Modelagem de Dados Relacional-Objeto e SGBDs-OR

Profa. Dra. Sarajane Marques Peres
EACH – USP
each.uspnet.usp.br/sarajane

Introdução

- As linguagens de programação persistentes acrescentam persistência e outras características de banco de dados às linguagens de programação existentes, do tipo orientação a objeto.
- Modelos de dados relacionais-objeto estendem o modelo de dados relacional fornecendo um tipo de sistema mais rico, incluindo orientação a objetos e acrescentando estruturas especiais à linguagem de consultas relacionais, como a SQL, para tratar os tipos de dados acrescentados.
- É introduzido um sistema de tipos aninhados que permitem que os atributos de tuplas sejam de tipos complexos.
- Essas extensões tentam preservar os fundamentos relacionais – em particular, o acesso declaratório ao dado – enquanto estendem o poder da modelagem.
- Os sistemas de banco de dados relacionais-objeto fornecem um caminho de migração conveniente para usuários de bancos de dados relacionais que desejam usar características orientadas a objetos.

Relações Aninhadas

- Primeira forma normal: exige que todos os atributos de uma relação tenham domínios atômicos.
  - Um domínio é atômico se os elementos do domínio são considerados unidades indivisíveis.

- O modelo relacional aninhado é uma extensão do modelo relacional em que os domínios podem ser definidos como atômicos ou como relações. Então, o valor de uma tupla sob um atributo pode ser uma relação, e relações podem ser armazenadas dentro de relações.

- Assim, um objeto complexo pode ser representado por uma única tupla de uma relação aninhada.

Exemplo

Considere um sistema de recuperação de documentos em que armazenamos, para cada documento, as seguintes informações:

- Título do documento;
- Lista de autores;
- Data de aquisição;
- Lista de palavras-chave.

- Uma relação para esta informação conterá domínios não-atômicos:

título	lista_autores	data	lista_palavra_chave
		(dia, mês, ano)	
Plano de vendas	{Smith, Jones}	(1,abril, 89)	{lucro, estratégia}
Relatório de status	{Jones, Frick}	(17, julho, 94)	{lucro, pessoal}

Relação documento não-1NFdoc.

Exemplo

título	autor	dia	mês	ano	palavra_chave
Plano de vendas	Smith	1	abril	89	lucro
Plano de vendas	Jones	1	abril	89	lucro
Plano de vendas	Smith	1	abril	89	estratégia
Plano de vendas	Jones	1	abril	89	estratégia
Relatório de status	Jones	17	junho	94	lucro
Relatório de status	Frick	17	junho	94	lucro
Relatório de status	Jones	17	junho	94	pessoal
Relatório de status	Frick	17	junho	94	pessoal

Relação doc-flat, uma versão 1NF da relação não-1NFdoc.

Tipos complexos e orientação a objetos

- Sistemas com tipos complexos e orientação a objetos permitem que os conceitos de E-R, como atributos multivalorados, generalizações e especializações, sejam representados diretamente sem uma tradução complexa para o modelo relacional.

- XSQL – linguagem de consulta que estende a SQL com características de orientação a objetos
  - SQL3 (SQL:99): padrão que acrescentou a orientação objetos e outras características ao SQL padrão atual (SQL 92)

Suporte Objeto-Relacional

- Principais inclusões:
  - Construtores de tipos para especificar objetos complexos:
    - Tipo row (fila)
    - Tipo array
    - Set, list, bag
  - Identidade de objeto: uso do tipo referência
  - Encapsulamento de operações: é possível por mecanismos de tipos definidos pelos usuários que podem incluir operações como parte de suas declarações
  - Mecanismos de herança

Construtores de tipo

create type MinhaSeqüência **char varying**

create type MinhaData
(dia **integer**,
mês **char(10)**,
ano **integer**)

create type Documento
(nome MinhaSeqüência,
lista_autor **setof** (MinhaSeqüência),
data MinhaData,
lista_palavra_chave **setof** (MinhaSeqüência))

create table doc **of type** Documento.

Construtores de tipo (SQL:99)

create type nome_do_tipo **as** [**row**] (<declarações de componentes>)

/ a palavra chave row é opcional*

create type tipo_Endereço **as** (
rua **varchar**(45),
cidade **varchar**(25),
cep **char**(5));

create type tipo_Empregado **as** (
nome **varchar**(35),
endereco *tipo_Endereco*,
idade **integer**,);

Tipo ROW.

create type tipo_companhia **as** (
nomccomp **varchar**(20),
localizacao **varchar**(20) **array** [10]
);

Possíveis usos:
localização[1]
endereço.rua

Limitação: array de array

Construtores de tipo

- Os tipos criados usando as declarações anteriores são gravados no esquema banco de dados. Então, outras declarações que acessam o banco de dados podem fazer uso das definições desses tipos.

- Usualmente, sistemas de tipos complexos aceitam outros tipos de conjuntos, como matrizes e multiconjuntos.

  - matriz_autor MinhaSeqüência[10] : matriz de nomes de autor. Podemos determinar quem é o primeiro autor, o segundo autor e assim por diante.
  - Imprime-execuções **multiset(integer) :** poderia representar o número de cópias de um documento impresso em cada execução de impressão do documento.

Identificadores de Objetos com uso de referências

- Um atributo de um tipo pode ser uma referência a um objeto de um tipo especificado.
- Exemplo: referências a pessoas são do tipo ref(Pessoa). O campo lista_autor do tipo Documento pode ser redefinido como:

lista_autor **setof** (**ref** (Pessoa))

Notação Korth!

que é um conjunto de referências a objetos Pessoa.

- As tuplas de uma tabela também podem ter referências a elas. A chave primária da tabela pode ser usada para criar essas referências.
- Alternativamente, cada tupla de uma tabela pode ter um identificador de tupla como um atributo implícito, e uma referência a uma tupla é simplesmente o identificador de tupla.

Identificadores de Objetos com uso de referências (SQL:99)

```
create table Empregado of tipo_Empregado ref is emp_id system
    generated;
create table Companhia of tipo_Companhia (
    ref is comp_id system generated,
    primary key (nomecomp));
```

Indica o identifcador de tupla.

Gerado pelo sistema, mas poderia ser DERIVED que usa o método tradicional de chave primária especificada pelo usuário.

Indica o nome da tabela cujas tuplas podem ser referenciadas pelo atributo de referência.

```
create type tipo_Emprega as (
    empregado ref (tipo_Empregado) scope (Empregado)
    companhia ref (tipo_Companhia) scope (Companhia));
create table Emprega as tipo_Emprega;
```

Similar à chave estrangeira. Aqui se utiliza o identificador de objeto, e não a chave primária, para fazer a referência..

Encapsulamento de operações SQL

- É possível criar uma especificação comportamental para um tipo, por meio da especificação de métodos em adição a atributos.

```
create type <nome do tipo> (
    lista de atributos membros com tipos individuais
    declaração de restrições
    declaração de outras funções (métodos)
);
```

- Instanciando:

```
create type tipo_Endereço as (
    rua varchar(45)
    cidade varchar(25)
    cep char(5))
method nro_apto() returns char(8);
```

Método que extrai o número do apartamento (se houver) a partir da cadeia de caracteres que forma o atributo membro rua do tipo tipo_Endereço.

Encapsulamento de operações SQL

- A implementação para o método ainda deve ser escrita.
- É possível referir-se à implementação do método pela especificação do arquivo que contém o código para o método, como em:

method

create function nro_apto() **returns char**(8) **for** tipo_Endereço **as**
external name '/x/y/nro_apto.class' **language** 'java';

Neste caso, como a função é escrita em JAVA,
trata-se de uma função externa à SQL.
Uma função interna seria escrita em uma linguagem estendida da SQL.

Herança

- A herança pode ser no nível de tipos ou no nível de tabelas.
- Herança de tipos:

```
create type Pessoa
    ( nome MinhaSeqüência,
      seguro_social integer)

create type Estudante
    (graduação MinhaSeqüência,
    departamento MinhaSeqüência)
        under Pessoa

create type Professor
    (salário integer,
    departamento MinhaSeqüência)
      under Pessoa
```

Notação Korth!

  - Tanto estudante quanto professor herdam os atributos de Pessoa – isto é, *nome* e *seguro_social*. Estudante e Professor são chamados subtipos de Pessoa e Pessoa é um supertipo de Estudante, assim como de Professor.

Herança

- Suponha que queiramos armazenar informação sobre assistentes de ensino, que são, simultaneamente, estudantes e professores, talvez até mesmo em diferentes departamentos → herança múltipla

create type AssistenteEnsino
under Estudante, Professor

Notação Korth!

- AssistenteEnsino deve herdar todos os atributos de Estudante e Professor.

- Entretanto, existe um problema: os atributos seguro_social, nome e departamento estão presentes em Estudante, assim como em Professor.

  - Seguro_social e nome são herdados de uma fonte comum, Pessoa. Então não há conflito causado por herdá-los a partir de Estudante, assim como de Professor.
  - Departamento é definido separadamente em Estudante e Professor. De fato, um assistente de ensino pode ser um estudante de um departamento e um professor em outro departamento. Assim, deve-se usar o recurso de renomeação.

Herança

Notação Korth!

create type AssistentedeEnsino
under Estudante **with** (departamento **as** estudante_depto),
Professor **with** (departamento **as** professor_depto)

- Em muitas linguagens de programação, uma entidade deve ter exatamente um **tipo mais específico**, isto é, se uma entidade tem mais de um tipo, deve haver um único tipo ao qual a entidade pertence e, esse único tipo deve ser um subtipo de todos os tipos aos quais a entidade pertence.

- Por exemplo, suponha que uma entidade seja do tipo Pessoa, assim como do tipo Estudante. Então, o tipo mais específico da entidade é Estudante, já que Estudante é um subtipo de Pessoa.

- Entretanto, uma entidade não pode ser do tipo Estudante e do tipo Professor, a menos que haja um tipo, como AssistentedeEnsino, que é um subtipo de Professor e de Estudante.

Herança

- A fim de que cada entidade tenha, exatamente, um tipo mais específico, teríamos de criar um subtipo para cada combinação possível dos supertipos. Dentro de um contexto universitário, por exemplo, teríamos os subtipos:
  - EstudantenãoGraduadoEstrangeiro;
  - EstudanteGraduadoEstrangeiroJogadordeFutebol;
  - Etc ....
- Uma abordagem melhor no contexto de sistemas de banco de dados é permitir a um objeto ter múltiplos tipos, sem possuir um tipo mais específico.
- Sistemas relacionais-objeto podem modelar tal característica pelo uso de herança no nível de tabelas, e permitindo que uma entidade exista em mais de uma tabela de uma vez.

Herança

- Herança de tabelas:

```
create table pessoas
   (nome MinhaSeqüência,
    seguro_social integer);

create table estudantes
   (graduação_MinhaSeqüência,
    departamento_Minhaseqüência)
under pessoas

create table professores
   (salário integer,
    departamento MinhaSeqüência)
under pessoas
```

Notação Korth!

Herança

- As sub-tabelas estudantes e professores herdam todos os atributos da tabela pessoas.
- Não há necessidade de criar tabelas adicionais, como *assistentesdeensino*, que herdam tanto de estudantes quanto de professores, a menos que hajam atributos específicos para entidades que são tanto estudantes como professores.
- Requisitos:
  - Cada tupla da supertabela *pessoas* pode corresponder a no máximo uma tupla em cada uma das tabelas *estudantes e professores* (isto é, ter os mesmos valores para todos os atributos herdados).
  - Cada tupla em *estudantes e professores* deve ter, exatamente, uma tupla correspondente em *pessoas* (isto é, com os mesmos valores para todos os atributos herdados).

Herança

- Sem a primeira condição, poderíamos ter duas tuplas em *estudantes* (ou *professores)* que corresponderiam à mesma pessoa.

- Sem a segunda condição, poderíamos ter uma tupla em *estudantes* ou em *professores* que não corresponderia a qualquer tupla em *pessoa*, ou que corresponderia a várias tuplas em *pessoa*.

- As sub-tabelas podem ser armazenadas de uma maneira eficiente sem replicação de todos os campos herdados. Atributos herdados, outros que não a chave primária da supertabela, não necessitam ser armazenados e podem ser derivados por meio de uma junção com a supertabela, baseada na chave primária.

Objetos complexos: SQL:99

- BLOBs: objetos binários longos
  - gráficos, audio, vídeos ...
- CLOBs: objetos texto longos

- E no Oracle, ainda tem-se:
  - BFILE: arquivo binário armazenado à parte do banco de dados
  - NCLOB: CLOB de tamanho fixo

- No Oracle: apenas o BFILE é armazenado externamente ao banco de dados.

Consultas com Tipos Complexos

- A SQL foi estendida para manipular tipos complexos.
- Exemplo: encontre o nome e o ano de publicação de cada documento.

```
select nome, data.ano
from doc
```

Notação Korth!

- Observe que o campo *ano* do atributo composto *data* é referido pelo uso de uma notação de ponto.

SQL: Atributos Relação-Valorados

- A **SQL estendida** permite que uma expressão para uma relação apareça em qualquer lugar em que o nome da relação possa aparecer, **como em uma condição *from*.**

- A habilidade de usar subexpressões livremente torna possível tirar vantagem da estrutura de relações aninhadas, de diferentes maneiras.

- Suponha a relação **pdoc** com o seguinte esquema:

```
create table pdoc
    (nome MinhaSeqüência,
     lista_autor setof ( ref (pessoa)),
     data MinhaData,
     lista_palavra_chave setof (MinhaSeqüência))
```

Notação Korth!

SQL: Atributos Relação-Valorados

- Encontre todos os documentos que têm a palavra “banco de dados” como uma de suas palavras-chaves:

  select nome

  from pdoc

  where “banco de dados” **in** lista_palavra_chave

  Notação Korth!

  Observe que foi usado o atributo relação_valorado lista_palavra_chave em uma posição em que a SQL teria exigido uma subexpressão **select-from-where**.

SQL: Atributos Relação-Valorados

- Suponha que queiramos uma relação contendo pares da forma “nome_documento, nome_autor” para cada documento e cada autor do documento.

select B.nome, Y.nome
from pdoc **as** B, B.lista_autor **as** Y

Já que o atributo lista_autor de pdoc é um campo conjunto_valorado, ele pode ser usado em uma condição from, onde deveria ser usado uma relação.

Da mesma forma, podemos usar as funções agregadas nestes campos:

select nome, **count** (lista_autor)
from pdoc

Notação Korth!

SQL: Expressões Path

- Suponha a tabela definida como a seguir:

  create table estudantes_phd
  (orientador **ref** (pessoa))
  under pessoa

  Notação Korth!

  Encontre agora os nomes dos orientadores dos estudantes de doutorado.

  select estudantes_phd.orientador.nome
  from estudantes_phd

  - Já que *estudante_phd.orientador* é uma referência à tupla na tabela *pessoa*, o atributo *nome* na consulta é o atributo *nome* da tupla da tabela *pessoa;*
  - As referências podem ser usadas para esconder operações de junção;
  - Sem o uso de referências, o campo *orientador* de estudantes_phd seria uma chave estrangeira.

Notação de ponto: SQL:99

select e.empregado **->** nome

from Emprega **as** e

where e.companhia **->** nomecomp = 'abcde'

-> é usado para dar acesso a atributos do tipo utilizado numa referência do tipo **ref**.

Aninhar e Desaninhar

- A transformação de uma relação aninhada para uma relação na 1NF é chamada de desaninhar.
- A relação doc tem dois atributos (lista_autor e lista_palavra_chave) que são relações aninhadas.
- A conversão desta relação, com relações aninhadas, em uma relação básica pode ser feita da seguinte forma:

```
select nome, A as autor, data.dia, data.mês, data.ano, k as palavra_chave
from doc as B, B.lista_autor as A, B.lista_palavra_chave as K.
```

Notação Korth!

Aninhar e Desaninhar

- O processo inverso de transformação de uma relação 1 NF em uma relação aninhada é chamado aninhar.

select título, **set** (autor) **as** lista_autor, (dia,mês,ano) **as** data, **set** (palavra_chave) **as** lista_palavra_chave
from doc_flat
group by título, data

Notação Korth!

Funções

- Sistemas relacionais-objeto permitem que funções sejam definidas pelos usuários.

- Suponha que queiramos uma função que, dado um documento, retorne a contagem do número de autores.

```
create function total_autor(um_doc Documento)
    returns integer as
    select count (lista_autor)
   from um_doc
```

Notação Korth!

Criação de valores e Objetos complexos

- Criando uma tupla do tipo definido pela relação doc:

  (“plano de vendas”, **set** (“Smith”, “Jones”), (1, “abril”, 89), **set** (“lucro”, “estratégia”))

- Inserindo a tupla em uma relação:

Notação Korth!

insert into doc

values (“plano de vendas”, **set** (“Smith”, “Jones”), (1, “abril”, 89), **set** (“lucro”, “estratégia”))

BDOO X BDRO

- Extensões persistentes das linguagens de programação e sistemas relacionais-objetos estão direcionados a diferentes mercados.
  - A natureza declaratória e o poder limitado da linguagem SQL (comparado com uma linguagem de programação) fornecem boa proteção de dados contra erros de programação e tornam otimizações de alto nível relativamente fáceis.
  - Linguagens de programação persistentes destinam-se a aplicações que tenham altas exigências de desempenho. Elas fornecem acesso com baixo overhead ao dado persistente e eliminam a necessidade de tradução do dado se os dados tiverem de ser manipulados usando uma linguagem de programação.

BDR X BDOO X BDRO

- Sistemas relacionais: tipos de dados simples, linguagens de consulta poderosas e alta proteção.
- Linguagem de programação simples baseada em BDOOs: tipos de dados complexos, integração com linguagem de programação, alto desempenho.
- Sistemas relacionais-objeto: tipos de dados complexos, linguagens de consultas poderosa, alta proteção.

Bibliografia

- Estes slides estão baseados nos livros:

  - Elmasri, R. & Navathe, S. B.. Sistemas de Banco de Dados. São Paulo: Addison Wesley, 2005.
  - Silberschatz, A., Korth, H. F. & Sudarshan, S.. Sistemas de Banco de Dados. São Paulo: Makron Books, 2006.