UNIVERSIDADE FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA
MESTRADO EM INFORMÁTICA

Estudo da Adequação do Uso de XML para o Armazenamento de Dados Históricos

CURITIBA
2004

NILO SÉRGIO FADEL

Estudo da Adequação do Uso de XML para o Armazenamento de Dados Históricos

Dissertação de Mestrado do Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná

Orientadora: Prof.ª Dr.ª Carmem Hara

CURITIBA

2004

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno *Nilo Sérgio Fadel*, avaliamos o trabalho intitulado, *"ESTUDO DA ADEQUAÇÃO DO USO DE XML PARA O ARMAZENAMENTO DE DADOS HISTÓRICOS"*, cuja defesa foi realizada no dia 21 de dezembro de 2004, às dez horas, no Auditório do Departamento de Informática do Setor de Ciencias Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 21 de dezembro de 2004.

Profª. Dra. Carmem Satie Hara
DINF/UFPR – Orientadora

Profª. Dra. Cristina de Aguiar Dutra Ciferri
UEM – Membro Externo

Prof. Dr. Marcos Sfair Sunye
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus por ter me dado esta oportunidade e também por me dar saúde, coragem, persistência e sabedoria para poder chegar a mais esta conquista.

À professora Carmem Hara, minha orientadora, por seus ensinamentos, dedicação e apoio, meu especial muito obrigado!

Agradeço também à minha família: Denise, minha mulher, Fernanda e Rodrigo, meus filhos, que sempre me apoiaram e tiveram que conviver com minha ausência. Dedico a vocês esta conquista!

Obrigado ao meu gerente Roberto Silva e a meu *head* Fernando Vardânega, quando eu era funcionário do DBA do HSBC, pelo apoio e por terem me liberado para assistir as aulas. Sem isto não teria conseguido completar os créditos para o mestrado.

Ao professor Sunye, obrigado pelo incentivo inicial durante a pós-graduação e pelo apoio durante todo o mestrado.

Obrigado também aos demais professores do mestrado, que dedicam suas vidas à arte de ensinar.

À professora Cristina Dutra Ciferri, por ter se deslocado de Maringá para Curitiba exclusivamente para assistir a defesa da dissertação.

Agradeço também o apoio de todos os colegas do mestrado, principalmente o Valmir, o Eduardo e a Juscélia que estiveram juntos durante os momentos difíceis.

E ainda, à Jucélia, secretária da pós-graduação, pela sua sempre pronta atenção.

SUMÁRIO

Índice de Figuras

RESUMO

O XML (*eXtensible Markup Language*) vem se tornando um padrão para representação de dados na Internet. Ele surgiu da necessidade de um formato de dados que facilitasse a publicação e extração de dados neste meio.

Por outro lado, a grande maioria das empresas utiliza bancos de dados relacionais para o armazenamento dos seus dados. Os bancos de dados relacionais consolidaram-se nos últimos 20 anos como uma estrutura sofisticada, estável e de alto desempenho.

Para que as empresas possam disponibilizar seus dados na Internet existe a necessidade de transformar os dados relacionais em XML e vice-versa. Esta necessidade fez com que surgissem diversas frentes de estudo, tanto na linha de conversão de dados entre os dois modelos, como na linha de desenvolvimento de um sistema gerenciador de banco de dados que utiliza o XML como seu modelo de dados. Chamaremos este último de XML nativo.

Em algumas empresas existe a necessidade de armazenar dados históricos, quer seja por questões legais ou por necessidades dos negócios. A manutenção desses dados históricos em um banco de dados relacional muitas vezes é onerosa, devido à necessidade de mantê-los consistentes com a base de dados corrente. Ou seja, alterações no esquema da base corrente podem causar alterações no esquema da base histórica.

Este trabalho visa analisar a adequação do modelo XML para o armazenamento de dados históricos, mais especificamente sua adequação para a modelagem de dados no caso de evolução de esquema. Além disto, apresentaremos um estudo de caso, mostrando como um SGBD XML nativo poderia ser utilizado em uma aplicação bancária. Inicialmente traçaremos um perfil das características do negócio e sua modelagem para, a seguir, apresentar o mapeamento do modelo relacional para

XML e o comportamento dos dois modelos em caso de uma evolução de esquema. Baseado neste resultado, apresentaremos as características da implementação física do sistema tanto em um banco de dados relacional como em um banco de dados XML nativo, traçando um perfil comparativo destas duas formas de armazenamento e suas linguagens de consulta.

Palavras-chave: banco de dados relacional, banco de dados XML nativo, formas de conversão do modelo relacional para XML, evolução de esquema, armazenamento de dados históricos.

ABSTRACT

XML (*eXtensible Markup Language*) has become a standard for data representation in the Internet. It has been defined to facilitate the publication and extration of data in this medium.

Nevertheless, most companies use relational data bases to store its data. Relational data bases have been used for the past 20 years and have been recognized as a sophisticated, stable and high performance systems.

When companies need to make their data available in the Internet, relational data must be mapped to XML and vice versa. Algorithms for transforming data between this two models have been extensively studied in the literature. Another fruitful line of research is the development of DBMS that adopt XML as its data model. We will refer to them as native XML DBMS.

Companies usually need to store historical data due to legal or business needs. Maintaining historical data in a relational database is expensive when it is necessary to keep them consistent with the current database. More specifically, schema modifications in the current base are reflected as updates in the schema of the historical base.

This dissertation analyzes the adequacy of XML to store historical data, more specifically, to determine how changes in the schema affects an XML schema definition and its data. We will present a case study, showing how a native SGBD CML can be used in a banking application. First, we will describe the characteristics of the business and its modeling. Then, we will present a mapping of relational model into XML and the behavior of the two models in case of a schema evolution. Based on this result, we will analyze the physical implementation of the system in a relational data base, as well as in a native data base XML, taking into consideration the two storage forms and their query languages.

Key-Words: relational data base, native XML data base, mapping from relational to XML, schema evolution, storage of historical data.

1 - INTRODUÇÃO

1.1 Contextualização

Na última década, os sistemas gerenciadores de bancos de dados (SGBD´s) relacionais consolidaram-se como uma estrutura sofisticada, estável e de alto desempenho, além de possuírem um otimizador para efetuar consultas de forma que o usuário não se preocupe com a estrutura dos dados. Diversos SGBD´s relacionais permitem interoperabilidade de acesso através da qual, em uma mesma consulta, é possível acessar dados de SGBD´s distintos. Com o aparecimento de novas tecnologias e com o crescimento do volume dos dados armazenados pelas empresas, questões como a forma de armazenamento dos dados e a maneira de disponibilizá-los vêm ganhando um enfoque cada vez maior nos últimos anos. O crescimento do volume de dados e a necessidade de manter dados históricos fez com que os recursos despendidos com discos para armazenamento dos bancos de dados aumentasse, bem como o custo de fitas e cartuchos para *backups*. Além disso, na manutenção de dados históricos deve ser considerado o tempo gasto na execução dos utilitários de banco de dados, tais como backup, reorganização, atualização de estatísticas e verificação de integridade, pois quanto maior o tamanho da base, maior o tempo de execução dos utilitários. Também deve ser considerado que a maioria dos utilitários exige a utilização de recursos adicionais, tanto para o processamento como para o armazenamento de dados. Boa parte desses problemas pode ser resolvido com a distribuição dos dados em diversos equipamentos com processamento paralelo, mas é comum, por exemplo, existirem processos de verificação de integridade de banco de dados com tempo de execução maior que 12 horas. Devido a esse alto custo de armazenamento, as empresas acabam procurando meios alternativos de armazenamento, principalmente para dados históricos, pois estes normalmente possuem baixo índice de consultas. Desta forma, é comum manter relatórios de dados históricos em microfilmes e armazenar dados

históricos em arquivos seqüenciais. Porém, a recuperação dessas informações nunca é imediata. Quando as necessidades do negócio exigem que os dados históricos sejam consultados instantaneamente, é necessário mantê-los em SGBD´s relacionais. Um problema que ocorre quando dados históricos são mantidos em SGBD´s relacionais é quando há a necessidade de alterar sua estrutura: criar ou alterar uma coluna para atender uma nova necessidade de negócio é bastante comum e esta alteração quase sempre deve ser refletida nos dados históricos. Porém isso pode ser bastante trabalhoso, pois algumas alterações exigem a recriação da tabela e uma nova carga dos dados.

1.2 Objetivos

O objetivo desta dissertação é analisar a adequação do formato XML [1], que tem se tornado um padrão para representação de dados na Internet, para o armazenamento de dados históricos. As mesmas características do XML que o difundiram para troca de informações na Internet, tais como ser semi-estruturado e auto-descritivo, podem também torná-lo interessante para o armazenamento de dados históricos. Com a adoção do XML pudemos observar a simplificação das manutenções no caso de evolução do esquema.

Um documento XML é um conjunto de dados semi-estruturados que permite a definição de diferentes tipos de elementos complexos, inclusive elementos multivalorados. O crescimento do uso do XML fez surgir a necessidade do armazenamento dos documentos XML em SGBD´s relacionais. Porém, o armazenamento estruturado de um documento XML em um banco de dados relacional nem sempre ocorre de forma direta devido a algumas incompatibilidades de estrutura. Para resolver essas incompatibilidades e poder armazenar documentos XML em bancos de dados relacionais é necessário fazer um mapeamento dos dados semi-estruturados para uma estrutura relacional [4]. Outra possibilidade para

armazenamento de XML é mantê-lo em sua forma original. Chamaremos os SGBD´s que utilizam o XML como modelo de dados de XML nativo.

Para analisar a adequação da utilização de SGBD´s XML nativo para o armazenamento de dados históricos, apresentaremos um estudo de caso que consiste em utilizar o SGBD Tamino XML Server [14] em uma aplicação bancária.

1.3 Estrutura do Trabalho

O restante da dissertação está organizado da seguinte forma:

Capítulo 2 – Descreve alguns conceitos sobre XML e o modelo relacional, além de apresentar algumas técnicas para conversão de modelo relacional para XML.

Capítulo 3 – Apresenta aspectos de evolução de esquema de dados históricos, tendo como exemplo uma aplicação bancária, apresentando as diversas formas de armazenamento de dados históricos e os fatores de motivação para uso do XML. Também é apresentado como a evolução de esquema é tratada no modelo orientado a objetos.

Capítulo 4 – Apresenta as propostas para armazenamento de XML pelos SGBD´s relacionais e um estudo de caso com a utilização de um esquema de armazenamento misto, mantendo os dados ativos em um SGBD´s relacional e os dados históricos em um SGBD XML nativo.

Capítulo 5 – Apresenta a conclusão do trabalho.

2 – MODELOS DE DADOS

Para facilitar o entendimento do trabalho, neste capítulo descreveremos os modelos de dados XML e relacional, apresentando também algumas técnicas para conversão de dados do modelo relacional para XML.

2.1 XML

Com a disseminação da Internet, houve a necessidade de se criar um formato de dados que facilitasse a publicação e extração de dados deste meio.

O XML (*Extensible Markup Language*) [1] é um novo padrão adotado pela *World Wide Web Consoltium* (W3C) para complementar a HTML (*HyperText Markup Language*) para troca de dados na Internet.

Assim como o HTML, o XML também é um subconjunto do SGML (*Standard Generalized Markup Language*). As marcações (*tags*) HTML têm o propósito primário de descrever como um item de dado é apresentado, enquanto as marcações XML descrevem o dado em si, o que permite que os programas consigam interpretá-los.

O XML representa os dados de forma semi-estruturada, permitindo uma maior heterogeneidade dos dados e podem, ou não, virem acompanhados de uma definição de esquema no formato DTD [2] ou XML Schema [24].

2.1.1 Sintaxe

No HTML as *tags* são utilizadas para a formatação do texto. Por exemplo: Título fazem com que **Título** seja apresentado em **negrito** no texto e Título fazem com que *Título* apareça em *itálico*. Já no XML, essas *tags* (marcações) têm a função de representação de metadados, pois através delas é possível

identificar dentro do texto o que os dados representam. No modelo relacional, o elemento básico é chamado de coluna, já no documento XML o elemento básico é chamado de elemento, que nada mais é do que um texto delimitado por marcações, como <cliente> e </cliente>. Dentro de um elemento podemos ter um texto, outros elementos, ou uma mistura de textos e elementos. A expressão <cliente> é chamada de marca de início e </cliente> é chamada de marca de fim e são definidas conforme a necessidade do usuário.

Para facilitar a compreensão das características do XML, na Figura 1 apresentamos um exemplo de um documento XML referente a um cadastro de clientes.

```
<cadastro_cliente>
  <cliente cli_id=”1”>
     <nome> João da Silva </nome>
     <cpf> 12345678901 </cpf>
     <conta_corrente cc_id=”10101”>
        <titularidade> 1 </tituralidade>
     </conta_corrente>
  </cliente>
 <cliente cli_id=”2”>
     <nome> Maria da Silva </nome>
     <cpf> 23456789012 </cpf>
     <conta_corrente cc_id=”10101”>
        <titularidade> 2 </tituralidade>
     </conta_corrente>
  </cliente>
</cadastro_cliente>
```

Figura 1 - Exemplo de Documento XML

Observamos no exemplo da Figura 1 como as *tags* XML, que descrevem os dados de um cliente, formam uma árvore estruturada, sendo que cada nó da árvore está em um nível. O encaixamento das *tags* <nome>, <cpf> e <conta_corrente cc_id=”10101”> dentro de <cliente cli_id=”1”> indicam que estes dados pertencem ao cliente cujo identificador é “1”.

São as características do XML de ser semi-estruturado e auto-descritivo, que difundiram o seu uso para troca de informações tanto na Internet, como em outras aplicações. Seu uso propiciou outras possibilidades como a execução de consultas

sobre os conteúdos destes documentos, extraindo, sintetizando e analisando seus conteúdos.

2.1.2 Expressões de Caminho

Podemos representar um documento XML em forma de grafo com arestas rotuladas no qual, partindo-se da raiz, existe um único caminho para se atingir cada nó. A notação (*o, r, d*) [5] representa que existe aresta rotulada de r partindo do nó *o* para o nó *d*.

Uma seqüência de arestas rotuladas *l1 / l2 / ... / ln* é denominada de expressão de caminho.

A Figura 2, apresentada a seguir, representa um documento XML em forma de grafo.

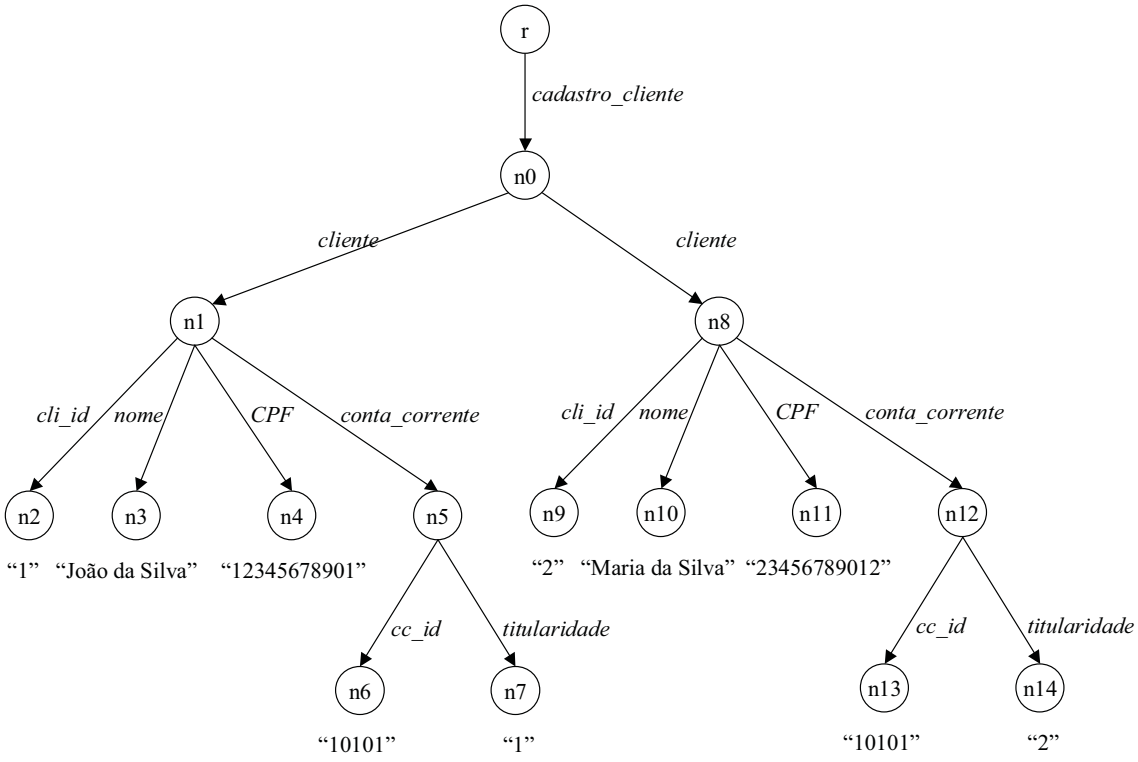

Figura 2 - Documento XML no formato de grafo

Na Figura 2, o caminho */cadastro_cliente/cliente/nome* retorna o conjunto de nós com o seguinte conjunto de *strings* {"João da Silva", "Maria da Silva"}.

Assim, o resultado de uma expressão de caminho *l1 / l2 / ... / ln* é um conjunto de nós v_n, tais que existam arestas (r, l_1, v_1), (v_1, l_2, v_2),...,(v_{n-1}, l_n, v_n) no grafo de dados, sendo que r é a raiz do grafo.

Para conseguirmos todos os nomes de clientes nesse documento XML, podemos também utilizar um símbolo "curinga". Assim, se "*" for tal símbolo, teremos *cadastro_cliente/*/nome* como expressão de caminho para todos os clientes no documento que tenham uma aresta entre *cadastro_cliente* e *nome*. Quando não sabemos quantas arestas existem entre duas arestas, podemos utilizar o símbolo "//". Por exemplo, para obter o nome de todas as titularidades no documento, podemos utilizar a expressão *cadastro_cliente//titularidade*. Desta maneira, pode haver 0 ou mais arestas entre *cadastro_cliente* e *titularidade*.

As expressões de caminho apresentadas representam um pequeno subconjunto das expressões de caminho da linguagem Xpath. [6].

2.1.3 DTD

Um documento XML pode opcionalmente vir acompanhado de uma definição de esquema na forma de uma DTD (*Data Type Definition*) [2]. Uma DTD consiste no nome da marca raiz do documento, seguida das várias declarações de marcação que indicam as marcas permitidas no documento e sua estrutura associada. A seguir apresentamos a DTD do documento XML da Figura 1.

```
<!DOCTYPE cadastro_cliente [
  <!ELEMENT cadastro_cliente (cliente*)>
  <!ELEMENT cliente (nome, cpf?|cnpj?, conta_corrente*)>
  <!ATTLIST cliente cli_id  ID #REQUIRED>
  <!ELEMENT nome (#PCDATA)>
  <!ELEMENT cpf  (#PCDATA)>
  <!ELEMENT cnpj  (#PCDATA)>
  <!ELEMENT conta_Corrente(titularidade)>
  <!ATTLIST conta_corrente cc_id  ID #REQUIRED>
  <!ELEMENT titularidade (#PCDATA)>
]>
```

Figura 3 - DTD para o Documento XML da Figura 1

Na DTD da Figura 3, a primeira linha indica qual é a raiz do documento que neste caso é cadastro_cliente. O símbolo "?" significa que o elemento em questão pode aparecer nenhuma ou apenas uma vez no elemento em que está contido. O símbolo "*" significa que o elemento em questão pode aparecer nenhuma ou mais vezes no elemento em que está contido. (#PCDATA) indica que o elemento em questão é um texto, semelhante a um *string* (conjunto de caracteres).

Atributos definidos como #REQUIRED devem ser sempre informados. Atributos definidos como #IMPLIED são facultativos.

O símbolo "|" indica um operador de união, e "cpf?|cnpj?" indica que será informado um cpf ou um cnpj.

<!ATTLIST cliente cli_id ID #REQUIRED> define um atributo para identificação do elemento complexo cliente.

Existem diversas propostas de linguagens de consultas para XML [3][10][11], mas a que vem se consolidando como padrão pelo W3C é o XQuery [3], que será descrita na próxima seção.

2.1.4 XQuery

Xquery [3] é uma linguagem de consulta composta basicamente de quatro cláusulas. Uma consulta utilizando Xquery é definida por uma expressão conhecida como FLWR sendo que:

. F – cláusula FOR: utilizada para associar uma variável a uma expressão de caminho.

. L – cláusula LET: utilizada para atribuir o conteúdo de um elemento a uma variável.

. W – cláusula WHERE: utilizada para restringir ou selecionar os elementos resultantes da consulta.

. R – cláusula RETURN: utilizada para construir o resultado da consulta no formado XML.

As variáveis possuem como prefixo “$” e os atributos possuem como prefixo “@”.

A seguir é exemplificada uma consulta XQuery com base no documento XML da Figura 1:

```
FOR
    $b IN doc(www.cadastroteste.com/cadastro.xml)//cadastro_cliente/cliente
WHERE $b/@cli_id = '1'
RETURN
    <RESULT> $b/nome, $b/cpf </RESULT>
```

A consulta apresenta como resultado:

```
<RESULT>
  <nome>João da Silva</nome><cpf>12345678901</cpf>
</RESULT>
```

No exemplo acima é pesquisado o documento XML www.cadastroteste.com/cadastro.xml, através da expressão de caminho //cadastro_cliente/cliente, selecionando os clientes que possuem @cli_id = '1'. O resultado retorna os elementos nome e cpf no formato XML, delimitados pelas *tags* **<RESULT>** e **</RESULT>**.

A seção 4.4 Consultas apresenta outros exemplos de XQuery.

2.2 Modelo Relacional

A definição de um esquema em um modelo relacional[35] nada mais é do que um conjunto de esquemas de tabelas e seus relacionamentos.

Uma tabela consiste de um conjunto de dados dispostos em forma de linhas e colunas. As linhas são conjuntos de dados do mesmo tipo, isto é, possuem os mesmos atributos. Estes dados devem ser atômicos, isto é, não podem ser multivalorados. Cada coluna é um atributo das linhas e todos os valores de uma mesma coluna devem ser do mesmo tipo (caracter, numérico, data, etc).

Para identificarmos uma linha, geralmente não precisamos saber o valor de todas as colunas, mas apenas de um subconjunto delas. Esse subconjunto que identifica unicamente uma linha é chamado de chave primária da tabela. Por exemplo, se definirmos uma tabela BANCO contendo os atributos: cod_bco, nome_bco, cod_cnpj_bco, podemos definir que cod_bco, que contém o código do banco, é chave primária da tabela se um código de banco identificar unicamente determinado banco. Ou seja, não existem dois bancos com o mesmo código.

2.2.1 Relacionamentos 1xN

Um relacionamento entre duas tabelas pode ser de três tipos: 1x1, 1xN e MxN. Todo relacionamento entre duas tabelas envolve um atributo de ligação conhecido como chave estrangeira. Em um relacionamento 1xN, a chave estrangeira reside na tabela na qual a relação é (N), e é uma replicação da chave primária da tabela na qual a relação é (1), como exemplificado no diagrama da Figura 4, segundo a notação UML [36].

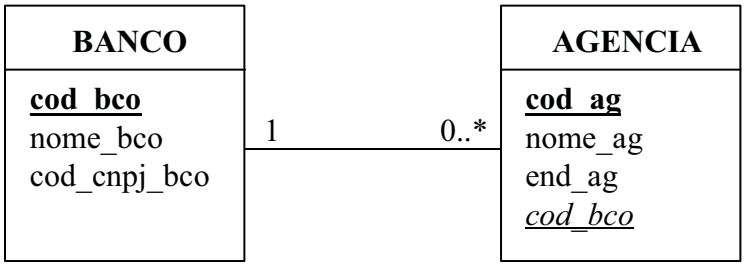

Figura 4 - Relacionamento 1xN

A Figura 4 apresenta as tabelas **BANCO** e **AGENCIA** e o relacionamento entre elas, com a chave primária de **BANCO**: **cod_bco** sendo relacionada com a tabela **AGENCIA** através da chave estrangeira *cod_bco*. Os atributos sublinhados e em **negrito** representam a chave primária e os atributos sublinhados e em *itálico* representam as chaves estrangeiras.

O uso da chave estrangeira implementa uma regra que chamamos de integridade referencial, na qual um valor só pode ser atribuído a uma coluna que é chave estrangeira, se existir uma tupla com este valor na tabela na qual esta coluna é chave primária. No exemplo acima, não pode existir uma tupla na tabela **AGENCIA** com um determinado valor de *cod_bco*, sem que este valor de **cod_bco** exista na tabela **BANCO**. A integridade referencial é implementada automaticamente pela maioria dos SGBD´s relacionais. O valor da chave estrangeira pode ser nulo para os relacionamentos considerados facultativos.

2.2.2 Relacionamentos MxN

Um relacionamento MxN entre tabelas A e B define que cada linha da tabela A pode estar relacionada a N linhas da tabela B, e que cada linha da tabela B pode estar relacionada a M linhas da tabela A. Assim, relacionamentos MxN teriam que implementar como chave estrangeira uma coluna com múltiplos valores referentes à chave primária da outra tabela. Como isso não é permitido no modelo relacional, relacionamentos MxN derivam uma terceira tabela conhecida como tabela associativa.

A tabela associativa derivada do relacionamento MxN passa a ter relacionamentos de Nx1 com as outras duas tabelas. A tabela associativa deve conter as chaves estrangeiras das tabelas com quem se relaciona. A combinação destas chaves estrangeiras forma a chave primária da tabela associativa.

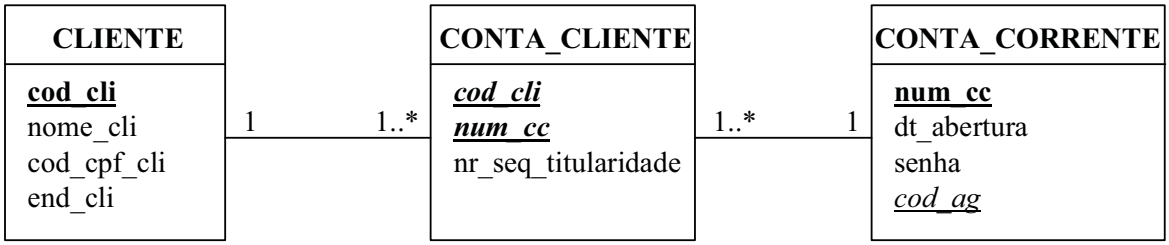

Figura 5 - Relacionamento MxN derivando uma tabela associativa

No exemplo da Figura 5 a tabela associativa **CONTA_CLIENTE** resulta do relacionamento entre **CLIENTE** e **CONTA_CORRENTE**, no qual um cliente pode possuir várias contas correntes e uma conta corrente pode pertencer a clientes de diferentes titulares.

Através da tabela **CONTA_CLIENTE** é possível determinar quais são as contas correntes de um cliente e qual sua titularidade em cada uma delas. A chave primária da tabela associativa **CONTA_CLIENTE** é a combinação das chaves estrangeiras ***cod_cli*** para a tabela **CLIENTE** e ***num_cc*** para a tabela **CONTA_CORRENTE**.

2.3 Conversão do Modelo Relacional para o XML

Existem diversos algoritmos para o mapeamento de tabelas relacionais para documentos XML. Nesta seção descreveremos três deles: DB2XML, *Flat Translation* e *Nesting-based Translation*. Relacionaremos as principais características de cada método e um exemplo de DTD gerado para as tabelas de **CONTA_CORRENTE** e **CONTA_CLIENTE** da Figura 5.

2.3.1 DB2XML

Este método [7] faz uma associação direta entre tabelas e elementos do documento XML. Ou seja, para cada tabela é criado um elemento e cada coluna desta tabela é associada a um sub elemento. Para representar se uma coluna pode ser nula ou não, o DB2XML adiciona o atributo especial *ISNULL* para cada coluna. Por exemplo, a Figura 6 apresenta a DTD resultante do mapeamento pelo método DB2XML para as tabelas CONTA_CORRENTE e CONTA_CLIENTE.

```
<!ELEMENT CONTA_CORRENTE  (num_cc, dt_abertura, senha, cod_ag)>
<!ELEMENT num_cc          (#PCDATA)>
<!ATTLIST   num_cc        ISNULL (true|false) #IMPLIED>
<!ELEMENT dt_abertura     (#PCDATA)>
<!ATTLIST   dt_abertura   ISNULL (true|false) #IMPLIED>
<!ELEMENT senha           (#PCDATA)>
<!ATTLIST   senha         ISNULL (true|false) #IMPLIED>
<!ELEMENT cod_ag          (#PCDATA)>
<!ATTLIST   cod_ag        ISNULL (true|false) #IMPLIED>

<!ELEMENT CONTA_CLIENTE  (cod_cli, num_conta, nr_seq_titularidade)>
<!ELEMENT cod_cli          (#PCDATA)>
<!ATTLIST   cod_cli        ISNULL (true|false) #IMPLIED>
<!ELEMENT num_conta        (#PCDATA)>
<!ATTLIST   num_conta      ISNULL (true|false) #IMPLIED>
<!ELEMENT nr_seq_titularidade  (#PCDATA)>
<!ATTLIST   nr_seq_titularidade  ISNULL (true|false) #IMPLIED>
```

Figura 6 - DTD resultante do mapeamento pelo método DB2XML

Observamos neste método de mapeamento que a integridade referencial garantida automaticamente pelos SGBD´s relacionais é perdida, uma vez que não é

possível expressar este tipo de restrição entre elementos. Notamos também que o sub elemento num_conta do elemento CONTA_CLIENTE teve que ser renomeado, pois não é permitido nomes de elementos repetidos em uma mesma DTD. Por outro lado, no exemplo da Figura 6, se num_cc do elemento CONTA_CORRENTE fosse mapeado para um atributo do tipo ID e num_conta do elemento CONTA_CLIENTE fosse mapeado para um atributo IDREF, a restrição estaria em parte sendo definida.

2.3.2 *Flat Translation*

Este método [4] pode utilizar tanto elementos como atributos para representar as colunas de uma tabela. A representação utilizando elementos é conhecida como modo orientado a elemento e é bastante similar ao DB2XML, porém sem a necessidade de adição de um atributo especial para representar a obrigatoriedade das colunas. Já na representação utilizando atributos, conhecida como modo orientado a atributos, cada coluna é definida como um atributo. Para representar se uma coluna pode ser nula ou não, o atributo pode ser definido como #REQUIRED, no caso de ser obrigatório, ou #IMPLIED, caso possa ser nulo. Por exemplo, a Figura 7 apresenta a DTD resultante da conversão pelo método *Flat Translation* no modo orientado a atributos para as tabelas CONTA_CORRENTE e CONTA_CLIENTE.

```
<!ELEMENT CONTA_CORRENTE  (EMPTY)>
<!ATTLIST   CONTA_CORRENTE
   num_cc              CDATA #REQUIRED
   dt_abertura         CDATA #IMPLIED
   senha               CDATA #IMPLIED
   cod_ag              CDATA #REQUIRED>

<!ELEMENT CONTA_CLIENTE  (EMPTY)>
<!ATTLIST   CONTA_CLIENTE
  cod_cli              CDATA #REQUIRED
  num_conta            CDATA #REQUIRED
  nr_seq_titularidade  CDATA #IMPLIED
```

Figura 7 - DTD resultante do mapeamento pelo método *Flat Translation* no modo orientado a atributos

Apesar de não automatizado pelo método, o atributo que é chave primária na tabela relacional pode ser mapeado para um atributo do tipo ID do elemento XML e o atributo que é chave estrangeira na tabela relacional pode ser mapeado para um atributo do tipo IDREF, ou seja, é possível fazer o mapeamento de num_cc do elemento CONTA_CORRENTE para um atributo do tipo ID e de num_conta do elemento CONTA_CLIENTE para um atributo IDREF.

2.3.3 *Nesting-based Translation*

Este método [4] tem por base o *Flat Translation* [4], apresentado no item 2.3.2 tanto no modo orientado a elementos, como no modo orientado a atributos, mais a implementação de um operador para aninhamento. Este operador de aninhamento visa agrupar tabelas interconectadas (resultantes de uma associação 1 x N), substituindo as chaves estrangeiras pelos elementos da tabela associada, que passam a ser representados na DTD com "*" ou "+". Na linguagem relacional, as tabelas estão sendo desnormalizadas através do uso de elementos complexos multivalorados.

Por exemplo, a Figura 8 apresenta uma DTD resultante da conversão da tabela CONTA_CORRENTE interconectada à tabela CONTA_CLIENTE, pelo método *Nesting-based Translation* no modo orientado a elementos.

```
<!ELEMENT CONTA_CORRENTE (num_cc, dt_abertura?, senha?, cod_ag,
conta_cliente* )>
<!ELEMENT num_cc                (#PCDATA)>
<!ELEMENT dt_abertura                   (#PCDATA)>
<!ELEMENT senha                         (#PCDATA)>
<!ELEMENT cod_ag                        (#PCDATA)>
<!ELEMENT conta_cliente (cod_cli, nr_seq_titularidade)>
<!ELEMENT cod_cli               (#PCDATA)>
<!ELEMENT nr_seq_titularidade           (#PCDATA)>
```

Figura 8 - DTD resultante do mapeamento pelo método *Nesting-based Translation* no modo orientado a elementos

Notamos que na Figura 8, o elemento multivalorado conta_cliente foi inserido em CONTA_CORRENTE com base nos atributos da tabela CONTA_CLIENTE,

com a qual possui um relacionamento 1xN. Com a inclusão do elemento multivalorado conta_cliente, os dados da tabela CONTA_CLIENTE já estariam mapeados no documento XML e não seria mais necessária a definição de um elemento para CONTA_CLIENTE.

A Figura 9 apresenta outro exemplo de DTD resultante da conversão da tabela CONTA_CORRENTE interconectada à tabela CONTA_CLIENTE pelo método *Nesting-based Translation* no modo orientado a atributos.

```
<!ELEMENT CONTA_CORRENTE  (conta_cliente *)>
<!ELEMENT conta_cliente (cod_cli,  nr_seq_titularidade)>
<!ELEMENT cod_cli           (#PCDATA)>
<!ELEMENT nr_seq_titularidade    (#PCDATA)>
<!ATTLIST   CONTA_CORRENTE
  num_cc              CDATA #REQUIRED
  dt_abertura         CDATA #IMPLIED
  senha               CDATA #IMPLIED
  cod_ag              CDATA #REQUIRED>
```

Figura 9 - DTD resultante do mapeamento pelo método *Nesting-based Translation* no modo orientado a atributos

Notamos na Figura 9 que, apesar do mapeamento no modo orientado a atributos representar os atributos de uma tabela como atributos em XML, os atributos da tabela relacionada que foi incorporada, neste caso CONTA_CLIENTE, foram mapeados como elementos.

O método *Nesting-based Translation* também sugere que, para as tabelas associativas (resultantes de uma associação M x N), os elementos que são chaves estrangeiras (referentes às chaves primárias das tabelas origem do relacionamento), sejam representados na DTD com "*" ou "+", pois seus valores em determinadas situações podem ser repetidos. Desta forma, em um mesmo registro, junto aos valores de uma determinada chave estrangeira e o conjunto dos demais atributos não chave, podemos ter valores repetidos da outra chave estrangeira.

Por exemplo, na Figura 10, o elemento CONTA_CLIENTE, que é resultante da tabela associativa CONTA_CLIENTE convertida pelo método *Nesting-based Translation*

no modo orientado a elementos, possui os elementos num_cc e cliente definidos como multivalorados.

```
<!ELEMENT CONTA_CLIENTE (num_cc*, cliente*)>
<!ELEMENT num_cc          (#PCDATA)>
<!ELEMENT cliente (cod_cli, nr_seq_titularidade)>
<!ELEMENT cod_cli         (#PCDATA)>
<!ELEMENT nr_seq_titularidade     (#PCDATA)>
```

Figura 10 - DTD resultante do mapeamento de uma tabela associativa pelo método *Nesting-based Translation* no modo orientado a elementos

Para a representação da Figura 10, um cliente que é primeiro titular em duas contas correntes teria os dois valores de num_cc num mesmo registro, como apresentado na Figura 11.

```
<CONTA_CLIENTE>
      <num_cc>  19232221 </num_cc>
      <num_cc>  17423525 </num_cc>
      <cliente>
            <cod_cli> 123 </cod_cli>
            <nr_seq_titularidade> 1 </nr_seq_titularidade>
      </cliente>
</CONTA_CLIENTE>
```

Figura 11 - XML resultante do mapeamento de uma tabela associativa pelo método *Nesting-based Translation*

Por explorar de forma mais adequada a possibilidade de aninhar dados relacionados no formato XML, nos próximos capítulos, sempre que for necessário, adotaremos o método *Nesting-based Translation* para obter uma DTD referente a um esquema relacional. A utilização do método não apresenta vantagens para implementação de tabelas associativas (resultantes de uma associação M x N), quando os elementos multivalorados possuírem poucos valores repetidos, pois a quantidade final de registros XML acaba ficando próxima da quantidade de linhas da tabela associativa.

3 – EVOLUÇÃO DE ESQUEMA DE DADOS HISTÓRICOS

Existem várias situações nas quais as empresas precisam manter os seus dados por vários anos, quer seja por necessidades legais ou por necessidades dos negócios. Para analisar o impacto da evolução de esquemas no armazenamento de dados históricos, utilizaremos como exemplo uma aplicação bancária que será descrita na seção 3.1. As diversas formas de armazenamento serão apresentadas na seção 3.2. A seção 3.3 descreve a motivação para a utilização do XML para armazenamento de dados históricos. A seção 3.4 conclui o capítulo apresentando os conceitos utilizados pelo modelo orientado a objetos para dar suporte à evolução de esquema, contrastando-os com as soluções oferecidas pelo modelo XML.

3.1 Exemplo de uma Aplicação Bancária

Utilizaremos as informações básicas de um cadastro de conta corrente como exemplo de uma aplicação bancária. O modelo relacional com base na notação UML do cadastro de conta corrente é apresentado na figura 12.

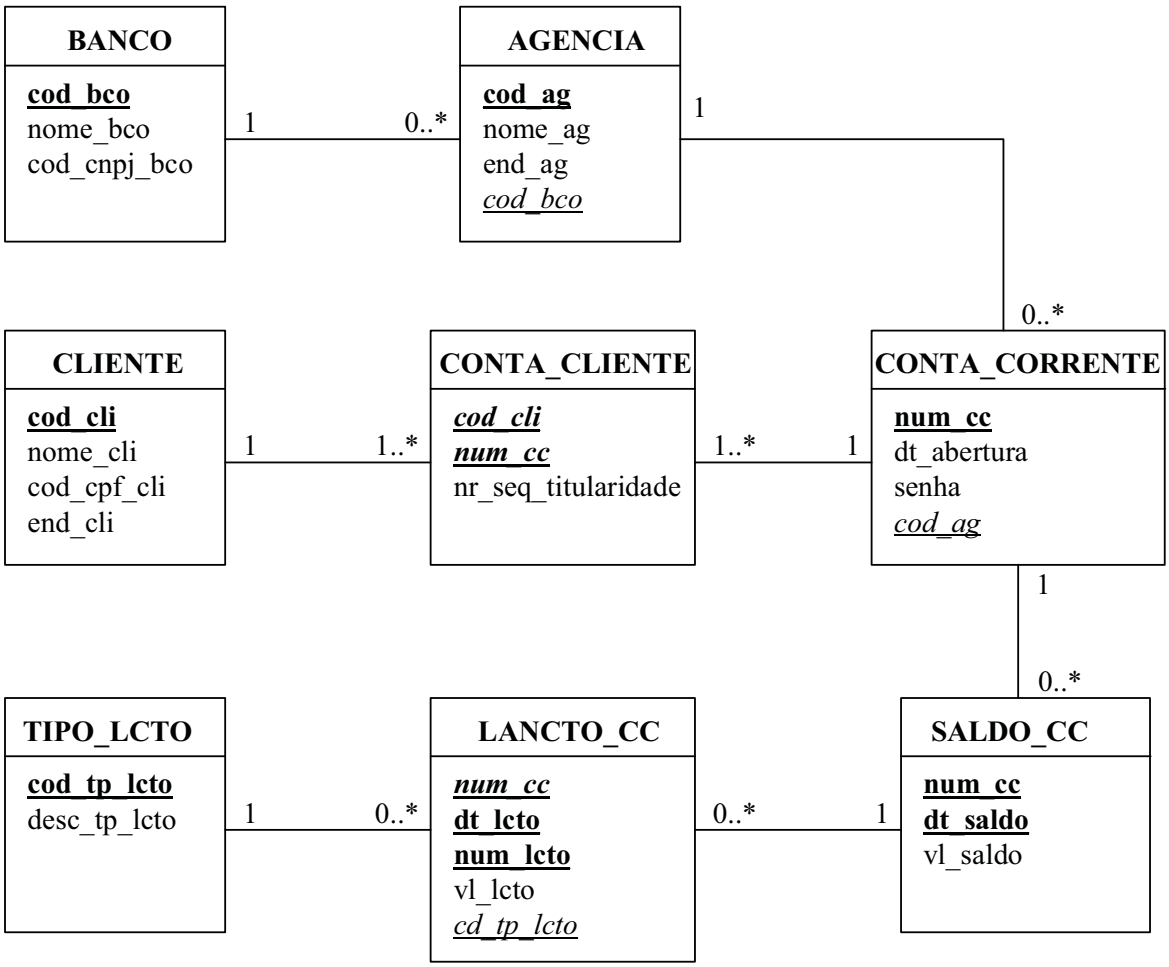

Figura 12 - Modelo Relacional do Cadastro de Conta Corrente

Na Figura 12, os atributos sublinhados e em **negrito** compõem a chave primária e os atributos sublinhados e em *itálico* são chaves estrangeiras.

Na Figura 12, as informações sobre as tabelas foram simplificadas de forma a tornar o modelo mais legível, resumindo-se às informações mais relevantes a este estudo de caso.

Para entender um pouco melhor o modelo da Figura 12, é preciso conhecer um pouco mais sobre suas tabelas e seus relacionamentos:

BANCO: é a tabela que contém os dados cadastrais dos bancos.

AGENCIA: é a tabela que contém os dados cadastrais das agências. Uma agência deve pertencer a um banco.

CONTA_CORRENTE: é a tabela que contém os dados cadastrais de uma determinada conta corrente. Para cadastrar uma conta corrente é preciso informar a agência à qual ela pertence.

CLIENTE: é a tabela que contém os dados cadastrais dos clientes.

CONTA_CLIENTE: é a tabela associativa que identifica qual é a titularidade do cliente. Ela é resultante da associação das tabelas **CLIENTE** com **CONTA_CORRENTE**.

SALDO_CC: é a tabela que contém o valor do saldo do início do dia de uma determinada conta corrente para uma determinada data. Para cada dia útil, é gerada uma nova linha nesta tabela contendo o saldo da respectiva data. O saldo é gerado com base no saldo do dia anterior e aplicados os respectivos lançamentos do dia.

LANCTO_CC: é a tabela com todos os lançamentos de uma determinada conta corrente, apresentados por data.

TIPO_LCTO: é a tabela que contém todos os tipos de lançamentos de uma conta corrente, a saber: depósito em dinheiro, depósito em cheque, saque em dinheiro, saque com cheque, cheque compensado, pagamento, encargos sobre empréstimos, depósito em aplicação, resgate de aplicação.

O modelo apresentado na Figura 12 representa uma situação atualizada das informações do negócio e a estrutura dos dados históricos acompanha a mesma estrutura dos dados ativos.

Algumas alterações de esquema devem ser refletidas nos dados históricos. Para se ter uma idéia dos tipos de alterações de esquema que ocorrem em uma base histórica, foi feito um levantamento, com base em um aplicativo bancário real, no qual foram identificadas as principais alterações de esquema ocorridos nos últimos cinco anos, as quais resultaram em 48 alterações em tabelas. Dessas alterações:

. 76% são referentes à criação de novas colunas;

. 8% são referentes à alteração do tamanho de coluna;

. 4% são referentes à alteração do nome de coluna;

. 8% são referentes à criação de novas colunas que passaram a fazer parte da chave primária;

. 4% são referentes à exclusão de colunas.

Uma alteração de esquema muitas vezes pode ser implementada através de um simples comando. Porém, existem diversas situações nas quais uma alteração exige a recriação da tabela. A necessidade da recriação da tabela se deve ao fato que a implementação da alteração de esquema implica na alteração da estrutura física da tabela, de forma que a mesma só pode ser implementada com a utilização implícita de uma tabela temporária auxiliar.

Dentre as alterações de esquema a pouco apresentadas, as que exigem a recriação da tabela são:

. Criação de novas colunas *NOT NULL*;

. Alteração de tamanho de coluna;

. Criação de novas colunas que passaram a fazer parte da chave primária (obrigatoriamente *NOT NULL*);

. Exclusão de colunas.

Este levantamento teve como objetivo identificar as principais situações em que uma alteração no esquema relacional tem grande impacto nos dados históricos.

3.2 Formas de Armazenamento de Dados Históricos

Conforme as características e a necessidade de recuperação dos dados históricos, e tendo em vista a minimização do custo de armazenamento destes dados, podemos identificar três tipos de armazenamento: microfilmagem, arquivos seqüenciais e bancos de dados. Na seqüência, descreveremos cada uma destas alternativas, analisando suas vantagens e desvantagens.

3.2.1 Microfilmagem

Este tipo de armazenamento é utilizado quando o prazo para a recuperação da informação pode ser relativamente grande. É o meio de armazenamento mais barato, porém a recuperação da informação, normalmente centralizada, demanda uma intervenção manual, o que torna o processo bastante lento. Aplica-se a recuperação de informações esporádicas e de baixo volume. Em aplicações bancárias tais informações envolvem, por exemplo, documentos assinados por clientes e extratos de conta corrente com mais de seis meses. O processo funciona da seguinte forma: o cliente faz uma solicitação de extrato de um determinado período na agência. Essa solicitação é processada com as outras solicitações das outras agências à noite pela área de microfilmagem, na qual um funcionário pesquisa, para cada solicitação, a microficha da referida conta corrente para uma determinada data, gerando uma cópia do extrato, que é então enviado à agência. Nesses casos, o cliente espera em média dois dias pelo extrato. Outro problema é a limitação das pesquisas, pois as microfichas ficam ordenadas por agência, conta corrente e ano/mês e o conjunto destas informações é exigido para poder chegar às microfichas e recuperar as informações do extrato.

A microfilmagem vem sendo substituída em alguns casos pelo armazenamento de imagens, que possui a vantagem das imagens poderem ser apresentadas diretamente no terminal do usuário. Porém o custo do armazenamento aumenta consideravelmente. Quando existe limitação de banda de rede para o tráfego de imagens, o fluxo para recuperação da informação acaba sendo o mesmo da

microfilmagem, no qual a consulta é centralizada e impressa em um setor de microfilmagem.

3.2.2 Arquivos Seqüenciais

Esta forma para armazenamento de dados históricos aplica-se quando não há necessidade de recuperação instantânea dos dados. Ela depende da execução de um processo para pesquisa e apresentação das informações. Os dados normalmente ficam armazenados em meios de armazenamento mais baratos como fitas e cartuchos. São dados fixos que não sofrem alteração no seu esquema e cujo formato está definido nos programas do processo. Os dados recuperados podem ser apresentados em relatórios ou disponibilizados temporariamente para consulta em bancos de dados relacionais. O tempo de recuperação pode variar de minutos, para processos disparados instantaneamente após a solicitação de consulta, até um dia, para processos programados para execução noturna, sendo que a informação estará disponível apenas no dia seguinte. Na aplicação bancária, este é o caso da maioria dos contratos vencidos a mais de cinco anos.

3.2.3 Bancos de Dados Relacionais

Esta é a forma mais desejada de armazenamento para dados históricos, pois os mesmos ficam sempre disponíveis para consulta, possibilitando inclusive a junção destes dados com os demais dados ativos da aplicação. Porém, é a forma mais cara de mantê-los. Além do alto custo dos discos para o armazenamento dos bancos de dados, também existe o custo das áreas de fitas e cartuchos para *backups*. Além disso, deve ser considerado o tempo de execução dos utilitários de banco de dados, tais como backup, reorganização, atualização de estatísticas e verificação de integridade, pois quanto maior o tamanho da base, maior o tempo de execução dos utilitários. Também deve ser considerado que a maioria dos utilitários exige a utilização de recursos adicionais, tanto para o processamento, como para o armazenamento dos dados. Outro fator importante são as alterações no esquema. Por exemplo, a implementação da alteração no tamanho de uma coluna exige a recriação da tabela e, conseqüentemente, a alocação de uma área adicional temporária no próprio banco de dados ou em disco equivalente ao tamanho da tabela. A manutenção envolve o tempo de descarga da tabela, o tempo de carga da tabela, mais o tempo de criação dos seus índices.

A facilidade de efetuar consultas através de visões com junções dos dados nos bancos de dados relacionais fez com que a separação entre os dados históricos e os ativos ficasse transparente para o usuário. É possível definir uma visão relacional com a junção dos dados ativos e históricos e efetuar consultas sobre esta visão.

O que define se um dado deve ir para um banco de dados histórico não é sua idade, mas sim o seu tempo de inatividade. Por exemplo, se um cliente é um cliente ativo e seus dados cadastrais não sofrem manutenção há mais de dez anos, seus dados cadastrais não devem ir para a base de dados históricos. Porém é obrigatório haver colunas com datas para controlar o início e fim da validade do dado. Por outro lado, se um cliente sofreu alteração nos seus dados cadastrais ou foi excluído do cadastro por deixar de ser cliente, seus dados anteriores devem ser armazenados como dados históricos.

Em toda consulta a dados históricos deve ser informada uma data de pesquisa. Com base nessa data, inicia-se a pesquisa na base de dados ativos, verificando se a data de pesquisa está entre a data de início de validade e a data de fim de validade do dado. Caso nenhuma linha tenha sido retornada na consulta de dados ativos, efetua-se a mesma pesquisa na base de dados históricos. Deve-se iniciar a pesquisa pela base de dados ativos, pois, se o dado não sofre manutenção há muito tempo, ele pode não constar na base de dados históricos.

A princípio, uma vez que haja controle das datas de início e fim de validade do dado, os dados históricos poderiam ser armazenados juntamente com os dados ativos. Porém, como a maioria das consultas são sobre os dados ativos, elas acabariam sendo prejudicadas, pois a estrutura e o volume dos dados envolvidos seria muito maior. Além disto, o tempo para realização de *backups* e execução dos demais utilitários também seria muito maior. Como os dados históricos não sofrem atualização, não seria necessária a realização de *backups* com a mesma freqüência que é necessária para os dados ativos. São esses os fatores que motivam a separação dos dados históricos.

3.3 Motivação para o Armazenamento de Dados Históricos em XML

Na seção anterior vimos que há vantagens em armazenar os dados históricos separados dos dados ativos. Além disso, o custo de manutenção dos dados históricos em bancos de dados relacionais é bastante alto no caso de alteração do esquema, pois algumas alterações exigem a recriação da tabela e uma nova carga dos dados. Nesta seção veremos que se os dados forem mantidos no formato XML, é possível fazer uma evolução do esquema sem a necessidade de alterar a estrutura dos dados. Ou seja, alterações no esquema de um documento XML na forma de DTD não necessariamente invalidam documentos gerados de acordo com o esquema antigo. A idéia geral está ilustrada na figura 13.

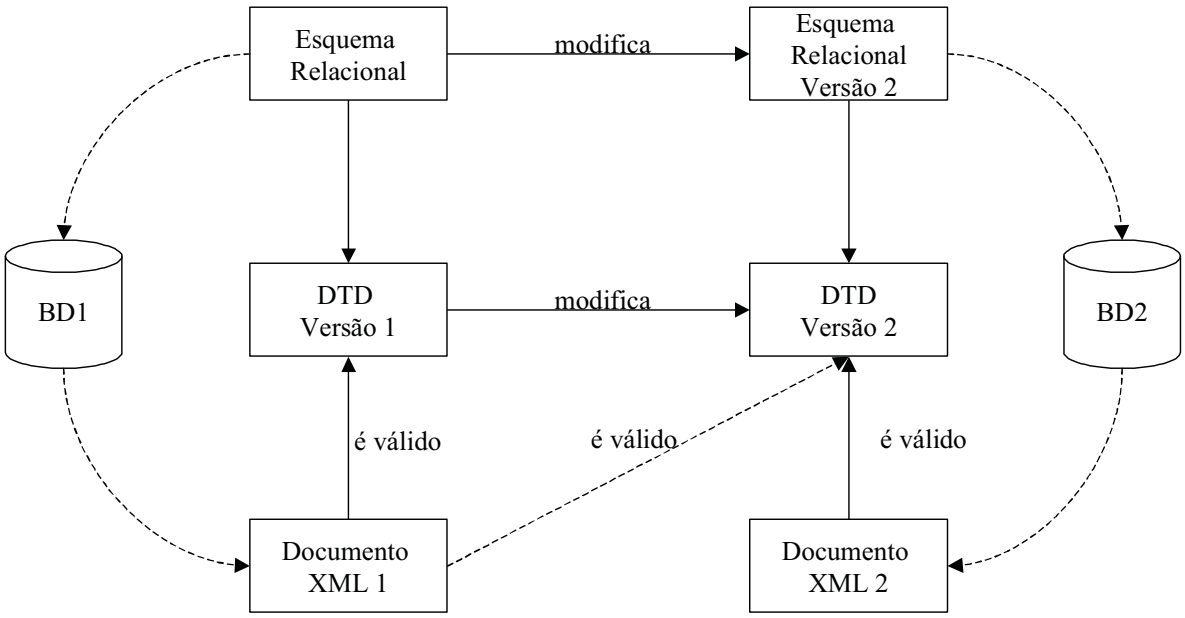

Figura 13 - Esquema da Evolução de Modelo

Para uma tabela relacional 1 de um determinado esquema relacional é feito um mapeamento para um documento XML 1 com base em uma DTD 1. Esta tabela relacional 1 sofre uma evolução de esquema e passa a se chamar tabela relacional 2, o que define agora uma versão 2 do esquema relacional. Ao ser feito o mapeamento para

um elemento XML 2, é feita uma evolução do esquema definido na DTD 1 para a DTD 2 de forma que tanto o elemento XML 1 como o elemento XML 2 são válidos para a DTD 2.

Para exemplificar esta abordagem, considere a tabela de tipo de lançamentos definida como:

```
Create Table TP_LCTO (
cd_tp_lcto  int not null,
desc_tp_lcto varchar(50) not null)
```

De acordo com o método de conversão *Nesting-based Translation* apresentado na seção 2.3.3, esta tabela foi mapeada na seguinte DTD, que chamaremos de DTD-v.1:

```
<!ELEMENT TP_LCTO (desc_tp_lcto)>
<!ELEMENT desc_tp_lcto    (#PCDATA)>
<!ATTLIST   TP_LCTO
  cd_tp_lcto            CDATA #REQUIRED>
```

Suponha que para atender à nova necessidade de saber a vigência do tipo de lançamento, a tabela precisou ser recriada e passou a ter a seguinte estrutura:

```
Create Table TP_LCTO (
cd_tp_lcto  int  not null,
dt_ini_validade datetime not null,
dt_fim_validade datetime not null,
desc_tp_lcto varchar(50) not null)
```

Este novo esquema de tabela pode ser mapeado para a seguinte DTD, que chamaremos de DTD-v.2:

```
<!ELEMENT TP_LCTO (desc_tp_lcto, dt_fim_validade?)>
<!ELEMENT desc_tp_lcto         (#PCDATA)>
<!ELEMENT dt_fim_validade    (#PCDATA)>
<!ATTLIST   TP_LCTO
  cd_tp_lcto            CDATA #REQUIRED
  dt_ini_validade       CDATA #IMPLIED>
```

Note que documentos gerados de acordo com a DTD-v.1 são também válidos de acordo com a DTD-v.2, visto que a evolução se dá pela definição de elementos e

atributos facultativos. Assim não há necessidade de fazer qualquer alteração nos documentos XML já existentes.

O exemplo acima mostrou o impacto da criação de novas colunas, tanto no modelo relacional, como na sua representação XML, onde a inclusão de novos elementos é implementada através da alteração da DTD com a definição de novos elementos facultativos, complexos ou não, e novos atributos opcionais. Na seqüência, analisaremos também o impacto dos demais tipos comuns de alterações, os quais foram apresentados no final da seção 3.1.

- Alteração do nome de uma coluna: Na representação XML, uma alteração de nome de coluna é implementada na DTD como união, representado pelo símbolo "|". Suponha que para o exemplo acima, a coluna desc_tp_lcto mudou de nome para desc_tipo_lancto. A nova definição da DTD com a utilização do operador de união para implementar a alteração de nome é:

```
<!ELEMENT TP_LCTO (desc_tp_lcto | desc_tipo_lancto, dt_fim_validade?)>
<!ELEMENT desc_tp_lcto    (#PCDATA)>
<!ELEMENT desc_tipo_lancto (#PCDATA)>
<!ELEMENT dt_fim_validade    (#PCDATA)>
<!ATTLIST   TP_LCTO
  cd_tp_lcto            CDATA #REQUIRED
  dt_ini_validade       CDATA #IMPLIED>
```

- Alteração do tamanho de coluna: No XML não há formatação limitando o tamanho das colunas, portanto não é necessária nenhuma alteração na DTD para atender uma alteração de tamanho de coluna.

- Exclusão de colunas: Na representação XML, a alteração do esquema para exclusão de colunas é implementada tornando os elementos e atributos facultativos. A alteração na DTD necessária para sua implementação é a definição dos elementos a serem excluídos sucedidos de "?", tal qual no exemplo acima, segundo a notação apresentada para o elemento dt_fim_validade ou, se forem atributos, a alteração da sua definição para #IMPLIED, segundo a notação apresentada para o atributo dt_ini_validade no exemplo acima.

Como apresentado na seção 3.1, boa parte das alterações de esquema para o modelo relacional exigem a recriação da tabela. Com a utilização do XML, os impactos destas manutenções seriam minimizados, através da utilização das definições de DTD apresentadas a pouco.

3.4 Orientação a Objetos e a Evolução de Esquemas

Técnicas para apoio à evolução de esquemas [17] foram bastante estudadas e pesquisadas na década passada, principalmente utilizando conceitos de orientação a objetos. Os principais desafios nesta área são:

. prover uma boa evolução de esquema, no qual o sistema deve implementar uma interface de programação simples e flexível de forma a permitir mudanças arbitrárias no esquema;

. preservar a consistência dos dados, pois os sistemas de vida muito longa estão sujeitos a muitas evoluções de esquemas, que podem introduzir erros no sistema;

. preservar a disponibilidade do sistema, pois os sistemas são compostos por muitos elementos interligados e acessados por diversos usuários. A execução de processos para implementação da evolução de esquemas podem causar a indisponibilidade destes dados;

. garantir um bom desempenho, sendo que os processos de evolução de esquema devem ser eficientes e otimizados.

Em linguagens de programação[18], o paradigma da orientação a objetos (OO) consolidou-se na última década por implementar de forma natural da idéia de "reaproveitamento de código". Reaproveitamento de código é um desafio antigo que, até então, era implementado através da criação de bibliotecas de código e modelos de trabalho. A orientação a objetos baseia-se em alguns conceitos, que descreveremos na seqüência.

. Classes: este termo em OO significa a unidade básica de todo o planejamento, de toda organização em objetos. Uma classe é um modelo para um

objeto. A partir da definição de uma classe podemos gerar diversos objetos. A isso denominamos instanciação. Como exemplo: a raça humana, na qual todos são da classe “ser humano". Apesar de serem da mesma classe, são indivíduos separados, com características próprias como, por exemplo, o nome e a data de nascimento.

. Atributos: As características que mudam de indivíduo para indivíduo, chamamos de "Atributos".

. Método: O método define o comportamento de um objeto. Métodos são blocos de código que utilizam os atributos do objeto para atingir seus objetivos. Às vezes estes códigos necessitam receber informações do mundo externo, em forma de parâmetros, para a execução de suas atividades. O método é o serviço que a classe oferece.

. Encapsulamento: O encapsulamento é o conceito de “esconder” a implementação de uma classe, definindo quais atributos e métodos são visíveis para outras classes e quais são privativos. Quando atributos são privativos, são necessários métodos de leitura e de gravação para que eles possam ser acessados por outros objetos. Os métodos e os atributos visíveis para outras classes definem a interface da classe.

. Herança: o conceito de herança determina que classes podem herdar características de outras, permitindo que comportamentos que já estão definidos em uma classe não precisem ser redefinidos em suas subclasses, bastando criar entre elas um relacionamento de herança. Resumindo, classes mais específicas herdam as características das mais genéricas. A classe de nível superior na associação de herança é chamada de superclasse.

. Polimorfismo: Polimorfismo ocorre quando classes distintas possuem métodos com a mesma interface. Outro tipo de Polimorfismo ocorre quando um método aparece mais de uma vez em uma classe, com o mesmo nome, mas com assinaturas diferentes. Assinatura é a identificação de um método, composta pelo nome do método e pelo número e tipo dos seus argumentos, incluindo aí os valores de retorno.

No modelo orientado a objetos, um esquema é composto de uma relação de classes que formam um grafo de heranças.

Em uma evolução de esquema é preciso manter a consistência estrutural do esquema. A consistência estrutural consiste da definição estática de classes, incluindo seus atributos e assinaturas de métodos, e da descrição de um grafo de herança, pela definição de relações de superclasses e subclasses no esquema. Para isto, é necessário seguir algumas definições, como unicidade de nomes, grafos de herança sem fechar ciclos, herança completa de atributos e métodos, etc.

O impacto da evolução de esquema em um banco de dados orientado a objetos pode ser minimizado se for possível definir uma subclasse da classe modificada contendo novos atributos ou atributos já existentes com tipo distinto. Porém, esta abordagem não pode ser aplicada sempre e, em alguns casos, a própria classe precisa ser modificada. Devido ao conceito de encapsulamento, alterações na implementação da classe não têm impacto sobre as demais classes. Porém, se a interface da classe for afetada pela modificação, o impacto nas demais classes pode ser grande.

Considerando a definição de que uma alteração deve ser refletida na hierarquia de classes e nos métodos preservando a consistência do esquema, os diversos bancos de dados orientados a objetos implementam diferentes estratégias para preservação da compatibilidade do esquema [17], conforme apresentado na Figura 14.

Uma manutenção resultante de uma evolução de esquema são as possíveis operações, como inclusão, alteração ou exclusão, para objetos básicos do modelo orientado a objetos, como atributos, métodos, classes e ligações de heranças.

Dentre as diferentes formas de implementação de alterações, temos a conversão imediata, na qual a alteração é aplicada de imediato no objeto e em todos os demais objetos impactados pela alteração, quer seja por herança, no caso das classes, ou pela alteração de uma definição de parâmetro, no caso dos métodos, obrigando a estrutura do banco de dados a ficar de conformidade com a nova definição de classe e mantendo a consistência do banco de dados.

Outro tipo é a conversão tardia, na qual a alteração é definida, mas só é implementada à medida que os objetos são utilizados. Para isto é necessário manter um histórico das mudanças de cada objeto até a primeira chamada do mesmo em memória para que a mudança de sua estrutura ocorra de acordo com toda a mudança do esquema, o qual ocorreu desde o último acesso.

Um terceiro tipo de conversão é o chamado *screening*, que aplica um tipo de conversão lógica pela definição de uma visão nas instâncias antigas, possibilitando um valor *default* para o novo atributo ou projetando para fora atributos excluídos.

Outra técnica para preservação de compatibilidade é o versionamento de esquema, sendo que em cada atualização de esquema, o esquema inteiro ou a classe afetada é copiada para uma nova versão.

BDOO	Conversão Imediata	Conversão Tardia	*Screening*	Versionamento de esquema	Versionamento de classes
GemStone [28]	SIM				
O2 [25]	SIM				
OTGEN [31]	SIM				
LispO2 [33]		SIM			
OBJM [34]		SIM			
Encore [26]			SIM		SIM
COCOON [29]			SIM		
ORION [27]			SIM	SIM	
Clamen [32]					SIM
CLOSQL [30]					SIM

Figura 14 – Bancos de Dados Orientados a Objetos e suas Estratégias para Preservação de Compatibilidade

As discussões apresentadas neste capítulo fazem considerar que o armazenamento de dados históricos no formato XML apresenta vantagens tanto em relação ao modelo relacional como ao modelo orientado a objetos. No próximo capítulo apresentaremos formas alternativas de implementação desta abordagem.

4 – ARMAZENAMENTO DE DADOS HISTÓRICOS EM XML

Neste capítulo apresentaremos as propostas de alguns SGBD´s relacionais para possibilitar o armazenamento de XML. Além disso, analisaremos a adequação da utilização de um esquema de armazenamento misto, mantendo os dados ativos em um SGBD Relacional e os dados históricos em um SGBD XML Nativo.

4.1 Suporte a XML em Bancos de Dados Relacionais

Os principais bancos de dados relacionais atualmente disponíveis no mercado oferecem recursos para a utilização de informações em formato XML. Porém, para serem gravadas, elas precisam sofrer alterações de forma que fiquem compatíveis com o modelo utilizado pelo banco de dados. Estas alterações fazem com que algumas propriedades do XML sejam reduzidas.

Os SGBD´s relacionais continuam investindo para melhorar o armazenamento de XML, devido a sua flexibilidade e devido ao fato do XML vir se tornando um padrão para troca de informações entre sistemas. O uso do XML facilita o esforço de desenvolvimento para quem transmite os dados e permite a adaptação do uso dos dados por quem os recebe. Para isto é preciso estar preparado para consultar, juntar e transformar os dados XML em dados relacionais.

Quanto à forma de armazenamento físico dos documentos XML pelos bancos de dados relacionais, podemos identificar três métodos[8]:

a) Não-Estruturado: Este método usa o tipo de dado BLOB (Binary Large Object) para armazenar um documento XML inteiro em uma coluna. Isto limita a capacidade para pesquisas no documento, mas ainda pode ser útil. Embora a estrutura XML esteja preservada, não é possível fazer pesquisas que envolvam também os dados relacionais.

b) Fragmentado: Neste caso, fragmentação refere-se à extração dos dados relevantes da estrutura hierárquica do XML, colocando-os em uma estrutura relacional plana. Ou seja, os dados XML são armazenados em colunas relacionais, mas a fragmentação do documento XML faz com que os relacionamentos hierárquicos entre os dados sejam perdidos. Existem diversas soluções para o mapeamento fragmentado de XML para relacional, mas a mais utilizada é a que faz o mapeamento direto dos elementos XML em colunas relacionais. Para ilustrar esta solução tomaremos por base o documento XML apresentado na Figura 11. A partir do elemento raiz <CONTA_CLIENTE> é definida a tabela CONTA_CLIENTE. Para os elementos <num_cc>, <cod_cli> e <nr_seq_titularidade> são criadas as colunas num_cc, cod_cli e nr_seq_titularidade respectivamente. Os valores repetidos de num_cc são apresentados em tuplas diferentes, com os demais atributos tendo o seu valor repetido. Ao listar as tuplas da tabela CONTA_CLIENTE temos:

num_cc	cod_cli	nr_seq_titularidade
19232221	123	1
17423525	123	1

O mapeamento de XML para relacional exige o conhecimento prévio da estrutura do documento XML. A recomposição do documento XML a partir da tabela relacional é, na maioria das vezes, complexa e custosa.

c) Estruturado: Os dados XML são armazenados em colunas relacionais preservando os relacionamentos hierárquicos entre os dados XML. Para isto, um documento XML pode ser mapeado em várias tabelas relacionais. Quanto maior a hierarquia do documento XML, maior a quantidade de tabelas geradas e maior a quantidade de junções das tabelas para recompor o documento XML. Sua principal vantagem é poder combinar sua consulta com dados relacionais e produzir documentos XML como resultado. Para ilustrar esta solução, também tomaremos por base o documento XML apresentado na Figura 11. A partir do elemento raiz <CONTA_CLIENTE> é definida a tabela CONTA_CLIENTE. Para o elemento <cliente> é

definida uma segunda tabela de nome cliente. Para estabelecer o relacionamento hierárquico entre as tabelas cliente e CONTA_CLIENTE é necessário criar uma coluna auxiliar, de valor auto-incrementável, que será chave primária na tabela cliente e chave estrangeira na tabela CONTA_CLIENTE. Os elementos <cod_cli> e <nr_seq_titularidade> são associados à tabela cliente através das colunas cod_cli e nr_seq_titularidade. O elemento num_cc é associado à tabela CONTA_CLIENTE através da coluna num_cc. Ao listar as tuplas da tabela cliente temos:

clientekey	cod_cli	nr_seq_titularidade
1	123	1

E ao listar as tuplas da tabela CONTA_CLIENTE temos:

num_cc	clientekey
19232221	1
17423525	1

Uma vez definidas as tabelas e seus relacionamentos que definem a hierarquia do documento XML, podem ser criados índices para as colunas auxiliares, neste caso clientekey, para melhorar o desempenho das junções das tabelas, necessárias para recompor o documento XML.

Os quatro principais SGBD´s do mercado (IBM - DB2 v. 8.1[22], Oracle 10g[21], Sybase ASE v. 12.5[20] e Microsoft SQL Server 2000[15]) implementam diferentes funcionalidades para XML, uns extremamente ricos e outros mais limitados. Para se ter uma idéia de como estes gerenciadores de bancos de dados estão com relação às principais funcionalidades do XML, apresentamos um quadro resumo na figura 15 com uma posição de Abril de 2004 das respectivas versões comercializadas[8]. Para os quatro principais SGBD´s relacionais são apresentadas sua posição com relação ao armazenamento de XML fragmentado, armazenamento de XML estruturado, consultas concatenadas com dados relacionais, suporte Xquery e suporte Sql/xml.

SGBD	Armazenamento XML Fragmentado	Armazenamento XML Estruturado	Consultas Concatenadas com Dados Relacionais	Suporte Xquery	Suporte Sql/xml
DB2 v. 8.1	Sim	Sim	Sim	Não	Sim
SQL Server 2000	Sim	Não	Não	Não	Sim
Oracle 10g	Sim	Sim	Sim	Sim	Sim
Sybase ASE 12.5	Sim	Não	Sim	Não	Sim

Figura 15 - Quadro Resumo Comparativo dos SGBD´s Relacionais com relação ao XML

Com base na Figura 15, podemos identificar que o Oracle 10g é o mais rico em funcionalidades para XML, seguido de perto pelo DB2. Mas como em todo mercado competitivo, a Microsoft já anunciou sua nova versão do SQL Server 2005 [19] com uma série de novas funcionalidades para o XML.

Os SGBD´s relacionais que armazenam documentos XML de forma estruturada se apresentam como SGBD´s XML Nativos, porém isto pode ser contestado.

Apesar de toda evolução para o armazenamento estruturado de XML, os SGBD´s relacionais não utilizam a DTD para definir os documentos XML. Como é necessário conhecer previamente a estrutura do documento XML, é preciso definir um esquema de documento XML no SGBD relacional, o qual é posteriormente mapeado para uma tabela ou *n* tabelas relacionais. Esse mapeamento faz com que algumas propriedades do XML sejam reduzidas ou então não permitidas. Um dos itens que pode ser contestado é a impossibilidade de armazenamento de elementos multivalorados pelos SGBD´s relacionais. Quando um elemento multivalorado é definido em um esquema XML de um SGBD relacional, durante sua implementação ocorre o desmembrado em *n* tabelas hierarquicamente ligadas. Além disso, outras possibilidades de definições em uma DTD não são comportadas pelo esquema de documento XML dos SGBD´s relacionais como, por exemplo, operadores de união.

4.2 Evolução de esquema XML em Bancos de Dados Relacionais

Para armazenar um documento XML em um SGBD relacional é feita uma associação de cada elemento complexo XML com uma tabela Relacional. Os principais tipos de evolução de esquema incluem mudanças tais como a criação ou exclusão de um elemento ou atributo, alteração no formato ou tamanho de um elemento e a criação ou exclusão de um elemento complexo. Por mapear elementos XML em tabelas relacionais, os documentos XML armazenados em SGBD´s Relacionais acabam encontrando as mesmas dificuldades para as evoluções de esquema das tabelas relacionais, estas descritas na seção 3.1.

A seguir apresentamos algumas formas apresentadas pelos SGBD´s Relacionais para implementação da evolução de esquema de documentos XML [23] neles armazenados.

4.2.1 Evolução através da atribuição de uma nova versão

A implementação desta evolução conduz a preservação dos dados antigos onde estavam antes armazenados e com o mesmo esquema, e a definição um novo esquema com as novas definições. Este novo esquema pode ser incluído na mesma *collection* se for necessário acessar os dados em uma mesma consulta, ou em uma nova *collection*. *Collection* nada mais é do que uma coleção contendo todos os documentos XML. É preciso identificar o novo esquema por um nome diferente. O principal problema desta forma de evolução são as consultas simultâneas aos dois diferentes esquemas, sendo que as consultas pré-definidas devem ser reescritas para consultar o novo esquema. Ao utilizar a evolução através da atribuição de uma nova versão, considera-se que os dados do esquema anterior serão consultados separadamente dos dados do esquema evoluído, sendo necessária a criação e/ou alteração das consultas. Este tipo de evolução se faz necessário quando as características dos dois esquemas mudaram muito e é preciso tratar as informações separadamente.

4.2.2 Evolução através da extensão do esquema existente sem revalidação

Por associar um elemento a uma coluna de uma tabela relacional, um SGBD relacional possibilita a implementação de regras para este elemento. As coleções de esquemas em XML permitem a evolução de um esquema existente através da inclusão da declaração de um novo elemento ou atributo. Uma vez que a inclusão deste novo elemento ou atributo não torne o atual esquema mais restritivo, não se faz necessária sua revalidação. A revalidação se aplica quando o esquema existente possuir regras que se aplicam a estes novos elementos. Exemplo: ao criar um elemento denominado sigla da UF e associar este elemento a uma regra de validação com base em uma lista de UF´s válidas, somente pode ser atribuído o valor a um elemento se seu conteúdo estiver dentre os 26 válidos para UF. Caso existam elementos cuja sigla de UF não esteja entre os 26 válidos e a alteração de esquema associar estes elementos à regra, ocorre um erro no processo. Quando se faz necessária a criação de elementos associados às regras de validação, é preciso criar um novo esquema, que é o próximo tipo de evolução a ser apresentado.

4.2.3 Evolução através da alteração do esquema existente

Esta evolução ocorre quando é necessário alterar um esquema existente e é exigida a transformação ou revalidação do esquema. Como a alteração de esquema exige a aplicação de alguma regra, se faz necessária a criação de um novo esquema com as alterações e a execução da transformação do dado para o novo esquema. Este tipo de evolução implica na execução de um processo para efetuar a transformação dos elementos do esquema anterior para o novo ou para revalidar alguns elementos dentro do esquema alterado.

4.3 SGBD XML Nativo

Tendo em vista que as atuais propostas de extensão dos SGBD´s relacionais para suporte a XML não contemplam de forma adequada a evolução de esquema, nesta seção apresentaremos um estudo de caso de utilização de um SGBD XML nativo para o armazenamento de dados históricos.

O objetivo é analisar a adequação de um esquema de armazenamento misto, no qual dados ativos são mantidos em um SGBD Relacional e os dados históricos em um SGBD XML nativo. Utilizamos como exemplo a aplicação bancária descrita na seção 3.1. Para o estudo foram escolhidos o SGBD relacional MS SQL Server 2000 [15], no qual foi criada uma base de dados fictícia de 200 Mbytes, e o SGBD XML nativo Tamino XML Server v.4.1.4 [14]. Ambos foram instalados em sua versão de demonstração.

Existem diversos projetos de implementação de SGBD´s que utilizam o XML como seu modelo de dados, como o Galax [13] e Rainbow Core[12]. Porém, dentre eles, o Tamino é o mais completo, sendo também o primeiro a ser comercializado. Nas próximas seções apresentaremos como foi realizada a conversão e carga dos dados gerados no SGBD relacional para o Tamino, utilizando as ferramentas existentes no sistema.

4.3.1 Tamino v.4.1.4

O Tamino XML Server é um banco de dados diferenciado por utilizar a XML como principal formato para gravar e apresentar os dados. As principais vantagens apresentadas pelo Tamino são: otimização do uso de documentos XML, facilidade na comunicação via Internet, armazenamento dados XML e não-XML e acesso a esses dados por sistemas e aplicações externas.

Através da instalação do Tamino foram disponibilizadas algumas ferramentas para sua utilização, as quais são apresentadas a seguir.

4.3.2 Tamino Manager

O Tamino Manager é uma ferramenta com interface gráfica baseada em browser para administração dos bancos de dados do Tamino. Com esta ferramenta podemos criar, configurar e excluir um banco de dados, ativar e desativar um banco de dados, definir usuários, executar funções de backup e restauração de banco de dados e outras atividades similares.

Utilizando o Tamino Manager, criamos e ativamos um banco de dados chamado ccdb001 com tamanho de 20 Mb, o qual utilizamos para a implementação do nosso exemplo bancário.

4.3.3 Interface Interativa do Tamino

A Interface Interativa do Tamino é uma interface de usuário através da qual é possível enviar requisições para o Servidor Tamino para efetuar a carga e recuperação de objetos XML e parte deles, através da linguagem de consulta Xquery, apresentando o resultado em uma segunda janela. Para o novo usuário Tamino, a interface é uma ferramenta conveniente para uma interação básica com o Tamino Server.

Esta ferramenta foi bastante útil para fazer a carga dos Documentos XML no Tamino Server e para escrever e executar as consultas em Xquery.

4.3.4 Editor de Esquema do Tamino

O Editor de Esquema do Tamino é uma ferramenta através da qual é possível definir esquemas para o Tamino. Um esquema nada mais é do que a definição da estrutura do documento XML e também pode ser gerado a partir de uma DTD. Um esquema é descrito através de uma linguagem de definição de esquema específico do Tamino, a qual é baseada nos padrões de esquema XML. O Editor de Esquema permite definir o esquema como uma árvore gráfica e automaticamente cria a definição do

esquema na linguagem de esquema do sistema. Após definido um esquema sintaticamente correto, é possível carregar a descrição do esquema no Servidor. Só é possível carregar um documento XML no Servidor se tiver sido definido um esquema para o documento.

O Editor de Esquema possui algumas características específicas para a formatação de um esquema. Primeiramente é possível acessar diversos documentos XML a partir de uma mesma consulta desde que os mesmos estejam definidos em uma mesma *collection*. Devido a esta característica, definimos todos os documentos XML derivados das Tabelas do nosso exemplo bancário como elementos complexos em uma única *collection*, para podermos elaborar as consultas com junções de forma equivalente nos dois modelos.

Outra característica encontrada que gerou dúvida é quando definir um atributo, representado no editor de esquema por  , e quando definir um elemento, representado por  . Pelos exemplos apresentados, pudemos identificar que são definidos atributos para os elementos que fazem parte da chave, tal qual os campos que compõe uma chave primária de uma tabela relacional. Os elementos complexos são representados por  . Definimos um elemento complexo para cada uma das tabelas relacionais. Procurando seguir os exemplos, definimos atributos para os campos que fazem parte da chave. Todo os demais campos que não são chave foram definidos como elementos dentro de uma *sequence*  do elemento complexo. A *sequence* é obrigatória para relacionar os elementos que compõem o elemento complexo. Em uma DTD, uma *sequence* é definida relacionando os nomes dos sub-elementos junto ao elemento complexo. A seguir é apresentado um exemplo do trecho de uma DTD com a definição de uma *sequence* do elemento complexo cliente:

```
<!ELEMENT cliente (nome, cpf?|cnpj?, conta_corrente*)>
```

Um elemento pode ser definido como um elemento global se mais de um elemento complexo fizer referência a ele. Neste caso o elemento global é referenciado

como um elemento derivado, representado por  , em mais do que um elemento complexo.

Para cada tabela relacional foi definido um elemento complexo contendo como atributos os elementos da chave primária e os demais elementos definidos como elemento. Para caracterizar o elemento complexo diferenciando-o dos demais é preciso definir um "*doctype*", representado por  , com o mesmo nome do elemento complexo. Resumindo, para cada tabela relacional tem-se um elemento complexo com seu respectivo "*doctype*".

Como é possível definir elementos multivalorados no esquema XML, algumas tabelas com poucos atributos e poucas ocorrências dos elementos de relacionamento foram desnormalizadas, utilizando o método *Nesting-based Translation* apresentado na seção 2.3.3. Assim, no nosso exemplo bancário, a tabela associativa CONTA_CLIENTE, que faz a associação entre a tabela de CONTA_CORRENTE e CLIENTE, identificando qual a titularidade do cliente na conta corrente foi eliminada com a criação do elemento multivalorado cod_cli na tabela de CONTA_CORRENTE. Desta forma passou a existir uma relação direta de conta_corrente com cliente e a ocorrência do elemento cod_cli multivalorado identifica qual a titularidade do cliente na conta corrente.

Outra forma de representação utilizada que é exclusiva do XML é o "*choice*" representado por  . O *choice* equivale à construção "|" em DTDs. Com o *choice* é possível definir vários elementos / elementos complexos para utilizar um ou outro dentro do documento XML. Esta é uma das estruturas que possibilita a implementação de alterações de esquema de maneira direta no XML, as quais exigem a recriação das tabelas no modelo relacional.

A estrutura final da *collection* do nosso exemplo bancário está ilustrada na figura 16.

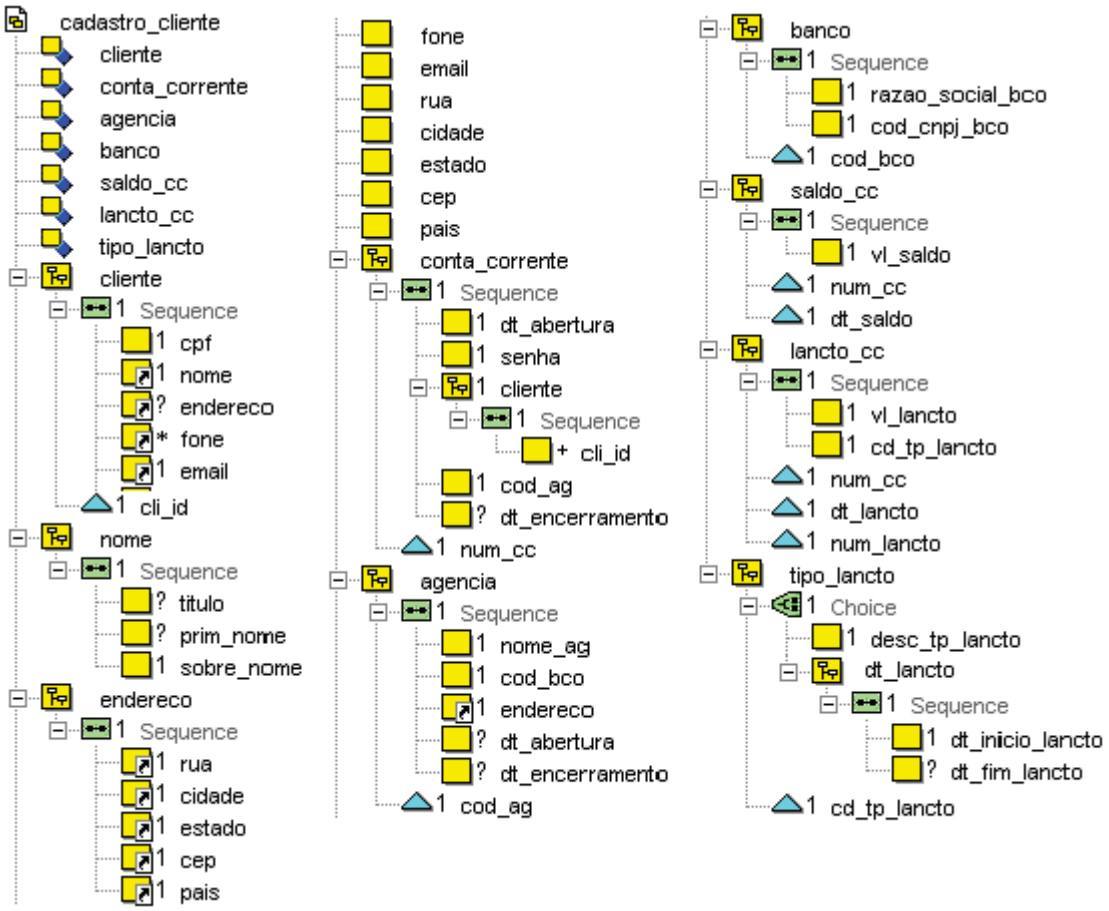

Figura 16 - Estrutura Representando o Exemplo Bancário implementado pelo Editor de Esquema do Tamino

Note na estrutura representada na Figura 16 que a tabela CONTA_CLIENTE não foi implementada e ela foi substituída pelo elemento multivalorado cli_id do elemento complexo conta_corrente. A ordem do elemento cod_cli multivalorado identifica qual a titularidade do cliente na conta corrente. Se o cliente está na primeira ocorrência é o primeiro titular e assim por diante.

Um exemplo de alteração de esquema que, para uma tabela relacional exigiu a recriação da Tabela, neste modelo foi implementado como "*choice*"  é a tipo_lancto. O que levou a alteração da estrutura de tipo_lancto com a criação dos elementos dt_inicio_lancto e dt_fim_lancto foi a criação da CPMF pelo Governo Federal. Para os lançamentos de CPMF na conta corrente existe um cd_tp_lancto em tipo_lancto cadastrado. Porém, a CPMF foi criada em um determinado ano e deixou de vigorar em outro, sendo que na tabela de tipo de lançamento foi preciso armazenar a informação de data para garantir que não fossem feitos lançamentos de CPMF fora do seu período

de vigência. Já os demais tipos de lançamentos como depósito, saque, depósito em cheques, etc, não precisavam do controle de vigência através desta data.

É possível definir índices para elementos e atributos de pesquisa.

Após a definição da *collection* no Editor de Esquema do Tamino, a mesma foi criada no Tamino Server.

4.3.5 Carga dos Dados no Tamino

Para que os dados sejam carregados no Tamino Server, eles precisam estar no formato XML e o documento deve ser válido de acordo com o esquema definido pela *collection* previamente criada. Para isto foi criada uma consulta em SQL para gerar os dados no formato XML a partir das tabelas criadas no SQL Server 2000. Foi criada uma *procedure* para cada tabela. Para exemplificar seu formato, apresentamos na Figura 17 a *procedure* executada para a tabela saldo_cc e na Figura 18 o documento resultado:

```
1.  create procedure sp_dba_gera_xml
2.  as
3.  declare @dt_saldo datetime, @num_cc int, @vl_saldo float, @linha char(60),
4.          declare cur1 cursor for
5.          select num_cc, dt_saldo,  vl_saldo
6.          from saldo_cc
7.          order by  num_cc, dt_saldo
8.  set nocount on
9.  open cur1
10. fetch cur1 into @num_cc , @dt_saldo , @vl_saldo
11. while @@sqlstatus = 0
12. begin
13.    select  @linha =  "<saldo_cc num_cc=" +  convert (char(5),@num_cc) +
14.                            "  dt_saldo=" + convert(char(10) , @dt_saldo, 103)  +  " >"
15.    print @linha
16.    select  @linha =  "<vl_saldo> " +  convert (char(12),@vl_saldo) +  " </vl_saldo>"
17.    print @linha
18.    print "</saldo_cc>"
19.    fetch cur1 into @num_cc , @dt_saldo , @vl_saldo
20. end
21. close cur1
22. return 0
```

Figura 17 - *Procedure* para Geração dos Documentos XML a partir das Tabelas Relacionais

A consulta efetuada na *procedure* apresentada na Figura 17 para selecionar os dados é relativamente simples. Como é preciso formatar a saída, foi necessário utilizar um *cursor* para processar linha a linha do resultado da consulta. O *cursor* (linha 4) executa um *select* na tabela saldo_cc (linha 5) ordenado por num_cc e vl_saldo (linha 7). Para cada linha retornada com o conteúdo de num_cc, dt_saldo e vl_saldo (linha 10) são formatadas as *tags* XML e impressas (linhas 13 a 18), gerando o documento XML apresentado na Figura 18.

```
Select * from saldo_cc    (Lista o conteúdo da tabela)

num_cc          dt_saldo          vl_saldo
----------      ----------------  ----------
10001           "2001-12-12"      1013
10001           "2001-12-13"      1225
10001           "2001-12-14"      1238
20101           "2001-12-12"      123
20101           "2001-12-13"      125
...

sp_dba_gera_xml   (execução da procedure, apresentando as primeiras linhas da saída:)

<saldo_cc num_cc="10001" dt_saldo="2001-12-12">
<vl_saldo>1013</vl_saldo>
</saldo_cc>
<saldo_cc num_cc="10001" dt_saldo="2001-12-13">
<vl_saldo>1225</vl_saldo>
</saldo_cc>
<saldo_cc num_cc="10001" dt_saldo="2001-12-14">
<vl_saldo>1238</vl_saldo>
</saldo_cc>
<saldo_cc num_cc="20101" dt_saldo="2001-12-12">
<vl_saldo>123</vl_saldo>
</saldo_cc>
<saldo_cc num_cc="20101" dt_saldo="2001-12-13">
<vl_saldo>125</vl_saldo>
</saldo_cc>
...
```

Figura 18 - Documento XML gerado a partir das Tabelas Relacionais

A partir dos documentos XML gerados foram feitas cargas pela Interface Interativa do Tamino informando a *collection* previamente cadastrada no Tamino Server e o arquivo XML de carga.

4.4 Consultas

Nesta seção apresentaremos como uma consulta que é bastante freqüente nos dados históricos de uma aplicação bancária pode ser realizada no SGBD Tamino.

A linguagem de consulta implementada pelo Tamino é o XQuery [3]. Uma consulta é executada em uma *collection* no Tamino Server através da Interface Interativa do Tamino.

Existem algumas características que não foram apresentadas na seção 2.1.4 de XQuery e se aplicam para o acesso aos documentos XML armazenado no Tamino.

Na cláusula *for*, a expressão de caminho é subsituída pelo nome do documento precedido pela palavra chave *input()*.

Ex. `for $cc in input()/conta_corrente`

Para fazer a junção de documentos basta relacionar os vários documentos na cláusula for separando-os por "," e fazer a associação dos elementos/atributos na cláusula where.

Ex.
```
for $cc in input()/conta_corrente,
$ag in input()/agencia
where $cc/@num_cc="20101" and
$ag/@cod_ag=$cc/cod_ag
```

Os atributos são precedidos de um @ no nome. Ex. `$ag/@cod_ag`. Os elementos são referenciados pelo caminho completo: Ex. `$cl/endereco/rua`.

Para efetuar consultas com diversos níveis de encaixamento, basta escrever outro *for* dentro da cláusula *return* da consulta de menor nível de encaixamento e fazer a associação dos elementos/atributos na cláusula *where*.

Ex.
```
for $s in input()/saldo_cc
where $s/@num_cc="10002"
return <saldo> {$s/@dt_saldo} {$s/vl_saldo}
  { for $l in input()/lancto_cc,
   where $l/@dt_lancto = $s/@dt_saldo and
         $l/@num_cc = $s/@num_cc
   return <lancamento> {$l/@num_lancto}
             {$l/vl_lancto}
      </lancamento> }
  </saldo>
```

Apresentaremos a seguir duas consultas completas de uma conta corrente, uma em SQL acessando um SGBD relacional e outra XQuery acessando o Tamino.

A Figura 19 apresenta a consulta completa em SQL de uma conta corrente com seus saldos e lançamentos, conforme tabelas do modelo apresentado na Figura 12, e a Figura 20 apresenta o resultado da consulta.

```
1. select cc.num_cc, ag.cod_bco, ag.nome_ag, cl.nome,
2.       cl.endereço, s.dt_saldo, s.vl_saldo,
3.       l.num_lancto, tl.desc_tp_lancto, l.vl_lancto
4. from conta_corrente cc, agencia ag, cliente_conta clc,
5.     cliente cl, saldo_cc s, lancto_cc l, tipo_lancto tl
6. where cc.num_cc="10002" and
7.    ag.cod_ag=cc.cod_ag and
8.    clc.num_cc = cc.num_cc and
9.    cl.cli_id = clc.cli_id and
10.   s.dt_saldo > "2001-12-10" and
11.   s.dt_saldo < "2001-12-15" and
12.   s.num_cc = cc.num_cc and
13.   l.dt_lancto *= s.dt_saldo and
14.   l.num_cc *= s.num_cc and
15.   t.cd_tp_lancto = l.cd_tp_lancto
16. order by cc.num_cc, s.dt_saldo, l.num_lancto
```

Figura 19 - Consulta em SQL de uma conta corrente, seus saldos e lançamentos

```
cc.num_cc           ag.cod_bco     ag.nome_ag      cl.nome
cl.endereço                s.dt_saldo      s.vl_saldo
l.num_lancto        tl.desc_tp_lancto       l.vl_lancto
----------          ------------    -------------- -------------
--------------------        ------------    -------------
-------------       -------------------     --------------
10002               001             Agência 1       Burke Bates
130 West 5th Avenue         "2001-12-12"    223
001                 deposito                2
10002               001             Agência 1       Burke Bates
130 West 5th Avenue         "2001-12-13"    225
001                 deposito                6
10002               001             Agência 1       Burke Bates
130 West 5th Avenue         "2001-12-13"    225
002                 saque                   3
10002               001             Agência 1       Burke Bates
130 West 5th Avenue         "2001-12-14"    228
```

Figura 20 - Resultado da Consulta em SQL de uma conta corrente

A consulta em SQL faz a junção das tabelas conta_corrente, agencia, cliente_conta, cliente, saldo_cc, lancto_cc e tipo_lancto para apresentar as colunas destas tabelas que compõem um extrato de conta corrente.

A Figura 21 apresenta a consulta completa em XQuery de uma conta corrente com seus saldos e lançamentos, conforme esquema apresentado na Figura 16, e a Figura 22 apresenta o resultado da consulta.

```
for $cc in input()/conta_corrente,
   $ag in input()/agencia,
   $cl in input()/cliente
where $cc/@num_cc="10002" and
   $ag/@cod_ag=$cc/cod_ag and
   $cl/@cli_id=$cc/cliente/cli_id
return <extrato> {$cc/@num_cc} {$ag/cod_bco}
    {$ag/nome_ag}  {$cl/nome} {$cl/endereco}
  { for $s in input()/saldo_cc
    where    $s/@dt_saldo > "2001-12-10" and
             $s/@dt_saldo < "2001-12-15" and
             $s/@num_cc=$cc/@num_cc
    return <saldo> {$s/@dt_saldo} {$s/vl_saldo}
      { for $l in input()/lancto_cc,
            $t in input()/tipo_lancto
       where $l/@dt_lancto = $s/@dt_saldo and
             $l/@num_cc = $s/@num_cc and
             $t/@cd_tp_lancto = $l/cd_tp_lancto
       return <lancamento> {$l/@num_lancto}
                  {$t/desc_tp_lancto}
                  {$l/vl_lancto}
          </lancamento> }
      </saldo> }
  </extrato>
```

Figura 21 - Consulta em XQuery de uma conta corrente, seus saldos e lançamentos

A consulta em XQuery utiliza diversos níveis de encaixamento para fazer a junção das tabelas, apresentando os saldos ordenados por data e agrupando os lançamentos de um determinado saldo.

Observe que a visualização dos dados apresentados no formato XML na Figura 22, apresentada a seguir, é bem mais clara, comparada aos dados relacionais apresentados na Figura 20, na qual os valores das colunas `num_cc`, `cod_bco`, `nome_ag`, `nome` e `endereço` aparecem várias vezes repetidos.

```
<extrato num_cc="10002">
    <cod_bco>001</cod_bco>
    <nome_ag>Agência 1</nome_ag>
  - <nome>
      <prim_nome>Burke</prim_nome>
      <sobre_nome>Bates</sobre_nome>
    </nome>
  - <endereco>
      <rua>130 West 5th Avenue</rua>
      <cidade>Monterey</cidade>
      <estado>California</estado>
      <cep>10153</cep>
      <pais>USA</pais>
    </endereco>
  - <saldo dt_saldo="2001-12-12">
      <vl_saldo>223</vl_saldo>
    - <lancamento num_lancto="001">
        <desc_tp_lancto>deposito</desc_tp_lancto>
        <vl_lancto>2</vl_lancto>
      </lancamento>
    </saldo>
  - <saldo dt_saldo="2001-12-13">
      <vl_saldo>225</vl_saldo>
    - <lancamento num_lancto="001">
        <desc_tp_lancto>deposito</desc_tp_lancto>
        <vl_lancto>6</vl_lancto>
      </lancamento>
    - <lancamento num_lancto="002">
        <desc_tp_lancto>saque</desc_tp_lancto>
        <vl_lancto>3</vl_lancto>
      </lancamento>
    </saldo>
  - <saldo dt_saldo="2001-12-14">
      <vl_saldo>228</vl_saldo>
    </saldo>
  </extrato>
```

Figura 22 - XML Resultado da Consulta em XQuery de uma conta corrente

4.5 Avaliação da Estrutura Física do Tamino

Em uma primeira análise é possível imaginar que o espaço utilizado para o armazenamento de um documento XML seja muito maior do que uma tabela relacional, principalmente devido ao uso das *tags* para identificação de cada elemento. Analisando o espaço utilizado pelo Tamino para armazenar os documentos XML, identificamos que este espaço era bem menor do que o tamanho do arquivo contendo o documento XML utilizado para carga. Depois, conhecendo mais detalhes da forma de armazenamento do Tamino, descobrimos que o Tamino Server utiliza a estrutura física do banco de dados Adabas [16], um SGBD que utiliza o modelo de rede e armazena os dados de forma comprimida. Em contrapartida, os SGBD´s mapeiam os documentos XML em tabelas relacionais, quando armazenados de forma estruturada. Devido a este mapeamento, os dados XML também são armazenados em um espaço menor, pois os mesmo são armazenados sem as *tags*. Porém, os SGBD´s relacionais não armazenam os dados de forma comprimida. Nas próximas seções procuraremos estabelecer uma associação entre o Tamino e as principais características do Adabas.

4.5.1 Banco de Dados Adabas

O Adabas surgiu na década de 80 tendo como principal característica a flexibilidade na implementação de modelos, pois seu modelo é em rede e até então os principais bancos de dados eram hierárquicos. Embora o modelo em rede seja anterior ao modelo relacional, sua implementação no Adabas apresenta diversas vantagens sobre o relacional, quando adaptado para dar suporte a XML.

A primeira é o esquema de compressão de dados. O Adabas tem compressão em dois níveis: de Campo e de Registro. Em nível de campo, cada campo tem um byte de tamanho e só são armazenados os bytes efetivamente preenchidos. Em nível de registro, a compressão é feita quando se tem dois ou mais campos sem valor e o controle sobre estes campos é feita pelo contador de campos nulos (EFC – *Empty Field*

Counter). Desta forma, é possível implementar diversas visões em um mesmo arquivo físico sem perda de espaço, pois quando os campos de uma determinada visão estiverem preenchidos, os demais campos das outras visões naquele registro estarão suprimidos pelo contador de campos nulos.

Outra vantagem do Adabas, que não existe nos bancos de dados relacionais, é a implementação de campos multivalorados, conhecidos como campos múltiplos e estruturas multivaloradas, conhecidas como grupos periódicos. Para cada campo múltiplo / grupo periódico existe um campo contendo a quantidade de ocorrências da respectiva estrutura multivalorada. Pode-se definir índices para qualquer campo múltiplo ou campo do grupo periódico. Pode-se definir um campo múltiplo dentro de um grupo periódico, mas não se pode definir um grupo periódico dentro de um outro grupo periódico. Assim, no Adabas é possível implementar algumas estruturas desnormalizadas, com possível ganho de desempenho de consultas.

Um índice para o Adabas pode ser composto de apenas um campo, conhecido como descritor, de parte de um campo, conhecido como sub-descritor ou de um conjunto de campos, conhecido como superdescritor. A estrutura de índices é implementada em uma estrutura separada da estrutura de dados e é conhecida como *Associator*.

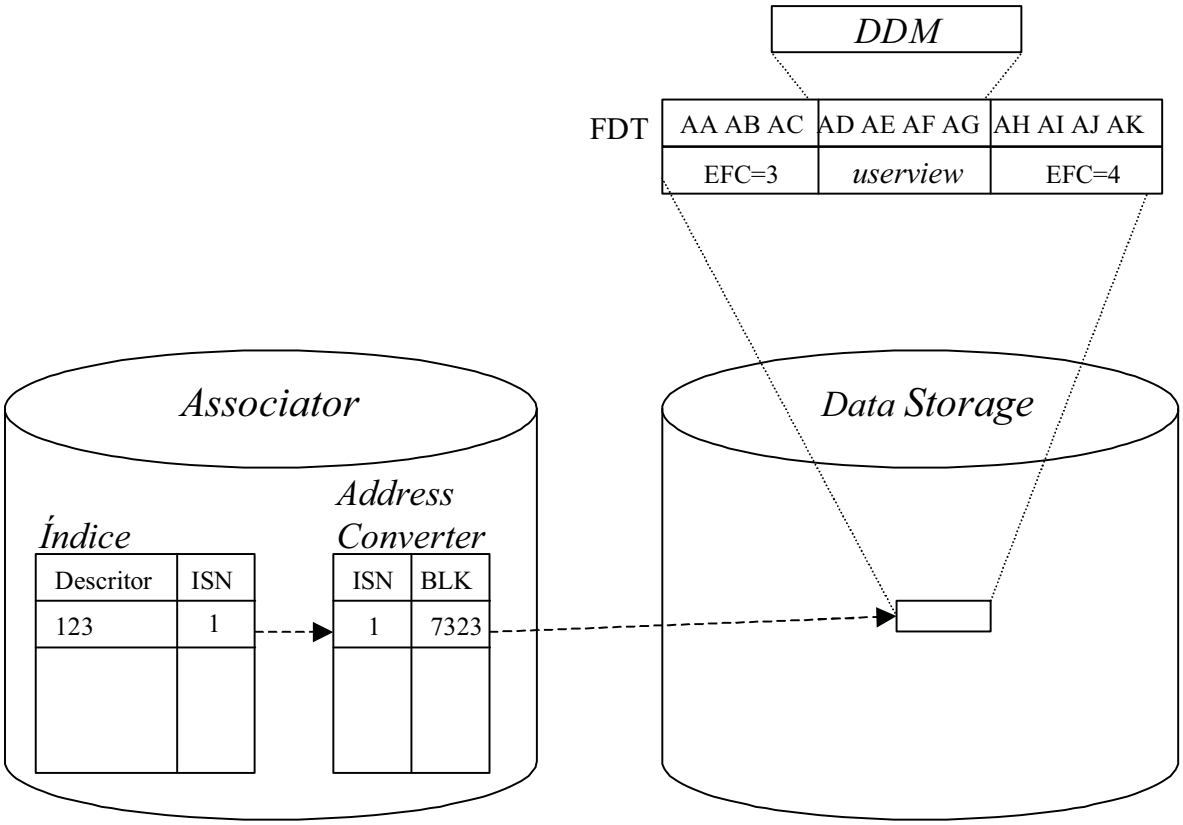

Figura 23 - Estrutura do Adabas

Cada registro de um arquivo Adabas tem um campo como identificador conhecido como número de seqüência interno (ISN – *Internal Sequence Number*). Este número identifica um registro na área de dados (*Data Storage*) e é utilizado dentro da estrutura de índice como ponteiro para a estrutura de dados. Para se chegar fisicamente em um registro na área de Dados a partir de um índice é preciso passar por um conversor de endereços (*Address Converter*) que relaciona o número do ISN ao endereço do bloco físico da estrutura de dados onde está o registro.

O nome de campo no Adabas possui apenas duas letras e é conhecido como *short name*. A estrutura física de um arquivo Adabas possui um *layout* pré-definido conhecido como FDT (*File Description Table*). Na linguagem de consulta do Adabas, para acessar um arquivo é preciso ter uma DDM (*Data Definition Module*), também conhecida como *userview*, cuja função é associar o nome externo de um campo, como é visto dentro dos programas, ao *short name* definido na FDT.

Note na Figura 23 que a *userview* é formada pelos campos AD, AE, AF e AG. Estes campos são carregados a partir registro armazenado no *Data Storage*, posicionados com base na FDT, deslocando os campos conforme o valor do contador de campos nulos (EFC=3). Com isto, os outros campos da FDT (AA, AB, AC, AH, AI, AJ, AK), utilizados por outras *userviews* e que são nulos para este registro, ocupam apenas o espaço do tamanho do EFC dentro do *Data Storage*.

4.5.2 Associação do Banco de Dados Adabas com o Tamino

Uma vez apresentadas as principais características do Adabas, podemos agora fazer uma associação de suas características com as características do Tamino.

Inicialmente podemos associar uma *collection* do Tamino a uma FDT (*File Description Table*) do Adabas. A *collection* nada mais é do que o *layout* dos diversos elementos complexos do documento XML, na qual cada elemento é associado a um tipo de estrutura de campos no Adabas, da seguinte forma:

- Elementos multivalorados do XML são associados a campos múltiplos do Adabas
- Elementos complexos multivalorados são associados a campos periódicos do Adabas
- Elementos complexos são associados a itens de grupo.
- Atributos e elementos do XML são associados a itens elementares do Adabas.
- Índices são implementados através de campos descritores e super descritores do Adabas.

As estruturas *choice* do XML são associadas a itens de grupos facultativos, equivalentes às *userviews*, nas quais os itens de grupos não preenchidos são suprimidos pela utilização do contador de campos nulos (EFC).

A Figura 24 apresenta a DDM Adabas do elemento conta_corrente apresentado no modelo XML da Figura 16, associando o tipo de estrutura de campos do Adabas com o tipo de elemento em XML.

```
Em XML                          DDM                          Em Adabas
------------  ------------------------------------ --------------
Elem.complexo  1 aa  conta_corrente                  ítem de grupo
Elem.simples   2 ab  dt_abertura       date          ítem elementar
Elem.simples   2 ac  senha             char(25)      ítem elementar
Elem.simples   2 ad  cod_ag            integer       ítem elementar
Elem.simples   2 ae  dt_encerramento   date          ítem elementar
Elem.complexo  1 af  cliente                         item de grupo
Elem.multival. 2 ag  cli_id            integer   MU() campo múltiplo
Atributo       1 ah  num_cc            integer   DE   ítem elementar
               | |   |                 |         |
               | |   |                 |         |
               | |   |                 |         Tipo: Múltiplo,
               | |   |                 Formato         Periódico,
               | |   Nome Externo                      Descritor
               | Short Name
              Nível
```

Figura 24 - DDM Adabas do Elemento conta_corrente

Uma vez tendo definido o *layout* do que vai ser armazenado, no caso do Tamino, a *collection*, não é necessário armazenar fisicamente as *tags* do documento XML. As *tags* são montadas no momento da recuperação e apresentação dos dados. Internamente a *collection* possui uma associação física com a FDT do Adabas. A associação de cada elemento da *collection* é feita pelo posicionamento físico do campo dentro do registro. Por exemplo, suponha que o quinto elemento de uma *collection* é o nome, então para o quinto campo do arquivo Adabas é montada a tag: <nome> campo </nome>. Além de armazenar fisicamente os dados sem as *tags*, o espaço utilizado também é reduzido devido ao esquema de compressão do Adabas já mencionado anteriormente. Estima-se um ganho médio de espaço em torno de 30%.

As consultas em Xquery puderam ser facilmente implementadas no Tamino, porém não foi possível realizar uma comparação de desempenho entre os bancos de dados devido à limitação de tamanho nas versões de demonstração.

Um estudo mais detalhado da adequação das estruturas de índice implementadas no Tamino para o modelo XML está fora do escopo deste trabalho.

4.6 Evolução de Esquema no Tamino

Um passo muito importante do uso do XML para armazenamento de dados históricos é o mapeamento do modelo Relacional para um esquema XML. Uma vez que os dados históricos que serão armazenados em XML derivaram de uma estrutura relacional, é possível tirar vantagens da nova estrutura de dados semi-estruturados XML, principalmente através da utilização de elementos multivalorados e junção de tabelas relacionadas, características que não são possíveis de implementar de forma direta em SGBD´s relacionais, mesmo estes armazenando documentos XML na forma estruturada. Além disso, com a utilização do Tamino para o armazenamento de XML, ocorre uma simplificação da manutenção de dados em caso de evolução de esquema, ao contrário do que ocorre com os SGBD´s relacionais, conforme apresentado na seção 4.2.

Os principais tipos de evolução de esquema incluem mudanças tais como a criação ou exclusão de um elemento ou atributo, alteração do nome, formato ou tamanho de um elemento e a criação ou exclusão de um elemento complexo. O Tamino permite implementar em seu esquema XML as principais definições de DTD, tais como elementos multivalorados e estruturas *choice* ("|" em DTD), implementando tais estruturas de forma direta, sem precisar fazer um mapeamento para tabelas relacionais, como ocorre com os SGBD´s relacionais.

A implementação de uma evolução de esquema XML no Tamino tem impacto direto na estrutura da FDT do Adabas. No Adabas, através da utilização de campos e itens de grupos nulos ou facultativos, suprimidos fisicamente através do contador de campos nulos (EFC), é possível a implementação da maioria das evoluções de esquema XML do Tamino. Por exemplo, considerando a evolução de esquema do elemento tipo_lancto, apresentado na Figura 16, onde foi implementada uma estrutura "*choice*"  com a criação dos elementos dt_inicio_lancto e

dt_fim_lancto, podemos identificar como ocorreu a evolução de esquema no Tamino analisando as informações apresentadas na Figura 25.

```
Em XML                          DDM                          Em Adabas
-------------  ------------------------------------ --------------
Elem.complexo  1 da  tp_lancto                      ítem de grupo
Elem.simples   2 db  desc_tp_lancto    varchar(50)  ítem elementar
Elem.complexo  1 dd  dt_lancto                      item de grupo
Elem.simples   2 de  dt_inicio_lancto  date         ítem elementar
Elem.simples   2 df  dt_fim_lancto     date         ítem elementar
Atributo       1 dc  cd_tp_lancto      integer   DE   ítem elementar
               | |   |                 |         |
               | |   |                 |         |
               | |   |                 |       Tipo: Múltiplo,
               | |   |                Formato       Periódico,
               | |   Nome Externo                   Descritor
               | Short Name
              Nível
```

Figura 25 - DDM Adabas do Elemento tp_lancto

Observe na Figura 25 que, apesar do novo elemento complexo `dt_lancto` estar definido fisicamente no Adabas no nível 1, na estrutura XML do Tamino apresentada na Figura 16 ele faz parte do elemento complexo tp_lancto. Ou seja, os novos campos são criados fisicamente dentro das características com que são possíveis de serem implementados no Adabas, para então serem associados ao elemento XML segundo sua definição de estrutura XML do Tamino.

Outros tipos de alterações como mudança de nome de elemento e alteração de formato e tamanho não têm impacto na estrutura física do Adabas, pois os elementos são associados a campos do Adabas, os quais são referenciados internamente pelo seu nome interno (*short name*), sendo que o *layout* dos elementos está definido na estrutura XML do Tamino. Desta forma, ao associar um elemento definido na estrutura XML do Tamino a um campo do Adabas, seu conteúdo é armazenado tal qual se apresenta no documento XML. Para o campo ser armazenado no Adabas, não é levada em conta a definição lógica do elemento na *collection*, pois o campo é armazenado com um byte de tamanho na frente e só é convertido para o layout do elemento da *collection* no momento da apresentação. Por exemplo, se um elemento está definido na *collection* como char, fisicamente no Adabas será criado um

campo char, sem especificar o tamanho e com o byte de tamanho na frente. Qualquer que seja o tamanho do elemento no documento XML, ao ser armazenado no Adabas o tamanho do mesmo é ajustado segundo seu preenchimento e o valor do tamanho armazenado no byte à frente do campo. Desta forma, ao aumentar o tamanho de um campo ou alterar seu formato na definição de estrutura XML do Tamino, alterando conseqüentemente também a *collection*, não é preciso fazer nenhuma alteração no Adabas, pois ao ser selecionado o campo será apresentado segundo o novo *layout* do elemento.

5 – CONCLUSÃO E RECOMENDAÇÕES

5.1 Conclusão

O armazenamento de dados históricos em bancos de dados relacionais tem um alto custo. Este custo envolve desde a área em discos para o armazenamento dos dados no banco de dados, como também as áreas em fitas e cartuchos para backups. Também é considerado o alto tempo de execução dos utilitários de banco de dados, tais como backup, reorganização, atualização de estatísticas e verificação de integridade. Outro fator negativo do uso de bancos de dados relacionais para armazenamento de dados históricos são as alterações no esquema. Uma alteração de esquema pode requerer a recriação da tabela, exigindo para isto uma área adicional equivalente ao tamanho da tabela no próprio banco de dados ou uma área equivalente em disco para descarregá-la.

Um passo muito importante do uso do XML para armazenamento de dados históricos é o mapeamento do modelo Relacional para um esquema XML. Uma vez que os dados históricos que serão armazenados em XML derivaram de uma estrutura relacional, é possível tirar vantagens da nova estrutura de dados semi-estruturada XML, principalmente através da utilização de elementos multivalorados e operadores de união, além da facilidade semântica na modelagem da evolução de esquema.

O estudo apresentado neste trabalho mostra que o modelo XML é adequado para dar suporte à evolução de esquemas. Além disso, a forma de armazenamento adotado pelo SGBD XML Nativo Tamino apresenta características interessantes para a implementação das construções de XML necessárias para se adequar a alterações no esquema. Estas características são: compressão de dados nos níveis de campo e registro e implementação do operador de união diretamente no nível físico. Tais características não existem em SGBD´s relacionais. Concluímos que a adoção de um esquema misto de armazenamento, armazenando os dados históricos no formato XML

e os dados correntes em um SGBD relacional, apresenta a vantagem de simplificar manutenções no caso de evolução de esquema. Tal proposta poderia ser implementada através da utilização de SGBD´s de modelos distintos para os dois tipos de dados. Este estudo também aponta características desejáveis de um SGBD relacional em suas futuras extensões para dar suporte ao armazenamento de XML. O XML vem sendo cada vez mais utilizado, porém ainda não se sabe se o futuro será a utilização de SGBD´s relacionais com extensão a XML ou a utilização de SGBD´s XML Nativo.

5.2 Sugestões para Trabalhos Futuros

Como continuidade a este trabalho outros itens podem ser explorados, entre os quais sugere-se:

a) Análise da adequação de índices;
b) Análise de desempenho de consultas com um volume mais significativo de dados;
c) Implementação de uma camada para fazer o mapeamento para uma visão relacional a partir do XML, fazendo com que as consultas a dados históricos armazenados em documentos XML ocorram de forma transparente para os usuários.
d) Aplicação em outros contextos em que há necessidade de manutenção de dados históricos como DW e OLAP.

BIBLIOGRAFIA

[1] XML - www.w3.org/TR/REC-xml

[2] DTD - www.w3schools.com/dtd/default.asp

[3] Xquery – www.w3.org/TR/xquery

[4] Dongwon Lee, Murali Mani, Frank Chiu, Wesley w. Chu - May 2001. *Nesting-based Relational-to-XML Schema Translation*. Proceedings of the ACM SIGMOD International Workshop on the Web and Databases (WebDB). Santa Barbara, CA.

[5] Wenfei Fan, Carmem Hara, Wang-Chiews Tan, Peter Buneman – *Keys for XML* Computer Networks, Volume 39, Issue 5, 5 August 2002, Pages 473-487

[6] Xpath – www.w3.org/TR/xpath

[7] DB2XML - www.informatik.fh-wiesbaden.de/~turau/DB2XML/index.html

[8] Sean McCown - April 23, 2004 - *Databases flex their XML* – InfoWorld - www.infoworld.com/article/04/04/23/17FExml_1.html

[9] Vinod Kumar – July 30, 2004 - *Changing XML Schema attached in SQL Server 2005* - www.extremeexperts.com/SQL/Yukon/ChangingXMLSchema.aspx

[10] Jonathan Robie, Joe Lapp, David Schach - Sep/1998 – *XQL – XML Query Language* - www.w3.org/TandS/QL/QL98/pp/xql.html

[11] Alin Deutsch, Mary Fernandez , Daniela Florescu , Alon Levy, and Dan Suciu - Aug/1998 – *XML-QL: A Query Language for XML* - www.w3.org/TR/1998/NOTE-xml-ql-19980819

[12] Raibow Core - davis.wpi.edu/~dsrg/rainbow/RainbowCore/

[13] Galax – www.galaxquery.org/

[14] Software AG – Tamino XML Server v.4.1.4 - www.xmlstarterkit.com/down/download_XMLServer.htm

[15] *Microsoft SQL Server 2000 Enterprise Edition Eval Ver.2* – www.microsoft.com/train_cert

[16] Software AG – *Adabas D 13. Documentation* - www2.softwareag.com/Corporate/products/adabas/adad/download/default.asp

[17] Thomas Lemke – *Schema Evolution in OODBMS: A Selective Overview of Problems and Solutions* - IDEA working paper IDEA.WP.22.O.002, September 1994

[18] Unicamp – Introdução a Orientação a Objetos - www.dca.fee.unicamp.br/courses/POOCPP/node3.html

[19] Brad Sarsfield, Srik Raghavan - Microsoft Corporation - March 2004 - *Overview of Native XML Web Services for Microsoft SQL Server 2005* - msdn.microsoft.com/sql/default.aspx?pull=/library/en-us/dnsql90/html/sql2005websvc.asp

[20] Sybase ASE 12.5 - http://sybooks.sybase.com/database.html

[21] Oracle 10g - www.oracle.com/technology/documentation/database10g.html

[22] IBM – DB2 v.8.1 - www-306.ibm.com/software/data/db2/udb/support/manualsv8.html

[23] Michael Rys – *XML Schema evolution in SQL Server 2005* – Yukon – sqljunkies.com/WebLog/mrys/archive/2004/05/11/2438.aspx

[24] XML Schema - www.w3.org/XML/Schema

[25] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Objetct-Oriented Database System. The Story of O2*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[26] Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object-oriented database. In B. Shriver and P. Wagner, editors, *Research Directions in Object-Oriented Programming*, pages 1-15. MIT Press, 1987.

[27] Won Kim and Hong-Tai Chou. Versions of schema for object-oriented database systems. In François Bancilhon and Davis J. DeWitt , editors, *Proceedings of the Fourteenth International Conference on Very Large Data Bases (VLDB 88)*, pages 148-159, Los Angeles, USA, August 1988. Morgan Kayfmann Publishers.

[28] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In *Object Oriented Programming: Systems, Languages and Applications. Proceedings of the Conference (OOPSLA 87),* pages 111-117, 1987.

[29] Markus Tresch and Marc H. Scholl. Meta object management and its application to database evolution. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, pages 299-321, Karlsruhe, Germany, October 1992. Spring-Verlag.

[30] Simon Monk and Ian Sommerville. A model for versioning classes in object-oriented databases. In *Proceedings of the Tenth British National Conference on Databases (BNCOD10)*, Aberdeen, Scotland, July 1992.

[31] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In Norman Meyrowotz, editor, *Object-Oriented Programming: Systems, Languages and Applications. (OOPSLA / ECOOP 90),* pages 67-76. ACM Press, October 1990.

[32] Stewart M. Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, School of Computer Science, Pittsburg, USA, June 1992.

[33] Gilles Barbedette. Schema modifications in the lisp O_2 persistent object –oriented language. In *European Conference on Object-Oriented Programming (ECOOP 91)* , volume 512 of *Lecture Notes in Computer Science*, pages 77-96. Geneva, Switzerland, July 1991, Springer-Verlag.

[34] Lichao Tan and Takuya Katayama. Meta operations for type management in object-oriented-databases – a lazy mechanism for schema evolution. In *Deductive and Object-Oriented Databases. Proceedings of the First International Conference (DOOD 89),* pages 241-258, Kyoto, Japan, December 1989. North-Holland.

[35] Jan L. Harrington – Projetos de Bancos de Dados Relacionais - Campus – 2002 – Elsevier Editora Ltda.

[36] UML – Documentação Oficial da UML – www.rarional.com/uml