

AULA 04: SQL 2

SUMÁRIO	PÁGINA
1. Introdução	1
2. Visões	2
3. Joins	5
3.1. Inner Join	7
3.2. Left Join	10
3.3. Right Join	11
3.4. Full Join	12
4. DCL	17
5. Fechando o SQL	18
6. Qualidade do Projeto de Banco de Dados	23
6.1. Dependência Funcional	24
6.2. Formas Normais	25
7. Resumo	31
8. Exercícios Apresentados	32
9. Gabarito	37

1. INTRODUÇÃO

Saudações caros(as) amigos(as),

Hoje vamos à nossa quinta aula de **Conhecimentos de Banco de Dados**, continuando a tratar do tema SQL. É a nossa última aula no assunto Banco de Dados propriamente dito, depois partiremos para a Mineração de Dados para fechar nosso curso.

Nessa aula poderemos continuar utilizando os Bancos de Dados que estão no MySQL. Os assuntos de hoje são um pouco mais avançados, mas nos meus chutes tradicionais, acho que a ESAF pode até não cobrar a maioria dos assuntos que veremos nessa aula. Bem, se cobrar, temos que estar preparados.

Então, vamos trabalhar.

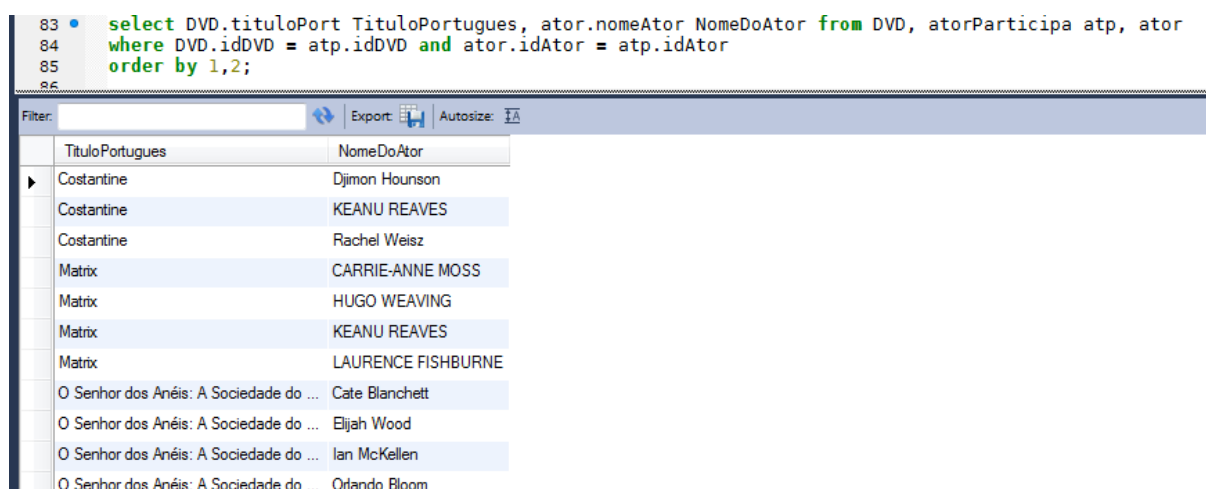
2. Visões

Vamos começar nossa aula bem leve, com um conceito muito fácil. Vamos estudar mais um objeto implementado por todos os SGBDs relacionais, que são as visões.

Uma visão é simplesmente uma query SQL, que por algum motivo, é armazenada para ser reusada. Vamos a um exemplo prático. Imaginem que eu constantemente quero uma listagem geral dos filmes e seus atores. Claro, eu não quero códigos, que o título em português dos filmes e os nomes dos atores que participam daquele filme. Essa query pode ser escrita da seguinte forma:

```
SELECT DVD.tituloPort TituloPortugues, ator.nomeAtor NomeDoAtor
FROM DVD, atorParticipa atp, ator
WHERE DVD.idDVD = atp.idDVD and ator.idAtor = atp.idAtor
ORDER BY 1,2;
```

Certo, essa query funciona, apesar de existirem formas mais elegantes de escrevê-la. Fizemos o produto cartesiano de três tabelas, e vejam que eu coloquei um apelido (ALIAS) depois da tabela atorParticipa (atp). Feito isso, bastou colocar duas condições de igualdade depois do WHERE. Vamos ver o resultado (parcial):



```
83 • select DVD.tituloPort TituloPortugues, ator.nomeAtor NomeDoAtor from DVD, atorParticipa atp, ator
84 where DVD.idDVD = atp.idDVD and ator.idAtor = atp.idAtor
85 order by 1,2;
86
```

TituloPortugues	NomeDoAtor
Costantine	Djimon Hounsou
Costantine	KEANU REAVES
Costantine	Rachel Weisz
Matrix	CARRIE-ANNE MOSS
Matrix	HUGO WEAVING
Matrix	KEANU REAVES
Matrix	LAURENCE FISHBURNE
O Senhor dos Anéis: A Sociedade do ...	Cate Blanchett
O Senhor dos Anéis: A Sociedade do ...	Elijah Wood
O Senhor dos Anéis: A Sociedade do ...	Ian McKellen
O Senhor dos Anéis: A Sociedade do ...	Orlando Bloom

Tudo bem, e o que isso tem a ver com a views? É que eu posso, a partir da query anterior, criar uma view, que nada mais é que uma consulta dinâmica guardada no Banco de Dados. Ficaria assim:

```
CREATE VIEW listaAtoresFilmes AS  
SELECT DVD.tituloPort TituloPortugues, ator.nomeAtor NomeDoAtor  
FROM DVD, atorParticipa atp, ator  
WHERE DVD.idDVD = atp.idDVD and ator.idAtor = atp.idAtor;
```

Com esse comando criei uma view, baseado na consulta acima. Vejam que bastou eu colocar o CREATE VIEW <nome da view> AS, e na sequência coloquei a query que eu tinha criado. É como se eu tivesse criado uma nova tabela, chamada listaAtoresFilmes, mas essa tabela não existe, na verdade ela é criada dinamicamente a partir de uma query. O que acontece quando eu uso essa view em um SELECT, vamos ver?

```
SELECT * FROM listaAtoresFilmes;
```

```
92 • select * from listaAtoresFilmes;  
93
```

Filter:	Export:	Autosize:
TituloPortugues	NomeDoAtor	
Costantine	Djimon Hounsou	
Costantine	KEANU REAVES	
Costantine	Rachel Weisz	
Matrix	CARRIE-ANNE MOSS	
Matrix	HUGO WEAVING	
Matrix	KEANU REAVES	
Matrix	LAURENCE FISHBURNE	
O Senhor dos Anéis: A Sociedade do Anel	Cate Blanchett	
O Senhor dos Anéis: A Sociedade do Anel	Elijah Wood	
O Senhor dos Anéis: A Sociedade do Anel	Ian McKellen	
O Senhor dos Anéis: A Sociedade do Anel	Orlando Bloom	

E já que as views funcionam como tabelas dinâmicas, posso usar as opções do SELECT que normalmente utilizo com as tabelas? Pode sim, por exemplo, posso executar essas queries:

```
SELECT COUNT(*) FROM listaatoresfilmes;
```

```
SELECT * FROM listaatoresefilmes  
WHERE nomedoator = 'HUGO WEAVING';
```

Então é isso. Com uma view, eu posso fazer qualquer SELECT que posso fazer com uma tabela. Inclusive, posso fazer o produto cartesiano de uma tabela e uma view, posso usar uma view em uma subquery e assim sucessivamente. Mas não esqueçam, quando você cria uma view esses dados não são duplicados em algum lugar da tabela. A view é apenas lógica, então ela não armazena de fato os registros, ela é apenas um SELECT gravado, por assim dizer.

Querem ver outra aplicação de uma view? Imaginem que tenhamos em uma aplicação uma tabela com todos os dados dos funcionários, inclusive os salários. Mas salário é sigiloso (menos os nossos agora). Então, como eu posso dar acesso a todos os campos dessa tabela, menos no campo salário? Bem, se eu der acesso à tabela, quem a acessar vai ver todos os campos. Uma solução seria criar uma VIEW, com todos os campos da tabela, menos o campo salário. Pronto, agora passo a VIEW para que todos acessem, e deixo a tabela salário só para aqueles que podem realmente acessar esse dado.

Com relação às views, é só isso. Vamos ver uma questão:

1. (ESAF/TRF/SRF 2006) Analise as seguintes afirmações relacionadas a Bancos de Dados e à linguagem SQL:

I. A cláusula GROUP BY do comando SELECT é utilizada para dividir colunas em conjuntos maiores de dados relacionados entre si.

II. Uma view é uma forma predeterminada de visualizar dados de uma ou mais tabelas como se fosse apenas uma tabela.

III. Quando o comando DROP TABLE é executado, as estruturas da tabela e os dados são excluídos. Porém, quando o DROP VIEW é executado, nenhum dado referenciado pela view é afetado.

IV. O trigger é um tipo de view criado quando um evento em particular ocorre.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e II**
- b) III e IV**
- c) II e III**
- d) I e III**
- e) II e IV**

Comentários:

I. O Group By é utilizado para agrupamentos de dados em determinado critério, que pode ser um campo, um conjunto deles etc. Ele não é usado para dividir colunas em grupos maiores.

II. Exatamente, através de uma View podemos visualizar dados que estão em uma ou diversas tabelas. A View é uma unidade lógica, não gera um objeto físico. Ela é criada especificando um comando Select com acesso a uma ou mais tabelas.

III. Isso mesmo, Drop Table apaga a tabela, com a estrutura e todos os seus dados. Como citei, uma view é apenas uma estrutura lógica, assim o Drop View em nada afeta a(s) tabela(s) que faz referência.

IV. A Trigger, como já vimos, não é um tipo de view, mas sim uma espécie de procedure.

Gabarito: Letra c

3. Joins

Bem, finalmente vamos encarar os Joins. Por que esse suspense todo professor, é muito difícil esse assunto? Não, não é que seja difícil, é

que às vezes os joins confundem um pouco. Eles são uma ferramenta importante para substituir a operação de produto cartesiano. Mas então, por que substituir os produtos cartesianos mesmo?

Bem, apesar de a operação produto cartesiano ser poderosa e importante, ela tem um grande defeito: custa muito caro. Quando eu faço esses produtos cartesianos em tabelas como as nossas, todos com menos de 20 tuplas, o SGBD processa isso rapidinho. Lembrem que, por exemplo, fazer um produto cartesiano de uma tabela com 15 tuplas com outra tabela com 8 tuplas, vai gerar um tabelão de $15 \times 8 = 120$ tuplas, pois a operação produto cartesiano combina cada tupla de uma tabela com todas as tuplas de outra tabela. Naquela view que acabamos de criar, o produto cartesiano gera a quantidade de tuplas da multiplicação de três tabelas.

Contudo, em aplicações reais, as tabelas são grandes, à vezes gigantescas. Imaginem fazer um produto cartesiano de uma tabela contendo 120 mil notas fiscais, com outra tabela contendo 800 mil itens dessas notas fiscais. O SGBD vai tentar gerar um tabelão com $120.000 \times 800.000 = 96.000.000.000$. Nada mal, não acham? Bem, espero ter convencido vocês que o produto cartesiano para tabelas grandes é inviável. É como arrumar uma namorada top model. Deve ser uma maravilha, não é mesmo? Mas como é que faz para pagar carros, viagens, casamento no castelo da França, jantares etc etc? Só jogador de futebol mesmo (minhas caras alunas, me desculpem o machismo do exemplo!).

Para resolver esse problema foram projetados os joins, que são instruções em SQL usadas para obter informações de duas ou mais tabelas, baseado no relacionamento entre certas colunas dessas tabelas, em outras palavras, baseado em chaves primárias e estrangeiras.

Essa é a primeira grande diferença entre os joins e o produto cartesiano. Neste, eu posso relacionar tabelas que não têm nenhum

campo em comum (apesar disso não fazer muito sentido). Já nos joins, eu preciso ter um relacionamento entre as tabelas, senão não tem join.

Temos os seguintes tipos de joins:

- Inner Join: Retorna linhas quando existe pelo menos uma combinação em ambas as tabelas;
- Left Join: Retorna todas as linhas da tabela da esquerda, mesmo que as linhas não combinem com a tabela da direita.
- Right Join: Retorna todas as linhas da tabela da direita, mesmo que as linhas não combinem com a tabela da esquerda.
- Full join: retorna linhas quando tem uma combinação em uma das tabelas.

Vamos ver um por um, com bastante calma.

3.1. Inner Join

Esse é o join que costuma ser mais usado. Como já descrevi, retorna linhas quando existe pelo menos uma combinação em ambas as tabelas. Vamos ver um exemplo para ficar mais fácil.

Temos no esquema world, uma tabela chamada city. Essa tabela tem um campo chamado countrycode (chave estrangeira), que é o código do país, e que está relacionado com a tabela country. Para listar as cidades contidas em city, mas mostrando o nome do país ao invés do código, executo a seguinte query:

```
SELECT city.id, city.name Cidade, country.name Pais,  
city.population Populacao FROM city  
INNER JOIN country  
ON city.countrycode = country.code;
```

Primeira observação. Vejam que depois do FROM eu tenho apenas uma tabela, chamada city. Então, por ai já vemos que não se trata de produto cartesiano. Depois vem a nossa novidade, o INNER JOIN <tabela> ON <condição>. Depois do INNER JOIN colocamos a tabela que queremos fazer a ligação, no caso, a tabela country. Depois, colocamos a condição para que a tupla faça parte do resultado. Neste caso, estamos casando a chave estrangeira de city (city.countrycode) com a chave primária de country (country.code). Assim, na minha projeção mostro o nome do país (country.name) ao invés do código.

```
3 • select city.id, city.name Cidade, country.Name Pais, city.population Populacao
4 from city
5 inner join country
6 on city.countrycode = country.code;
```

id	Cidade	Pais	Populacao
1	Kabul	Afghanistan	1780000
2	Qandahar	Afghanistan	237500
3	Herat	Afghanistan	186800
4	Mazare-Sharif	Afghanistan	127800
5	Amsterdam	Netherlands	731200
6	Rotterdam	Netherlands	593321
7	Haag	Netherlands	440900
8	Utrecht	Netherlands	234323
9	Eindhoven	Netherlands	201843
10	Tilburg	Netherlands	193238
11	Groningen	Netherlands	172701
12	Breda	Netherlands	160398
13	Apeldoorn	Netherlands	153491

Segunda observação. Na tabela city, todas as tuplas têm valores no campo countrycode (afinal, toda cidade fica em um país). Mas se eu tivesse, por qualquer motivo, uma tupla que esse código fosse NULL, o que aconteceria? Bem, simplesmente essa tupla não seria retornada pela minha query, uma vez que ela não combina com a tabela do INNER JOIN. Querem ver isso na prática? Tudo bem, nosso curso também é feito para os céticos.

Vou pegar a primeira cidade que aparece, limpar o campo countrycode e executar novamente minha query. Executo a seguinte sequência de comandos:


```
ALTER TABLE city CHANGE COLUMN `CountryCode` `CountryCode`  
CHAR(3) NULL DEFAULT '' ;  
UPDATE city SET countrycode = NULL WHERE id = 1;  
SELECT city.id, city.name Cidade, country.name Pais,  
city.population Populacao FROM city  
INNER JOIN country  
ON city.countrycode = country.code;
```

O primeiro comando faz com que a coluna contrycode que era NOT NULL passe a aceitar nulos. Essa sintaxe que está aí só funciona no MySQL, portanto não liguem para ela, só executem, se desejarem. O segundo comando atualiza o campo countrycode para NULL, do registro com id = 1. O terceiro comando é o mesmo lá de cima. Vamos comparar os resultados.

```
3 • select city.id, city.name Cidade, country.Name Pais, city.population Populacao  
4 from city  
5 inner join country  
6 on city.countrycode = country.code;  
7
```

Filter:	id	Cidade	Pais	Populacao
	2	Qandahar	Afghanistan	237500
	3	Herat	Afghanistan	186800
	4	Mazar-e-Sharif	Afghanistan	127800
	5	Amsterdam	Netherlands	731200
	6	Rotterdam	Netherlands	593321
	7	Haag	Netherlands	440900
	8	Utrecht	Netherlands	234323
	9	Eindhoven	Netherlands	201843
	10	Tilburg	Netherlands	193238
	11	Groningen	Netherlands	172701
	12	Breda	Netherlands	160398
	13	Apeldoorn	Netherlands	153491
	14	Nijmegen	Netherlands	152463

Como pode ser visto, a cidade com id 1, que é a primeira que aparece na consulta anterior (Kabul), agora não aparece. Isso acontece pelo que expliquei, o INNER JOIN só retorna tuplas onde existe a correspondência entre as chaves comparadas (o que vem depois do ON). Como a tupla com o id 1 passou a ter nulo no countrycode, ela fica de fora da query.

3.2. Left Join

Esse join é também muito utilizado, menos que o INNER, mas acho que fica em segundo lugar. Como já descrevi, retorna todas as linhas da tabela da esquerda, mesmo que as linhas não combinem com a tabela da direita.

Mas professor, que estória é esse de esquerda e direita? É simples, a tabela da esquerda é a que vem depois do FROM, e a da direita é que vem depois do JOIN. Vamos reescrever a última consulta que formulamos, agora com o LEFT JOIN. Fica assim:

```
SELECT city.id, city.name Cidade, country.name Pais,  
city.population Populacao FROM city  
LEFT JOIN country  
ON city.countrycode = country.code;
```

Então, já imaginam o que acontecerá agora? Aquela tupla, que tem o valor NULL em countrycode, vai ser retornada ou não? Pela explicação, vocês podem ver que todas as tuplas da tabela da esquerda (city) serão retornadas, mesmo que não encontrem correspondência na tabela da direita. Vamos ver como ficou o resultado:

```
23 • select city.id, city.name Cidade, country.Name Pais, city.population Populacao  
24 from city  
25 left join country  
26 on city.countrycode = country.code;  
27
```

id	Cidade	Pais	Populacao
1	Kabul	NULL	1780000
2	Qandahar	Afghanistan	237500
3	Herat	Afghanistan	186800
4	Mazare-Sharif	Afghanistan	127800
5	Amsterdam	Netherlands	731200
6	Rotterdam	Netherlands	593321
7	Haag	Netherlands	440900
8	Utrecht	Netherlands	234323
9	Eindhoven	Netherlands	201843

Observem que Kabul apareceu novamente, mas agora com o valor NULL na coluna País, já que a cidade não está relacionada com nenhuma tupla da tabela country.

3.3. Right Join

Vocês que são muito espertos já devem ter entendido agora o RIGHT JOIN, não é mesmo? Bem, se o LEFT JOIN retorna todas as tuplas da tabela da esquerda, mesmo que não tenha correlação, o RIGHT JOIN retorna as tuplas da tabela da direita.

Para testar o RIGHT JOIN vou inserir um registro na tabela Diretor do nosso esquema, depois vou fazer uma query que retorne todos os diretores dos filmes que tenho cadastrado, e até diretores que não tenham nenhum filme cadastrado. Vamos ver como fazer isso:

```
INSERT INTO DIRETOR (nomeDiretor) VALUES ('LEONARDO LIMA');  
SELECT DP.IDDVD, DIR.NOMEDIRETOR  
FROM DIRETORPARTICIPA DP  
RIGHT JOIN DIRETOR DIR ON  
DP.IDDIRETOR = DIR.IDDIRETOR;
```

Pronto, primeiro me auto nomeei diretor (até parece, não dirijo nem minha vida ☺). Depois fiz a query com o RIGHT JOIN, de forma que todos os diretores sejam mostrados, mesmo aquele que não tem nenhum filme. Vamos ver o resultado:

```
43 • SELECT DP.IDDVD, DIR.NOMEDIRETOR FROM DIRETORPARTICIPA DP
44 RIGHT JOIN DIRETOR DIR ON
45 DP.IDDIRETOR = DIR.IDDIRETOR;
```

Filter:	Export:	Autosize:
IDDVD	NOMEDIRETOR	
1	Andy Wachowski	
1	Larry Wachowski	
2	Francis Lawrence	
3	Peter Jackson	
4	James McTeigue	
5	George Lucas	
NULL	LEONARDO LIMA	

Meu nome até aparece na lista de diretores, mas sem estar relacionado com qualquer filme. Se eu fizer essa mesma query, mas mudando de RIGHT para LEFT, meu nome desaparece.

Caros, estou mostrando as aplicações mais básicas dos joins. Assim como quase tudo em SQL, posso fazer mil combinações. Posso colocar joins nas subqueries, posso ter joins aninhados e por ai vai. Acredito sinceramente que a ESAF não vai ter a cara de pau de cobrar de vocês um comando join extremamente complexo, pois nem nas provas específicas para TI vi questões que explorassem isso de forma mais avançada. Mas vamos lá, eu seleciono a matéria e depois fico sem dormir preocupado com vocês. Já pensou se tudo que eu disse que não vai acontecer acaba virando realidade. Vocês vão querer minha caveira!!!

3.4. Full Join

O FULL JOIN é a aplicação do RIGHT JOIN e do LEFT JOIN ao mesmo tempo, ou seja, se tiver tuplas na tabela da esquerda que não tenha na tabela da direita, é mostrada mesmo assim, e a mesma coisa com a tabela da direita.

Infelizmente o MySQL não implementa o FULL JOIN. Que peninha! Mas e aí, vamos ficar por isso mesmo? Claro que não, quem não tem FULL JOIN caça com UNION. Vou aproveitar para mostrar o operador UNION para vocês. UNION é mais uma operação da álgebra relacional, a união. Mas que união é essa professor? É aquela boa e velha União de dois conjuntos, da matemática. Então, para fazer um FULL JOIN, vou pegar o resultado de um LEFT JOIN e unir ao resultado de um RIGHT JOIN. Como vocês devem lembrar que a tia Maricota ensinou, na união de dois conjuntos os elementos comuns não são repetidos.

Para fazer o teste, primeiro vamos bagunçar mais um pouco a tabela city. Tem um país lá, a Albânia, que só tem uma cidade cadastrada, e essa cidade tem o id 34. Então, vamos deixar a Albânia sem nenhuma cidade;

```
UPDATE CITY SET COUNTRYCODE = NULL WHERE ID = 34;
```

Certo, agora eu tenho cidades sem países, e um país sem cidade. Fazendo o FULL JOIN, eu quero que todas as cidades apareçam, mesmo que não estejam atreladas a um país, e que todos os países apareçam, mesmo sem cidades atreladas, o comando fica assim:

```
select city.id, city.name Cidade, country.Name Pais,  
CITY.COUNTRYCODE, city.population Populacao  
from city  
left join country  
on city.countrycode = country.code  
UNION  
select city.id, city.name Cidade, country.Name Pais,  
CITY.COUNTRYCODE, city.population Populacao  
from city  
right join country  
on city.countrycode = country.code  
ORDER BY 3;
```

Vejam que os campos nos dois SELECT são os mesmos e estão na mesma ordem. Isso é essencial para a operação de união funcionar. Vamos ver o resultado:

```
55 • select city.id, city.name Cidade, country.Name Pais, CITY.COUNTRYCODE, city.population Populacao
56 from city
57 left join country
58 on city.countrycode = country.code
59 union
60 select city.id, city.name Cidade, country.Name Pais, CITY.COUNTRYCODE, city.population Populacao
61 from city
62 right join country
63 on city.countrycode = country.code
64 ORDER BY 3;
65
```

	id	Cidade	Pais	COUNTRYCODE	Populacao
▶	1	Kabul	NULL	NULL	1780000
	34	Tirana	NULL	NULL	270000
	4	Mazare-Sharif	Afghanistan	AFG	127800
	2	Gandahar	Afghanistan	AFG	237500
	3	Herat	Afghanistan	AFG	186800
	NULL	NULL	Albania	NULL	NULL
	47	Tâmbessa	Algeria	DZA	112007
	42	Skikda	Algeria	DZA	128747
	37	Constantine	Algeria	DZA	443727

Pronto, observamos que Kabul e Tirana não tem país associado, mas retornam por causa do LEFT JOIN, e que Albânia não tem cidade associada, mas retorna por conta do RIGHT JOIN.

Tudo certo, mas aí alguém pode me perguntar: E se eu quiser que as linhas duplicadas não sejam eliminadas, como eu faço? Para tudo tem solução meus amigos, menos para a morte. Criaram junto com o Union uma cláusula chamada ALL. Então, se usar UNION ALL, todas as linhas são retornadas, mesmo as repetidas. Podem testar e ver o resultado.

Uma última observação sobre os joins. Existe uma palavra chave que pode ser usada com o RIGHT, o LEFT e o FULL. É a palavra chave OUTER. Assim, ao invés de escrever LEFT JOIN <nomedatabela> posso escrever LEFT OUTER JOIN <nomedatabela>. Qual a diferença prática disso professor? NENHUMA!!!! Então, é como se a palavra OUTER estivesse subentendida no LEFT e o RIGHT. Claro, o OUTER é o oposto do INNER, assim não podem ser usados juntos.

Com isso, vimos as principais aplicações dos joins. Vamos ver algumas questões:

2. (ESAF/AFC/CGU 2006) Analise as seguintes afirmações relacionadas a conceitos básicos de banco de dados e linguagem SQL.

I. Na linguagem SQL um INNER JOIN retorna todas as tuplas comuns às duas tabelas.

II. Em uma Junção entre duas tabelas a cláusula USING só poderá ser usada quando o nome do atributo for igual nas duas tabelas.

III. Na linguagem SQL um RIGHT OUTER JOIN retorna todas as tuplas que não são comuns às duas tabelas.

IV. Uma Junção é usada para compor informações complexas a partir de tabelas sem nenhum tipo de relacionamento.

Indique a opção que contenha todas as afirmações verdadeiras.

a) I e III

b) II e III

c) III e IV

d) I e II

e) II e IV

Comentários:

I. Isso mesmo, o Inner Join vai retornar as tuplas que combinam nas duas tabelas.

II. Está correto. Não tinha citado o USING, mas mostro agora. Ele pode ser usado no lugar do ON, mas somente quando as colunas

relacionadas nas duas tabelas possuem o mesmo nome. Assim, as duas queries a seguir são equivalentes:

```
SELECT DP.IDDVD, DIR.NOMEDIRETOR  
FROM DIRETORPARTICIPA DP  
left JOIN DIRETOR DIR ON  
DP.IDDIRETOR = DIR.IDDIRETOR;  
SELECT DP.IDDVD, DIR.NOMEDIRETOR  
FROM DIRETORPARTICIPA DP  
left JOIN DIRETOR DIR USING (IDDIRETOR);
```

III. Incorreto, como já sabemos o RIGHT JOIN retorna todas as linhas comuns e também as linhas da tabela da direita que não tem correspondente com a tabela da esquerda.

IV. Conforme vimos, as junções só existem em tabelas que têm relacionamento, ou seja, que compartilham chaves.

Gabarito: Letra d

3. (FCC/Analista de Arquitetura de Software/INFRAERO 2011) Considere:

I. Retorna linhas quando houver pelo menos uma correspondência entre duas tabelas.

II. Operador usado para combinar o resultado do conjunto de duas ou mais instruções SELECT.

III. Operador usado em uma cláusula WHERE para pesquisar um padrão específico em uma coluna.

I, II e III correspondem em SQL, respectivamente, a

a) SELECT, UNIQUE e BETWEEN.

b) INNER JOIN, JOIN e DISTINCT.

c) LEFT JOIN, UNIQUE e LIKE.

d) SELECT, JOIN e BETWEEN.

e) INNER JOIN, UNION e LIKE.

Comentários:

A FCC tem questões muito boas em TI, justamente por fazer muitas provas dessas disciplinas. Vamos à questão.

I. Como vimos, é o conceito do INNER JOIN.

II. Acabamos de ver também. Trata-se do UNION.

III. Padrão específico, o que é isso? Ora lembram do LIKE e do %. Quando colocamos no predicado LIKE 'banana%', estamos procurando por um padrão específico, que são strings que começam por banana.

Gabarito: Letra e

4. DCL

O Data Control Language é a parte do SQL que trata do controle de acesso aos objetos do Banco de Dados. Tem basicamente dois comandos, o GRANT e o REVOKE.

O GRANT serve para dar permissão de uma operação DML a um usuário em determinado objeto. Por exemplo, posso usar:

GRANT DELETE, INSERT, SELECT ON DBEMPRESTIMO.AMIGO TO FULANO;

Neste caso, estou concedendo o poder de executar as operações DELETE, INSERT e SELECT na tabela Amigo, para o usuário chamado FULANO.

Para tirar qualquer privilégio, usamos o REVOKE da seguinte forma:

REVOKE DELETE ON DBEMPRESTIMO.AMIGO FROM FULANO;

Agora, o usuário FULANO não pode mais deletar registros desta tabela, pode apenas fazer INSERT e SELECT.

É somente isso, bem simples mesmo.

5. Fechando o SQL

Para finalizar o SQL vou mostrar mais alguns operadores, comandos ou funções muito utilizados.

Vamos começar com o TOP. Na aula passada encontramos o país com a maior população, através do MAX(), lembram? Pois é, então imaginem que eu quero agora, na tabela Country, encontrar os três países mais populosos. Como eu faço isso? Ora, primeiramente eu sei que posso ordenar as tuplas retornadas com o ORDER BY, confere? Tudo bem, ordenando as tuplas com o ORDER BY DESC, eu coloco os países em ordem de população do maior para o menor, assim:

```
SELECT * FROM COUNTRY ORDER BY POPULATION DESC;
```

Certo, assim eu vejo quais são os países, do mais populoso ao menos populoso, mas isso ainda não resolve meu problema, porque não quero como retorno todos os países, quero apenas aquele três mais populosos. Ai que entra o TOP. Eu escreveria o TOP da seguinte forma:

```
SELECT TOP 3 * FROM COUNTRY ORDER BY POPULATION DESC;
```

Beleza, com esse comando retornam somente as 3 primeiras tuplas, ignorando as demais. Ai vocês testam no MySQL e o que acontece? Dá erro. Imediatamente vem aquele pensamento. Esse professor lá sabe o que fala. Bem, a culpa não é minha ☺. Lembram que eu disse no início do SQL que cada SGBD implementa a linguagem ao seu bel prazer? Pois é, o MySQL, tihoso que só ele, implementa o TOP diferente. Funciona da seguinte forma.

```
SELECT * FROM COUNTRY ORDER BY POPULATION DESC LIMIT 3;
```

Pronto, ai está a versão do TOP para o MySQL. Ao invés de Top, colo esse LIMIT no final. Mas qual você deve estudar? Bem, o TOP é mais conhecido, e esse LIMIT ai só funciona no MySQL. Paciência meus amigos, as coisas são assim mesmo. Para vocês terem uma idéia, no Oracle já é feito de outra forma, e olha que o MySQL é um produto da Oracle.

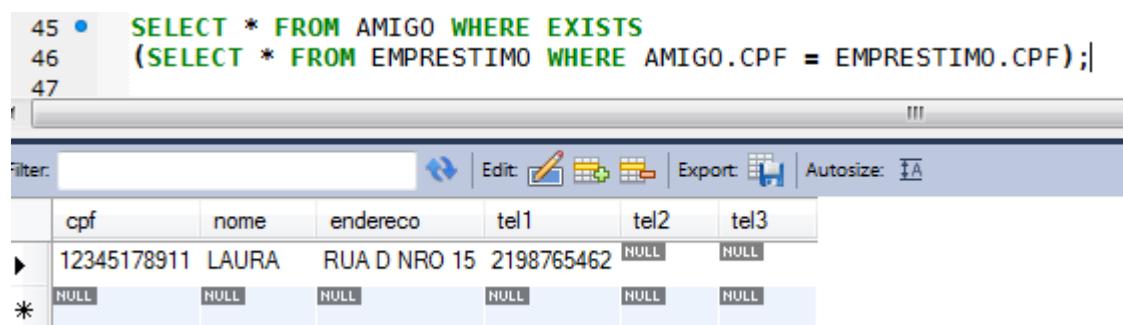
Agora vamos ao operador EXISTS. Ele é usado para testar o resultado de uma subquery. Primeiro, vou inserir uma linha na minha tabela de empréstimos para executar as queries que desejo. Uso o seguinte comando:

```
INSERT INTO EMPRESTIMO (CPF,IDDVD,DATAEMPREST,DATAPREV)
VALUES
('12345178911',1,DATE_FORMAT('2012-01-01 00:00:00' , '%y-%m-%d'), DATE_FORMAT('2012-01-10' , '%y-%m-%d'));
```

Não vou entrar em detalhes dessa função DATE_FORMAT, pois ela só serve para o MySQL. O que ela faz é transforma uma string em data. Bem, inseri esse registro e agora quero fazer as seguintes queries: quero a relação dos amigos que já emprestaram pelo menos um filme, e a relação dos amigos que nunca emprestaram filme algum. Vamos à primeira:

```
SELECT * FROM AMIGO WHERE EXISTS
(SELECT * FROM EMPRESTIMO
WHERE AMIGO.CPF = EMPRESTIMO.CPF);
```

Essa primeira query retorna as linhas de amigo, contanto que existam na tabela emprestimo. Ou seja, retorna os amigos que emprestaram pelo menos uma vez. Vamos ver o resultado:



```
45 • SELECT * FROM AMIGO WHERE EXISTS
46 (SELECT * FROM EMPRESTIMO WHERE AMIGO.CPF = EMPRESTIMO.CPF);|
47
```

cpf	nome	endereço	tel1	tel2	tel3
12345178911	LAURA	RUA D NRO 15	2198765462	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

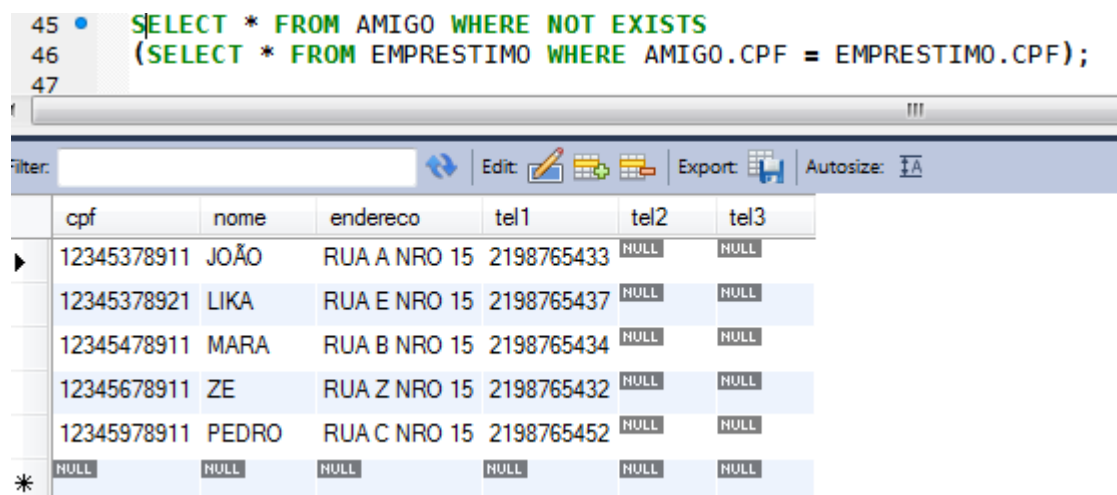
Como só a Laura tem empréstimos, só ela aparece como resultado da nossa query.

Agora eu quero os amigos que nunca emprestaram nenhum DVD. Basta eu negar o predicado anterior, da seguinte forma:

```
SELECT * FROM AMIGO WHERE NOT EXISTS  
(SELECT * FROM EMPRESTIMO  
WHERE AMIGO.CPF = EMPRESTIMO.CPF);
```

Pronto, esse NOT faz com que a query tenha o resultado contrário.

Vamos ver como ficou:



```
45 SELECT * FROM AMIGO WHERE NOT EXISTS  
46 (SELECT * FROM EMPRESTIMO WHERE AMIGO.CPF = EMPRESTIMO.CPF);  
47
```

cpf	nome	endereco	tel1	tel2	tel3
12345378911	JOÃO	RUA A NRO 15	2198765433	NULL	NULL
12345378921	LIKA	RUA E NRO 15	2198765437	NULL	NULL
12345478911	MARA	RUA B NRO 15	2198765434	NULL	NULL
12345678911	ZE	RUA Z NRO 15	2198765432	NULL	NULL
12345978911	PEDRO	RUA C NRO 15	2198765452	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

Agora aparecem todos menos a Laura. Então é isso que o EXISTS faz, ele testa se o subquery retorna pelo menos um resultado. Se retornar, o predicado é verdadeiro. Senão, é falso.

Outro operador, chamado ANY, pode substituir o EXISTS. Por exemplo, a query:

```
SELECT * FROM AMIGO WHERE EXISTS  
(SELECT * FROM EMPRESTIMO  
WHERE AMIGO.CPF = EMPRESTIMO.CPF);
```

Pode ser substituída, voltando exatamente o mesmo resultado, por:

```
SELECT * FROM AMIGO WHERE cpf = ANY  
(SELECT CPF FROM EMPRESTIMO);
```

O que essa query diz ao SGBD é o seguinte: Me retorne todas as tuplas de Amigo, contanto que o CPF esteja na tabela Empréstimo. Em outras palavras, me retorne os amigos que já efetuaram algum empréstimo. Assim como negamos o EXISTS, podemos negar o operador

igual que acompanha o ANY, achando a query que equivale ao NOT EXISTS. Qual a negação do igual? É o diferente. A query fica assim:

```
SELECT * FROM AMIGO WHERE cpf <> ANY  
(SELECT CPF FROM EMPRESTIMO);
```

Agora tenho como retorno amigos que não tem seu CPF na tabela de empréstimos, ou seja, amigo que nunca emprestaram nada.

4. (ESAF/Analista Informática/IRB 2006) Analise as seguintes afirmações relacionadas a banco de dados e ao uso da linguagem SQL.

I. A cláusula ORDER BY permite criar consultas com as linhas em uma ordem específica, modificando a ordem de apresentação do resultado da pesquisa. As linhas podem ser ordenadas de forma ascendente ou descendente de acordo com as colunas declaradas na cláusula ORDER BY.

II. Os dois comandos SELECT apresentados a seguir são equivalentes.

```
SELECT * FROM Carro  
WHERE CodCarro <> ANY (SELECT CodCarro  
FROM Vendido).
```

```
SELECT * FROM Carro  
WHERE CodCarro NOT IN (SELECT  
CodCarro FROM Vendido).
```

III. O comando SQL abaixo retornará a relação dos carros cujo código CodCarro seja igual a 51.

```
SELECT Ano, COUNT(*) AS Total de Carros  
FROM Veiculos  
GROUP BY Ano
```

HAVING CodCarro = 51

IV. A cláusula GROUP BY pode ser colocada antes ou depois da cláusula HAVING. Se ela for colocada antes, os grupos são formados e as funções de grupos são calculadas antes de resolver a cláusula HAVING. Se for colocada depois, a cláusula HAVING filtra a seleção antes da formação dos grupos e do cálculo das funções.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e II**
- b) II e III**
- c) III e IV**
- d) I e III**
- e) II e IV**

Comentários:

I. Exato, o order by pode ser usado para ordenar um ResultSet. Além disso, a ordenação pode ser por mais de uma coluna, e cada coluna especificada pode ter as cláusulas ASC e DESC, que ordenam em ordens ascendente ou descendente, respectivamente,

II. Nos dois comandos Select há uma subquery que retorna os valores de CodCarro da tabela Vendido. E ambos mostram as tuplas da tabela Carro, contando que seu código não esteja na tabela Vendido. Assim <> (diferente) ANY e NOT IN exercem nesse caso a mesma função.

III. Incorreto. Na verdade este Select deveria estar escrito da seguinte forma:

```
SELECT Ano, COUNT(*) AS Total de Carros  
FROM Veiculos
```

Where CodCarro = 51

GROUP BY Ano

Lembrem-se que o Having é usado para tratar das funções de agregação (nesse caso, o Count)

IV. O Group by é colocado **antes** do Having. Item incorreto.

Gabarito: Letra a

Amigos, aqui eu encerro o assunto SQL. Mas não estou enganando ninguém, como expliquei várias vezes, nem de longe vimos tudo. O que vimos é aquilo que é mais comum, mais usado. Acredito que muito dificilmente vocês deixam de responder uma questão de SQL com o que foi dado, nem que seja por eliminação. Vamos agora encerra nosso assunto de Banco de Dados com um último tópico.

6. Qualidade do Projeto de Banco de Dados

Esse é mais um tópico que deixei para o final porque acredito que a ESAF não deveria cobrá-lo, e explico o porquê. Apesar de pedir o assunto modelagem no edital, não acredito que a CGU queira que vocês efetivamente modelem um Banco de Dados. É mais provável que o órgão deseje que vocês entendam o que é o modelo, e como chegar nele, e já vimos isso com algum esmero.

Bem, mostrei para vocês uma metodologia que parte do entendimento do problema, depois vem a criação de um modelo conceitual, depois transformado em um modelo lógico, e finalmente culminando com um modelo físico. Também comentamos que cada um faz o modelo da forma como entendeu e interpretou as informações passadas. Mas isto significa que qualquer modelo está correto? Não necessariamente.

Pensando em critérios mínimos para que se aceite um modelo de dados como correto, foram criados parâmetros de aceitabilidade. Esses parâmetros são conhecidos como formas normais, e é isso que vamos estudar nesse tópico. Mas antes de entrar em formas normais e no processo de normalização, vamos ver o importante conceito de dependência funcional

6.1. Dependência funcional

Dependência Funcional! Pense num assunto fácil, mas que se você for pesquisar em um livro de Banco de Dados, vai ter toda uma teoria matemática por detrás, toda uma explicação bem complicadinha. Como nosso objetivo é passar em concursos, vou “dropar” toda essa teoria matemática do nosso estudo e escrever umas poucas linhas sobre dependência funcional.

Dependência funcional, em apertada síntese, é um relacionamento que existe entre atributos de uma Relação. Vamos pegar o exemplo mais simples, a partir das Relações do nosso projeto.

Ator (IdAtor, nomeAtor)

Considerando a Relação Ator, temos que nomeAtor depende funcionalmente de idAtor. O que significa isso? Ora, significa que pelo idAtor eu descubro o nomeAtor. Ou seja, idAtor determina nomeAtor, e nomeAtor depende funcionalmente de idAtor. Isso é representado da seguinte forma:

idAtor → nomeAtor

Mas professor, o contrário é verdadeiro? Não, não é. Eu posso ter dois atores com o mesmo nome, mas eles teriam idAtor diferentes. Então, pelo nomeAtor eu não determino idAtor. **Dica: Uma chave primária sempre determina os demais atributos de uma Relação.**

Mas aí vem a pergunta, somente as chaves primárias determinam outro atributo? Não, posso construir a seguinte relação:

ItemNotaFiscal (IdNota, IdItem, CodProduto, Quantidade, ValorUnitario, Total)

Bem, já sabemos que a chave primária determina os outros atributos. Mas olhem o atributo Total. O que é total? Simplesmente é Quantidade x ValorUnitario. Então, esses dois campos determinam Total, existindo assim uma dependência funcional (DF):

Quantidade,Valor → Total

Por fim, um terceiro exemplo. Imaginem que eu resolvi alterar minha Relação AtorParticipa, que terá agora o seguinte formato:

AtorParticipa (IdDVD, IdAtor, NomeAtor)

Mas professor, pode fazer a relação assim? Claro que pode, a Relação é minha e faço como quiser!!! Bem, sabemos que a chave primária determina os demais atributos, nesse caso temos que:

IdDVD,IdAtor → NomeAtor

Contudo, existe outra DF aqui, que é a seguinte:

idAtor → NomeAtor

Todo mundo concorda que idAtor determina NomeAtor? Espero que sim ☺.

Então é isso, dependência funcional sem matemática é o que acabamos de ver

6.2. Formas Normais

Agora sim vamos passar aos nossos critérios de qualidade. Eles se chamam formas normais. Aplicar as formas normais também é conhecido como processo de normalização.

A normalização de dados é uma série de passos que se segue no projeto de um banco de dados que permite um armazenamento consistente e um eficiente acesso aos dados em um banco de dados relacional. Esses passos reduzem a redundância de dados e as chances dos dados se tornarem inconsistentes.

As primeiras formas normais foram criadas na década de 1970 por um cidadão chamado Codd, e persistem até hoje. Elas são importantes para verificar se seu Banco de Dados está bem projetado. Vamos ver aqui as principais formas normais.

Um projeto de Banco de Dados está na **primeira formal normal, ou 1FN**, se todos os atributos são atômicos, ou seja, não existem atributos multivalorados ou compostos. Essa é muito fácil, pois vimos que quando mapeamos nosso MER em Modelo Relacional, todos os atributos são atômicos. Então, nosso projetinho está na primeira forma normal. Ela pode parecer meio ridícula, mas quando os SGBDs relacionais surgiram, quem dominava o mercado eram tecnologias que permitiam, por exemplo, campos multivalorados. Ai o povo acostumado com isso fazia projetos de Bancos de Dados prevendo esse tipo de campo. Contudo, não era possível usar campos multivalorados em SGBDs relacionais.

Um projeto de Banco de Dados está na **segunda formal normal, ou 2FN**, se está na 1FN e os atributos que fazem parte de sua chave primária definem, em conjunto, todos os demais atributos. Ou seja, eu não posso ter um atributo na chave primária que, sozinho, define um outro atributo não chave.

Lembram de como eu reescrevi a Relação AtorParticipa:

AtorParticipa (IdDVD, IdAtor, NomeAtor)

Essa relação não está na 2FN, porque um dos atributos que fazem parte da chave primária, de forma isolada, determina um atributo não chave. Ou seja, idAtor determina NomeAtor. E como fazer para deixar essa relação na 2FN? Simples, voltamos a como era, tirando o atributo nomeAtor dessa relação e deixando apenas na Relação Ator.

E uma relação que tem somente um campo na chave primária? Bem, se só tiver um campo na chave primária, então a relação está automaticamente em 2FN.

ATENÇÃO!!! Polêmica no ar. Alguns autores defendem que para estar em 2FN, nenhum dos atributos que fazem parte de qualquer chave **candidata**, de forma isolada, pode determinar um atributo não chave. Vejam a diferença, enquanto a primeira definição fala em chave primária, e ela vem do livro do Navathe, alguns autores falam em chaves candidatas. Isso, caros amigos, dá uma confusão daquelas, porque as próprias bancas oram cobram de um jeito, ora cobram de outro. Parece aquele problema da contabilidade das insubsistências passivas e insubsistências do passivo.

Bem, então para onde eu corro nesse mato? Se a questão pedir só o conceito, e a alternativa mais provável citar que o critério deve ser aceitos para todas as chaves candidatas, então aceita a questão. Agora se for para avaliar uma relação, e a banca explicitar as dependências funcionais, recomendo considerar a chave primária somente.

É meus amigos, vida de concursando não é fácil. Por incrível que pareça, todas as provas que fiz para concursos de TI, sem exceção, cobraram alguma coisa de formas normais. Confesso que quase sempre eu errava a questão, ou por esquecimento dos conceitos, ou mesmo por essas confusões que a bancas colocam.

Bem, entendam a 2FN como eu expliquei, e vamos rezar para a banca não vir com gracinhas.

Um projeto de Banco de Dados está na **terceira forma normal, ou 3FN**, se está na 2FN e não existe dependência transitiva. Mas o que é isso? Muito simples, trago de novo a seguinte relação:

ItemNotaFiscal (IdNota, IdItem, CodProduto, Quantidade, ValorUnitario, Total)

Bem, a chave idNota e IdItem determinam CodProduto, Quantidade e ValorUnitario. Mas Total não depende funcionalmente diretamente da chave. Esse atributo depende funcionalmente de Quantidade e ValorUnitario. Assim temos:

IdNota, IdItem → Quantidade, ValorUnitario → Total

Dessa forma, temos uma dependência transitiva. Mas trocando em miúdos, para uma relação estar em 3FN um campo não chave não pode ser determinado por outro(s) campo(s) não chave. **No caso da 3FN, fazemos essa checagem para cada chave candidata.**

Por fim, temos a **Forma Normal de Boyce-Codd, ou BCNF**. Uma Relação está em BCNF, se e somente se estiver em 3FN, e para todas as dependências funcionais que existirem na relação, o determinante (aquele que está antes da seta) for uma superchave. Olha é difícil encontrar uma relação que esteja em 3FN e que não esteja em BCNF. Um conselho, não se preocupem muito com ela, eu citei apenas imaginando que a ESAF possa cobrar o conceito, e ver se vocês pelo menos ouviram falar dela. Entendendo os conceitos de 1FN a 3FN está muito bom.

Vamos a alguns exercícios:

5. (ESAF/ATI/SUSEP 2006) Em um Banco de Dados Relacional:

a) uma relação está na 1FN (primeira forma normal) se nenhum domínio contiver valores atômicos.

b) uma Chave Primária corresponde ao Identificador único de uma determinada relação. Em uma relação pode haver mais que uma coluna candidata a chave primária.

c) as colunas que irão compor as Chaves Primárias devem ser inicializadas com valores nulos.

d) em uma tabela existirão tantas Chaves Primárias quantas forem as colunas nela existentes.

e) uma Chave Externa é formada por uma coluna de uma tabela que se referencia a uma Coluna qualquer de outra tabela. Essas colunas, na tabela destino, não aceitam valores nulos. Uma tabela destino pode ter apenas uma Chave Externa.

Comentários:

a) A 1FN (1ª forma normal) é alcançada se todos os atributos forem atômicos, ou seja, nenhum deles for multivalorado. Por exemplo, se seu modelo contempla um atributo telefones, sendo que podem ser vários, então não está na 1FN.

b) Conforme já estudamos, essa item.

c) Colunas que fazem parte de uma chave primária não podem conter o valor nulo, senão, como a chave primária vai identificar unicamente uma tupla, se tem o valor nulo?

d) Cada tabela tem uma, e apenas uma, chave primária.

e) Uma chave externa, ou estrangeira, faz referência a uma chave primária (ou candidata) em outra tabela. Normalmente aceitam valores nulos, e podem existir diversas chaves estrangeiras.

Gabarito: Letra b

6. (FCC/Analista de BD/INFRAERO 2011) Em relação à normalização de dados, considere:

I. Se existir um atributo multivalorado, deve-se criar um novo atributo que individualize a informação multivalorada.

II. Se existir um atributo não atômico, deve-se dividi-lo em outros atributos que sejam atômicos.

III. Todos os atributos primos devem depender funcionalmente de toda a chave primária.

Os itens I, II e III referem-se direta e respectivamente a

a) 1FN, 1FN e 2FN.

b) 1FN, 2FN e 2FN.

c) 1FN, 2FN e 3FN.

d) 2FN, 2FN e 3FN.

e) 2FN, 3FN e 3FN.

Comentários:

I. Trata-se da 1FN

II. Trata-se da 1FN

III. Atributos primos são aqueles que não fazem parte da chave. Dessa forma, não pode um atributo primo depender de apenas parte da chave primária. É a definição de 2FN

Gabarito: Letra a

7. (FCC/Analista Judiciário-TI/TRT19 2011) Para uma tabela estar na FNBC (Forma Normal Boyce- Codd), ela

a) não precisa da normalização 1FN.

b) precisa estar somente na 2FN.

c) também está normalizada na 3FN.

d) tem de estar normalizada até a 4FN.

e) tem de estar normalizada até a 5FN.

Comentários:

Só por isso eu falei da BCNF, para matar essas questões mais fáceis. Como vimos, para estar em BCNF deve estar na 3FN.

Gabarito: Letra c

8. (FCC/Analista Judiciário-TI/TRT23 2011) No contexto de normalização, quando a tabela não contém tabelas aninhadas e não possui colunas multivaloradas; não contém dependências parciais, embora contenha dependências transitivas, diz-se que ela está na

- a) primeira forma normal (1FN).**
- b) segunda forma normal (2FN).**
- c) terceira forma normal (3FN).**
- d) quarta forma normal (4FN).**
- e) quinta forma normal (5FN).**

Comentários:

Não possui tabelas aninhadas, ou seja, não tem atributos compostos, e não possui campos multivalorados. Até ai está pelo menos na 1FN. Não contém dependências parciais, ou seja, nenhum atributo não chave depende funcionalmente de apenas uma parte da chave primária. Então, está pelo menos na 2FN. Possui dependências transitivas. Então não está na 3FN. Ficamos então na 2FN.

Gabarito: Letra b

7. Resumo

As VIEWS são objetos do banco de dados que na verdade representam uma query que se deseja utilizar com frequência. Elas

podem ser usadas para esconder determinadas informações de usuários específicos, e não geram armazenamento no Banco de Dados.

As operações de JOIN servem para ligar duas tabelas através de campos em comum. Elas são mais baratas que a operação de produto cartesiano, e oferecem uma série de funcionalidades e combinações que podem tornar as consultas rápidas e poderosas.

O Data Control Language é a parte do SQL que trata do controle de acesso aos objetos do Banco de Dados. Tem basicamente dois comandos, o GRANT, que dá permissões nos objetos, e o REVOKE, que retira permissões nos objetos.

Para que um projeto seja considerado de boa qualidade, e não tenha redundâncias desnecessárias, devemos normalizar as relações. Existem quatro formas normais mais utilizadas, que são 1FN, 2FN, 3FN e BCNF.

8. Exercícios Apresentados

1. (ESAF/TRF/SRF 2006) Analise as seguintes afirmações relacionadas a Bancos de Dados e à linguagem SQL:

I. A cláusula GROUP BY do comando SELECT é utilizada para dividir colunas em conjuntos maiores de dados relacionados entre si.

II. Uma view é uma forma predeterminada de visualizar dados de uma ou mais tabelas como se fosse apenas uma tabela.

III. Quando o comando DROP TABLE é executado, as estruturas da tabela e os dados são excluídos. Porém, quando o DROP VIEW é executado, nenhum dado referenciado pela view é afetado.

IV. O trigger é um tipo de view criado quando um evento em particular ocorre.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e II**
- b) III e IV**
- c) II e III**
- d) I e III**
- e) II e IV**

2. (ESAF/AFC/CGU 2006) Analise as seguintes afirmações relacionadas a conceitos básicos de banco de dados e linguagem SQL.

I. Na linguagem SQL um INNER JOIN retorna todas as tuplas comuns às duas tabelas.

II. Em uma Junção entre duas tabelas a cláusula USING só poderá ser usada quando o nome do atributo for igual nas duas tabelas.

III. Na linguagem SQL um RIGHT OUTER JOIN retorna todas as tuplas que não são comuns às duas tabelas.

IV. Uma Junção é usada para compor informações complexas a partir de tabelas sem nenhum tipo de relacionamento.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e III**
- b) II e III**
- c) III e IV**

d) I e II

3. (FCC/Analista de Arquitetura de Software/INFRAERO 2011) Considere:

I. Retorna linhas quando houver pelo menos uma correspondência entre duas tabelas.

II. Operador usado para combinar o resultado do conjunto de duas ou mais instruções SELECT.

III. Operador usado em uma cláusula WHERE para pesquisar um padrão específico em uma coluna.

I, II e III correspondem em SQL, respectivamente, a

a) SELECT, UNIQUE e BETWEEN.

b) INNER JOIN, JOIN e DISTINCT.

c) LEFT JOIN, UNIQUE e LIKE.

d) SELECT, JOIN e BETWEEN.

e) INNER JOIN, UNION e LIKE

4. (ESAF/Analista Informática/IRB 2006) Analise as seguintes afirmações relacionadas a banco de dados e ao uso da linguagem SQL.

I. A cláusula ORDER BY permite criar consultas com as linhas em uma ordem específica, modificando a ordem de apresentação do resultado da pesquisa. As linhas podem ser ordenadas de forma ascendente ou descendente de acordo com as colunas declaradas na cláusula ORDER BY.

II. Os dois comandos SELECT apresentados a seguir são equivalentes.

SELECT * FROM Carro

WHERE CodCarro <> ANY (SELECT CodCarro

FROM Vendido).

SELECT * FROM Carro

WHERE CodCarro NOT IN (SELECT

CodCarro FROM Vendido).

III. O comando SQL abaixo retornará a relação dos carros cujo código CodCarro seja igual a 51.

SELECT Ano, COUNT(*) AS Total de Carros

FROM Veiculos

GROUP BY Ano

HAVING CodCarro = 51

IV. A cláusula GROUP BY pode ser colocada antes ou depois da cláusula HAVING. Se ela for colocada antes, os grupos são formados e as funções de grupos são calculadas antes de resolver a cláusula HAVING. Se for colocada depois, a cláusula HAVING filtra a seleção antes da formação dos grupos e do cálculo das funções.

Indique a opção que contenha todas as afirmações verdadeiras.

a) I e II

b) II e III

c) III e IV

d) I e III

e) II e IV

5. (ESAF/ATI/SUSEP 2006) Em um Banco de Dados Relacional:

a) uma relação está na 1FN (primeira forma normal) se nenhum domínio contiver valores atômicos.

b) uma Chave Primária corresponde ao Identificador único de uma determinada relação. Em uma relação pode haver mais que uma coluna candidata a chave primária.

c) as colunas que irão compor as Chaves Primárias devem ser inicializadas com valores nulos.

d) em uma tabela existirão tantas Chaves Primárias quantas forem as colunas nela existentes.

e) uma Chave Externa é formada por uma coluna de uma tabela que se referencia a uma Coluna qualquer de outra tabela. Essas colunas, na tabela destino, não aceitam valores nulos. Uma tabela destino pode ter apenas uma Chave Externa.

6. (FCC/Analista de BD/INFRAERO 2011) Em relação à normalização de dados, considere:

I. Se existir um atributo multivalorado, deve-se criar um novo atributo que individualize a informação multivalorada.

II. Se existir um atributo não atômico, deve-se dividi-lo em outros atributos que sejam atômicos.

III. Todos os atributos primos devem depender funcionalmente de toda a chave primária.

Os itens I, II e III referem-se direta e respectivamente a

a) 1FN, 1FN e 2FN.

b) 1FN, 2FN e 2FN.

c) 1FN, 2FN e 3FN.

d) 2FN, 2FN e 3FN.

e) 2FN, 3FN e 3FN.

7. (FCC/Analista Judiciário-TI/TRT19 2011) Para uma tabela estar na FNBC (Forma Normal Boyce- Codd), ela

- a) não precisa da normalização 1FN.**
- b) precisa estar somente na 2FN.**
- c) também está normalizada na 3FN.**
- d) tem de estar normalizada até a 4FN.**
- e) tem de estar normalizada até a 5FN.**

8. (FCC/Analista Judiciário-TI/TRT23 2011) No contexto de normalização, quando a tabela não contém tabelas aninhadas e não possui colunas multivaloradas; não contém dependências parciais, embora contenha dependências transitivas, diz-se que ela está na

- a) primeira forma normal (1FN).**
- b) segunda forma normal (2FN).**
- c) terceira forma normal (3FN).**
- d) quarta forma normal (4FN).**
- e) quinta forma normal (5FN).**

9. Gabarito

1 – c ; 2 – d ; 3 – e ; 4 – a ; 5 – b ; 6 – a ; 7 – c ; 8 – b