Código do Cliente | Nome da Empresa | Nome do Contato | Cargo do Contato |
|---|-------------------|-------------------------------|-------------------|---------------------------|
| + | FAMIA | Família Arquibaldo | Aria Cruz | Assistente de Marketing |
| + | COMMI | Comércio Mineiro | Pedro Afonso | Assessor de Vendas |
| + | RICAR | Ricardo Adocicados | Janete Limeira | Assistente do Agente de V |
| + | TRADH | Tradição Hipermercados | Anabela Domingues | Representante de Vendas |
| + | QUEDE | Que Delícia | Bernardo Batista | Gerente Financeiro |
| + | QUEEN | Queen Cozinha | Lúcia Carvalho | Assistente de Marketing |
| + | LAUGB | Laughing Bacchus Wine Cellars | Yoshi Tannamuri | Assistente de Marketing |
| + | MEREP | Mère Paillarde | Jean Fresnière | Assistente de Marketing |
| + | BOTTM | Bottom-Dollar Markets | Elizabeth Lincoln | Gerente Financeiro |
| + | SIMOB | Simons bistro | Jytte Petersen | Proprietário |
| ▶ | VAFPE | Vaffeljernet | Palle Ibsen | Gerente de Vendas |
| + | BOLID | Bóldo Comidas preparadas | Martín Sommer | Proprietário |

Registro: 32 de 91

Código exclusivo de cinco caracteres baseado no nome do cliente.

Figura 1: Tabela Cliente e seus Campos - CódigoDoCliente, NomeDaEmpresa e assim por diante

No exemplo da figura anterior temos entidade: "Clientes" e seus diversos atributos: "Código do Cliente", "Nome da Empresa", "Nome do Contato", "Cargo do Contato", "Endereço", etc. Em cada linha temos um conjunto de atributos e seus valores. Cada linha forma um Registro. Cada Coluna é um atributo da Tabela Clientes.

Um dos grandes desafios em se projetar um Banco de Dados com sucesso é a correta Determinação das Entidades que existirão no Banco de Dados, bem como dos Atributos de Cada Entidade.

Chave Primária

Objetivo: Neste item falarei sobre o conceito de Chave Primária e a sua importância no Modelo Relacional de dados.

Chave Primária

O Conceito de "**Chave Primária**" é fundamental para o correto entendimento de como funciona um Banco de Dados baseado no modelo relacional. Vamos entender o que significa um campo ser a Chave Primária de uma Tabela e como tornar um Campo a Chave Primária de uma Tabela.

"Ao Definirmos um Campo como sendo uma Chave Primária, estamos informando ao Microsoft Access que não podem existir dois registros com o mesmo valor no campo que é a Chave Primária, ou seja, os valores no campo Chave Primária precisam ser únicos".

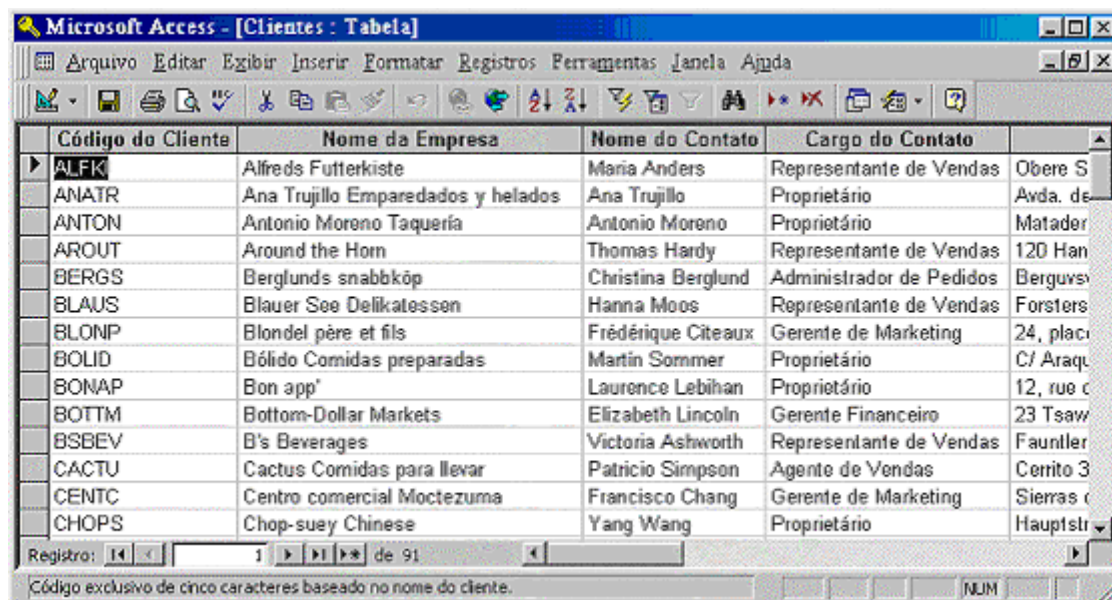
Por exemplo, se defino um campo "Número da Identidade", da tabela Clientes, como sendo um campo do tipo Chave Primária, estou dizendo que não podem ser cadastrados dois clientes com o mesmo valor no campo "Número da Identidade". Na prática estou garantindo que não possam ser cadastrados dois clientes com o mesmo Número de Identidade".

Em outras palavras poderíamos dizer que o Campo Chave Primária identifica de Maneira Única cada Registro de uma Tabela, isto é, de posse do valor da Chave Primária somente localizaremos um registro com aquele valor no campo Chave Primária. Outros exemplos de campos que podem ser definidos como chaves primária:

- . Campo CPF em uma tabela de cadastro de clientes.
- . Campo CNPJ em uma tabela de cadastro de fornecedores.
- . Matrícula do aluno em uma tabela de cadastro de alunos.
- . Código da Peça em uma tabela de cadastro de peças.
- . Matrícula do funcionário em uma tabela de cadastro de funcionários.
- . Número do pedido em uma tabela de cadastro de pedidos

Este é um conceito muito importante, pois conforme veremos mais adiante os conceitos de Integridade Referencial e Normalização estão diretamente ligados ao conceito de Chave Primária.

Na próxima figura apresento um exemplo da tabela Cliente onde o Campo "Código do Cliente" é definido como uma Chave Primária. Observe que não existem dois clientes com o Mesmo Código.



The screenshot shows the Microsoft Access interface with a table named 'Clientes'. The table has five columns: 'Código do Cliente', 'Nome da Empresa', 'Nome do Contato', 'Cargo do Contato', and a fifth column with a dropdown menu. The 'Código do Cliente' column is highlighted with a key icon, indicating it is the primary key. The table contains 14 records, each with a unique code.

| Código do Cliente | Nome da Empresa | Nome do Contato | Cargo do Contato | |
|-------------------|------------------------------------|--------------------|--------------------------|-----------|
| ALFK | Alfreds Futterkiste | Maria Anders | Representante de Vendas | Obere S |
| ANATR | Ana Trujillo Emparedados y helados | Ana Trujillo | Proprietário | Avda. de |
| ANTON | Antonio Moreno Taquería | Antonio Moreno | Proprietário | Matader |
| AROUT | Around the Horn | Thomas Hardy | Representante de Vendas | 120 Han |
| BERGS | Berglunds snabbköp | Christina Berglund | Administrador de Pedidos | Berguvs |
| BLAUS | Blauer See Delikatessen | Hanna Moos | Representante de Vendas | Forsters |
| BLONP | Blondel père et fils | Frédérique Citeaux | Gerente de Marketing | 24, plac |
| BOLID | Bólido Comidas preparadas | Martin Sommer | Proprietário | C/ Araqu |
| BONAP | Bon app' | Laurence Leblan | Proprietário | 12, rue c |
| BOTTM | Bottom-Dollar Markets | Elizabeth Lincoln | Gerente Financeiro | 23 Tsaw |
| BSBEV | B's Beverages | Victoria Ashworth | Representante de Vendas | Fauntler |
| CACTU | Cactus Comidas para llevar | Patricio Simpson | Agente de Vendas | Cenito 3 |
| CENTC | Centro comercial Moctezuma | Francisco Chang | Gerente de Marketing | Sierras c |
| CHOPS | Chop-suey Chinese | Yang Wang | Proprietário | Hauptstr |

At the bottom of the table, there is a status bar that reads: 'Código exclusivo de cinco caracteres baseado no nome do cliente.'

Figura 2: Campo "Código do Cliente" definido como Chave Primária.

Após ter definido um campo como sendo a Chave Primária da tabela, o próprio banco de dados (quer seja Access, SQL Server, ORACLE ou qualquer outro), garante que não sejam inseridos dados duplicados no campo que é a chave primária.

Por exemplo, se o usuário tentar cadastrar um pedido com o mesmo número de um pedido já existente, o registro não será cadastrado e uma mensagem de erro será emitida.

Um último detalhe importante para lembrarmos é que a Chave Primária pode ser formada pela combinação de Mais de Um Campo. Podem existir casos em que um único campo não é capaz de atuar como chave primária, pelo fato deste apresentar valores repetidos. Nestes casos podemos definir uma combinação de 2 ou mais campos para ser a nossa chave primária.

Além disso, uma tabela somente pode ter uma Chave Primária, seja ela simples ou composta. Ou seja, não pode definir dois ou mais campos de uma tabela para serem cada um, uma chave primária separada. Não confundir com o caso de uma chave primária composta, onde a união de dois ou mais campos é que forma a única chave primária da tabela. Ou seja, cada tabela pode ter uma única

chave primária.

Conclusão

Nesta primeiro artigo da série, você aprendeu sobre os conceitos de entidades (tabelas), atributos (campos) e registros. Estes são os elementos básicos do modelo relacional de dados. Em seguida falei sobre o conceito de Chave Primária. Falei sobre as características do campo chave primária e da possibilidade de existir uma chave primária composta por dois ou mais campos. No próximo artigo da série você aprenderá sobre "Relacionamentos entre Tabelas".

O Modelo Relacional de Dados – Parte 02

Introdução

Na primeira parte deste artigo falei sobre Entidades (tabelas), Atributos (campos) e sobre o conceito de Chave Primária. Nesta segunda parte vou abordar os seguintes tópicos:

- . Relacionamentos entre entidades (tabelas) - conceito
- . Relacionamentos entre entidades (tabelas) - tipos

Nota: Os exemplos apresentados utilizarão telas do Microsoft Access e o arquivo de exemplos Northwind.mdb, o qual é instalado juntamente com o Microsoft Access. Este arquivo está disponível, por padrão, no seguinte caminho:

<C:\Arquivos de programas\Microsoft Office\Office\Samples>

Porém os princípios básicos do modelo relacional aplicam-se a qualquer banco de dados baseado no modelo relacional de dados. Estes bancos de dados são algumas vezes denominados: SGBDR - Sistemas Gerenciadores de Banco de Dados Relacionais.

Relacionamentos entre Tabelas

Neste item vou apresentar o conceito de relacionamento entre tabelas, este um conceito fundamental para o Modelo Relacional de Dados. Também falarei sobre os diferentes tipos de relacionamentos existentes. Serão apresentados exemplos práticos.

Conforme descrito na Parte I, um banco de dados é composto por diversas tabelas, como por exemplo: Clientes, Produtos, Pedidos, Detalhes do Pedido, etc. Embora as informações estejam separadas em cada uma das Tabelas, na prática devem existir **relacionamentos** entre as tabelas. Por exemplo: Um Pedido é feito por um Cliente e neste Pedido podem existir diversos itens, itens que são gravados na tabela Detalhes do Pedido. Além disso cada Pedido possui um número único (Código do pedido), mas um mesmo Cliente pode fazer diversos pedidos e assim por diante.

Em um banco de dados, precisamos de alguma maneira para representar estes relacionamentos da vida Real, em termos das tabelas e de seus atributos. Isto é possível com a utilização de "Relacionamentos entre tabelas", os quais podem ser de três tipos:

- . Um para Um
- . Um para Vários
- . Vários para Vários

Relacionamento do Tipo Um para Um:

Esta relação existe quando os campos que se relacionam são ambos do tipo Chave Primária, em suas respectivas tabelas. Cada um dos campos não apresenta valores repetidos. Na prática existem

poucas situações onde utilizaremos um relacionamento deste tipo. Um exemplo poderia ser o seguinte: Imagine uma escola com um Cadastro de Alunos na tabela Alunos, destes apenas uma pequena parte participa da Banda da Escola. Por questões de projeto do Banco de Dados, podemos criar uma Segunda Tabela "Alunos da Banda", a qual se relaciona com a tabela Alunos através de um relacionamento do tipo Um para Um. Cada aluno somente é cadastrada uma vez na Tabela Alunos e uma única vez na tabela Alunos da Banda. Poderíamos utilizar o Campo Matrícula do Aluno como o Campo que relaciona as duas Tabelas.

Importante: O campo que relaciona duas tabelas deve fazer parte, ter sido definido, na estrutura das duas tabelas.

Na tabela Alunos da Banda poderíamos colocar apenas o Número da Matrícula do aluno, além das informações a respeito do Instrumento que ele toca, tempo de banda, etc. Quando fosse necessário buscar as informações tais como nome, endereço, etc, estas podem ser recuperadas através do relacionamento existente entre as duas tabelas, evitando, com isso, que a mesma informação (Nome, Endereço, etc) tenha que ser duplicada nas duas tabelas, inclusive aumentando a probabilidade de erros de digitação.

Na Figura a seguir vemos o exemplo de um Relacionamento do tipo Um para Um entre as tabelas Alunos e Alunos da Banda.

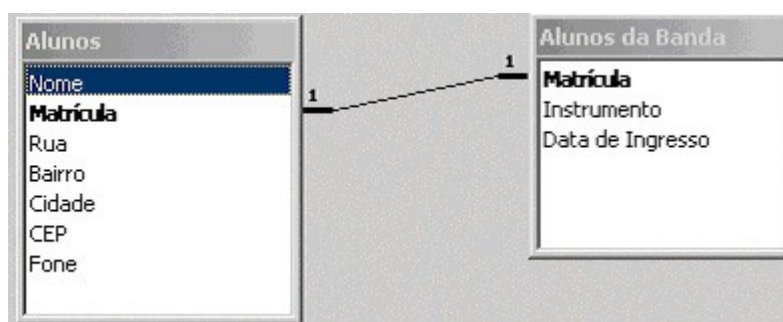


Figura 1: Relacionamento Um para Um entre as Tabelas Alunos e Alunos da Banda

Com a criação deste relacionamento estamos evitando a repetição desnecessária de informações em diferentes tabelas.

Relacionamento do Tipo Um para Vários:

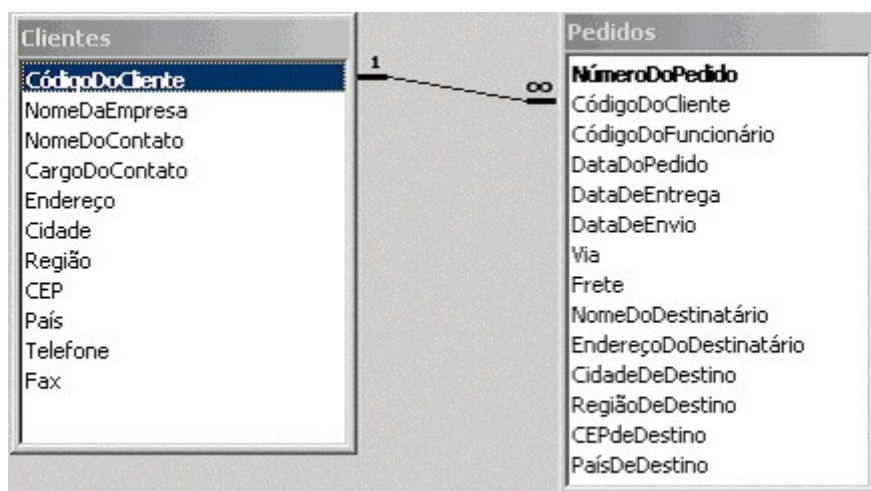
Este é, com certeza, o tipo de relacionamento mais comum entre duas tabelas. Uma das tabelas (o lado **um** do relacionamento) possui um campo que é a **Chave Primária** e a outra tabela (o lado **vários**) se relaciona através de um campo cujos valores relacionados **podem se repetir várias vezes**.

Considere o exemplo entre a tabela Clientes e Pedidos. Cada Cliente somente é cadastrado uma única vez na tabela de Clientes (por isso o campo Código do Cliente, na tabela Clientes, é uma chave primária, indicando que não podem ser cadastrados dois clientes com o mesmo código), portanto a tabela Clientes será o lado um do relacionamento. Ao mesmo tempo cada cliente pode fazer diversos pedidos, por isso que o mesmo Código de Cliente poderá aparecer várias vezes na tabela Pedidos: **tantas vezes quantos forem os pedidos que o Cliente tiver feito**. Por isso que temos um relacionamento do tipo Um para Vários entre a tabela Clientes e Pedidos, através do campo Código do Cliente, indicando que **um mesmo Cliente pode realizar diversos (vários) pedidos**.

Na próxima figura vemos um exemplo de um Relacionamento **Um para Vários** entre as Tabelas

Clientes e Pedidos do banco de dados Pedidos.mdb, através do campo código do cliente:

Figura 2:



Relacionamento Um para Vários entre as Tabelas Clientes e Pedidos

Observe que o lado Vários do relacionamento é representado pelo símbolo do infinito (∞). Esta é a representação utilizada no Microsoft Access. Diferentes representações poderão ser utilizadas por outros bancos de dados.

No lado Um do relacionamento o campo é definido como uma Chave Primária (Campo CódigoDoCliente na tabela Clientes) e no lado Vários não (campo CódigoDoCliente na tabela Pedidos), indicando que no lado vários o Código do Cliente pode se repetir várias vezes, o que faz sentido, uma vez que um mesmo cliente pode fazer diversos pedidos.

Importante: Observe que o campo que é o lado vários do relacionamento não pode ser definido como chave primária. Lembrando do conceito de Chave Primária, apresentado na Parte I: Chave Primária é o campo no qual não podem haver valores repetidos. Ora, se o campo está no lado "vários", significa que ele poderá ter o seu valor repetido em vários registros. Por exemplo, na tabela pedidos, poderá haver vários registros para o mesmo cliente. Se o campo terá que ter valores repetidos, então ele não pode ser definido como chave primária.

No Banco de Dados NorthWind.mdb, Northwind.mdb, o qual é instalado juntamente com o Microsoft Access. Este arquivo está disponível, por padrão, no seguinte caminho:

C:\Arquivos de programas\Microsoft Office\Office\Samples

Existem diversos outros exemplos de relacionamentos do tipo Um para Vários, conforme descrito na Próxima Tabela:

| Tipo | Lado Um | Lado Vários |
|----------------|--|--|
| Um para Vários | CódigoDoFornecedor na tabela Fornecedores | CódigoDoFornecedor na tabela Produtos |

| | | |
|----------------|---|---|
| Um para Vários | CódigoDaCategoria na tabela Categorias | CódigoDaCategoria na tabela Produtos |
| Um para Vários | CódigoDoProduto na tabela Produtos | CódigoDoProduto na tabela Detalhes do Pedido |
| Um para Vários | CódigoDoFuncionário na tabela Funcionários | CódigoDoFuncionário na tabela Pedidos |
| Um para Vários | NúmeroDoPedido na tabela Pedidos | NúmeroDoPedido na tabela Detalhes do Pedido |
| Um para Vários | CódigoDaTransportadora na tabela Transportadoras | Via na tabela Pedidos |
| Um para Vários | CódigoDoCliente na tabela Clientes | CódigoDoCliente na tabela Pedidos |

Em um dos próximos artigos mostrarei como implementar, na prática, estes relacionamentos. Algumas observações importantes sobre relacionamentos:

- O Nome dos Campos envolvidos no Relacionamento, não precisa ser, necessariamente, o mesmo, conforme indicado pelo relacionamento entre os campos CódigoDaTransportadora e Via, na tabela anterior. O tipo dos campos é que precisa ser o mesmo, por exemplo, se um dos campos for do tipo Texto, o outro também deverá ser do tipo Texto.
- Sempre o Lado um do Relacionamento deve ser uma chave primária, já o lado vários não pode ser uma chave Primária.
- De Preferência, antes de Criar os Relacionamentos verifique se o tipo dos campos a serem relacionados é o mesmo, além de características como máscaras de entrada e formato.

Relacionamento do tipo Vários para Vários:

Este tipo de relacionamento "**aconteceria**" em uma situação onde em ambos os lados do relacionamento os valores poderiam se repetir. Vamos considerar o caso entre Produtos e Pedidos. Posso ter **Vários Pedidos** nos quais aparece um determinado produto, além disso **vários Produtos** podem aparecer no mesmo Pedido. Esta é uma situação em que temos um Relacionamento do Tipo Vários para Vários.

Na prática não é possível implementar um relacionamento deste tipo, devido a uma série de problemas que seriam introduzidos no modelo do banco de dados. Por exemplo, na tabela Pedidos

teríamos que repetir o Número do Pedido, Nome do Cliente, Nome do Funcionário, Data do Pedido, etc para cada item do Pedido.

Para evitar este tipo de problema é bastante comum "**quebrarmos**" um relacionamento do tipo Vários para Vários em dois relacionamentos do tipo Um para Vários. Isso é feito através da criação de uma nova tabela, a qual fica com o lado Vários dos relacionamentos. No nosso exemplo vamos criar a tabela Detalhes do Pedido, onde ficam armazenadas as informações sobre os diversos itens de cada pedido, aí ao invés de termos um relacionamento do tipo Vários para Vários, teremos dois relacionamentos do tipo um para vários, conforme descrito pela próxima tabela:

| Tipo | Lado Um | Lado Vários |
|----------------|---|---|
| Um para Vários | CódigoDoProduto na tabela Produtos | CódigoDoProduto na tabela Detalhes do Pedido |
| Um para Vários | NúmeroDoPedido na tabela Pedidos | NúmeroDoPedido na tabela Detalhes do Pedido |

Na figura abaixo temos a representação dos dois relacionamentos Um para Vários, resultantes da quebra do relacionamento vários-para-vários:

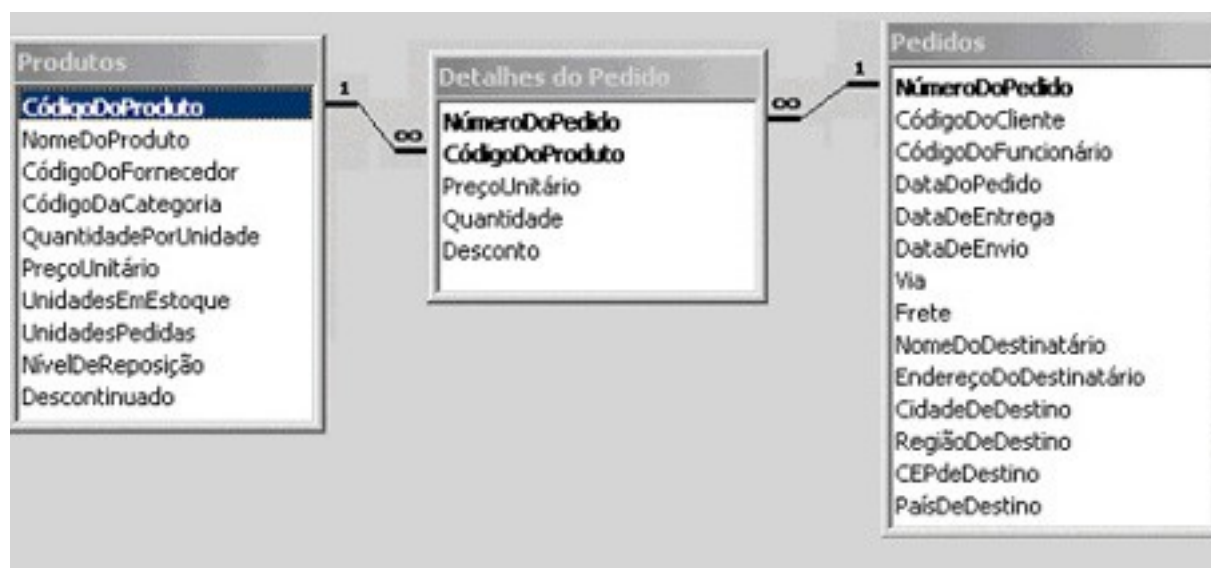


Tabela Detalhes do Pedido ficou com o lado Vários dos Relacionamentos

Esta situação em que um relacionamento um para Vários é "quebrado" em dois Relacionamentos do tipo Um para Vários é bastante comum. Diversas vezes utilizamos esta técnica para eliminar uma série de problemas no Banco de Dados, tais como informação repetida e inconsistência de Dados.

Agora que já conhecemos os Tipos de Relacionamentos existentes, na próxima Parte III, falarei sobre o conceito de Integridade Referencial como uma maneira de Garantir a Consistência dos

Dados.

Conclusão:

Nesta segundo artigo da série, você aprendeu sobre os conceitos de relacionamentos, sem dúvidas um dos conceitos mais importantes do Modelo Relacional. Implementar um banco de dados sem se preocupar com um projeto cuidados dos relacionamentos, é garantia certa de "encrenca" mais adiantes. São consultas que retornam dados inesperados, são relatórios que retornam valores "absurdos" e por aí vai.

É fundamental que os profissionais de desenvolvimento e de administração de banco de dados entendem o quão imputante é planejar o banco de dados, cuidadosamente, definindo quais tabelas farão parte do banco de dados, quais os campos de cada tabela, quais campos serão chave primária e qual o relacionamento entre as tabelas. Muitos acham que esta é uma "perda de tempo", que o bom mesmo é sentar e começar a implementar o banco de dados, depois "a gente vê o que dá". Ledo engano. Não é perda de tempo, muito pelo contrário, é um ganho de tempo e principalmente de qualidade no produto final. Quanto tempo é perdido depois, tentando corrigir erros e inconsistências que muitas vezes são decorrentes de um banco de dados mal projetado? Uma boa leitura a todos e até a Parte III.

O Modelo Relacional de Dados - Parte 03

Objetivo: Na primeira parte deste artigo falei sobre Entidades (tabelas), Atributos (campos) e sobre o conceito de Chave Primária. Na segunda parte falei sobre Relacionamentos e tipos de relacionamentos. Nesta terceira parte vamos aprender sobre um dos conceitos mais importantes do modelo relacional de dados: Integridade Referencial. Também mostrarei um exemplo prático de como configurar Relacionamentos e Integridade Referencial no Microsoft Access.

Nota: Os exemplos apresentados utilizarão telas do Microsoft Access e o arquivo de exemplos Northwind.mdb, o qual é instalado juntamente com o Microsoft Access. Este arquivo está disponível, por padrão, no seguinte caminho:

<C:\Arquivos de programas\Microsoft Office\Office\Samples>

Porém os princípios básicos do modelo relacional aplicam-se a qualquer banco de dados baseado no modelo relacional de dados. Estes bancos de dados são algumas vezes denominados: SGBDR - Sistemas Gerenciadores de Banco de Dados Relacionais.

Integridade Referencial

A Integridade Referencial é utilizada para garantir a Integridade dos dados entre as tabelas relacionadas. Por exemplo, considere um relacionamento do tipo Um-para-Vários entre a tabela Clientes e a tabela Pedidos (um cliente pode fazer vários pedidos). Com a Integridade Referencial, o banco de dados não permite que seja cadastrado um pedido para um cliente que ainda não foi cadastrado. Em outras palavras, ao cadastrar um pedido, o banco de dados verifica se o código do cliente que foi digitado já existe na tabela Clientes. Se não existir, o cadastro do pedido não será aceito. Com o uso da Integridade Referencial é possível ter as seguintes garantias (ainda usando o exemplo entre as tabelas Clientes e Pedidos):

- Quando o Código de um cliente for alterado na Tabela Clientes, podemos configurar para o banco de dados atualizar, automaticamente, todos os Códigos do Cliente na Tabela Pedidos, de tal maneira que não fiquem Registros Órfãos, isto é, registros de Pedidos com um Código de Cliente para o

qual não existe mais um correspondente na Tabela Clientes. Essa ação é conhecida como **"Propagar atualização dos campos relacionados"**.

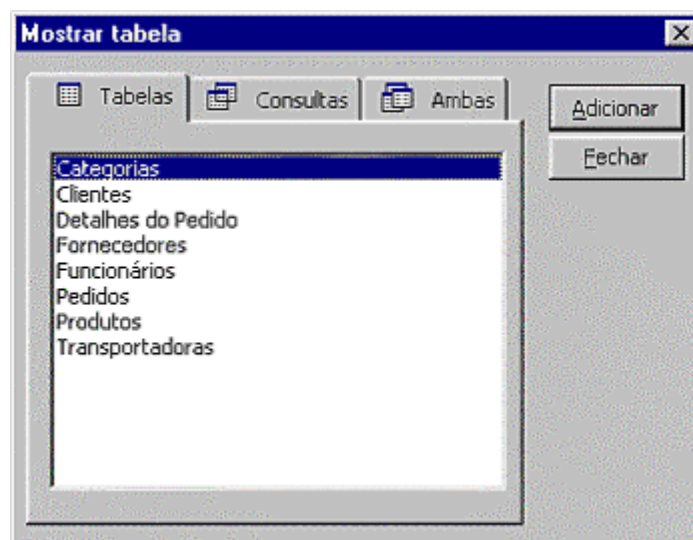
- Quando um Cliente for excluído da Tabela Clientes, podemos configurar para que o banco de dados exclua, automaticamente, na tabela Pedidos, todos os Pedidos para o Cliente que está sendo Excluído. Essa opção é conhecida como **"Propagar exclusão dos registros relacionados"**.

Essas opções, conforme mostrarei logo em seguida, podem ser configuradas quando da Definição dos Relacionamentos (no exemplo prático mais adiante utilizarei o Microsoft Access, mas estes conceitos são válidos para qualquer banco de dados). Estas opções não são obrigatórias, isto é, podemos optar por não Atualizar ou não Excluir em cascata. A Opção de **"Propagar atualização dos campos relacionados"** é utilizada na maioria das situações, já a opção de **"Propagar exclusão dos registros relacionados"** deve ser estudada caso a caso. Por exemplo, se nos quiséssemos manter um histórico com os Pedidos de cada Cliente, não utilizaríamos a opção "Propagar exclusão dos registros relacionados"; caso não nos interessasse manter um histórico dos pedidos, poderíamos utilizar esta opção.

Exemplo prático: Como Criar e Configurar Relacionamentos no Microsoft Access:

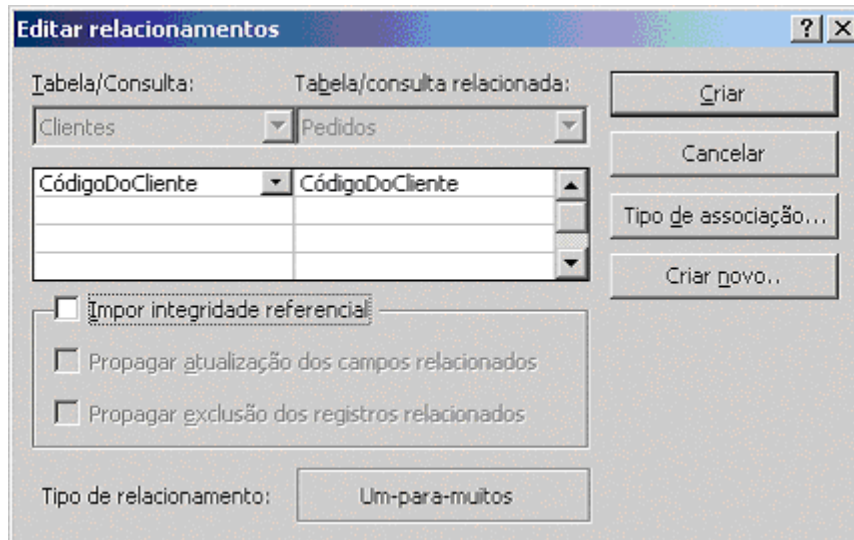
Para Definir Relacionamentos no Microsoft Access siga os passos indicados a seguir:

01. Abra o banco de dados onde estão as tabelas nas quais serão definidos os relacionamentos.
02. Selecione o comando Ferramentas -> Relacionamentos.
03. Surgirá a Janela indicada na próxima Figura. Nesta Janela você adicionará as Tabelas que farão parte de algum dos relacionamentos. Para Adicionar uma Tabela, basta marcá-la e dar um clique no botão "Adicionar". Você pode adicionar todas as tabelas de uma única vez. Para isto dê um clique na primeira, libere o mouse, pressione a tecla SHIFT e fique segurando SHIFT pressionado e dê um clique na última tabela. Com isso todas serão selecionadas, agora ao dar um clique no botão Adicionar, todas as tabelas selecionadas serão adicionadas. Caso não queira adicionar todas mas somente algumas e de uma maneira intercalada, ao invés de usar a tecla SHIFT, utilize a tecla CTRL. Com a tecla CTRL uma tabela é selecionada a medida que você vai clicando com o mouse sobre o nome da tabela.



Dê um clique para marcar a Tabela e depois dê um clique no botão Adicionar

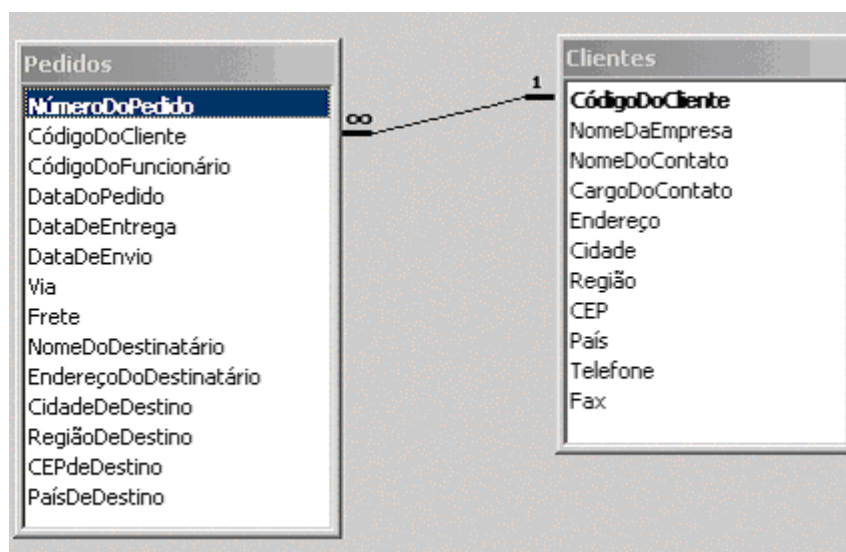
04. Após ter adicionado as tabelas, para criar um relacionamento, basta arrastar um campo de uma tabela sobre o campo da outra tabela na qual será estabelecido o relacionamento. Por exemplo, para estabelecer o relacionamento Um-para-Vários entre as tabelas **Clientes** e **Pedidos**, arraste o campo "CódigoDoCliente" da tabela Clientes, sobre o campo "CódigoDoCliente" da Tabela Pedidos. Ao largar um campo sobre o Outro, o Microsoft Access abre uma janela conforme indicado na figura a seguir (Definindo as características do Relacionamento):



Definindo as Características do Relacionamento

05. Observe que, por padrão, o campo "**Importar Integridade Referencial**" não está marcado. Ao marcá-lo serão habilitadas as Opções de "Propagar atualização dos campos relacionados" e "Propagar exclusão dos registros relacionados". Observe, também, que o Microsoft Access já definiu este relacionamento como sendo do tipo Um-para-Vários. Isso acontece porque o Microsoft Access identifica o campo CódigoDoCliente na tabela Clientes como sendo do tipo chave primária e na tabela Pedidos como não sendo chave primária, o que automaticamente transforma o Relacionamento como sendo do tipo Um para Vários. Se em ambas as tabelas o campo CódigoDoCliente fosse definido como Chave Primária, o relacionamento, automaticamente, seria do tipo Um-para-Um.

06. Após marcar as Opções desejadas, basta dar um clique no botão "Criar" e pronto, o Microsoft Access cria o Relacionamento, o qual é indicado através de uma linha entre as duas tabelas (Clientes e Pedidos), com o número 1 no lado da Chave Primária e o Sinal de infinito no lado Vários. Caso você precise alterar as características de um determinado relacionamento, basta dar um duplo clique sobre a linha do relacionamento, que o Microsoft Access abrirá a janela indicada na figura anterior, para que você possa fazer as alterações desejadas. Na Figura a seguir indico o relacionamento já criado entre as tabelas Pedidos e Clientes:



Relacionamento entre Pedidos e Clientes

07. Observe também que os campos Chave Primária aparecem em Negrito no Diagrama dos Relacionamentos.

08. Este diagrama que exibe as Tabelas e os Relacionamentos entre as tabelas é conhecido como "**Diagrama Entidades x Relacionamentos (DER)**".

09. Antes de fechar o Diagrama Entidades x Relacionamentos, dê um clique no botão com o desenho do disquete para salvar as alterações que foram feitas. A qualquer momento você pode acessar o Diagrama Entidades x Relacionamentos para fazer alterações ou para revisar os relacionamentos, para isto basta ir no menu Ferramentas e clicar em Relacionamentos.

Conclusão

Nesta terceiro artigo da série, você aprendeu sobre o conceitos de Integridade Referencial, sem dúvidas um dos conceitos mais importantes do Modelo Relacional, juntamente com os conceitos de Relacionamentos, abordados na Parte II. Na Parte IV falarei sobre os princípios básicos de normalização de tabelas. Até lá.

O Modelo Relacional de Dados - Parte 04

Na primeira parte deste artigo falei sobre Entidades (tabelas), Atributos (campos) e sobre o conceito de Chave Primária. Na segunda parte falei sobre Relacionamentos e tipos de relacionamentos. Na terceira parte falei sobre um dos conceitos mais importantes do modelo relacional de dados: Integridade Referencial. Também mostrei um exemplo prático de como configurar Relacionamentos e Integridade Referencial no Microsoft Access. Nesta quarta parte sobre os fundamentos do Modelo Relacional de dados, falarei sobre Normalização de banco de dados e as três principais formas normais.

Nota: Os exemplos apresentados utilizarão telas do Microsoft Access e o arquivo de exemplos Northwind.mdb, o qual é instalado juntamente com o Microsoft Access. Este arquivo está disponível, por padrão, no seguinte caminho:

C:\Arquivos de programas\Microsoft Office\Office\Samples

Porém os princípios básicos do modelo relacional aplicam-se a qualquer banco de dados baseado no modelo relacional de dados. Estes bancos de dados são algumas vezes denominados: SGBDR - Sistemas Gerenciadores de Banco de Dados Relacionais.

Normalização de tabelas

Objetivo: O objetivo da normalização é evitar os problemas provocados por falhas no Projeto do Banco de Dados, bem como eliminar a "mistura de assuntos" e as correspondentes repetições desnecessárias de dados. Uma Regra de Ouro que devemos observar quando do Projeto de um Banco de Dados baseado no Modelo Relacional de dados é a de **"não misturar assuntos em uma mesma Tabela"**. Por exemplo na Tabela Clientes devemos colocar somente campos relacionados com o assunto Clientes. Não devemos misturar campos relacionados com outros assuntos, tais como Pedidos, Produtos, etc. Essa "Mistura de Assuntos" em uma mesma tabela acaba por gerar repetição desnecessária dos dados bem como inconsistência dos dados.

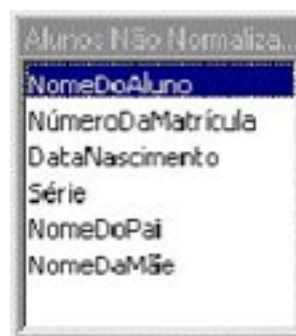
O Processo de Normalização aplica uma série de Regras sobre as Tabelas de um Banco de Dados, para verificar se estas estão corretamente projetadas. Embora existam 5 formas normais (ou regras de Normalização), na prática usamos um conjunto de 3 Formas Normais.

Normalmente após a aplicação das Regras de Normalização, algumas tabelas acabam sendo divididas em duas ou mais tabelas, o que no final gera um número maior de tabelas do que o originalmente existente. Este processo causa a simplificação dos atributos de uma tabela, colaborando significativamente para a estabilidade do modelo de dados, reduzindo-se consideravelmente as necessidades de manutenção. Vamos entender o Processo de Normalização na Prática, através de exemplos.

Primeira Forma Normal:

"Uma Tabela está na Primeira Forma Normal quando seus atributos não contém grupos de Repetição".

Por isso dissemos que uma Tabela que possui Grupos de Repetição não está na Primeira Forma Normal. Considere a estrutura da Tabela Indicada na Próxima Figura:



| |
|-------------------------|
| Alunos Não Normaliza... |
| NomeDoAluno |
| NúmeroDaMatricula |
| DataNascimento |
| Série |
| NomeDoPai |
| NomeDaMãe |

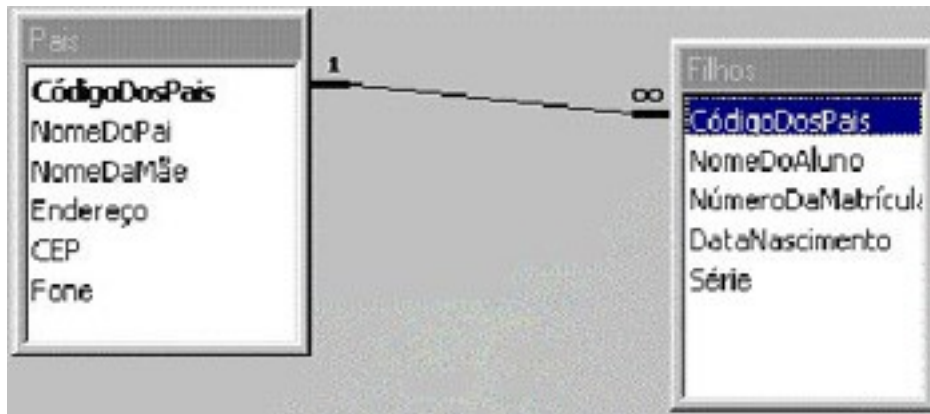
Tabela que não está na Primeira Forma Normal

Uma tabela com esta estrutura apresentaria diversos problemas. Por exemplo se um casal tiver mais de um filho, teremos que digitar o Nome do Pai e da Mãe diversas vezes, tantas quantos forem os filhos. Isso forma um Grupo de Repetição. Além do mais pode ser que por erro de digitação o Nome dos Pais não seja digitado exatamente igual todas as vezes, o que pode acarretar problemas na hora de fazer pesquisas ou emitir relatórios.

Este problema ocorre porque "Misturamos Assuntos" em uma mesma tabela. Colocamos as

informações dos Pais e dos Filhos em uma mesma tabela. A solução para este problema é simples: Criamos uma tabela separada para a Informação dos Pais e Relacionamos a tabela Pais com a Tabela Filhos através de um relacionamento do tipo Um para Vários, ou seja, um casal da Pais pode ter Vários Filhos.

Observe na figura abaixo as duas tabelas: Pais e Filhos, já normalizadas.



Informações sobre Pais e Filhos em Tabelas Separadas

As duas tabelas Resultantes da Aplicação da Primeira Forma Normal: Pais e Filhos estão na Primeira Forma Normal, a Tabela Original, a qual misturava informações de Pais e Filhos, não estava na Primeira forma Normal

Segunda Forma Normal:

Ocorre quando a chave Primária é composta por mais de um campo. Neste caso, devemos observar se todos os campos que não fazem parte da chave dependem de todos os campos que compõem a chave. Se algum campo depender somente de parte da chave composta, então este campo deve pertencer a outra tabela. Observe o Exemplo Indicado na Tabela da Figura abaixo:

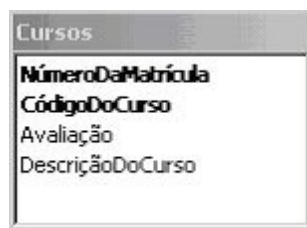


Tabela com uma Chave Primária Composta. Não está Na Segunda Forma Normal

A Chave Primária Composta é formada pela combinação dos Campos "NúmeroDaMatrícula" e "CódigoDoCurso". O Campo Avaliação depende tanto do CódigoDoCurso quanto do NúmeroDaMatrícula, porém o campo DescriçãoDoCurso, depende apenas do CódigoDoCurso, ou seja, dado o código do curso é possível localizar a respectiva descrição, independentemente do NúmeroDaMatrícula. Com isso temos um campo que não faz parte da Chave Primária e depende apenas de um dos campos que compõem a chave Primária Composta, por isso que dizemos que esta tabela não está na Segunda Forma Normal.

A Resolução para este problema também é simples: "Dividimos a Tabela que não está na Segunda Forma Normal em duas outras tabelas, conforme indicado pela figura abaixo, sendo que as duas tabelas resultantes estão na Segunda Forma Normal.



Informações sobre Avaliações e Cursos em Tabelas Separadas

Obs.: A Distinção entre a Segunda e a Terceira forma normal, que veremos logo em seguida, muitas vezes é confusa. A Segunda Forma normal está ligada a ocorrência de Chaves Primárias compostas.

Terceira Forma Normal:

Na definição dos campos de uma entidade podem ocorrer casos em que um campo não seja dependente diretamente da chave primária ou de parte dela, mas sim dependente de um outro campo da tabela, campo este que não a Chave Primária.

Quando isto ocorre, dizemos que a tabela não está na Terceira Forma Normal, conforme indicado pela tabela da figura abaixo:

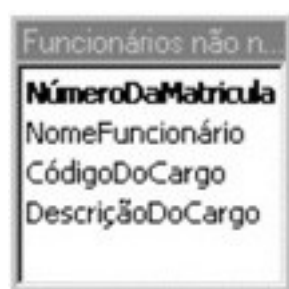


Tabela com um Campo dependente de Outro campo que não a Chave Primária. Não está na Terceira Forma Normal

Observe que o Campo **DescriçãoDoCargo** depende apenas do Campo **CódigoDoCargo**, o qual não faz parte da Chave Primária. Por isso dizemos que esta tabela não está na terceira forma normal. A Solução deste problema também é simples. Novamente basta dividir a tabela em duas outras, conforme indicado pela figura a seguir. As duas tabelas resultantes estão na Terceira Forma Normal.



Tabelas Resultantes que estão na Terceira Forma Normal

Com isso podemos concluir que como resultado do Processo de Normalização, iremos obter um número maior de tabelas, porém sem problemas de redundância e inconsistência dos dados.

Conclusão:

Neste quarto artigo da série você aprendeu sobre os conceitos de Normalização e formas normais.

Sem dúvidas um dos conceitos mais importantes do Modelo Relacional. Na quinta e última parte falarei sobre os princípios básicos para projeto de um banco de dados. Até lá!

O Modelo Relacional de Dados - Parte 05

Objetivo: Na primeira parte deste artigo falei sobre Entidades (tabelas), Atributos (campos) e sobre o conceito de Chave Primária. Na segunda parte falei sobre Relacionamentos e tipos de relacionamentos. Na terceira parte falei sobre um dos conceitos mais importantes do modelo relacional de dados: Integridade Referencial. Também mostrei um exemplo prático de como configurar Relacionamentos e Integridade Referencial no Microsoft Access. Na quarta parte sobre os fundamentos do Modelo Relacional de dados, falei sobre Normalização de banco de dados e as três principais formas normais.

Nesta quinta e última parte falarei sobre o projeto de banco de dados e ressaltarei a importância de fazer a Modelagem do Banco de Dados, antes de partir para a implementação prática. Falarei também sobre a Arquitetura do Microsoft Access. O entendimento do Modelo Relacional e da Arquitetura do Microsoft Access são fundamentais.

Muitos dos e-mails que recebo, com dúvidas sobre o Access, são relativo à dúvidas que seriam solucionadas com o conhecimento do Modelo Relacional e da Arquitetura do Access. O que acontece, muitas vezes, é que o usuário parte diretamente para o uso do Access, criando tabelas, consultas, formulários e relatórios, sem antes ter feito um projeto cuidadoso da estrutura do banco de dados, implementando relacionamentos, fazendo a normalização e impondo Integridade referencial. Trabalhar desta maneira é como fazer um prédio sem ter antes feito um estudo do terreno, ter feito a planta e os cálculos estruturais. Você moraria em um prédio construído desta maneira? Eu não.

Nota: Os exemplos apresentados utilizarão telas do Microsoft Access e o arquivo de exemplos Northwind.mdb, o qual é instalado juntamente com o Microsoft Access. Este arquivo está disponível, por padrão, no seguinte caminho:

C:\Arquivos de programas\Microsoft Office\Office\Samples

Porém os princípios básicos do modelo relacional aplicam-se a qualquer banco de dados baseado no modelo relacional de dados. Estes bancos de dados são algumas vezes denominados: SGBDR - Sistemas Gerenciadores de Banco de Dados Relacionais.

Normalização de tabelas

Objetivo: O objetivo da normalização é evitar os problemas provocados por falhas no Projeto do Banco de Dados, bem como eliminar a "mistura de assuntos" e as correspondentes repetições desnecessárias de dados.

Projetando um Banco de Dados

Neste item você aprenderá a projetar um Banco de Dados. Você aplicará os conhecimentos sobre Tabelas, Campos, Relacionamentos, Chave Primária e Normalização, vistos anteriormente, nas primeiras partes deste artigo.

Antes de começar a trabalhar com o Microsoft Access, é preciso fixar bem os conceitos vistos nas partes anteriores, aplicando-os no Projeto de um Banco de Dados. Um banco de dados bem projetado fornece um acesso conveniente às informações desejadas. Com uma boa estrutura, gasta-se menos tempo na construção de um banco de dados e, ao mesmo tempo, assegura-se resultados

mais rápidos e precisos. Nunca é demais lembrar que jamais devemos misturar assuntos em uma mesma tabela.

Etapas na estruturação e projeto de um Banco de dados:

- Determinar qual o objetivo do banco de dados: Isto ajuda na determinação de quais os dados devem ser armazenados. É fundamental ter bem claro qual o objetivo a ser alcançado com o banco de dados. É fazer o acompanhamento das despesas, a evolução das vendas ou outro objetivo qualquer.
- Determinar as tabelas necessárias: Após definirmos os objetivos do Banco de Dados, as informações devem ser definidas e separadas em assuntos diferentes, tais como "Clientes", "Empregados", "Pedidos", pois cada um irá compor uma tabela no banco de dados. Lembre-se da regrinha número um: "Não misturar assuntos na mesma tabela", ou seja, uma coisa é uma coisa e outra coisa é outra coisa.
- Determinar os Campos de cada Tabela: Definir quais informações devem ser mantidas em cada tabela. Por exemplo, a tabela Clientes poderia ter um campo para o Código Do Cliente, outro para o Nome Do Cliente e assim por diante.
- Determinar a Chave Primária de cada tabela, sendo que pode haver tabelas onde não exista uma chave primária: Determinar, em cada tabela, quais campos serão utilizados como Chave Primária. Esta é uma etapa importantíssima para a definição dos Relacionamentos que vem a seguir. Pode haver tabelas onde não exista uma chave primária.
- Determinar os Relacionamentos: Decidir como os dados de uma tabela se relacionam com os dados de outras tabelas. Por exemplo, Clientes podem Fazer Vários Pedidos, então existe um relacionamento do tipo Um-para-vários entre a tabela Clientes (lado um) e a tabela Pedidos (lado vários). Fornecedores podem fornecer Vários Produtos, etc.
- Refinar a Estrutura do Banco de Dados: Antes de inserir muitos dados, ou até mesmo antes de inserir qualquer dado, verificar se a estrutura contém erros, isto é, verificar se os resultados obtidos são os desejados. Isto, normalmente, pode ser obtido através do processo de Normalização. Caso necessário, deve-se alterar a estrutura do banco de dados.

Com uma boa estrutura, gasta-se menos tempo na construção e manutenção do banco de dados e, ao mesmo tempo, assegura-se resultados mais rápidos e precisos.

Dicas para determinação dos campos de uma Tabela:

- Relacionar diretamente cada campo ao assunto da tabela: Se um campo descreve o assunto de uma tabela diferente, este campo deve pertencer a outra tabela. O mesmo acontece quando uma informação se repete em diversas tabelas. Este é um indício de que existem campos desnecessários em algumas tabelas.
- Não Incluir dados Derivados ou Calculados: Não é recomendado armazenar o resultado de cálculos nas tabelas. O correto é que o cálculo seja executado quando necessitarmos do resultado, normalmente em uma consulta.
- Incluir todas as informações necessárias: Como é fácil esquecer informações importantes, deve-se ter em mente todas as informações coletadas desde o início do processo e perguntar se com elas é possível obter todos os resultados desejados.
- Armazenar todas as informações separadamente: Existe uma tendência em armazenar informações

em um único campo. Por exemplo, o nome do curso e o tempo de duração em uma mesmo campo. Como as duas informações foram combinadas em um único campo, ficará difícil conseguir um relatório classificado pelo tempo de duração dos cursos.

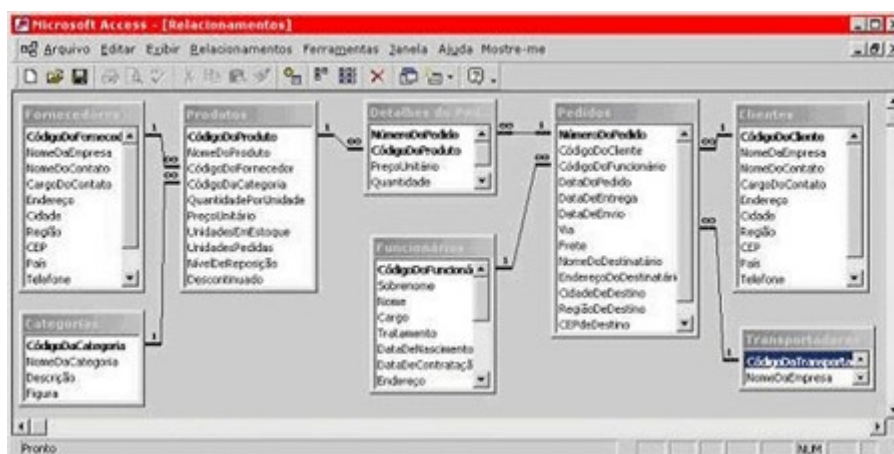
Como selecionar o campo que será a Chave Primária?

Um bom Sistema Gerenciador de Banco de Dados (SGBD) é aquele que encontra e nos fornece, rapidamente, todas as informações necessárias que nele estejam armazenadas, mesmo que estas informações estejam em diferentes tabelas. Para que isto seja possível é necessário incluir um campo ou conjunto de campos que identifiquem de modo único cada registro de uma tabela. Esta informação é chamada Chave Primária. Deve-se ter certeza que este campo (ou conjunto de campos) seja sempre diferente para cada registro, por não ser permitido valores duplicados em um campo de chave primária.

Ao escolher campos de Chave Primária, considere os seguintes detalhes:

- Não é permitido duplicidade de valores ou nulos (informações desconhecidas).
- Caso não exista um identificador único para uma determinada tabela, pode-se usar um campo que numere os registros sequencialmente.
- Pode-se utilizar o valor deste campo para encontrar registros.
- O tamanho da chave primária afeta a velocidade das operações, portanto, para um melhor desempenho, devemos utilizar o menor tamanho que acomode os valores necessários que serão armazenados no campo.

Agora que já revisamos diversos conceitos importantes sobre banco de dados vamos colocá-los em prática, através de um exercício de Projeto de Banco de Dados. Será apresentada uma determinada situação e você deverá projetar o Banco de Dados para atender a Situação Solicitada. Projetar o banco de dados significa fazer um diagrama Entidade x Relacionamentos onde são indicadas quais tabelas farão parte do banco de dados, quais os campos de cada tabela, qual o campo que será a Chave Primária nas tabelas que terão Chave Primária e quais os relacionamentos entre as tabelas. Na figura a seguir temos um exemplo de um diagrama Entidades x Relacionamentos:



Clique na imagem para vê-la em tamanho real:

Nota: Os campos que aparecem em **negrito** representam a Chave Primária de cada tabela.

Exercício: Imagine que você está projetando um Banco de Dados para uma Escola. Este Banco de

Dados deverá conter informações sobre os Alunos, os Pais dos Alunos, As matérias em que cada aluno está matriculado (imagine que alunos da mesma série podem estar matriculados em diferentes matérias), as notas do aluno em cada matéria e em cada bimestre, bem como todo o histórico do aluno na escola. O histórico inclui as notas do aluno em cada matéria em cada um dos anos em que ele esteve na escola. O banco de dados deve manter um cadastro de alunos, dos pais dos alunos, das disciplinas ofertadas, da nota de cada aluno em cada disciplina, e em que disciplina cada aluno está matriculado.

O Sistema deverá ser capaz de fornecer, a qualquer momento, a situação atual do aluno em termos de suas notas, bem como todo o seu histórico. O Sistema não deve permitir que seja cadastrado um aluno sem antes serem cadastrados os seus pais. Além disso todo aluno terá um número de matrícula que é único. Cada disciplina também terá um código único.

O Sistema deve ser capaz de emitir relatórios com as notas por turma e por bimestre, além das médias para cada disciplina.

Projete um Banco de Dados capaz de atender a estas necessidades. O Resultado final do seu trabalho será o "Diagrama Entidades x Relacionamentos", com as Tabelas, Campos de Cada Tabela, Chaves Primárias e Relacionamentos entre as tabelas.

Ao Final do Processo, aplique o processo de Normalização para verificar se a estrutura apresenta algum tipo de Problema.

Arquitetura do Microsoft Access

Neste item iremos analisar a Arquitetura do Microsoft Access Veremos os diversos elementos que podem fazer parte de um Banco de Dados do Microsoft Access, bem como os relacionamentos entre estes diversos elementos. Veremos também alguns exemplos práticos de solução de problemas.

Abordarei os seguintes tópicos:

- Os diversos elementos do Microsoft Access e a Relação entre os eles.
- Alguns Exemplos de Situações do dia-a-dia.

Os Diversos Elementos do Access e a Relação entre eles:

Um banco de dados é uma coleção de informações relacionadas a um determinado assunto ou finalidade, como controle de pedidos dos clientes ou manutenção de uma coleção de CDs e assim por diante. Se o seu banco de dados não está armazenado em um computador, ou se somente partes dele está, você pode estar controlando informações de uma variedade de fontes, tendo que coordená-las e organizá-las você mesmo.

Um arquivo .mdb é um Banco de Dados do Microsoft Access. Esse banco de dados contém diversos elementos: Tabelas, Consultas, Formulários, Relatórios, Macros, Páginas de dados e Módulos.

Utilizando o Microsoft Access, você pode gerenciar todas as suas informações a partir de um único arquivo de banco de dados. Dentro do arquivo, divida seus dados em compartimentos de armazenamento separados denominados tabelas (ou entidades); visualize, adicione e atualize os dados da tabela utilizando formulários on-line; localize e recupere apenas os dados desejados utilizando consultas; e analise ou imprima dados em um layout específico utilizando relatórios.

Para armazenar seus dados, crie uma tabela para cada tipo de informação que você registra. Para reunir os dados de várias tabelas em uma consulta, formulário ou relatório, você define relacionamentos entre as tabelas. Aqui nos temos dois fatos de grande importância:

- Todos os dados ficam armazenados em Tabelas. Quando uma Consulta exhibe os resultados com

base em um Critério, na verdade ele está buscando os dados em uma determinada tabela. Quando um formulário exibe um determinado registro, ele também está buscando estes dados em uma determinada tabela. No Microsoft Access, o único local onde os dados ficam armazenados é nas tabelas.

- Mesmo que as informações estejam separadas em diferentes tabelas (Clientes, Pedidos, Detalhes do Pedido, etc) é possível reuni-las em Consultas, Relatórios e Formulários. Por exemplo, posso criar um Relatório de Vendas por Cliente, classificados pelo País de Destino.

Na Figura a seguir, vemos os diversos elementos que formam um Banco de Dados do Microsoft Access, bem como o Relacionamento entre os diversos elementos.



Os Diversos Elementos de um Banco de Dados do Microsoft Access.

Nunca é demais salientar que o único local onde ficam armazenados os dados é nas tabelas. Por isso que ao construirmos uma consulta, formulário ou relatório, o Microsoft Access solicita os dados para a tabela na qual a consulta, formulário ou relatório está baseado.

Algumas Observações sobre os Elementos do Microsoft Access:

- Observe o Relacionamento que existe entre os Elementos. Uma consulta é baseada em uma tabela, isto é, os dados que a consulta exibe são buscados a partir de uma ou mais tabelas. Se os dados forem alterados na consulta, na verdade estas alterações são refletidas diretamente na tabela. Por isso uma seta de dupla mão entre tabelas e consultas. As mesmas observações são válidas para a relação entre formulários e tabelas.
- Um relatório também pode ser baseado diretamente em uma consulta, assim como um Formulário também pode ser baseado diretamente em uma Consulta. Quando o Formulário (ou Relatório) é aberto, o Microsoft Access executa a consulta, a qual busca os dados na Tabela, e retorna os dados para o Formulário (Ou Relatório).
- Observe que as Macros e Módulos foram colocados ao redor, envolvendo os demais elementos. Isto significa que posso ter Macros e Módulos interagindo com qualquer elemento de um banco de dados do Microsoft Access. Por exemplo, posso criar uma macro que Maximize um formulário quando o formulário é aberto. Posso criar um módulo para calcular o Dígito Verificador de um campo CPF, de tal forma que quando um CPF é digitado, o CPF não é aceito se estiver com o Dígito Verificador incorreto.

• Uma situação bastante comum é o caso em que precisamos de um relatório, porém os dados necessários não estão na forma necessária nas tabelas. Neste caso podemos criar uma consulta que selecione os dados necessários e faça as consolidações necessárias e criamos o Relatório baseado nesta consulta e não diretamente na tabela.

Estes seis elementos: Tabelas, Consultas, Formulários, Relatórios, Macros e Módulos podem ser criados e gerenciados a partir da Janela Principal do Banco de Dados do Microsoft Access, conforme indicado a seguir:

Janela "Banco de Dados", dando acesso aos diversos elementos do Microsoft Access.

Alguns exemplos e situações do dia-a-dia:

Para ilustrar o relacionamento entre os diversos elementos do Microsoft Access, vamos considerar algumas situações usuais do dia-a-dia. Nossas situações serão baseadas no arquivo Northwind.mdb, o qual é instalado juntamente com o Microsoft Access, conforme descrito na Introdução.

Situação 01: Vamos supor que seja solicitado um Relatório com os totais por Pedido. Como atender esta demanda?

Solução: Os dados necessários não estão disponíveis diretamente na tabela Pedidos. Criaremos uma consulta baseada nas tabelas Pedidos e Detalhes do Pedido. Esta consulta fará o cálculo do total por Número de Pedido. Nosso Relatório será baseado nesta consulta. Quando o Relatório for aberto, a consulta será acionada, buscará os valores nas tabelas Pedidos e Detalhes dos Pedidos, realizará os cálculos e fornecerá os valores para o Relatório. Aprenderemos a criar este tipo de consulta no decorrer deste curso.

Situação 02: Como fazer um relatório que exiba o total de Vendas por país de Destino e por Ano?

Solução: Novamente os dados necessários não estão disponíveis diretamente nas tabelas Pedidos e Detalhes do Pedido. Crio uma consulta do tipo Tabela de Referência Cruzada, baseada na Consulta que calcula os totais por Pedido. Nesta consulta adiciono um campo para o Ano do Pedido e outro campo para o País de Destino. Crio o meu relatório baseado nesta consulta. Ao ser aberto o Relatório é acionada a consulta do tipo Tabela de Referência Cruzada, a qual por sua vez aciona a Consulta na qual ela é baseada, a qual busca os dados nas tabelas. O Caminho inverso é percorrido, até que os dados são fornecidos para o Relatório, o qual exibe os mesmos na tela ou imprime na Impressora. Aprenderemos a criar este tipo de consulta no decorrer deste curso.

Situação Desafio: Os dados sobre o cabeçalho do Pedido (tabela Pedidos) e os diversos itens de cada Pedido (Tabela Detalhes do Pedido) estão separados. Como reunir estas informações em um formulário de tal maneira que o mesmo se pareça com uma Nota Fiscal?

Solução: Desafio!

Conclusão:

Bem amigo leitor, com este artigo, encerramos a série sobre o Modelo Relacional de dados. A recomendação mais enfática que eu posso fazer é a seguinte: "Não pense que fazer modelagem de dados é perda de tempo". Muito pelo contrário: É ganho de tempo e de qualidade em nossos programas. Sem a modelagem, você vai criando o banco na tentativa e erro, quando uma consulta não dá certo o programador faz um "gambiarra", adapta aqui, adapta ali e assim vai. Isso não é programação. Perdoem-me por ser enfático neste ponto. Isso é remendo, é tentativa e erro. Por isso que temos muitos programas com péssimo desempenho, funcionalidades que deixam a desejar, sem documentação, impossíveis de serem alterados a não ser por quem os criou.

É meu mais sincero desejo que eu possa ter colaborado para, pelo menos, despertar uma reflexão no amigo leitor. Não deixe de acompanhar a série de artigos sobre TCP/IP, a qual passarei a publicar quinzenalmente. Um forte abraço.

Links dos originais:

http://imasters.uol.com.br/artigo/2419/bancodedados/o_modelo_relacional_de_dados_-_parte_01/

http://imasters.uol.com.br/artigo/2455/bancodedados/o_modelo_relacional_de_dados_%E2%80%93_parte_02/

http://imasters.uol.com.br/artigo/2477/bancodedados/o_modelo_relacional_de_dados_-_parte_03/

http://imasters.uol.com.br/artigo/2521/bancodedados/o_modelo_relacional_de_dados_-_parte_04/

http://imasters.uol.com.br/artigo/2530/bancodedados/o_modelo_relacional_de_dados_-_parte_05/

4) Projeto de Bancos de Dados (Arquitetura)

- 2.1) Conceitos
- 2.2) Modelo Entidade Relacionamento
- 2.3) Modelo Objeto Relacional
- 2.4) Normalização
- 2.5) Integridade Referencial
- 2.6) Planejamento e Otimização

Iniciar com os conceitos básicos: dado, informação e conhecimento.

Dado – representação simbólica, quantificada ou quantificável. Um texto, um número e uma foto são exemplos de dados. O computador somente trabalha com dados, nada de informações e muito menos de conhecimento.

Informação – são mensagens sob forma de dados, que são recebidas e compreendidas. Caso não seja compreendida elas são apenas dados.

Portanto dados são entes meramente *sintáticos* (estruturas).

As informações contém *semântica*, ou seja, contém um significado próprio. A informação é algo objetivo e subjetivo.

Conhecimento – enquanto a informação é um conhecimento teórico, o conhecimento é algo prático.

Só podemos transmitir dados, jamais informações e conhecimento. Os dados podem ser transmitidos, ao serem interpretados tornam-se informações, mas somente ao serem experimentados transformam-se em conhecimento.

Fonte: Livro Bancos de Dados de Valdemar W. Setzer e Flávio Soares Corrêa da Silva.

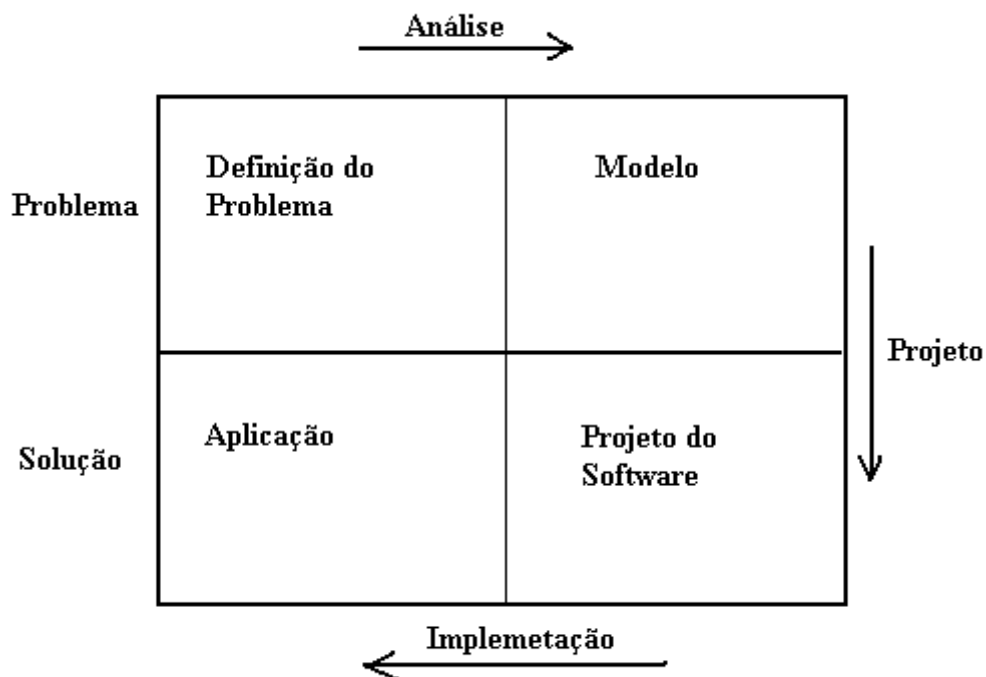
2.1) Conceitos

| Termo | Conceito |
|----------------------------|---|
| Modelo Relacional | A maioria dos SGBDs existentes são baseados no modelo relacional. |
| SGBD | É um sistema de computador cuja função principal é a de manter bancos de dados e fornecer informações os mesmos quando solicitados. |
| DBA | DataBase Administrator (Administrador de Bancos de Dados), é quem trabalha criando os bancos de dados, implementando controle técnico e administrativo dos dados e implementando a segurança. |
| Entidades | Ou classes, podem ser concretas ou abstratas e representam pessoas, lugares ou coisas (objetos). Entidade no projeto representa Tabela na implementação. |
| Atributos | Ou propriedades. As entidades são compostas por atributos. Atributos no projeto correspondem a campos na implementação. |
| Relacionamentos | Associação entre as entidades. |
| Projeto | Modelo, lógico. |
| Implementação | Criação e administração, físico. |
| Modelo de dados | É uma definição lógica e abstrata de um banco de dados. |
| Implementação | É a aplicação física (prática) do modelo de dados em um computador. |
| Aplicação cliente | Também chamada de front-end do banco de dados. Roda sobre um SGBD. |
| DDL | Conjunto de instruções para definir ou declarar objetos do banco de dados. |
| DML | Conjunto de instruções para manipular ou processar os objetos do banco de dados. |
| Dicionário de dados | É um conjunto de tabelas do banco de dados que contém informações sobre os objetos do banco, dados sobre dados (metadados). |
| Bancos distribuídos | Banco que é logicamente centralizado e fisicamente distribuído. A distribuição pode ser em vários sites. Os sites podem estar em vários SO diferentes, redes e máquinas diferentes. |
| Projeto lógico | É a fase de identificação das entidades de interesse para a empresa e de identificação das informações a serem armazenadas nas entidades. Deve acontecer antes do projeto físico (implementação). |
| CAST | Valores podem ser convertidos de um tipo para outro através do |

| | |
|----------------------------|---|
| | operador CAST ou através de coerção implícita. O PostgreSQL em sua versão 8.3 está passando a usar bem mais as coerções explícitas. |
| Projeto de banco | Projetar bancos tem mais de arte que de ciência. Um projeto adequado previne perda da integridade dos dados e permite consultas adequadas e eficientes. Qualquer projeto requer uma noção exata do que o banco deverá armazenar. Temos que colher o máximo de informações junto ao cliente e ir além percebendo necessidades que ele próprio não percebe. |
| Casos de uso | São textos com formatação livre, que descrevem operações do ponto de vista de um eventual usuário interagindo com o sistema. Uma boa forma de capturar as finalidades do banco é construindo casos de uso. |
| Modelagem dos dados | Exige que reflitamos cuidadosamente sobre os diferentes conjuntos ou classes (entidades) necessárias para solucionar o problema. |
| Dados | Representação simbólica (abstrata). |
| Informação | Dados com significado. Mensagem compreendida é informação, caso contrário são apenas dados. Computadores armazenam apenas dados e não informação. |
| MER | É um modelo que não pode ser representado num computador, existindo apenas na mente de uma pessoa. Pode ser representado como texto ou em forma de diagrama gráfico (DER). |
| IDs | Caso um registro tenha todos os valores dos campos semelhantes aos valores de outro registro, com exceção do ID, então isso mostra que o ID não é adequado para chave primária da tabela. |
| Chave Primária | Formada por um campo ou mais que identifica todos os registros de uma tabela de forma única. |
| Chave Candidata | É a chave onde nenhum subconjunto de campos é também uma chave. |
| Superchave | É a chave formada por um conjunto de campos além do necessário para identificar todos os registros de forma única, ou seja, contém a chave primária mais um ou mais campos. |
| Chave alternativa | Quando uma tabela contém uma chave primária, qualquer outra chave será uma chave alternativa. |

Projeto do Banco de Dados – Quando um atributo de uma entidade armazena valores duplicados, isso indica que devemos criar uma nova entidade para armazenar os valores deste atributo e então relacionar as entidades através desse atributo.

CPF no Brasil não é uma boa escolha para **chave primária** em algumas entidades, pois membros de uma família podem utilizar o mesmo CPF.

Diagrama de Zelzowitz

Este é um diagrama usado para o desenvolvimento de software mas também podemos aplicá-lo ao projeto de bancos de dados.

Análise – partindo do mundo real, define o problema e gera o modelo

Projeto – partindo do modelo gera o projeto do software (banco)

Implementação – partindo do projeto do software (banco) cria a aplicação.

Partimos da situação real, que deve ser **analisada** gerando um modelo que será transformado em **projeto** e então se **implementará** no computador.

Classes – objetos – atributos (propriedades e métodos)

Tabelas – registros – campos

Uma boa estratégia é portar um conjunto de perguntas a serem respondidas:

- 1) Qual o principal objetivo do sistema?
- 2) Que dados são necessários para satisfazer este objetivo
- 3) Que informações já existem: algum sistema de computador anterior, fichas, formulários, etc.
- 4) Que entidades foram identificadas?
- 5) Quais as informações de cada entidade?
- 6) Qual o rascunho do modelo?
- 7) Que saídas são esperadas?

Ao final elaborar um primeiro diagrama de classes e vários casos de uso, que relatem no formato texto as interações do sistema.

Anotar o que pode dar errado em cada fase do sistema.

História

As três primeiras formas normais foram definidas por **Ted Codd**.

O Modelo de Entidades e Relacionamentos foi introduzido por Peter Chen.

O MR (Modelo Relacional) foi definido em 1970 por E. F. Codd.

Modelo Relacional Normalizado

- Cada tabela tem um nome distinto
- Cada coluna tem um nome distinto
- Não existem dois registros iguais
- A ordem dos registros e dos campos é irrelevante

SQL – É uma linguagem de declaração e não de programação e atualmente é a linguagem padrão dos SGBDRs e SGBDRO.

Referências:

- Bancos de Dados – Aprenda o que são, Melhore seu conhecimento, Construa os seus
 - De Valdemar W. Seltzer e Flávio Soarea Corrêa da Silva
- Beginning Database Design
 - De Clare Churcher
- A Introduction to Database Systems
 - De C. J. Date

2.2) O Modelo Relacional

- ! Todos os dados são representados como tabelas
 - ! Os resultados de qualquer consulta é apenas mais uma tabela!
- ! Tabelas são formadas por linhas e colunas
- ! Linhas e colunas são (oficialmente) desordenadas (isto é, a ordem com que as linhas e colunas são referenciadas não importa).
 - ! Linhas são ordenadas apenas sob solicitação. Caso contrário, a sua ordem é arbitrária e pode mudar para um banco de dados dinâmico
 - ! A ordem das colunas é determinada por cada consulta
- ! Cada tabela possui uma **chave primária**, um identificador único constituído por uma ou mais colunas
- ! A maioria das chaves primárias é uma coluna apenas (por exemplo, TOWN_ID)

- | Uma tabela é ligada (conectada) à outra incluindo-se a chave primária da outra tabela. Esta coluna incluída é chamada uma **chave externa**
- | Chaves primárias e chaves externas são os conceitos mais importantes no projeto de banco de dados. Gaste o tempo necessário para entender o que são!

Qualidades de um Bom Projeto de Banco de Dados

- | Reflete a estrutura real do problema
- | Pode representar todos os dados esperados ao longo do tempo
- | Evita armazenamento redundante de itens de dados
- | Fornece acesso eficiente aos dados
- | Suporta a manutenção da integridade dos dados ao longo do tempo Limpa, consistente e fácil de entender
- | *Nota: Estes objetivos são algumas vezes contraditórios!*

Introdução à Modelagem Entidades - Relacionamento

- | Modelagem E-R: Um método para projetar banco de dados
- | Uma versão simplificada é apresentada aqui
- | Representa o dado por **entidades** que possuem **atributos**.
- | Uma entidade é uma classe de objetos ou conceitos identificáveis distintos
- | Entidades possuem **relações** umas com as outras
- | O resultado do processo é um banco de dados **normalizado** que facilita o acesso e evita dados duplicados

*Nota: Muito do projeto formal de banco de dados é focado em **normalizar** o banco de dados e assegurar que o projeto adere ao nível de normalização (isto é, primeira forma normal, segunda forma normal, etc.). Este nível de formalidade está além desta discussão, mas deve-se saber que tais formalizações existem.*

Processo de Modelagem E-R

- | Identifique as entidades que o seu banco de dados deve representar
- | Determine as relações de **cardinalidade** entre as entidades e classifique-as como
 - | **Um-para-um** (por exemplo, um imóvel tem um endereço)
 - | **Um-para-muitos** (por exemplo, um imóvel pode ser envolvido em muitos incêndios)
 - | **Muitos-para-muitos** (por exemplo, venda de imóveis: um imóvel pode ser vendido para muitos proprietários, e um proprietário individual pode vender muitos imóveis)
- | Desenhe o diagrama entidade-relação
- | Determine os atributos de cada entidade
- | Defina a chave primária (única) de cada entidade

Do modelo E-R para o Projeto de Banco de Dados

- | Entidades com relações **um-para-um** deve ser fundidas em uma única entidade
- | Cada entidade restante é modelada por uma tabela com uma chave primária e atributos, alguns dos quais podem ser chaves externas
- | Relações **Um-para-muitos** são modeladas por um atributo de chave externa na tabela

representando a entidade do lado "muitos" da relação

Relações **Muitos-para-muitos** entre duas entidades são modeladas por uma terceira tabela que possui chaves externas que referem-se às entidades. Estas chaves externas devem ser incluídas na chave primária da tabela da relação, se apropriado

Ferramentas disponíveis comercialmente podem automatizar o processo de conversão de um modelo E-R para um esquema de banco de dados

Fonte: <http://www.universia.com.br/mit/11/11208/pdf/lecture5-2.pdf> (de Thomas H. Grayson)

2.3) Modelo Objeto Relacional

A maioria dos sistemas de gerência de bancos de dados (SGBD) utilizados hoje em dia são baseados no modelo relacional, definido por E.F. Codd. Devido a grande demanda de dados complexos que devem ser armazenados (textos, imagens, som, etc), está havendo uma grande migração para sistemas que suportam esses tipos de dados. Entre as opções de sistemas, estão os bancos de dados orientados a objetos e os objeto-relacionais. Os bancos de dados objeto-relacionais estão situados entre os relacionais e os orientados a objetos. Com este resumo, queremos mostrar algumas características do modelo objeto-relacional.

Principais características do SGBD Objeto-Relacional

Devido a falta de padronização do modelo objeto-relacional, cada fabricante definiu as características do seu SGBD. Analizaremos algumas das principais características dos SGBDs objeto-relacional e daremos uma ênfase no SGBD Oracle 8.x, devido a sua grande popularidade como ORDBMS.

Uma das principais características é permitir que o usuário defina tipos adicionais de dados – especificando a estrutura e a forma de operá-lo – e use estes tipos no modelo relacional. Desta forma, os dados são armazenados em sua forma natural [SAPM 98].

Além dos tipos base já existentes, é permitido ao usuário criar novos tipos e trabalhar com dados complexos como imagens, som e vídeo, por exemplo.

Em última análise, o tipo objeto-relacional está evoluindo mais do que o orientado a objeto, tornando-se cada vez mais um novo produto, que não pode ser comparado com o orientado a objeto, pois este já é um produto maduro [MS 97].

Fonte:

<http://paginas.terra.com.br/informatica/arruda/Downloads/Artigos/artigo04/index.htm>

2.4) Normalização

Normalização de Tabelas

Normalizar bancos tem o objetivo de tornar o banco mais eficiente, reduzindo as redundâncias, evitando retrabalho em eventuais alterações futuras do modelo.

Uma regra muito importante advinda da normalização ao criar tabelas é atentar para que cada tabela contenha informações sobre um único assunto, de um único tipo.

Tabelas não devem conter campos nulos.

Tabelas não devem conter campos com valores repetidos.

Todas as tabelas de um banco devem estar interligadas por relacionamentos.

Geralmente a aplicação das três primeiras formas normais atende à maioria dos projetos e dificilmente precisamos recorrer à quarta e/ou quinta (somente quando estritamente necessário).

Normalização total não é obrigatório no MER mas fortemente recomendado.

1a Forma Normal

Uma entidade encontra-se nesta forma quando nenhum dos atributos se repete.

Exemplo: evitar cadastrar um produto duas vezes. Quando houver repetição devemos criar uma outra entidade com os campos repetidos e relacionar as tabelas por esse campo.

Exemplo:

clientes

cod nome uf

1 – joão – CE

2 – pedro – CE

3 – manoel – MA

Solução: Criar uma tabela para o campo UF e relacionar ambas.

estados

1 – CE

2 - MA

clientes

1 – joão – 1

2 – pedro – 1

3 – manoel - 2

Outro Exemplo:

Alunos: matricula, nome, data_nasc, serie, pai, mae

Se a escola tem vários filhos de um mesmo casal haverá repetição do nome dos pais. Estão para atender à primeira regra, criamos outra tabela com os nomes dos pais e a matrícula do aluno.

2ª Forma Normal

Quando a chave primária é composta por mais de um campo.

Todos os campos que não fazem parte da chave devem depender da chave (de todos os seus campos).

Caso algum campo dependa somente de parte da chave, então devemos colocar este campo em outra tabela.

Exemplo:

TabelaAlunos

Chave (matricula, codigo_curso)

avaliacao descricao_curso

Neste caso o campo descricao_curso depende apenas do codigo_curso, ou seja, tendo o código do curso conseguimos sua descrição. Então esta tabela não está na 2ª Forma Normal.

Solução:

Dividir a tabela em duas (alunos e cursos) e incorporar à chave existente:

TabelaAlunos

Chave (matricula, codigo_curso)

avaliacao

TabelaCursos

codigo_curso

descricao_curso

3ª Forma Normal

Quando um campo não é dependente diretamente da chave primária ou de parte dela, mas de outro campo da tabela que não pertence à chave primária. Quando isso ocorre esta tabela não está na terceira forma normal e a solução é dividir a tabela.

Um atributo comum não deve depender de outro atributo comum, mas somente da PK.

Exemplo:

Campo tipo total, que depende de deus fatores: quantidade e preco.

Solução:

Não devemos armazenar campos calculados na tabela, pois além de gerar inconsistência são desnecessários, já que podemos consegui-lo com uma operação.

4a. Forma Normal

Uma tabela está na 4FN, se e somente se, estiver na 3FN e não existirem dependências multivaloradas.

Exemplo: Dados sobre livros

Relação não normalizada: Livros(nrol, (autor), título, (assunto), editora, cid_edit, ano_public)

1FN: Livros(nrol, autor, assunto, título, editora, cid_edit, ano_public)

2FN: Livros(nrol,título, editora, cid-edit, ano_public)

AutAssLiv(nrol, autor, assunto)

3FN: Livros(nrol, título, editora, ano_public)

Editoras(editora, cid-edit)

AutAssLiv(nrol, autor, assunto)

* Redundância para representar todas as informações

* Evitar todas as combinações: representação não-uniforme (repete alguns elementos ou posições nulas)

Passagem à 4FN:

* Geração de novas tabelas, eliminando Dependências Multivaloradas

* Análise de Dependências Multivaloradas entre atributos:

* autor, assunto - Dependência multivalorada de nrol

5a. Forma Normal

Está ligada à noção de dependência de junção Se uma relação é decomposta em várias relações e a reconstrução não é possível pela junção das outras relações, dizemos que existe uma dependência de junção.

Existem tabelas na 4FN que não podem ser divididas em duas relações sem que se altere os dados originais.

Exemplo: Seja as relações R1(CodEmp, CodPrj) e R2(CodEmp, Papel) a decomposição da relação ProjetoRecuro(CodEmp, CodPrj, Papel).

Dica:

Em caso de dúvida, o modelo ideal é o menos engessado, mesmo que pague o preço de alguma redundância.

- Ao aplicar a 3a. FN ainda persistirem redundâncias, então divide-se a tabela em duas.

Exemplos de Relacionamentos tipo N para N

Autores – Músicas

Alunos – Disciplinas

Produtos – Fornecedores

Produtos – Pedidos

Fases na criação de um banco de dados (aplicativo):

- Análise
- Projeto
- Implementação
- Testes
- Homologação
- Administração

Análise

Nesta fase "enfrentamos" o mundo real para analisar e diagnosticar o problema para então passar um esboço do modelo para a fase de projeto.

Nessa fase procedemos ao levantamento, gerenciamento e validação de informações importantes para o modelo.

Análise é a fase que tem contato com o mundo real, coletando informações junto ao cliente (usuário). Depois da coleta realiza a modelagem das entidades e relacionamentos e das normalizações. Ao final deve gerar um diagrama das entidades e relacionamentos DER.

Projeto - definição das tabelas, índices, chaves (PK e FK), relacionamentos, views, etc.

Melhorando a Normalização de Tabelas

A tabela de CEPs

```
create table cep_full_index
(  
    cep char(8),  
    tipo char(72),  
    logradouro char(70),
```



```
bairro char(72),
municipio char(60),
uf char(2)
);
```

Este é um exemplo de tabela não normalizada.

O que podemos melhorar? Praticamente todos os campos se repetem muito, com exceção do cep, que é a chave primária.

Então deveríamos criar outras tabelas para cada um dos campos que se repetem.

Idealmente devemos normalizar as tabelas, pelo menos até a terceira forma normal, no momento da criação do banco, na fase de projeto do mesmo, mas para os casos em que

já encontramos a tabela em uso e com muitos registros, como é o caso desta, veja como se pode proceder.

Vamos mostrar como proceder para normalizar o campo uf, criando uma tabela externa com as ufs brasileiras importando da tabela atual:

Como Exemplo criaremos a Tabela ufs

-- Criar a tabela de ufs

```
create table ufs as select distinct(uf) as uf from cep_full_index;
select * from ufs
```

-- Adicionar o campo codigo na tabela ufs

```
alter table ufs add column codigo serial
```

-- Atualizar o campo uf na tabela cep_full_index para números de 1 a 27, ainda como CHAR(2)

```
update cep_full_index set uf=
case
  when uf ='AC' then '1'
  when uf ='AL' then '2'
  when uf ='AM' then '3'
  when uf ='AP' then '4'
  when uf ='BA' then '5'
  when uf ='CE' then '6'
  when uf ='DF' then '7'
  when uf ='ES' then '8'
  when uf ='GO' then '9'
  when uf ='MA' then '10'
  when uf ='MG' then '11'
  when uf ='MS' then '12'
  when uf ='MT' then '13'
  when uf ='PA' then '14'
```

```
when uf ='PB' then '15'  
when uf ='PE' then '16'  
when uf ='PI' then '17'  
when uf ='PR' then '18'  
when uf ='RJ' then '19'  
when uf ='RN' then '20'  
when uf ='RO' then '21'  
when uf ='RR' then '22'  
when uf ='RS' then '23'  
when uf ='SC' then '24'  
when uf ='SE' then '25'  
when uf ='SP' then '26'  
when uf ='TO' then '27'  
end;
```

```
-- Alterando o campo com CAST, de CHAR(2) para INT  
alter table cep_full_index alter column uf type int using cast(uf as int)  
select distinct(uf) from cep_full_index
```

```
select uf from cep_full_index where cep='60420440'
```

```
-- Adicionar o relacionamento na tabela de CEP  
alter table ufs add constraint ufs_pk primary key (codigo)  
alter table cep_full_index add constraint cep_uf_fk foreign key (uf) references ufs(codigo)
```

```
-- Testar novamente a mesma consulta  
select uf from cep_full_index where cep='60420440'
```

Fases do Projeto do Banco de Dados:

- Modelagem Conceitual
- Projeto Lógico

Modelo Conceitual - Define apenas quais os dados que aparecerão no banco de dados, sem se importar com a implementação do banco. Para essa fase o que mais se utiliza é o DER (Diagrama Entidade-Relacionamento).

Modelo Lógico - Define quais as tabelas e os campos que formarão as tabelas, como também os campos-chave, mas ainda não se preocupa com detalhes como o tipo de dados dos campos, tamanho, etc.

Não devemos misturar as tarefas de cada uma das fases.

Implementação - criação do banco de acordo com o projeto e também criação dos usuários, grupos e privilégios.

Análise e projeto são fases lógicas e implementação é física.

Comparando Construção de bancos de dados com edifícios:

Análise - reconhecimento do terreno e elaboração de croquis (esboço da planta).

Projeto - Elaboração das plantas de acordo com o esboço: planta baixa, de elevação, detalhes, de situação, etc.

Implementação - Construção do edifício.

Administração - manutenção do edifício.

2.5) Integridade Referencial

Tradução livre da documentação "CBT Integrity Referential":
http://techdocs.postgresql.org/college/002_referentialintegrity/.

Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere às informações em outra tabela e o banco de dados reforça a integridade.

Tabela1 -----> Tabela2
Onde é Utilizado?

Onde pelo menos em uma tabela precisa se referir para informações em outra tabela e ambas precisam ter seus dados sincronizados.

Exemplo: uma tabela com uma lista de clientes e outra tabela com uma lista dos pedidos efetuados por eles.

Com integridade referencial devidamente implantada nestas tabelas, o banco irá garantir que você nunca irá cadastrar um pedido na tabela pedidos de um cliente que não exista na tabela clientes.

O banco pode ser instruído para automaticamente atualizar ou excluir entradas nas tabelas quando necessário.

Primary Key (Chave Primária) - é o campo de uma tabela criado para que as outras tabelas relacionadas se refiram a ela por este campo. Impede mais de um registro com valores iguais. É a combinação interna de UNIQUE e NOT NULL.

Qualquer campo em outra tabela do banco pode se referir ao campo chave primária, desde que tenham o mesmo tipo de dados e tamanho da chave primária.

Exemplo:

clientes (codigo INTEGER, nome_cliente VARCHAR(60))

| codigo | nome_cliente |
|--------|-----------------|
| 1 | PostgreSQL inc. |

2 RedHat inc.

pedidos (relaciona-se à Clientes pelo campo cod_cliente)
cod_pedido cod_cliente descricao

Caso tentemos cadastrar um pedido com cod_cliente 2 ele será aceito.

Mas caso tentemos cadastrar um pedido com cod_cliente 3 ele será recusado pelo banco.

Criando uma Chave Primária

Deve ser criada quando da criação da tabela, para garantir valores exclusivos no campo.

```
CREATE TABLE clientes(cod_cliente BIGINT, nome_cliente VARCHAR(60)  
PRIMARY KEY (cod_cliente));
```

Criando uma Chave Estrangeira (Foreign Keys)

É o campo de uma tabela que se relaciona (associa) com o campo Primary Key de outra. O campo pedidos.cod_cliente refere-se ao campo clientes.codigo, então pedidos.cod_cliente é uma chave estrangeira, que é o campo que liga esta tabela a uma outra.

```
CREATE TABLE pedidos(  
cod_pedido BIGINT,  
cod_cliente BIGINT REFERENCES clientes,  
descricao VARCHAR(60)  
);
```

Outro exemplo:

```
FOREIGN KEY (camboa, campob)  
REFERENCES tabela1 (camboa, campob)  
ON UPDATE CASCADE  
ON DELETE CASCADE);
```

Cuidado com exclusão em cascata. Somente utilize com certeza do que faz.

Dica: Caso desejemos fazer o relacionamento com um campo que não seja a chave primária, devemos passar este campo entre parênteses após o nome da tabela e o mesmo deve obrigatoriamente ser UNIQUE.

```
...  
cod_cliente BIGINT REFERENCES clientes(nomecampo),  
...
```

Parâmetros Opcionais:

ON UPDATE parametro e ON DELETE parametro.

ON UPDATE paramentros:

NO ACTION (RESTRICT) - quando o campo chave primária está para ser atualizado a atualização é abortada caso um registro em uma tabela referenciada tenha um valor mais antigo. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2. Ele vai tentar atualizar o código para 5 mas como em pedidos existem registros do cliente 2 haverá o erro.

CASCADE (Em Cascata) - Quando o campo da chave primária é atualizado, registros na tabela referenciada são atualizados.

Exemplo: Funciona: Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2. Ele vai tentar atualizar o código para 5 e vai atualizar esta chave também na tabela pedidos.

SET NULL (atribuir NULL) - Quando um registro na chave primária é atualizado, todos os campos dos registros referenciados a este são setados para NULL.

Exemplo: UPDATE clientes SET codigo = 9 WHERE codigo = 5;
Na clientes o codigo vai para 5 e em pedidos, todos os campos cod_cliente com valor 5 serão setados para NULL.

SET DEFAULT (assumir o Default) - Quando um registro na chave primária é atualizado, todos os campos nos registros relacionados são setados para seu valor DEFAULT.

Exemplo: se o valor default do codigo de clientes é 999, então

UPDATE clientes SET codigo = 10 WHERE codigo = 2. Após esta consulta o campo código com valor 2 em clientes vai para 999 e também todos os campos cod_cliente em pedidos.

ON DELETE parametros:

NO ACTION (RESTRICT) - Quando um campo de chave primária está para ser deletado, a exclusão será abortada caso o valor de um registro na tabela referenciada seja mais velho. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO em DELETE FROM clientes WHERE codigo = 2. Não funcionará caso o cod_cliente em pedidos contenha um valor mais antigo que codigo em clientes.

CASCADE - Quando um registro com a chave primária é excluído, todos os registros relacionados com aquela chave são excluídos.

SET NULL - Quando um registro com a chave primária é excluído, os respectivos campos

na tabela relacionada são setados para NULL.

SET DEFAULT - Quando um registro com a chave primária é excluído, os campos respectivos da tabela relacionada são setados para seu valor DEFAULT.

Excluindo Tabelas Relacionadas

Para excluir tabelas relacionadas, antes devemos excluir a tabela com chave estrangeira. Tudo isso está na documentação sobre CREATE TABLE:

<http://www.postgresql.org/docs/8.0/interactive/sql-createtable.html>

ALTER TABLE

<http://www.postgresql.org/docs/8.0/interactive/sql-altertable.html>

Chave Primária Composta (dois campos)

```
CREATE TABLE tabela (  
  codigo INTEGER,  
  data DATE,  
  nome VARCHAR(40),  
  PRIMARY KEY (codigo, data)  
);
```

Integridade Referencial

Luiz Paulo de O. Santos

Os bancos de dados relacionais disponibilizam facilidades aos desenvolvedores que os tornam mais produtivos, mais confiáveis, rápidos e consistentes no acesso às informações em tabelas.

Uma dessas facilidades é o recurso de Integridade Referencial (IR).

Quem desenvolveu aplicativos DOS em Clipper deve lembrar bem de como era complicada e lenta a instrução SET RELATION, que “criava” um relacionamento lógico entre tabelas.

No mundo relacional, a integridade referencial feita através de chaves cuida para que os dados estejam sempre consistentes, evitando, por exemplo, o aparecimento de registros órfãos.

Vejamos os tipos de chave disponíveis:

Chave Primária

Utilizada para identificar um único registro em uma tabela. A chave primária impede que tenhamos mais de um registro com valores iguais nos campos definidos por ela, ou seja, se tornarmos um campo chave primária, não poderemos ter mais de um registro com o mesmo valor nessa coluna.

Por exemplo: Em um cadastro de pessoas, seria interessante tornar o campo CPF como chave primária, pois não queremos ter uma mesma pessoa cadastrada duas vezes. NOME é um exemplo de campo que não deve ser utilizado como chave primária, pois podemos ter diversos homônimos (pessoas diferentes com mesmo nome). Além dessa verificação, a definição de uma chave primária cria automaticamente um índice para os campos envolvidos, o que pode gerar ganho de performance para determinados tipos de consulta nessa tabela.

Supondo que temos a seguinte tabela em um banco de dados:

```
CODIGO NOME CIDADE SALARIO
A0001 José da Silva Americana R$ 2.450,00
A0010 Bárbara Alves Piracicaba R$ 8.310,00
B1201 Manoela Tavares São Paulo R$ 1.890,00
```

Na tabela acima, a chave primária é a coluna CODIGO, logo, em situação alguma podemos incluir outro código A0001, A0010 ou B1201. O banco irá recusar toda entrada duplicada.

Chave Estrangeira

É um tipo de chave criada basicamente para definir um relacionamento entre registros de tabelas diferente, chamado também de relacionamento pai-filho, ou mestre-detalle, ou mesmo mestre-escravo. No caso, para cada ocorrência do registro da tabela mestre (pai), poderá existir nenhum, um ou mais de um registro relacionados na tabela detalhe (filha), definidos pela chave estrangeira. Diferente da chave primária, a chave estrangeira aceita duplicação de conteúdo nas colunas envolvidas. Assim como na chave primária, a definição de uma chave estrangeira acarreta na criação de um índice composto pelos campos definidos pela chave, podendo haver ganho de performance em determinadas pesquisas, devido à existência do índice.

Cuidado! - Alguns SGBDs permitem o uso da cláusula UNIQUE no momento da definição de uma chave estrangeira, o que impedirá duplicidade na chave, fazendo com que somente relacionamentos de 1 para 1 sejam possíveis. Seguindo as formas normais (teoria da normalização de tabelas), num caso como esse deveríamos aglutinar essas duas tabelas em uma única tabela.

Supondo que temos a seguinte tabela de vendas, abaixo:

```
NUMVENDA CODPESSOA DATA VALOR
1 A0001 01/09/2005 R$ 120,00
2 A0001 02/09/2005 R$ 30,00
3 A0010 10/09/2005 R$ 210,00
4 B0010 20/09/2005 R$ 230,00
```

Observe que a chave primária da tabela acima é a coluna NUMVENDA, definida por um tipo auto-incremento, onde os valores são gerados automaticamente conforme registros forem sendo inseridos. A

coluna em vermelho é uma chave estrangeira, onde claramente podemos observar duplicidade, ou “n” ocorrências da chave no decorrer da tabela. No exemplo, a chave estrangeira estaria ligando o campo CODPESSOA da tabela de vendas ao campo CODIGO de uma outra tabela de PESSOAS, através de uma relação de 1 (PESSOAS) para “n” (VENDAS).

No caso de alterações na tabela “pai”, as tabelas relacionadas sofrerão algum tipo de ação para garantir a consistência do relacionamento.

As ações sempre partem da tabela “pai” para a(s) tabela(s) filha(s), sendo elas:

1. Nenhuma ação.

Nessa situação, quando a tabela “pai” (1) sofrer alterações ou exclusões, nenhuma ação automática será realizada em relação às tabelas “filhas”. Se você não fizer um tratamento manual para tornar os dados consistentes, um erro será retornado, pois estaria gerando registros órfãos.

2. Cascade

Nesse caso, sempre que a tabela pai sofre exclusões ou alterações, essas mudanças são repassadas à(s) tabela(s) filha(s). Se o campo relacionado da tabela “pai” foi alterado, automaticamente o campo relacionado nas tabelas filhas será alterado para o mesmo valor, sem a necessidade de implementação de nenhum código adicional. Se algum registro da tabela “pai” for excluído, os registros relacionados a ele nas tabelas “filhas” também serão removidos.

3. Atribuir NULL

Caso a tabela “pai” sofra alterações ou exclusões, os campos relacionados da chave estrangeira nas tabelas filhas assumirão o valor NULL, ficando “jogados” na tabela, sem estarem relacionados com qualquer registro “pai”. Vale lembrar que NULL não é valor, e sim um estado indefinido. Veja mais a frente o item 5 de “Dicas de otimização de chaves”.

4. Assumir o default

Nesse caso, com a alteração ou exclusão dos registros “pai”, os campos relacionados nas tabelas filhas assumirão um valor default.

Observação - *Tanto as chaves primárias como as estrangeiras baseiam-se em índices, ou seja, sua implementação torna-se impossível sem o uso destes. Índices são estruturas organizadas de forma crescente ou decrescente, e que permitem encontrar registros em uma tabela rapidamente, sem precisar percorrê-los um a um. Além disso, os índices podem ajudar na ordenação do resultado das pesquisas (selects), poupando trabalho para o SGBD.*

A maioria dos SGBDs atuais possui um *engine* de pré-processamento de instruções SQL (otimizador), que analisa o código SQL e otimiza sua execução, verificando se é interessante ou não utilizar determinados índices para a realização da consulta. Em algumas situações específicas, o otimizador determina que uma varredura seqüencial dos registros é mais rápida do que a utilização de um índice. Alguns SGBDs permitem que o desenvolvedor pré-determine o “plano” de consulta utilizado para uma determinada pesquisa, pulando assim a etapa de otimização.

Dicas de otimização de chaves

É notório e do conhecimento de qualquer desenvolvedor que um projeto mal-feito geralmente gera re-trabalho, e no pior dos casos, precisa ser refeito completamente. Logo, deve-se pensar no que fazer (e como fazer), antes de começar a implementar um sistema. Abaixo citarei algumas dicas que podem acelerar o desenvolvimento de um projeto:

1 Use chaves com campos pequenos

Principalmente na **chave primária**, que tem uma importância muito grande nos bancos de dados relacionais. Quanto menor for o tamanho do campo, mais rápido será o acesso à chave.

2 Evite campos com tamanhos variáveis

Campos de tamanho fixo são mais indicados para serem chaves primárias. Na maioria dos bancos de dados, o ganho de performance no uso de campos de tamanho fixo é significativo e sensivelmente notado pelos usuários.

3 Evitar excesso de índices

A indexação acelera as buscas, porém, o excesso de índices definidos em tabelas que sofrem muitas inserções (como tabelas de *logs* ou de históricos diversos), exclusões ou mesmo alterações do conteúdo de campos que são chaves, poderá gerar perda de performance, pois a estrutura de todos os índices será atualizada a cada alteração sofrida na tabela. Portanto, não saia por aí criando índices a torto e a direito. Estude caso a caso onde um índice é aconselhável ou não.

4 Evitar, sempre que possível, uso de chaves compostas

Sempre que empregamos chaves compostas (dois ou mais campos), o banco necessita de mais tempo de CPU para processar essas informações, principalmente se os campos forem de conteúdo variável, como proposto no item 2. Logo, apesar de ser um recurso bastante interessante e prático para o desenvolvedor, deve ser usado com certo cuidado, principalmente em tabelas onde o conteúdo dos campos que compõe a chave sofre alterações constantemente, e necessitam ser atualizados em *cascade*.

5 Atribuir NOT NULL para campos que serão empregados como chaves

Deve-se atribuir NOT NULL para todos os campos que serão empregados como chave. Isso é obrigatório no caso das chaves primárias.

No caso das chaves estrangeiras, tome cuidado principalmente se tiver utilizando a ação mencionada no item 4 mais acima (Atribuir NULL).

Definindo chaves em SQL

As instruções para criação de chaves primárias e secundárias são básicas em todos os bancos de dados, sofrendo pequenas mudanças de sintaxe de banco para banco. A seguir citaremos como proceder para sua criação, utilizando uma sintaxe tradicional:

Chaves Primárias:

As chaves primárias são criadas com a instrução:

```
PRIMARY KEY (campo, [...])
```

Exemplo:

```
CREATE TABLE TAB01 (  
  col_1 integer not null,  
  col_2 integer not null,  
  col_3 varchar(2000),  
  col_4 varchar(500),  
  PRIMARY KEY (col_1, col_2)  
);
```

A chave primária pode ser montada no instante da criação da tabela (CREATE TABLE) ou incluída depois, com a tabela já existente, através do comando ALTER TABLE.

Chaves estrangeiras

As chaves estrangeiras são criadas com a instrução:

```
FOREIGN KEY (campo, [...])  
REFERENCES <tabela> (campo, [...])
```

Exemplo:

```
CREATE TABLE TAB02 (  
  col_a integer,  
  col_b integer,  
  col_c varchar(1000),  
  FOREIGN KEY (col_a, col_b)  
  REFERENCES TAB01 (col_1, col_2)  
);
```

A chave estrangeira pode ser montada no instante da criação da tabela (CREATE TABLE), ou incluída depois com a tabela já existente através do ALTER TABLE.

Conclusão

Espero que esse artigo tenha deixado claro para o leitor a importância da integridade referencial, bem como da sua correta implementação.

Um complemento ao assunto abordado nesse artigo seriam as formas de normalização, mas deixaremos isso para uma outra oportunidade.

Autor - Luiz Paulo de Oliveira Santos

http://www.metodosistemas.com.br/db/DBFreeMagazine_008.pdf

Um bom artigo sobre projeto de software:

Programação (I) - Planejamento e Otimização, de Edvaldo Silva de Almeida Júnior no Viva o Linux: <http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=8057>

Tabelas são mais conceituais e menos físicas que arquivos.

Simplificando:

- DDL e DCL é para o DBA
- DML é para o programador

SGBDs

SGBDR – SGBD Relacional - tabelas são formadas por linhas e colunas (bidimensional)

SGBDOR - Combina os modelos OO e Relacional

SGBDOO - Modelo OO puro

Características de SGBDs:

- Controle de redundância
- Compartilhamento de dados
- Controle de acesso
- Esquematização (relacionamentos armazenados no banco)
- Backups

Objetivo do SGBD - armazenar objetos de forma a tornar ágil e segura a manipulação das informações.

MER - Modelo Entidades e Relacionamentos - é o modelo composto de entidades e relacionamentos

DER - Diagrama Entidades e Relacionamentos - é o resultado da fase inicial do projeto (análise)

Exemplos de Entidades:

- Físicas ou jurídicas - pessoas, funcionários, clientes, vendedores, fornecedores, empresas, filiais
- Documentos - ordens de compra, ordens de serviço, pedidos, notas fiscais
- Tabelas - CEPs, UF, cargos, etc

Tupla (termo para a fase de projeto)

Registro (termo para a fase de implementação)

Fase de Análise

Ao definir uma entidade pergunte:

- Há informação relevante sobre esta entidade para a empresa?

Faça um diagrama da entidade com alguns atributos

Relacionamentos

A B
1 ----- N

Perguntar se A tem 1 ou + B

Perguntar se B tem somente 1 A

Relacionamentos N - N devem ser divididos em dois 1 para N:

1 ----- N e N ----- 1

Referência: Livro Instant SQL Programming - Joe Celko

Independência de Banco de Dados

Publicado Fevereiro 5, 2008 [Engenharia de software](#) , [Todos 0 Comentários](#)

Tags: [arquitetura](#), [banco de dados](#), [camada](#), [DAO](#), [engenharia](#), [hibernate](#), [independência](#), [negócio](#), [padrão de projeto](#), [persistência](#), [procedures](#), [software](#)

Um requisito não-funcional importante na engenharia de software é a escolha do repositório de dados da aplicação. A princípio, isso não constitui um grande problema. Afinal, durante a construção do sistema, é suficiente se adaptar a infra-estrutura já existente no estabelecimento que irá usar o sistema. Se não houver um parque de informática definido, é comum escolher um banco de dados gratuito que normalmente atende a maioria das situações ou, mesmo, comprar licenças de uso de algum banco de dados comercial.

O problema começa na manutenção do sistema. A empresa decide, por exemplo, mudar o banco de dados padrão. Isso pode acontecer por vários motivos: contenção de custos, limitações do banco de dados atual, mudança da plataforma, etc. Aí pode ser um transtorno para os desenvolvedores: migrar os dados e modificar o sistema para se adaptar à nova situação. Sem falar nas mudanças e reinstalações que poderão ocorrer nas máquinas dos usuários. E se forem muitos usuários?! E se for pra ontem?! E se o pessoal pra fazer essas mudanças for limitado?! Veja que o desgaste gerado por esta “mudança” pode crescer de maneira exponencial. Eu já passei por isso. Posso garantir que os problemas envolvidos podem exigir várias noites de pesquisas e aprendizado instantâneo para poder continuar merecendo a confiança dos seus superiores. Fui obrigado a mudar duas vezes de plataforma de desenvolvimento em menos de dois anos. Hoje cheguei a conclusão que só existem duas plataformas que se adaptam bem a quase todos os cenários no mundo da micro-informática: .NET ou Java.

Pensando nisso decidi postar algumas considerações visando atingir o tão sonhado grau de **independência de banco de dados**.

Vamos analisar a seguinte situação:



Fig 01: Acesso direto a base

Note o seguinte:

1. A aplicação desktop faz acesso direto a base de dados. Todas as regras de negócio e persistência de dados são feitas diretamente pela aplicação. Essa arquitetura é muito utilizada pelos programadores Delphi, VB e similares.
2. Numa aplicação que utiliza servidor WWW, o browser faz requisições ao servidor que por sua vez acessa a base diretamente, da mesma forma que uma aplicação desktop. Os sites no servidor web contêm regras de negócio e são responsáveis pelo acesso aos dados. Isso é comum para programadores ASP, JSP, PHP, ColdFusion e similares.

Uma maneira de atingir a independência de banco neste contexto é utilizar somente instruções SQL ANSI para persistir dados. Mesmo assim isso possui alguns inconvenientes. Numa aplicação desktop que está instalada em várias máquinas, qualquer mudança na regra de negócio implicará em reinstalação em todas as máquinas que fazem uso dessa aplicação. Se forem muitos usuários? Se os usuários não sabem instalar sistema? Certamente o help desk vai ter trabalho dobrado pra implantar a nova versão. Utilizando servidor WWW isso ameniza e muito o problema. Basta atualizar o servidor e tudo bem. Todos os usuários terão a nova versão no seu browser. Mas a depender de como essas páginas foram criadas os programadores terão que atualizar a regra ou as configurações de banco em cada página do site. Sem falar que nem sempre é possível desenvolver com instruções SQL ANSI apenas. Uma função de banco, uma VIEW, procedure ou trigger precisa ser reescrita no momento que o banco for trocado.

Quando eu utilizava o Delphi ou o PHP optei pela seguinte solução: Minhas aplicações serviam apenas como GUI. As regras de negócio e a persistência de dados era responsabilidade das procedures de banco. Usava também usuários de banco, functions, roles e views. Triggers não utilizava por serem um tipo de procedures.

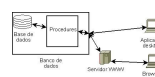


Fig 02: Acesso a base por procedure

Quando tinha que mudar alguma regra, apenas fazia nas procedures. Mudanças na GUI eram mínimas. Assim não precisava reinstalar as aplicações Delphi toda vez que mudasse alguma regra da aplicação. No PHP, da mesma forma. Só mudava a página se houvesse extrema necessidade. Era ótimo a princípio. Mas... E a independência de banco??? Assim a situação só piora o problema. Ganha de um jeito, na manutenção da regra e help desk, mas perde na independência de banco, ou seja, teria que reescrever todas procedures caso mudasse o banco. Conclusão: abandonei completamente o uso das procedures, views, roles, functions, usuários de banco e tudo mais que dependa diretamente do banco de dados. Minhas aplicações tiveram que ser reescritas completamente em outra arquitetura.

Uma solução padrão para o problema da independência de banco é utilizar um padrão de projeto chamado DAO. Isso mesmo, PADRÃO DE PROJETO. Padrão DAO faz parte de um conjunto de padrões que resolvem problemas recorrentes. Mas isso é outra história...



Fig 03: Acesso a base por DAO

Cria-se um meio, que normalmente é uma classe da orientação a objetos, por onde a aplicação fará todas as chamadas ao banco, de forma que mudar o banco significaria apenas mudar alguns atributos dessa classe. Ou seja, a classe DAO iria se moldando de acordo com algumas informações passadas no seu construtor. É possível que alguns métodos precisem ser adaptados por causa do SQL. Mas utilizar SQL ANSI diminui bastante esta necessidade. Eu comecei a fazer isso também. O problema maior é que construir uma classe dessas não pode ser um trabalho solitário. A programação da classe cresce muito. Afinal, para que fique boa, todas as possibilidades do SQL tem que estarem previstas: inserts, updates, deletes e selects e para todos os bancos de dados envolvidos. Se não, pode-se construir uma classe DAO com menos recursos e ir aprimorando a medida que necessidades surjam.

Uma solução que eu considero muito boa é construir uma classe DAO que faça uso de algum framework de persistência.



A responsabilidade de persistir os dados fica para o framework que normalmente já vem preparado para utilizar vários bancos de dados, enquanto a classe DAO da aplicação apenas herda essas características. Eu aconselho a utilização de um framework muito bom, tanto para o Java quanto para .NET: o [Hibernate](#).

O Hibernate consegue ser independente de banco de dados devido a algumas características principais:

1. através de mapeamento de entidades do SGBD como classes da linguagem e via arquivos XML;
2. ampliando a linguagem SQL para HQL (Hibernate Query Language). Esta linguagem manipula objetos e coleções que, depois, o próprio framework converte para SQL que o SGBD entende. Essa conversão é feita utilizando os arquivos XML de mapeamento, configurações de dialeto do banco e drivers de comunicação. Também pode-se utilizar outras formas de montagem do SQL, por exemplo, utilizando métodos de classes. Esses métodos recebem parâmetros que montam o SQL também utilizando os arquivos XML de mapeamento, configurações de dialeto do banco e drivers de comunicação;

Parece complicado mas não é. Inclusive tudo isso é muito produtivo. Desenvolver aplicações sobre essa ótica aumenta bastante a produtividade. Linhas de cada tabela do banco tornam-se objetos e as colunas, propriedades desses objetos. O resultado dos selects do SQL é visto como coleções de objetos e os DML's são vistos como métodos das classes de regra de negócio. É responsabilidade do framework converter tudo isso para o SQL que o banco entenda, e essa conversão deve ser de acordo com o dialeto do banco escolhido e definido em algum arquivo de configuração.

Dessa forma mudar de SGBD significa migrar os dados de um banco pra outro, mudar a configuração de dialeto e alterar, se for o caso, os arquivos XML de mapeamento das entidades. Posso garantir que poucas modificações são necessárias para migrar um sistema do SQL Server para Oracle, se voce utilizar um framework de persistência como Hibernate durante o desenvolvimento. Se a migração for, por exemplo, do Firebird para Oracle ou vice-versa as modificações se resumem a uma ou duas linhas no arquivo de configuração, além, claro, da migração dos dados. Bom, é isso. Espero que este pequeno artigo seja útil e sirva como roteiro pra quem deseja programar com independência de banco.

Link do original: <http://reginaldojr.wordpress.com/tag/arquitetura/>