Modelagem de Bancos de Dados

Normalização
Modelo Relacional
Fases de um Projeto de Banco de Dados
Modelo Relacional
Integridade Referencial
Lidando com Nulos em SQL
Tipos e Domínios
Normalizando uma tabela de CEPs
Instalando a Linguagem Procedural PL/PHP no PostgreSQL
Modelando Bancos de Dados
Expressões Regulares para uso em Modelagem de Bancos de Dados

Algumas Demonstrações sobre Normalização

Valor default
Chaves naturais x artificiais
Null

Default
create table nula(c1 serial primary key, c2 int, c3 int default 0);
insert into nula (c1) values (default),(default),(default),(default);
select * from nula;
c1 | c2 | c3
----+----+----
1 | | 0
2 | | 0
3 | | 0
4 | | 0
(4 registros)

Veja só que "riqueza" de registros! Tudo isso graças a permissão de nulo e ao valor default.

Nulo
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido. Importante: use not null
insert into nula2(c1,c2,c3) values (2,-3,4)
select * from nula2;
c1 | c2 | c3
----+----+----
1 | | 4
(1 registro)

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso.

Chave artificial
create table artificial(c1 serial primary key, t1 text, t2 text);
insert into artificial(t1,t2) values ('a','b'),('a','b'),('a','b'),('a','b'),('a','b'),('a','b');
select * from artificial;
c1 | t1 | t2
----+----+----
1 | a | b
2 | a | b
3 | a | b
4 | a | b
5 | a | b
6 | a | b
(6 registros)

Este ganha dos demais, em minha opinião. O cara cria uma chave tipo ID, que ela é a única coisa que não pode ser duplicada.
Então veja que todos os registros estão duplicados, pois a responsabilidade do SGBD é apenas a de não duplicar o campo c1.

Sugestão: para tabelas secundárias (daquelas com código e descrição), sempre usar a restrição UNIQUE no campo descrição.
Para tabelas primárias sempre que possível use chaves naturais.

Normalizando Tabelas com Leandro Dutra

- Atributos Multivalorados (telefones, municipio, uf, etc) indicam necessidade de criação de outra tabela.
- A repetição de valores de atributos também deve ser evitada criando-se uma outra tabela.
- A presença de nulos também é resolvida com a normalização e conseqüente criação de outras tabelas.
- O modelo relacional foi fundamentado na teoria dos conjuntos e na lógica dos predicados.
Se fosse buscar por inspiração no sentido mais exato, diria que era na crise de software: facilitar o desenvolvimento de grandes bases de dados usadas simultaneamente por muitos usuários.
- Seus termos principais são: relações, atributos, restrições e tipos!
- Relacionamento não é um termo técnico deste modelo, mas do MER, aqui temos as restrições de integridade referencial.
- A linguagem SQL não é inteiramente relacional. SGBD relacionais restringem o modelo para usarem essa linguagem.
Por isso mesmo não os chamo de SGBDRs, mas de SGBDs SQL. Os SGBDRs que conheço são o IBM BS/12, o Ingres QUEL original, o Alphora Dataphor e outros atualmente em desenvolvimento, anteriormente listados.
- É bom distinguir MR (modelo relacional) de MER (modelo entidade relacionamento). Este último surgiu depois do relacional, mas a grande maioria dos SGBDs atuais implementam o modelo relacional ou uma versão adaptada dele.

Em princípio, todas as chaves naturais são boas, e todas as artificiais são ruins.

O caso é que tem muita gente, principalmente usuários de ORMs, que não gosta de chaves compostas. Acha que fica difícil programar. Eu nunca usei ORM, nunca senti na pele, e daí vem minha impressão de que muitos ORMs criam tantos problemas quanto resolvem.

Imagine uma relação pai com uma chave natural composta razoável, digamos mais de três atributos; e uma relação filha com muito mais tuplas, digamos milhões ou dezenas de milhões a mais, isso rodando num sistema que é gargalo de desempenho. Nesse caso, pode ser que, após testes, valha a pena uma chave primária artificial para emagrecer a tabela filha.

(Trechos de respostas na lista pgbr-geral, por Leandro Dutra)

Modelo Relacional segundo C. J. Date

- Um modelo de dados é uma definição abstrata, lógica, dos objetos, operadores (funções) e demais.
- Objetos são usados para modelar a estrutura de dados
- Operadores são usados para modelar o comportamento dos dados
- A implementação de um modelo é a realização física na máquina real.
- Diagramas E/R não representam tudo que interessa dos relacionamentos entre as entidades.
- Uma relação que contém nulos não é uma relação.
- Um modelo relacional com nulos não é um modelo relacional.

- Nulos são um erro e nunca deveriam ter sido adotados.
- Toda função é um operador mas nem todo operador é uma função.
- Relações - Atributos - Tipos

Correspondência de termos entre os modelos lógico e físico*:*
- Relações - Tabelas
- Tupla - Registros
- Atributos - Campos ou colunas
- Modelo - Implementação

As quatro Operações das Relações:
- Relações são normalizadas
- Atributos são desordenados, da esquerda para a direita
- Tuplas são desordenadas, de cima para baixo
- Não existem tuplas duplicadas

Dicionário de dados - dados sobre dados - metadados (catálogo do sistema no PostgreSQL).

Esta última subentende chave natural em todas as tabelas. Mas se os SGBDs implementarem assim, como vamos vazer nossos testes e demonstrações? (nota)

- Valores de atributos são valores simples, mas esses valores podem ser absolutamente qualquer coisa. Nós rejeitamos a antiga noção de "valor atômico".

Ou seja, justificando tipos como geométricos, arrays, etc. (nota)

- A cardinalidade de um conjunto é seu número de elementos e de uma relação é seu número de tuplas.

- O projeto de um banco de dados tem mais de arte que de ciência.

DER
Cuidado com diagramas, use-os apenas para ajudar no processo do projeto, lembrando que um DER não faz todo o trabalho de modelagem e nem representa tudo de um modelo.

(CJ Date em seu livro: An Introduction To Database Systems)

Fases de um Projeto de Banco de Dados

Projeto de Banco de Dados Relacional

Objetivo - gerar um banco de dados que permita armazenar informações sem redundância e recuperá-las com facilidade.

Fases de um Projeto

Na modelagem de um banco de dados devemos primeiramente ter conhecimento do ambiente real ao qual será aplicado o modelo. Durante o processo de modelagem deverá ser usado o bom senso e as regras inerentes ao ambiente de aplicação do banco de dados.

1) Levantamento de Requisitos

Um modelo de dados de alto nível oferece ao p+rojetista conceitos que o possibilitam especificar as necessidades dos usuários e como o banco será estruturado para atender plenamente todas as necessidades. Aqui são importantes as entrevistas e a avaliação do projetista.

Resultado dessa fase - Especificação das necessidades dos usuários (levantamento de requisitos).

2) Projeto Conceitual

A fase conceitual depende muito da habilidade do projetista e das qualidades do modelo de dados adotado.

Escolha do modelo de dados, para com ele transcrever as necessidades e informações coletadas para um esquema de banco de dados.

O projeto conceitual indicará as necessidades funcionais da empresa, as consultas, exclusões, etc. Então deve-se rever o esquema dos dados para adequar às necessidades funcionais.

O projeto conceitual gera o esquema conceitual. No projeto conceitual não se leva em conta o SGBD que será utilizado.

O propósito do projeto conceitual é descrever o conteúdo de informação do banco de dados ao invés das estruturas de armazenamento.

3) Projeto Lógico

Tem por objetivo avaliar o esquema conceitual frente às necessidades de uso do banco de dados pelos usuários e aplicações, realizando possíveis refinamentos com a finalidade de melhorar o desempenho das operações.

Um esquema lógico é uma descrição da estrutura do banco de dados que pode ser processada por um SGBD.

Depende do modelo de dados adotado pelo SGBD, mas não especificamente do SGBD.

Neste mapeamos o modelo conceitual para o modelo de implementação (físico).

O projeto lógico gera o esquema lógico.

4) Projeto Físico

Este toma por base o esquema lógico para gerar o esquema físico e é direcionado para um específico SGBD.

O projeto físico gera o esquema físico.

Fases práticas:

- Análise de requisitos
- Identificação das relações e atributos
- Identificar as chaves das relações
- Analisar as pendências anteriores e aprofundar a modelagem
- Focar nos atributos, seus tipos, domínios e constraints. Substituir multivalorados, repetidos e nulos
- Gerar relacionamentos

Um Bom Projeto de Banco de Dados Evita:

- Inconsistência e redundância
- Dificuldade de acesso pela falta de planejamento
- Isolamento de dados
- Problemas de integridade

- Problema na falta de atomicidade nas transações
- Anomalias no acesso concorrente
- Problemas de segurança
- Operações entre disco e memória (minimizar)

Para normalizar as informações precisamos colher informações detalhadas sobre a empresa real para a qual iremos modelar o banco de dados.

O que evitar?

- informações repetidas
- dificuldade na recuperação de informações

Repetições

- desperdiçam espaço
- dificultam (engessam) atualizações

Exemplo:

Temos um cadastro de clientes assim:
(nome, fone, numero, rua, bairro, cidade, uf).
Com uma grande quantidade de registros, caso sofra alteração em algum dos campos multivalorados, como uf, teremos que atualizar todos os registros dos clientes.
Caso normalizemos a tabela de clientes e ufs sejam uma tabela separada apenas se relacionando com clientes, ao alterar uma uf apenas atualizaremos um registro na tabela ufs, sem contar que não cadastraremos a uf em cada cliente, mas apenas uma vez na tabela ufs.

Decomposição

Quando decompomos uma relação em várias, devemos ter cuidado para que não haja perda de informações nas dependências.

Modelo Relacional

Modelo Relacional (MR)
- Consiste de uma coleção de relações (tabelas no modelo físico), cada uma com um nome único (por esquema).
- Cada relação é composta por atributos (campos, no modelo físico).
- Cada atributo tem seu tipo ou domínio.
- Temos também as constraints (restrições). Relacionamentos são formados por constraints. No caso chamado de integridade referencial.
- Uma relação é formada por um conjunto de tuplas (registros no modelo físico).
- No modelo relacional uma relação tanto representa os dados quanto as relações entre eles.

NULOS - acarretam sérias dificuldades e devem ser evitados.

Chaves:

- **primária** – Uma chave composta por um ou mais campos e que não repete em nenhum registro. Formata internamente pela constraint UNIQUE e a NOT NULL.
  - estrangeira

Uma Chave Estrangeira é uma Chave de Relacionamento ou seja ela tem como propósito representar os Relacionamentos entre Tabelas num BDR.

A chave estrangeira de uma tabela pode ser definida como um conjunto de atributos da tabela (um ou mais atributos) com a propriedade de estabelecer relacionamento entre linhas de tabelas.

Uma chave estrangeira corresponde sempre a uma chave primária previamente definida em alguma tabela.

Em outras palavras um valor de chave estrangeira sempre aponta para um valor de chave primária previamente existente no banco de dados. Essa restrição é denominada de “Restrição de Integridade Referencial”.

Uma chave estrangeira pode conter valor nulo.

Chave Estrangeira é o mecanismo através do qual o Modelo de Dados Relacional implementa relacionamentos entre tabelas. Não constitui um atributo da entidade. Só aparece no momento do projeto físico.
- **candidata** – Chave criada com a adição da constraint UNIQUE em um campo.
- **super** – uma chave que contém mais campos que o necessário para garantir que seja PK.
- **artificial** – uma chave formada por um campo que nada reperesenta para a tabela, como um ID, registro, codigo, etc.
- **natural** – uma chave formada por um ou mais campos que de fato representam todos os registros de uma tabela.

Uma chave é uma propriedade do conjunto de entidades e não de uma entidade específica.

Um documento com boas informações sobre modelagem e normalização:

Modelagem e Administração de Dados em PostgreSQL
Fundamentos e práticas em bases de dados livres

De Leandro Guimarães Faria Corcete DUTRA (para a Conferência PostgreSQL Brasil)
Em: http://leandro.gfc.dutra.googlepages.com/adpg.art.pdf

Integridade Referencial

Garante que um mesmo valor apareça em duas relações.
Tradução livre do documentação "CBT Integrity Referential":
http://techdocs.postgresql.org/college/002_referentialintegrity/
Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere à informações em outra tabela e o banco de dados reforça a integridade.

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 refere ao 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 (campoa, campob)
REFERENCES tabela1 (campoa, 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.

PK 1 -----> N FK

Do lado 1 é exigida uma PK ou uma constraint UNIQUE.

Lado 1 não permite nulos.
Lado N permite nulos mas se existir a integridade garantida.

PK (Chave Primária) - é formada internamente por UNIQUE e NOT NULL
UNIQUE (Chave candidata) - permite nulos
FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constrint em consequência em consequência violada a integridade.

Recomendação: sempre usar NOT NULL nos campos da FK.

Lidando com Nulos em SQL

Como a presença de NULLs geralmente torna mais frágil um modelo de dados, como também ferem os princípios do modelo relacional e é um assunto pouco debatido, resolvi estudar e experimentar o uso do NULL em várias consultas para conhecer melhor seu comportamento. Este não é um assunto exclusivo do SGBD PostgreSQL, mas que diz respeito a todos os SGBDs que adotam SQL.

Expondo e analisando diversos cenários onde o NULL aparece e pode complicar a nossa vida: em foreign keys, agregações, distinct, etc. Como também mostrando alternativas ao seu uso. O conhecimento do comportamento do NULL nos leva a viver melhor com e sem ele.

O NULL é como a SQL lida com valores inexistentes, desconhecidos, não aplicáveis ou perdidos. NULL é global em termos de tipos de dados e não se restringe a um único tipo de dados.

Valor Padrão
Quando criamos uma tabela e em um determinado campo não adicionamos nenhuma constraint, por padrão o SQL dos principais SGBDs (inclusive o PostgreSQL) adiciona NULL como default.
De forma que quando se insere registro nesta tabela e não se adiciona nenhum valor neste campo, será inserido o valor NULL. Geralmente ao usar assim na inclusão:

```
create table nulos(chave serial primary key, nulo int);
INSERT INTO nulos (chave, nulo) VALUES (1, 1);
INSERT INTO nulos (chave, nulo) VALUES (2, NULL);
INSERT INTO nulos (chave, nulo) VALUES (3, DEFAULT);
INSERT INTO nulos (nulo) VALUES (DEFAULT);
```

Em todos estes exemplos será adicionado NULL no campo nulo.

NULL se Propaga
Regra geral: NULL se propaga, o que significa que com quem NULL se combina o resultado será um NULL, apenas com uma exceção, para FALSE.

O Que NULL não é:

- NULL não é zero,
- Não é string vazia
- Nem string de comprimento zero.
- Não é zero
- Não é vazio

Para evitar confusão recomenda-se usar um flag char(1).
Com valores como 'T' e 'F' ou 't' e 'f'.

Um exemplo
Num cadastro de alunos, para o aluno que ainda não se conhece a nota, não é correto usar zero para sua nota, mas sim NULL.

Não se pode efetuar cálculos de expressões onde um dos elementos é NULL.

COMPARANDO NULLs
NOT NULL com NULL -- Unknown
NULL com NULL – Unknown

CONVERSÃO DE/PARA NULL
NULLIF() e COALESCE()
NULLIF(valor1, valor2)

NULLIF – Retorna NULL se, e somente se, valor1 e valor2 forem iguais, caso contrário retorna valor1.

Algo como:
if (valor1 == valor2){
then NULL
else valor1;
Retorna valor1 somente quando valor1 == valor2.

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

create table nulos(nulo int, nulo2 int, nulo3 int);
insert into nulos values (1,null,null);
select coalesce(nulo, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;
select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.

GREATEST - Retorna o maior valor de uma lista - SELECT GREATEST(1,4,6,8,2); - - 8
LEAST - Retorna o menor valor de uma lista.

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.
Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs
A regra é: NULL se propaga. Qualquer que se concatene com NULL gerará NULL, cm exceção de FALSE, que gerará FALSE.
STRING || NULL -- NULL

Usos:
- Como valor default para campos que futuramente receberão valor.
- Valor default para campos que poderão ser sempre inexistentes.

Tabelas Verdade

AND

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

OR

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

NOT

<EXP>	NOT		
TRUE	FALSE		
UNKNOWN	UNKNOWN		
FALSE	TRUE		

Nessas tabelas UNKNOWN significa TRUE ou FALSE mas não se sabe qual dos dois.
Caso se remova o ONKNOWN dessas tabelas elas voltam a ser tabelas boolean normais.

Tabela verdade com operador IS

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE

FALSE	FALSE	TRUE	FASLE
UNKNOWN	FALSE	FALSE	TRUE

Experimentando:
create table boleanos(texto text, valor boolean);

insert into boleanos(texto, valor) values ('TRUE', TRUE);
insert into boleanos(texto, valor) values ('FALSE', FALSE);
insert into boleanos(texto, valor) values ('NULL', NULL);

select texto,valor from boleanos;
select * from boleanos where valor = true;
select * from boleanos where valor = false;
select * from boleanos where valor is null;
select * from boleanos where valor = true OR valor=false;
select * from boleanos where valor = true OR valor=false OR valor is null;

select * from boleanos where valor in(select valor from boleanos);
select * from boleanos where valor in(select valor from boleanos where valor=true or valor=false);
select * from boleanos where valor in(select valor from boleanos where valor=true or valor is null);
select * from boleanos where valor in(select valor from boleanos where valor=false or valor is null);
select * from boleanos where valor in(select valor from boleanos where valor is null);

select (false or false);
select (false or true);
select (true or true);
select (true or null);
select (true and null);
select (true and not null);
select (false or null);
select (false or not null);
select (false and null);
select (null or null);
select (null and null);
select (null or not null);
select (null and not null);

A comparação com NULL
resulta num estado *desconhecido* chamado UNKNOWN.

Por exemplo, a expressão SQL
"Cidade = 'Porto Alegre'"
retorna FALSE para um registro contendo "Rio de Janeiro" no campo Cidade,
mas retorna UNKNOWN para um registro contendo NULL no mesmo campo.
Usando a lógica ternária, o SQL consegue usar UNKNOWN para resolver expressões booleanas.
Considerando a expressão

"Cidade = 'Porto Alegre' OR Balanco < 0.0".

Essa expressão retorna TRUE para qualquer registro contendo um valor negativo no campo

Balanco.

A mesma expressão retorna TRUE para qualquer registro contendo "Porto Alegre" no campo Cidade.

Já FALSE é retornado somente para um registro contendo explicitamente uma cadeia diferente de "Porto Alegre" e cujo campo Balanco é explicitamente não negativo.

Em qualquer outro caso, o retorno é UNKNOWN.

Na linguagem de manipulação de dados do SQL, um retorno TRUE duma expressão inicia uma ação, enquanto UNKNOWN ou FALSE não iniciam ações.

Dessa forma a lógica ternária é transformada em binária para o utilizador.

Sobre NULLs:

- NULLs acarretam sérias dificuldades e devem ser evitados.
- Uma relação que contém nulos não é uma relação. (Date)
- Nulos são um erro e nunca deveriam ter sido adotados. (Date)
- A presença de nulls é corrigida com a normalização e conseqüente criação de outras tabelas.

FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constrint em consequência em consequência violada a integridade.

Recomendação: sempre usar NOT NULL nos campos da FK.

Comparar NULL com outro NULL

Requer operador especial, o operador IS e não podem ser comparados com os operadores =, >, <, >=, etc, pois retornará UNKNOWN.

Teste com Null

```
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido.
insert into nula2(c1,c2,c3) values (2,-3,4); -- Não válido

select * from nula2;
c1 | c2 | c3
----+----+----
1 | | 4
(1 registro)
```

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso. Veja que o campo c2, como permite null, aceitou o valor NULL, mesmo com a check > 0.

Uso de NULL em Alguns Cenários

Com Chaves Estrangeiras:

FK (para relacionamentos 1 – N ou N – 1)

```
create table clientes(cpf char(11) primary key, nome char(45)not null );
```

```
create table pedidos(
produto int primary key,
valor numeric(12,2) not null,
data date not null,
cpf_cliente char(11),
constraint cpf_fk foreign key (cpf_cliente) references clientes(cpf)
);

insert into clientes(cpf, nome) values ('12345678901', 'João Pereira Brito');

insert into pedidos(produto, valor, data, cpf_cliente) values (1, 37.45, '2008-04-26', '12345678901');
insert into pedidos(produto, valor, data, cpf_cliente) values (2, 87.45, '2008-04-27', DEFAULT);

select * from pedidos;
produto | valor | data | cpf_cliente
---------+-------+------------+-------------
1 | 37.45 | 2008-04-26 | 12345678901
2 | 87.45 | 2008-04-27 |
(2 registros)
```

Nesse caso espera-se normalmente que pedidos sejam cadastrados somente para clientes já cadastrados, mas se for permitida a FK como NULL isso não é garantido.

NOT IN

Do Artigo: NULLs vs. NOT IN()

select * from tabela where id not in (select campo from tabela2);

Mesmo com alguns ids que não encontram-se na tabela2, ainda assim nada é retornado.

CREATE TABLE objects (id INT4 PRIMARY KEY);

CREATE TABLE secondary (object_id INT4);

INSERT INTO objects (id) VALUES (1), (2), (3);

INSERT INTO secondary (object_id) VALUES (NULL), (2), (4);

SELECT * FROM objects WHERE id NOT IN (SELECT object_id FROM secondary);

SELECT id, id NOT IN (SELECT object_id FROM secondary) FROM objects;

SELECT id, id NOT IN (SELECT object_id FROM secondary WHERE object_id IS NOT NULL) FROM objects;

COUNT

O count(*) conta com NULLs.
COUNT(campo) não conta com nulls.

```
create table nulos(texto text not null, campo int);
insert into nulos(texto, campo) values ('campo1', 1);
insert into nulos(texto, campo) values ('campo2', DEFAULT);
insert into nulos(texto, campo) values ('campo3', 3);
insert into nulos(texto, campo) values ('campo4', NULL);
```

```
select count(*) from nulos
select count(texto) from nulos
select count(campo) from nulos
```

SUM

```
select * from nulos
select sum(campo) from nulos
```

AVG

```
select * from nulos
select avg(campo) from nulos
```

MAX

```
select * from nulos
select max(campo) from nulos
```

MIN

```
select * from nulos
select min(campo) from nulos
```

SUM, AVG, MAX, MIN não consideram os NULLs.

GROUP BY

```
select count(*) from nulos group by texto
select count(*),campo from nulos group by campo; -- agrupa campo 1 e 3 (nulos)
```

ORDER BY

```
select * from nulos order by campo;
select * from nulos order by campo desc;
```

Obs.: os NULL são os maiores, os primeiros quando na ordem DESC.

DISTINCT

```
select distinct(texto) from nulos;
select distinct(campo) from nulos; -- Traz inclusive os nulos, mas uma só vez
select distinct(count(campo)) from nulos; -- Como o count(campo) não traz nulos, retornará somente 2
```

NULL Se Propaga:

- Operações aritméticas com NULL gera NULL
- Operações de strings com NULL gera NULL
- Operações booleans com NULL gera NULL com TRUE e FALSE com FALSE

Exemplos de uso:

- Cadastro de funcionários, campo dependente
- Cadastro de exames laboratorias, campo do diagnóstico

Do Artigo: None, nil, Nothing, undef, NA, and SQL NULL

```
=> -- aggregate with one NULL input
=> select sum(column1) from (values(NULL::int)) t;
sum
-----

(1 row)

=> -- aggregate with two inputs, one of them NULL
=> select sum(column1) from (values(1),(NULL)) t;
sum
-----
1
(1 row)

=> -- aggregate with no input
=> select sum(column1) from (values(1),(NULL)) t where false;
sum
-----

(1 row)

=> -- + operator
=> select 1 + NULL;
?column?
----------
(1 row)
```

Boleanos

create table boleanos(entrada text, saida **boolean**);
insert into boleanos values('t','t'),('true','true'),('TRUE','TRUE'),('TRUEs/aspas',TRUE),('y','y'), ('yes','yes'), ('1','1');
insert into boleanos values('f','f'),('false','false'),('FALSE','FALSE'),('FALSEs/aspas',FALSE),('n','n'), ('no','no'), ('0','0');
Caso use o psql a consulta abaixo retornará tudo t ou f na saída. Já o pgadmin mostratá tudo TRUE ou FALSE.
select * from boleanos;

O mais interessante é perguntar ao psql se tem algum TRUE por lá e ele responder que sim:

```
testes=# select * from boleanos where saida='TRUE';
entrada | saida
-------------+-------
t | t
true | t
TRUE | t
TRUEs/aspas | t
y | t
yes | t
1 | t
(7 rows)
```

Então, agora de fato percebi que o psql (realmente, acho mais confiável e acabo confundindo com o próprio PG), o psql é só um cliente e pode exibir de uma forma, o pgadmin de outra e pelo visto qualquer uma das formas de entrada pode ser vista na saída.

Como Evitar NULLs

Campos como telefone, CPF, Inscrição Estadual e outros são criados permitindo NULL pelo fato de que esses não são obrigatórios para todos os registros.

Nesses casos, quando um cliente não tem telefone, o campo é deixado sem valor (NULL) gerando as várias dificuldades decorrentes do NULL.

Algumas providências ajudam a reduzir esses problemas. Podemos definir uma entrada para o caso do cliente não ter telefone, como por exemplo a palavra 'sem'. Todo cliente que não tenha telefone recebe a entrada 'sem'.
Com isso podemos criar um domínio que valide o campo telefone com expressões regulares.
Também podemos definie um índice parcial e único que será aplicado somente para as entradas válidas de telefone (exceto as entradas com 'sem') e usar este índice no domínio.

Assim podemos proceder para praticamente todos os casos onde haja necessidade de se deixar como opcional algum campo.

Outro exemplo Prático

Temos uma tabela de login e nela um campo expira_em para abrigar a data em que o login do usuário expira.
Acontece que alguns usuários nunca expiram, portanto para estes o campo conterá sempre NULL.

Esta é uma saída e pelo visto a mais cômoda, portando com tendência de ser a mais adotada por quem tem pressa.

Uma alternativa para evitar usar NULL seria criar uma segunda tabela, somente para os usuários que não expiram.

```
create table login_expira
(
```

```
login char(12) primary key,
senha char(32) not null,
expira_em date not null
);
create table login_nao_expira
(
login char(12) primary key,
senha char(32) not null
);
```

Criar então, para facilitar a inclusão, uma função que insere condicionalmente.

```
-- Quando não expira entrar com NULL para a data
create or replace function insere_login(text, text, date) returns void as
$$
begin
if ($3 isnull) then
insert into login_nao_expira(login, senha) values ($1, $2);
else
insert into login_expira(login, senha, expira_em) values ($1, $2, $3);
end if;
return;
end;
$$
language 'plpgsql';

-- Quando não expirar entrar NULL para data
select insere_login('ribafs', 'senha',NULL);
select insere_login('login2', 'senha2','2008-12-31');

select * from login_nao_expira;
select * from login_expira;
```

Neste caso a aplicação não cadastrará os registros diretamente nas tabelas mas indiretamente através desta função.

Referências:

- Livro - Instant SQL Programming de Joe Celko, editora WROX.
- Wikipédia – http://pt.wikipedia.org/wiki/Lógica_ternária
- Artigo - None, nil, Nothing, undef, NA, and SQL NULL
- Artigo - NULLs vs. NOT IN()

Tipos e Domínios

A criação de novos tipos e domínios reforçam e muito a robustez dos bancos de dados. Podemos criar um tipo que seja mais adequado para nosso campo e depois, criar um domínio em cima desse tipo para queseja reforçada a restrição, usando-se constraints como CHECK, NOT NULL, UNIQUE e outras. Veja exemplos.

Tipos

```
create type t_humor as enum(
'triste','normal','alegre'
);
create table humores(chave int, humor t_humor);
insert into tipos values(1, 'alegre');


create type t_sexo as enum(
'masculino','feminino'
);
create table sexos(chave int, sexo t_sexo);
insert into sexos values(1, lower('Masculino')::tsexo);
select * from sexos;
select chave, initcap(sexo::text) from sexos;
```

Domínios

```
CREATE DOMAIN dom_cep AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^\\d{8}$') NOT NULL;
-- Um domínio pode ser usado como tipo, com vantagem de ampliar as restrições
-- Veja este: numérico, tamanho 8
CREATE FUNCTION formata_cep(cep dom_cep) RETURNS TEXT AS $$
BEGIN
RETURN substr(cep,1,5) || '-' || substr(cep,6,3);
END;
$$ LANGUAGE plpgsql;

CREATE TABLE tbl_cep (cep dom_cep);
insert into tbl_cep values ('60430440');
insert into tbl_cep values (60430440);
insert into tbl_cep values ('123mjhyu');
insert into tbl_cep values ('123456789');

SELECT formata_cep(CAST('60420440' AS dom_cep)); -- retorna 60420-440

SELECT formata_cep('60420440');

CREATE DOMAIN dom_cep_nn AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^\\d{8}$') NOT NULL;
```

Exemplo na versão 8.0:

```
create type sexo as
(
m text,f text
);
```

```
CREATE DOMAIN dsexo AS text
CONSTRAINT chk_sexo CHECK (VALUE = 'feminino' OR VALUE = 'masculino');
create table tab_sexo(s dsexo, sx sexo);
insert into tab_sexo values('feminino', ('d','g'));
insert into tab_sexo values('feminino', ('d',8));
insert into tab_sexo values('masculino', ('d',8));
insert into tab_sexo values('dará erro', ('d',8));
```

Normalizando uma tabela de CEPs

Cenário atual:

Aproveitando o modelo de pessoa, vamos normalizar uma tabela de CEPs.

Tenho um arquivo CSV de ceps do tempo que os Correios distribiam gratuitamente em seu site, contendo
633.401 registros.
Agora vou usá-lo como exercício de normalização e tentar reaproveitar seus dados.

Esta tabela, ou melhor, após a normalização, serão algumas tabelas que poderão ser utilizadas num cadastro de pessoas.

su - postgres

psql

create database cep encoding 'latin1';
Latin1 é para compatibilizar com conteúdo da tabela ceps.
Pois o recomendado atualmente é a codificação UNICODE. Em um banco com codificação latin1 (iso-8859-1) tente representar por exemplo, o símbolo do euro (€).
Não consegue, pois é outra codificação, portanto sempre que possível devemos usar UTF-8.

Para comprovar crie essa tabela, num banco em latin1:
create table codificacao(c char(1))

Tente inserir este registro:
insert into codificacao values ('€')

E receberá a mensagem:
ERRO: caracter 0xe282ac da codificação "UTF8" nÃ£o tem equivalente em "LATIN1"

Tabela de CEPs original:

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

Importar dados (script em: http://pg.ribafs.net/down/scripts//cep.sql.zip)
\copy ceps from /home/ribafs/cep_brasil.csv

Adicionar PK

alter table ceps add constraint cep_pk primary key(cep);

Tabela municipios

```
create sequence municipio_seq;
create table municipios as select distinct(municipio), uf from ceps order by uf;
alter table municipios rename column municipio to descricao;
alter table municipios add column municipio int;
update municipios set municipio=nextval('municipio_seq');
alter table municipios add constraint municipio_pk primary key(municipio);
alter table municipios add constraint municipio_unk unique(descricao);
```

municipios(descricao, uf, municipio)

Tabela bairros

```
create sequence bairro_seq;
create table bairros as select distinct(bairro) from ceps order by bairro;
alter table bairros rename column bairro to descricao;
alter table bairros add column bairro int;
update bairros set bairro=nextval('bairro_seq');
alter table bairros add constraint bairro_pk primary key(bairro);
alter table bairros add constraint bairro_unk unique(descricao);
```

bairros(descricao, bairro)

Tabela logradouros

```
create sequence logradouro_seq;
create table logradouros as select distinct(logradouro) from ceps order by logradouro;
alter table logradouros rename column logradouro to descricao;
alter table logradouros add column logradouro int;
update logradouros set logradouro=nextval('logradouro_seq');
alter table logradouros add constraint logradouro_pk primary key(logradouro);
alter table logradouros add constraint logradouro_unk unique(descricao);
```

logradouros(descricao, logradouro)

Tabela tipos

```
create sequence tipo_seq;
create table tipos as select distinct(tipo) from ceps order by tipo;
alter table tipos rename column tipo to descricao;
alter table tipos add column tipo int;
update tipos set tipo=nextval('tipo_seq');
alter table tipos add constraint tipo_pk primary key(tipo);
alter table tipos add constraint tipo_unk unique(descricao);
```

tipos(descricao, tipo)

Tabela ceps normalizada

```
create table cepsn
(
   cep char(8) not null,
   tipo int,
   logradouro int,
   bairro int,
   municipio int,
   primary key(cep, logradour),
   constraint tipo_fk foreign key (tipo) references tipos(tipo),
   constraint logradouro_fk foreign key (logradouro) references logradouros(logradouro),
   constraint bairro_fk foreign key (bairro) references bairros(bairro),
   constraint municipio_fk foreign key (municipio) references municipios(municipio)
);

-- Como atualmente podem existir mais de um CEP por logradouro, então CEP não pode ser a PK,
-- portanto teremos uma chave natural formada pelo CEP e pelo logradouro
```

Criar assim:

```
create table cepsn as select distinct(cep) from ceps
alter table cepsn add column tipo int;
alter table cepsn add column logradouro int;
alter table cepsn add column bairro int;
alter table cepsn add column municipio int;
alter table cepsn add constraint tipo_fk foreign key (tipo) references tipos(tipo);
alter table cepsn add constraint logradouro_fk foreign key (logradouro) references
logradouros(logradouro);
alter table cepsn add constraint bairro_fk foreign key (bairro) references bairros(bairro);
alter table cepsn add constraint municipio_fk foreign key (municipio) references
municipios(municipio);
```

Agora vamos popular a tabela cepsn com os registros da tabela ceps.
Veja que não é somente importar todos os tipos da tabela tipos para o campo tipo de cepsn.
Temos que trazer os tipos corretos de todos os 644 mil registros. Cada um com seu tipo correspondente.
Postanto não será uma tarefa simples nem direta. Exigirá um pouco de conhecimento da linguagem SQL.

Quando não temos ceerteza se a nossa consulta é coerente e que poderá demorar muito, então ajuda muito
consultar o PostgreSQL como ele faria essa consulta.
Execute a consulta com o EXPLAIN que ele vai dar uma dica, em espacial os valores do custo final e rows.

CEPs - 633401
Tipos - 189

Logradouros - 316499
Bairros - 16766
Municípios - 346

Atualizar tipo em cepsn com o valor tipo de tipos, mas correspondentes aos de ceps

Agora para receber os valores do campo tipo tenho que pensar assim:
tomar de cepsn o cep e testar se igual ao cep de ceps
Ainda pegar neste cep o valor do tipo em ceps e testar se é igual ao valor da descricao em tipos.
Então trazer o resultado. A consulta abaixo faz isso:

```
update cepsn cn set tipo = (select t.tipo from ceps c,tipos t where cn.cep = c.cep and c.tipo =
t.descricao);

update cepsn cn set logradouro = (select l.logradouro from ceps c,logradouros l where cn.cep =
c.cep and c.logradouro = l.descricao);
```

Esta demora exageradamente.

Usei apenas:

```
update cepsn cn set logradouro = (select l.logradouro from ceps c,logradouros l where cn.cep =
c.cep
   and c.logradouro = l.descricao and l.logradouro >= 305372 and l.logradouro <=305375);
```

Custo total: 10.609.291,16 Tempo: 41.419 ms

```
update cepsn cn set bairro = (select b.bairro from ceps c,bairros b where cn.cep = c.cep and c.bairro
= b.descricao);
```

Usei somente:

```
update cepsn cn set bairro = (select b.bairro from ceps c,bairros b where cn.cep = c.cep and c.bairro
= b.descricao
   and b.bairro < 9110 and b.bairro >9101);

update cepsn cn set municipio = (select m.municipio from ceps c,municipios m where cn.cep =
c.cep and c.municipio = m.descricao);
```

Usei somente:

```
update cepsn cn set municipio = (select m.municipio from ceps c,municipios m where cn.cep =
c.cep
   and c.municipio = m.descricao and m.uf='CE');
```

Obs.: só para ter uma idéia, o tempo desta consulta foi de 133.330ms e o curto total acusado pelo Explain era de 14.417.399.32 e
rows 633401 (total)

Agora, finalmente uma consulta de CEP na tabela normalizada

```
select cep, t.descricao, l.descricao, b.descricao, m.descricao, m.uf from cepsn c, tipos t,
   logradouros l, bairros b, municipios m where c.cep='60420440' and t.tipo=c.tipo and
l.logradouro=c.logradouro
   and b.bairro=c.bairro and m.municipio=c.municipio;
```

Alterei a tabela cepsn adicionando índice único nos campos: tipo, bairro, logradouro e municipio:

```
CREATE UNIQUE INDEX unq_tipo ON pepsn (tipo);
```

Consultas úteis:
select * from municipios group by uf,municipio,descricao having count(municipio) = 1 order by uf,descricao;
select count(municipio) from municipios group by uf having count(municipio)>1 and uf='CE';

Cuidado com operadores boolean:
select * from municipios where uf='SP' and uf='DF' order by descricao; -- Nada retorna
select * from municipios where uf='SP' or uf='DF' order by descricao; -- esta retorna

Instalando a Linguagem Procedural PL/PHP no PostgreSQL

Observações iniciais
Como eu trabalho com a linguagem PHP e o PostgreSQL, então fica bem mais produtivo criar funções na linguagem PHP. Esta linguagem procedural (PLPHP) para o PostgreSQL ainda não é muito popular e me deu um pouco de trabalho para instalar. Como existe muita gente que programa em PHP este tutorial poderá ser útil a outros.
Também, como precisa que compilemos o PHP juntamente com ela, sua instalação em ambiente Windows ainda não é coisa simples.

Meu ambiente:
- Linux Ubuntu 8.04
- PostgreSQL 8.3 (instalado pelos repositórios)
- plphp 1.3.3

Obs.: Não há necessidade de se instalar o Apache nem mesmo a instalação do PHP irá interferir em outro PHP que exista na máquina.
Requisitos:
- PostgreSQL-8.1 ou superior (para a versão 1.32 ou superior da plphp)
- PHP-5 ou superior
1) Instalação do PostgreSQL

Caso tenha instalado PostgreSQL pelos repositórios instale também o pacote server-dev:
sudo apt-get install postgresql-server-dev-8.3
Caso tenha instalado dos fontes deve ter os binários no path.
Obs.: Para instalar o pg_config, instale o pacote: sudo apt-get install libpq-dev

2) Instalar PHP
Fontes do PHP 5: http://www.php.net/downloads.php

Antes instale a biblioteca libxml2-dev:
sudo apt-get install libxml2-dev

Acessar o diretório onde descompactou o php, pelo terminal e executar:
sudo ./configure --prefix=/usr/local/plphp --enable-embed
sudo make
sudo make install
Obs.: Após a instalação do PHP recebemos uma mensagem:
You may want to add: /usr/local/plphp/lib/php to your php.ini include_path

3) Instalar a plphp:
Acesse o diretório onde descompactou e execute:
sudo ./configure --with-php=/usr/local/plphp
sudo make
sudo make install
Sempre que precisar usar o configure novamente, antes execute "make clean"

No caso de ter compilado o PostgreSQL use:
./configure --with-php=/usr/local/plphp --with-postgres=/path/do/postgres

4) Instalar a linguagem plphp num banco do PostgreSQL
Acesse o PostgreSQL como super-usuário.
Criar link simbólico para a biblioteca libphp5.so em $(pg_config --libdir)

sudo updatedb
locate libphp5.so (como aqui está em /usr/local/plphp/lib/libphp5.so):
sudo ln -sf /usr/local/plphp/lib/libphp5.so $(pg_config --libdir)
Para a versão versão 8.3 use:
Acesse o SGBD com o PGAdmin ou outro cliente e execute, como super-usuário:
INSERT INTO pg_pltemplate VALUES
('plphp', 't','t','plphp_call_handler', 'plphp_validator', '$libdir/plphp', NULL);

INSERT INTO pg_pltemplate VALUES
('plphpu', 'f', 'f','plphp_call_handler', 'plphp_validator', '$libdir/plphp', NULL);

Na documentação oficial a recomendação que existe é apenas para a versão 8.2.
Na versão 8.3 esta tabela de sistema passou a ter mais um campo, então adicionei.

5) Testando:
Crie um novo banco e acesse o banco criado:
create language plphp;
Então crie suas funções.
CREATE FUNCTION teste(integer) RETURNS integer AS $$
$r = 0;
for($x=0;$x<3;$x++){
$r +=$args[0];
}
return $r;
$$ STRICT LANGUAGE 'plphp';
select teste(5);
Referências:
Muito obrigado ao Agras Zeta Atino que elaborou este howto:
http://blog.santiago.zarate.net.ve/archives/1-Como-instalar-plphp-en-ubuntu-7.04-y-7.10

Site oficial - https://projects.commandprompt.com/public/plphp/wiki
http://www.commandprompt.com/community/plphp/
Documentação - https://projects.commandprompt.com/public/plphp/wiki/Documentation
Download - https://projects.commandprompt.com/public/plphp/wiki/Downloads
Tutorial - http://www.google.com.br/url?sa=t&ct=res&cd=1&url=http%3A%2F%2Fpeople.planetpostgresql.org%2Fxzilla%2Fuploads%2Fplphp-public.odp&ei=SZN-SKboEYi28ASS8vGnDQ&usg=AFQjCNG7WFmnIpHAfWqjEYDsrParowLfAg&sig2=fha31FEuhk9WcVJzGgy8vw

Na versão 8.2 ao tentar criar a linguagem recebo o erro:
ERROR: could not access file "$libdir/plphp": Arquivo ou diretório inexistente
Sugestões serão bem-vindas, tanto para corrigir a ortografia quanto o conteúdo.

Modelando Bancos de Dados

Estou iniciando uma coleção de modelos free, que oferecem os diagramas e também os scripts de alguns bancos de dados comuns. Pois ao efetuar uma boa busca na Internet não consegui encontrar algo parecido. Encontrei um ótimo site, com bons diagramas (http://www.databaseanswers.org/data_models/index.htm)
e agora estou com a intenção de gerar os scripts para alguns diagramas e em especial com foco em detalhes da nossa realidade brasileira.

Já elaborei modelos de dois bancos:
controle de estoque e vídeo locadora,
com os diagramas e respectivos scripts. Os scripts foram testados no PostgreSQL mas devem rodar em qualquer SGBD, já que usam apenas SQL padrão.

Na lista de PostgreSQL (pgbr-geral) estamos discutindo sobre os mesmos no intuito de melhorá-los. Traga sua sugestão e outros modelos que queira compartilhar. Deixe comentários na seção projetos do site:
http://pg.ribafs.net/content/view/15/34/

Torne-se um colaborador enviando artigos para o site: http://pg.ribafs.net (basta entrar em contato).

Participe da lista de discussão do PostgreSQL:
https://listas.postgresql.org.br/cgi-bin/mailman/listinfo/pgbr-geral

Elaborar um projeto, por pequeno que seja, mesmo riscando algo no papel antes de começar de fato é algo que ajuda muito quando se cria um aplicativo, quando se cria um banco de dados e mesmo em muitas outras áreas.

A modelagem de bancos de dados é uma etapa que evita retrabalho na criação de bancos de dados e juntamente com outros conhecimentos da teoria dos bancos de dados, normalização, integridade referencial e outros ajuda a criar bancos robustos.

Controle de Estoque:
http://pg.ribafs.net/down/projeto//modelos_free/controle_estoque.jpg
http://pg.ribafs.net/down/projeto//modelos_free/controle_estoque.sql

Vídeo Locadora:
http://pg.ribafs.net/down/projeto//modelos_free/video_locadora.jpg
http://pg.ribafs.net/down/projeto//modelos_free/video_locadora.sql

Expressões Regulares para uso em Modelagem de Bancos de Dados

As expressões regulares são um grande recurso para ajudar a garantir a integridade das informações, em especial no uso com domínios.

Em ciência da computação, uma expressão regular (ou o estrangeirismo regex, abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que ou serve um gerador de analisador sintático ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de expressões regulares inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.
(Wikipedia - http://pt.wikipedia.org/wiki/Express%C3%B5es_regulares)

Correspondência com o Padrão e Expressões Regulares no PostgreSQL

O PostgreSQL suporta várias formas de correspondência com o padrão (pattern matching): o tradicional operador SQL LIKE,
o mais recente operador SIMILAR TO (adicionado no SQL 1999) e as expressões regulares estilo POSIX (também implementada na unção substring).

As regex são um recurso muito útil aos DBAs. As expressões regulares oferecem força e agilidade.

LIKE

LIKE - case sensitive
ILIKE - case insensitive

Caracteres Coringa:

% - 0 ou mais caracteres
_ - 1 único caractere

NOT LIKE
~~ = LIKE
~~* = ILIKE
!~~ = NOT LIKE
!~~* = NOT ILIKE

SIMILAR TO

Semelhante ao LIKE mas usa expressões regulares do SQL.
Assim como o LIKE somente é válido quando toda a string corresponde ao padrão.

Em expressões regulares qualquer parte da string pode corresponder ao padrão.

Caracteres coringa:
% - 0 ou mais caracteres
_ - 1 único caractere

Adiciona:

| - ou
* - 0 ou mais vezes
+ - 1 ou mais vezes
() - agrupar itens
[] - similar ao POSIX

POSIX

~ - case sensitive e corresponde
~* - case insensitive e corresponde
!~ - case sensitive e não corresponde
!~* - case insensitive e não corresponde

Átomos do Padrão

(er) - expressão regular. Correspondência para a er
[caracteres] - corresponde a qualquer dos caracteres
\k - caractere não alfanumérico
\c - caractere alfanumérico
. - qualquer único caractere
x - este é o único caractere sem função, representa 'x' mesmo

[:alnum:] Caracteres alfanuméricos, o que no caso de ASCII corresponde a [A-Za-z0-9].
[:alpha:] Caracteres alfabéticos, o que no caso de ASCII corresponde a [A-Za-z].
[:blank:] Espaço e tabulação, o que no caso de ASCII corresponde a [\t].
[:cntrl:] Caracteres de controle, o que no caso de ASCII corresponde a [\x00-\x1F\x7F].
[:digit:] Dígitos, o que no caso de ASCII corresponde a [0-9]. O Perl oferece o atalho \d.
[:graph:] Caracteres visíveis, o que no caso de ASCII corresponde a [\x21-\x7E].
[:lower:] Caracteres em caixa baixa, o que no caso de ASCII corresponde a [a-z].
[:print:] Caracteres visíveis e espaços, o que no caso de ASCII corresponde a [\x20-\x7E].
[:punct:] Caracteres de pontuação, o que no caso de ASCII corresponde a [-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~].
[:space:] Caracteres de espaços em branco, o que no caso de ASCII corresponde a [\t\r\n\v\f]. O Perl oferece o atalho \s, que, entretanto, não é exatamente equivalente; diferente do \s, a classe ainda inclui um tabulador vertical, \x11 do ASCII.[4]
[:upper:] Caracteres em caixa alta, o que no caso de ASCII corresponde a [A-Z].
[:xdigit:] Dígitos hexadecimais, o que no caso de ASCII corresponde a [A-Fa-f0-9].

Expressões regulares não podem terminar com \.

Quantificadores dos caracteres do Padrão:

* - uma seqüência de 0 ou mais correspondências do átomo
+ - uma seqüência de 1 ou mais correspondências do átomo
? - uma seqüência de 0 ou 1 correspondência do átomo
{m} - uma seqüência de exatamente m correspondências do átomo
{m,} - uma seqüência de m ou mais correspondências do átomo
{m,n} - uma seqüência de m a n (inclusive) correspondências do átomo; m não pode ser maior do que n

Caracteres Delimitadores do Padrão

^ - início da string
$ - final da string

Alguns exemplos:

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
regexp_split_to_table function splits a string using a POSIX regular expression pattern as a delimiter.
The regexp_split_to_array function behaves the same as regexp_split_to_table, except that regexp_split_to_array returns its result as an array of text. It has the syntax regexp_split_to_array(string, pattern [, flags]). The parameters are the same as for regexp_split_to_table.

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog', E'\\\s+') AS foo;
  foo
--------
 the
 quick
 brown
 fox
 jumped
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\s+');
            regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumped,over,the,lazy,dog}
(1 row)
```

SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;

Example of how to escape "_" in s aimple query

```
  create table foo (str varchar(16));
  insert into foo (str) values ('abc.defghi');
  insert into foo (str) values ('abc_defghi');
```

I want to select all strings starting with abc_def

```
  select * from foo where str like E'abc\\_def%';
```

> Fernando Brombatti wrote:
>> Alguém já usou função para extrair números de uma string?
>>
>> Ex.: AB345CD234 => 345234
>
>
> lista=# select regexp_replace('AB345CD234', '[A-Z]', '','g');
>   regexp_replace
> ----------------
>   345234
> (1 row)
>

Dica na lista pgbe-geral:

Complementando a resposta do Shander:

Caso sua string possa conter outros caracteres não numéricos, além das
letras [A-Z], o uso de '[^[:digit:]]' é mais abrangente.
http://www.postgresql.org/docs/current/interactive/functions-matching.html#FUNCTIONS-POSIX-REGEXP

```
bdteste=# SELECT regexp_replace('AB3,45CD/xz234', '[^[:digit:]]', '',
'g');
 regexp_replace
----------------
 345234
```

Osvaldo

Bom tutorial no site:
http://www.oreillynet.com/pub/a/databases/2006/02/02/postgresq_regexes.html

Reprozindo aqui os exemplos do tutorial.
Vamos criar uma tabela:

```
CREATE DATABASE regex;

CREATE TABLE myrecords(record text);
```

Insira os registros:
insert into myrecords (record) values
('a'),
('ab'),
('abc'),
('123abc'),
('132abc'),
('123ABC'),
('abc123'),
('4567'),
('5678'),
('6789');

Consultas simples usam o operador ~ (til) seguido por uma string e retornam somente os que atendem ao case.
SELECT record FROM myrecords WHERE record ~ '1'; -- Retornaram todos os registros que contenham '1'.
SELECT record FROM myrecords WHERE record ~ 'a';
SELECT record FROM myrecords WHERE record ~ 'A';
SELECT record FROM myrecords WHERE record ~ '3a';

Para retornar sem olhar o case usamos ~*:
SELECT record FROM myrecords WHERE record ~* 'a';
SELECT record FROM myrecords WHERE record ~* '3a';

Agora não trazendo o que contém a string e sensível ao case !~:
SELECT record FROM myrecords WHERE record !~ '1';

Trazendo sem olhar o case e não trazendo onde tem a string !~*:
SELECT record FROM myrecords WHERE record !~* 'c';

Trazendo as strings que comecem com um certo caractere (^):
SELECT record FROM myrecords WHERE record ~ '^1';
SELECT record FROM myrecords WHERE record ~ '^a';
SELECT record FROM myrecords WHERE record ~* '^a';

Terminados com ($):
SELECT record FROM myrecords WHERE record ~ 'c$';
SELECT record FROM myrecords WHERE record ~ 'bc$';
SELECT record FROM myrecords WHERE record ~* 'bc$';

Veja agora algumas consultas e analise seus resultados:
SELECT record FROM myrecords WHERE record ~ '[a]'; -- Qualquer que tenha a
SELECT record FROM myrecords WHERE record ~ '[A]'; -- Qualquer que tenha A
SELECT record FROM myrecords WHERE record ~* '[a]'; -- Qualquer que tenha a ou A
SELECT record FROM myrecords WHERE record ~ '[ac]'; -- Qualquer que tenha a ou c
SELECT record FROM myrecords WHERE record ~ '[ac7]'; -- Qualquer que tenha a ou c ou 7
SELECT record FROM myrecords WHERE record ~ '[a7A]'; -- Qualquer que tenha a ou 7 ou A
SELECT record FROM myrecords WHERE record ~* '[ac7]'; -- Qualquer que tenha a ou c ou 7 sem olhar o case

SELECT record FROM myrecords WHERE record ~ '[z]';

SELECT record FROM myrecords WHERE record ~ '[z7]';

SELECT record FROM myrecords WHERE record !~ '[4a]';

Procurar por uma faixa de valores:
SELECT record FROM myrecords WHERE record ~ '[1-4]';

Outros de faixa:
SELECT record FROM myrecords WHERE record ~ '[a-c5]';
SELECT record FROM myrecords WHERE record ~* '[a-c5]';
SELECT record FROM myrecords WHERE record ~ '[a-cA-C5-7]'; -- 3 faixas, a-c, A-C e 5-7

Correspondendo 2 ou mais caracteres:
SELECT record FROM myrecords WHERE record ~ '3[a]';
SELECT record FROM myrecords WHERE record ~ '[3][a]';
SELECT record FROM myrecords WHERE record ~ '[1-3]3[a]';

SELECT record FROM myrecords WHERE record ~ '[23][a]';
SELECT record FROM myrecords WHERE record ~ '[2-3][a]';
SELECT record FROM myrecords WHERE record ~ '[a-b][b-c]';

Nesta ordem:
SELECT record FROM myrecords WHERE record ~ '[a][c]';

Iniciando com dígitos:
SELECT record FROM myrecords WHERE record ~ '^[0-9]$';

Fazendo escolhas (ou |):
SELECT record FROM myrecords WHERE record ~ '^a|c$';

Começando com a ou 5 ou terminando com c:
SELECT record FROM myrecords WHERE record ~ '^a|c$|^5';

SELECT record FROM myrecords WHERE record ~ '[^0-9|^a-z]';

Repetindo Caracteres:
SELECT record FROM myrecords WHERE record ~ 'a*'; -- 0 ou mais

SELECT record FROM myrecords WHERE record ~ 'b+'; -- 1 ou mais

SELECT record FROM myrecords WHERE record ~ 'a?'; -- 0 ou 1

SELECT record FROM myrecords WHERE record ~ '[0-9]{3}'; -- Exatamente uma quantidade, usar {#}

SELECT record FROM myrecords WHERE record ~ '[0-9]{4,}'; -- Exatamente ou mais, usar {#,}

SELECT record FROM myrecords WHERE record ~ '[a-c0-9]{2,3}'; -- Exatamente de 2 até 3, inclusive

Exemplos com a função Substring:

CREATE TABLE log(record text);

Inserir registros:
insert into log (record) values
('a'),
('ab'),
('abc'),
('123abc'),
('132abc'),
('123ABC'),
('abc123'),
('4567'),
('5678'),
('6789');

SELECT substring(record, '[a-zA-Z0-9:.]{1,}') FROM log LIMIT 1;

SELECT date(substring(record, '[a-zA-Z]{1,}[0-9]{1,}') || ' 2005') AS "Date" FROM log LIMIT 1;

SELECT substring('Nov 3 07:37:51 localhost', '[:0-9]{2,}') AS "Time";

SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{2,}') AS "Time";

SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{3,}') AS "Time";

SELECT substring(record, '[:0-9]{3,}') AS "Time" FROM log LIMIT 1;

SELECT substring(record, 'SRC=*([.0-9]{2,})') AS "IP Address" FROM log LIMIT 1;

SELECT substring(record, 'SPT=*([.0-9]{2,})') AS "Remote Source Port"
   FROM log LIMIT 1;
SELECT substring(record, 'DPT=*([.0-9]{2,})') AS "Destination Port"
   FROM log LIMIT 1;

O SQL completo:

```
SELECT
   date(substring(record, '[a-zA-Z ]{1,}[0-9]{1,}') || ' 2005') AS "Date",
   substring(record, '[:0-9]{3,}') AS "Time",
   substring(record, 'SRC=*([.0-9]{2,})') AS "Remote IP Address",
   substring(record, 'SPT=*([.0-9]{2,})') AS "Remote Source Port",
   substring(record, 'DPT=*([.0-9]{2,})') AS "Destination Port"
FROM log;
```

Mais detalhes no tutorial e na documentação do PostgreSQL:
http://pgdocptbr.sourceforge.net/pg80/functions-matching.html
http://www.postgresql.org/docs/8.3/interactive/functions-matching.html

Ribamar FS – http://postgresql.ribafs.org