PARTIX: PROJETO DE FRAGMENTAÇÃO DE DADOS XML

Alexandre da Silva Andrade

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Profª. Marta Lima de Queirós Mattoso, D.Sc.

Profª. Fernanda Araujo Baião, D.Sc.

Profª. Vanessa de Paula Braganholo, D.Sc.

Profª. Ana Maria de Carvalho Moura, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2006

ANDRADE, ALEXANDRE DA SILVA

PartiX: Projeto de Fragmentação de dados XML [Rio de Janeiro] 2006

XIII, 98p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Banco de Dados Distribuídos
2. Fragmentação de Documentos XML
3. Banco de Dados XML

I. COPPE/UFRJ II. Título (série)

À minha família que amo tanto.

Agradecimentos

A Deus, pela ajuda, pela saúde, pela vida.

À minha mãe, Maria Teresa, que me apoiou nas minhas escolhas durante toda minha vida.

À Gabriela Ruberg e Vanessa Braganholo, pela ajuda em diversos pontos teóricos e conversas esclarecedoras sobre XML

À Fernanda Baião, por ter aceitado ser minha co-orientadora e pela compreensão em alguns momentos difíceis que passei.

À minha orientadora, Marta Mattoso, que me ajudou muito e compreendeu minhas ausências devidas a alguns problemas pessoais, e me animou a continuar e não perder o foco no trabalho.

À professora Ana Maria, por ter aceitado participar desta banca.

À Ana Luisa Duboc. Uma pessoa muito especial pra mim, apesar das nossas brigas e discussões.

Aos meus amigos Kate Revoredo e Matheus Wildemberg, que me mostraram como usar o Latex! Se tivesse uma ferramenta de revisão boa, eu o estaria usando, ao invés do Word.

À minha amiga Melise Veiga. Agradeço as conversas que tivemos na COPPE sobre mestrado, doutorado e outros assuntos.

Aos meus amigos Noemi Sgambato (Aradya), Fabiano Dall'Aqua (Sweven), Lee, Leon, Torast, Merryadock, Ditto e outros amigos do clã "A Torre". Sem vocês, eu teria "pirado" no meio de tanto artigo pra ler durante a noite.

À minha amiga Mariana Sgambato (Nevan), do clã "A Torre". Nossas conversas no MSN enquanto eu preparava a dissertação foram (e são) ótimas.

A todos os meus amigos, companheiros de turma e trabalho, entre eles, Viviane, Grazziela, Talitta, Márcio Duran, Gustavo Pinto, Von Held, Uilton e todos os outros, já

que não tenho espaço para pôr todos. Obrigado pelas viagens, festas e o companheirismo de vocês.

À Patrícia Leal, pela valiosa amizade conquistada durante esses anos e por seu trabalho sério e competente como secretária da linha de banco de dados.

Ao projeto COPPETec, que me permitiu continuar em contato com o mercado de trabalho.

A CAPES e ao governo brasileiro, pelo apoio financeiro.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PARTIX: PROJETO DE FRAGMENTAÇÃO DE DADOS XML

Alexandre da Silva Andrade

Março/2006

Orientadores: Marta Lima de Queirós Mattoso
Fernanda Araújo Baião

Programa: Engenharia de Sistemas e Computação

O volume de dados de coleções de documentos XML e o tempo de resposta do processamento de consultas em sistemas de bancos de dados (SGBD) com tais coleções tornaram-se pontos críticos para muitas aplicações, especialmente aplicações Web. Uma alternativa interessante para melhorar o desempenho de consultas seria reduzir o tamanho das coleções de documentos XML através de um projeto de fragmentação de dados. Contudo, as definições existentes de fragmentação não são diretamente aplicadas a coleções de documentos XML. Além disso, a ausência de avaliações de desempenho de consultas sobre bases fragmentadas dificulta as decisões de um administrador de dados quanto ao projeto de fragmentação. Esta dissertação apresenta uma formalização para a fragmentação de coleções de documentos XML e propõe uma arquitetura para o processamento de consultas XQuery sobre coleções de dados XML fragmentados. Esta arquitetura foi implementada em um protótipo chamado PartiX, o qual provê o paralelismo intra-consulta através de uma camada de software entre a aplicação e um conjunto de SGBD seqüenciais que adotam a XQuery como linguagem de consulta. O PartiX foi avaliado através de vários conjuntos de dados experimentais, e os resultados mostraram um ganho de desempenho de até 72 vezes comparado ao SGBD centralizado.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PARTIX: FRAGMENTATION DESIGN OF XML DATA

Alexandre da Silva Andrade

March/2006

Advisors: Marta Lima de Queirós Mattoso
Fernanda Araújo Baião

Department: Computer and Systems Engineering

The data volume of collections of XML documents and the response time of query processing in database systems with those collections have become critical issues for many applications, especially for those in the Web. An interesting alternative to improve query processing performance consists in reducing the size of collections of XML documents through data fragmentation design. However, traditional fragmentation definitions do not directly apply to collections of XML documents. Moreover, the absence of performance evaluations of queries on fragmented databases makes it difficult for the data administrator to make decisions on fragmentation design. This dissertation formalizes the fragmentation definition for collections of XML documents and proposes an architecture for XQuery processing on top of fragmented XML data. This architecture was implemented in a system prototype named PartiX, which exploits intra-query parallelism through a software layer between the user application and a set of XML-enabled sequential DBMS modules. We have analyzed several experimental settings, and our results showed a performance improvement of up to a 72 scale up factor against centralized databases.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1 - Introdução

1.1 Motivação

XML e seus padrões relacionados representam uma contribuição aos sistemas de integração de informações e à gerência de dados distribuídos. Essas contribuições proporcionaram o desenvolvimento de muitos sistemas de banco de dados XML nativos (EXIST DEVTEAM, 2005, FIEBIG, HELMER et al, 2002, SCHÖNING, 2001) ou que armazenam XML de alguma outra forma, permitindo que os dados sejam consultados através de expressões de caminho (ditos *XML-enabled DBMS*) (ORACLE, 2005, MICROSOFT, 2005). E, embora muitos dos documentos XML sejam visões de dados relacionais, o número de aplicações usando documentos XML nativos tem aumentado rapidamente. Estas aplicações utilizam-se de sistemas de bancos de dados nativos e atualizam dados XML (ARENAS, LIBKIN, 2004). Ainda assim, a utilização de dados XML representa muitos desafios para o processamento de consultas. Apesar dos esforços de pesquisa acadêmica e industrial na área de banco de dados nos últimos anos (ABITEBOUL, BONIFATI et al, 2003, BERTINO, CATANIA et al, 2004, CHEN, DAVIDSON et al, 2004, RIZZOLO, MENDELZON, 2001), o processamento eficiente de consultas sobre grandes repositórios XML permanece como um problema complexo mesmo para os cenários centralizados.

Muitos fatores influenciam o desempenho do processamento de consultas XML. Em particular, o tamanho da base de dados tem um impacto significante nas ferramentas típicas para recuperação de dados XML. Isto acontece devido a uma funcionalidade básica destas ferramentas, chamada de análise (*parsing*) do documento XML, que consiste em verificar a validade do documento XML com as regras definidas pelo padrão XML (W3C, 2005) e com respeito aos tipos pré-definidos através de DTDs ou XML Schema (W3C, 2005). A maioria dos analisadores (*parsers*) de XML são baseados em uma estrutura de árvore como o *Document Object Model* (DOM) (W3C, 2005), os quais possuem o documento XML como unidade de dado. Isso acarreta que a validação de tipos e da estrutura do documento XML freqüentemente demanda por estruturas em memória que, em geral, são proporcionais ao tamanho do documento. Conseqüentemente, este tipo de tarefa de validação pode se tornar bastante cara em

termos de consumo de tempo em grandes repositórios de dados XML (RUBERG, RUBERG et al , 2004).

Um outro aspecto do desempenho é a complexidade dos padrões (e recomendações) de linguagens de consulta XML, como o XPath e o XQuery (W3C, 2005), os quais são muito poderosos e requerem algoritmos sofisticados. Vale a pena notar que as consultas típicas em XML, mesmo as mais simples, muito freqüentemente envolvem operações custosas, tais como busca baseada em texto e avaliações de expressões de caminho. Uma forma, vastamente empregada, de melhorar o desempenho desse tipo de consulta consiste na utilização de índices pré-computados. Não obstante, algumas classes importantes de consultas XML, especialmente *ad-hoc*, tornam difícil a definição prévia de índices específicos. Para estas consultas, o grande volume de dados representa um problema de desempenho mesmo para sistemas com técnicas avançadas de otimização de consultas (RIZZOLO, MENDELZON, 2001).

Diferentes *tipos de fragmentação de dados* têm sido utilizados para auxiliar os sistemas de banco de dados a lidar com grande volume de dados, tanto em banco de dados relacionais (YAO, ÖZSU et al, 2004) e orientados a objetos (BAIÃO, MATTOSO et al, 2004). Entretanto, estes tipos não se ajustam em banco de dados XML, principalmente pela grande flexibilidade do modelo semi-estruturado. Informações redundantes e incompletas são algumas das características inerentes do XML que complicam a definição de fragmentos XML. A checagem da *correção* de um esquema XML fragmentado (com respeito à completude, disjunção e reconstrução (YAO, ÖZSU et al, 2004)) é um problema, em geral, complexo. Além disso, este problema é amplificado pela complexidade típica dos operadores de consulta XML. Por exemplo, seleções XML (tipicamente utilizadas para definição de fragmentos horizontais) usualmente envolvem expressões de caminho contendo termos regulares, tais como "//", e predicados com funções e/ou quantificadores.

Alguns guias de como definir propriamente fragmentos XML foram delineados em (BREMER, GERTZ, 2003) e (MA, SCHEWE, 2003). Entretanto, quando tentamos usar esses guias para diferentes tipos de repositórios XML, houve dificuldade em identificar uma clara distinção entre fragmentação XML horizontal, vertical e híbrida.

A primeira dificuldade advém da não distinção entre repositórios de dados XML tais como Único Documento (*Single Document*, SD) e Múltiplos Documentos (*Multiple Document*, MD)(YAO, ÖZSU et al, 2004). Em repositórios do tipo SD, todas as informações estão armazenadas em um único documento XML (que apresenta um esquema definido em XML ou DTD). Já em repositórios MD, também existe a definição de um esquema, porém, o repositório é formado por múltiplas instâncias (documentos XML) desse esquema.

Por exemplo, em (BREMER, GERTZ, 2003), a fragmentação vertical e a horizontal estão diluídas na especificação de um fragmento XML. Isso também ocorre em (MA, SCHEWE, 2003), onde a fragmentação horizontal pode envolver reestruturação dos dados. Conseqüentemente, é igualmente complexo checar a correção destes tipos de fragmentação. Por outro lado, a definição de fragmentos em (BREMER, GERTZ, 2003) pode ser somente aplicada a repositórios SD. Similarmente, para fragmentar um repositório MD em (MA, SCHEWE, 2003), o usuário deve especificar uma visão SD do banco de dados para cada definição de fragmento. A saber, cada documento XML tem de estar propriamente conectado a uma visão intermediária. É importante notar que esta é uma abordagem incômoda mesmo para pequenas coleções de documentos.

Outra dificuldade está na especificação formal da fragmentação. Esta definição está inerentemente ligada à álgebra XML. Entretanto, devido à ausência de um consenso quanto às diversas álgebras propostas, diversos operadores podem ser utilizados, mas nem sempre de modo completo. Consideramos um ponto importante o uso de uma álgebra já estabelecida e com analisadores léxicos, sintáticos e otimizadores disponíveis para o mapeamento automático de consultas centralizadas sobre itens de dados fragmentados. Muitas das propostas existentes se referem a uma álgebra particular sem fazer referência a álgebras abrangentes e bem sucedidas, como TAX (JAGADISH, LAKSHMANAN et al, 2001) e TLC (PAPARIZOS, WU, et al, 2004).

Sendo assim, não conseguimos utilizar nenhuma das propostas de definição de fragmentação para coleções de documentos XML. Além disso, até o nosso conhecimento, os trabalhos atuais acerca de fragmentação de dados XML carecem de uma análise experimental de diferentes tipos de fragmentação com respeito ao desempenho do processamento de consultas XML sobre grandes repositórios.

1.2 Objetivos

Esta dissertação tem como principal objetivo contribuir para a solução dos problemas existentes em projetos de fragmentação em coleções de dados XML. Para tanto foi necessário formalizar as principais alternativas para fragmentação de coleções de documentos XML. Nosso modelo de fragmentação é formal e, ainda assim simples, quando comparado com os trabalhos relacionados. O modelo considera tanto repositórios SD como os MD tendo sido definidas as regras de correção do modelo e delineada a verificação dessas regras de correção para cada caso apresentado.

Além disso, realizamos a avaliação empírica de tipos de fragmentação de dados para repositórios de XML. Essa avaliação visa oferecer subsídios na elaboração de um projeto de fragmentação adequado ao perfil de consultas da aplicação. Para tanto, propomos uma arquitetura para processamento de consultas XQuery sobre repositórios XML, fragmentados segundo um projeto de fragmentação. Um resumo dessa avaliação pode ser encontrado em (ANDRADE, RUBERG et al, 2006).

Diversos testes foram realizados experimentalmente utilizando essa arquitetura sobre bases de dados em um SGBD XML nativo, de código aberto, chamado eXist (EXIST DEVTEAM, 2005). Focando na carência de informações sobre os ganhos potenciais que podem ser obtidos com a fragmentação de repositórios XML, nós apresentamos resultados experimentais para diferentes projetos de fragmentação, envolvendo fragmentações horizontais, verticais e híbridas, em coleções de documentos XML dos tipos SD e MD. Os resultados mostraram ganhos de desempenho substanciais, sendo, em alguns cenários, até 72 vezes mais rápidos quando comparados com os conjuntos de dados processados pelo mesmo SGBD de modo centralizado. Nesses experimentos foram utilizados tanto exemplos criados por nós para explorar todas as definições de fragmentação, como também bases de *benchmarks* conhecidos como o XMark (SCHMIDT, WAAS et al, 2001) e XBench (YAO, ÖZSU et al, 2004). Acreditamos que esses resultados, até então desconhecidos, irão contribuir para a definição de um bom projeto de fragmentação de coleções de documentos XML.

1.3 Organização dos Capítulos

O texto está organizado da seguinte forma. No Capítulo 2, discutimos a fragmentação em outros modelos de dados, os principais trabalhos relacionados e as

álgebras existentes para apoiar a fragmentação. No Capítulo 3, apresentamos os fundamentos necessários e uma definição para os tipos de fragmentação de dados em XML. Também são definidas as regras de correção necessárias e uma análise de verificação de correção em alguns casos específicos que podem ocorrer em determinados tipos de documentos. Uma arquitetura para o processamento de consultas distribuídas é apresentada no Capítulo 4. O Capítulo 5 é reservado para a apresentação e análise de resultados dos testes realizados. Por fim, o Capítulo 6 resume as conclusões desta dissertação e os trabalhos futuros.

Capítulo 2 - Distribuição em XML

Este capítulo apresenta alguns conceitos básicos para o entendimento de distribuição, tanto no modelo XML como nos demais modelos. Discute também os trabalhos relacionados, entre eles aqueles que tratam de fragmentação de dados XML e álgebras XML. O capítulo está organizado da seguinte forma. Na seção 2.1 discutiremos o contexto da dissertação e o que deveremos avaliar para definirmos o que são fragmentos XML. A seção 2.2 discute como é a fragmentação em outros modelos de dados. Na seção 2.3 e 2.4 são discutidos trabalhos que abordam álgebras para XML e fragmentação de documentos XML Por fim, a seção 2.5 faz algumas considerações sobre o que foi discutido no capítulo.

2.1 Introdução

O tópico sobre fragmentação de dados é bem conhecido tanto em sistemas de banco de dados relacionais (ÖZSU, VALDURIEZ, 1999) como em bancos de dados orientados a objetos (BAIÃO, MATTOSO et al, 2004).

Um projeto de distribuição de dados visa a dividir uma base de dados em fragmentos e alocá-los entre diversos nós de um ambiente distribuído. Assim, com base no padrão de consultas mais freqüentes da aplicação, o projetista identifica, dentre os conjuntos de dados da base, aqueles sub-conjuntos mais utilizados. A fase de fragmentação de dados visa a encontrar os subconjuntos que, por sua vez, definem os fragmentos. Em seguida, os fragmentos são alocados de acordo com o nó que dispara as consultas. Em geral, o projeto segue uma metodologia (BAIÃO, MATTOSO et al, 2004) que direciona o projetista durante as fases de fragmentação e alocação de dados.

Vale a pena lembrarmos as razões para realizarmos uma fragmentação e suas conseqüentes desvantagens. Realizamos a fragmentação sobre um conjunto de dados, pois a aplicação que os utiliza, em geral, somente considera um sub-conjunto desses dados. Outro motivo ocorre quando aplicações remotas acessam conjuntos diferentes de uma mesma fonte de dados. Nestes casos, existem duas alternativas: o conjunto de dados não é replicado e é somente armazenado em um sítio; ou é replicado em todos os sítios, ou somente naqueles em que as aplicações são executadas. No primeiro caso, verifica-se o problema de acesso remoto desnecessário a uma grande

quantidade de dados; na segunda alternativa, temos replicação desnecessária, o que poderá causar problemas com atualizações e pode não ser uma opção desejável, se o espaço de armazenagem é limitado.

Finalmente, a decomposição desse conjunto de dados em fragmentos permite que transações feitas sobre eles operem concorrentemente. Além disso, permite que uma consulta seja executada em paralelo, já que pode ser dividida em sub-consultas a serem executadas em cada fragmento. Assim, a fragmentação reduz o tempo de processamento dessa consulta e conseqüentemente aumenta o nível de concorrência e com isso a vazão do sistema.

Porém, existem situações em que a fragmentação pode não ser indicada. Se uma aplicação tem requisitos conflitantes que impeçam a decomposição do conjunto de dados em fragmentos mutuamente exclusivos, pode haver uma degradação de desempenho. Pode ser necessário, por exemplo, recuperar dados de dois fragmentos e efetuar uma operação de junção ou união sobre eles, e isso pode ser custoso.

Outro problema é relativo ao controle semântico dos dados, especificamente, controle de integridade. Como resultado da fragmentação, atributos que participam de uma determinada regra de integridade podem ser decompostos em diferentes fragmentos. Neste caso, uma tarefa simples de verificação de integridade pode envolver recuperação de informação em sítios remotos, o que pode também acarretar em perda de desempenho.

A presente dissertação apresenta uma proposta de definição de fragmentos para coleções de dados XML. Uma das principais preocupações ao longo do trabalho foi manter, até certo ponto, um paralelo com o modelo relacional, já que este apresenta uma teoria muito bem estabelecida acerca da fragmentação de relações.

Essa teoria não pode ser aplicada diretamente ao modelo XML pois, como veremos mais adiante, a unidade de dado no modelo XML não é uma tupla (ou seja, um conjunto de pares atributos/valor). Outro fato que contribui para esta diferença é a estrutura em forma de árvore de um documento XML.

O foco da dissertação é analisar conjuntos de documentos homogêneos, ou seja, que apresentam o mesmo esquema XML, o qual pode ser definido através de um

XML Schema ou DTD. É importante notar que, pelas características do modelo XML, dois documentos podem apresentar o mesmo esquema, porém árvores de estruturas *diferentes*.

Nesta dissertação, estamos interessados não somente em especificar como definir fragmentos sobre documentos XML, mas especificá-los com base em uma álgebra XML bem estabelecida. Atualmente existem várias álgebras, algumas específicas para soluções de problemas como o processamento de consultas em *streams* de dados XML (FEGARAS, LEVINE, et al, 2002, BOSE, FEGARAS, 2005, BOSE, FEGARAS et al, 2003), outras processam os documentos como nodos (FRANSICAR, HOUBEN et al, 2002) e outras como árvores TAX e TLC (JAGADISH, LAKSHMANAN et al, 2001, PAPARIZOS, WU et al, 2004). Neste capítulo apenas iremos tratar de algumas álgebras e analisar diferenças entre elas. No Capítulo 3 descreveremos a TLC (PAPARIZOS, WU et al, 2004), a álgebra escolhida para as nossas definições e os motivos dessa escolha.

2.2 Fragmentação em outros Modelos de Dados

É interessante olharmos como é definida a fragmentação em outros modelos de dados já consolidados. Como o nosso objetivo é ter uma definição para fragmentos XML que seja, até certo ponto, análoga à já existente para esses modelos. Com isso, podemos aproveitar muitos dos conceitos já existentes e bastante discutidos.

2.2.1 Modelo Relacional

O Modelo Relacional é baseado em *atributos*, *tuplas* e *relações*. Todo atributo possui um determinado domínio, e só pode ter um valor atômico neste domínio; uma tupla representa um conjunto de atributos; e por sua vez, a relação representa o conjunto de tuplas. Uma relação pode ser representada por uma tabela, com várias colunas, uma para cada atributo. Para maiores detalhes com respeito ao modelo de dados relacional, consulte (ELMASRI, NAVATHE, 2000).

Existem três alternativas de fragmentação de uma relação no modelo relacional: Horizontal, Vertical e Híbrida. A fragmentação horizontal divide a relação em vários fragmentos (que também são relações) com o mesmo esquema da relação original. Esta divisão é baseada em predicados do operador de seleção da álgebra relacional. Ainda existe um tipo particular de fragmentação horizontal, chamado

Fragmentação Horizontal Derivada, onde uma relação dependente (possui uma chave estrangeira) de uma outra relação já fragmentada horizontalmente é fragmentada com base nessa dependência. Esta divisão visa a otimizar o processamento de operadores de junção da álgebra relacional, reconhecidamente como os operadores mais custosos no processamento de consultas. A fragmentação vertical utiliza o operador relacional de projeção para dividir a relação original. Cada fragmento contém a chave primária da relação e possui diferentes estruturas de esquema. Já a fragmentação híbrida é definida como a aplicação de uma fragmentação horizontal seguida de uma vertical, ou vice-versa. Maiores detalhes com respeito à fragmentação no modelo relacional pode ser encontrada em (ÖZSU, VALDURIEZ, 1999).

Todos os tipos de fragmentação do modelo relacional são baseados na álgebra relacional. A fragmentação horizontal utiliza o operador de seleção em sua definição. A fragmentação vertical, por sua vez, utiliza o operador de projeção. A fragmentação híbrida é definida através da operação aninhada dos operadores de seleção e projeção e vice-versa. E os operadores de união e junção são utilizados para reconstruir os fragmentos, na fragmentação horizontal e vertical, respectivamente.

2.2.2 Modelo Orientado a Objetos

Uma das principais dificuldades de pesquisa na área de SGBDOO diz respeito à falta de consenso sobre o modelo de dados formal universalmente aceito (BAIÃO, MATTOSO et al, 2004, ÖZSU, VALDURIEZ, 1999, ELMASRI, NAVATHE, 2000).

Vamos citar aqui o modelo de dados utilizado em (BAIÃO, MATTOSO et al, 2004). Um *tipo* representa tanto aspectos estruturais quanto comportamentais neste modelo e representa um modelo para os objetos. Ele é identificado pelas suas propriedades (atributos e relacionamentos) que refletem o estado do objeto, e as operações que podem ser realizadas sobre o objeto. Uma *classe* é um agrupamento de todas as instâncias de objetos de um dado *tipo*. Uma *classe* é formada por um identificador, um conjunto de atributos, um conjunto de métodos e um conjunto de objetos. Um objeto é a primitiva fundamental de modelagem e acesso em um SGBDOO. Ele é um encapsulamento de um identificador de objeto (OID) representando sua identidade única no sistema e um conjunto de variáveis de instância (atributos) definindo seu estado. O conjunto de variáveis de instância do objeto determinam o valor

atual dos atributos definidos em sua *classe*. Uma *coleção* é um agrupamento de objetos definido pelo usuário. Este pode ser um conjunto, um *bag* (um conjunto onde repetições são permitidas), uma lista (ordenada) ou um vetor (acesso direto aos elementos dada a posição) de objetos.

Para maiores detalhes com respeito ao modelo de dados orientado a objetos, consulte (BAIÃO, MATTOSO et al, 2004). Em (ELMASRI, NAVATHE, 2000), é apresentado o modelo de dados utilizado pelo ODMG (*Object Data Management Group*), o qual é utilizado como base para a ODL (*Object Definition Language*) e para a OQL (*Object Query Language*).

Para fragmentar uma classe, é possível a utilização de dois tipos básicos: a fragmentação vertical, a qual distribui atributos e métodos da classe entre os fragmentos; e a fragmentação horizontal, a qual distribui instâncias da classe (objetos) entre os fragmentos. Também é possível realizar uma fragmentação híbrida sobre uma classe, obtendo os benefícios de ambas as técnicas. A definição de fragmentos é discutida em (ÖZSU, VALDURIEZ, 1999, ELMASRI, NAVATHE, 2000). Em (BAIÃO, MATTOSO et al, 2004) é apresentado um trabalho detalhado e completo sobre aspectos envolvidos no projeto de distribuição em bases orientadas a objetos. Os autores propõem uma definição para os três tipos de fragmentação, com uma contribuição para a definição da fragmentação horizontal derivada, regras de correção, análise experimental de desempenho e uma metodologia eficiente para o projeto de distribuição neste tipo de base de dados.

2.3 Álgebras para XML

Uma parte importante desta dissertação foi a análise das álgebras disponíveis para o processamento de consultas XQuery. Vários trabalhos na literatura (FEGARAS, LEVINE et al, 2002, FRANSICAR, HOUBEN et al, 2002, SU, RUNDENSTEINER et al, 2005, JAGADISH, LAKSHMANAN et al, 2001, AL-KHALIFA, JAGADISH, 2002, PAPARIZOS, WU et al, 2004, PAPARIZOS, JAGADISH, 2005) tratam do processamento de consultas XQuery em vários aspectos.

Em (FEGARAS, LEVINE et al, 2002), os autores tratam do processamento eficiente de *streams* de dados XML contínuas, nas quais o servidor envia dados XML para múltiplos clientes concorrentemente, através de "multicast data stream", enquanto

que o cliente é inteiramente responsável pelo processamento da *stream* de dados recebida. No modelo dos autores, um servidor pode disseminar fragmentos XML de múltiplos documentos, pode repetir ou trocar fragmentos, e pode acrescentar novos e excluir os fragmentos inválidos. Um cliente utiliza um banco de dados simples baseado na álgebra proposta por eles para armazenar em cache as *streams* de dados e avaliar consultas XQuery sobre esses dados. Em si, o trabalho também propõe uma forma de fragmentar documentos XML, baseado num conceito de *fillers* e *holes*. Esse conceito é fundamental para a álgebra por eles apresentada. Os fragmentos criados quebram a estrutura do documento (fragmentação vertical) para o envio através da *stream*, e baseados nesses fragmentos, a álgebra é capaz de identificar quais partes são importantes para as consultas. Seria interessante a possibilidade de quebra desses documentos baseados em algum critério que leve em conta as consultas mais importantes, porém, isto não é citado no trabalho. Mas as suas grandes contribuições são o arcabouço utilizado, o XFrag, apresentado em (BOSE, FEGARAS, 2005)) e a álgebra específica para esse arcabouço. A formalização mais detalhada sobre os operadores dessa álgebra pode ser encontrada em (BOSE, FEGARAS et al, 2003).

Também em (SU, RUNDENSTEINER et al, 2005) temos em foco as consultas XQuery sobre *streams* de dados. O trabalho se apoia na otimização das consultas sobre dados XML baseados em um esquema. Critérios são estabelecidos com base em quais restrições do esquema são úteis para uma consulta em particular. Tendo isso em foco, os autores discursam sobre como aplicar várias técnicas de otimização. Os autores apresentam testes realizados com dados reais e sintéticos, mostrando que existem ganhos com a aplicação de suas técnicas.

Por outro lado, considerando um processamento em bases de dados centralizados, temos vários trabalhos na literatura. Em geral, podemos dividir as álgebras encontradas em dois ramos: umas focam no processo de coleções de **nodos dos documentos**; outras focam no processamento de coleções de **árvores de documentos**.

A primeira álgebra analisada foi a XAL (FRANSICAR, HOUBEN et al, 2002). Esta define operadores independentes da camada de armazenamento. São definidos três grupos de operadores: operadores de extração, que recuperam a informação necessária através dos documentos XML; meta-operadores, que controlam a avaliação das expressões; e operadores de construção, que constroem novos documentos

XML com base nos dados extraídos. Os autores citam que seu conjunto de operadores é suficientemente poderoso para expressar um grande conjunto de consultas XQuery. Todos esses operadores avaliam uma entrada como sendo uma coleção de vértices. Entre os operadores temos seleção, projeção, ordenação, junção, produto cartesiano, união, diferença, interseção, distinção (eliminação de duplicações) e um operador para retirada da ordem de uma coleção. A álgebra resultante do trabalho dos autores possui leis de otimização semelhantes às utilizadas para as consultas relacionais.

2.3.1 A Álgebra TAX para Árvores XML

Já a TAX (JAGADISH, LAKSHMANAN et al, 2001) e a TLC (PAPARIZOS, WU et al, 2004) não operam sobre coleções de nodos, mas sim sobre coleções de árvores. Estamos interessados em manipular documentos XML utilizando uma álgebra semelhante à utilizada no modelo relacional. Em (JAGADISH, LAKSHMANAN et al, 2001) temos uma discussão do que considerar como uma analogia às tuplas do modelo relacional. O artigo apresenta várias desvantagens para o uso de nodos e de sub-árvores como o análogo para as tuplas. O análogo escolhido pelos autores é a árvore que forma o documento. Porém, sendo um documento XML muito mais complexo quando comparado a uma tupla, algo deve ser criado para amenizar essa complexidade e podermos olhar de maneira uniforme para uma coleção de árvores.

Quando mencionamos *árvores heterogêneas* ao longo do texto, nos referimos à estrutura das árvores, e não ao seu esquema. Mesmo árvores com o mesmo esquema, graças à flexibilidade do XML-Schema e DTDs, podem ter estruturas diferentes.

A TAX criou o conceito de **Padrão de Árvore** (*pattern tree*) (JAGADISH, LAKSHMANAN et al, 2001). Esta define uma estrutura de interesse que pode ser aplicada a qualquer árvore dentro de uma coleção. Um padrão é fixo para uma dada operação da álgebra, provendo assim a necessária padronização sobre um conjunto de árvores heterogêneas. Com isso, eles foram capazes de definir grande parte dos operadores existentes na álgebra relacional, com algumas modificações apropriadas para o modelo XML.

Com respeito ao modelo de dados adotado pela TAX, temos como unidade básica de informação uma árvore de dados, o análogo à tupla no modelo relacional.

Uma **árvore de dados** (*data tree*) é uma árvore ordenada, na qual cada nodo carrega dados (seus rótulos) na forma de um conjunto de pares atributo-valor. É sabido que um banco de dados relacional é um conjunto de relações. Correspondentemente, um banco de dados XML deve ser um conjunto de coleções. Ou seja, em ambos os casos, um banco de dados é um conjunto de coleções. Enquanto que isso é raramente confundido no contexto do modelo relacional, existe uma tendência no contexto do modelo XML de tratarmos o banco de dados como um conjunto unitário, uma enorme estrutura aninhada (JAGADISH, LAKSHMANAN et al, 2001). Os autores assumem a existência de um atributo virtual existente em cada nodo, chamado de *Pedigree*. Este atributo pode ser acessado pelos operadores e tem um papel importante para os operadores de ordenação, agrupamento e eliminação de duplicação. O Pedigree pode ser implementado como sendo o identificador do documento concatenado com a posição relativa do nodo com relação à raiz. Esta foi a forma com a qual foi implementada no Timber (JAGADISH, AL-KHALIFA et al, 2002).

Ainda seguindo a analogia com o modelo relacional, cada um de seus operadores toma uma ou mais relações como entrada e produz uma relação como saída. Na TAX, cada operador toma uma ou mais coleções (de árvores de dados) como entrada e produz uma coleção como saída.

Como vimos, a TAX baseia-se no uso de padrões de árvores em seus operadores de alto nível para a resolução de consultas. Uma preocupação discutida em (AL-KHALIFA, JAGADISH, 2002) é a eficiente combinação desses operadores de alto nível (*macro-level operators*) e operadores de baixo nível (*micro-level operators*). Exemplos dessas operações, são, respectivamente, usar um padrão de árvore para especificar uma seleção; e utilizar múltiplas junções de caminho para instanciar uma simples expressão XPath. Os autores desenvolveram novos métodos de acesso que combinam um ou mais operadores primitivos, e demonstram que existem ganhos de desempenho nessas combinações.

2.3.2 A Álgebra TLC para Árvores XML

A outra álgebra analisada foi uma evolução da TAX, a TLC (*Tree Logical Classes Algebra*) (PAPARIZOS, WU et al, 2004). Neste trabalho, são apresentadas técnicas para manipulação de conjuntos heterogêneos de árvores. Em especial, o conceito de *padrão de árvore* é estendido por anotações em suas arestas, e

generalizando a semântica da verificação dos padrões de árvore para produzir conjuntos heterogêneos de árvores como saída. Também é apresentado o conceito de **classe lógica** (*logical class*) para nodos em uma árvore. O objetivo da classe lógica é evitar acessos redundantes e aumentar a eficiência do processamento da consulta.

As álgebras existentes para árvores usam padrões de árvore para o casamento de estruturas. As árvores resultantes deste casamento devem ser por definição homogêneas, com estrutura idêntica ao padrão. Para driblar esta limitação, o conceito de padrão de árvore foi estendido para padrão de árvore anotado (*annotaded pattern tree*, ou, *APT*). As anotações se referem a quantos filhos determinado nodo poderá ter na árvore de saída. As possibilidades são mostradas na Tabela 1.

Tabela 1: Anotações permitidas em uma APT

"-": um e somente um casamento é permitido na árvore de saída.
"?": zero ou um casamento é permitido na árvore de saída.
"+": um ou mais casamentos são permitidos na árvore de saída.
"*": zero ou mais casamentos são permitidos na árvore de saída.

As classes lógicas são conjuntos de nodos correspondentes a um nodo da árvore gerada. Para cada nodo do padrão de árvore anotado é criada uma classe lógica. Assim, as classes lógicas são utilizadas como parâmetros nos operadores da TLC. A grande vantagem das classes lógicas é permitir o reuso dos padrões de árvores anotados, evitando que novos padrões tenham que ser definidos e novos casamentos tenham que ser feitos para toda coleção. Fora isso, a TLC apresenta novos operadores que permitem também um maior reuso dos conjuntos de árvores gerados pela aplicação dos operadores já existentes. Em (PAPARIZOS, WU et al, 2004) foi feita uma comparação da utilização do TAX e da TLC para o processamento de consultas. Foi verificado que a TLC fornece melhores resultados de desempenho que a TAX. Em (PAPARIZOS, WU et al, 2004) também podemos encontrar um algoritmo para, a partir de uma especificação em XQuery, gerar um plano de consulta em TLC.

Em (PAPARIZOS, JAGADISH, 2005), temos definições de operadores para a TLC para garantir coleções ordenadas e eliminação de duplicações. Um aspecto que teve de ser considerado no trabalho dos autores é como determinar que uma árvore é idêntica à outra. Essa foi uma extensão importante da TLC, tornando-a uma álgebra mais poderosa para o processamento de consultas.

2.4 Fragmentação em Dados XML

Os princípios do projeto de distribuição em banco de dados XML foram inicialmente abordados por (BREMER, GERTZ, 2003) e (MA, SCHEWE, 2003). Ma e Schewe ampliaram idéias vindas da fragmentação de banco de dados orientados a objetos para tratar os dados em XML. Eles propuseram três tipos de fragmentação XML: *horizontal*, que agrupa elementos de um documento XML de acordo com algum critério de seleção; *vertical*, para re-estruturar o documento através do desaninhamento de alguns elementos; e um tipo especial chamado *split*, para quebrar a estrutura do documento XML em um novo conjunto de documentos. Entretanto, estes conceitos de fragmentação não estão claramente definidos. Por exemplo, a fragmentação horizontal envolve re-estruturação de dados e projeção de elementos, produzindo assim fragmentos com diferentes definições de esquema. Além disso, a fragmentação vertical requer a especificação de elementos artificiais para conectar corretamente os fragmentos. Mesmo que o usuário possa definir fragmentos a partir de vários documentos XML, esta abordagem não é apropriada para repositórios MD. Neste caso, para definirmos fragmentos horizontais, o usuário deve primeiro integrar todos os documentos XML em uma visão SD, o que é uma tarefa complexa, já que depende da representação específica do banco de dados. Além disso, não são apresentadas regras de correção para as regras de fragmentação definidas.

Bremer e Gertz (BREMER, GERTZ, 2003) propõem uma abordagem para o projeto de distribuição de XML, abrangendo tanto a fragmentação como a alocação dos dados. Contudo, a abordagem é voltada somente para repositórios SD. Adicionalmente, o formalismo proposto pelos autores não distingue entre fragmentações verticais e horizontais, que são combinadas em um tipo de fragmentação híbrida. Eles focam em maximizar a avaliação local da consulta através da replicação de informações globais, e distribuição de algumas estruturas de índices. Eles também apresentam ganhos de desempenho importantes, mas seus experimentos empíricos focam nos benefícios de suas estruturas de índices.

Outro trabalho que apresenta definições de fragmentos trata da troca de documentos XML de forma eficiente (AMER-YAHIA, KOTIDIS, 2004). Os autores definem uma arquitetura para troca de documentos XML através de serviços web, em conjunto com uma definição bem interessante de fragmentos verticais de um esquema

XML. Apesar de não ser baseado em uma álgebra, os autores definem as operações realizadas e um modelo de custos para a avaliação de desempenho das consultas realizadas. Os resultados obtidos foram bastante satisfatórios, mas recaem no problema de não definir os demais tipos de fragmentação. Além disso, consideram que os dados XML são expostos através de serviços web. Os dados XML não estão armazenados em um SGBD nativo XML, mas sim em uma base de dados relacional.

A QSSL (Query Set Specification Language) (PETROPOULOS, DEUTSCH et al, 2003), apesar de não definir uma forma de fragmentação, pode ser utilizada com esse fim. Ela é uma extensão do WSDL que, dada uma base de dados, permite uma descrição concisa e semanticamente correta de um conjunto de consultas parametrizadas. O objetivo da QSSL é descrever, de forma simples, uma grande quantidade de consultas através de um padrão pré-determinado. A especificação das consultas é feita através de padrões de árvore. Os fragmentos citados na QSSL se referem a partes de padrões de árvores que podem ser juntos ou retirados de acordo com o pedido do usuário.

Recentemente, sistemas P2P (*peer-to-peer*) têm sido utilizados para explorar o gerenciamento de dados distribuídos em ambientes dinâmicos. A maioria dos sistemas P2P é baseada em estruturas de índice especiais, tais como Tabelas *Hash* Distribuídas (THD). Bonifati e outros (BONIFATI, MATRANGOLO et al, 2004) propuseram uma abordagem interessante para consulta de dados XML em uma rede P2P. Os fragmentos verticais de um documento XML podem ser identificados através de expressões de caminho em XPath e distribuídas em uma THD, de modo que cada par mantém algum dado fragmentado e suas respectivas expressões de caminho (isto é, a lista de expressões de caminho de seus fragmentos filhos e seus fragmentos ancestrais). Desta forma, consultas podem ser facilmente roteadas em um sistema P2P. Outra abordagem P2P é apresentada em (ABITEBOUL, BONIFATI et al, 2003). Ela considera a distribuição de documentos XML Ativos (AXML), os quais são documentos dinâmicos com elementos especiais que denotam chamadas a Serviços Web. Para acelerar o acesso ao conteúdo de um documento AXML, eles propuseram um mecanismo para localmente replicar fragmentos de dados remotos que são retornados por alguma chamada de serviço. Os fragmentos de um documento AXML podem ser distribuídos de modo que o documento original mantenha referências para os fragmentos externos (e vice-versa), similarmente à linguagem XInclude (W3C, 2004).

2.5 Considerações quanto à fragmentação em XML

Uma parte importante desta dissertação foi a análise das álgebras disponíveis para o processamento de consultas XQuery. Como vimos na seção 2.3, existem várias álgebras XML. O objetivo é escolhermos uma que leve em consideração coleções de documentos, ao invés de processar apenas um documento. Além disso, a álgebra deve levar em consideração a árvore, e não nodos.

A álgebra apresentada por Fegaras e outros (FEGARAS, LEVINE, et al, 2002) foi idealizada para fragmentar verticalmente *streams* de dados. Todo um arcabouço foi montado por eles para o tratamento deste problema. Já a XAL (FRANSICAR, HOUBEN et al, 2002), possui um ótimo conjunto de operadores, porém trata os documentos como um conjunto de nós, o que torna complexa a identificação da unidade que queremos, a árvore.

Atendendo as nossas necessidades de operadores e à unidade desejada por nós, existe a TAX (JAGADISH, LAKSHMANAN et al, 2001) e a TLC (PAPARIZOS, WU et al, 2004). Conforme já citado na seção 2.3.1, as TAX e a TLC têm a árvore como um análogo para a tupla do modelo relacional. E com isso, poderemos nos guiar através dos conceitos de fragmentação do modelo relacional para traçar os conceitos de fragmentação do modelo XML. Porém, ainda existe uma desvantagem. A TAX considera que as coleções são compostas de documentos estruturalmente idênticos. Nós desejamos que a coleção seja de um esquema único, mas que o documento possa ter diversas estruturas (dada a existências de elementos com cardinalidade N, 1, ou 0). Para sanar essa restrição da TAX, surgiu a TLC, que elimina a restrição através de padrões de árvores anotados. Por esses motivos, a TLC foi escolhida como sendo a álgebra para apoiar nossa definição de fragmentos.

Quanto aos trabalhos sobre fragmentação XML, nenhum desses trabalhos claramente define os principais tipos de fragmentação (i.e., horizontal, vertical, e híbrida) sobre XML, ou se propõe a tratar mais de um tipo de repositório (SD e MD). Isso diminui o leque de possibilidades de uso das soluções apresentadas.

Em Ma e Schewe (MA, SCHEWE, 2003), apesar de existir a definição para vários tipos de fragmentação, estes não estão claramente elucidados. Além disso, a definição somente se aplica a repositórios SD. Bremer e Gertz (BREMER, GERTZ,

2003) definem a fragmentação com base em um guia, não fazendo distinção sobre fragmentação vertical e horizontal. Também somente é aplicado a repositórios SD.

Amer-Yahia e Kotidis (AMER-YAHIA, KOTIDIS, 2004) apesar de tratar a fragmentação de documentos, dão ênfase a dados XML expostos através de serviços web. Apesar de ser um trabalho interessante, dada a arquitetura criada por eles, suas operações e o modelo de custos criado não são o objetivo dessa dissertação. Já a QSSL (Query Set Specification Language) (PETROPOULOS, DEUTSCH et al, 2003), apesar de não definir fragmentação, pode ser usada com esse intuito, já que define como representar consultas pré-formatadas em XQuery.

Embora adequada aos requisitos do P2P, a solução proposta em (BONIFATI, MATRANGOLO et al, 2004) não se aplica ao nosso cenário, pois estamos interessados em fragmentar utilizando diferentes tipos de fragmentação (i.e. vertical, horizontal e híbrida) em diferentes tipos de repositório (i.e. SD e MD) visando atender um maior número de casos. A solução apresentada por Bonifati e outros foca numa fragmentação vertical (já que as tabelas *hash* são construídas sobre as expressões de caminho) e se aplica a repositórios SD. Em (ABITEBOUL, BONIFATI et al, 2003) os autores apresentam o AXML, que também é uma abordagem P2P. Mais uma vez, uma abordagem interessante, mas não define o que seriam os tipos de fragmentação nesse cenário. O foco é uma fragmentação vertical do que pode ser gerado em sítios remotos através de chamadas existentes no documento. O principal objetivo deles é manter o documento XML o mais atualizado possível.

Nenhum desses trabalhos atinge todos os objetivos da presente dissertação, entre eles a definição dos vários tipos de fragmentos para XML e seu uso em repositórios SD e MD. Alguns, como (BREMER, GERTZ, 2003) foram utilizados como guias. Além disso, em nenhum dos trabalhos pesquisados foram encontradas análises sobre o tempo de resposta do processamento de consultas em repositórios XML fragmentados.

Capítulo 3 - Projeto de Fragmentação em Bases XML

3.1 Introdução

Neste capítulo vamos apresentar nossa proposta para o projeto de fragmentação em bases de coleções de documentos XML. O projeto de fragmentação tem por objetivo definir as funções de fragmentação a serem aplicadas sobre as coleções de documentos para gerar os fragmentos. Devido à diversidade de definições formais para documentos, coleções e álgebras XML, dedicamos as primeiras seções às definições dos conceitos básicos sobre os quais os fragmentos serão definidos. Assim, a seção 3.2 descreve primeiramente o modelo de dados utilizado para os documentos XML considerados. Ele é importante para sabermos quais elementos do XML estão sendo tratados dentro do documento. A seguir, a seção 3.3 traz a definição para o que chamamos de Coleção, peça fundamental em nossa proposta de formalização de fragmentos XML. Sem o conceito de coleção de documentos, não poderemos utilizar os operadores da TLC na especificação dos fragmentos. Também mostramos os dois esquemas dos documentos XML que serão utilizados em nossos exemplos. A seção 3.4 traz esclarecimentos sobre quais são os tipos de expressões de caminho que serão válidas para utilizarmos nas nossas especificações.

Em seguida à apresentação da álgebra utilizada e à correspondência com nossos operadores na seção 3.5, apresentamos a proposta de definição de fragmentos na seção 3.6. Na seção 3.7, discutimos alguns guias para a validação das regras de correção que devem ser aplicadas aos fragmentos gerados. A seção 3.8 conduz uma discussão sobre considerações acerca da validade dos documentos, já que em alguns casos teremos dificuldades em mantê-los válidos.

3.2 Modelo de Dados XML

Documentos XML são representados por árvores com nodos rotulados por nomes de elementos, atributos ou valores constantes. Seja $\mathcal{L}$ o conjunto distinto de nomes de elementos, $\mathcal{A}$ o conjunto distinto de nomes de atributos, e $\mathcal{D}$ o conjunto distinto de valores de dados. Uma *árvore de dados XML* é denotada pela expressão $\Delta :=$

$\langle t,\ell,\Psi\rangle$, onde: t é uma árvore ordenada finita, ℓ é uma função que rotula os nodos em t com símbolos em $\mathcal{L} \cup \mathcal{A}$; e Ψ é uma função que mapeia nodos folha em t para valores em $\mathcal{D}$. O nodo raiz de Δ é denotado por $raiz_\Delta$. Assumimos que nodos em Δ não possuem conteúdo misto; se um dado nodo v é mapeado em $\mathcal{D}$, então v não tem descendentes em Δ. Note que, entretanto, isto não é uma limitação. Nós preferimos não considerar o conteúdo misto, para simplificar a apresentação de nossas idéias. No momento em que estivermos discutindo os tipos de fragmentação, iremos então discutir sobre os elementos de conteúdo misto e o que acarreta a remoção dessa limitação. Além disso, nodos com rótulos em $\mathcal{A}$ têm um único filho, o qual deve estar rotulado em $\mathcal{D}$. Um documento XML é uma árvore de dados.

3.3 Coleções

Basicamente, nomes de elementos XML correspondem a nomes de tipos de dado, descritos em uma DTD ou XML Schema. Seja S um esquema. Dizemos que o documento $\Delta := \langle t,\ell,\Psi\rangle$ *satisfaz* um tipo τ, onde $\tau \in S$, se e somente se $\langle t,\ell\rangle$ é uma árvore derivada da gramática definida por S tal qual $\ell(raiz_\Delta) \rightarrow \tau$. Uma *coleção* de documentos XML representa um conjunto de árvores de dados. Dizemos que ela é *homogênea* se todos os documentos na coleção C satisfazem o mesmo tipo de documento XML. Se não, dizemos que a coleção é *heterogênea*. Dado um esquema S, uma coleção homogênea C é denotada pela expressão $C := \langle S, \tau_{raiz}\rangle$, onde τ_{raiz} é um tipo em S e todas as instâncias de Δ em C satisfazem τ_{raiz}.

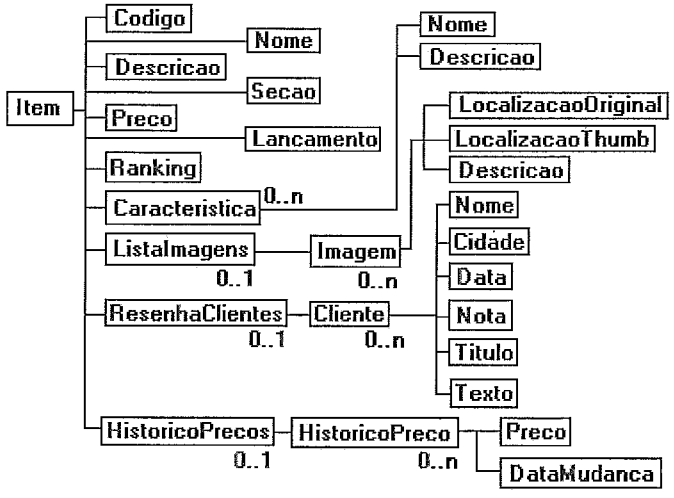


Figura 1: Tipo *Item* do Esquema $S_{loja\_virtual}$

A Figura 1 e a Figura 2 ilustram o esquema $S_{loja\_virtual}$, que utilizamos para ilustrar os exemplos ao longo desta dissertação. Nessas figuras, indicamos a cardinalidade mínima e máxima (assumindo a cardinalidade 1..1 quando omitida). Esta notação será utilizada ao longo do texto. Os dois tipos principais em $S_{loja\_virtual}$ são Loja e Item, os quais descrevem uma loja virtual e os itens que esta oferece ao público. Cada item está associado a uma seção e pode ter características descritivas.

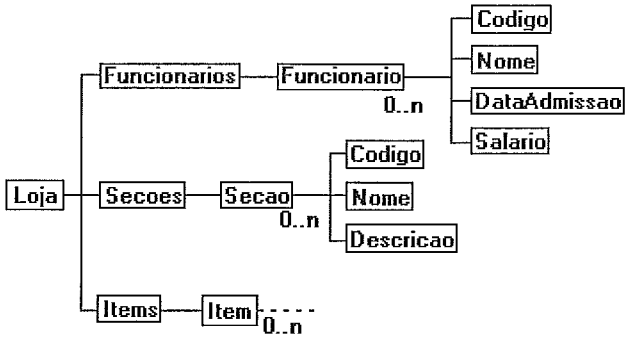


Figura 2: Tipo *Loja* do Esquema $S_{loja\_virtual}$

Os itens também podem possuir uma lista de imagens para serem utilizadas na loja virtual, uma lista com resenha de clientes que já compraram o produto e um histórico de preços, para uso interno da loja. Toda loja possui uma lista dos itens que ela vende, a lista dos funcionários que trabalham nela e as seções que ela possui. Como pode ser observado, o tipo Loja utiliza o tipo Item na sua definição da lista de itens utilizados por ela. Com isso, o esquema $S_{loja\_virtual}$ é capaz de definir duas coleções distintas, conforme ilustrado na Figura 3. São estas as coleções homogêneas C_{loja} e C_{itens}.

$$C_{loja} := \langle S_{virtual\_loja}, \text{Loja} \rangle$$
$$C_{itens} := \langle S_{virtual\_loja}, \text{Item} \rangle$$

Figura 3: Especificação das coleções C_{loja} e C_{itens}

Yao et al, (YAO, ÖZSU et al, 2004) definem uma classificação para repositórios XML. Um repositório XML pode ser composto por vários documentos, assim dito repositório de Múltiplos Documentos (*Multiple Documents*, MD); ou por um único grande documento que contém toda a informação necessária, dito repositório de Documento Único (*Single Document*, SD). Um repositório seria o local onde estão armazenados os documentos XML.

Nesta dissertação faremos uso uso dessa classificação para qualificar nossa definição de coleção. Assim, uma coleção poderá ser também SD ou MD. A coleção C_{itens}, ilustrada na Figura 3 corresponde a um repositório MD, enquanto que a coleção C_{loja} é um repositório SD.

3.4 Expressões de caminho

Uma *expressão de caminho* P é definida como sendo uma seqüência $/e_1/.../\{e_k|@a_k\}$, onde $e_x \in \mathcal{L}$, $1 \leq x \leq k$, e $a_k \in \mathcal{A}$. P pode opcionalmente conter os símbolos "*" para indicar qualquer elemento, e "//" para indicar qualquer seqüência de elementos descendentes. Além disso, o termo $e(i)$ pode ser usado para denotar a i-ésima ocorrência do elemento e. A avaliação da expressão de caminho P em um documento Δ representa a seleção de todos os nodos com o rótulo e_k (ou a_k) os quais os passos a partir da $raiz_\Delta$ satisfazem P. P é dito ser um terminal se o conteúdo dos nodos selecionados é simples (isto é, se eles tem um domínio em $\mathcal{D}$).

Podemos definir um *predicado simples* p como a expressão lógica:

$$p := P\ \theta\ valor \mid \phi_v(P)\ \theta\ valor \mid \phi_b(P) \mid Q$$

onde P é uma expressão de caminho terminal, $\theta \in \{=, <, >, \neq, \geq, \leq\}$, $valor \in \mathcal{D}$, ϕ_v é uma função que retorna valores em $\mathcal{D}$, ϕ_b é uma função booleana e Q denota uma expressão de caminho arbitrária. Neste último caso, p é verdadeiro para todos os nodos que são selecionados por Q (teste existencial).

É importante notar que esses predicados serão utilizados também nos padrões de árvore. Os autores da TAX (JAGADISH, LAKSHMANAN et al, 2001) definem os predicados que podem ser utilizados, mas compreendem que a lista de predicados possíveis é extensível e não afeta diretamente a álgebra, ou seja, a definição dos predicados é ortogonal à mesma.

3.5 O uso da Álgebra TLC

Escolhemos a TLC para ser a álgebra que será a base da nossa definição teórica de fragmentos XML. Esta álgebra opera sobre coleções de árvores, além de possuir operadores semelhantes aos já existentes no modelo relacional. Essa semelhança

traz diversas vantagens e torna a especificação um pouco mais intuitiva, porém, não de todo simples.

Uma fragmentação horizontal deve ser feita através da seleção de parte da coleção original com base em um conjunto de predicados. Porém, o operador de seleção da TLC possui como parâmetro um padrão de árvore anotado (*Annotated Pattern Tree* ou APT). Surge uma questão: Como adaptá-lo para que a definição da fragmentação horizontal seja simples e feita sobre predicados? Como pressupomos que existe um esquema único e conhecido para a coleção, é possível gerar um APT que reflita toda a estrutura da coleção. Logo, o APT é sempre o mesmo e conhecido para determinada coleção.

Tabela 2: Sintaxe dos operadores da TLC e utilizados nesta dissertação.

	Projeção	**Seleção**
TLC	P(nl)(C)	S(apt)(C)
Nosso operador	$\pi_{P,\Gamma}$	σ_{μ}

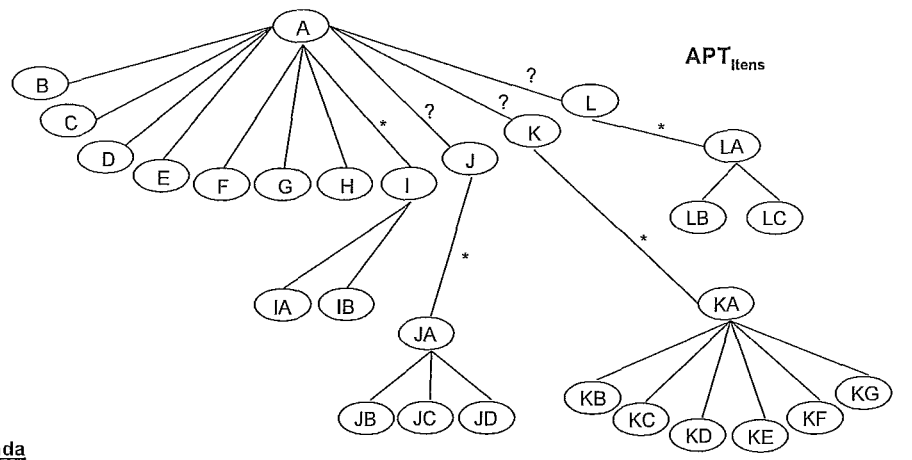
O APT carrega especificações de predicados para filtrar quais árvores são as desejadas. Então, dada uma lista de predicados na forma de expressões de caminho válidas para o esquema da coleção, podemos inserir no padrão de árvore os predicados necessários. Assim, conseguimos simplificar a notação do operador, para apenas utilizarmos uma lista de predicados. Na Tabela 2 temos a sintaxe do operador que utilizamos e o operador da TLC. A sintaxe dos operadores da TLC pode ser encontrada em (PAPARIZOS, WU et al, 2004). Na Figura 4 temos um exemplo da APT para o esquema de itens. Na Figura 5, o exemplo da definição de um fragmento sobre um determinado predicado e como este se apresentaria na APT.

Na Tabela 2, o operador *P* da TLC realiza uma projeção a partir da coleção de documentos especificada em *C*. Os elementos projetados correspondem às classes lógicas da lista *nl*. Já o operador de seleção é denotado por S. A seleção é feita sobre a coleção de documentos especificada em *C*, com base no padrão de árvore *apt* fornecido.

Uma fragmentação vertical, no modelo relacional, quebra a estrutura da relação. Desejamos a mesma semântica para a fragmentação vertical no modelo XML, ou seja, a quebra da estrutura da coleção. Para isso, utilizaremos um operador de

projeção, como utilizado em outros modelos. Desejamos que o operador de projeção especifique, através de XPath, quais são os caminhos de poda da árvore.

A TLC possui um operador de projeção que projeta os nodos de uma lista de classes lógicas. Contudo, como iremos adaptá-lo para que seja usado de forma simples, e nos permita especificar apenas pontos de poda da árvore? Mais uma vez, conhecemos a estrutura do esquema, e conseqüentemente temos o APT para ele, onde cada nodo possui uma classe lógica. Então, através das expressões de caminho, podemos gerar a lista de nodos que devem ser projetados. Na Tabela 2 temos a sintaxe do operador que utilizamos e o operador da TLC. Na Tabela 3 temos um exemplo de fragmentação vertical e como seria traduzido para o nosso operador e no operador da TLC. Utilizamos o APT do esquema de itens, mostrado na Figura 4. A numeração das classes lógicas é facilmente obtida através da ordem alfabética dos elementos.



Legenda

A: Item	IB: Descricao	KD: Data
B: Codigo	J: ListaImagens	KE: Nota
C: Nome	JA: Imagem	KF: Titulo
D: Descricao	JB: LocalizacaoOriginal	KG: Texto
E: Secao	JC: LocalizacaoThumb	L: HistoricoPrecos
F: Preco	JD: Descricao	LA: HistoricoPreco
G: Lancamento	K: ResenhaClientes	LB: Preco
H: Ranking	KA: Cliente	LC: DataMudanca
I: Caracteristica	KB: Nome	
IA: Nome	KC: Cidade	

Figura 4: APT para a coleção C_{Itens}

O objetivo com a re-escrita dos operadores é apenas facilitar a escrita das operações de fragmentação, para que estes fiquem mais intuitivos. Os operadores da TLC demandam sempre uma APT, o que tornaria confuso a todo o momento especificá-la. Além do fato de que, sempre consideraríamos todo um esquema e não partes específicas dele como seria a tradução normal da XQuery.

Tabela 3: Exemplo das Definições de Fragmento Vertical em TLC

TLC	P(12,13,14,15,16)(C_{Itens})
Nosso operador	$\pi_{/Item/ListaImagens,\{\}}$

Fragmento: Todos os itens que são lançamentos

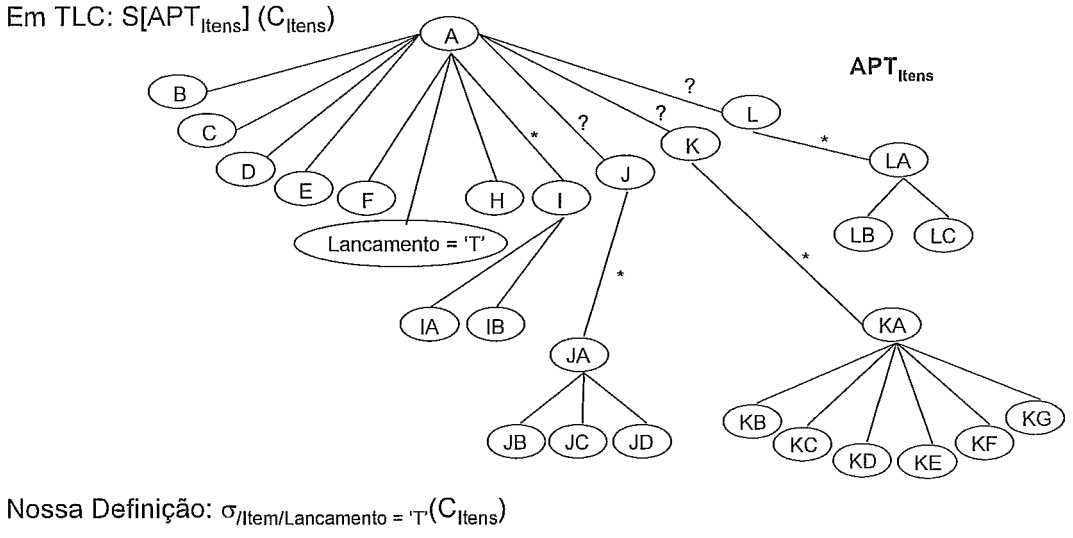


Figura 5: Exemplo das Definições de Fragmento Horizontal em TLC

3.6 Especificação dos Fragmentos

Tradicionalmente, podemos ter três tipos de fragmentação: *horizontal*, onde instâncias são agrupadas por predicados de seleção; *vertical*, que "divide" a estrutura de dados através de projeções; e *híbrida*, que combina as operações de seleção e projeção em sua definição. Nossa definição de fragmentação segue a semântica dos operadores da álgebra TAX, já que esta é uma das poucas álgebras XML que usam coleções de documentos e que é adequada ao modelo de dados XML definido na seção 3.2.

Em repositórios XML, consideramos que a fragmentação é definida sobre o esquema de uma coleção XML. No caso de um repositório XML MD, assumimos que a fragmentação somente pode ser aplicada em coleções homogêneas. Isso garante que, ao fragmentar o esquema, estaremos fragmentando todos os documentos que o compõem, não sendo necessário, então, um tipo de fragmentação para cada documento distinto da coleção.

Definição 1: *Um fragmento F de uma coleção homogênea C é uma coleção representada por F:= ⟨C, γ⟩, onde γ denota uma operação definida sobre C. F é horizontal se γ denota uma seleção; vertical, se o operador γ é uma projeção; ou híbrido, quando ocorre uma composição de operadores de seleção e projeção.*

As instâncias de um fragmento F são obtidas aplicando-se γ a cada documento em C. O conjunto dos documentos resultantes forma o fragmento F. Um fragmento F é válido se todos os documentos gerados por γ são bem-formados (W3C, 2005) (em particular, eles devem possuir uma raiz única). Nas seções 3.6.1, 3.6.2 e 3.6.3 detalharemos os principais tipos de fragmentação em XML.

3.6.1 Fragmentação Horizontal

O principal objetivo da fragmentação horizontal é agrupar as informações que são freqüentemente acessadas por um conjunto de consultas com um determinado predicado de seleção. Um fragmento horizontal F de uma coleção C é definido pelo operador de seleção (σ) aplicado sobre os documentos em C, onde o predicado de σ é uma expressão booleana com um ou mais predicados simples. Neste caso, F tem o mesmo esquema que C. Um detalhe importante que será observado ao longo do texto é que uma fragmentação horizontal não se aplica a coleções do tipo SD.

Definição 2: *Seja μ uma conjunção de predicados simples sobre a coleção C. O fragmento horizontal de C definido por μ é dado pela expressão* $F := \langle C, \sigma_\mu \rangle$, *onde* σ_μ *denota a seleção de documentos em C que satisfazem μ, isto é, F contém documentos de C para os quais* σ_μ *é verdadeiro.*

(a)	$F1_{CD} := \langle C_{itens}, \sigma_{/Item/Secao="CD"} \rangle$ $F2_{CD} := \langle C_{itens}, \sigma_{/Item/Secao\neq"CD"} \rangle$
(b)	$F1_{banda} := \langle C_{itens}, \sigma_{contains(//Descricao, "banda")} \rangle$ $F2_{banda} := \langle C_{itens}, \sigma_{not(contains(//Descricao, "banda"))} \rangle$
(c)	$F1_{com\_imagens} := \langle C_{itens}, \sigma_{/Item/ListaImagens} \rangle$ $F2_{com\_imagens} := \langle C_{itens}, \sigma_{empty(/Item/ListaImagens)} \rangle$

Figura 6: Exemplos de alternativas de definição de fragmentos sobre a coleção C_{itens}

A Figura 6 apresenta a especificação de três alternativas para a fragmentação horizontal da coleção C_{itens} da Figura 1. A escolha entre as alternativas irá depender das consultas mais freqüentes. Por exemplo, o fragmento $F1_{banda}$ (Figura 6 (b)) agrupa documentos de C_{itens} que têm nodos Descricao que satisfazem a expressão de

caminho //Descricao (isto é, Descricao pode estar em qualquer nível em C_{itens}) e que contém a palavra "banda". Alternativamente, pode existir o interesse em separar, em diferentes fragmentos, documentos que têm ou não uma dada estrutura. Isto pode ser feito através de um teste existencial, que é ilustrado na Figura 6(c). Embora $F1_{com\_imagens}$ e C_{itens} tenham o mesmo esquema, na prática, documentos em conformidade com esse esquema podem ter instâncias de documentos com diferentes estruturas, já que o elemento utilizado no teste existencial é necessário em $F1_{com\_imagens}$, e em $F2_{com\_imagens}$, ele deve ser vazio. Observe que $F1_{com\_imagens}$ não pode ser classificado com um fragmento vertical nem mesmo híbrido, pois a estrutura do esquema não foi alterada.

É importante notar que, por definição, coleções SD não podem ser fragmentados horizontalmente, já que a fragmentação horizontal é definida sobre o esquema dos documentos (ao invés de nodos). Entretanto, os elementos em uma coleção SD podem estar distribuídos em fragmentos usando fragmentação híbrida, conforme será descrito mais adiante.

Não é possível especificar fragmentos horizontais sobre um elemento que possui cardinalidade superior a 1, como é o caso dos elementos descendentes de /Item/ListaImagens/Imagem (ver Figura 7(a)). O objetivo desta restrição é garantir que os fragmentos sejam disjuntos. Uma ressalva a esta restrição é a especificação da ordem do elemento que é utilizado como predicado de seleção. Seguindo o exemplo, se utilizarmos /Item/ListaImagens/Imagem(1) (ver Figura 7(b)) será possível utilizar qualquer um dos seus descendentes como predicado de seleção. A semântica dos nossos exemplos envolve a localização das imagens. Queremos separar os itens que possuem imagens de alta resolução dos que possuem baixa resolução.

(a)
$$F1_{imagens} := \langle C_{itens}, \sigma_{\text{contains(/Item/ListaImagens/Imagem/LocalizacaoOriginal, "/HighRes/")}} \rangle$$
$$F2_{imagens} := \langle C_{itens}, \sigma_{\text{not(contains(/Item/ListaImagens/Imagem/LocalizacaoOriginal, "/LowRes/"))}} \rangle$$

(b)
$$F1_{imagens} := \langle C_{itens}, \sigma_{\text{contains(/Item/ListaImagens/Imagem(1)/LocalizacaoOriginal, "/HighRes/")}} \rangle$$
$$F2_{imagens} := \langle C_{itens}, \sigma_{\text{not(contains(/Item/ListaImagens/Imagem(1)/LocalizacaoOriginal, "/LowRes/"))}} \rangle$$

Figura 7: Definição de Fragmento Horizontal, com(a) e sem(b) problemas de disjunção

Na Figura 8 temos representados dois documentos da coleção C_{itens}. Se aplicarmos a fragmentação definida na Figura 7(a) a essa coleção, teremos que o Item de código "001" irá fazer parte de dois fragmentos ao mesmo tempo, formando assim, fragmentos não disjuntos, já que este item possui tanto imagens de alta resolução como

de baixa. Utilizando a fragmentação definida na Figura 7(b), os fragmentos serão disjuntos, já que o primeiro filho do elemento ListaImagens é que irá determinar aonde os documentos serão colocados. Assim, o Item de código "001" cairia no fragmento $F2_{Imagens}$ e o Item de código "002" no fragmento $F1_{Imagens}$.

```
<Item>
 <Codigo>001</Codigo>
 <Nome>Once - Nightwish</Nome>
 ...
 <ListaImagens>
 <Imagem>
  <LocalizacaoOriginal>/LowRes/Once001.jpg<LocalizacaoOriginal>
  ...
 </Imagem>
 <Imagem>
  <LocalizacaoOriginal>/HighRes/Once002.jpg<LocalizacaoOriginal>
  ...
 </Imagem>
 </ListaImagens>
 ...
</Item>
<Item>
 <Codigo>002</Codigo>
 <Nome>Wishmaster - Nightwish</Nome>
 ...
 <ListaImagens>
  <Imagem>
   <LocalizacaoOriginal>/HighRes/Wishmaster001.jpg<LocalizacaoOriginal>
   ...
  </Imagem>
  <Imagem>
   <LocalizacaoOriginal>/HighRes/Wishmaster002.jpg<LocalizacaoOriginal>
   ...
  </Imagem>
 </ListaImagens>
 ...
</Item>
```

Figura 8: Exemplos de Documentos que podem ferir a disjunção da fragmentação

3.6.2 Fragmentação Vertical

A fragmentação vertical é obtida pela aplicação do operador de projeção (π) para podar a estrutura de dados em partes menores, que são freqüentemente acessadas em consultas. Em repositórios XML, o operador de projeção possui uma semântica mais sofisticada do que a utilizada em outros modelos de dados. É possível especificar projeções que excluem sub-árvores, cuja raiz está localizada em qualquer nível da árvore XML. Uma projeção sobre uma coleção C recupera, em cada documento de C (note que C pode ter um único documento, no caso de ser do tipo SD), um conjunto de sub-árvores representado por uma expressão de caminho, que são possivelmente podadas em algum nodo filho.

Definição 3: *Seja P uma expressão de caminho sobre a coleção C. Seja* $\Gamma = \{E_1, ..., E_x\}$ *um (possivelmente vazio) conjunto de expressões de caminho contidos em P (isto é, expressões de caminho em que P é um prefixo). Um fragmento vertical de C definido por P é caracterizado por* $F := \langle C, \pi_{P,\Gamma} \rangle$, *onde* $\pi_{P,\Gamma}$ *caracteriza a projeção das sub-árvores enraizadas pelos nodos selecionados por P, excluindo do resultado os nodos selecionados pelas expressões em* Γ. *O conjunto* Γ *é chamado de* critério de poda *de F.*

É importante notar que não mencionamos a existência de uma chave primária que identifique cada documento da coleção. Em outros modelos, a existência dessa chave se faz necessária, porém, no modelo XML isso irá depender do tipo de coleção a ser fragmentada. Quando estamos lidando com repositórios SD, a distinção entre documentos não se faz necessária, já que cada coleção possui apenas um único documento.

Quando trabalhamos com repositórios MD, é necessária uma chave para identificar as sub-árvores do mesmo documento que foram espalhadas pelos fragmentos. Caso o documento possua um elemento que represente a chave, este deveria estar presente em todos os fragmentos, porém, acreditamos que tal abordagem não seja muito interessante. Se esta chave estiver num caminho relativamente longo a partir da raiz, pode ser que existam dificuldades nas operações de junção. Além disso, todos os fragmentos teriam de carregar essa chave. Uma solução mais simples é a identificação de cada documento por uma chave interna no SGBD, como o OID existente no modelo Orientado a Objetos. Com isso, cada documento, ao ser fragmentado, carregaria essa mesma chave para todos os fragmentos, o que facilitaria as operações de junção.

Contudo, essa abordagem nos levaria à seguinte restrição. A expressão de caminho *P* não pode recuperar nodos que possam ter cardinalidade maior do que 1. Por exemplo, se criarmos um fragmento onde somente exista o caminho /Item/ListaImagens/Imagem (na Figura 1), criaremos um documento sem raiz. Podemos considerar que cada caminho selecionado para o fragmento gere um novo documento, e assim, possua uma raiz única com um identificador. Porém, desejamos que exista uma correspondência um para um entre os fragmentos. Além disso, neste caso, não seria possível garantir a reconstrução do documento na ordem correta. Então, esta restrição assegura que a fragmentação irá resultar em documentos bem-formados, sem a necessidade de gerar elementos artificiais para reorganizar as sub-árvores

projetadas no fragmento. Uma exceção é feita quando nós indicamos a ordem do elemento (p.ex.. /Item/ListaImagens/Imagem(1)). Neste caso, sabemos exatamente a ordem, e somente um caminho será selecionado.

Para a reconstrução do documento original, iremos utilizar operações de junção, partindo do pressuposto que cada fragmento possua um ID em comum, este sendo o ID do documento original. Caso o elemento raiz do fragmento criado possua cardinalidade 0 ou 1, existe a possibilidade de não existir um documento no fragmento correspondente, já que no documento original o elemento pode não existir. Esse exemplo pode ser observado na Figura 9, onde o elemento ListaImagens pode não existir. Nesse caso, ao invés de uma operação normal de junção, deve ser realizada uma junção externa.

Para a geração deste ID temos várias alternativas possíveis. Podemos utilizar um elemento que possua um valor único, e utilizá-lo como chave primária. Neste caso, o esquema XML deve ser especificado em XML Schema, e a coleção deve ter um elemento definido como *Unique*. Se for especificado em DTD, o elemento deve ter sido especificado como um *Key*. Outra alternativa é a utilização de características próprias ao SGBD XML sendo utilizado. O Tamino (SCHÖNING, 2001) utiliza identificadores para cada documento. Já o eXist utiliza o nome do documento de origem para identificá-los unicamente dentro de uma coleção.

Cada abordagem tem seus prós e contras. Utilizar as próprias características do documento possui vantagens semânticas, mas, em alguns casos pode ser impossível identificar uma chave que garanta unicidade. E é claro, chaves com muitos elementos podem acarretar problemas com junções. Já uma abordagem que utiliza recursos do SGBD para identificar os documentos é oposta em termos de características. As chaves não possuem nenhuma semântica, o que em alguns casos pode não ser desejado. Ao ponto que, criar e gerenciar esses identificadores únicos é bem mais simples e menos custoso em termos de junções.

Temos em vista que a implementação dos fragmentos é um assunto interno do SGBD. Estes mecanismos devem estar transparentes para o usuário (um dos princípios básicos da fragmentação de bases de dados e processamento de consultas

distribuído). Por isso escolhemos utilizar estruturas internas do próprio SGBD para gerenciar esses documentos fragmentados.

(a)	$F1_{loja} := \langle C_{loja}, \pi_{/Loja,\{ /Loja/Secoes , /Loja/IItens \}} \rangle$ $F2_{loja} := \langle C_{loja}, \pi_{/Loja/Secoes} \rangle$ $F3_{loja} := \langle C_{loja}, \pi_{/Loja/IItens} \rangle$
(b)	$F1_{itens} := \langle C_{itens}, \pi_{/Item, \{/Item/ListaImagens\}} \rangle$ $F2_{itens} := \langle C_{itens}, \pi_{/Item/ListaImagens} \rangle$

Figura 9: Exemplos de definições de fragmentos verticais nas coleções C_{loja} e C_{itens}

A Figura 9 mostra exemplos de definição de fragmentos verticais das coleções C_{loja} e C_{itens}, definidas na Figura 2 e Figura 1, respectivamente. O fragmento $F2_{itens}$ representa documentos que contêm todos os nodos ListaImagens que satisfazem a expressão de caminho /Item/ListaImagens na coleção C_{itens} (nenhum critério de poda foi utilizado). Por outro lado, os nodos que satisfazem /Item/ListaImagens são exatamente os nodos que são podados das sub-árvores enraizadas por /Item no fragmento $F1_{itens}$, preservando assim a disjunção entre os documentos dos dois fragmentos. É importante notar que a raiz da árvore gerada é indicada pela primeira parte do operador.

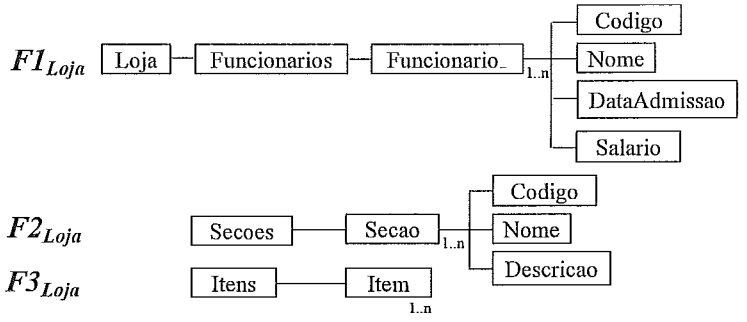


Figura 10: Fragmentação Vertical sobre Cloja

No exemplo (a) da Figura 9, podemos notar que não é necessário mencionar a poda das sub-árvores na coleção C_{loja}, já que estamos separando cada elemento que é a raiz de um conjunto de informações. Na Figura 10 e Figura 11 temos, respectivamente, a representação em forma de árvore para os exemplos (a) e (b) da Figura 9.

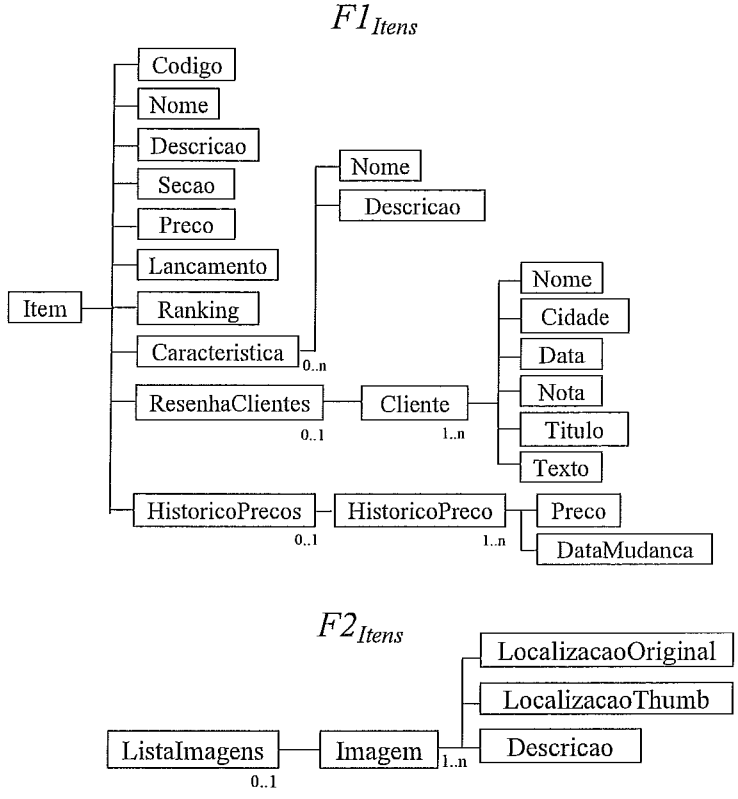


Figura 11: Fragmentação Vertical sobre C_{Itens}

3.6.3 Fragmentação Híbrida

A fragmentação híbrida de uma coleção XML pode ser definida pela aplicação de uma fragmentação vertical seguida por uma fragmentação horizontal, ou de uma fragmentação horizontal seguida por uma fragmentação vertical. Um uso interessante deste tipo seria a implementação de fragmentos horizontais sobre esquemas de coleções de repositórios SD. Nesses casos, a fragmentação horizontal é impedida pela existência de um único documento. A fragmentação híbrida quebra a estrutura do documento original, permitindo formar fragmentos horizontais sobre fragmentos verticais que caracterizam uma coleção "do tipo MD".

Definição 4: *Sejam* σ_μ *e* $\pi_{P,\Gamma}$ *os operadores de seleção e projeção, respectivamente, definidos sobre a coleção C. Um fragmento híbrido de C é representado por* $F := \langle C, \pi_{P,\Gamma} \bullet \sigma_\mu \rangle$, *onde* $\pi_{P,\Gamma} \bullet \sigma_\mu$ *caracteriza a seleção das sub-árvores projetadas por* $\pi_{P,\Gamma}$ *que satisfazem* σ_μ.

É importante notar que a ordem na aplicação das operações em $\pi_{P,\Gamma} \bullet \sigma_\mu$ depende do projeto de fragmentação, uma vez que geram resultados distintos. Neste

exemplo, primeiro é realizada a projeção, e na coleção resultante desta, é realizada a seleção.

$$F1_{itens} := \langle C_{Loja}, \pi_{/Loja/Items, \{\}} \bullet \sigma_{/Item/Secao="CD"} \rangle$$
$$F2_{itens} := \langle C_{Loja}, \pi_{/Loja/Items, \{\}} \bullet \sigma_{/Item/Secao="DVD"} \rangle$$
$$F3_{itens} := \langle C_{Loja}, \pi_{/Loja/Items, \{\}} \bullet \sigma_{/Item/Secao \neq "CD" \wedge /Item/Secao \neq "DVD"} \rangle$$
$$F4_{itens} := \langle C_{Loja}, \pi_{/Loja, \{/Loja/Itens\}} \rangle$$

Figura 12: Exemplo de Fragmentação Híbrida sobre a coleção C_{Loja}

A Fragmentação Híbrida deve seguir os mesmos cuidados já citados nas demais fragmentações. Na Figura 12 temos um exemplo de fragmentação híbrida sobre a coleção C_{Loja}. Desejamos que seja feita a separação dos itens dos demais elementos do documento. Além disso, faz-se necessária a separação em fragmentos menores dos itens da seção de CDs e DVDs. Note que o fragmento $F4_{itens}$ é o fragmento complementar da fragmentação vertical realizada, assim como o $F3_{itens}$, é o complemento da fragmentação horizontal realizada. A Figura 13 mostra como fica o esquema fragmentado.

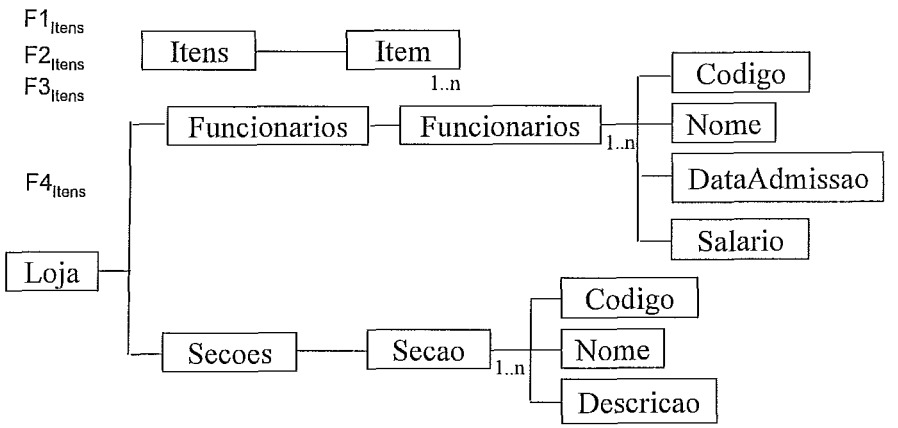


Figura 13: Esquema C_{Loja} após a fragmentação híbrida.

3.7 Regras de Correção

A definição de como fragmentar uma coleção XML e seus operadores ficaria incompleta se não apresentássemos regras de correção desta fragmentação. Como utilizamos uma álgebra semelhante à álgebra relacional (a TLC), utilizando projeções e seleções, iremos definir regras de correção também semelhantes às utilizadas no modelo relacional para validar a fragmentação.

3.7.1 Completude

Se uma coleção C é decomposta em fragmentos F_1, F_2, ..., F_n, cada item de dado que pode ser encontrado em C também tem que poder ser encontrado em um dos F_i. $(1 \leq i \leq n)$.

Esta propriedade é importante para a fragmentação, já que garante que os dados da coleção global podem ser mapeados para os fragmentos, sem nenhuma perda. Nota-se que, no caso da fragmentação horizontal, o "item de dado" refere-se a um documento, enquanto que na fragmentação vertical, refere-se a uma sub-árvore.

3.7.2 Reconstrução

Se uma coleção C é decomposta em um conjunto de fragmentos $\Phi = \{F_1, F_2, ..., F_n\}$, deve ser possível definir um operador Δ tal que:

$$C = \Delta F_i, \forall F_i \in \Phi$$

O operador Δ será diferente para diferentes tipos de fragmentação. É importante, porém, que este operador exista na álgebra adotada. Para a fragmentação horizontal, este operador será a União; e para a fragmentação vertical, será a junção. Ambos os operadores podem ser encontrados na definição da TAX e da TLC.

Em (JAGADISH, LAKSHMANAN et al, 2001) é dito que o comportamento das operações de conjunto (União) é praticamente o mesmo do modelo relacional. O único problema é como identificar que duas árvores de dados são idênticas. Como assumimos que cada documento possui um identificador único, o qual é preservado para cada fragmento, este pode ser utilizado para determinar a diferença entre duas árvores. Uma outra solução é aplicar a união sem remoção de duplicações, e após isso, executar a operação de eliminação de duplicatas definida pela TLC. A diferenciação é feita com base no conjunto de nodos passados como referência ao operador. A operação de junção é descrita pela TLC em (PAPARIZOS, WU et al, 2004).

3.7.3 Disjunção

Se a coleção C é horizontalmente decomposta em um conjunto de fragmentos $\Phi = \{F_1, F_2, ..., F_n\}$, e o documento d_i está em F_j, então este não pode estar

em nenhum outro fragmento F_k (k $\neq$ j). Este critério garante que os fragmentos horizontais são disjuntos.

Se a coleção C é verticalmente decomposta em um conjunto de fragmentos $\Phi = \{F_1, F_2, ..., F_n\}$, temos que ter cuidados adicionais. Precisamos, para isso, de uma lista com todas as expressões de caminho P que levem a nodos terminais p_i. Então, os fragmentos são disjuntos se a expressão de caminho p_i que está no fragmento Fj não estiver em mais nenhum outro fragmento F_k (k $\neq j$).

3.7.4 Verificação das regras de correção

Apesar de estarmos apenas definindo as regras, e não uma metodologia para um projeto de fragmentação, sentimos a necessidade de esclarecer alguns pontos com respeito à verificação das regras de integridade.

Inicialmente, vamos discutir como verificar uma definição de fragmentação horizontal. A completude e a disjunção podem ser verificadas através de operações sobre os predicados. Como temos a lista de predicados e o domínio de cada elemento, podemos determinar se todos os valores do domínio foram contemplados. Caso não sejam, pode ser criado um fragmento especial para conter os valores restantes do domínio. Isso garante a completude. A disjunção pode ser verificada de forma semelhante, garantindo que dois valores do domínio não estejam ao mesmo tempo em mais de um predicado. É importante ressaltar que nem sempre essas verificações vão ser facilmente computáveis. A reconstrução é garantida pelo operador de União da álgebra.

Quanto à verificação da fragmentação vertical, esta deve ser verificada com mais cuidado. Sabemos que um documento XML é uma árvore. Então, com base no esquema, temos a estrutura dessa árvore. Então podemos executar um algoritmo simples de busca em árvore, que gere como saída uma lista com todos os caminhos até todos os elementos folha da árvore. Chamaremos essa lista de ListaCaminhos(C_{Itens}). A Tabela 4 apresenta essa lista de caminhos para a coleção C_{Itens}.

Tabela 4: Expressões de Caminho para a coleção C_{Itens}

1	/Item/Codigo	11	/Item/ListaImagens/Imagem/LocalizacaoThumb
2	/Item/Nome	12	/Item/ListaImagens/Imagem/Descricao
3	/Item/Descricao	13	/Item/ResenhaClientes/Cliente/Nome
4	/Item/Secao	14	/Item/ResenhaClientes/Cliente/Cidade
5	/Item/Preco	15	/Item/ResenhaClientes/Cliente/Data
6	/Item/Lancamento	16	/Item/ResenhaClientes/Cliente/Nota

7	/Item/Ranking	17	/Item/ResenhaClientes/Cliente/Titulo
8	/Item/Característica/Nome	18	/Item/ResenhaClientes/Cliente/Texto
9	/Item/Característica/Descricao	19	/Item/HistoricoPrecos/HistoricoPreco/Preco
10	/Item/ListaImagens/Imagem/LocalizacaoOriginal	20	/Item/HistoricoPrecos/HistoricoPreco/DataMudanca

Agora, com base no esquema e na definição dos fragmentos verticais, podemos gerar uma lista com todos os caminhos para os nós folhas que constam em cada fragmento. Chamaremos estas listas de ListaCaminhos(F_n) onde F_n é cada um dos fragmentos gerados. A Tabela 5 mostra as listas para o fragmento F_1 e F_2 da fragmentação vertical feita na Figura 11.

Tabela 5: Expressões de Caminho para os fragmentos da coleção C_{Itens}

F1	
/Item/Codigo	/Item/ResenhaClientes/Cliente/Nome
/Item/Nome	/Item/ResenhaClientes/Cliente/Cidade
/Item/Descricao	/Item/ResenhaClientes/Cliente/Data
/Item/Secao	/Item/ResenhaClientes/Cliente/Nota
/Item/Preco	/Item/ResenhaClientes/Cliente/Titulo
/Item/Lancamento	/Item/ResenhaClientes/Cliente/Texto
/Item/Ranking	/Item/HistoricoPrecos/HistoricoPreco/Preco
/Item/Característica/Nome	/Item/HistoricoPrecos/HistoricoPreco/DataMudanca
/Item/Característica/Descricao	
F2	
/Item/ListaImagens/Imagem/LocalizacaoOriginal	
/Item/ListaImagens/Imagem/LocalizacaoThumb	
/Item/ListaImagens/Imagem/Descricao	

Tendo essas duas listas, podemos constatar a disjunção verificando se cada caminho existente em ListaCaminhos(C_{Itens}) somente aparece em uma e somente uma das listas ListaCaminhos(F_n). A completude pode ser verificada observando-se que cada caminho da ListaCaminhos(C_{Itens}) aparece em alguma ListaCaminhos(F_n). A reconstrução é garantida pela operação de junção, utilizando o identificador do documento.

3.8 Considerações sobre a validade dos documentos

Nesta seção iremos fazer considerações sobre algumas características especiais existentes nos documentos XML que podem atrapalhar ou até mesmo impedir o uso de fragmentação.

3.8.1 Existência de Namespaces

Namespaces são nomes qualificados utilizados para retirar a ambigüidade de documentos XML que utilizam elementos com mesmo nome, mas que constam em mais de um esquema. Como já especificamos, cada coleção possui apenas uma definição de tipo do documento, associado a um esquema. Portanto, não é possível fragmentar

documentos que possuam mais de uma definição de *namespaces* no corpo do documento. Neste caso, os namespaces podem ser utilizados dentro da definição do tipo, em XML Schema ou DTD.

3.8.2 Existência de atributos

Vamos considerar a existência de uma quantidade considerável de atributos em algum elemento do esquema. Podemos considerar que a separação destes atributos em fragmentos distintos é uma Fragmentação Vertical, já que está alterando a estrutura do esquema. As Fragmentações Horizontais podem ser realizadas sem nenhum problema, já que apenas especificam um conjunto de predicados.

Supondo que o usuário deseje realizar a fragmentação dos atributos, surge o problema da reconstrução, já que toda vez que o elemento em questão for selecionado na saída da consulta, este deverá apresentar todos os seus atributos, e essa reconstrução pode trazer degradação de desempenho.

A fragmentação dos atributos somente faria sentido se muitas consultas remontassem o resultado apenas utilizando um pequeno conjunto desses atributos. Porém, como tratamos principalmente com elementos em nossas definições, surge uma solução mais simples. A aplicação de regras de normalização (ARENAS, LIBKIN, 2004) pode transformar os atributos em elementos, transformando o problema de fragmentar atributos em um caso normal de fragmentação vertical.

Caso esta normalização não seja desejada no escopo do esquema, ou seja, os atributos devem existir para a aplicação do usuário, esta ainda pode ser realizada de forma transparente para o usuário. O SGBD no momento da implementação do fragmento pode criar os elementos para os atributos (de acordo com as regras de normalização), e, no momento de retorná-los ao usuário, voltar à representação de atributos. Consideramos que a normalização do esquema seria o caminho mais simples para esse tipo de caso, já que não mudaria a definição do que é um fragmento vertical. Atualmente, no PartiX, não fazemos nenhuma normalização automática, ficando esta a cargo do projetista.

3.8.3 Existência de elementos do tipo ID/IDREF

Em nosso modelo, não fizemos restrições quanto ao domínio dos elementos do documento XML. Portanto, é possível fragmentar documentos que possuam elementos do tipo ID/IDREF, porém, temos que tomar alguns cuidados. Estamos assumindo que todos os documentos armazenados devam ser documentos válidos (W3C, 2005). Recordando o que é dito na definição de validade do XML, um documento somente é válido se todos os elementos IDREF referenciam elementos ID dentro do próprio documento.

Dada essa regra, é trivial observar que documentos fragmentados horizontalmente não apresentam nenhum problema, já que todos os pares ID/IDREF se manterão íntegros. Agora, quando tratamos de fragmentações verticais, surgem alguns problemas. Se quebrarmos a estrutura de maneira impensada, podemos provocar a invalidação de um ou mais fragmentos.

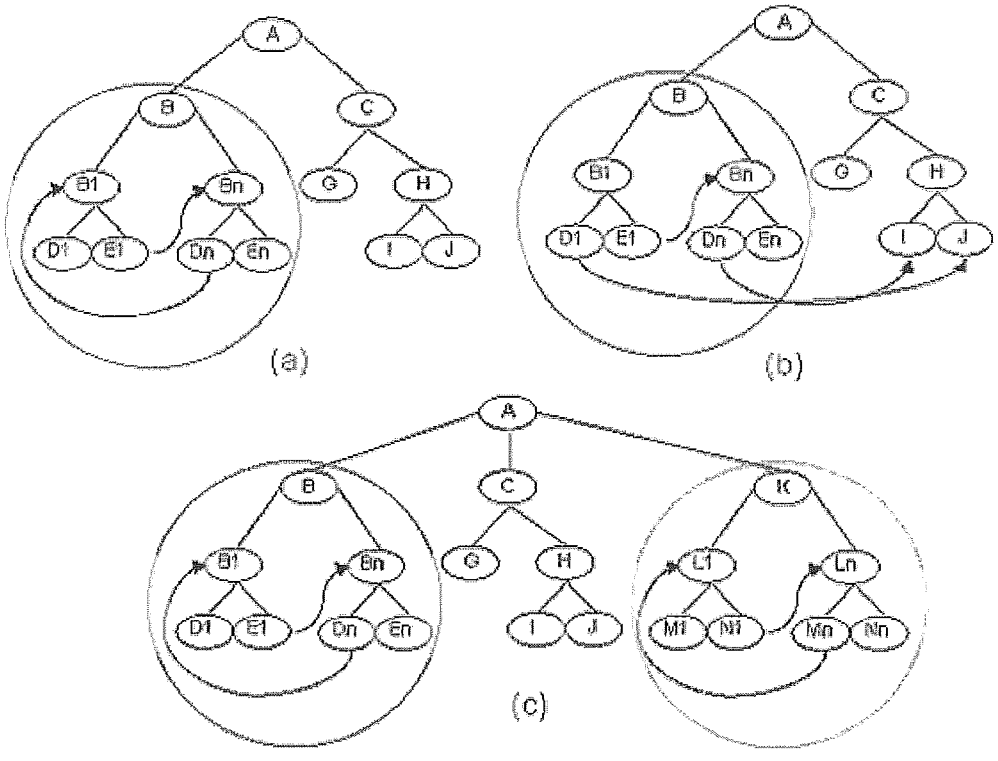


Figura 14: Exemplos de Documentos com ID/IDREFs

Então, temos de garantir que, ao fazer a fragmentação vertical desses documentos, um dos fragmentos apresente todos os pares ID/IDREF relacionados, de

modo que nenhum fique inválido. Ou seja, devemos descobrir para aonde eles apontam. A Figura 14 mostra um documento válido (a) e um documento inválido (b) devido à quebra de sua estrutura em um ponto que separa o grupo de ID/IDREFs. Nos casos em que não for possível isolar os pares ID/IDREF em um determinado fragmento, a fragmentação vertical deve ser proibida. Os círculos indicam um fragmento distinto. O restante da estrutura está em um fragmento a parte. Logo, os exemplos (a) e (b) possuem dois fragmentos e o exemplo (c), três fragmentos. Em todos os exemplos da Figura 14, os nodos B_n e L_n possuem atributos ID. Os nodos D_n , E_n , M_n e N_n possuem atributos do tipo IDREF. Nota-se que os atributos podem ser transformados em elementos. Essa transformação não foi feita para manter o exemplo simples. Vale ressaltar o exemplo (c), no qual as referências ID/IDREF foram agrupadas em seus fragmentos.

Para verificarmos se os ID/IDREFs ficarão isolados, devemos tomar com base no esquema os caminhos para os elementos folha que contenham tipos ID. Essa lista será chamada ListaID(C_{Itens}). Ainda com base no esquema, criaremos uma lista ListaIDREF(C_{Itens}) com todos os caminhos que contém tipos IDREF. Se todos os elementos que constem das listas ListaID(C_{Itens}) e ListaIDREF(C_{Itens}) estiverem em uma determinada lista ListaCaminhos(F_k) para um k qualquer, estes não devão constar em nenhum outro fragmento F_j, onde $j \neq k$, e a fragmentação está correta.

Uma observação importante deve ser feita. A semântica de definição de ID/IDREFs não leva em conta no esquema um padrão que defina com qual IDREF um ID deve casar. Portanto, um IDREF deverá casar com qualquer ID definido no documento, não importando em qual elemento esteja. O valor do elemento ID é um valor numérico, único dentro do documento XML. Isso pode ocasionar alguns problemas, como o ilustrado na Figura 14 (c), onde podemos observar nitidamente que pela semântica adotada para o casamento dos ID/IDREFs, os mesmos se encontram isolados. Porém, ao rodar a verificação proposta por nós, constatar-se-á que a fragmentação vertical não é possível. Não temos como garantir, olhando o esquema, que o elemento /A/K/Ln/Mn (que é um IDREF) não irá referenciar o elemento /A/B/B1 (que é um ID), por exemplo. A conversão para XML Schema, utilizando key/keyrefs resolve o problema de ID/IDREFs.

3.8.4 Existência de elementos do tipo *Key/KeyRef*

Os casos de casamentos de ID/IDREF ocorrem principalmente quando o esquema é especificado em DTD, apesar de ser possível expressá-los também através de XML Schema. Quando desejamos fazer referências a outros elementos, e estamos modelando o esquema através de XML Schema, é natural o uso das ferramentas fornecidas pelo padrão. Nesse caso, o uso da especificação de *Keys* e *KeyRefs*.

Keys e *KeyRefs* são definidas com base em expressões de caminho. Isso torna bem mais simples o seu tratamento. Se o SGBD validar os pares *key/keyref*, este também deverá ser capaz de validar, de forma transparente, quando o esquema estiver fragmentado. O SGBD já conhece o esquema da coleção e como esta está fragmentada.

3.8.5 Existência de Elementos Mistos

Certos esquemas possuem a característica especial de apresentar elementos mistos (*Mixed Elements*) em suas estruturas. Um elemento misto é um elemento que permite ter como nodos filhos texto e outros elementos. Isso traz problemas para a fragmentação vertical do esquema. A fragmentação horizontal não é afetada pois nodos deste tipo não podem ser usados como predicados.

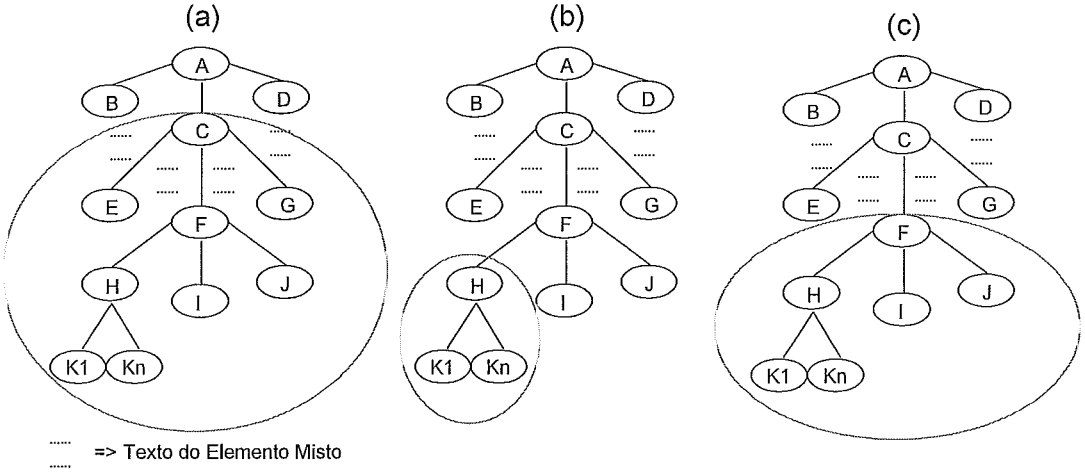


Figura 15: Cenários de Fragmentação Vertical em elementos mistos

A Figura 15 mostra três cenários de fragmentação vertical para o esquema de exemplo apresentado. O elemento C é um elemento misto. Existem sempre dois fragmentos, sendo que um dos quais está circulado. No caso (a) não existe problema na fragmentação, pois a raiz do fragmento é o próprio elemento misto. No caso (b) também não existem problemas, já que é o neto do elemento misto que é a raiz do fragmento. No

caso (c), onde a raiz do fragmento é o filho do elemento misto, surge o seguinte problema. Ao remontar o fragmento, em que posição do texto do elemento misto C, o elemento F deverá ser inserido? É esse o problema que deveremos tratar.

A abordagem mais simples de tratar a questão é não permitir esse tipo de fragmentação. Porém, vamos analisar com mais cuidado a situação. Nosso problema se resume à reconstrução do documento, uma vez que, removido o elemento filho, não saberemos mais onde encaixá-lo, já que este estará circundado por texto.

Para solucionar o problema do posicionamento, temos duas abordagens: I) criar um nodo artificial no fragmento que contém o elemento misto, indicando de onde o filho deste foi removido; II) armazenarmos no nodo filho (e raiz de um dos fragmentos) a informação do posicionamento deste no elemento misto.

Essas abordagens devem ser implementadas no próprio SGBD ou através de um mediador, de modo que fique transparente para o usuário. A opção menos intrusiva na estrutura do documento é a abordagem II. O seu único requisito é uma informação a mais como metadado do fragmento, sendo esta armazenada na raiz do fragmento.

3.8.6 Existência de PI e Comentários

Em geral, os elementos do tipo PI (*Processing Instructions*) e Comentários (*Comments*) não são levados em conta em esquemas de fragmentação. Estes foram omitidos do modelo por simplicidade, já que o objetivo da fragmentação é ganhar desempenho sobre consultas. Alem disso, estes elementos não são usados como predicados de consultas.

Porém, supondo que eles existam no documento, como tratá-los? Em uma fragmentação horizontal não existe nenhum problema, já que eles não irão participar dos predicados. Contudo, em fragmentações verticais, o posicionamento deles deve ser analisado. O principal problema é como inferir o motivo pelo qual um PI/Comentário foi posto. Ele se refere a que elemento? O DOM representa esses elementos especiais como nodos na árvore. Nossa sugestão para esses casos é adotar a heurística de manter o PI/Comentário junto ao elemento pai. Assim, estes elementos poderão acompanhar o esquema de fragmentação desejado pelo usuário.

Capítulo 4 - Arquitetura para Processamento de Consultas XQuery sobre repositórios XML fragmentados

Neste capítulo apresentamos a arquitetura utilizada para apoiar o processamento de consultas sobre nossas definições de fragmentações, e efetuar os testes de desempenho nas bases de dados XML escolhidas. A seção 4.1 traz uma motivação para o uso de tal arquitetura. Já a seção 4.2 faz a definição da arquitetura e dos módulos que a compõem. E por fim, nas seções 4.3 e 4.4 discutimos a re-escrita de consultas e a localização destas, respectivamente.

4.1 Introdução

A principal motivação para pensarmos em uma arquitetura é a falta de um SGBD XML que seja capaz de lidar com coleções XML fragmentadas. Na realidade, os próprios SGBDs XML estão no início de sua maturidade, ainda não tendo todas as características existentes em um SGBD relacional. E, mesmo em bancos de dados relacionais consagrados (ORACLE, 2005, MICROSOFT, 2005), nem todas os tipos de fragmentação são implementados. Esse aparente "atraso" na agregação de novas funções aos SGBDs XML se deve ainda à falta de consenso sobre alguns conceitos que seriam fundamentais para se definir os fragmentos XML. Como vimos nas seções anteriores, são várias propostas diferentes, com diversos pontos de vista para serem analisados. Possivelmente, um SGBD XML irá apenas utilizar uma delas, ou ter módulos específicos para tratar cada uma, de acordo com a necessidade do usuário.

A formalização da definição de fragmentação sobre coleções de documentos XML proposta nesta dissertação foi elaborada para facilitar seu o uso pelos SGBDs atuais. Alguns deles utilizam já o conceito de coleção, como o eXist (EXIST DEVTEAM, 2005), Tamino (SCHÖNING, 2001), Oracle 8i e demais versões superiores (ORACLE, 2005) e Berkeley XMLDB (SLEEPYCAT SOFTWARE, 2005). Nossa intenção é tentar nos aproveitar de toda teoria já existente para o modelo

relacional e trazê-la para o modelo XML, sem com isso podar as características do XML.

Para este fim, a arquitetura proposta contempla diversos módulos componentes interligados: os SGBDs que irão hospedar as bases de dados; um catálogo do esquema de cada coleção original, bem como os metadados que indicam como essa coleção foi fragmentada através das bases de dados.

Com base no esquema de cada coleção e nos metadados, ao receber uma consulta, a arquitetura deve ser capaz de decompô-la em sub-consultas, localizando-as em cada fragmento necessário, e enviando-as para as bases de dados. Esse processamento pode ser feito com um plano de consulta distribuído, onde teremos operações para transferência dos dados recuperados em um fragmento para outro que necessite dessa informação, ou com um plano de consulta centralizado, com todos os fragmentos enviando suas respostas para um único módulo da arquitetura, responsável por montar o resultado final e enviar para o usuário. Sabemos que este tipo de arquitetura não possui nenhuma inovação, mas se faz necessária, já que nenhum SGBD XML é capaz de tratar fragmentos.

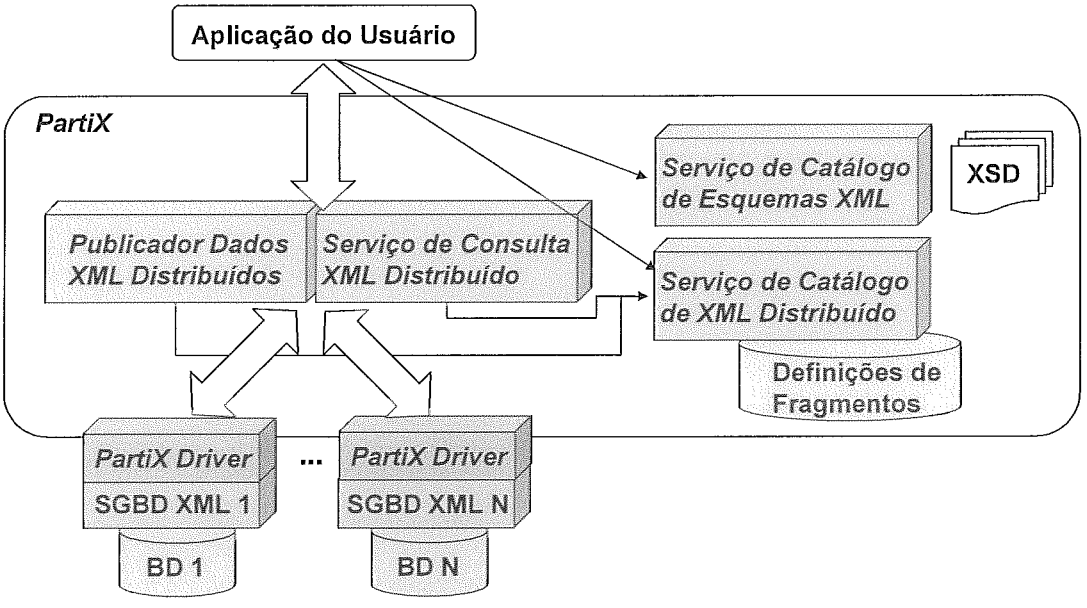


Figura 16: Arquitetura do PartiX

4.2 Definição da Arquitetura

Nossa proposta de arquitetura, implementada pelo protótipo PartiX (nome que se originou em Partições em XML), é ilustrada pela Figura 16. Em linhas gerais, o PartiX intercepta uma consulta XQuery antes que esta chegue ao SGBD XML. O PartiX analisa as definições dos fragmentos existentes no catálogo de metadados e, baseado nas coleções utilizadas pela consulta, reescreve a consulta como sub-consultas para os fragmentos. Então, essas sub-consultas são enviadas para componentes PartiX instalados nos nodos onde se encontram os SGBDs correspondentes. Este componente PartiX local a cada nó se comunica diretamente com o SGBD XML, enviando a sub-consulta e coletando os resultados parciais, que são enviados ao Serviço de Consulta XML Distribuído. A arquitetura é composta por três partes principais: (i) *serviço de catálogo*, que contém catálogos com informações sobre os esquemas e metadados de distribuição, alimentados durante a publicação dos dados; (ii) *serviço de publicação de dados* para a armazenagem de dados XML Distribuídos; e (iii) *serviço de consulta*, responsável por resolver as consultas distribuídas.

O *Serviço de Catálogo de Esquemas XML* é responsável por registrar os tipos usados em cada coleção de documentos XML. Os esquemas são informados através de XML Schemas (XSDs) e cada documento XSD deve definir apenas um tipo documento. Esta restrição é por mera simplicidade, já que um documento XSD pode definir diversos tipos em um mesmo documento. O elemento raiz do tipo é a raiz de cada documento que irá compor a coleção de documentos definida por ele. Caso o SGBD não faça nenhuma verificação sobre o esquema de determinado documento, a arquitetura terá de fazê-lo. Isto é, verificar, no momento da inclusão de um documento em uma coleção, se este é válido para o esquema da coleção. Os fragmentos já são informados com as respectivas informações de alocação, todas fornecidas pelo projetista.

O *Serviço de Catálogo de XML Distribuído* é responsável pelo catálogo dos metadados sobre como uma determinada coleção está fragmentada. Ele também é responsável, uma vez que sejam definidas as regras, por verificar a correção da fragmentação passada ao PartiX.

O Publicador de Dados XML Distribuído recebe documentos XML dos usuários, aplica as regras de correção da fragmentação que foram previamente definidas

para a coleção, e envia os fragmentos resultantes para serem armazenados nos SGBDs remotos. Outra funcionalidade deste módulo é realizar a carga de massas de dados, que também devem ser fragmentadas para após serem armazenadas. Essas regras de correção da fragmentação estão contidas nos metadados de distribuição.

As consultas XQuery são submetidas da aplicação do usuário para o *Serviço de Consultas XML Distribuídas*, o qual analisa as expressões de caminho e predicados da consulta, assim como os metadados de distribuição, identificando assim quais são os fragmentos referenciados em cada consulta. Ele é responsável por escrever as sub-consultas que são enviadas aos SGBDs correspondentes em cada nodo, reconstruir o resultado após recolher todas as respostas e enviá-lo ao usuário.

Nossa arquitetura considera a existência de um *Driver PartiX*, que permite o acesso aos SGBDs remotos para armazenar e recuperar documentos XML. Este *driver* provê uma comunicação uniforme entre os módulos do PartiX e os SGBDs nos nodos aonde residem as coleções distribuídas. O *Driver PartiX* permite que diferentes SGBDs XML participem do sistema. O único requisito é que estes SGBDs XML sejam capazes de processar consultas XQuery.

Como o foco principal da dissertação é a definição de fragmentos e a avaliação do desempenho do processamento de consultas através da fragmentação em bases XML distribuídas, foi implementado um protótipo segundo a arquitetura descrita, com algumas simplificações, tomando-se o devido cuidado para não interferir na generalidade da proposta original. O *Driver PartiX* foi desenvolvido para um SGBD XML específico, o eXist (EXIST DEVTEAM, 2005). O *Serviço de Consultas XML Distribuídas* foi desenvolvido com a capacidade de coordenar a execução de sub-consultas já previamente localizadas.. Em geral, todas as agregações e reconstruções foram delegadas através de planos de consulta, ao determinado SGBD remoto. Os planos de consulta executados também foram previamente gerados. O *Publicador de Dados XML Distribuído* foi implementado com as funções de carga em massa dos dados que utilizam scripts feitos pelo usuário para determinar como as coleções serão armazenadas. As cargas podem ser feitas com base em consultas sobre coleções previamente carregadas, ou através de documentos XML individuais armazenados no disco. Também pode ser feito uso de um documento XSLT (W3C, 2005) para realizar

algumas transformações que podem ser onerosas se feitas com XQuery. Os documentos gerados mais tarde podem ser carregados para as coleções.

4.3 Re-escrita das Consultas

A linguagem de consulta XQuery é muito expressiva e complexa. É bem conhecido na literatura que a re-escrita de consultas XQuery é uma tarefa complexa. Prevemos a localização da consulta e sua conseqüente re-escrita.

Nos testes realizados, consideramos que a re-escrita e localização foram feitas pelo usuário. Estes tópicos são mais detalhados nessa seção e na seção 4.4. Essas consultas já são previamente encaminhadas para os fragmentos corretos, retornando os dados necessários para a remontagem da resposta. Como tomamos como base os conceitos de fragmentação existentes no modelo relacional, surgiu a idéia de utilizarmos alguns conceitos existentes no modelo relacional como base para a re-escrita da consulta. Em (YAO, ÖZSU et al, 2004) encontramos como localizar consultas baseadas em um determinado plano de execução. Para aplicar regras semelhantes, seria necessária uma álgebra XML que contivesse operadores semelhantes de seleção, projeção e junção. Além disso, existem outros trabalhos (POTTINGER, LEVY, 2000, LEVY, RAJARAMAN, 1996) que utilizam uma abordagem genérica pra definir algoritmos para a re-escrita de consultas. Tais algoritmos podem ser re-definidos para comportar um conjunto abrangente de consultas XQuery.

Um complicador das operações de seleção do XQuery é a possível definição de predicados de seleção ao longo da expressão de caminho (predicados aninhados (GREEN, GUPTA et al, 2004)), como nos exemplos da Tabela 6. Em (a) retornamos todas as caixas postais dos proprietários de itens da região asiática que estejam localizados em Tóquio. Em (b), desejamos o nome de todos os clientes residentes em Tóquio, que tenham como sobrenome "Sato".

Tabela 6: Exemplos de predicados aninhados.

(a)	/site/regions/asia/item(location="Tokyo")/mailbox
(b)	/site/people/person(address/city="Tokyo")/name(lastname="Sato")

Consideramos que as consultas utilizadas sempre possam ser escritas utilizando-se expressões de caminho simples, isto é, expressões de caminho que não

possuam nenhum predicado aninhado. Em (PAPARIZOS, WU et al, 2004) os autores citam que sempre é possível transformar uma expressão de caminho em uma expressão de caminho simples, utilizando-se XQuery. As consultas da Tabela 6 foram re-escritas na Tabela 7, somente utilizando-se expressões de caminho simples.

Tabela 7: Re-Escrita das expressões em XQuery

(a)	for $i in document("XMark.xml")/site/regions/asia/item where $i/location = "Tokyo" return $i/mailbox
(b)	for $p in document("XMark.xml")/site/people/person where $p/address/city = "Tokyo" and $p/name/lastname = "Sato" return $p/name

Outro complicador, como citado anteriormente nesta dissertação, é a falta de consenso em uma álgebra para XQuery. Como já vimos, algumas propostas já foram feitas. Mas, dado o nosso conceito de fragmentação embasado na definição de coleções XML, necessitamos de operadores que considerem florestas de documentos, não apenas nodos, como a maioria das álgebras consideram (FRANSICAR, HOUBEN et al, 2002). Em (PAPARIZOS, JAGADISH, 2005, PAPARIZOS, WU, et al, 2004) temos definições de operadores que consideram essas florestas. Também em (PAPARIZOS, JAGADISH, 2005) temos uma álgebra que opera sobre coleções de árvores que mantêm a ordem em que estas se encontram na coleção.

Uma outra questão que precisa ser levantada é a questão da remontagem da consulta. O operador de projeção deve retornar somente os nodos ou sub-árvores que sejam necessários para a resposta da consulta, reduzindo assim o volume de dados trafegado. Ainda assim, questões de otimização como essas devem ser estudadas com mais cuidado, pois dadas as características do XML, talvez esse processamento de redução do resultado seja mais dispendioso que enviar o documento todo. Em (FLORESCU, HILLERY et al, 2004) é feito um estudo do processador de consultas XQuery utilizado na ferramenta de integração *BEA WebLogic Integrate* (BEA SYSTEMS, 2005). Neste artigo temos exemplos de otimizações e normalizações feitas pelo processador de consulta.

4.4 Localização de Consultas

Apesar de não ser o foco principal da dissertação, vamos apresentar aqui um algoritmo simples para a localização de consultas em fragmentos horizontais do PartiX. Como já dissemos acima, a tarefa de reconstrução de consultas é uma tarefa complexa. No atual protótipo PartiX fizemos algumas considerações que simplificam a tarefa.

Por definição, o esquema dos fragmentos horizontais é o mesmo da coleção original. Devido a esse fato, é garantido que todos os elementos referenciados em uma consulta também estarão nos fragmentos. É importante notar que elementos com cardinalidade 0..1 fazem parte do esquema. A sua existência ou não é uma mera característica do documento em questão. Tendo isso em mente, nosso único trabalho é localizar os fragmentos em que os elementos consultados apareçam, que são os fragmentos necessários para a consulta. O ponto principal do nosso algoritmo é descobrir quais são os fragmentos envolvidos em uma dada consulta, e alterar o parâmetro de entrada da função *collection* apropriadamente.

Podemos observar que uma especificação trivial para os fragmentos horizontais usa conectores lógicos OR ou negações. Entretanto, consideramos que fragmentos horizontais podem somente ser definidos através do uso do conector lógico AND (ou seja, como conjunções), então devemos aplicar as regras de De Morgan para obter essas conjunções.

Seja C a coleção decomposta em um conjunto de fragmentos horizontais $\Phi := \{F_1, ..., F_n\}$, $n \geq 1$. Por definição, $F_i := \langle C, \sigma_{\mu i} \rangle$, tem documentos que satisfazem $\sigma_{\mu i}$. O critério de seleção μi é uma conjunção de predicados simples $PredF_i = \{p_1, ..., p_x\}$. Quando μi é negada, negamos cada predicado no conjunto.

Seja Q uma consulta expressa em XQuery com um conjunto de predicados $Preds = \{p_1, ..., p_k\}$, $k \geq 0$, e um conjunto de expressões de caminho $Paths = \{P_1, ..., P_m\}$, $m \geq 1$. As expressões em *Paths* são as expressões encontradas nos predicados e na cláusula de retorno de Q.

Para identificar os fragmentos para os quais Q deve ser enviada, analisaremos os seguintes itens. Esta análise constrói um conjunto de *Frags* com todos os fragmentos requeridos para responder Q.

1) Para cada p_j em *Preds*, se $p_j \in PredF_i$, então adicione F_i em *Frags*.

2) Para cada $P_i \in Paths$, se $p_j \in PredF_i$ é pré-fixado por um caminho simples P_i usado em algum μi, e p_j não tem nenhuma comparação de valor e nenhuma aplicação de função, então Q deve ser enviado para F_i. Adicione F_i em *Frags*.

O passo 1 analisa os predicados usados na consulta que também são utilizados na definição dos fragmentos. O passo 2 trata dos casos em que as consultas contêm expressões de caminho que possuem um prefixo que participa em alguma definição de fragmentos baseada em um teste existencial. Mais de um fragmento pode ser escolhido neste caso. No caso de funções, elas devem casar exatamente com a definição do fragmento, sendo tratadas no passo 1).

É importante observar que os pontos anteriores não são exclusivos. Ambos podem ser usados em uma única consulta. Se nenhum deles se aplicar, adicione todos os fragmentos em *Frags*. Uma vez que o conjunto *Frags* esteja preenchido, precisamos mapear a consulta Q em sub-consultas de acordo com os sítios onde cada fragmento *Frags* está alocado. Então, apenas geramos uma sub-consulta para cada fragmento $F_i \in Frags$, alterando o parâmetro de entrada da função *collection* para a localização de F_i.

<table>
<tr><td colspan="3">Q = for $i in collection("Items")/Item
where $i/Secao = "DVD" and $i/Lancamento = "T"
return $i</td></tr>
<tr><td>Φ =</td><td>F_1:=⟨C_{itens}, $\sigma_{/Item/ecao="DVD"}$⟩
F_2:=⟨C_{itens}, $\sigma_{/Item/Secao="CD"}$⟩
F_3:=⟨C_{itens}, $\sigma_{/Item/Secao\neq"DVD" \text{ and } /Item/Secao\neq"CD"}$⟩</td><td>$Pred_{F1}$ = {/Item/Secao = "DVD"}
$Pred_{F2}$ = {/Item/Secao = "CD"}
$Pred_{F3}$ = {/Item/Secao ≠ "DVD",
/Item/Secao ≠ "CD"}</td></tr>
<tr><td colspan="3">Preds = {/Item/Secao = "DVD", /Item/Lancamento = "T"}</td></tr>
<tr><td colspan="3">Paths = {/Item/Secao, /Item/Lancamento, /Item}</td></tr>
<tr><td>1)</td><td colspan="2">Frags = {F_1}</td></tr>
<tr><td>2)</td><td colspan="2">Frags = {F_1}, sem modificações, já que todos os p_j possuem comparação de valor.</td></tr>
</table>

Figura 17: Exemplo de localização com predicados simples

<table>
<tr><td colspan="3">Q = for $i in collection("Items")/Item
where $i/Secao = "DVD" and $i/Lancamento = "T"
return $i</td></tr>
<tr><td>Φ =</td><td>F_1:=⟨C_{itens}, $\sigma_{/Item/Secao}$⟩
F_2:=⟨C_{itens}, $\sigma_{empty(/Item/Secao)}$⟩</td><td>$Pred_{F1}$ = {/Item/Secao}
$Pred_{F2}$ = {empty(/Item/Seção)}</td></tr>
<tr><td colspan="3">Preds = {Secao = "DVD", Lancamento = "T"}</td></tr>
<tr><td colspan="3">Paths = {/Item/Secao, /Item/Lancamento, /Item}</td></tr>
<tr><td>1)</td><td colspan="2">Frags = {}, nenhum predicado da consulta casa com um predicado dos fragmentos</td></tr>
<tr><td>2)</td><td colspan="2">Frags = {F_1}, /Item/Secao é caminho utilizado no predicado de F_1</td></tr>
</table>

Figura 18: Exemplo de localização com predicado existencial

Na Figura 17 e Figura 18 temos exemplos do algoritmo para o caso em que existem apenas predicados simples baseados em comparações de valor, e com um predicado baseado em um teste existencial.

Capítulo 5 - Avaliação Experimental

Neste capítulo analisamos os resultados obtidos com os testes utilizando o protótipo PartiX, usando a arquitetura proposta na seção 4.2. Discutimos como os testes foram conduzidos, quais bases foram utilizadas nos testes e como as bases de dados foram geradas e carregadas no banco de dados. Além disso, demonstramos casos em que a fragmentação dessas coleções traz ótimos resultados em termos de desempenho, bem como, os casos em que não é recomendada uma fragmentação.

O capítulo está dividido da seguinte forma. Na seção 5.1 mencionamos qual máquina foi utilizada e detalhes acerca do banco de dados e geração de bases. Na seção 5.2 descrevemos quais bases foram utilizadas para os testes. O processo de carga dessas bases foi um passo importante em nossos testes, então a seção 5.3 é dedicada a esse assunto. E por fim, a seção 5.4 é dedicada à avaliação dos resultados obtidos nas bases de dados utilizando-se fragmentação horizontal, fragmentação vertical e fragmentação híbrida.

5.1 Ambiente Experimental

Para avaliar os benefícios da fragmentação no desempenho do processamento de consultas em bases XML, conduzimos uma série de testes utilizando o protótipo do PartiX com o SGBD XML nativo eXist, que permite expressar consultas em XQuery. O computador utilizado nos testes foi um Athlon XP 2.4Ghz, com 512Mb de memória RAM DDR333.

A principal razão para nossa escolha pelo eXist é a possibilidade de utilização de XQuery. Outros bancos de dados XML livres existentes não suportam essa linguagem de consulta, como por exemplo, o XIndice (THE APACHE SOFTWARE FOUNDATION, 2005). No momento da escolha do SGBD, o Berkeley DB XML (SLEEPYCAT SOFTWARE, 2005) ainda não suportava XQuery, mas hoje ele já apresenta esta característica.

Um ponto importante que deve ser considerado é a resolução de planos de consulta pelo eXist. O algoritmo utilizado por ele é proprietário e desconhecemos se ele

utiliza uma álgebra e como são suas heurísticas. Pode ser que em outros SGBDs o comportamento de algumas consultas seja diferente. Porém temos ciência de que esses testes são apenas indicativos de que a fragmentação funciona e é vantajosa, mas que ainda são necessários testes adicionais com outros sistemas.

Utilizamos o gerador de bases ToXgene (BARBOSA, MENDELZON et al, 2002) para gerar todos os nossos exemplos. Não tivemos dificuldade em seu uso e este se mostrou eficiente na geração de todas as bases, apresentando apenas algumas limitações de tamanho para geração de um número muito grande de documentos. Porém, em uma versão mais atualizada disponível recentemente, os desenvolvedores citam que essas limitações já foram contornadas.

Nossos testes foram feitos utilizando-se uma série de bases distintas, onde estas bases possuem certas características próprias. Tomamos o tempo desde o momento em que a consulta é submetida ao servidor até o momento em que o servidor retorna com a resposta para o usuário.

5.2 Bases de Dados

Optamos pelo uso de bases com esquemas e características de construção variadas para a nossa avaliação. Essa variedade nos ajuda a explorar diversos aspectos e tipos de fragmentação, permitindo-nos ter uma visão mais ampla dos casos em que as fragmentações serão ou não eficientes.

Além das bases que já apresentamos e explicamos no Capítulo 3, utilizamos dois esquemas do benchmark XBench e o esquema do benchmark XMark. Do XBench utilizamos o esquema de Artigos e o esquema de Catálogo de Itens.

A Tabela 8 descreve as bases utilizadas e os tipos de testes realizados sobre cada uma delas. A primeira coluna apresenta a identificação da base que será utilizada deste ponto em diante do texto. Assim, toda vez que nos referirmos à base de dados do XMark, estaremos nos referindo à Base xmark, por exemplo.

Tabela 8: Bases de Dados utilizadas nos testes

	Base	Tipo	Fragmentação	Tamanho
ItemHom	Itens	MD	Horizontal (Dados Homogêneos)	Média 2Kb p/ doc
ItemHet	Itens	MD	Horizontal (Dados Heterogêneos)	Média 2Kb p/ doc
Item80	Itens	MD	Horizontal	Média 80Kb p/ doc
xmark	XMark	SD	Vertical	Varia 5Mb-15Mb
ArtHor	Artigos	MD	Horizontal	Média 12Kb p/ doc
ArtVert	Artigos	MD	Vertical	Média 12Kb p/ doc
LojaHib	Loja	SD	Híbrida	Varia 5Mb-500Mb

A coluna *Tipo* da tabela se refere ao tipo de documento que compõe a coleção, onde MD é utilizado para Múltiplos Documentos (*Multiple Documents*) e SD para Único Documento (*Single Document*). A coluna *Fragmentação* indica qual tipo de fragmentação foi utilizado na base de dados em questão. A coluna *Tamanho* indica o tamanho médio dos documentos utilizados nas bases MD. Nas bases SD, o tamanho da base já é o tamanho do próprio documento.

5.2.1 Bases de Itens

As bases ItemHom, ItemHet e Item80, têm seu esquema definido na Figura 1. Nas bases ItemHom, ItemHet e Item80, o principal foco será uma fragmentação horizontal através do elemento *Secao*, que indica em qual seção da loja o item pode ser encontrado. Tendo isso em mente, a base ItemHom foi gerada considerando uma amostragem homogênea na distribuição dos itens por essas seções, enquanto que as bases ItemHet e Item80, consideram uma amostragem heterogênea. Na amostragem heterogênea, consideramos que 25% da base é composta de itens de Livraria e que as seções CD, DVD e Eletrônicos possuem, cada uma, 15% da base.

As bases ItemHom e ItemHet foram geradas considerando-se que os elementos *ResenhaCliente*, *HistoricoPreco* e *ListaImagens* possuem cardinalidade 0. Já a base Item80 possui esses mesmos elementos com cardinalidade 1, o que justifica a diferença nos tamanhos de documento. O objetivo dessa diferença é avaliar qual o tamanho do documento que melhor é tratado pelo SGBD XML no processamento da consulta.

Na Tabela 9 temos as consultas utilizadas na avaliação do desempenho das bases ItemHom, ItemHet e Item80.

Tabela 9: Consultas utilizadas na Base de Itens

	Consulta XQuery
Q_1	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where $x/Item/Lancamento = "T" return $x/Item
Q_2	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where $x/Item/Secao = "Livraria" return $x/Item/Nome
Q_3	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where $x/Item/Secao = "Games" return $x/Item/Nome
Q_4	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where $x/Item/Secao = "Livraria" and $x/Item/Lancamento = "T" return $x/Item/Nome
Q_5	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where contains(string($x/Item/Descricao),"silent") return $x/Item
Q_6	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where contains(string($x/Item/Descricao),"silent") and $x/Item/Secao = "CD" return $x/Item
Q_7	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where count($x/Item/Caracteristica) >= 4 return $x/Item
Q_8	for $x in collection("/db/Exemplos/Horizontal/Hete5Mb/Sample1") where $x/Item/Secao = "DVD" and count($x/Item/Caracteristica) >= 4 return $x/Item

Conforme podemos notar na Tabela 9, a base de itens possui 4 consultas (Q_1 a Q_4) apenas com predicados simples. Já as consultas Q_5 e Q_6 já apresentam uma busca em texto, enquanto que as duas últimas consultas, Q_7 e Q_8 realizam uma consulta com uma cláusula de agregação (um *count*).

As bases ItemHom, ItemHet e Item80 não foram utilizadas em testes de fragmentação vertical, pois possuem um conjunto grande de documentos pequenos. Entendemos que a fragmentação vertical não iria trazer nenhum benefício, já que temos as consultas retornando o documento por completo.

5.2.2 Base XMark

A base de dados Xmark foi gerada utilizando-se a aplicação do benchmark XMark. Essa base de dados de um único documento representa um site de leilões que abrange itens em vários continentes. São registrados quais os leilões estão em aberto, os itens sendo vendidos, bem como os leilões que já foram encerrados. A listagem de clientes inclui informações como cartão de crédito, renda e quais itens foram postos à venda pelo cliente. A estrutura do banco de dados é complexa, como podemos ver na Figura 19.

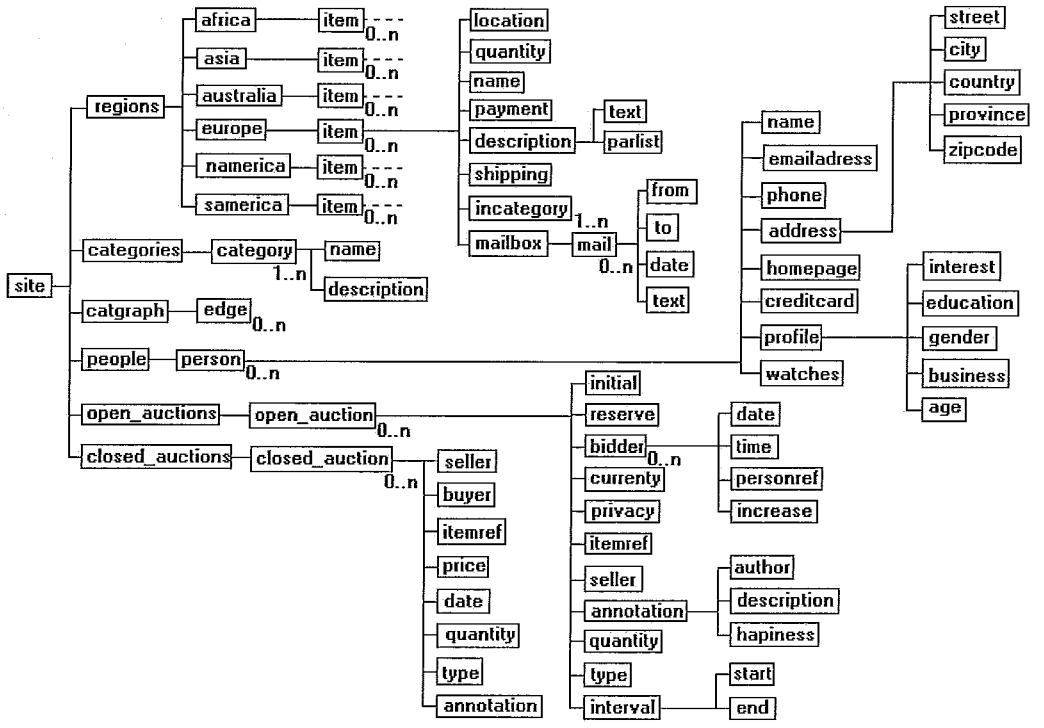


Figura 19: Esquema da Base de Dados do XMark

A Tabela 10 mostra quais são as consultas utilizadas nesta base. A consulta Q_1 retorna o lance inicial de todos os leilões em aberto. Q_2 retorna os IDs de todos os leilões abertos nos quais o último lance é duas vezes maior que o lance inicial. As consultas Q_3 a Q_8 utilizam a função de agregação *count*. Q_3 diz quantos itens que foram vendidos com preço superior a 40, Q_4 retorna quantos itens estão listados em todos os continentes e Q_5 retorna quantos trechos de texto existem no banco de dados. Cada uma utiliza a função *count* de uma forma distinta. Já as demais utilizam *counts* em conjuntos mas em expressões um pouco mais complexas, com variáveis declaradas com a cláusula *let*. A consulta Q_6 lista os nomes das pessoas e o número de itens que cada uma compra. Q_7 retorna para cada pessoa, a quantidade de itens que estão atualmente à venda, cujo preço não excede em 0.02% o que a pessoa ganha. Já a Q_8 efetua a mesma consulta, só que filtrando apenas as pessoas que ganham 50000. Em Q_9 e Q_{10}, listamos os nomes dos itens registrados na Austrália e Europa, respectivamente, junto com suas descrições. Existem essas duas consultas, pois a cardinalidade de itens difere bastante nos dois continentes e queremos medir o desempenho nos dois casos. Já Q_{11} faz uma busca em texto, retornando o nome de todos os itens cuja descrição contenha a palavra "gold". Em Q_{12} utilizamos uma função existencial, onde retornamos o nome de todos os

usuários que não possuem uma *homepage*. Em Q_{13}, temos uma ordenação, aonde retornamos os itens em ordem alfabética junto com a sua localização. Por fim, Q_{14} agrupa os clientes pelo valor que ganham e retorna a cardinalidade de cada grupo.

Tabela 10: Consultas utilizadas na Base do XMark

	Consultas XQuery
Q_1	for $b in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/open_auctions/open_auction return <increase> {$b/bidder(1)/increase/text()} </increase>
Q_2	for $b in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/open_auctions/open_auction where $b/bidder(1)/increase/text()*2 <= $b/bidder(last())/increase/text() return <increase first="{$b/bidder(1)/increase/text()}" last="{$b/bidder(last())/increase/text()}"/>
Q_3	count(for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/closed_auctions/closed_auction where $i/price/text() >= 40 return $i/price)
Q_4	for $b in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/regions return count($b//item)
Q_5	for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site return (count($p//description) + count($p//annotation) + count($p//email))
Q_6	for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/people/person let $a := (for $t in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/closed_auctions/closed_auction where $t/buyer/@person = $p/@id return $t) return <item person="{$p/name/text()}"> {count($a)} </item>
Q_7	for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/people/person let $l := (for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/open_auctions/open_auction/initial where $p/profile/@income > (0.02 * $i/text()) return $i) return <person name="{$p/name/text()}"> {count ($l)} </person>
Q_8	for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/people/person let $l := (for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/open_auctions/open_auction/initial where $p/profile/@income > (0.02 * $i/text()) return $i) where $p/profile/@income > 50000 return <person person="{$p/name/income/text()}"> {count($l)} </person>
Q_9	for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/regions/australia/item return <item name="{$i/name/text()}"> {$i/description} </item>
Q_{10}	for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/regions/europe/item return <item name="{$i/name/text()}"> {$i/description} </item>
Q_{11}	for $i in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site//item where contains ($i/description,"gold") return $i/name/text()
Q_{12}	for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/people/person where empty($p/homepage/text()) return <person name="{$p/name/text()}"/>
Q_{13}	for $b in

	collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/regions//item let $k := $b/name/text() order by $b/name/text() return <item name="{$k}"> {$b/location/text()} </item>
Q_{14}	<result><preferred> {count (collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/people/person/profile(@income >= 100000))}</preferred>, <standard> {count (collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/people/person/profile(@income < 100000 and @income >= 30000))}</standard>, <challenge> {count (collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1") /site/people/person/profile(@income < 30000))}</challenge>, <na> {count (for $p in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample1")/site/people/person where empty($p/@income) return $p)} </na></result>

A base Xmark é um exemplo clássico para o uso de fragmentação vertical. Essa base constitui um único documento que armazena todas as informações necessárias. Muitas consultas são localizadas, podendo então se beneficiar desse tipo de fragmentação. As Fragmentações híbridas são inviáveis, já que os conjuntos de dados estão fortemente amarrados por IDREFs.

5.2.3 Bases de Artigos - XBench

As bases de dados ArtHor e ArtVert foram gerada utilizando-se os modelos fornecidos pelo benchmark XBench. O esquema dessa base pode ser visto na Figura 20. O banco de dados consiste em uma série de documentos que contêm informações de identificação dos autores, palavras-chaves e gênero do artigo, bem como seu texto completo, resumo e um epílogo, contento suas referências. A tabela Tabela 11 mostra quais são as consultas utilizadas nesta base. Esses documentos são essencialmente compostos por texto, possuindo aninhamento de elementos (as sub-seções do artigo, dentro do elemento *body*).

Muitas das consultas realizadas nessa base são diretas, recuperando um artigo específico. A consulta Q_1 retorna o título do artigo que possui o ID 200. Q_2 retorna o título do artigo quem tem "Ben Yang" como autor. Já Q_3 retorna o cabeçalho da seção seguinte à seção intitulada "Introduction" em um determinado artigo cujo ID é 350. Q_4 já realiza uma busca em texto, retornando o titulo dos artigos que têm as duas palavras-chaves "the" e "hockey" em um parágrafo do resumo.

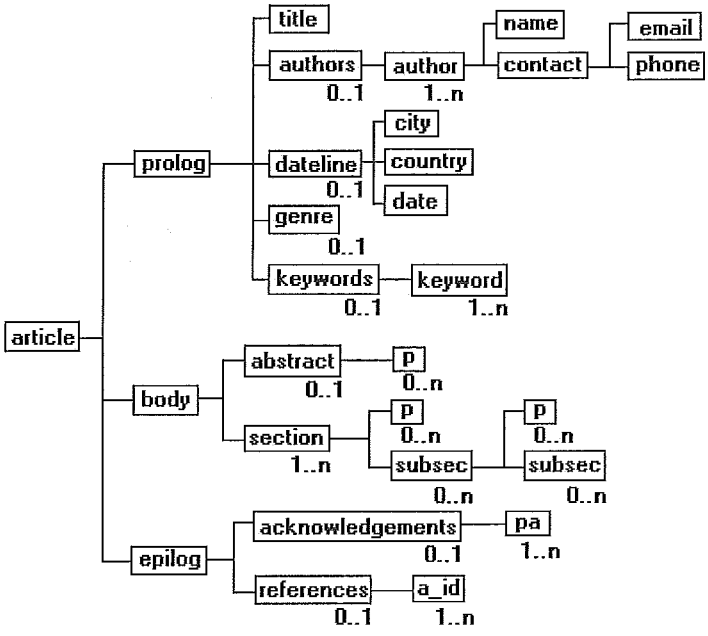


Figura 20: Esquema da base de dados de Artigos

Em Q_5 utilizamos o caractere coringa "*" para selecionar o nome de todos os autores de um determinado artigo. Na consulta Q_6, temos uma ordenação. Ela lista o título dos artigos que publicados no Canadá, ordenados por data. A consulta Q_7 é outra consulta direta (que tem por objetivo retornar um registro específico), que retorna todo o artigo dado um determinado valor de ID.Na consulta Q_8 temos a construção de um resumo de informações de um determinado artigo, incluindo o título, nome, o primeiro autor, a data e o resumo. Em Q_9, temos outra busca em texto, onde retornamos o título dos artigos que contêm a palavra "hockey" em alguma parte do texto. Nessa consulta, utilizamos o operador "//". Por fim, a consulta Q_{10} lista o nome dos autores os quais não possuem informação de contato em algum artigo.

Tabela 11: Consultas utilizadas na Base de Artigo

	Consultas XQuery
Q_1	for $art in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article(@id="200") return $art/prolog/title
Q_2	for $prolog in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article/prolog where $prolog/authors/author/name="Ben Yang" return $prolog/title
Q_3	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1") /article(@id="350")/body/section(@heading="introduction"), $p in

	collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article(@id="350") /body/section(. >> $a)(1) return <HeadingOfSection>{$p/@heading}</HeadingOfSection>
Q_4	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article where some $b in $a/body/abstract/p satisfies (contains($b, "the") and contains($b, "hockey")) return $a/prolog/title
Q_5	for $art in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article(@id="100") return $art/prolog/*/author/name
Q_6	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article/prolog where $a/dateline/country="Canada" order by $a/dateline/date return <Output>{$a/title}{$a/dateline/date}</Output>
Q_7	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article(@id="6") return $a
Q_8	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article(@id="120") return <Output> {$a/prolog/title} {$a/prolog/authors/author(1)/name} {$a/prolog/dateline/date} {$a/body/abstract} </Output>
Q_9	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article where contains ($a//p, "hockey") return $a/prolog/title
Q_{10}	for $a in collection("/db/Exemplos/Horizontal/TCMD5Mb/Sample1")/article/prolog/authors/author where empty($a/contact/text()) return <NoContact>{$a/name}</NoContact>

5.2.4 Base da Loja

O esquema utilizado pela base LojaHib tem seu esquema definido na Figura 2. A base LojaHib utiliza a mesma amostragem explicada na seção 5.2.1 para a geração dos elementos *Item*, que constituem os itens vendidos na loja.

A Tabela 12 enumera as consultas utilizadas na base LojaHib. As consultas utilizadas na base de loja apresentam as mesmas características quanto às primeiras 8 consultas utilizadas nas demais bases de item (ver seção 5.2.1). A consulta Q_9 realiza uma consulta com um predicado simples, enquanto que as consultas Q_{10} e Q_{11} exploram agregações como resultados da consulta (um *sum* e um conjunto de *counts*, respectivamente).

Tabela 12: Consultas utilizadas na Base de Loja

	Consulta Xquery
Q_1	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where $x/Lancamento = "T" return $x
Q_2	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where $x/Secao = "CD" return $x/Nome
Q_3	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where $x/Secao = "Papelaria" return $x/Nome
Q_4	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where $x/Secao = "CD" and $x/Lancamento = "T" return $x/Nome
Q_5	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where contains(string($x/Descricao),"execute") return $x
Q_6	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where contains(string($x/Descricao),"execute") and $x/Secao = "CD" return $x
Q_7	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where count($x/Caracteristica) >= 4 return $x
Q_8	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1")/Loja/Itens/Item where $x/Secao = "CD" and count($x/Caracteristica) >= 4 return $x
Q_9	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Funcionarios/Funcionario return $x
Q_{10}	for $x in collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Funcionarios return <TotalPagamento>{sum($x/Funcionario/Salario)}</TotalPagamento>
Q_{11}	Let $f := collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Funcionarios, $s1:=collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Itens/Item(Secao="CD"), $s2:=collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Itens/Item(Secao="DVD"), $s3:=collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Itens/Item(Secao="Papelaria"), $s4:=collection("/db/Exemplos/FragHibrida/LojaItem5Mb/Sample1") /Loja/Itens/Item(Secao="Livraria") return <Totais> <TotalFuncionario>{count($f/Funcionario)}</TotalFuncionario> <TotalCD>{count($s1)}</TotalCD> <TotalDVD>{count($s2)}</TotalDVD> <TotalPapelaria>{count($s3)}</TotalPapelaria> <TotalLivraria>{count($s4)}</TotalLivraria> </Totais>

A base LojaHib foi idealizada para explorar um caso de fragmentação híbrida. Não faz sentido para nós fragmentá-la horizontalmente, já que é constituída de apenas um único documento. Apesar de poder ser utilizada uma fragmentação exclusivamente vertical sobre essa base (separando os itens das seções e pessoal), optamos por não fazê-lo, já que a própria fragmentação híbrida conduzirá a essa fragmentação vertical como parte dela.

5.3 Carga das Bases de Dados

O processo de carga foi realizado utilizando-se um módulo do próprio protótipo fartes, que é responsável pela carga de dados XML em massa. Esse módulo recebe um arquivo XML que descreve a origem dos dados e quais são as coleções onde serão armazenadas as informações.

É importante notar que este módulo foi escrito para carga de dados no banco de dados eXist, já que ele possui algumas funções capazes de fazer a carga direta de informações de uma coleção em outra. Nosso módulo é capaz de criar coleções a partir de arquivos XML armazenados em um sistema de arquivos, necessário para a carga inicial das coleções; ou a criação de coleções com base em uma outra coleção principal, uma vez que esta já exista dentro do eXist.

Vamos supor que o banco de dados XML utilizado não seja o eXist. E que, neste outro banco, não seja possível criar uma coleção a partir de outra coleção principal. Ainda assim, poderemos exportar os dados desta coleção principal para arquivos XML, previamente fragmentados pelo critério escolhido, e utilizar esta ferramenta para carga. Por fim, se o banco de dados possuir alguma ferramenta de carga própria, esta ainda poderá ser usada.

5.3.1 Estrutura do Arquivo de Carga

A estrutura do arquivo foi pensada para ser simples. Na Figura 21 temos um exemplo de um arquivo de carga. O carregador de dados sempre trabalha em apenas um servidor. Caso cargas entre servidores tenham de ser feitas o conteúdo do servidor origem deverá ser exportado para dados XML para depois ser carregado usando nosso módulo.

```
<Loader>
  <Server>http://127.0.0.1:8080/exist/xmlrpc</Server>
  <Collection name="C1" operation="create">
    <Path>/db/Exemplos/Horizontal/Complexo100Mb/Sample1/</Path>
    <DataLocation>K:\DatabaseCargas\FH_hete_100Mb</DataLocation>
  </Collection>
  <Collection name="C2" operation="create">
    <Path>/db/Exemplos/Horizontal/Complexo100Mb/Sample2/F1/</Path>
    <Query>declare namespace  xdb="http://exist-db.org/xquery/xmldb";
        let $out := xdb:collection("xmldb:exist:////db/Exemplos/Horizontal/Complexo100Mb/Sample2/F1/", "admin", "")
        for $rec in collection("/db/Exemplos/Horizontal/Complexo100Mb/Sample1/")
          where $rec/Item/Secao = "Brinquedos" or
              $rec/Item/Secao = "Games"     or
              $rec/Item/Secao = "Perfumaria" or
              $rec/Item/Secao = "Eletronicos"
          return xdb:store($out, concat("Item",concat( $rec/Item/Codigo/text(), ".xml")), $rec/Item )</Query>
  </Collection>
</Loader>
```

Figura 21: Exemplo de arquivo de carga

Todo arquivo de carga tem a raiz Loader. O elemento Server indica qual é o servidor onde estamos trabalhando. Toda a lógica de carga está nos elementos Collection que compõem o resto do arquivo. É ele que irá descrever a coleção sendo carregada. O atributo name desse elemento dá um nome genérico para a coleção dentro do script. Esse nome é para objetivos de *debug* na hora da carga, e caso, no futuro, uma coleção faça referência a outras. O atributo operation dita qual a operação que será feita na coleção. Se possuir o valor *"create"*, a coleção será criada; caso possua o valor *"delete"*, a coleção será excluída do banco; e, caso tenha o valor *"append"*, a coleção será acrescida dos documentos sendo carregados.

Os elementos filhos do elemento Collection dependem da operação sendo realizada. Em todos os casos, o elemento Path deve ser preenchido. É ele que informa o caminho que identifica a coleção dentro do banco de dados. A semântica utilizada é do eXist e deve ser um caminho absoluto. Caso seja utilizada a operação de criação ou inclusão de dados em uma coleção que existia previamente, somente um dos seguintes elementos pode aparecer: DataLocation ou Query. Caso estivermos excluindo uma coleção, somente o elemento Path deve aparecer.

O elemento DataLocation informa o caminho no sistema de arquivo onde os documentos XML a serem carregados para a coleção residem. Este caminho é absoluto e dependente do sistema operacional sendo utilizado. O elemento Query informa uma consulta XQuery que é feita na base de dados. A resposta dessa consulta deve ser carregada na coleção informada em Path. No caso do eXist, simplesmente

executamos a consulta contra o banco de dados, já que este SGBD XML possui mecanismos para criarmos coleções dentro do próprio XQuery.

O eXist possui um *namespace xmldb*, que contém funções de manipulação de coleções e documentos. Através do uso da função *xdb:collection*, podemos recuperar uma referência para uma coleção. A coleção deve ser passada como uma URI, além do usuário e senha, para que ela possa ser acessada. E com o uso de uma outra função, *xdb:store*, declarada sob o mesmo *namespace*, podemos armazenar um documento em qualquer coleção, desde que tenhamos a referência para ela. A sintaxe do comando recebe a referência para a coleção, um nome para o arquivo (que deve ser único pelas regras do eXist), e o documento XML a ser armazenado.

5.3.2 Carga das Bases de Teste

O processo de carga em todas as bases foi semelhante. No mesmo script de carga, criamos uma coleção com todos os dados não fragmentados. A partir dessa coleção, fazemos consultas e geramos as bases fragmentadas, cada uma com seu próprio critério. O critério que foi usado será discutido nas seções sobre o desempenho das fragmentações.

A base de teste LojaHib apresentou problemas na carga dosarquivos com mais de 100 Mb. Ocorreram erros utilizando a interface RPC com o eXist para fazer a carga. Aparentemente, ele não suportou o tamanho do arquivo. Nesses casos, tivemos que usar a interface gráfica nativa do eXist para realizarmos a carga destes documentos.

Outro problema interessante aconteceu com a base de teste D, gerada com o XMark. É sabida a complexidade do documento utilizado por este benchmark. Ao importarmos um arquivo com mais de 30Mb, sem nenhuma fragmentação, ocorreu um erro no eXist, informando que o documento era muito complexo para ser armazenado na base de dados. Com isso, nossos testes com o XMark ficaram muito limitados.

5.4 Avaliação dos Resultados

Os testes foram realizados com tomadas de tempos nas bases já citadas, sendo que a primeira execução da consulta é descartada (o tempo frio) e consideramos a média das demais tentativas (o tempo quente). Fizemos execuções de 10 consultas em série. O tempo registrado é apenas o tempo de resposta de cada consulta. O tempo de

transmissão do resultado é avaliado através de amostragem do tempo de transmissão de um bloco de dados do retorno das respectivas consultas em alguns casos, e em outros, é calculado através de informações como a cardinalidade de consulta e distribuição dos dados.

Nossos testes simulam que os fragmentos estão alocados em nodos separados em uma rede interna. A velocidade dessa rede, para efeitos de cálculo, é uma Gigabit Ethernet, ou seja, velocidade de 1 Gbit/s. Uma rede dessas é perfeitamente factível hoje em dia. Na Figura 22 temos um esquema do cenário de transmissão considerado.

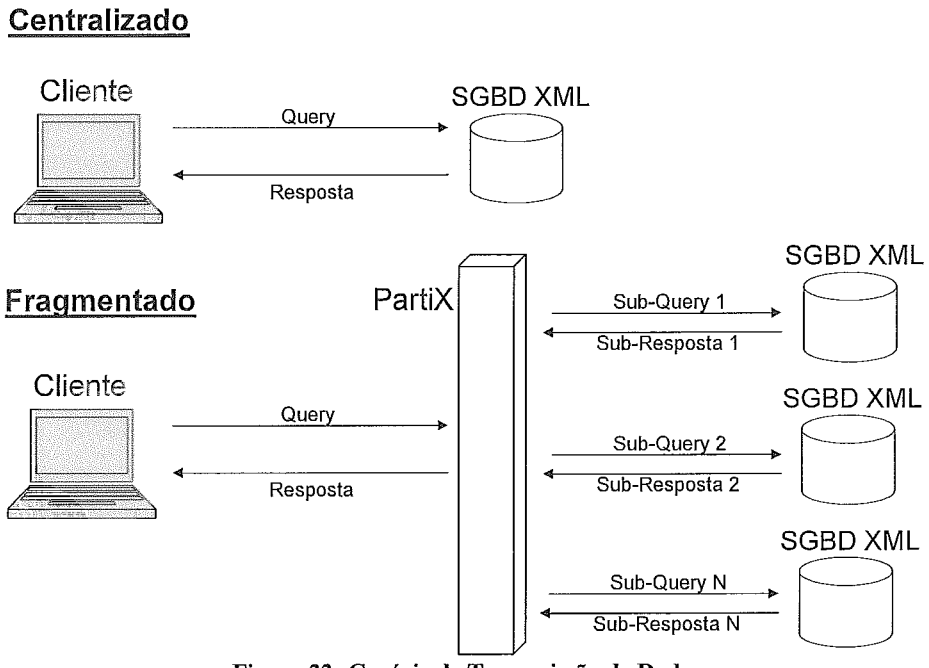


Figura 22: Cenário de Transmissão de Dados

Podemos observar que o tempo total de espera no cenário centralizado (*Tec*) é dado pela soma do tempo de processamento (*Tpc*) com o tempo de envio para o cliente (t_{rc}). Já no cenário fragmentado, o tempo total de espera (*Tef*) é dado pela soma do tempo de processamento da camada PartiX (*Tpp*), do envio da resposta para o cliente (t_{rc}) e o maior valor obtido individualmente dentre os tempos de processamento em cada fragmento (*Tpf*) adicionado ao tempo de envio da resposta dos fragmentos para a camada PartiX (*Trf*). Na Figura 23 está ilustrada a fórmula para o cenário centralizado e, na Figura 24, para o cenário fragmentado.

$$Tec = Trc + Tpc$$

Figura 23: Fórmula do Tempo de Espera – Centralizado

$$Tef = Trc + Tpp + \max_{i=1}^{n}(Tpf_i + Trf_i)$$

Figura 24: Fórmula do Tempo de Espera – Fragmentado

É importante considerar o tempo de transmissão de cada fragmento para a camada PartiX. Isso em alguns casos, irá introduzir um atraso no tempo total, que poderá inviabilizar a consulta fragmentada.

Outro ponto de gargalo nesse tipo de arquitetura é o processamento adicional que pode ser realizado na camada PartiX. Se ela for responsável por montagem de resultados, ou até mesmo agregação/ordenação de algum resultado, mais atraso indesejado é acrescido. Porém, esses problemas podem ser minimizados através de um Projeto de Fragmentação, que irá criar os fragmentos e alocá-los de acordo com as consultas mais freqüentes e os custos de transmissão. Nessa dissertação, não consideramos a existência de uma metodologia de projeto de fragmentação, e não é nossa intenção discuti-lo aqui. Nosso objetivo é definir e avaliar a fragmentação de coleções de documentos XML.

Note-se que as fórmulas aqui apresentadas retornam o tempo final. Esse tempo irá variar de acordo com a rede existente entre o cliente e banco (ou, no caso cenário fragmentado, entre o cliente e a camada PartiX). O tempo fornecido nos testes ignora esse tempo, já que o volume de dados retornado por ambos os cenários é idêntico.

5.4.1 Documentos de Consulta e de Resultados

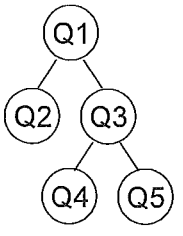
As consultas são informadas para o protótipo fartes, já alteradas para o processamento distribuído. No futuro, ele irá possuir os metadados necessários para fazer o processamento por si só. O documento XML contém as consultas que serão executadas, já em sub-consultas, e o número de vezes que serão executadas. Na Figura 25 temos um exemplo de um documento com as consultas.

```
<Script>
  <Sample>10</Sample>
  <QueryPlan name="QP1">
    <Query name="QP1_1" server="http://127.0.0.1:8080/exist/xmlrpc" ancestor="">
      <Text><![CDATA[
      for $b in collection("/db/Exemplos/FragVertical/XMark15Mb/Sample2/F3")/site/open_auctions/open_auction
      return <increase> {$b/bidder[1]/increase/text()} </increase>
      ]]>
      </Text>
    </Query>
  </QueryPlan>
</Script>
```

Figura 25: Exemplo de Documento de Consulta

Todo documento de consulta tem como raiz o elemento Script. O elemento Sample define quantas vezes cada consulta é executada. O resto do documento é composto de um elemento QueryPlan para cada consulta a ser realizada. Cada elemento QueryPlan possui um atributo chamado name, que define o nome da consulta dentro do script, além deste ser utilizado para identificar a consulta no documento de saída. Além disso, o elemento QueryPlan pode conter um ou vários elementos Query, que irão definir qual consulta será realizada, em que servidor (definido no atributo server), e em qual ordem (definida no atributo ancestor).

Um elemento Query também tem um nome (atributo name) que é usado para indicar qual consulta deve ser executada antes dela. Todo elemento Query possui um elemento Text com o texto da consulta. Eventualmente, pode ter um elemento Cleanup que contém um atributo collection (identifica o nome da coleção a ser removida). Esse elemento é usado quando desejamos remover uma coleção após uma consulta. A idéia é limpar eventuais coleções temporárias que são criadas para auxiliar no processamento da consulta. Na Figura 26 temos um exemplo do uso do atributo ancestor para definir um plano de consulta que tenha dependências.




```
<Query name="Q1" ancestor="">
<Query name="Q2" ancestor="Q1">
<Query name="Q3" ancestor="Q1">
<Query name="Q4" ancestor="Q3">
<Query name="Q5" ancestor="Q3">
```

Árvore do Plano Preenchimento do ancestor

Figura 26: Exemplo de utilização do elemento ancestor

A saída do módulo de testes é um documento XML com o tempo gasto para a realização da consulta no servidor. Duas informações são fornecidas: o tempo quente e

o tempo frio da consulta. É importante notar que o tempo frio da primeira consulta realizada na base de dados é sempre mais alto que os demais. Na Figura 27 temos um exemplo de um documento de saída.

```
<Resultados>
  <Consulta>
    <Nome>Q3</Nome>
    <TempoFrio>828.0</TempoFrio>
    <TempoQuente>364.44444444444446</TempoQuente>
  </Consulta>
</Resultados>
```

Figura 27: Exemplo do documento de saída

Todo arquivo de saída deve ter como raiz o elemento Resultados. O documento é formado por múltiplos elementos Consulta. Cada um desses elementos possui os seguintes descendentes: Nome, que é o identificador da consulta realizada; TempoFrio sendo o tempo gasto na primeira execução da consulta na base; e TempoQuente sendo a média do tempo gasto nas demais execuções da consulta.

5.4.2 Resultados utilizando Fragmentação Horizontal

5.4.2.1 Bases de Itens

As bases de dados ItemHom, ItemHet e Item80 foram todas fragmentadas pelo mesmo tipo de fragmentação. Todas utilizam o elemento Secao na definição do predicado de fragmentação. Os testes foram executados para a base centralizada e para a mesma base, fragmentada em dois (a), quatro (b) e oito (c) fragmentos . A Tabela 13 mostra a definição destes fragmentos.

Tabela 13: Definição dos Fragmentos para as Bases ItemHom, ItemHet e Item80.

	Definição de Fragmentos
(a)	$F1 := \langle C_{itens}, \sigma_{/Item/Secao='Brinquedos' \wedge /Item/Secao='Games' \wedge /Item/Secao='Perfumaria' \wedge /Item/Seção='Eletrônicos'} \rangle$ $F2 := \langle C_{itens}, \sigma_{/Item/Secao='Livraria' \wedge /Item/Secao='CD' \wedge /Item/Secao='DVD' \wedge /Item/Seção='Vestuário'} \rangle$
(b)	$F1 := \langle C_{itens}, \sigma_{/Item/Secao='Brinquedos' \wedge /Item/Secao='Games'} \rangle$ $F2 := \langle C_{itens}, \sigma_{/Item/Secao='Perfumaria' \wedge /Item/Seção='Eletrônicos'} \rangle$ $F3 := \langle C_{itens}, \sigma_{/Item/Secao='CD' \wedge /Item/Secao='DVD'} \rangle$ $F4 := \langle C_{itens}, \sigma_{/Item/Secao='Livraria' \wedge /Item/Seção='Vestuário'} \rangle$
(c)	$F1 := \langle C_{itens}, \sigma_{/Item/Secao='Brinquedos'} \rangle$ $F2 := \langle C_{itens}, \sigma_{/Item/Secao='Games'} \rangle$ $F3 := \langle C_{itens}, \sigma_{/Item/Secao='Perfumaria'} \rangle$ $F4 := \langle C_{itens}, \sigma_{/Item/Seção='Eletrônicos'} \rangle$ $F5 := \langle C_{itens}, \sigma_{/Item/Secao='CD'} \rangle$ $F6 := \langle C_{itens}, \sigma_{/Item/Secao='DVD'} \rangle$ $F7 := \langle C_{itens}, \sigma_{/Item/Secao='Livraria'} \rangle$

	$F8 := \langle C_{itens}, \sigma_{/Item/Seção='Vestuário'} \rangle$

Os resultados apresentados nas Figura 28, Figura 29, Figura 30 e Figura 31 correspondem aos gráficos de desempenho obtidos na base ItemHom.

Na base com tamanho de 5Mb podemos perceber que a consulta Q_2 não apresentou ganhos, já que o tempo de transmissão influencia o tempo de resposta final. Já nas consultas Q_6 e Q_8, apesar de apresentarem em todos os casos ganhos consideráveis, nota-se a perda de performance da consulta feita em 4 fragmentos e 8 fragmentos. A justificativa para tal discrepância foi a re-escrita da consulta. Com 4 fragmentos, é mencionado o predicado de seleção. Contudo, na base de 8 fragmentos, este predicado é omitido, já que todos os documentos do fragmento são da seção estipulada. Pode ser que este predicado altere o plano de consulta realizado pelo eXist, o que ocasiona um maior tempo de resposta. A consulta Q4 apresenta um comportamento anômalo. Com quatro fragmentos, a consulta apresenta pior desempenho que na base centralizada, e nos demais casos, o resultado é melhor. O predicado de seleção da seção é removido na consulta utilizada na base com oito fragmentos. Isso explicaria o aumento do tempo, conforme já citado. Porém, não justificaria o péssimo desempenho da consulta com quatro fragmentos. A possível explicação é que o eXist tenha feito alguma operação de *refresh* de *cache* ou cálculo de estatística que afetou essa consulta em particular. O SGBD está sempre gerando estatística. Nas demais consultas, por afetarem um conjunto maior de dados, em base também maiores, esse tempo não foi significante. Já na consulta citada, foi decisivo para a alteração do resultado.

Na base de 20Mb todas as consultas em todos os fragmentos são melhores que na base centralizada. Já nas bases de 100Mb e 250Mb, todas as consultas apresentam resultados melhores quando executadas em bases fragmentadas. Porém, a consulta Q_6 com oito fragmentos apresenta um tempo maior que com quatro fragmentos. A justificava é a mesma já apresentada para a base de 5Mb.

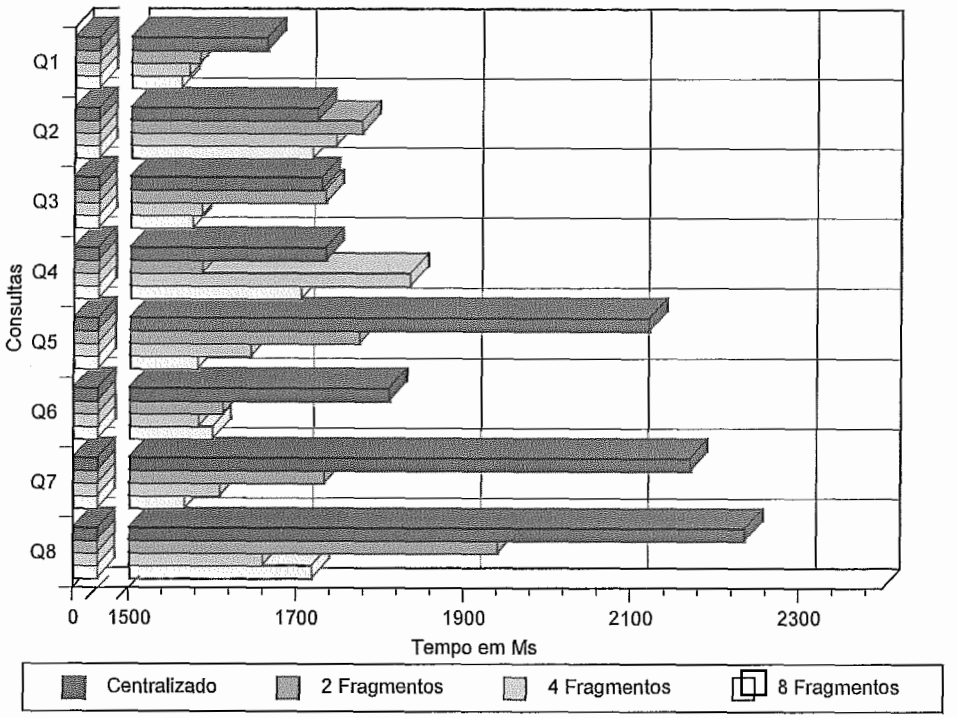


Figura 28: Gráfico de Desempenho. Base ItemHom de 5Mb.

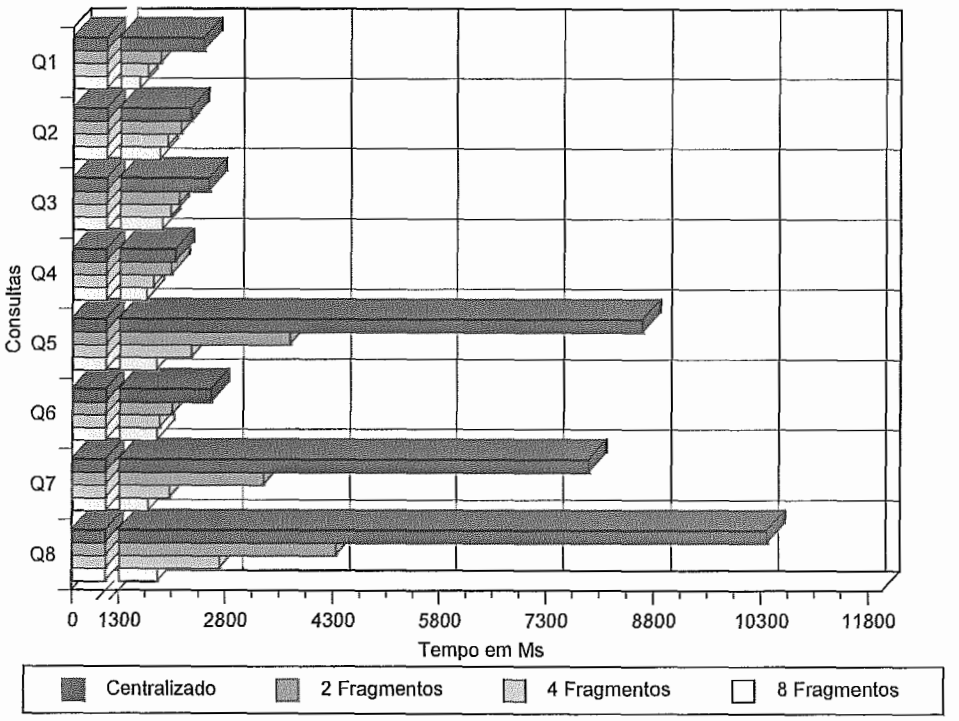


Figura 29: Gráfico de Desempenho. Base ItemHom de 20Mb.

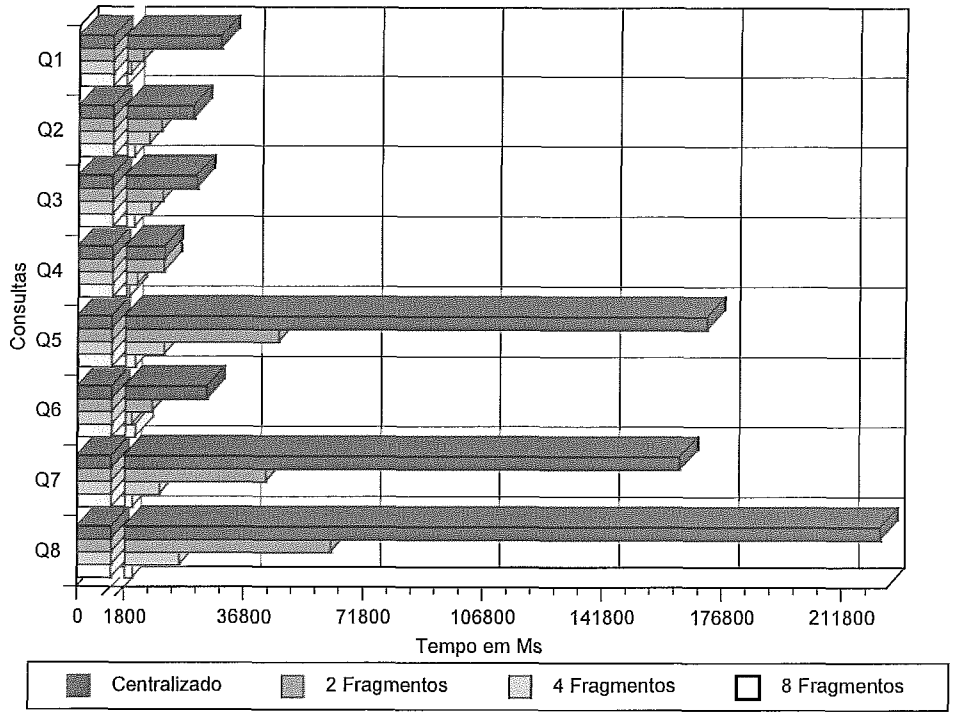


Figura 30: Gráfico de Desempenho. Base ItemHom de 100Mb.

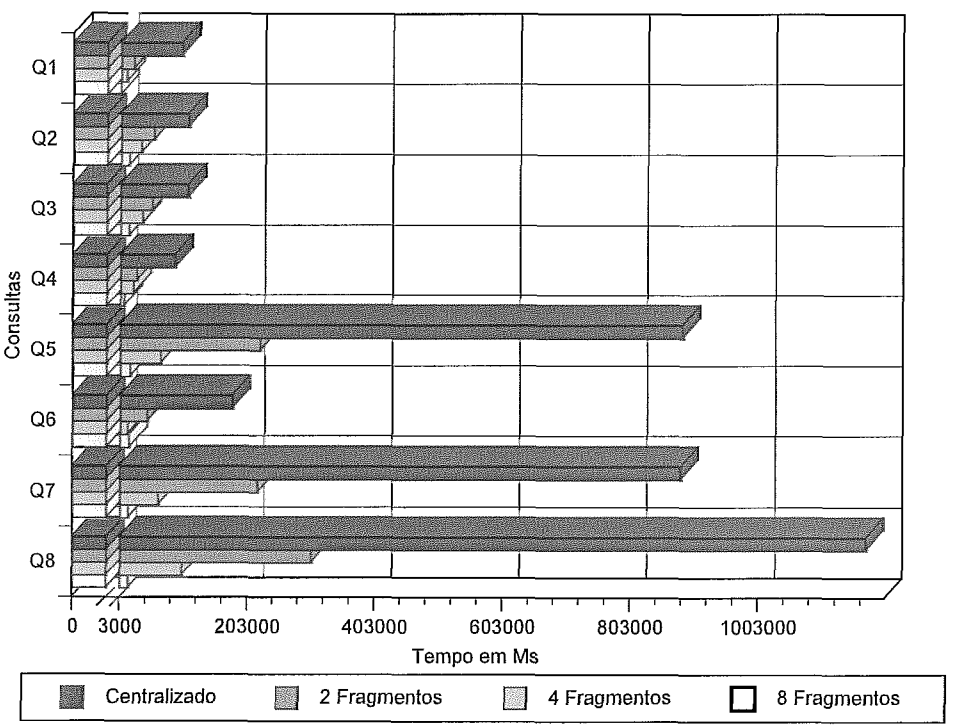


Figura 31: Gráfico de Desempenho. Base ItemHom de 250Mb.

Os resultados apresentados nas Figura 32, Figura 33, Figura 34 e Figura 35 correspondem aos gráficos de desempenho obtidos na base ItemHet, com distribuição heterogênea dos dados. Fizemos testes com bases com distribuições homogêneas (base ItemHom) e heterogêneas (base ItemHet), para avaliar a diferença de desempenho nesses casos. Analisando os gráficos das bases ItemHom e ItemHet, podemos perceber que não houve diferenças significativas, a não ser aquelas já esperadas. Em algumas consultas o tempo de resposta na base ItemHom foi superior, por se tratar de um conjunto de dados ligeiramente maior que na base ItemHet. Porém, conforme aumentamos o tamanho da base como um todo, as consultas tendem a ficar cada vez mais com tempos semelhantes.

Na base de 5Mb, as consultas Q_1 e Q_2 (utilizando-se de 2 fragmentos) e a consulta Q_4 (em todos os fragmentos) apresentaram perda de desempenho, em virtude da transmissão do resultado. Já nos testes com as demais bases, somente a consulta Q6 apresentou anomalias no desempenho da consulta. O tempo de resposta com 8 fragmentos foi superior ao tempo de resposta com 4 fragmentos. Um dos motivos para essa alteração pode ter sido uma mudança na forma do acesso à coleção. Com 8 fragmentos a consulta não possui o predicado que especifica "*Seção = DVD*". Nas demais esse predicado existe. Porém, ainda foi melhor que na consulta centralizada.

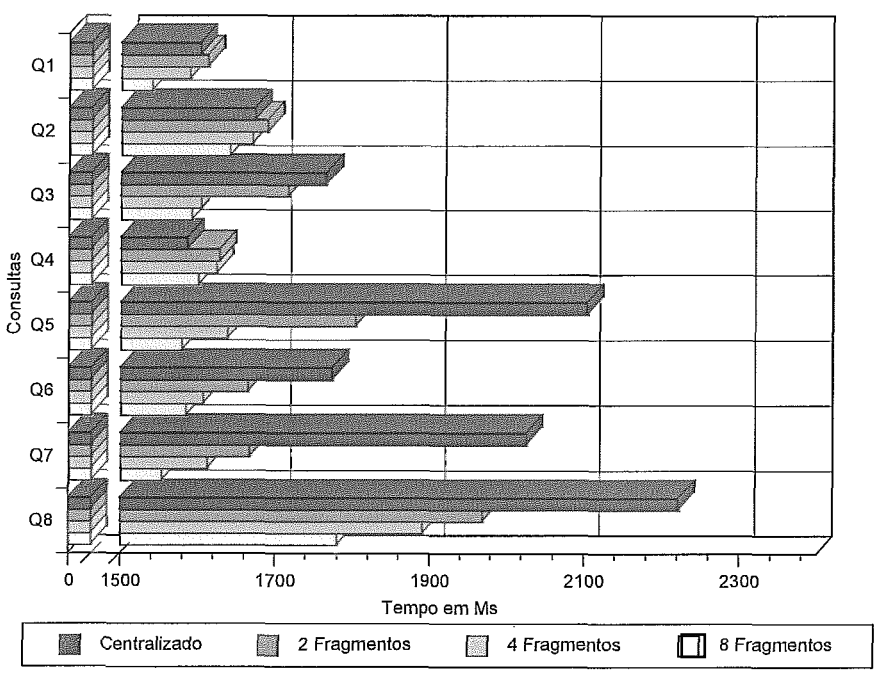


Figura 32: Gráfico de Desempenho. Base ItemHet de 5Mb.

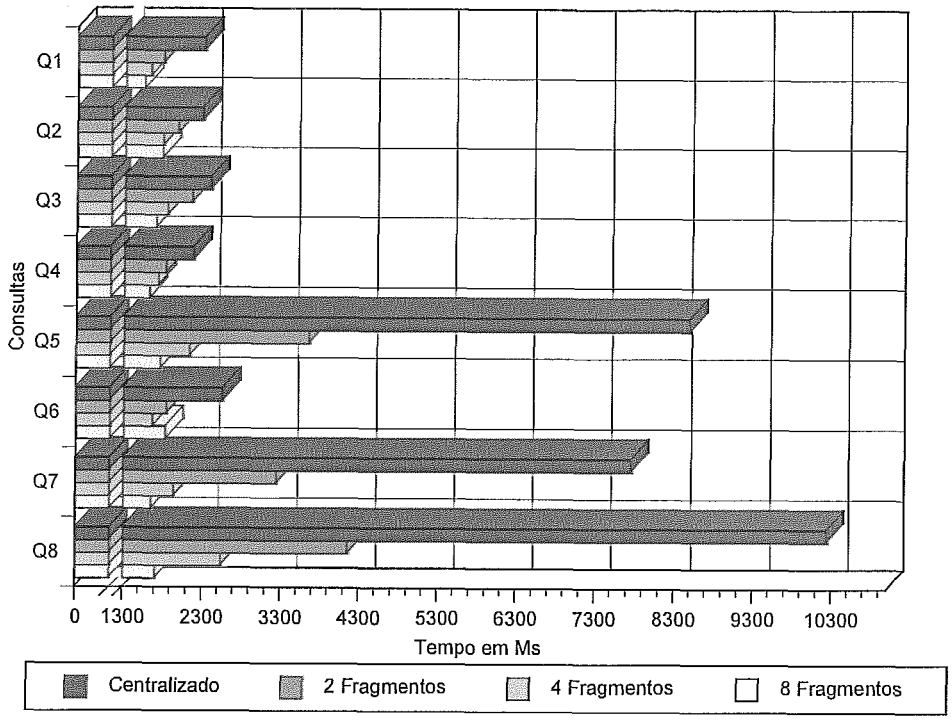


Figura 33: Gráfico de Desempenho. Base ItemHet de 20Mb.

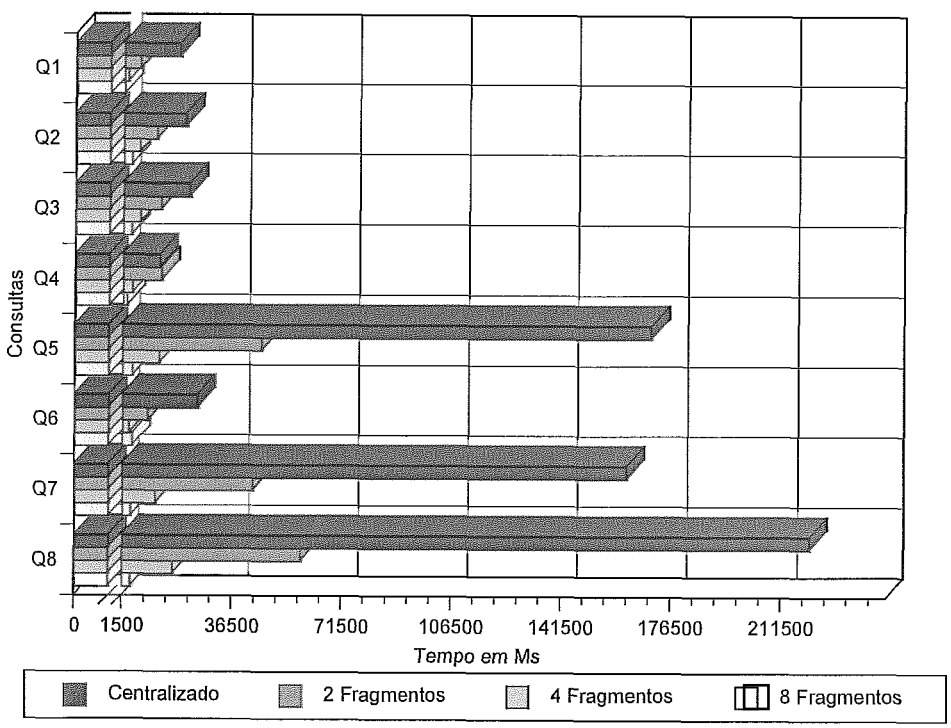


Figura 34: Gráfico de Desempenho. Base ItemHet de 100Mb.

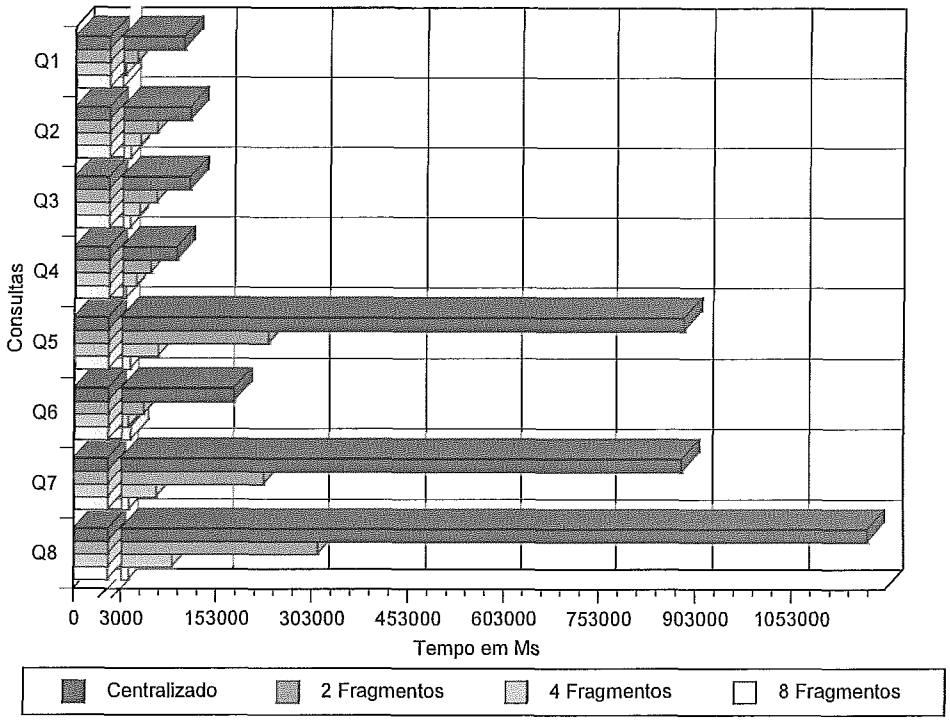


Figura 35: Gráfico de Desempenho. Base ItemHet de 250Mb.

Podemos notar nos dois casos já apresentados que as consultas mais beneficiadas pela fragmentação foram as consultas Q_5, Q_6, Q_7 e Q_8. Essas são justamente as consultas que apresentam busca em texto (Q_5 e Q_6) e uso de funções de agregação (Q_7 e Q_8). Com a existência de conjuntos de dados cada vez menores, é mais simples e menos custoso em termos de memória para o processador de consultas, o que agiliza esse tipo de operação.

Em Figura 36, Figura 37, Figura 38, Figura 39 e Figura 40 temos os resultados obtidos com o uso da Base de Dados Item80.

Podemos observar que existe perda de desempenho na base de 5Mb para as consultas Q_1, Q_2, Q_4 e Q_7, prejudicadas pelo tempo de transmissão. Na Base Item80, apesar do uso das mesmas consultas da Base ItemHom e ItemHet, temos um tamanho maior do documento, o que irá influir mais no tempo de transmissão. A consulta Q_6 continua apresentando o mesmo comportamento encontrado nos testes anteriores, em todos os gráficos de desempenho da Base Item80.

Na base com 20Mb, somente as consultas Q_1 e Q_2 ainda são prejudicadas pelo tempo de transmissão. A consulta Q_3 se torna equivalente à base centralizada, enquanto que Q_7 já apresenta melhoras.

A partir da base de 100Mb, existe sempre ganho em todas as consultas.

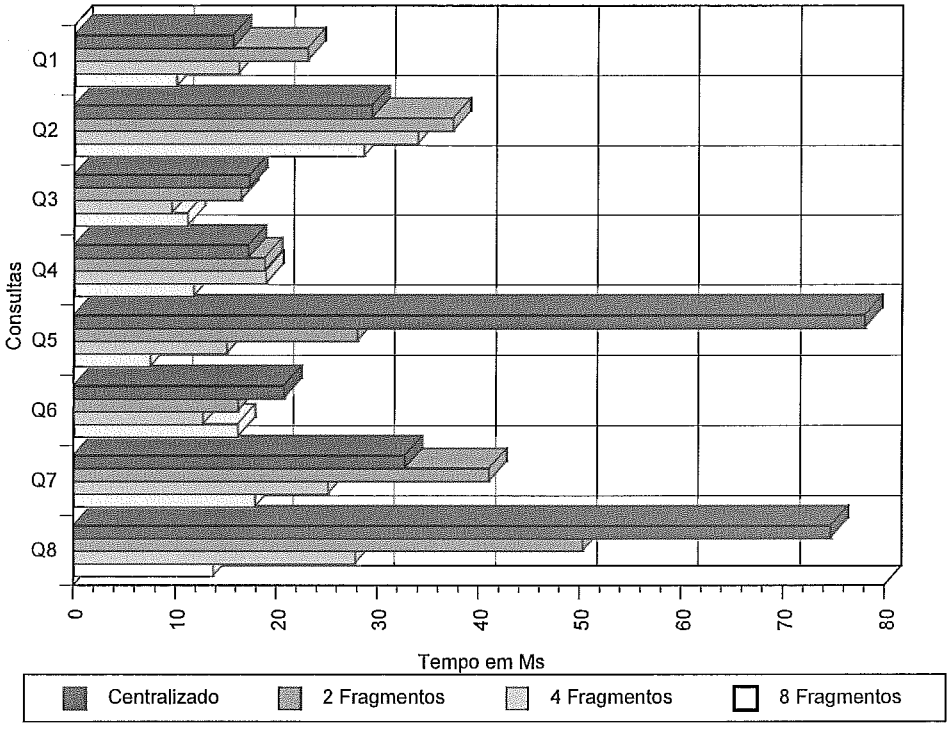


Figura 36: Gráfico de Desempenho. Base Item80 de 5Mb.

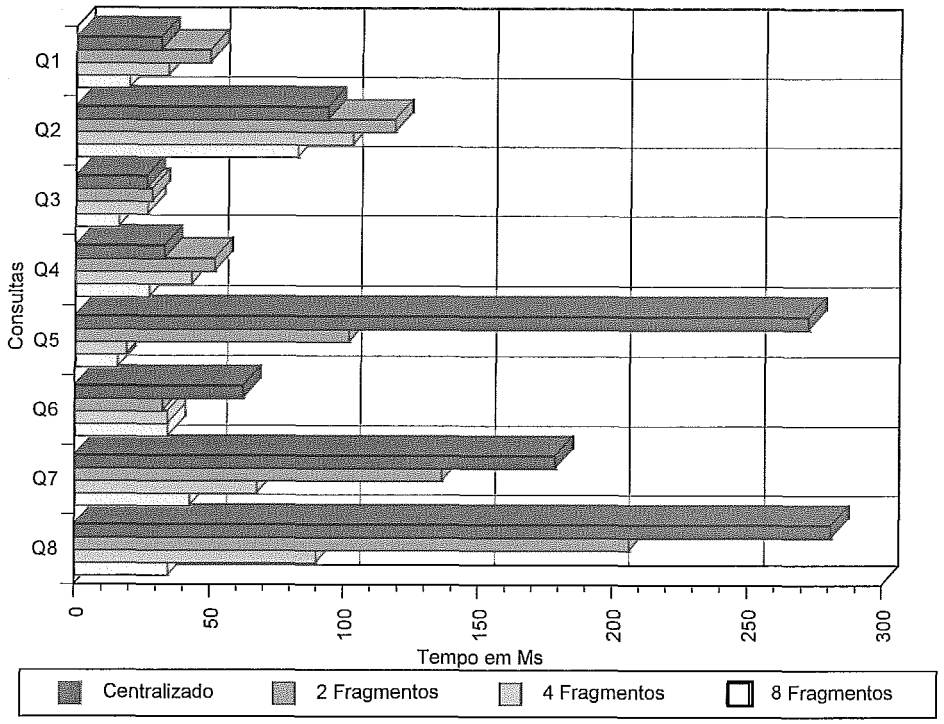


Figura 37: Gráfico de Desempenho. Base Item80 de 20Mb.

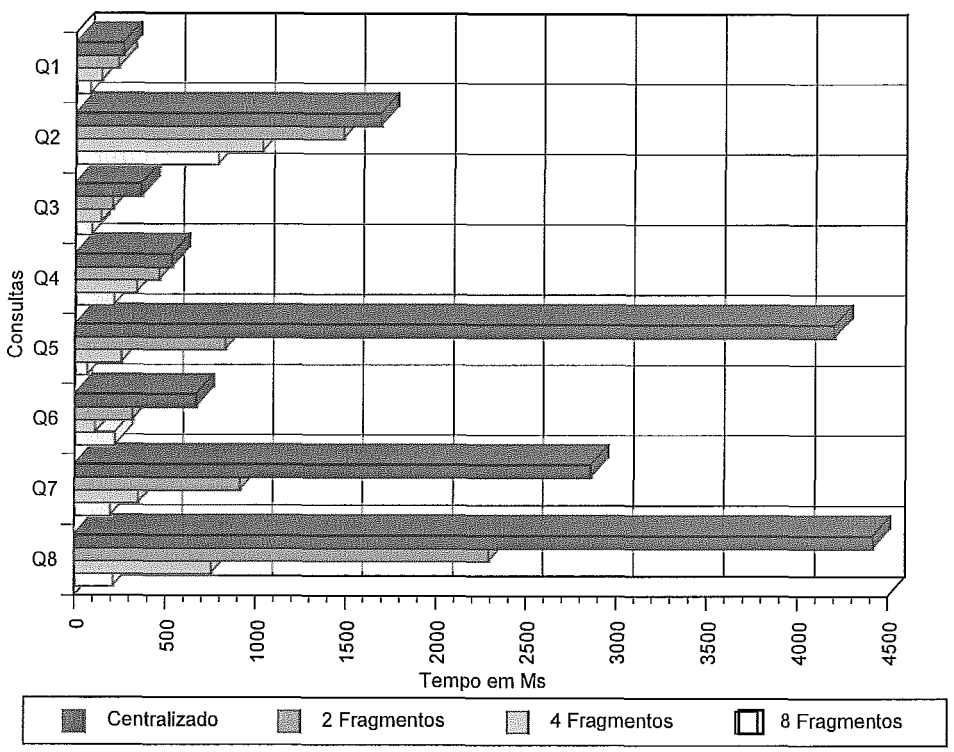


Figura 38: Gráfico de Desempenho. Base Item80 de 100Mb.

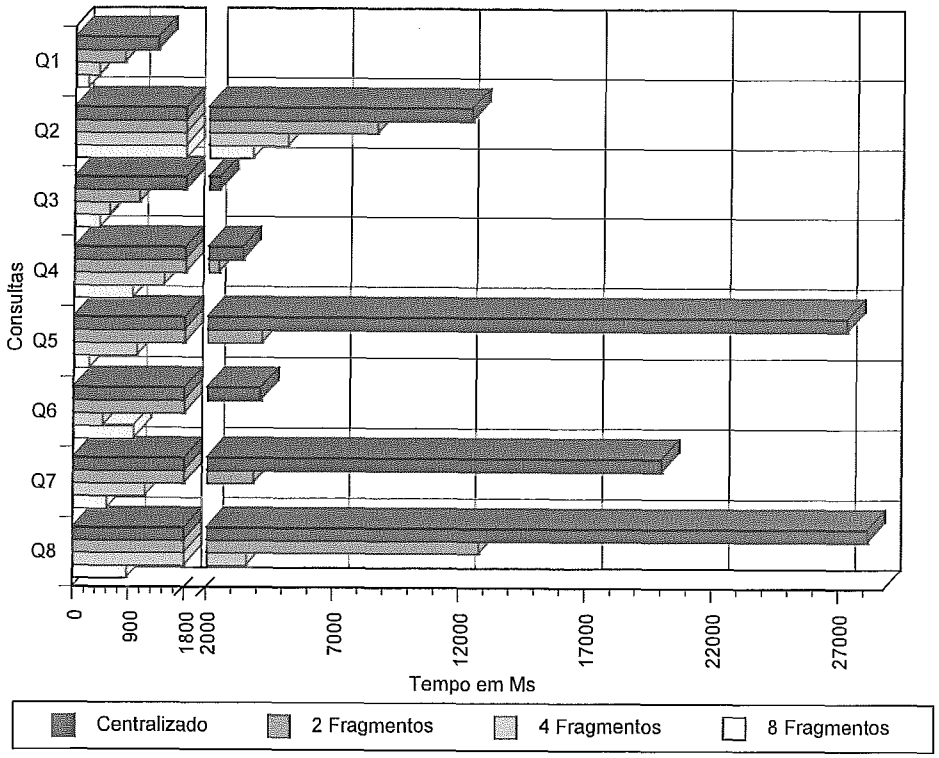


Figura 39: Gráfico de Desempenho. Base Item80 de 250Mb.

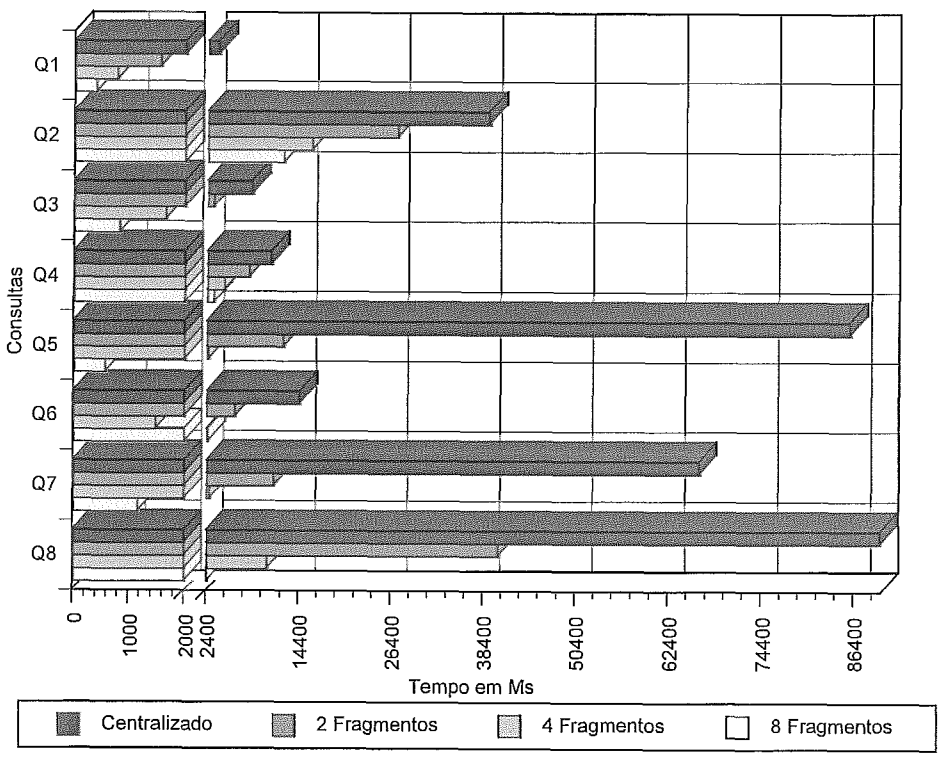


Figura 40: Gráfico de Desempenho. Base Item80 de 500Mb.

Com respeito à Base Item80, é importante notar que houve benefícios claros para as consultas que utilizam funções de agregação (Q_7 e Q_8) e busca em texto (Q5 e Q6), como era previsto, dados os resultados anteriores com as bases ItemHom e ItemHet. Um fato interessante são os tempos das consultas na base Item80. Todos são menores que nas bases ItemHom e ItemHet. A causa dessa melhora no tempo de resposta, apesar de serem as mesmas consultas no mesmo esquema, está no tamanho do documento. Realizar a carga de milhares de documentos pequenos, um *parsing* nestes e produzir uma resposta causa um atraso significativo. Como os arquivos da Base Item80 são maiores, temos menos documentos a serem analisados, apesar de os tamanhos absoluto das bases serem praticamente idênticos. Ou seja, o tempo de *parsing* e carga dos documentos em memória são significativos para consultas XQuery.

5.4.2.2 Base de Artigos – XBench

Agora vamos analisar uma base com um esquema diferente. A Base ArtHoz, como já foi descrita na Seção 5.2, é uma base de artigos usada pelo XBench. Muitas de suas consultas são diretas, visando a recuperar um único documento. A Tabela 14 mostra como foi feita a fragmentação nesta coleção, que foi dividida em dois, quatro e oito fragmentos. Somente mostramos a definição da fragmentação para o caso com dois fragmentos, pois os demais são semelhantes. A fragmentação foi feita em cima do atributo id existente no elemento artigo. A fragmentação foi uniforme, portanto cada fragmento possui o mesmo número de documentos.

Tabela 14: Definição de Fragmento para Base ArtHor.

Definição de Fragmentos
$F1 := \langle C_{artigos}, \sigma_{/article/@id >= 1 \wedge /article/@id <= 235} \rangle$
$F2 := \langle C_{artigos}, \sigma_{/article/@id > 235} \rangle$

As Figura 41, Figura 42, Figura 43 e Figura 44 mostram o desempenho das consultas nas diversas bases de testes utilizadas para este esquema.

Podemos observar que, na base de 5Mb, somente as consultas Q_4, Q_9 e Q_{10} apresentaram melhoras. As demais se tornaram equivalentes. Mas é importante notar que, destas, somente Q_2, Q_3 e Q_6 não são consultas diretas (que recuperam apenas um único documento). Porém, utilizam predicados de seleção bem particulares, que restringem bastante o resultado da consulta. Além disso, ainda consideramos o tempo de

transmissão do resultado. O mesmo ocorre para as consultas na base de 20Mb. Somente a partir da base de 100Mb todas as demais consultas começam a apresentar ganhos mais significativos. As consultas Q_4, Q_9 e Q_{10} já apresentam ganhos expressivos. Estas são justamente as consultas que envolvem busca em texto (Q_4 e Q_9) e um teste existencial (Q_{10}). Com isso, comprovamos que a fragmentação horizontal é recomendada principalmente em casos de processamento de texto e uso de funções de agregação. Um caso que merece atenção é a anomalia ocorrida na consulta Q9 para as bases de 100Mb e 250Mb. Nestes casos, a consulta na base com quatro fragmentos foi superior à base com dois fragmentos, porém, ainda menor que na base centralizada. Não encontramos explicação para essa discrepância, e acreditamos ter ocorrido alguma anomalia no processamento da função de busca em texto nesses casos. Os testes foram repetidos com essa consulta e apresentaram resultados semelhantes. Como não houve mudança na consulta, descartamos problemas com o plano de consulta. Também não foi devido ao tempo de transmissão, já que o próprio processamento da consulta já apresentava o comportamento anômalo.

Analisando esta base, podemos ver que a fragmentação horizontal não é tão útil se as consultas a serem realizadas envolvem muitas consultas diretas (retornando apenas um documento) se a base é pequena (menor que 100Mb). Mesmo em bases menores, se houver muitas consultas que usam buscas em texto e agregações, ressaltamos a importância da fragmentação para amenizar o tempo de resposta.

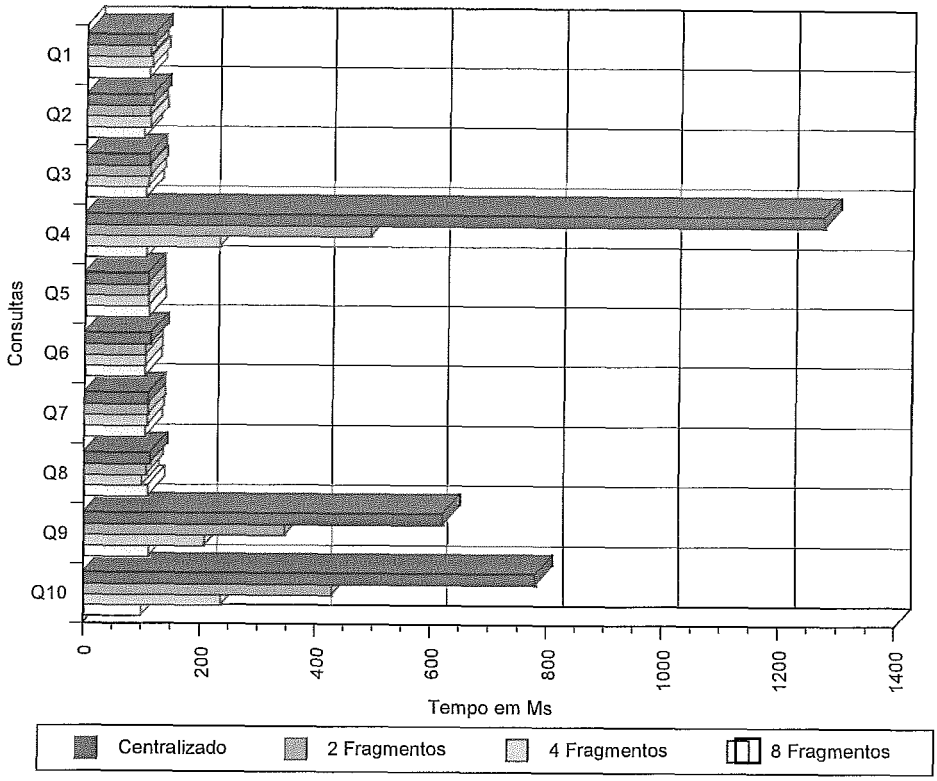


Figura 41: Gráfico de Desempenho. Base ArtHor de 5Mb.

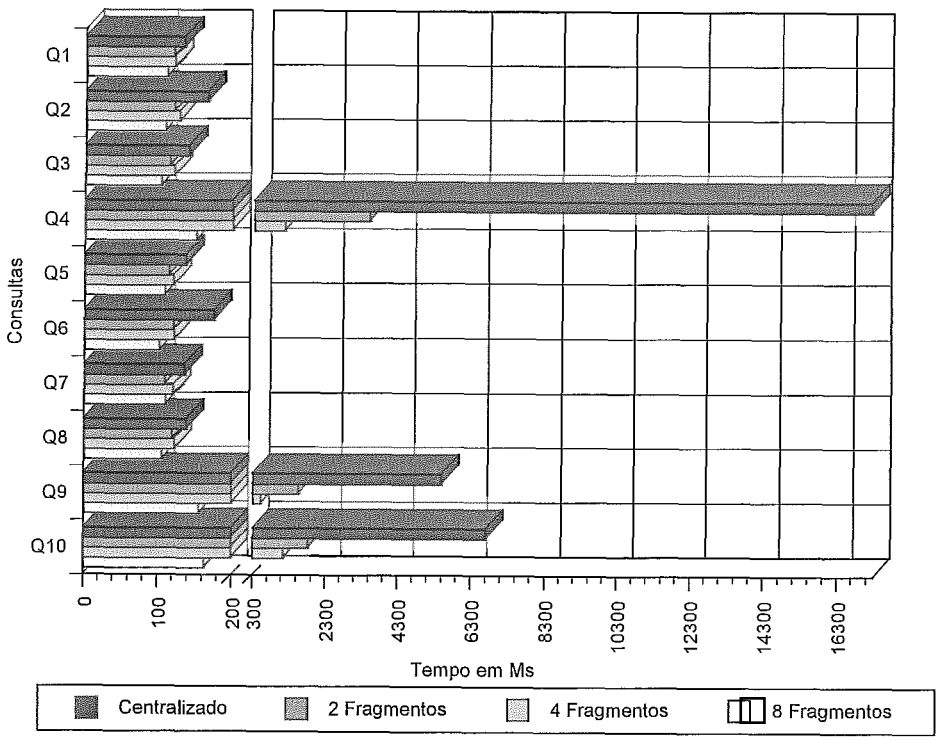


Figura 42: Gráfico de Desempenho. Base ArtHor de 20Mb.

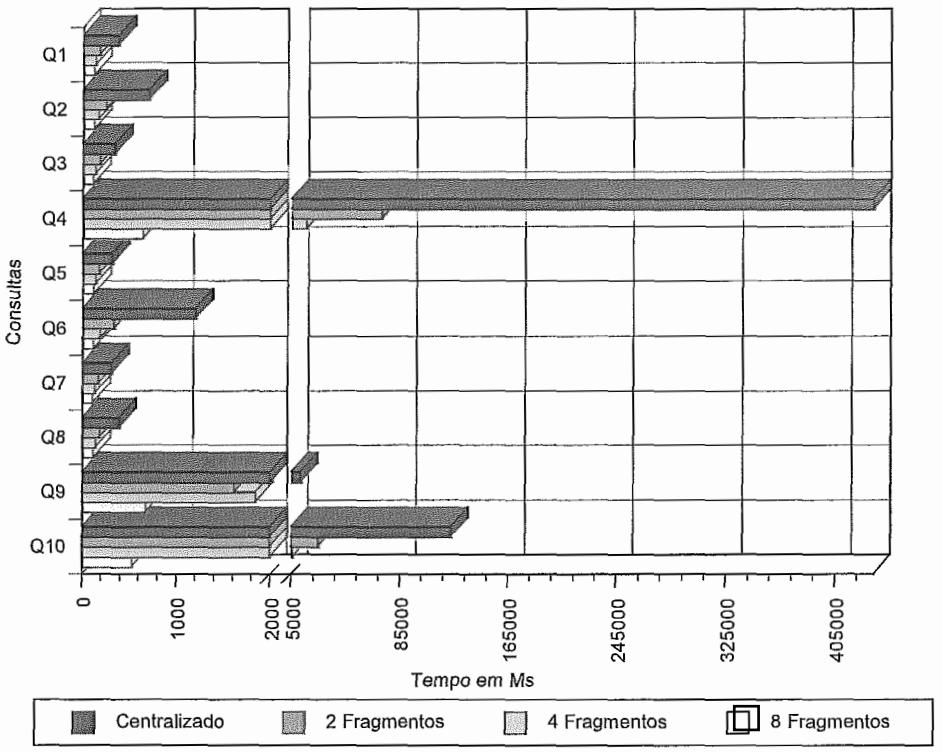


Figura 43: Gráfico de Desempenho. Base ArtHor de 100Mb.

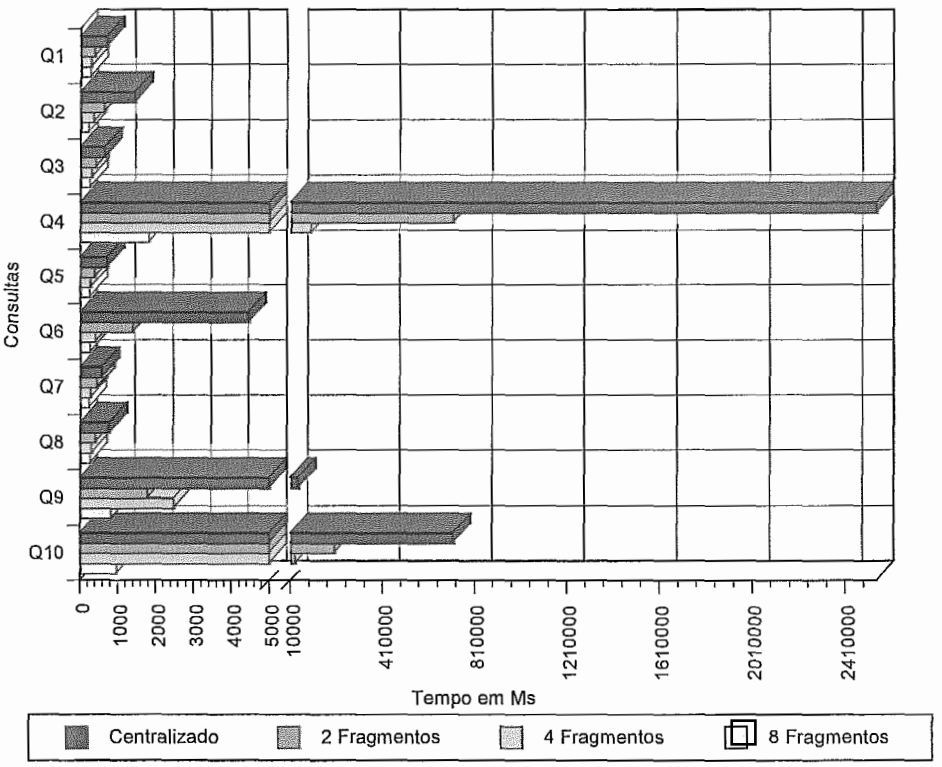


Figura 44: Gráfico de Desempenho. Base ArtHor de 250Mb.

5.4.3 Resultados utilizando Fragmentação Vertical

5.4.3.1 Base do XMark

Nos testes realizados foi prevista a execução de uma rodada de testes para a base de dados do benchmark XMark. Porém, vários problemas ocorreram na fase de carga e fragmentação do esquema, que impediram testes detalhados de desempenho. O esquema do XMark foi basicamente criado para avaliação de quão completa é a implementação da XQuery em um SGBD XML (ou outra ferramenta que use processamento de XQuery).

Como pudemos observar na seção 5.2.2, o esquema é bem aninhado, e em sua DTD existe a definição de vários pares ID/IDREFs (entre Itens e Leilões, entre itens e compradores, etc). O primeiro problema encontrado foi exatamente este. Pelas considerações feitas na seção 3.8.3, é inviável a fragmentação do esquema, dada a quantidade de pares ID/IDREF existentes.

Mas, mesmo assim, tentamos ignorar essa recomendação e fragmentar o esquema sem a preocupação com os pares ID/IDREFs. Neste caso, o problema foi o próprio SGBD XML. Ao ser feita a carga do documento com do XMark, o eXist acusou erros, mencionando que o documento é profundo e possuía muitos aninhamentos.

5.4.3.2 Base de Artigos – XBench

Utilizamos a Base ArtVert, com o esquema de artigos do XBench (vide seção 5.2) para realizar testes com a fragmentação vertical. Nas Figura 46, Figura 47, Figura 48 e Figura 49 temos os gráficos de desempenho para essa base. Na Figura 45 vemos como foram definidos os fragmentos. Diante das consultas utilizadas, vimos necessidade de apenas três fragmentos.

$$F1_{artigos} := \langle C_{artigos}, \pi_{/article/prolog} \rangle$$
$$F2_{artigos} := \langle C_{atigos}, \pi_{/article/body} \rangle$$
$$F3_{artigos} := \langle C_{artigos}, \pi_{/article/epiog} \rangle$$

Figura 45: Definição de Fragmentos para Base ArtVert

Na base com 5Mb, podemos perceber ganhos em todas as consultas, com exceção das consultas Q_4 e Q_{10}. Na fragmentação vertical, as principais beneficiadas são consultas que utilizam apenas um único fragmento. As consultas Q_4, Q_7, Q_8 e Q_9 precisam de dados de mais de um fragmento. Logo, elas podem ser naturalmente

prejudicadas pela fragmentação. A consulta Q_4 não apresenta ganhos de desempenho significativos em nenhum caso, exceto na base de 100Mb. Nesse caso, houve um pequeno ganho de desempenho, que não consideramos ser significativo. Acreditamos que, mais uma vez, alguma estatística ou plano de consulta favoreceu a execução da consulta nessa base de dados, já que nos demais casos só houve perda. Em geral pequena, mas sempre mais lenta que na base centralizada. Já na base de 20Mb, todas as consultas apresentam ganhos (menos Q_4). Inclusive a consulta Q_{10}, que na base anterior apresentou perda de desempenho.

Conforme o tamanho da base de dados vai aumentando, os ganhos vão diminuindo cada vez mais. Na base de 100Mb, as consultas Q_6 e Q_9 são equivalentes à base centralizada. Já com 250Mb, as consultas Q_6, Q_9 e Q_3 também se tornam equivalentes. Com esses resultados, podemos ver que a fragmentação vertical é útil para quando temos base de tamanho pequeno (menor de 100Mb) e consultas diretas ou com predicados de seleção bem restritivos. As consultas com pior desempenho envolveram buscas em texto.

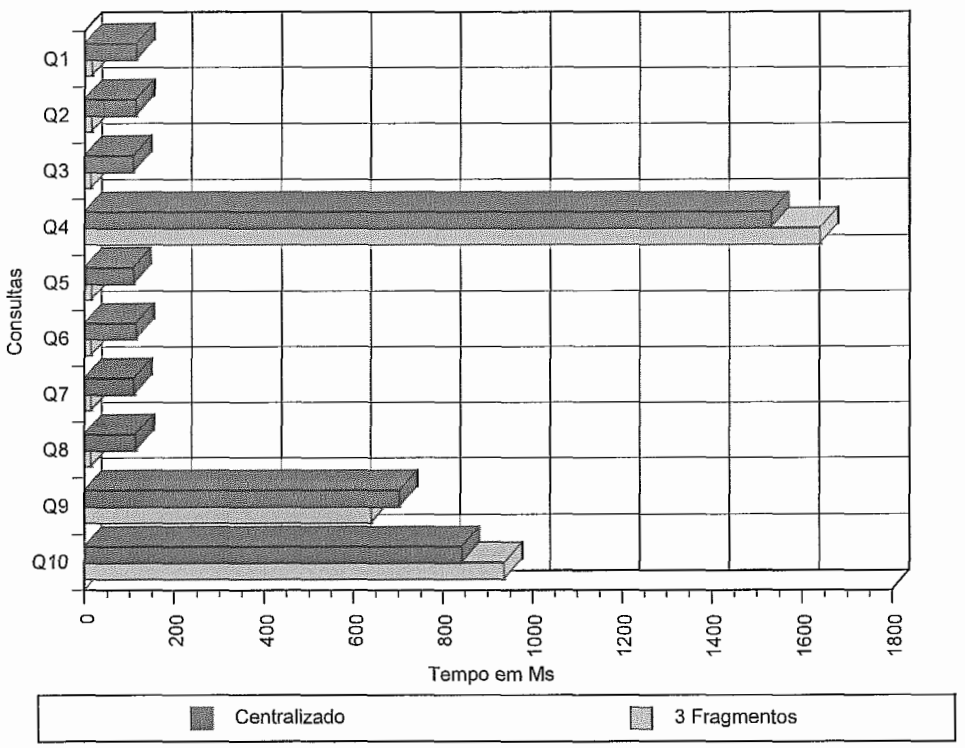


Figura 46: Gráfico de Desempenho. Base ArtVert de 5Mb.

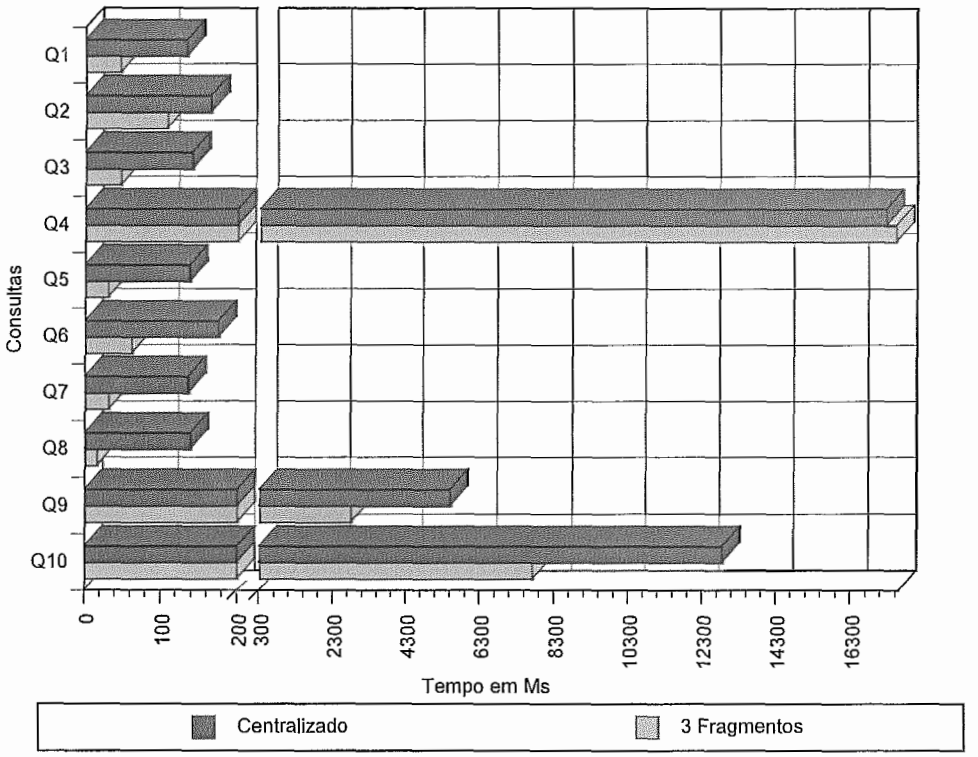


Figura 47: Gráfico de Desempenho. Base ArtVert de 20Mb.

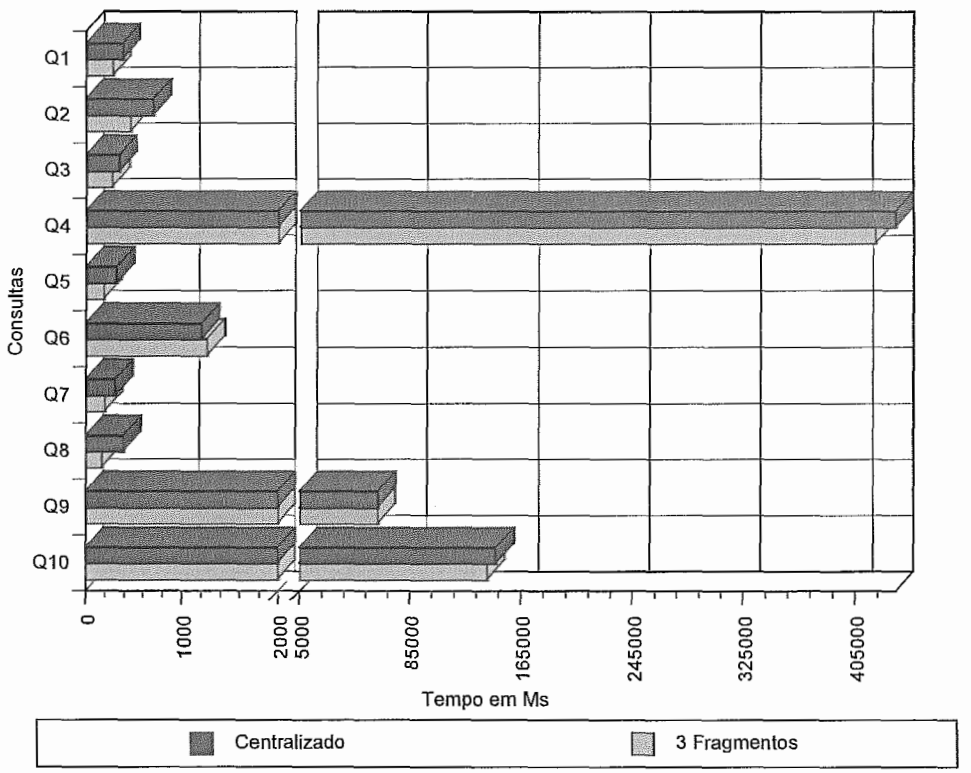


Figura 48: Gráfico de Desempenho. Base ArtVert de 100Mb.

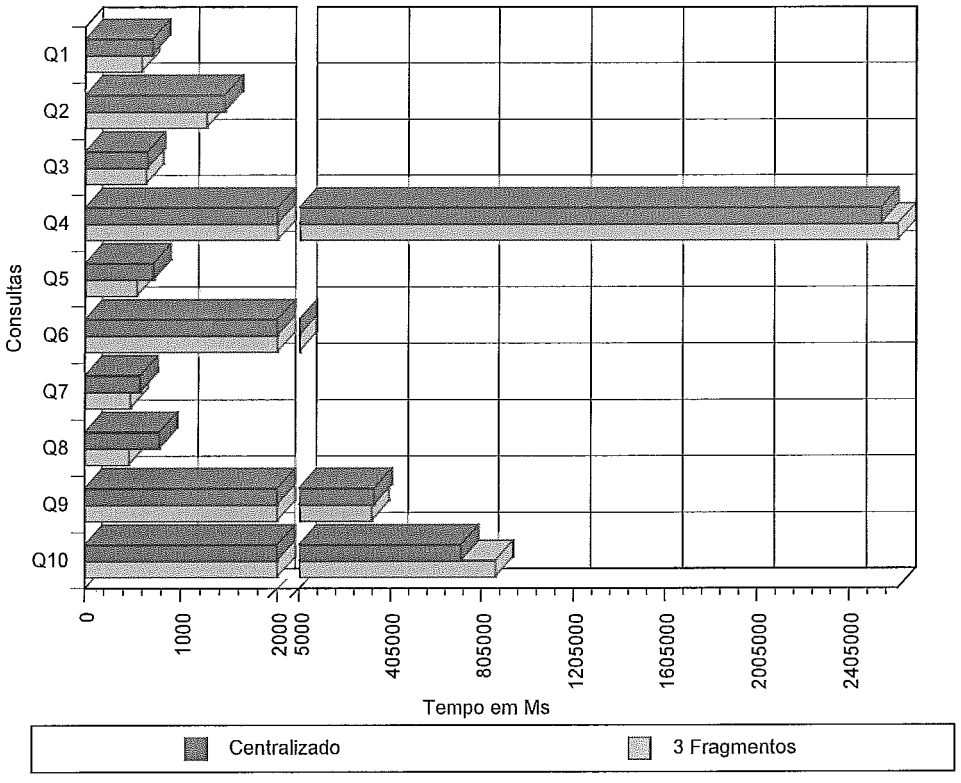


Figura 49: Gráfico de Desempenho. Base ArtVert de 250Mb.

5.4.4 Resultados utilizando Fragmentação Híbrida

5.4.4.1 Base de Loja

A fragmentação híbrida foi avaliada através dos fragmentos descritos na Tabela 15, utilizando-se as consultas descritas na seção 5.2 para a Base LojaHib. Como iremos ver mais adiante, a fragmentação híbrida foi muito influenciada pelo tamanho de retorno dos documentos. Por isso, apresentamos os resultados de desempenho com e sem os tempos de transmissão. Nas Figura 50, Figura 51, Figura 52, Figura 53 e Figura 54 temos os gráficos de desempenho considerando o tempo de transmissão. Nas Figura 55, Figura 56, Figura 57, Figura 58 e Figura 59 temos os gráficos de desempenho sem considerar o tempo de transmissão.

Tabela 15: Definição de fragmentos para a Base LojaHib.

$$F1 := \langle C_{loja}\ ,\ \pi_{/Loja,\{/Loja/Itens\}}\ \rangle$$
$$F2 := \langle C_{loja},\ \pi_{/Loja/Itens} \bullet \sigma_{/Loja/Itens/Item/Seção="CD"} \rangle$$
$$F3 := \langle C_{loja},\ \pi_{/Loja/Itens} \bullet \sigma_{/Loja/Itens/Item/Seção="DVD"} \rangle$$
$$F4 := \langle C_{loja},\ \pi_{/Loja/Itens} \bullet \sigma_{/Loja/Itens/Item/Seção="Papelaria"} \rangle$$
$$F5 := \langle C_{loja},\ \pi_{/Loja/Itens} \bullet \sigma_{/Loja/Itens/Item/Seção="Livraria"} \rangle$$

Consideramos praticamente as mesmas consultas e os mesmos valores de seleção utilizados nas Bases ItemHom, ItemHet e Item80. Com isso, boa parte das consultas passou a retornar todo o conteúdo do elemento Item. Este foi o grande problema de desempenho encontrado, que afetou todas as consultas. Isso serve pra ressaltar que além de um bom projeto de fragmentação, devem ser feitas consultas que retornem exatamente o que é necessário para uma dada aplicação. Pelo fato de o formato XML ser verborrágico, um elemento desnecessário pode trazer uma árvore descendente de tamanho razoável, o que irá prejudicar o tempo geral da consulta.

Uma outra característica geral que notamos ao realizar nossos testes foi a forma de implementação do fragmento horizontal na fragmentação híbrida. Foi para nós natural pensar em tomarmos o documento único que representa a coleção a ser fragmentada, utilizarmos a operação de poda, e, para cada nodo Item selecionado, gerarmos um documento independente e armazená-lo. Essa abordagem, que chamamos de Modo de Implementação 1 (Modo Impl 1, nos gráficos), se mostrou péssima. O principal motivo é que anteriormente, o processador de consulta tinha que realizar um *parsing* em um documento enorme, e realizar a consulta nele. Agora, nesse fragmento, o processador passa a ter que realizar um *parsing* em uma série de centenas de documentos menores. Ora, o tempo de *parsing* do documento único é muito menor do que a carga e *parsing* das centenas de documentos menores. Para resolver este problema, implementamos o fragmento horizontal com um documento único, exatamente como o documento original, só que apresentando somente os elementos Item obtidos com o operador de seleção. Esta abordagem for chamada Modo de Implementação 2 (Modo Impl 2, nos gráficos). E podemos ver, que esta irá vencer da base centralizada na maioria dos casos.

Quando consideramos o tempo de transmissão, o Modo de Implementação 1 perde em todas as bases, para todas as consultas, com exceção das consultas Q_9, Q_{10} e Q_{11}. As consultas Q_9 e Q_{10} são exatamente as consultas que se aproveitam da poda do elemento Itens para agilizar o *parsing* do documento. A consulta Q_{11} utiliza uma função de agregação, que se apresentou com um bom desempenho a partir da base com 100Mb. Nas demais bases, apresentou perda (base de 5Mb) ou um comportamento anômalo (base de 20Mb).

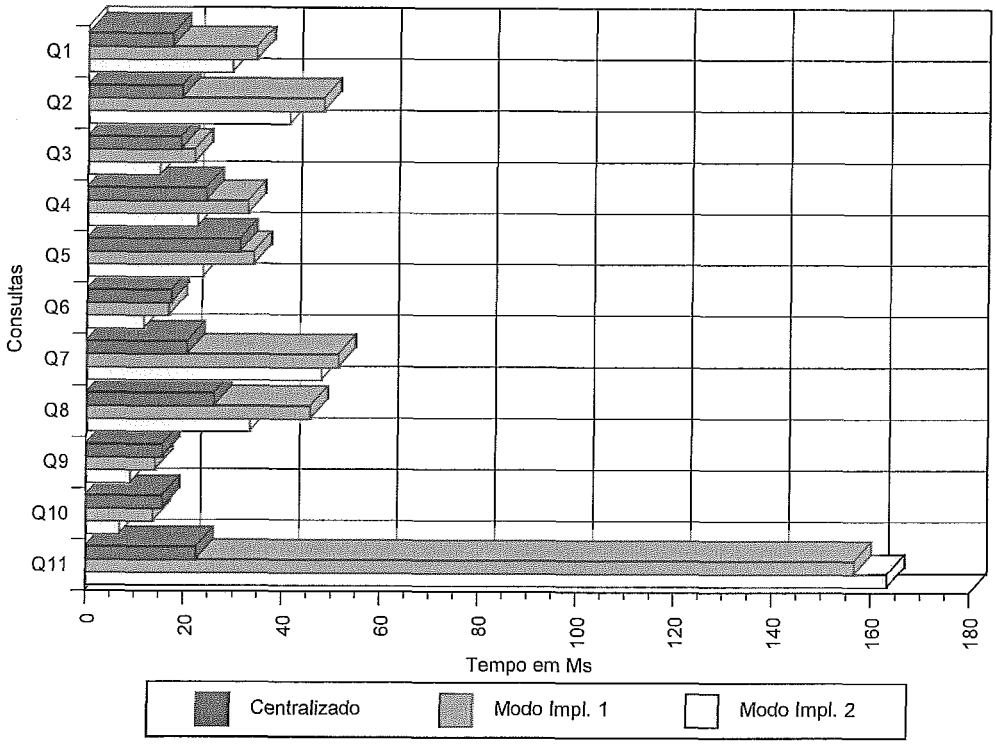


Figura 50: Gráfico de Desempenho. Base LojaHib de 5Mb.

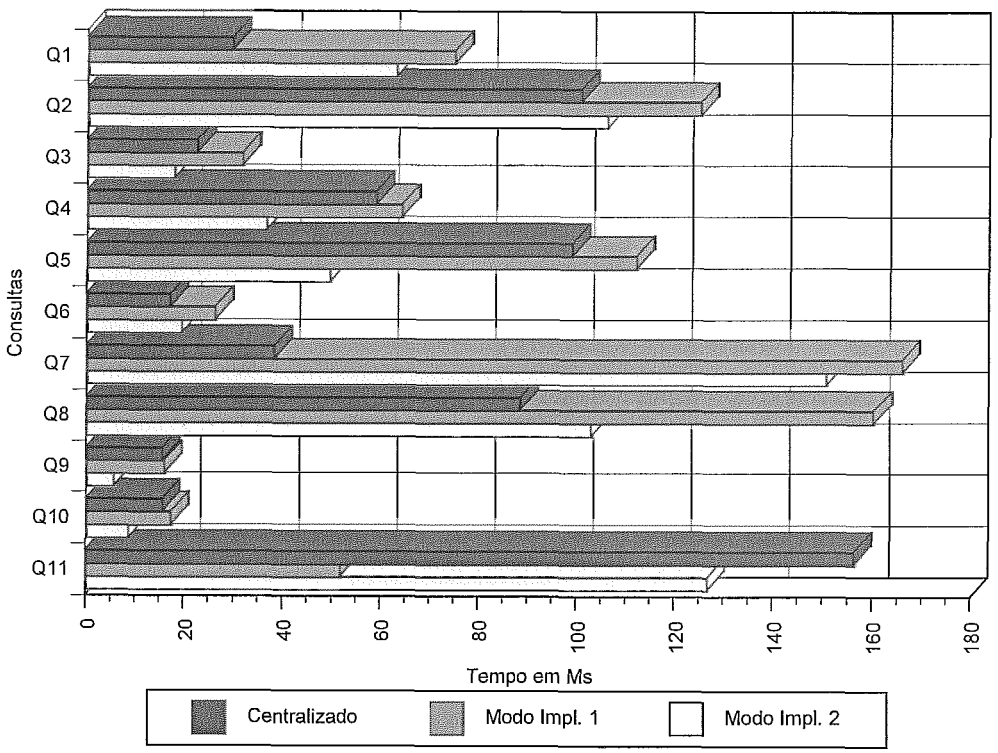


Figura 51: Gráfico de Desempenho. Base LojaHib de 20Mb.

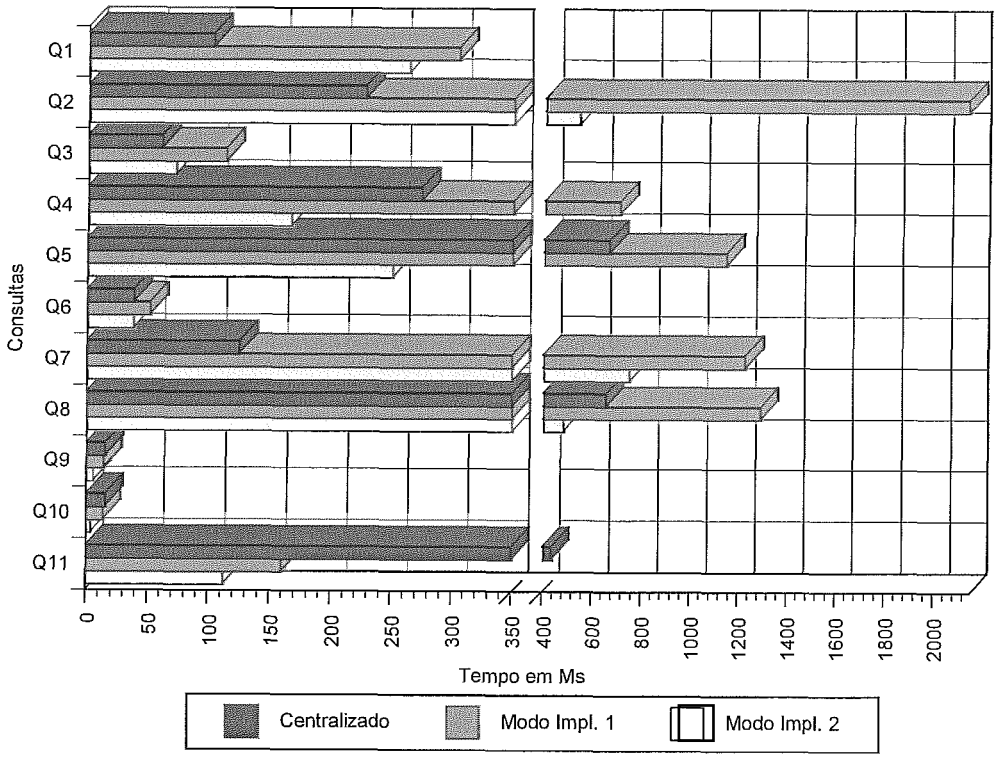


Figura 52: Gráfico de Desempenho. Base LojaHib de 100Mb.

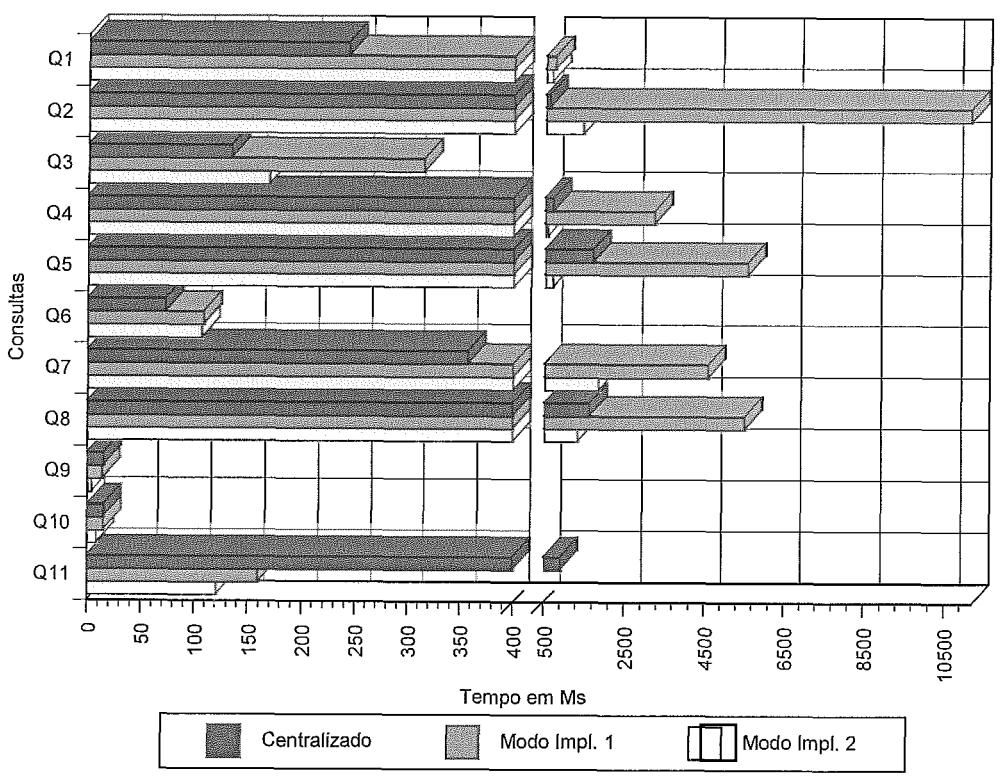


Figura 53: Gráfico de Desempenho. Base LojaHib de 250Mb.

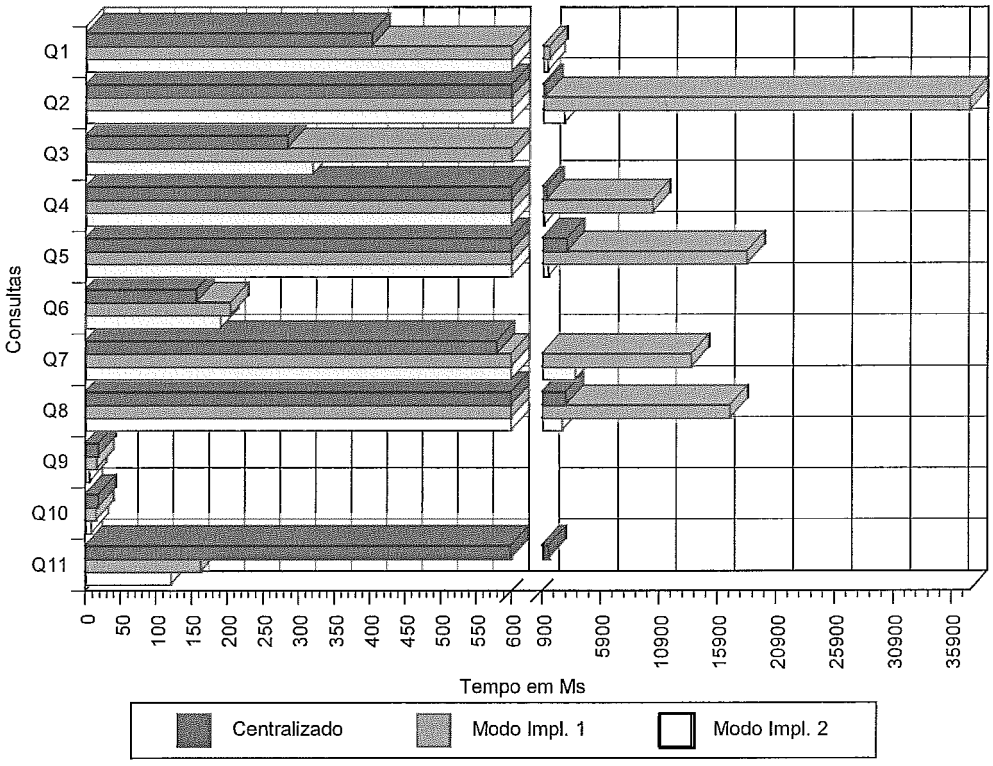


Figura 54: Gráfico de Desempenho. Base LojaHib de 500Mb.

Podemos notar que o Modo de Implementação 2 se sai bem melhor em termos de desempenho, apesar de não ganhar em todos os casos. Com a base de 5Mb, ele vence nas consultas Q_3, Q_4, Q_5 e Q_6. Essas consultas beneficiam-se do paralelismo dos fragmentos e do direcionamento para um fragmento específico. Como no Modo de Implementação 1, sempre há ganho nas consultas Q_9 e Q_{10}. A consulta Q_{11} só apresenta perda na base com 5Mb.

Conforme o tamanho da base aumenta, o resultado das consultas também aumenta, o que cada vez mais vai onerando o tempo total da consulta. Na base de 20Mb, a consulta Q_6 passa a ser equivalente à centralizada. Já na base de 100Mb, Q_3 e Q_6 passam a ser equivalentes. Com 250Mb, essas duas consultas já perdem para a base centralizada. Por fim, com 500Mb, a consulta Q_4 também passa a ser equivalente, e as demais perdem.

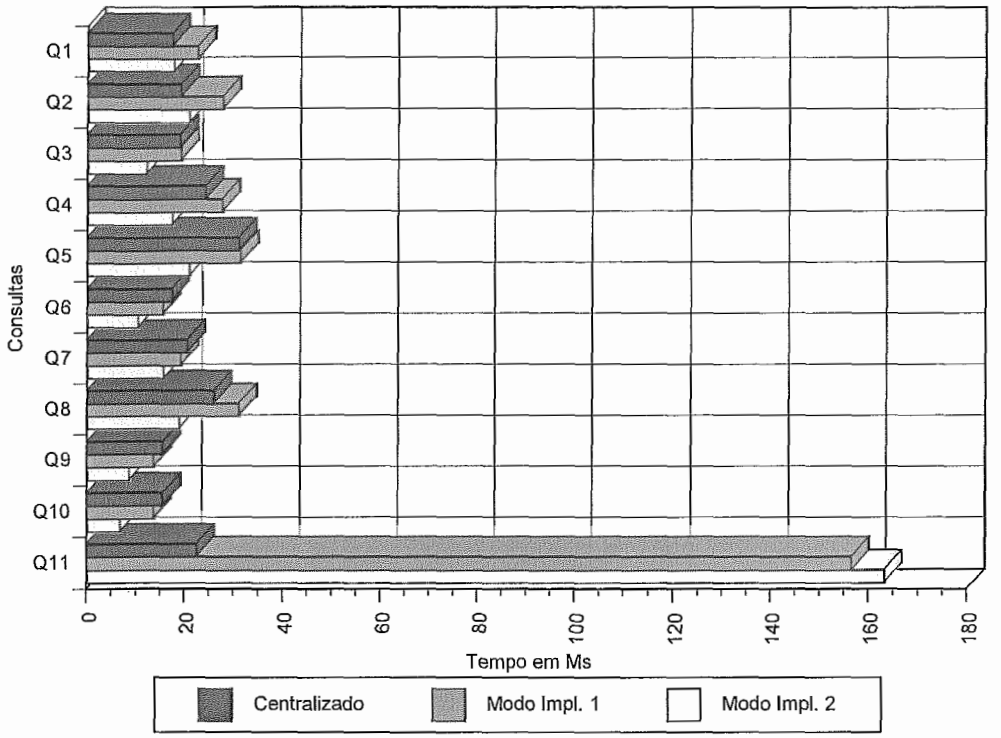


Figura 55: Gráfico de Desempenho sem Tempo Transmissão. Base LojaHib de 5Mb.

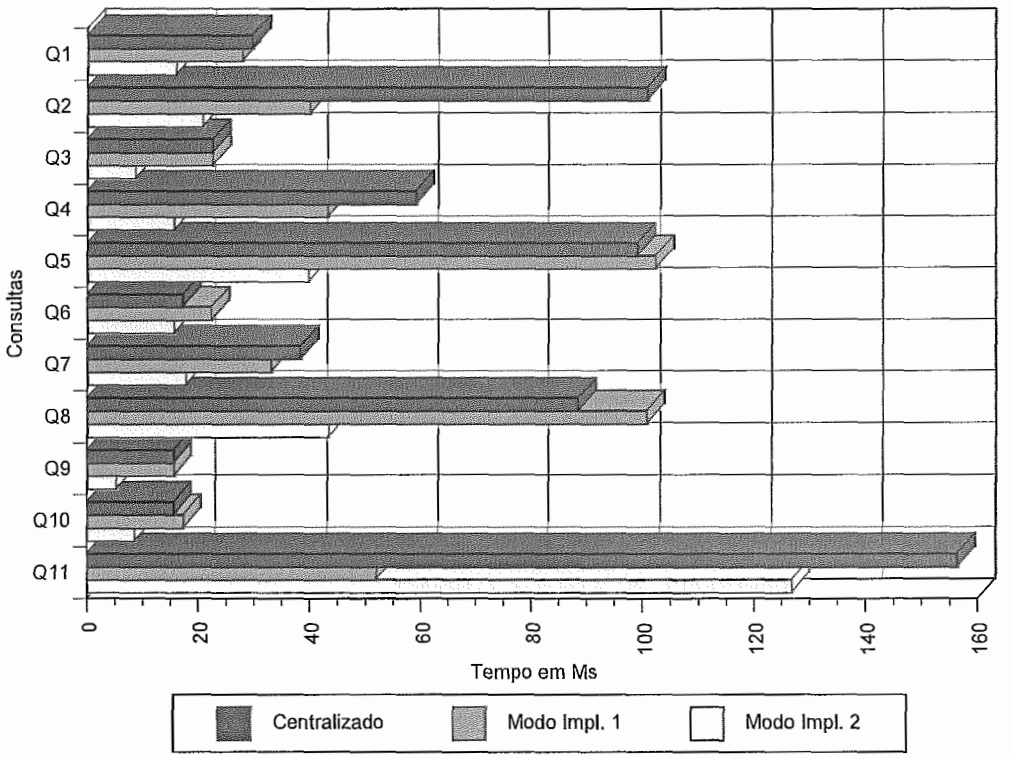


Figura 56: Gráfico de Desempenho sem Tempo Transmissão. Base LojaHib de 20Mb.

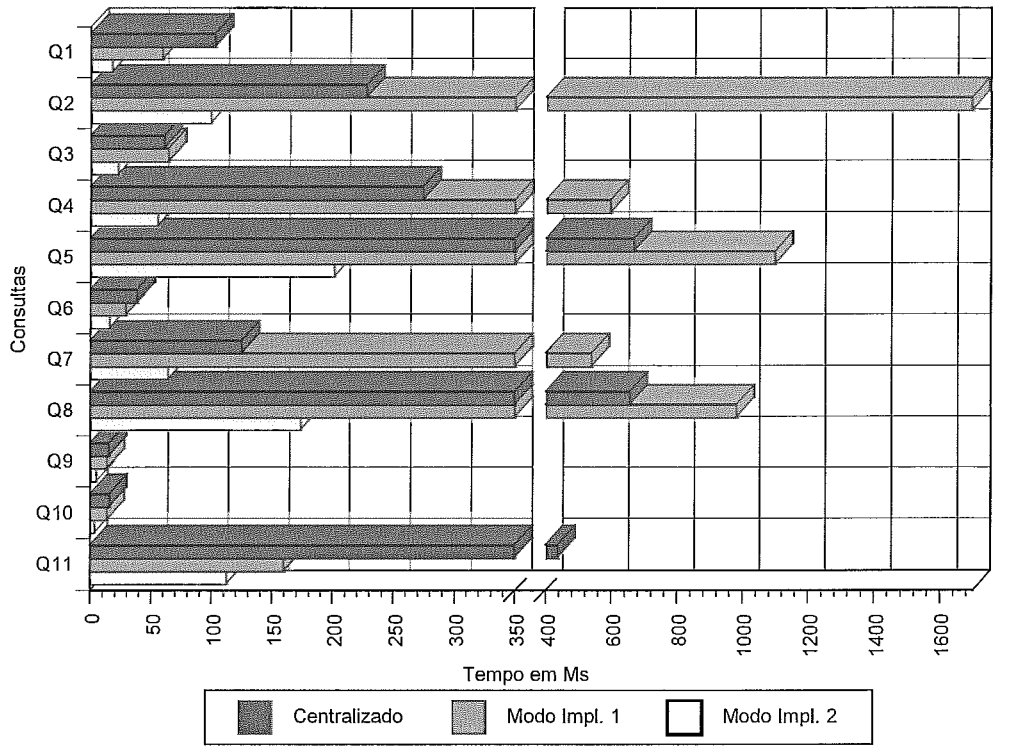


Figura 57: Gráfico de Desempenho sem Tempo Transmissão. Base LojaHib de 100Mb.

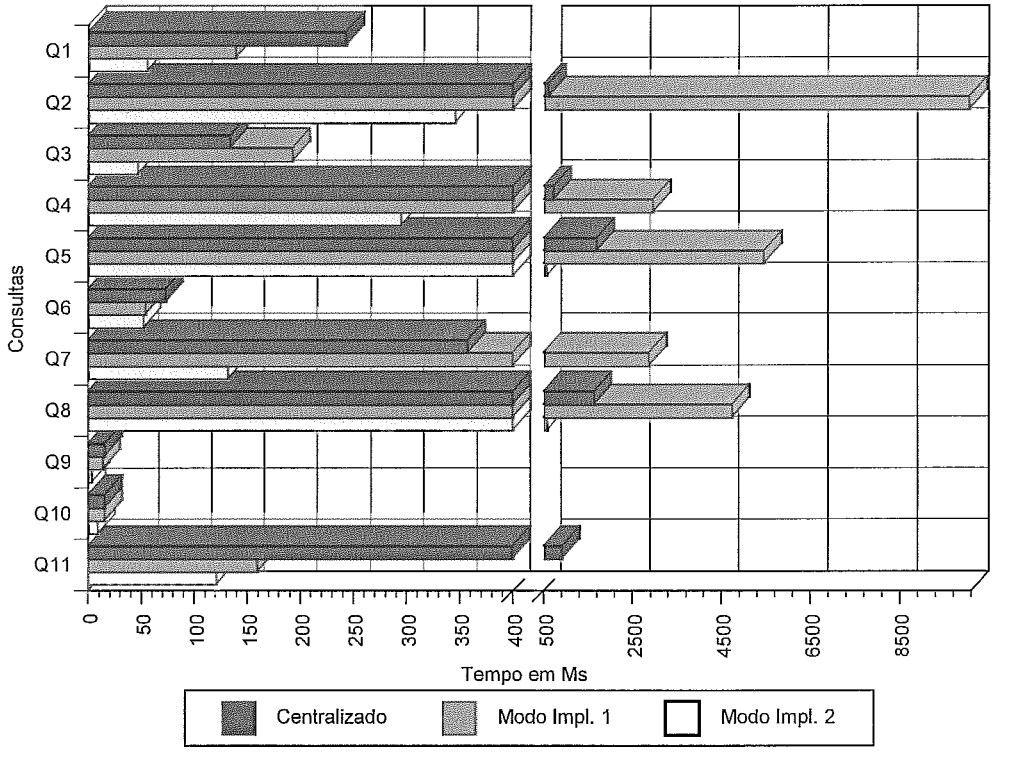


Figura 58: Gráfico de Desempenho sem Tempo Transmissão. Base LojaHib de 250Mb.

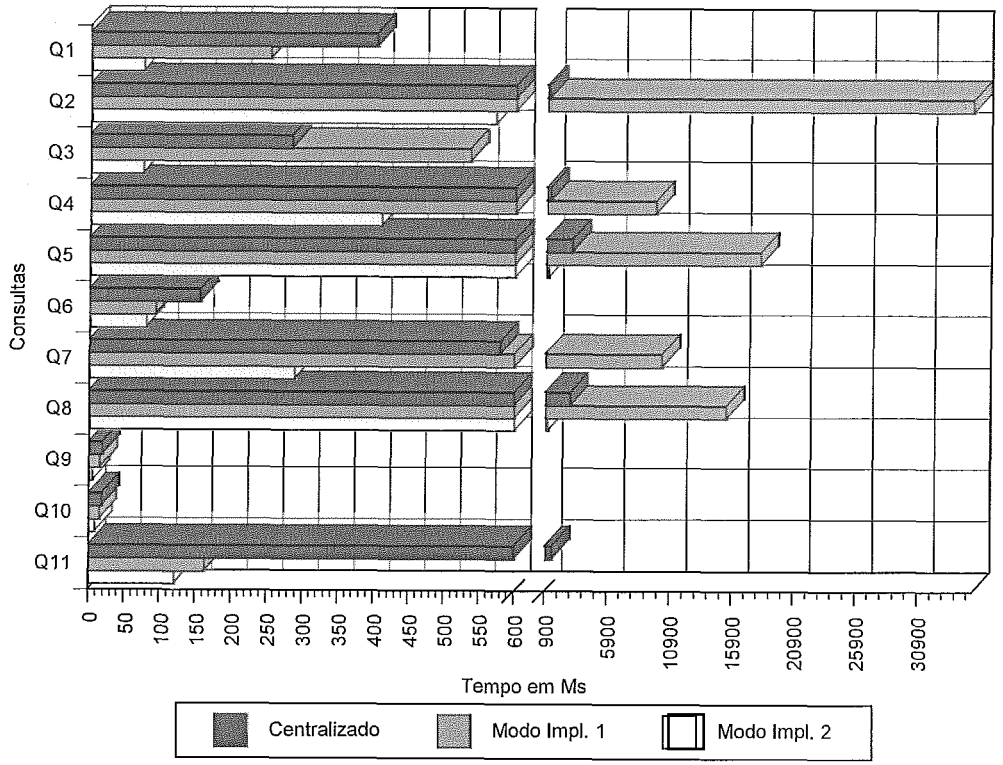


Figura 59: Gráfico de Desempenho sem Tempo Transmissão. Base LojaHib de 500Mb.

Como pudermos ver, o tempo de transmissão foi decisivo nos resultados. Sem considerar esse tempo, o Modo de Implementação 2 ganhou em todas as bases e em todas as consultas, com exceção da consulta Q_{11} para a base de 5Mb. E, ainda, o Modo de Implementação 1 se mostrou efetivo em alguns casos.

Capítulo 6 - Conclusões

Esta dissertação apresentou uma solução para a obtenção de ganhos de desempenho na execução de consultas XQuery sobre repositórios XML. Este objetivo foi atingido através do uso de técnicas de fragmentação nas bases de dados XML. Essas técnicas foram apresentadas através de uma definição formal dos diferentes tipos de fragmentos XML. Além disso, definimos critérios para a correção dessas fragmentações. Através do uso do conceito de coleção, criamos uma abstração onde as definições de fragmentos tanto se aplicam para coleções compostas de apenas um documento (SD) ou com múltiplos documentos (MD). Nossa definição apresenta uma contribuição frente às propostas existentes, uma vez que além de não restringir o tipo de coleção dos documentos, ela é baseada em uma álgebra pré-existente, o que formaliza também o mapeamento da consulta centralizada sobre os documentos fragmentados. Como vimos ao longo do texto, esses conceitos não são encontrados nos trabalhos relacionados, e são fundamentais para realizarmos a decomposição da consulta e a composição do resultado.

Os experimentos apresentados destacam o ganho de desempenho com o uso da fragmentação em vários cenários. Os ganhos foram mais expressivos principalmente na fragmentação horizontal, sobretudo nas consultas envolvendo busca em texto e agregações. A redução da execução do tempo de consulta foi obtida pelo paralelismo intra-consulta, além da execução local direcionada para um conjunto menor de dados, evitando a busca em fragmentos desnecessários. No cenário vertical, pudemos constatar que a fragmentação funcionou melhor em pequenos conjuntos de dados, inferiores a 100Mb. Com isso, pudemos observar que o custo das operações de junção ainda são elevados em XML. Já no cenário híbrido, pudemos observar as diferenças em como implementar os fragmentos horizontais a partir de uma fragmentação vertical. Além disso, vimos que compensa usar uma fragmentação híbrida se as consultas retornarem conjuntos de elementos pequenos. A maior perda no cenário híbrido foi na transmissão do documento.

A arquitetura do PartiX é genérica, e pode ser adaptada a qualquer SGBD XML que processe consultas XQuery. Essa arquitetura segue a abordagem de clusters

de banco de dados que vem apresentando excelente desempenho sobre SGBDs relacionais (LIMA, MATTOSO et al, 2004).

A partir desta dissertação, podem surgir diversos trabalhos futuros. Temos a intenção de usar o modelo de fragmentação proposto para definir uma metodologia para fragmentação de banco de dados XML. Tal metodologia pode ser utilizada para definir algoritmos para o projeto de fragmentação, e implementar ferramentas capazes de automatizar o processo de fragmentação. Uma outra linha seria um trabalho para detalhar algoritmos para a re-escrita automática de consultas para serem executadas em banco de dados fragmentados. É importante notar que, como nossa definição está baseada em uma álgebra, esta pode ser usada para auxiliar a re-escrita das consultas.

Além disso, testes com outros SGBDs XML seriam interessantes, para comparar resultados entre SGBDs. Entre eles, a utilização do próprio TIMBER (JAGADISH, AL-KHALIFA et al, 2002) seria muito interessante, já que este implementa módulos de processamento de consulta utilizando a TLC e a TAX.

Referências Bibliográficas

ABITEBOUL, S., BONIFATI, A., COBENA, G., et al, 2003, "Dynamic XML documents with distribution and replication". Em: *SIGMOD Conference 2003*, pp. 527-538

AL-KHALIFA, S., JAGADISH, H.V., 2002, "Multi-level Operator Combination in XML Query Processing". Em: *CIKM 2002*, 4-9 Novembro, 2002, McLean, Virgínia, EUA

AMER-YAHIA, S., KOTIDIS, Y., 2004, "A Web-Services Architecture for Efficient XML Data Exchange". Em: *ICDE 2004*

ANDRADE, A., RUBERG, G., BAIÃO, F., et al, 2006, "Efficiently processing XML queries over fragmented repositories with PartiX". Em: *Database Technologies for Handling XML Information on the Web in EDBT 2006*, Munich, Germany

ARENAS, M., LIBKIN, L., 2004, "A Normal Form for XML Documents.", *ACM Transactions on Database Systems*, v. 29, n. 1, pp. 195-232

BAIÃO, F., MATTOSO, M., ZAVERUCHA, G., 2004, "A Distribution Design Methodology for Object DBMS", *Distributed and Parallel Databases*, v. 16, n. 1, pp. 45-90

BARBOSA, D., MENDELZON, A., KEENLEYSIDE, J., et al, 2002, "ToXgene: a template-based data generator for XML". Em: *WebDB 2002*, pp. 621-632

BEA SYSTEMS, 2005, "BEA WebLogic Integration". Em: http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/integrate/.

BERTINO, E., CATANIA, B., WANG, W.Q., 2004, "XJoin Index: Indexing XML Data for Efficient Handling of Branching Path Expressions". Em: *ICDE 2004*, pp. 828

BONIFATI, A., MATRANGOLO, U., CUZZOCREA, A., et al, 2004, "XPath lookup queries in P2P networks". Em: *WIDM 2004*, pp. 48-55

BOSE, S., FEGARAS, L., LEVINE, D., et al, 2003, "A Query for Fragmentes XML Stream Data". Em: *9th International Conference on Data Base Programming Languages*, Postdam, Alemanha, Setembro 6-8

BOSE, S., FEGARAS, L., 2005, "XFrag: A Query Processing Framework for Fragmentes XML Data". Em: *WebDB 2005*, Junho 16-17, Baltimore, Maryland.

BREMER, J.M., GERTZ, M., 2003, "On Distributing XML Repositories". Em: *WebDB 2003*, pp. 73-78

CHEN, Y., DAVIDSON, S.B., ZHENG, Y., 2004, "BLAS: An Efficient XPath Processing System". Em: *SIGMOD Conference 2004*, pp. 47-58

ELMASRI, R., NAVATHE, S.B., 2000, *Fundamentals of Database Systems*, Addison-Wesley

EXIST DEVTEAM, 2005, "eXist: Open Source Native XML Database". Em: http://exist.sourceforge.net.

FEGARAS, L., LEVINE, D., BOSE, S., et al, 2002, "Query Processing of Streamed XML Data". Em: *CIKM 2002*, 4-9 Novembro, 2002, McLean, Virgínia, EUA

FERNÁNDEZ, M.F., SIMÉON, J., WADLER, P., 2000, "An Algebra for XML Query". Em: *FSTTCS 2000*, pp. 11-45

FIEBIG, T., HELMER, S., KANNE, C.C., et al, 2002, "Natix: A Technology Overview". Em: *Web, Web-Services, and DB Systems 2002*, pp. 12-33

FRANSICAR, F., HOUBEN, G., PAU, C., 2002, "XAL: an Algebra for XML Query Optimization". Em: *Thirteenth Australasian Database Conference*, Melbourne, Australia

FLORESCU, D., HILLERY, C., KOSSMANN, D., et al, 2004, "The BEA streaming XQuery processor", *VLDB Journal*, v. 13, pp. 294-315

GREEN, T.J., GUPTA, A., MIKLAU, G., et al, 2004, "Processing XML Streams With Deterministic Automata and Stream Indexes", *ACM Transactions on Database Systems*, v. 29, n. 4, pp. 752-788

JAGADISH, H.V., AL-KHALIFA, S., LAKSHMANAN, L.V.S., et al, 2002, "Timber: A native XML database", *VLDB Journal*, v. 11, pp. 274-291

JAGADISH, H.V., LAKSHMANAN, L.V.S., SRIVASTAVA, D., et al, 2001, "TAX: A tree algebra for XML". Em: *DBPL 2001*, pp. 149-164

LEVY, A., RAJARAMAN, A., ORDILLE, J.J., 1996, "Querying Heterogeneous Information Sources Using Source Descriptions". Em: *22th VLDB Conference*, Bombay, India

LIMA, A., MATTOSO, M., VALDURIEZ, P., 2004, "Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster". Em: *SBBD 2004*, pp. 92-105

MA, H., SCHEWE, K.-D., 2003, "Fragmentation of XML Documents". Em: *SBBD 2003*, pp. 200-214

MICROSOFT, 2005, "Microsoft SQL Server 2005". Em: http://www.microsoft.com/sql/2005/productinfo/sql2005features.mspx.

NAVATHE, S., KARLAPALEM, K., RA, M., 1995, "A mixed fragmentation methodology for initial distributed database design", *Journal of Computer and Software Engineering*, v. 3, n. 4

ORACLE, 2005, *Oracle Database 10g Release 2 XML DB Technical Overview*

ÖZSU, M.T., VALDURIEZ, P., 1999, *Principles of Distributed Database Systems*, Prentice Hall

PAPARIZOS, S., WU, Y., LAKSHMANAN, L.V.S., et al, 2004, "Tree Logical Classes for Efficient Evaluation of XQuery". Em: *SIGMOD 2004*, Junho 13-18, Paris, França

PAPARIZOS, S., JAGADISH, H.V., 2005, "Pattern tree algebras: sets or sequences?". Em: *VLDB Conference 2005*, Trondheim, Norway

PETROPOULOS, M., DEUTSCH, A., PAPAKOSTANTINOU, Y., 2003, "Query Set Specification Language(QSSL)". Em: *WebDB 2003*, San Diego, California, Junho 12-13

POTTINGER, R., LEVY, A., 2000, "A Scalable Algorithm for Answering Queries Using Views". Em: *26th VLDB Conference*, Cairo, Egito

RITTINGER, J., 2005, "Pathfinder/MonetDB: A High Performance Relational Runtime for XQuery". Em: *BTW 2005*, pp. 5-7

RIZZOLO, F., MENDELZON, A., 2001, "Indexing XML Data with ToXin". Em: *WebDB 2001*, pp. 49-54

RUBERG, N., RUBERG, G., MANOLESCU, I., 2004, "Towards cost-based optimization for data-intensive web service computations". Em: *SBBD 2004*, pp. 283-297

SCHMIDT, A.R., WAAS, F., KERSTEN, M.L., et al, 2001, *The XML Benchmark Project*

SCHÖNING, H., 2001, "Tamino - A DBMS designed for XML". Em: *ICDE 2001*, pp. 149-154

SLEEPYCAT SOFTWARE, 2005, "Berkeley DB XML". Em: http://www.sleepycat.com/products/xml.shtml.

SU, H., RUNDENSTEINER, E.A., MANI, M., 2005, "Semantic Query Optimization for Query over XML Streams". Em: *31th VLDB Conference*, Trondheim, Norway

THE APACHE SOFTWARE FOUNDATION, 2005, "Apache XIndice". Em: http://xml.apache.org/xindice/.

W3C, 2005, "World Wide Web Consortium (W3C)". Em: http://www.w3.org.

W3C, 2004, "XML Inclusions (XInclude) 1.0". Em: http://www.w3.org/TR/2004/PR-xinclude-20040930.

YAO, B., ÖZSU, M.T., KHANDELWAL, N., 2004, "XBench Benchmark and Performance Testing of XML DBMSs". Em: *ICDE 2004*, pp. 621-632

ZHANG, M., YAO, J.T., 2004, "XML Algebras for Data Mining". Em: *DMKD 2004*, pp. 209-217

ZHANG, X., PIELECH, B., RUNDESNTEINER, E.A., 2002, "Honey, I shrunk the XQuery!: an XML algebra optimization approach". Em: *WIDM 2002* , pp. 15-22