GABRIEL PITON TESSAROLLI

UM ESTUDO SOBRE A FRAGMENTAÇÃO VIRTUAL SIMPLES E O IMPACTO DO NÚMERO DE FRAGMENTOS NA EXECUÇÃO DISTRIBUÍDA DE CONSULTAS XQUERY

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Sistemas e Informação.

Orientador: Prof. D.Sc. Vanessa Braganholo Murta

Niterói

2016

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

T338 Tessarolli, Gabriel Piton
Um estudo sobre a fragmentação virtual simples e o impacto do número de fragmentos na execução distribuída de consultas XQuery / Gabriel Piton Tessarolli. – Niterói, RJ : [s.n.], 2016.
141 f.

Dissertação (Mestrado em Computação) – Universidade Federal Fluminense, 2016.
Orientador: Vanessa Braganholo Murta.

1. XML (Linguagem de marcação de documento). 2. Processamento distribuído. 3. Base de dados. I. Título.

CDD 005.133

GABRIEL PITON TESSAROLLI

UM ESTUDO SOBRE A FRAGMENTAÇÃO VIRTUAL SIMPLES E O IMPACTO DO NÚMERO DE FRAGMENTOS NA EXECUÇÃO DISTRIBUÍDA DE CONSULTAS XQUERY

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Sistemas e Informação.

Aprovada em Julho de 2016.

BANCA EXAMINADORA

Vanessa Braganholo Murta

Profa. D.Sc. Vanessa Braganholo Murta – Orientadora
Universidade Federal Fluminense

Prof. D.Sc. Alexandre Plastino de Carvalho
Universidade Federal Fluminense

Profa. D.Sc. Marta Lima de Queirós Mattoso
Universidade Federal do Rio de Janeiro

Prof. Ph.D. Victor Teixeira de Almeida
Universidade Federal Fluminense

Niterói
2016

Dedico este trabalho à minha amada esposa, que muito me apoiou nesta conquista.

AGRADECIMENTOS

Gostaria de, primeiramente, agradecer à minha família e, em especial, à minha esposa Fernanda Garcia Cordeiro Tessarolli, pela força, incentivo e compreensão demonstrados nos incontáveis períodos em que estive ausente, dedicando-me a este trabalho.

À Professora Vanessa Braganholo Murta, pelas inúmeras dicas, orientações e o apoio irrestrito durante todo o período do mestrado. Sem dúvida, foram cruciais para que eu pudesse concluir esse grande passo em minha vida.

Aos meus chefes e ex-chefes no Subdepartamento Técnico do Departamento de Controle do Espaço Aéreo (DECEA), Major-Brigadeiro Eng Fernando Cesar Pereira Santos, Brig Eng Ronaldo Yuan, Cel Eng Alexandre Henrique Nogueira, Cel Av Paulo Eduardo Albuquerque de Magella e Ten Cel Eng Alessander de Andrade Santoro, por terem sempre me apoiado nesta minha iniciativa de capacitação e desenvolvimento profissional, autorizando a minha ausência em diversas ocasiões e possibilitando que eu cursasse as disciplinas necessárias e participasse das reuniões e eventos para que eu concretizasse este trabalho.

Aos professores do Instituto de Computação da UFF, pelos ensinamentos a mim repassados durantes as disciplinas que cursei. Em especial, ao Prof. Dr. Vinod Rebello pela disponibilização de um cluster Hadoop na infraestrutura do Instituto para que eu realizasse os experimentos iniciais.

Ao Laboratório Nacional de Computação Científica (LNCC), por ter aceitado o pedido de projeto que viabilizou a utilização do cluster SunHPC e a todos os técnicos mantenedores desse cluster, que sempre me atenderam nas configurações necessárias e todos os problemas que meus experimentos sofreram.

Aos meus colegas do Grupo GEMS/DEW, que sempre contribuíram com ótimas sugestões e dicas nas reuniões em que participei, além de servirem de exemplo e estímulo durante as fases mais difíceis.

“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.” (Arthur Schopenhauer)

RESUMO

Com grandes quantidades de dados XML para analisar e consultas geralmente complexas e imprevisíveis (*ad hoc*), a velocidade de obtenção dos resultados pode determinar o sucesso ou fracasso de uma decisão em sistemas de apoio à decisão. Desta forma, cada vez mais faz-se uso de abordagens de processamento paralelo para reduzir o tempo total de execução. Nestas abordagens, a fragmentação dos dados é uma etapa crucial, pois determina como se dará o processamento paralelo. Enquanto a fragmentação física realiza a fragmentação e distribuição dos dados antes da execução das consultas, a fragmentação virtual determina em tempo de execução os subintervalos sobre os quais cada nó vai trabalhar. Assim, a fragmentação virtual torna-se mais adequada para cenários *ad hoc*, nos quais as consultas realizadas são imprevisíveis. No que se refere a consultas sobre dados XML, a única abordagem que emprega a fragmentação virtual utiliza uma distribuição estática de trabalho entre os nós ao processar somente um fragmento virtual em cada um deles. É possível que a existência de uma distribuição não uniforme dos dados (*data skew*) ocasione problemas de balanceamento de carga entre os nós, afetando assim o tempo de execução. Na fragmentação virtual aplicada sobre bases relacionais, a literatura mostra que é possível diminuir o tempo de execução de consultas ao se dividir o intervalo de dados de entrada em um número maior de fragmentos, distribuindo-os dinamicamente durante a execução. Desta forma, o principal objetivo deste trabalho é realizar esse aumento no número de fragmentos e analisar o impacto no tempo de execução de consultas com a técnica de fragmentação virtual sobre dados XML. Como a análise exige que seja utilizado algum mecanismo de distribuição dinâmica de execução, propõe-se adicionar isso à técnica de duas formas diferentes e analisar as diferenças. A abordagem VPHadoop utiliza um *framework* MapReduce para cuidar dos aspectos inerentes ao processamento paralelo, enquanto a Partix-EVP faz uso de uma implementação interna de distribuição da execução sobre uma infraestrutura MPI. Em experimentos, verificou-se que as novas abordagens distribuídas propostas são capazes de reduzir o tempo de execução de consultas XML de alto custo, se comparadas com a abordagem centralizada e com a abordagem PartiX-VP, que utiliza a distribuição estática de trabalho. Contudo, a implementação da fragmentação virtual depende da infraestrutura e arquitetura adotada, o que pode limitar a sua aplicabilidade.

Palavras-chave: XML, Fragmentação Virtual, Processamento Distribuído, Banco de dados, XQuery.

ABSTRACT

The ever-growing availability of XML data brought up the need for new XML processing techniques, so the results of this processing can be swiftly obtained and used. Decision support systems are an example of such need. With a great amount of data to analyze and queries usually complex and unpredictable (*ad hoc*), timely responses might dictate the success or failure of a decision. In this sense, parallel processing approaches have been increasingly employed to reduce the execution time. In these approaches, data partitioning is a very important step, as it sets how the parallel processing will take place. While physical partitioning breaks down the data into different fragments and distributes them before query execution, virtual partitioning determines subintervals for each node during execution, considering data replication among nodes. Thus, virtual partitioning is suitable for ad-hoc scenarios, in which queries are unknown beforehand. Considering queries over XML data, the only known approach that applies virtual partitioning uses static distribution of execution by processing only one virtual partition on each node. With that, data skew can lead to load unbalancing issues among processors during parallel execution, thus affecting total execution time. Research in virtual partitioning over relational data showed it is possible to reduce the query execution time by partitioning the input data range into more partitions and dynamically distributing them to available processors. Thus, the main objective of this work is to increase the number of partitions and analyze how it influences total execution time in virtual partitioning technique over XML data. Since varying the number of fragments requires the dynamic distribution of execution, this work proposes two approaches to realize that and compares their differences. VPHadoop employs a MapReduce framework to take care of parallel execution, while Partix-EVP implements a simple master-slave technique and uses a pure MPI infrastructure. Experimental evaluation shows that the new distributed approaches can reduce query execution time compared to both standalone and the original approach for virtual partitioning over XML, which uses static distribution. However, virtual partitioning implementation depends on the adopted architecture and infrastructure, which might limit its applicability.

Keywords: XML, XQuery, Virtual Partitioning, Distributed Processing, Database

LISTA DE ILUSTRAÇÕES

Figura 1. Arquitetura de processamento distribuído de consultas XML (FIGUEIREDO; BRAGANHOLO; MATTOSO, 2010, adaptado) 27
Figura 2. Visão Geral da arquitetura do PowerDB (AKAL; BÖHM; SCHEK, 2002, adaptado) 28
Figura 3. Exemplo de consulta e subconsultas resultantes da fragmentação virtual no modelo relacional (AKAL; BÖHM; SCHEK, 2002, adaptado) 29
Figura 4. Fragmentação virtual adaptativa em um nó processador (LIMA, 2004, adaptado) 30
Figura 5. Documento XML e representação da função XPath *position()* 32
Figura 6. Exemplo de consulta (a) e subconsulta após fragmentação virtual (b) 33
Figura 7. Consulta sobre coleção (a) e subconsulta após reescrita (b) 34
Figura 8. Análise da função XPath *position()* (SILVA; MATTOSO; BRAGANHOLO, 2013, adaptado) 35
Figura 9. Arquivo ecommerce.xml 39
Figura 10. Catálogo para o arquivo exemplo ecommerce.xml 40
Figura 11. Pseudocódigo para composição por coleção temporária 41
Figura 12. Pseudocódigo para composição por concatenação 41
Figura 13. Pseudocódigo para determinar a estratégia de composição 42
Figura 14. Processamento de resultados de consulta antigo 43
Figura 15. Processamento de resultados de consulta modificado 43
Figura 16. Diagrama de Classes simplificado da Biblioteca de Fragmentação Virtual 44
Figura 17. Pseudocódigo de exemplo de uso da biblioteca 45
Figura 18. Consulta C13 para avaliação do catálogo 48
Figura 19. Comparação entre o tempo de execução da fragmentação utilizando catálogo e consultas à base de dados, para duas consultas, em cenário SD e MD (escala de tempo logarítmica) 48
Figura 20. Comparação entre os tempos totais de execução de consultas para as estratégias de coleção temporária e concatenação 49
Figura 21. Arquitetura da proposta VPHadoop 52
Figura 22. Execução de consulta utilizando o VPHadoop 53
Figura 23. Pseudocódigo para a função *map* 55
Figura 24. Preprocessamento e execução paralela do VPHadoop 56

Figura 25. Pseudocódigo para a função *reduce* 58
Figura 26. Arquitetura da abordagem Partix-EVP..... 61
Figura 27. Esquema da base de dados do *benchmark* XMark (RODRIGUES, 2011) . 69
Figura 28. Análise do impacto do número de *splits* e *records* nos TTE para (a) 128 fragmentos e (b) 256 fragmentos..... 79
Figura 29. Análise do incremento no número de fragmentos no tempo de execução .. 80
Figura 30. TTE normalizados para as consultas de alto (a) e baixo (b) custo 82
Figura 31. Consulta C2+, variante da consulta C2 83
Figura 32. Comportamento do tempo de execução na análise do número de fragmentos 83
Figura 33. Subconsulta de C2 (a) e otimização realizada pelo SGBDX BaseX (b) 84
Figura 34. Subconsulta de C2+ (a) e otimização realizada pelo SGBDX BaseX (b) ... 85
Figura 35. Comparação entre os TTE da abordagem centralizada e da abordagem distribuída (1GB SD)..... 88
Figura 36. TTE x número de processadores para as consultas C1 (a), C5 (b), C9 (c) e C10 (d) no cenário 1GB SD 89
Figura 37. TTE x número de processadores para as consultas no cenário 1GB MD ... 92
Figura 38. Comparação entre SD e MD para abordagens distribuídas (1 GB) 93
Figura 39. TTE x Número de processadores para consultas no cenário SD com 10 GB 96
Figura 40. Comportamento do TTE com o crescimento da base..... 97
Figura 41. Curvas de utilização de processadores para C10, caso SD de 1GB, 64 processadores..... 99
Figura 42. Tempo de processamento dos processadores para C10, caso SD de 1GB, com 64 processadores..... 99
Figura 43. Curvas de utilização de processadores para C1 100
Figura 44. Tempo de processamento dos processadores para C1..... 100
Figura 45. Programa WordCount em ChuQL(KHATCHADOURIAN; CONSENS; SIMÉON, 2011, adaptado) 105
Figura 46. Preprocessamento do HadoopXML (CHOI *et al.*, 2012, adaptado) 106
Figura 47. Primeiro *job* MapReduce do processamento HadoopXML (CHOI *et al.*, 2012, adaptado) 107
Figura 48. Segundo *job* MapReduce do processamento HadoopXML (CHOI *et al.*, 2012, adaptado) 108

Figura 49. Tradução de XQuery para o modelo abstrato de MapReduce (SAUER; BÄCHLE; HÄRDER, 2013, adaptado) 109

Figura 50. Fase 1 (a) e Fase 2 (b) do carregamento de dados (LEWANDOWSKI, 2012, adaptado) 110

Figura 51. Arquitetura do PAXQuery (CAMACHO-RODRÍGUEZ *et al.*, 2015, adaptado) 112

Figura 52. Arquitetura do VxQuery (CARMAN JR. *et al.*, 2015, adaptado) 113

Figura 53. Execução do algoritmo PageRank em MapReduce (KWON *et al.*, 2012, adaptado) 115

Figura 54. Arquitetura do SkewTune (KWON *et al.*, 2012, adaptado) 116

LISTA DE TABELAS

Tabela 1. Características das consultas utilizadas na avaliação 70
Tabela 2. TTE para análise de fragmentos com o VPHadoop C1 a C6 (em s) 78
Tabela 3. TTE para análise de fragmentos com o VPHadoop C7 a C12 (em s) 78
Tabela 4. Eleição da combinação de fragmentos, *splits* e *records* 80
Tabela 5. Tempos para análise de fragmentos com o PartiX-EVP (em seg.)..... 81
Tabela 6. Tempos para análise de fragmentos com o Partix-EVP (em seg.)..... 82
Tabela 7. Resultado da análise do número de fragmentos para o PartiX-EVP 82
Tabela 8. TTE para as consultas C2 e C2+ (em seg.)..... 83
Tabela 9. TTE para a base de dados SD de 1 GB..... 87
Tabela 10. TTE para a base de dados MD de 1GB..... 87
Tabela 11. TTE para a base de dados SD de 10GB..... 95
Tabela 12. TTE para a base de dados MD de 10GB 95
Tabela 13. Características das abordagens 119

LISTA DE ABREVIATURAS E SIGLAS

AVP	*Adaptive Virtual Partitioning*
HDFS	*Hadoop Distributed File System*
MD	*Multiple Document*
OLAP	*On-line Analytical Processing*
SD	*Single Document*
SGBD	Sistema de Gerência de Banco de Dados
SGBDX	Sistema de Gerência de Banco de Dados XML Nativo
SVP	*Simple Virtual Partitioning*
TCF	Tempo de composição final do resultado
TES	Tempo da execução em paralelo das subconsultas
TFV	Tempo de Execução da Fragmentação Virtual
TTE	Tempo Total de Execução
W3C	*World Wide Web Consortium*
XML	*Extensible Markup Language*

SUMÁRIO

Capítulo 1 – Introdução 17
1.1 Motivação 17
1.2 Objetivos 20
1.3 Organização 22
Capítulo 2 – A Fragmentação Virtual de Bases de Dados XML 24
2.1 Introdução 24
2.2 Fragmentação Física 25
2.3 Fragmentação Virtual 28
2.3.1 Modelo Relacional 28
2.3.2 Modelo XML 31
2.4 Considerações Finais 36
Capítulo 3 – Melhorias na Técnica de Fragmentação Virtual 38
3.1 Descrição das Melhorias e Correções 38
3.1.1 Catálogo da Base de Dados 38
3.1.2 Estratégia de Composição do Resultado Final 40
3.1.3 Correções na Implementação 42
3.2 A Biblioteca de Fragmentação Virtual Simples 43
3.3 Avaliação Experimental 45
3.3.1 Implementação e Infraestrutura 45
3.3.2 Uso do Catálogo da Base de Dados 47
3.3.3 Estratégia de Composição do Resultado Final 49
3.4 Considerações Finais 50
Capítulo 4 – A Abordagem VPHadoop 51
4.1 Arquitetura 51
4.2 Funcionamento 52
4.2.1 Preprocessamento 53

4.2.2 Processamento Paralelo 54
4.2.3 Composição do Resultado Final 58
4.3 Considerações Finais 58
Capítulo 5 – A Abordagem Partix-EVP 60
5.1 Arquitetura 60
5.2 Funcionamento 62
5.2.1 Preprocessamento 63
5.2.2 Processamento Paralelo de Subconsultas 63
5.2.3 Composição do Resultado Final 64
5.3 Considerações Finais 65
Capítulo 6 – Avaliação Experimental 66
6.1 Ambiente de Execução 66
6.2 Base de Dados e Consultas 68
6.3 Protótipos 70
6.3.1 VPHadoop 71
6.3.2 PartiX-EVP 73
6.3.3 Partix-VP 73
6.4 Metodologia 74
6.5 Análise do Número de Fragmentos 76
6.5.1 VPHadoop 76
6.5.2 Partix-EVP 81
6.5.3 Consultas de Alto Custo 82
6.6 Comparação Entre Abordagens 85
6.7 Distribuição de Carga 97
6.8 Considerações Finais 101
Capítulo 7 – Trabalhos Relacionados 103
7.1 Trabalhos de Base 103

7.2 Processamento de Consultas Distribuídas Sobre Documentos XML com MapReduce 104
7.2.1 ChuQL 104
7.2.2 HadoopXML 106
7.2.3 BrackitMR 108
7.2.4 BaseX MapReduce 109
7.2.5 MRParBox 111
7.3 PAXQuery 111
7.4 VxQuery 113
7.5 Outras Melhorias Para o Processamento MapReduce 114
7.5.1 SkewTune 114
7.5.2 FP-Hadoop 116
7.6 Considerações Finais 118
Capítulo 8 – Conclusões e Trabalhos Futuros 121
8.1 Considerações Finais 121
8.2 Limitações 123
8.3 Trabalhos Futuros 124
ANEXO A – Consultas XQuery Utilizadas nos Experimentos 130
ANEXO B – Arquivo de Configuração do Protótipo VPHADOOP 132
ANEXO C – Gráficos das execuções para a Base de Dados SD de 1 GB 134
ANEXO D – Gráficos das execuções para a Base de Dados MD de 1 GB 136
ANEXO E – Gráficos das execuções para a Base de Dados SD de 10 GB 138
ANEXO F – Gráficos das execuções para a Base de Dados MD de 10 GB 140

CAPÍTULO 1 – INTRODUÇÃO

1.1 MOTIVAÇÃO

Com a grande quantidade de dados em formato XML disponível nos dias atuais (como exemplos, tem-se a base DBLP[1] com um arquivo XML de 1.72 GB e a base do Wikipedia[2] com uma coleção de documentos XML de 49 GB), processar e executar consultas XQuery se tornou uma atividade bastante custosa no que se refere ao tempo necessário para a obtenção dos resultados. Muito embora existam situações em que o tempo de execução não é fator crítico (processamento *batch* de arquivos), para cenários de apoio à tomada de decisão baseado na análise de grandes massas de dados, a obtenção antecipada de respostas às consultas pode proporcionar diversos benefícios a uma empresa (SHIM *et al.*, 2002).

Apesar da evolução dos sistemas de gerência de banco de dados (SGBD) e dos seus mecanismos internos que visam a otimizar a execução de consultas, a sua capacidade de execução de consultas é geralmente limitada pelos recursos (CPU, memória e espaço de armazenamento) disponíveis na infraestrutura em que está instalado. Portanto, é inevitável que, ao lidar com grandes quantidades de dados, estes sistemas apresentem um longo tempo de execução para consultas complexas, ou até mesmo não consigam chegar ao resultado final, devido a limitações de memória ou espaço de armazenamento.

Com o objetivo de melhorar o desempenho na execução de consultas, várias abordagens propõem o emprego do processamento paralelo (AKAL; BÖHM; SCHEK, 2002; ANDRADE *et al.*, 2005; FADIKA; HEAD; GOVINDARAJU, 2009; LEE *et al.*, 2012; RODRIGUES; BRAGANHOLO; MATTOSO, 2011). Nestas abordagens, a execução da consulta é dividida e distribuída entre vários processadores por meio de uma rede de dados, e o resultado final é posteriormente consolidado a partir dos resultados parciais gerados por cada processador. Embora eficaz, o processamento distribuído de consultas sobre documentos XML não é trivial. O formato XML possui características estruturais e de ordenação inerentes em sua definição, o que dificulta a fragmentação de documentos que o utilizam e, por consequência, a distribuição do processamento (BRAGANHOLO; MATTOSO, 2014).

Assim como nas abordagens que atuam sobre dados relacionais, as abordagens de execução distribuída de consultas sobre documentos XML necessitam, de alguma forma, dividir o esforço de execução entre nós processadores, geralmente interconectados por uma

[1] http://dblp.uni-trier.de/

[2] https://en.wikipedia.org/wiki/Wikipedia:Database_download

rede de computadores. Esta divisão de esforço de execução geralmente está associada à fragmentação do conjunto total de dados a serem processados. Na literatura, esta fragmentação se apresenta de duas formas distintas, denominadas fragmentação física e fragmentação virtual (BRAGANHOLO; MATTOSO, 2014).

Na fragmentação física, o projetista do banco de dados deve determinar como o conjunto total dos dados será fragmentado e armazenado no sistema distribuído (ÖZSU; VALDURIEZ, 2011). Este sistema é geralmente composto de várias instâncias de SGBD, sendo que cada instância armazena um subconjunto (fragmento) dos dados. As consultas recebidas são analisadas e subconsultas específicas para cada fragmento são criadas, baseadas em seus conteúdos. É possível também que alguns fragmentos não possuam dados relevantes à consulta original, e não precisem ser envolvidos na execução. Ao final, é preciso coletar os resultados das subconsultas e compilar o resultado final.

É fato que a fragmentação física causa um grande impacto na execução das consultas (ÖZSU; VALDURIEZ, 2011). Este impacto pode ser positivo ou negativo. Se o projeto de fragmentação foi realizado considerando-se as consultas mais frequentes que são realizadas no sistema e a afinidade dos atributos nelas envolvidos, é possível reduzir drasticamente os seus tempos de resposta. Contudo, consultas que não façam parte deste conjunto de consultas frequentes podem não obter qualquer benefício da fragmentação física, pois podem requerer muitas operações síncronas de junção (no caso de fragmentação vertical) e união (no caso de fragmentação horizontal) para gerar o resultado final a partir dos fragmentos, o que pode impactar negativamente o tempo de execução. Em cenários *ad hoc*, não se sabe de antemão quais consultas serão realizadas, e elas podem variar com grande frequência. Isto demandaria modificações constantes no projeto de fragmentação física e, por consequência, na distribuição dos dados entre os nós. Logo, conclui-se que a fragmentação física não é adequada para estes cenários.

A fragmentação virtual (MATTOSO, 2009), por sua vez, é uma técnica que se adequa às peculiaridades dos cenários *ad hoc*. Assim como na fragmentação física, cada nó que compõe a solução também trabalha com uma instância exclusiva de SGBD. Contudo, nesta abordagem uma instância de SGBD contém uma réplica completa de todo o conjunto de dados, e não somente um fragmento. Um sistema que empregue a fragmentação virtual recebe a consulta a ser executada, analisa-a e então determina o intervalo de dados sobre o qual cada nó processador deverá trabalhar. Este intervalo de dados é então explicitado em subconsultas criadas a partir da consulta original, com o uso da reescrita de consulta (*query rewriting*), sendo que o número de subconsultas geradas é igual ao número de nós de processamento

envolvidos na execução. O sistema distribui estas subconsultas para execução nos diversos nós de processamento disponíveis para o trabalho. Ao término, o sistema coleta todos os resultados parciais e gera o resultado final da consulta de entrada. Como a fragmentação ocorre durante a execução da consulta e a técnica considera a replicação completa de dados, a fragmentação virtual é aplicável às mais variadas consultas, sem qualquer modificação. Trabalhos anteriores demonstraram que esta técnica apresenta grandes ganhos em desempenho durante a execução de consultas a tabelas relacionais (MATTOSO, 2009). Até o momento, a única abordagem conhecida que aplica a fragmentação virtual sobre dados XML é a Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), que usa instâncias de um SGBD nativo XML (SGBDX) para distribuir a execução de consultas XQuery, e uma infraestrutura de processamento paralelo para coordenar a execução e a montagem do resultado final.

Tanto as abordagens baseadas em técnicas de fragmentação física quanto a abordagem baseada na técnica de fragmentação virtual podem apresentar problemas de desbalanceamento de carga durante o processamento de consultas. Isto se deve à possível presença de uma distribuição não uniforme dos dados (também conhecida como *data skew*), e a falta de um mecanismo de balanceamento de carga nos *frameworks* de distribuição de processamento empregados nas abordagens (LIMA, 2004). No caso da abordagem Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), que emprega a fragmentação virtual simples (*Simple Virtual Partitioning - SVP*) sobre dados XML, a alocação da execução de cada fragmento é feita de forma estática: cada nó processador recebe uma subconsulta que considera somente um subintervalo específico. Caso um subintervalo, no contexto da consulta, exija um esforço maior de processamento, o desbalanceamento de carga é inevitável. Trabalhos anteriores (LIMA; MATTOSO; VALDURIEZ, 2010; SILVA, 2015) apontam que a diminuição do tamanho dos fragmentos considerados em cada subconsulta e o emprego de mecanismos de distribuição dinâmica destas subconsultas para execução são capazes de mitigar este problema, resultando na diminuição do tempo de execução total da consulta.

Existem técnicas de processamento distribuído, como o MapReduce (DEAN; GHEMAWAT, 2008), que foram projetadas para prover automaticamente o balanceamento de carga durante o processamento dos dados, enquanto oferecem um algo grau de escalabilidade. O processamento de dados XML utilizando MapReduce vem sendo estudado nos últimos anos (DAMIGOS; GERGATSOULIS; PLITSOS, 2014; FADIKA; HEAD; GOVINDARAJU, 2009; LEWANDOWSKI, 2012; SAUER; BÄCHLE; HÄRDER, 2013), o que demonstra a aplicabilidade do modelo para este cenário. Contudo, as abordagens

propostas geralmente precisam que todo o conjunto de dados de entrada seja relido a cada execução de consulta, o que pode não ser desejável em cenários que reutilizam os dados para realizar várias consultas diferentes, como em sistemas de apoio à tomada de decisão. Os avanços das pesquisas em sistemas de gerência de banco de dados não são aproveitados quando a abordagem precisa ler todos os dados a cada execução de consulta.

1.2 OBJETIVOS

Considerando os resultados obtidos com o emprego de um número maior de fragmentos no mundo relacional (LIMA; MATTOSO; VALDURIEZ, 2010), coloca-se então a primeira questão de pesquisa deste trabalho: como a variação do número de fragmentos impacta a execução de consultas utilizando a técnica de fragmentação virtual? É sobre esta questão que se define o principal objetivo deste trabalho, que é estudar o impacto da variação do número de fragmentos no tempo de execução de consultas XQuery utilizando a fragmentação virtual. É esperado que, em decorrência do maior número de subconsultas (fragmentos), o subintervalo considerado em cada uma delas seja reduzido, o que, por sua vez, faz com que intervalos de dados que demandam um esforço computacional maior sejam divididos. Além disso, a distribuição dinâmica das subconsultas permite que nós que tenham recebido subconsultas menos custosas possam executar mais subconsultas enquanto outros nós executam subconsultas mais custosas, possibilitando, assim, uma possível redução no tempo total de execução. Contudo, a utilização de um número maior de fragmentos acarreta o aumento de processamento extra exigido para distribuir dinamicamente a execução e posteriormente compor o resultado final.

Para que este estudo seja possível, é preciso incorporar a distribuição dinâmica de trabalho na técnica de fragmentação virtual original, proposta por Rodrigues, Braganholo e Mattoso (2011), permitindo que ela gere um número maior de subconsultas a partir da consulta original, e que cada nó processador execute mais de uma subconsulta no decorrer da execução. Para isto, surge então outra questão de pesquisa: é melhor utilizar a distribuição de um *framework* específico e deixá-la fora da abordagem, ou implementar um mecanismo próprio de distribuição? Para analisar esta diferença, este trabalho propõe duas novas abordagens, que utilizam diferentes mecanismos para a distribuição da execução. A primeira abordagem proposta é a VPHadoop, uma abordagem para a execução distribuída de consultas XQuery que integra à técnica proposta o modelo MapReduce e instâncias de um sistema de gerência de banco de dados XML nativo. A segunda abordagem proposta é o Partix-EVP, que adiciona ao Partix-VP original (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) as

modificações propostas por este trabalho e um algoritmo para a distribuição dinâmica da execução das subconsultas. No estudo, além da análise da variação do número de fragmentos, compara-se também os tempos obtidos por estas novas abordagens com aqueles obtidos pela abordagem centralizada e pela abordagem Partix-VP, para verificar se houve ganhos com a proposta.

O VPHadoop utiliza a técnica de SVP aprimorada por este trabalho para preprocessar a consulta do usuário e a partir dela derivar subconsultas (e com elas, intervalos de dados). As subconsultas são distribuídas para execução nos vários nós de um cluster MapReduce (DEAN; GHEMAWAT, 2008). Estas subconsultas constituem a fase *Map* do processamento pelo modelo MapReduce, e são executadas nas instâncias SGBDX locais durante as tarefas *map* criadas pelo *framework*. As tarefas *map* produzem os resultados parciais que alimentam a próxima fase do modelo: a *Reduce*. Na abordagem VPHadoop, esta fase consiste de uma única tarefa do tipo *reduce*, que recupera os resultados parciais, aplica os operadores de agregação e de ordenação apropriados com o auxílio da instância local do SGBDX e de acordo com a consulta de entrada, e produz o resultado final. Alinhada à filosofia MapReduce de utilizar muitas tarefas pequenas para o processamento, o VPHadoop não limita o número de fragmentos gerados pela técnica de SVP ao número de nós processadores do cluster. Ao invés disso, ele produz um número maior de fragmentos, de menor tamanho, que são distribuídos automaticamente conforme a disponibilidade dos nós no cluster. Com isso, espera-se minimizar o efeito negativo no processamento causado pela distribuição não uniforme dos dados e, portanto, melhorar o balanceamento de carga entre os nós processadores. Utilizar a infraestrutura de um *framework* MapReduce elimina a necessidade de se implementar um serviço de distribuição da execução, pois este já está incluído. Contudo, o processamento extra e a utilização de recursos computacionais utilizados pelo *framework* para realizar este trabalho podem afetar a execução.

A abordagem Partix-EVP, por sua vez, não emprega nenhum *framework* e contém a sua própria solução de distribuição das execuções de subconsultas. Esta abordagem é uma extensão da abordagem Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) e também faz uso da infraestrutura MPI de um cluster de computadores para realizar a distribuição do trabalho. Contudo, no Partix-EVP, o nó mediador também tem a responsabilidade de receber sinalizações dos nós trabalhadores e enviar novas subconsultas àqueles nós para serem processadas, até que todas as subconsultas criadas se acabem.

A avaliação experimental realizada mostra que consultas de alto custo se beneficiam da utilização de um número maior de fragmentos. Contudo, as consultas de baixo custo

podem sofrer queda no desempenho com o aumento da fragmentação. Ainda, apesar de eliminar a necessidade de se implementar um serviço de distribuição e adicionar à abordagem mecanismos automáticos de escalabilidade e tolerância a falhas, o emprego de um *framework* MapReduce também pode acarretar um acréscimo significativo no processamento das consultas, o que se observa quando se compara os tempos totais de execução obtidos com a abordagem VPHadoop com os tempos obtidos a partir das abordagens Partix-VP e Partix-EVP.

Finalmente, é também objetivo deste trabalho aprimorar a técnica de fragmentação virtual proposta anteriormente (RODRIGUES; BRAGANHOLO; MATTOSO, 2011). Desta forma, este trabalho propõe estender esta técnica com: (1) a utilização de um catálogo da base de dados durante a execução da técnica; e (2) a determinação *ad hoc* da estratégia de composição do resultado final, baseado na consulta de entrada. A utilização do catálogo pode reduzir em até 99% o tempo necessário para calcular as cardinalidades de elementos, determinar o atributo de particionamento e gerar as subconsultas. No que se refere a estratégias de composição do resultado final, para algumas consultas é possível utilizar estratégias simples como a concatenação dos resultados parciais, o que também reduz o tempo de execução. Propõe-se, portanto, utilizar heurísticas simples para se determinar a estratégia a ser adotada, baseando-se na consulta de entrada. A avaliação experimental mostrou que os ganhos obtidos com essa mudança de estratégia podem chegar a 78%.

1.3 ORGANIZAÇÃO

Os capítulos que compõem esta dissertação estão organizados como segue.

O Capítulo 2 apresenta a técnica de fragmentação virtual para a execução distribuída de consultas, focando na abordagem proposta para o modelo XML e consultas XQuery. Abstraem-se os mecanismos de infraestrutura para a distribuição da execução, e foca-se na análise e considerações desta técnica, que possibilita a divisão da tarefa de execução de uma consulta XQuery em muitas subtarefas independentes, que podem ser executadas em paralelo.

O Capítulo 3, por sua vez, descreve algumas melhorias à técnica de fragmentação virtual simples implementadas por este trabalho. Consta neste Capítulo também a avaliação experimental destas melhorias.

O Capítulo 4 detalha a abordagem VPHadoop, proposta por este trabalho, descrevendo a integração da fragmentação virtual com o modelo MapReduce.

A abordagem Partix-EVP é apresentada no Capítulo 5, com detalhes sobre a sua arquitetura e mecanismos para distribuição das subconsultas.

O Capítulo 6 descreve a validação experimental das duas abordagens propostas. A metodologia empregada é apresentada, assim como o ambiente utilizado durante os experimentos. Todos os resultados obtidos são apresentados e discutidos neste Capítulo.

Os trabalhos relacionados são discutidos no Capítulo 7, onde são também evidenciadas as diferenças quanto à abordagem proposta por este trabalho.

Finalmente, a conclusão e discussões de trabalhos futuros são apresentadas no Capítulo 8.

CAPÍTULO 2 – A FRAGMENTAÇÃO VIRTUAL DE BASES DE DADOS XML

2.1 INTRODUÇÃO

A quantidade de dados armazenada e disponibilizada em formatos digitais tem crescido muito nos últimos anos (MAYER-SCHNBERGER; CUKIER, 2014). A produção dos dados é geralmente automática, resultado de milhões de operações realizadas nos mais variados sistemas de informação atualmente utilizados, desde simples anotações de agenda online, passando por postagens em redes sociais até complexas operações de transações comerciais emitidas pelos sistemas bancários (RANGANATHAN, 2011). A evolução da internet e dos meios de comunicação permitiram que a disponibilidade de grandes massas de dados também aumentasse. Aliado a isso, o custo de armazenamento também reduziu drasticamente, o que possibilitou que grandes massas de dados, antes nem registradas, agora sejam devidamente guardadas e utilizadas. Um exemplo disto são os arquivos de *log* de aplicativos e sistemas, que podem conter informações valiosas sobre o comportamento de usuários (JANSEN, 2006).

Nesse contexto, cabe dizer o formato XML vem emergindo como padrão para troca de informações entre sistemas de informação, dada a sua flexibilidade e independência de plataforma ou sistema (ABITEBOUL; BUNEMAN; SUCIU, 2000). Como exemplo, temos os dados da Base Digital Nacional de Previsões do Tempo[3], dos Estados Unidos, entre vários outros.

É claro que uma grande quantidade de dados requer um esforço computacional muito maior para processá-la. Em muitos casos, a quantidade de dados e a complexidade das consultas necessárias tornam os tempos de resposta dos atuais sistemas de gerência de banco de dados proibitivos para o emprego em cenários como o de apoio à tomada de decisão. É preciso então procurar formas de reduzir esse tempo de processamento.

A computação paralela vem sendo bastante explorada para reduzir o tempo de resposta de consultas que envolvem uma quantidade significativa de dados, tanto para bases relacionais quanto para bases nativas XML (AKAL; BÖHM; SCHEK, 2002; FIGUEIREDO; BRAGANHOLO; MATTOSO, 2010; ÖZSU; VALDURIEZ, 2011; RODRIGUES; BRAGANHOLO; MATTOSO, 2011). Para isso, foi preciso encontrar maneiras de fragmentar e distribuir a tarefa de execução de uma consulta. De uma maneira geral, as técnicas de

[3] http://graphical.weather.gov/xml/

fragmentação se dividem entre dois grupos (ÖZSU; VALDURIEZ, 2011): (i) técnicas de fragmentação física, em que as bases de dados são previamente fragmentadas e seus dados divididos entre os nós de uma rede para que a execução de consultas possa também ser dividida; (ii) técnicas de fragmentação virtual, que utilizam a replicação de dados nos nós da rede e determinam intervalos de dados (fragmentos virtuais) em tempo de execução para que cada processador trabalhe de forma paralela e independente.

A Seção 2.2 apresenta sucintamente os conceitos de fragmentação física aplicados a bases de dados XML, e como a execução da consulta ocorre nas abordagens que a utilizam. Cabe dizer que o Capítulo 7 também traz informações de como algumas abordagens MapReduce realizam a fragmentação de coleções de documentos XML, muito embora a fragmentação nestes casos geralmente não seja baseada em operações algébricas específicas, mas sim na "quebra" dos documentos XML em sua extensão (CHOI *et al.*, 2012) ou na mera divisão de arquivos de uma coleção (CARMAN JR. *et al.*, 2015; LEWANDOWSKI, 2012). Em seguida, a Seção 2.3 explica em detalhes a técnica de fragmentação virtual de bases de dados XML, que é a técnica aplicada na abordagem proposta por esse trabalho.

2.2 FRAGMENTAÇÃO FÍSICA

As técnicas de fragmentação física são as mais encontradas na literatura quando se trata de realizar a execução distribuída de consultas sobre bases XML. Braganholo e Mattoso (2014) realizaram uma extensa revisão sobre os trabalhos que aplicam essa técnica, apresentando as definições formais de Andrade *et al.* (2006) e classificando os trabalhos da área segundo os critérios definidos por Özsu e Valduriez (2011). De uma maneira geral, estas técnicas "quebram" as bases de dados XML aplicando sobre elas operações de seleção e/ou projeção, de forma a definir fragmentos que possam ser armazenados de forma independente, cada um em um nó de uma rede. Obviamente, estas operações sobre as bases não são escolhidas a esmo, mas sim de forma criteriosa durante o projeto de fragmentação (ÖZSU; VALDURIEZ, 2011), que analisa o conjunto das consultas mais frequentes realizadas na base e determina qual tipo de fragmentação beneficiaria a execução destas consultas. Percebe-se então que o projeto de fragmentação é uma das principais etapas nas técnicas de fragmentação física. Entretanto, ele pode também ser considerado a principal limitação dessas técnicas. Vejamos o porquê.

Como já mencionado, o projeto de fragmentação geralmente considera as consultas mais frequentes realizadas na base, quais elementos são recuperados durante a execução destas consultas e como estes elementos são armazenados, de maneira a obter um melhor

aproveitamento da infraestrutura disponível e aumentar o desempenho na execução destas consultas. Contudo, se houver alterações no conjunto de consultas frequentes, é muito provável que o projeto de fragmentação tenha que ser refeito e, por consequência, os fragmentos deverão ser redistribuídos. Se houver alterações na estrutura (esquema) da base, o projeto de fragmentação também pode sofrer impactos, uma vez que a distribuição destes novos dados poderá ter impacto no desempenho de execução. Ainda, se houver a adição de registros de forma desproporcional entre os fragmentos, é possível que ocorra um desbalanceamento de carga durante a execução e, por consequência, um impacto no tempo de execução.

Com a base fragmentada fisicamente, o processamento das consultas geralmente passa por 4 grandes etapas:

(1) Um nó coordenador, detentor de informações sobre o projeto de fragmentação, realiza a análise da consulta, e dela deriva subconsultas específicas para cada fragmento envolvido (pode ser que nem todos os fragmentos sejam necessários para responder esta consulta);

(2) O nó coordenador envia as subconsultas criadas para execução em cada um dos nós detentores dos fragmentos envolvidos, e aguarda que todos tenham terminado a execução;

(3) Os nós detentores de fragmentos envolvidos executam as subconsultas recebidas, armazenando os resultados localmente;

(4) Após todos os nós terem terminado a execução de subconsultas, o nó coordenador requisita a cada um o resultado parcial obtido, e então realiza a combinação de todos os resultados parciais, obtendo o resultado final.

Consideremos, como exemplo, a Figura 1, que apresenta a arquitetura proposta por Figueiredo, Braganholo e Mattoso (2010) para processamento de consultas XQuery em um ambiente distribuído, com uso da fragmentação física. Nesta proposta, o nó coordenador é chamado de Mediador. As etapas 1, 2 e 4 ocorrem dentro do Mediador, mais especificamente dentro do processador de consultas, com o apoio do Catálogo, que possui informações sobre o projeto de fragmentação. A etapa 3 ocorre nos SGBDX por intermédio dos Adaptadores XQuery, que posteriormente comunicam ao Mediador o término de execução e enviam os resultados parciais. Cada instância de SGBDX mostrada possui um fragmento resultante do projeto de fragmentação.

Trabalhos como os de Andrade *et al.* (2006) e Figueiredo, Braganholo e Mattoso (2010) nos permitem concluir que o emprego da fragmentação física de documentos XML e o

emprego de ambientes de processamento paralelo possibilitam uma diminuição significativa do tempo total de execução de consultas. Em alguns casos, entretanto, o custo adicionado pela solução distribuída em relação ao tempo total de execução e o percentual de ganho obtido quando o tamanho das bases de dados é pequeno não justificam o seu emprego.

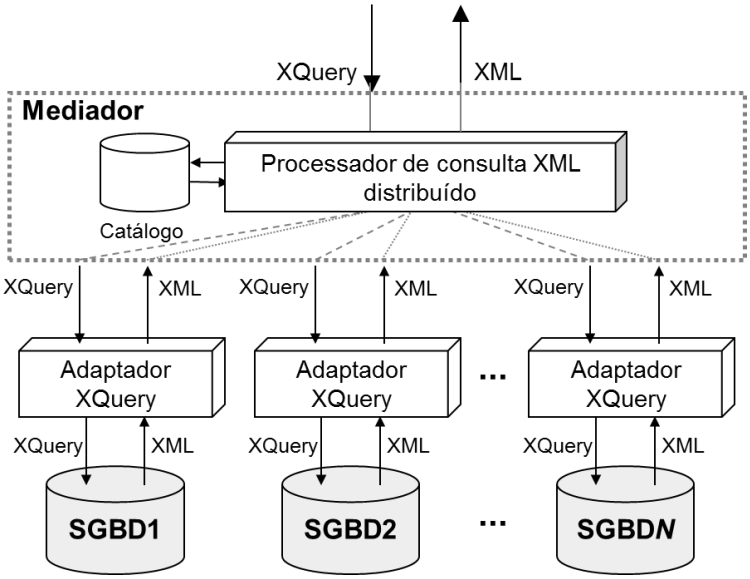


Figura 1. Arquitetura de processamento distribuído de consultas XML (FIGUEIREDO; BRAGANHOLO; MATTOSO, 2010, adaptado)

Outro grande problema também enfrentado pelas abordagens que empregam a fragmentação física é o desbalanceamento de carga durante a execução (BRAGANHOLO; MATTOSO, 2014). Conforme já discutido, as atualizações na base de dados podem invalidar o projeto de fragmentação previamente realizado, e causar impacto na execução das consultas. Além disso, experimentos na literatura mostram que consultas que não foram consideradas no projeto de fragmentação podem sofrer grande impacto no tempo de execução (KIDO; AMAGASA; KITAGAWA, 2006; PAPARIZOS *et al.*, 2004). Tais fatos limitam, por exemplo, o emprego de abordagens com fragmentação física para o processamento de consultas OLAP. O próprio projeto de fragmentação pode causar desbalanceamento devido à falta de relação com o ambiente de processamento distribuído empregado e seu dinamismo (adição ou remoção de nós). As podas (eliminação de fragmentos desnecessários) realizadas durante a avaliação das consultas para diminuir o volume de dados consultados (e assim ganhar desempenho) têm o efeito colateral de deixar nós do ambiente ociosos.

2.3 FRAGMENTAÇÃO VIRTUAL

2.3.1 MODELO RELACIONAL

A técnica de fragmentação virtual de dados foi introduzida pela primeira vez por Akal, Böhm e Schek (2002) para banco de dados relacionais, como uma proposta para melhorar o desempenho de consultas OLAP. A arquitetura proposta pelos autores está apresentada na Figura 2. Ela é formada primariamente por um cluster de bases de dados, ou seja, um conjunto de computadores comuns (nós), conectados por uma rede, e em cada computador é executado um SGBD relacional comum, contendo uma réplica completa da base de dados. Sobre este cluster, os autores propuseram uma camada de software (*middleware*) que realiza a coordenação entre as bases de dados e executa consultas que são requisitadas por aplicativos ou usuários clientes. Grande parte deste *middleware* roda em um nó da rede chamado de nó coordenador, o único nó visível pelos clientes da solução, e responsável pela recepção e distribuição das consultas recebidas aos nós de processamento. A arquitetura utiliza paralelismo inter-consulta (*inter-query*) e intra-consulta (*intra-query*) na execução, sendo que o paralelismo intra-consulta é obtido com a aplicação da fragmentação virtual.

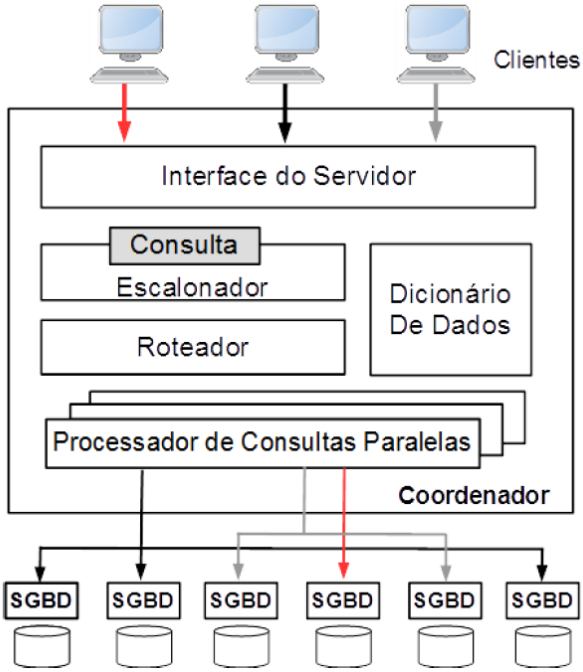


Figura 2. Visão Geral da arquitetura do PowerDB (AKAL; BÖHM; SCHEK, 2002, adaptado)

No modelo relacional, a fragmentação virtual é obtida com a adição de predicados de seleção à cláusula WHERE da consulta de entrada, conforme a Figura 3. Esses predicados definem intervalos de dados. Utilizando um atributo indexado pelo SGBD e informações

sobre sua cardinalidade, o *middleware* é capaz de dividir o intervalo de dados em partes iguais, considerando o número de processadores que irá envolver na execução, e gerar subconsultas derivadas da consulta original, para serem executadas em paralelo. No exemplo da Figura 3, este atributo (chamado de atributo de fragmentação) é o ChavePedido e seus valores variam entre 0 e 600.000, os quais foram divididos em dois fragmentos. Este tipo de fragmentação virtual é chamado de Fragmentação Virtual Simples (FVS). Para ele, o número de nós é um parâmetro de entrada e determina exatamente o número de fragmentos que são gerados pela técnica.

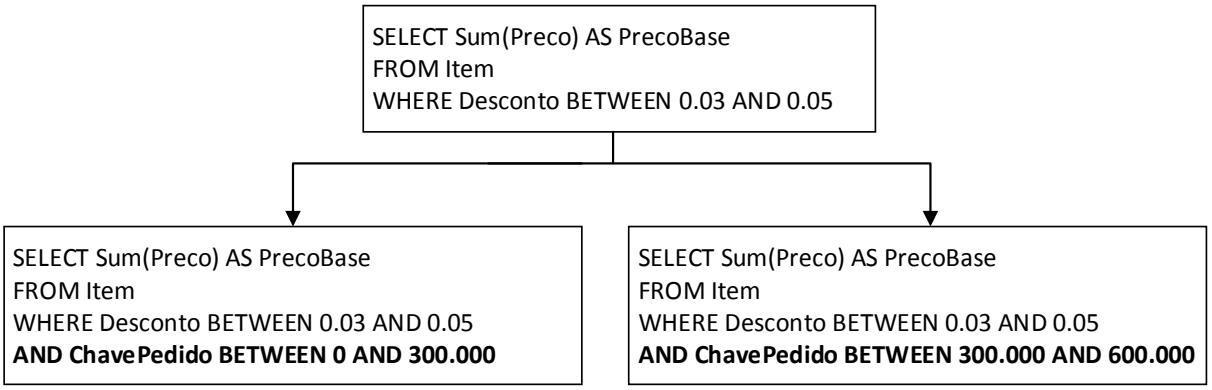


Figura 3. Exemplo de consulta e subconsultas resultantes da fragmentação virtual no modelo relacional (AKAL; BÖHM; SCHEK, 2002, adaptado)

Com a replicação total da base de dados nos nós processadores do cluster, a fragmentação virtual evita a transferência de dados que seria necessária para enviar a execução de uma subconsulta a um nó que não tivesse todos os dados suficientes. Envia-se tão somente a consulta parametrizada e os marcadores de início e fim do intervalo associado ao nó. Nota-se, também, que todos os nós processadores são elegíveis para executar qualquer um dos intervalos calculados, não limitando a execução de uma subconsulta a somente um nó, como ocorre nas técnicas de fragmentação física. Contudo, deve-se ter em mente que a replicação total da base de dados também pode apresentar desafios para garantir a consistência nos casos em que as bases são atualizadas com frequência (número de alterações), e em quantidade significativa (porcentagem de dados alterados).

Um dos problemas da FVS é que ela assume uma distribuição uniforme dos valores do atributo de fragmentação. Determinadas por intervalos de igual tamanho sobre o atributo de fragmentação, as subconsultas deveriam gerar cargas de trabalho similares para maximizar o paralelismo da solução. Contudo, em muitos casos isso não acontece, por vários motivos. Primeiro, pode não haver uma distribuição uniforme entre os intervalos gerados, e a quantidade de registros em cada um pode ser significativamente diferente. Segundo, a consulta pode possuir um predicado de seleção sobre o atributo de fragmentação. Por último,

pode haver uma correlação forte entre o atributo de fragmentação e outro predicado de seleção. Nestes cenários, a FVS não é capaz de utilizar completamente o paralelismo disponível, pois ocorre um desbalanceamento de carga entre os nós.

Como uma tentativa de minimizar o impacto do desbalanceamento de carga, a Fragmentação Virtual Adaptativa (FVA) foi proposta (LIMA, 2004; LIMA; MATTOSO; VALDURIEZ, 2010). Nesta abordagem, a técnica de FVS é empregada inicialmente para determinar o atributo de fragmentação, gerar as subconsultas iniciais e realizar a distribuição delas entre os nós do cluster. Entretanto, ao receber uma subconsulta, o nó processador não a executa diretamente. Este passo é ilustrado na Figura 4. O intervalo de dados inicial (I_i) considerado nesta subconsulta é mais uma vez fragmentado dentro do nó, utilizando o mesmo atributo de fragmentação. Gera-se um subintervalo I_{i0}, que parametriza a consulta C_i. Esta consulta é executada e, a partir do seu tempo de execução, o tamanho do intervalo é ajustado. Por exemplo, pode-se dobrar o tamanho do intervalo e verificar se o tempo de execução cresceu proporcionalmente. Caso a proporção seja menor ou igual a 1, pode-se continuar crescendo o tamanho do fragmento. Caso contrário, mantém-se o tamanho anterior ou diminui-se. A composição do resultado final segue as mesmas regras da composição da FVS.

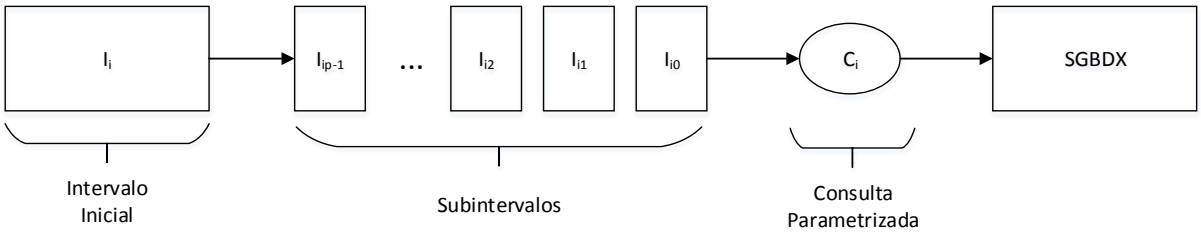


Figura 4. Fragmentação virtual adaptativa em um nó processador (LIMA, 2004, adaptado)

A criação de subintervalos dentro do nó processador por si só não é capaz de resolver o problema do desbalanceamento de carga originário do FVS. O nó processador continua a processar o mesmo intervalo que lhe foi atribuído pelo FVS, de uma forma diferente. Contudo, essa subdivisão em intervalos menores permite que o nó consiga determinar o que lhe falta processar e, se for o caso, contar com a ajuda de outros nós para terminar sua tarefa. As abordagens que empregam a FVA contam com um mecanismo que permite redistribuir subintervalos a nós processadores que fiquem ociosos, por meio de um mecanismo de oferta de ajuda.

2.3.2 MODELO XML

Rodrigues, Braganholo e Mattoso (2011) propuseram o emprego da fragmentação virtual simples para distribuir a execução de consultas XQuery sobre bases de dados XML em um ambiente distribuído, de forma similar ao realizado para o modelo relacional (AKAL; BÖHM; SCHEK, 2002). Na proposta, uma etapa de preprocessamento analisa as expressões XPath da consulta a ser executada e a reescreve de forma parametrizada, adicionando a ela filtros em elementos específicos. O preprocessamento termina com a geração de subconsultas, que consideram intervalos diferentes da base de dados XML e representam, portanto, os fragmentos da base total. Em uma etapa de processamento paralelo, as subconsultas geradas são enviadas simultaneamente pelo nó coordenador aos nós processadores para execução paralela. Na proposta, o número de subconsultas deve ser igual ao número de nós processadores. O nó coordenador aguarda o término da execução de todos os nós processadores, e então executa a etapa de composição do resultado final.

Apesar do procedimento empregado não parecer diferir muito daquele descrito para o modelo relacional, a falta de chaves no modelo XML faz com que a sua fragmentação não seja trivial. No modelo relacional, a fragmentação é obtida aplicando-se à consulta original predicados de seleção que operam sobre intervalos no domínio da chave da relação. Em bases de dados XML, apesar de esquemas XML permitirem a identificação de atributos que possuam as características de uma chave, não há qualquer regra que obrigue a sua definição no esquema. Não há nem mesmo a obrigatoriedade de um documento XML seguir um esquema específico. Logo, não há, por definição, uma caraterística de um elemento que o identifique diretamente e, portanto, não é possível a sua recuperação direta, como ocorre com uma chave em um banco relacional.

Como forma de substituir as chaves do modelo relacional, Rodrigues, Braganholo e Mattoso (2011) propuseram a utilização da função XPath *position()* (MALHOTRA *et al.*, 2010; SILVA; MATTOSO; BRAGANHOLO, 2013). Esta função retorna a posição de um elemento XML em relação ao seu elemento pai. A Figura 5 ilustra o valor de retorno em um documento XML de exemplo, em que se representa a base de dados de um *e-commerce*. Cada elemento *cliente* possui uma posição, que é relativa ao elemento pai, *clientes*. Nota-se que o elemento *pedido* também possui *position()* igual a 1, pois sua posição é relativa ao elemento pai, que é *pedidos*.

A escolha do atributo de fragmentação virtual é o primeiro passo da FVS. Para o modelo XML, esta escolha é determinada pela análise da cardinalidade dos elementos que

compõem os caminhos XPath especificados nas cláusulas *for*/*let* da consulta. Para ser um candidato a atributo de fragmentação virtual, um elemento deve ter relação 1:N com seu elemento pai, ou seja, a sua cardinalidade deve ser maior do que um, enquanto a cardinalidade do seu elemento pai deve ser exatamente um. Isso é necessário para garantir a unicidade dos elementos dentro do intervalo que será determinado. Caso existisse mais de uma ocorrência do elemento pai no documento, dois elementos iguais do mesmo nível hierárquico (elementos pai iguais) teriam também posições iguais, e isso inviabilizaria a fragmentação.

```
<ecommerce>
  <clientes>
    <cliente id="181">                                  cliente[position()=1]
      <cpf>211.122.233-44</cpf>
      <nome>Carlos Felipe Costa</nome>
      <data_nasc>1985-08-26</data_nasc>
      <e_mail>carlosfc@hotmail.com</e_mail>
    </cliente>
    <cliente id="20354">                                cliente[position()=2]
      <cpf>013.921.123-45</cpf>
      <nome>Gabriel Fagundes</nome>
      <data_nasc>1983-03-10</data_nasc>
      <e_mail>gabrielf@gmail.com</e_mail>
    </cliente>
..</clientes>
  <pedidos>
    <pedido id="12" clienteRef="181">                   pedido[position()=1]
      <preco_total>1500.00</preco_total>
      <qtde_parcelas>2</qtde_parcelas>
      <parc_rest>1</parc_rest>
      <data_comp>2015-05-12</data_comp>
    </pedido>
..</pedidos>
  <parcelas>
    <parcela id="167809" pedidoRef="12">                parcela[position()=1]
      <data_venc>2015-06-12</data_venc>
      <valor_parcela>800.00</valor_parcela>
      <mes_referencia>06/2015</mes_referencia>
      <situacao_parc>aguardando pagto</situacao_parc>
    </parcela>
..</parcelas>
</ecommerce>
```

Figura 5. Documento XML e representação da função XPath *position()*

Com o atributo de fragmentação virtual escolhido, a sua cardinalidade determina o tamanho total do intervalo considerado. Logo, tendo como parâmetro o número de nós processadores disponíveis para a execução da consulta em paralelo, é possível determinar os subintervalos que parametrizarão as subconsultas em cada um dos nós. A abordagem proposta prevê que cada nó processador execute exatamente uma subconsulta. Como um exemplo, suponhamos uma coleção composta por um único documento como o da Figura 5, mas que contenha 200.000 pedidos. Suponhamos também que se deseja executar a consulta mostrada na Figura 6(a) utilizando quatro nós de uma rede. Na etapa de escolha do atributo de fragmentação virtual, verifica-se que os elementos *pedidos* e *pedido* possuem relação 1:N, o que faz com que o elemento *pedido* seja escolhido. Sua cardinalidade, como já dito, é 200.000. Logo, os subintervalos resultantes da fragmentação virtual conterão 50.000

elementos *pedido*. Adiciona-se então predicados à consulta para produzir quatro subconsultas. A Figura 6(b) mostra a segunda subconsulta que considera os pedidos no intervalo determinado pelas posições 50.001 e 100.000. O texto destacado em negrito na Figura mostra o filtro que foi adicionado pela técnica de fragmentação virtual.

```
<resultados>
{
   for $p in doc('ecommerce.xml')/ecommerce/pedidos/pedido
   where $p/preco_total > 1000
   return
      <pedido>
         {$p/data_comp}
         {$p/preco_total}
      </pedido>
}
</resultados>
```

(a)

```
<resultados>
{
   for $p in doc('ecommerce.xml')/ecommerce/pedidos/pedido[position() >= 50001 and
                                                            position() < 100001]
   where $p/preco_total > 1000
   return
      <pedido>
         {$p/data_comp}
         {$p/preco_total}
      </pedido>
}
</resultados>
```

(b)

Figura 6. Exemplo de consulta (a) e subconsulta após fragmentação virtual (b)

O exemplo da Figura 6 não possui operações de junção e, portanto, a seleção do atributo de fragmentação virtual é direta, baseada no caminho XPath da cláusula *for/let*. Contudo, com a presença de operações de junção, é necessário analisar ambos os caminhos expressos na consulta, para determinar qual deles refere-se ao elemento primário da junção. O elemento primário sempre possui a cardinalidade menor, logo é possível determiná-lo analisando essa informação. Após isso, segue-se para a escolha do atributo de fragmentação virtual, considerando somente o caminho XPath referente ao elemento primário.

Para bases de dados compostas por uma coleção de documentos XML (bases MD – *multiple document*), a utilização direta da função *position()* para determinação dos subintervalos não é mais possível, uma vez que a coleção não possui mais elementos com cardinalidade igual a um. Rodrigues, Braganholo e Mattoso (2011) propuseram uma solução para este caso, que consiste em reescrever a consulta original, que envolve uma coleção, como uma união de todos os documentos armazenados nela. A Figura 7 ilustra este caso. Considere que a coleção *pedidos* da consulta da Figura 7(a) é composta pelos documentos ‘pedidos1.xml’ e ‘pedidos2.xml’. Nas subconsultas geradas, como a Figura 7(b) mostra, a

função *position()* é aplicada sobre a união das sequências obtidas por cláusulas *let*, uma para cada documento.

```
<resultados>
{
   for $p in collection('pedidos')/pedidos/pedido
   where $p/preco_total > 1000
   return
      <pedido>
         {$p/data_comp}
         {$p/preco_total}
      </pedido>
}
</resultados>
```

(a)

```
<resultados>
{
   let $d1 := doc('pedidos1.xml', 'pedidos')/pedidos/pedido
   let $d2 := doc('pedidos2.xml', 'pedidos')/pedidos/pedido
   for $p in ($d1 | $d2)/pedidos/pedido[position() >= 100.000
                                 and position() < 200.000]
   where $p/preco_total > 1000
   return
      <pedido>
         {$p/data_comp}
         {$p/preco_total}
      </pedido>
}
</resultados>
```

(b)

Figura 7. Consulta sobre coleção (a) e subconsulta após reescrita (b)

Após a execução das subconsultas, os resultados parciais obtidos devem ser combinados para gerar o resultado final. Contudo, esta etapa pode não ser trivial. Se a consulta de entrada faz uso de funções de agregação, como *sum(), avg()* ou *count()*, ou ainda, se a consulta determina a ordenação do resultado por algum elemento específico, deve-se garantir que estas operações ocorram sobre o resultado como um todo, e não somente sobre os resultados parciais. Para resolver esse problema, os autores propõem criar uma coleção temporária no SGBDX local ao nó coordenador, e nela adicionar todos os documentos de resultados parciais. Em seguida, a partir da consulta inicial e das funções que ela usa, deriva-se uma consulta final que é executada sobre a coleção temporária, para assim obter o resultado final. Cabe mencionar que esta etapa de composição do resultado final é bastante custosa, pois envolve: (1) transferir os resultados parciais dos nós processadores; (2) carregar todos os resultados parciais obtidos em uma coleção temporária no SGBDX; e (3) executar a consulta final. Dependendo do tamanho dos resultados parciais, o nível de influência desta etapa da execução no tempo de execução total é bastante significativo.

Para obter ganhos de desempenho na execução da consulta, é necessário que os elementos sejam indexados pela sua posição nos SGBDX, para que o processador da consulta possa acessar diretamente os elementos quando estiver processando uma subconsulta.

Varreduras completas precisam ser evitadas a todo custo, pois poderiam acarretar um desbalanceamento de carga. Silva, Mattoso e Braganholo (2013) realizaram um estudo com os principais SGBDX existentes para avaliar o comportamento deles ao utilizar a função *position()* na recuperação de elementos em diferentes posições de uma base de dados XML. Em linhas gerais, desejava-se saber se aqueles sistemas possuíam alguma indexação por posição dos elementos, pois nesse caso o tempo de acesso seria constante. A Figura 8 mostra os resultados obtidos. A linha sólida representa os resultados medidos, e a linha tracejada indica a tendência calculada, baseada nos resultados. Nota-se que os sistemas A, B, D, E apresentaram o tempo de acesso constante. Somente o sistema C apresentou um comportamento não desejado, em que o tempo de acesso cresce conforme a posição recuperada cresce, indicando que não possui indexação por posição e precisa realizar varreduras completas para recuperar o elemento. Este sistema não poderia ser empregado em uma solução que utilizasse a fragmentação virtual de bases de dados XML para execução de consultas.

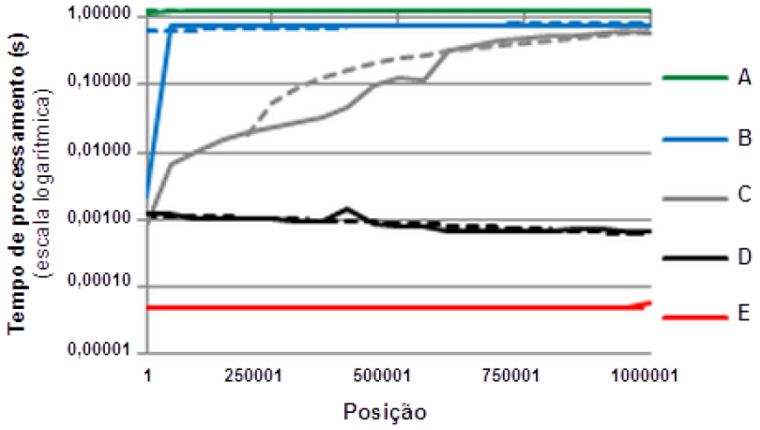


Figura 8. Análise da função XPath *position()* (SILVA; MATTOSO; BRAGANHOLO, 2013, adaptado)

Quando comparada à fragmentação física, a técnica de fragmentação virtual possui diversas vantagens. A principal delas é a sua flexibilidade: pode ser empregada para executar consultas com diferentes cargas de trabalho, em um número variado de nós processadores, e em diferentes ambientes de processamento de consultas, sem precisar de qualquer mudança. De fato, quando a fragmentação física é aplicada, o projeto de fragmentação considera a carga de trabalho das consultas mais frequentes para determinar como a base de dados deve ser fragmentada. Se esta carga mudar, seja pela mudança no conjunto de consultas frequentes, ou até mesmo pela quantidade de dados armazenada, o projeto de fragmentação deve mudar também. A fragmentação virtual não apresenta este problema pois utiliza a replicação dos

dados e, portanto, evita a custosa tarefa de movimentação dos dados durante o processamento de consultas. Contudo, utilizar a replicação de dados implica custos maiores de armazenamento, e garantir a consistência das diversas cópias é também custoso quando se tem uma atualização frequente da base de dados. Estes custos poderiam ser diminuídos com o emprego de abordagens híbridas, que combinam a fragmentação física e a virtual (LIMA *et al.*, 2009). Contudo, estas abordagens ainda não foram aplicadas a bases de dados XML.

É importante também mencionar que Silva (2015) vem conduzindo estudos para a aplicação da técnica de fragmentação virtual adaptativa (*Adaptative Virtual Partitioning*) (LIMA, 2004; LIMA; MATTOSO; VALDURIEZ, 2010) para o modelo XML, obtendo uma proposta para execução de consultas XQuery que, em tempo de execução, mitiga o problema da distribuição de carga de trabalho entre os nós do cluster.

2.4 CONSIDERAÇÕES FINAIS

Neste Capítulo apresentamos a técnica de fragmentação virtual simples, que é uma técnica que provê o paralelismo intra-consulta na execução de uma consulta, empregando uma rede de nós processadores munidos de instâncias de um SGBD e replicação total da base de dados entre os nós. Inicialmente, descrevemos a técnica proposta para as bases de dados relacionais. Em seguida, foi detalhado como esta técnica foi adaptada para bases de dados XML. Os ganhos no tempo de execução com o emprego desta técnica foram significativos, mas ainda há espaço para melhorias.

A fragmentação virtual pressupõe uma distribuição uniforme dos dados se considerado o atributo de fragmentação virtual selecionado (AKAL; BÖHM; SCHEK, 2002). Contudo, as bases de dados XML podem apresentar uma distribuição não uniforme dos dados (*data skew*) (KWON *et al.*, 2012), o que pode gerar cargas de trabalho desbalanceadas entre os nós que executam as subconsultas. Muito embora o emprego da função *position()* no predicado adicionado à consulta apresente como retorno um subintervalo de elementos numericamente determinado, os elementos acessados não necessariamente possuem tamanhos ou estruturas similares, o que faz com que um subintervalo (fragmento) possa demorar mais tempo para ser processado do que outro. Uma das maneiras de se mitigar este desbalanceamento de carga é não limitar o número de fragmentos virtuais gerados ao número de nós utilizados para execução. Cria-se um número maior de fragmentos, reduzindo-os em tamanho. Operando sobre uma quantidade menor de dados, as subconsultas apresentam menos os efeitos da distribuição não uniforme, pois um intervalo “problemático” seria dividido entre mais nós ao invés de ser executado somente em um nó. Obviamente, tal solução requer um escalonamento

das tarefas de execução das subconsultas, e isto acarreta custos adicionais na execução e um aumento na sua complexidade. Ainda, com mais fragmentos, aumenta-se também o custo de comunicação, uma vez que o escalonador precisa coordenar a execução dos nós. Uma das abordagens apresentada neste trabalho aplica o modelo MapReduce (DEAN; GHEMAWAT, 2008), mais especificamente sua implementação *open-source* (Hadoop) para escalonar os trabalhos de execução das subconsultas. A descrição da abordagem encontra-se no Capítulo 4. A outra abordagem proposta implementa este escalonamento com a utilização de um nó coordenador exclusivo. Esta abordagem está descrita no Capítulo 5.

Outra melhoria adicionada à técnica de fragmentação virtual por este trabalho é na determinação da cardinalidade dos caminhos XPath durante a seleção do atributo de fragmentação virtual. A abordagem inicial utilizava consultas específicas à base de dados para determinar esta cardinalidade, o que acarreta um aumento no tempo de execução total. Este trabalho propõe a utilização de um catálogo com informações sobre a cardinalidade dos elementos, as relações pai-filho e a composição das coleções de documentos. Por fim, durante as análises da abordagem proposta por este trabalho, verificou-se que a composição do resultado final pode ser obtida de forma mais rápida com a concatenação dos resultados parciais obtidos pelas subconsultas, desde que esta concatenação respeite a ordem determinada pelo início do subintervalo definido em cada subconsulta, e a consulta de entrada não possua sua própria cláusula de ordenação ou faça uso de funções de agregação sobre o resultado. Desta forma, este trabalho propõe também que durante a análise da consulta de entrada, aplique-se uma heurística simples a fim de determinar qual estratégia de composição final deve ser aplicada, e aplicar a concatenação sempre que possível. O Capítulo 3 apresenta descrições mais detalhadas destas melhorias, e analisa o impacto das mudanças no tempo de execução.

CAPÍTULO 3 – MELHORIAS NA TÉCNICA DE FRAGMENTAÇÃO VIRTUAL

Para viabilizar o estudo sobre o impacto da variação no número de fragmentos na técnica de fragmentação virtual, foi necessário flexibilizar a proposta original desta técnica (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), permitindo-a gerar mais fragmentos e distribuir dinamicamente a execução destes. Durante esta análise, foi possível detectar dois pontos passíveis de melhoria, além de alguns problemas na implementação. A primeira destas melhorias está localizada na etapa de geração de subconsultas, e consiste em utilizar um catálogo da base de dados para se determinar a cardinalidade de elementos e as relações entre eles nos documentos XML; eliminando assim a necessidade de consultas à base de dados nesta etapa. A segunda melhoria consiste em evitar a geração de coleções temporárias na etapa de composição do resultado final para os casos em que a consulta não possui cláusulas de ordenação, nem faz uso de funções de agregação. As correções realizadas durante a implementação referem-se a alguns casos não previstos inicialmente, bem como a diminuição do uso de memória. Este Capítulo descreve estas melhorias e correções na implementação, e por fim, apresenta e discute os ganhos obtidos com elas. Além dessas melhorias, para facilitar o emprego da técnica de fragmentação virtual simples em várias soluções, todas as funcionalidades desta técnica foram encapsuladas em uma biblioteca.

A Seção 3.1 apresenta as duas melhorias propostas em detalhes e as correções feitas. Em seguida, a Seção 3.2 detalha a composição da biblioteca de fragmentação virtual simples. A Seção 3.3 apresenta os experimentos realizados para verificar os ganhos obtidos com estas duas melhorias, e discute os resultados. Por fim, a Seção 3.4 apresenta as considerações finais.

3.1 DESCRIÇÃO DAS MELHORIAS E CORREÇÕES

3.1.1 CATÁLOGO DA BASE DE DADOS

A técnica de fragmentação virtual simples proposta por Rodrigues, Braganholo e Mattoso (2011) faz uso de execução de consultas auxiliares ao SGBDX local para: (i) determinar a cardinalidade das expressões XPath existentes na consulta e então selecionar o atributo de fragmentação virtual baseado nesta informação; (ii) recuperar o nome dos elementos ancestrais de um dado elemento, para então determinar o caminho absoluto a ele no documento; (iii) consultar quais documentos fazem parte de uma coleção na base de dados. Todas estas consultas exigem que o nó do cluster que executa o preprocessamento, conforme

descrito na Seção 2.3.2, possua também uma instância do SGBDX com uma réplica de dados. Ainda, para que o tempo de execução destas consultas auxiliares não interfira no tempo de execução total da consulta do usuário, é necessário que o SGBDX possua já indexadas as informações necessárias. Não é aceitável, por exemplo, que uma consulta de cardinalidade precise percorrer todos os elementos para contá-los. Para evitar estas consultas auxiliares, este trabalho propõe a utilização de um Catálogo externo com informações sobre a base de dados.

O Catálogo proposto por esta abordagem foi inspirado no DataGuide (GOLDMAN; WIDOM, 1997) e possui todas as três informações que são necessárias para a execução da técnica de fragmentação virtual. Além disso, é rapidamente carregado e mantido em memória para a execução de consultas. Fisicamente, ele é um arquivo XML que contém informações sobre os documentos, coleções e elementos XML da base de dados que está replicada em todas as instâncias SGBDX da solução. A Figura 9 apresenta um arquivo XML exemplo, e a Figura 10 apresenta uma parte do Catálogo gerado para ele. Para cada elemento são armazenadas informações sobre seu nome (*name*), o número de ocorrências (*count*), o caminho completo deste elemento (*path*), e qual elemento é o seu ancestral direto (*parent*).

```
<ecommerce>
  <clientes>
    <cliente>
      <cpf>211.122.233-44</cpf>
      <nome>Carlos Felipe Costa</nome>
      <data_nasc>1985-08-26</data_nasc>
      <e_mail>carlosfc@hotmail.com</e_mail>
    </cliente>
    <cliente>
      <cpf>013.921.123-45</cpf>
      <nome>Gabriel Fagundes</nome>
      <data_nasc>1983-03-10</data_nasc>
      <e_mail>gabrielf@gmail.com</e_mail>
    </cliente>
  </clientes>
  <pedidos>
    <pedido>
      <preco_total>1500.00</preco_total>
      <qtde_parcelas>2</qtde_parcelas>
      <parc_rest>1</parc_rest>
      <data_comp>2015-05-12</data_comp>
    </pedido>
  </pedidos>
  <parcelas>
    <parcela>
      <data_venc>2015-06-12</data_venc>
      <valor_parcela>800.00</valor_parcela>
      <mes_referencia>06/2015</mes_referencia>
      <situacao_parc>aguardando pagto</situacao_parc>
    </parcela>
  </parcelas>
</ecommerce>
```

Figura 9. Arquivo ecommerce.xml

A geração deste Catálogo é realizada de forma externa à solução, por um módulo específico. Este módulo lê os documentos XML que compõem a base de dados e então gera o catálogo em memória, que é posteriormente exportado para um arquivo XML. Tal técnica foi

escolhida pois as implementações de SGBDX nem sempre disponibilizam as informações necessárias para a produção deste catálogo. Por exemplo, o SGBDX Sedna (FOMICHEV; GRINEV; KUZNETSOV, 2006) não disponibiliza os nomes dos documentos que compõem uma coleção da base de dados. É importante notar que esta estratégia para o Catálogo não oferece suporte automático a atualizações da base de dados. Assim como a abordagem de execução de consulta proposta por este trabalho, o Catálogo assume que as bases de dados são somente leitura. O suporte a atualizações da base de dados pode ser adicionado em um trabalho futuro.

```
<?xml version="1.0" ?>
<catalog>
<document name="ecommerce.xml">
  <element id="0" name="ecommerce" count="1" path="/ecommerce" parent="-1"/>
  <element id="2" name="cliente" count="2" path="/ecommerce/clientes/cliente" parent="1"/>
  <element id="14" name="parcela" count="1" path="/ecommerce/parcelas/parcela" parent="13"/>
  <element id="13" name="parcelas" count="1" path="/ecommerce/parcelas" parent="0"/>
  ...
</document>
</catalog>
```

Figura 10. Catálogo para o arquivo exemplo ecommerce.xml

Com a utilização do Catálogo, foi possível reduzir o tempo de execução da fase de fragmentação em 99%. A Seção 3.3 traz mais análises acerca da utilização deste módulo.

3.1.2 ESTRATÉGIA DE COMPOSIÇÃO DO RESULTADO FINAL

A estratégia de composição do resultado final proposta por Rodrigues, Braganholo e Mattoso (2011) envolve a criação de uma coleção temporária no SGBDX composta pelos documentos de resultados parciais. Tal estratégia está ilustrada em linhas gerais na Figura 11, sob a forma de pseudocódigo. A função *CriaColeçãoSGBDX()* realiza a criação da coleção no SGBDX, deixando-a pronta para receber os documentos. Em seguida, os documentos com resultados parciais são, um a um, recuperados do sistema de arquivos distribuído da solução, e adicionados à coleção criada. A função *CriaConsultaFinal()* cria uma consulta final que leva em consideração a consulta de entrada para recuperar os resultados da coleção de forma ordenada, ou realizar as operações de agregação existentes no retorno do resultado, considerando todo o conjunto de resultados parciais. Por último, a consulta final criada é executada sobre a coleção temporária, obtendo-se assim o resultado final.

Embora esta abordagem seja aplicável para qualquer consulta de entrada, percebe-se que, para consultas que não envolvem uma cláusula de ordenação específica, ou uma função de agregação no retorno do resultado, ela adiciona uma complexidade e tempo de execução desnecessários. Para estes casos, é possível obter o mesmo resultado com a concatenação dos resultados parciais, sem a necessidade de comunicação com o SGBDX. Deve-se garantir,

contudo, que os arquivos contendo os resultados parciais possuam referência à ordenação entre as subconsultas que os geraram, de forma que seja possível montar o resultado final respeitando a ordem entre elementos existentes na base de dados de entrada. Isto pode ser facilmente realizado com a adição de informação no nome do arquivo de resultado parcial sobre o início do intervalo considerado na subconsulta que o gerou. Assim, basta realizar uma ordenação dos arquivos contendo os resultados parciais, e adicioná-los um a um no resultado final. A Figura 12 apresenta um pseudocódigo para esta estratégia. Com esta melhoria, foi possível reduzir em 78% o tempo total de execução da consulta para os casos em que a concatenação se aplica.

```
Função ExecutaComposiçãoComColeçãoTemporária(valores[], consulta)
       // valores é uma lista com os caminhos dos
       // arquivos no sistema de arquivos distribuído
       // com resultados parciais

       CriaColeçãoSGBDX(COLECAO)

       para cada documento em valores faça
              doc ← RecuperaDocumentoDFS(documento)
              AdicionaDocumentoColeçãoSGBDX(COLECAO,doc)
       fim-para

       consultaFinal ← CriaConsultaFinal(consulta,COLECAO)
       resultadoFinal ← ExecutaConsultaSGBDX(consultaFinal)
Fim-Função
```

Figura 11. Pseudocódigo para composição por coleção temporária

```
Função ExecutaConcatenação(valores[])
       // valores é uma lista com os caminhos dos
       // arquivos no sistema de arquivos distribuído
       // com resultados parciais

       Ordena(valores)
       resultadoFinal ← ""
       para cada documento em valores faça
              resultadoFinal ← resultadoFinal + documento
       fim-para
Fim-Função
```

Figura 12. Pseudocódigo para composição por concatenação

Para permitir composição do resultado final por concatenação, contudo, a consulta de entrada não deve possuir cláusulas de ordenação nem fazer uso de funções de agregação no retorno do resultado. Esta verificação é facilmente realizada com uma análise da consulta de entrada. A Figura 13 apresenta o pseudocódigo para esta análise.

```
Função DeterminaEstratégiaComposição(consulta)
      se consulta possui order by
            retorna "coleção"
      senão se consulta possui função de agregação no resultado
            retorna "coleção"
      senão
            retorna "concatenação"
      fim-se
Fim-Função
```

Figura 13. Pseudocódigo para determinar a estratégia de composição

3.1.3 CORREÇÕES NA IMPLEMENTAÇÃO

Os estudos realizados sobre a implementação original da técnica de fragmentação virtual resultaram também no robustecimento desta técnica. Constatou-se que a execução de consultas com algumas características especiais resultava em erros na implementação original. Foram corrigidos os seguintes problemas:

(1) Consultas que utilizavam ao mesmo tempo caminhos completos e incompletos causavam erro durante a execução da fragmentação;

(2) Consultas que utilizam caminho incompleto sobre coleções também resultavam em erro durante a fragmentação; e

(3) O incremento no tamanho das bases de dados causava rapidamente um erro de falta de memória.

Os itens (1) e (2) foram tratados com correções pontuais na lógica da implementação. O item (3), no entanto, exigiu um trabalho de investigação mais prolongado, com o emprego de ferramentas específicas de análise do uso de memória em aplicações Java, como o JVisualVM[4] e o *Eclipse Memory Analyzer*[5]. Afinal, esperava-se aplicar a técnica de fragmentação virtual em bases relativamente grandes (10 GB) e um alto consumo de memória durante a execução inviabilizava esta aplicação. Durante a investigação, descobriu-se que a causa do rápido incremento no uso de memória estava ligada à maneira como os resultados das consultas eram processados. A Figura 14 apresenta um pseudocódigo simplificado sobre como este processamento era feito. Em linhas gerais, o resultado da consulta era inteiramente escrito em memória antes de ser escrito no arquivo de resultado em disco, o que resultava em um objeto enorme em memória que, mesmo tendo um ciclo de vida curto, ocasionava uma necessidade pontual de recursos que muitas vezes não estavam disponíveis, o que resultava em um erro. A solução para este problema, apresentada no pseudocódigo da Figura 15, foi realizar a escrita do resultado no arquivo em disco enquanto o item de resposta é lido do

[4] http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm

[5] https://eclipse.org/mat

cursor retornado pela execução da consulta. É importante mencionar que a escrita no arquivo se dá por mecanismos de controle de fluxo, de maneira que mesmo um item de resposta grande não ocasiona um elevado consumo de memória. Apesar desta abordagem ser mais custosa e mais complexa em casos de erro, ela garante que o consumo de memória se mantenha baixo, e isto permite que a técnica de fragmentação virtual possa ser aplicada para bases de dados maiores.

```
cursor = executaConsulta(consulta)
resultado = ""
enquanto cursor.temResultados() faça
    resultado = resultado + cursor.proximo()
fim-enquanto
arquivoResultado = abreArquivo("resultado.txt")
adicionaAoArquivo(arquivoResultado, resultado)
fechaArquivo(arquivoResultado)
```

Figura 14. Processamento de resultados de consulta antigo

```
cursor = executaConsulta(consulta)
arquivoResultado = abreArquivo("resultado.txt")
enquanto cursor.temResultados() faça
    adicionaAoArquivo(arquivoResultado, cursor.proximo())
fim-enquanto
fechaArquivo(arquivoResultado)
```

Figura 15. Processamento de resultados de consulta modificado

3.2 A BIBLIOTECA DE FRAGMENTAÇÃO VIRTUAL SIMPLES

A técnica de fragmentação virtual simples para bases de dados XML foi proposta e implementada no PartiX-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), que por sua vez utilizou como base a abordagem PartiX (ANDRADE *et al.*, 2006; FIGUEIREDO; BRAGANHOLO; MATTOSO, 2010). Nesta implementação, o algoritmo da técnica de fragmentação virtual simples foi inserido na abordagem base sem uma estruturação que permitisse o seu posterior reuso em outras abordagens. Ainda, a implementação no PartiX-VP prevê a utilização de um SGBDX específico que compõe a abordagem, o Sedna (FOMICHEV; GRINEV; KUZNETSOV, 2006). Para este trabalho, desejávamos substituir este SGBDX por outro, devido aos resultados obtidos em estudo sobre o tempo de resposta da função *position()* em diversos SGBDX (SILVA; MATTOSO; BRAGANHOLO, 2013). Neste estudo, ficou constatado que o Sedna, indicado como SGBDX C na Figura 8, não apresenta um tempo de recuperação constante para elementos em diferentes posições. Do exposto, decidiu-se incluir no escopo deste trabalho uma reformulação da implementação da técnica de fragmentação virtual, encapsulando-a em uma biblioteca reutilizável. A ideia é permitir que a técnica de fragmentação virtual possa ser facilmente empregada em diversos *frameworks* (como o Hadoop), sem a necessidade de reimplementá-la. Ainda, com a abstração das

funcionalidades necessárias de um SGBDX em uma interface, foi possível tornar a técnica facilmente extensível para incorporar o suporte a outros SGBDX.

A Figura 16 mostra o diagrama de classes simplificado da biblioteca de fragmentação virtual criada. Embora não estejam inseridos todos os detalhes da implementação, as principais classes utilizadas por usuários desta biblioteca estão identificadas, em cores mais escuras.

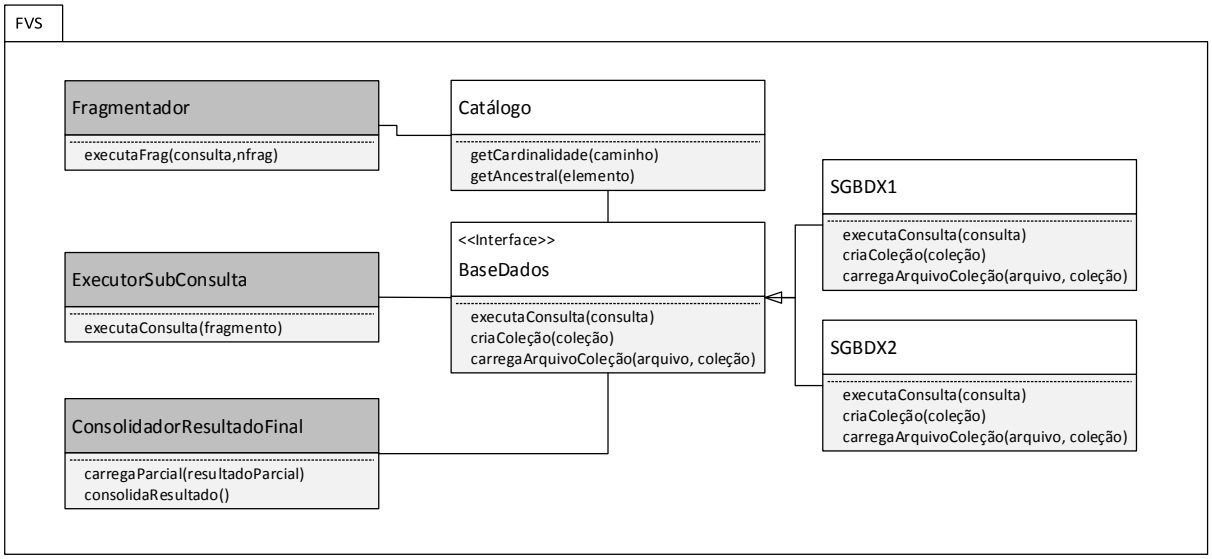


Figura 16. Diagrama de Classes simplificado da Biblioteca de Fragmentação Virtual

Um pseudocódigo exemplificando o uso da biblioteca está na Figura 17. O usuário inicia a execução da consulta instanciando um objeto *Fragmentador*, que é o responsável por realizar a fragmentação da consulta, gerando as subconsultas. Cabe dizer que é na instanciação do *Fragmentador* que o usuário opta por utilizar um catálogo existente ou fazer uso de consultas ao banco de dados para apoiar a técnica na determinação de cardinalidades e relacionamentos. Se o usuário passar um arquivo de catálogo ao instanciar um objeto *Fragmentador*, ele é carregado em memória e utilizado no algoritmo de fragmentação virtual. Caso contrário, o usuário deve passar informações (usuário, senha, endereço, porta, base de dados e tipo) sobre o SGBDX a ser utilizado pela técnica, e a biblioteca automaticamente instancia um objeto *BaseDados* para poder realizar as consultas de cardinalidade naquele sistema.

A fragmentação da consulta inicia quando o método *executaFrag()* do *Fragmentador* é executado, informando a consulta a ser considerada e o número de fragmentos a ser gerado. O algoritmo de fragmentação virtual é inteiramente executado nesta chamada, o que resulta na criação das subconsultas derivadas, cada uma operando sobre um intervalo de dados diferente.

É importante ressaltar também que, para esta biblioteca, o número de fragmentos é arbitrário e pode ser definido consulta a consulta.

A próxima etapa é a execução em paralelo das diversas subconsultas geradas. Para facilitar esta execução, a classe *ExecutorSubConsulta* foi criada. Durante a sua instanciação, ela recebe as informações da instância de SGBDX que será empregada, para que consiga se comunicar com ela e realizar a execução da subconsulta. A execução da consulta acontece de fato quando o método *executaConsulta()* é invocado. Nele, o SGBDX é contatado por meio da abstração de Base de Dados criada, e a subconsulta é então enviada a ele para execução.

Com todos os resultados parciais gerados, inicia-se a composição do resultado final. Também para essa fase da execução foi criada uma classe que encapsula todo o processamento requerido: a classe *ConsolidaResultadoFinal*. Em sua inicialização, esta classe recebe as informações sobre a instância de base de dados que pode ser empregada durante a execução da consolidação, muito embora, conforme discutido na Seção 3.1.2, é possível que não seja necessário utilizá-la. Em seguida, cada resultado parcial é individualmente carregado no objeto, utilizando o método *carregaParcial()*. Ao final, ao executar o método *consolidaResultado()*, o sistema executa a estratégia de consolidação aplicável (concatenação ou criação de coleção temporária), e retorna o resultado final.

```
Função ExecutaConsultaFVS(consulta,nfrag)
        // podemos utilizar ou o arquivo do catálogo da base de
        // dados, ou informações sobre a conexão com o SGBDX
        // (usuário, senha, endereço, porta, base de dados)
        Fragmentador frag = new Fragmentador(arquivoCatalogo)
        fragmentos = frag.executaFrag(consulta,nfrag)
        resultadosParciais = []

        // a etapa abaixo ocorre em paralelo, instanciando um
        // ExecutorSubConsulta em cada nó.
        para cada fragmento em fragmentos faça
                ExecutorSubConsulta exec = new ExecutorSubConsulta(infoBD)
                resultado = exec.executaConsulta(fragmento)
                resultadosParciais.add(resultado)
        fim-para

        // consolidação final
        ConsolidaResultadoFinal crf = new ConsolidaResultadoFinal(infoDB)
        para cada parcial em resultadosParciais faça
                crf.carregaParcial(parcial)
        fim-para
        resultadoFinal = crf.consolidaResultado()
Fim-Função
```

Figura 17. Pseudocódigo de exemplo de uso da biblioteca

3.3 AVALIAÇÃO EXPERIMENTAL

3.3.1 IMPLEMENTAÇÃO E INFRAESTRUTURA

Para avaliar os ganhos obtidos com as melhorias propostas, ambas foram implementadas em linguagem Java, utilizando o Oracle JDK 1.7, tendo como base o código-

fonte da implementação do PartiX-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), abordagem que introduziu o conceito de Fragmentação Virtual para o modelo XML. Contudo, iniciou-se esta implementação com a reestruturação do código fonte relativo ao algoritmo de fragmentação virtual, com o objetivo de compor uma biblioteca, que no caso da linguagem empregada, é um arquivo JAR. Esta biblioteca seria posteriormente empregada nas abordagens VPHadoop e Partix-EVP, propostas por este trabalho, de maneira que já se comprova a sua reusabilidade.

Como já mencionado, além da reestruturação do código-fonte e da criação de classes que simplificam a utilização da biblioteca, todas as chamadas específicas para o SGBDX Sedna (empregado no PartiX-VP) foram devidamente substituídas por abstrações contidas em uma interface, que foram posteriormente implementadas para utilizar alternativamente o SGBDX BaseX (GRÜN; HOLUPIREK; SCHOLL, 2007), versão 8.2.3. Outras implementações desta interface para utilizar outros SGBDX foram também criadas, mas para efeitos desta avaliação, foi considerado somente o SGBDX BaseX. Esta escolha se baseou principalmente no fato deste SGBDX apresentar tempo de acesso constante no uso da função *position()*, o que se pode observar na Figura 8, em que o BaseX está representado como SGBDX B. Além disso, foi também considerada a grande comunidade ativa de usuários deste sistema, o que proporciona suas constantes atualizações e facilidade no acesso a informações sobre o sistema.

Todas as chamadas específicas para consultas de cardinalidade e de relacionamento entre elementos, que antes utilizavam consultas à instância SGBDX também foram substituídas, de forma que passassem a utilizar o novo Catálogo, melhoria proposta por este trabalho. Internamente, o Catálogo é mantido em memória fazendo uso de tabelas *hash*, o que permite o acesso direto a objetos que representam os elementos. Estas tabelas são criadas quando o arquivo de Catálogo, criado externamente, é lido pela abordagem, utilizando SAX *Parser* (“SAX”, 1998), durante a etapa inicial de fragmentação. É possível também configurar o Catálogo para que não utilize suas estruturas internas e realize consultas à instância SGBDX para obter as informações solicitadas. Esta configuração foi utilizada nos experimentos para se medir os ganhos da utilização do catálogo.

A criação do arquivo de Catálogo é feita realizando-se uma única leitura de cada documento XML que compõe a base de dados, também utilizando SAX *Parser*. É importante salientar que tal abordagem de geração externa restringe a aplicabilidade da solução, visto que não lida com eventuais atualizações que a base possa sofrer. Contudo, é perfeitamente

aplicável em cenários nos quais a base de dados é somente leitura, premissa considerada neste trabalho.

Quanto à implementação das estratégias de composição do resultado final, também é possível configurar a biblioteca para sempre utilizar a composição pela criação de coleção temporária. Portanto, é possível fazer uso desta configuração para se medir os ganhos obtidos com a substituição desta estratégia pela concatenação nos casos aplicáveis.

Todos os experimentos foram executados utilizando-se um ou mais nós do cluster SunHPC[6] do Laboratório Nacional de Computação Científica (LNCC). Cada nó deste cluster possui 2 processadores Intel Xeon E5440 Quad Core, totalizando 8 cores, e 16GB de memória RAM DDR2.

Como base de dados, optou-se por utilizar o *benchmark* XMark (SCHMIDT *et al.*, 2002). Esta escolha se deu pela facilidade em se gerar bases SD e MD com tamanhos variados, através da ferramenta disponibilizada pelo *benchmark.*

3.3.2 USO DO CATÁLOGO DA BASE DE DADOS

Utilizando a ferramenta do *benchmark*, gerou-se 2 bases SD (1GB e 10GB), e 2 bases MD (1GB e 10GB), sendo que as bases MD foram constituídas por 100 documentos cada uma. Todas estas bases foram criadas em uma instância do SGBDX BaseX, versão 8.2.3.

A execução da fragmentação foi realizada em somente um dos nós do cluster do LNCC, sendo que este nó teve alocação exclusiva durante toda a execução. Desta forma, buscou-se minimizar as interferências oriundas de processos de outros usuários.

Como consultas, considerou-se aquela identificada como C1 no ANEXO A, e a consulta C13 apresentada na Figura 18. Ambas as consultas possuem 2 caminhos XPath cada, sendo que na C1 estes caminhos estão completos, enquanto que na C13 um dos caminhos é incompleto e requer informações adicionais de relacionamento entre elementos, que também estão disponibilizadas no catálogo. Para as bases MD, as consultas são as mesmas, porém utilizando a função *collection()* ao invés de utilizar a função *document()*. Todas as consultas foram executadas 3 vezes, e considerou-se a média dos tempos obtidos.

O gráfico da Figura 19 apresenta os resultados obtidos no experimento. Para todas as consultas consideradas, percebe-se claramente que o tempo de execução da fragmentação utilizando o Catálogo é sempre menor que o tempo de execução da fragmentação utilizando consultas à base de dados. O menor ganho obtido para a fragmentação na aplicação do

[6] http://www.lncc.br/sunhpc

catálogo foi de aproximadamente 38% no caso da Consulta C1, para a base de dados SD de 1GB. Para o caso da Consulta C13, considerando a base de dados MD de 10GB, este ganho ultrapassa os 99%. É interessante notar que a etapa de fragmentação em bases de dados MD utilizando consultas à base de dados é bastante custosa, uma vez que são necessárias várias consultas para: (i) determinar cardinalidade dos vários elementos envolvidos; (ii) determinar relacionamento entre elementos durante a expansão dos caminhos incompletos, e (iii) determinar a composição de coleções de documentos. Nota-se, também que o tempo de execução da fragmentação utilizando o Catálogo é praticamente o mesmo para os dois tamanhos de bases de dados consideradas. O mesmo não ocorre com a solução via consultas à base de dados.

```
<results>{
for $pe in doc('auction.xml')//person
for $oa in doc('auction.xml')/site/open_auctions/open_auction
where $pe/profile/education = "College" and $oa/type = "Featured" and $pe/@id =
$oa/seller/@person
order by $pe/address/country
return
    <seller>
        <country>{ $pe/address/country }</country>
        <person>{ $pe/name }</person>
        <item id='{$oa/itemref/@item}'>
            <initial_bid>{$oa/initial}</initial_bid>
            <current_bid>{$oa/current}</current_bid>
        </item>
    </seller>
}</results>
```

Figura 18. Consulta C13 para avaliação do catálogo

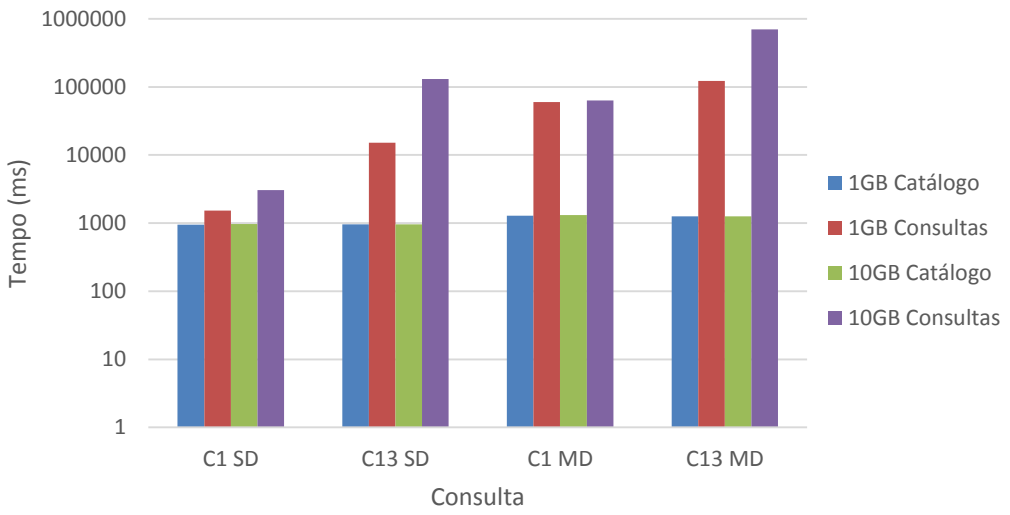


Figura 19. Comparação entre o tempo de execução da fragmentação utilizando catálogo e consultas à base de dados, para duas consultas, em cenário SD e MD (escala de tempo logarítmica)

3.3.3 ESTRATÉGIA DE COMPOSIÇÃO DO RESULTADO FINAL

Para se mensurar o ganho em tempo de execução ao se possibilitar a estratégia de concatenação nos casos aplicáveis, foram escolhidas três consultas nas quais são aplicáveis ambas as estratégias de composição do resultado final: a C1, a C3 e a C8, apresentadas no ANEXO A, sendo que estas consultas apresentam um resultado de tamanho equivalente a 0,84%, 0,01% e 14% da base de dados de entrada, respectivamente. As consultas foram fragmentadas em 32 subconsultas, que foram executadas no cluster LNCC utilizando a abordagem Partix-VP com as melhorias incorporadas, o que resultou em 32 documentos XML de resultados parciais (um para cada subconsulta). É importante mencionar que o ambiente utilizado não compartilhou nós com outros experimentos, mas cada um dos 8 núcleos de processamento foi mapeado para um nó da abordagem. Estes resultados parciais foram salvos localmente, e somente a execução da etapa de composição do resultado final foi considerada no escopo deste experimento. Esta etapa foi repetida 3 vezes para cada consulta, e considerou-se a média dos tempos obtidos.

Como base de dados para este experimento, considerou-se somente a base de dados de 1GB SD gerada pela ferramenta do *benchmark*. Os resultados para bases de dados maiores são proporcionais aos resultados para este tamanho. Além disso, o fato de base de dados ser SD ou MD não influencia a fase de consolidação do resultado final.

A Figura 20 apresenta os resultados da execução desta avaliação. Percebe-se que em todas as consultas a estratégia de concatenação apresentou um tempo de execução bastante inferior ao tempo de execução utilizando a estratégia de coleção temporária, o que demonstra o seu potencial em reduzir o tempo de execução total da consulta. Para o caso da consulta C1, a redução no tempo de execução foi de 34% sobre o tempo de execução da estratégia com coleção temporária. A consulta C3 foi a que apresentou a maior redução: 78%. A consulta C8 também apresentou uma redução expressiva: 56%.

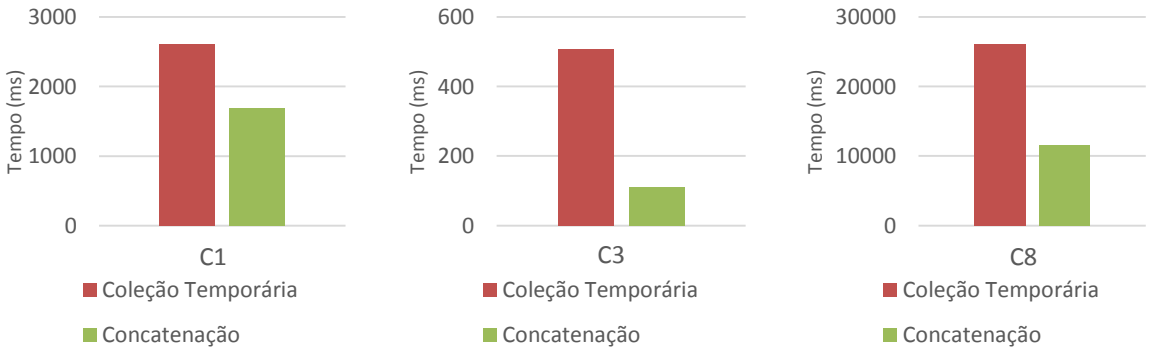


Figura 20. Comparação entre os tempos totais de execução de consultas para as estratégias de coleção temporária e concatenação

3.4 CONSIDERAÇÕES FINAIS

Este Capítulo apresentou as melhorias na técnica de fragmentação virtual simples propostas por este trabalho. Além da reestruturação das funcionalidades em uma biblioteca reutilizável, duas melhorias que resultam na diminuição do tempo de execução de uma consulta foram propostas. A primeira delas refere-se ao emprego de um Catálogo da base de dados para obter informações de cardinalidade e relacionamentos dos elementos XML da base, ao invés de realizar consultas para isso. Foi possível obter ganhos de até 99% no tempo de execução da etapa de fragmentação da consulta. A segunda melhoria refere-se ao emprego de uma estratégia de concatenação na composição do resultado final para os casos aplicáveis. Com esta melhoria, foi possível reduzir em até 78% o tempo total de execução de consulta, conforme os experimentos demonstraram. A partir deste ponto, as demais menções à técnica de fragmentação virtual referem-se à versão já com as melhorias implementadas, exceto quando explicitamente dito o contrário. No próximo Capítulo, descreve-se a abordagem VPHadoop, que já faz uso desta versão.

CAPÍTULO 4 – A ABORDAGEM VPHADOOP

Este Capítulo apresenta a primeira das abordagens propostas por este trabalho para execução distribuída de consultas XQuery, intitulada VPHadoop. Esta proposta aplica a técnica de fragmentação virtual simples para dados XML (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) para executar consultas XQuery em um cluster Hadoop, adicionando à sua infraestrutura as instâncias de SGBDX contendo réplicas da base de dados. Como já dito, o objetivo desta integração é viabilizar um estudo sobre o impacto da variação no número de fragmentos, fazendo uso da infraestrutura de execução paralela do *framework* Hadoop. Utiliza-se os benefícios de balanceamento de carga, tolerância a falhas e de escalabilidade deste *framework*, e assim, não é necessário desenvolver mecanismos próprios para isso.

A Seção 4.1 apresenta a arquitetura da abordagem proposta, detalhando os seus componentes. Em seguida, a Seção 4.2 apresenta todos os passos da execução de uma consulta XQuery utilizando o VPHadoop. A Seção 4.3 apresenta as considerações finais sobre a abordagem.

4.1 ARQUITETURA

A Figura 21 apresenta a arquitetura do VPHadoop, proposta deste trabalho. Como camada inferior, temos o *framework* Hadoop, composto por N nós trabalhadores (*TaskTrackers*). A cada um dos nós trabalhadores do *framework* é adicionada uma instância de um SGBDX, que será responsável por manter a réplica da base de dados naquele nó. A execução em si da consulta ocorre com o envio de um *job* Hadoop específico ao *framework*, composto pelo *Mapper* (implementação da função *map*), pelo *Reducer* (implementação da função *reduce*), e de uma camada de adaptação que sobrescreve o comportamento padrão do *framework* sobre como realizar as leituras dos blocos de dados. No caso, essa leitura não ocorre a partir do disco local dos trabalhadores, mas sim das instâncias SGBDX. É desta diferença que surge a necessidade deste adaptador. O funcionamento interno deste *job* e sua execução são descritos com detalhes na Seção 4.2. A Biblioteca de Fragmentação Virtual é utilizada por todos os componentes do *job* Hadoop, e é onde está implementada a técnica de fragmentação virtual e a execução das subconsultas. O Catálogo é o componente que fornece à Biblioteca de Fragmentação Virtual informações de cardinalidade e relacionamentos dos elementos da base de dados, evitando assim que sejam necessários acessos ao SGBDX para determiná-las. Por fim, a camada que possibilita ao usuário realizar a sua consulta utilizando a

abordagem proposta é a camada Cliente, que recebe a consulta e um conjunto de parâmetros para então construir o *job* a ser executado.

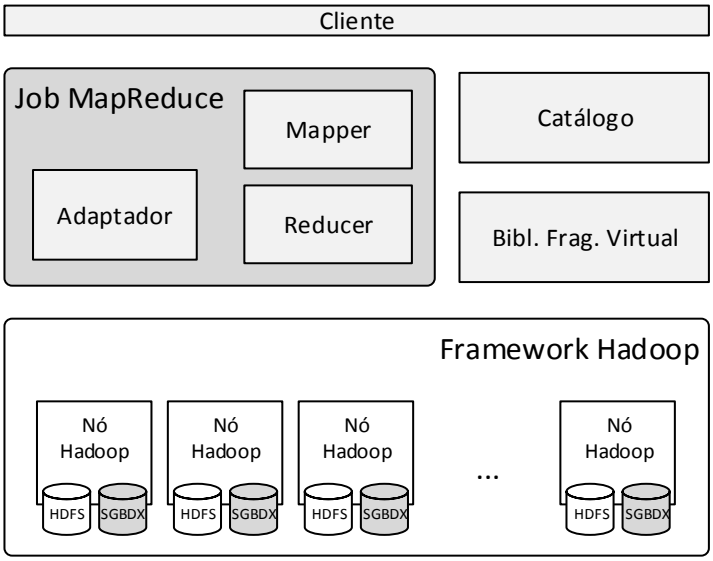


Figura 21. Arquitetura da proposta VPHadoop

4.2 FUNCIONAMENTO

A Figura 22 mostra a relação entre os diversos módulos da abordagem e o fluxo de execução de uma consulta. Este fluxo pode ser dividido em basicamente três passos: (i) preprocessamento, (ii) processamento paralelo e (iii) composição do resultado. Durante o preprocessamento, o VPHadoop analisa a consulta de entrada e aplica a técnica de fragmentação virtual para derivar subconsultas, cada uma atuando sobre um intervalo específico dos dados. No segundo passo, o *framework* Hadoop é utilizado para distribuir as subconsultas entre os *TaskTrackers*, utilizando tarefas do tipo *map*. Ao final de cada tarefa do tipo *map*, o resultado final é escrito no sistema de arquivos distribuído do *framework*, para que possa ser posteriormente recuperado e utilizado na tarefa do tipo *reduce*, em qualquer um dos *TaskTrackers*. Após o término de todas as tarefas do tipo *map*, a abordagem então utiliza uma única tarefa do tipo *reduce* para compor o resultado final, recuperando todos os resultados parciais do sistema de arquivos distribuído e aplicando as funções de ordenação ou agregação existente na consulta de entrada. O resultado final é também escrito no sistema de arquivos distribuído. As próximas subseções descrevem estes três passos em detalhes.

É importante mencionar que a abordagem está focada na execução da consulta. Desta forma, ela não considera a fase da distribuição de dados, que deve ser realizada previamente. Assume-se que cada nó trabalhador possui sua própria instância de SGBDX, com uma réplica completa dos dados de entrada. Desta forma, utiliza-se diversos recursos do SGBDX (como por exemplo, os índices) para a execução das subconsultas.

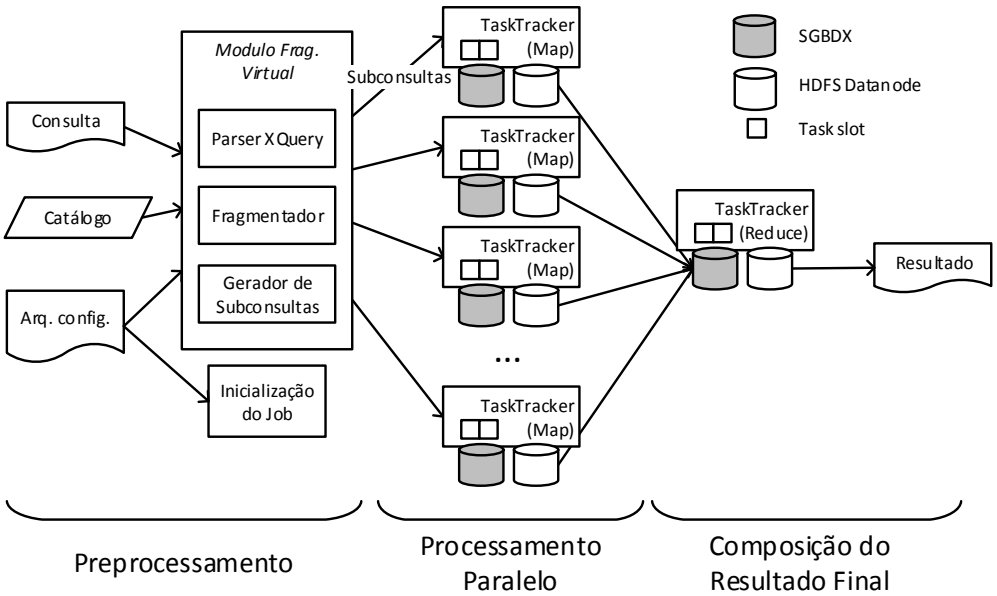


Figura 22. Execução de consulta utilizando o VPHadoop

4.2.1 PREPROCESSAMENTO

O passo de Preprocessamento compreende a preparação do *job* Hadoop e a execução da técnica de fragmentação virtual para gerar as subconsultas, que serão posteriormente executadas em paralelo, como descrevemos na Seção 2.3.2. O algoritmo considerado recebe como parâmetros a consulta original e o número de fragmentos que precisa gerar, e utiliza o Catálogo que contém as informações sobre cardinalidade e relacionamento dos elementos da base de dados. Este catálogo é carregado em memória principal a partir de um arquivo XML gerado externamente. A geração deste catálogo é comentada na Seção 3.1.1.

É importante dizer que, diferentemente da abordagem proposta por Rodrigues, Braganholo e Mattoso (2011), na abordagem proposta por este trabalho o número de fragmentos gerados pela técnica de fragmentação virtual não está ligado ao número processadores (*TaskTrackers*) existentes no cluster. Além disso, este parâmetro é configurável a cada consulta enviada para execução. Esta é uma caraterística muito importante que pretende minimizar os efeitos da distribuição não uniforme dos dados no tempo de execução. Quanto menor for um fragmento, menor será o impacto desta distribuição não uniforme no tempo de execução de uma consulta. Contudo, existe um custo adicional de comunicação e de processamento para distribuir a execução e combinar os vários resultados parciais, que aumenta conforme o número de fragmentos aumenta. Este fato precisa ser considerado quando se está decidindo o número exato de fragmentos. Quanto mais fragmentos se gera, maior será o custo decorrente destes fatores. É observado também que as características da

consulta também influenciam o melhor número de fragmentos para ela. A Seção 6.5 traz uma análise sobre o impacto em se aumentar o número de fragmentos em uma execução.

4.2.2 PROCESSAMENTO PARALELO

Durante a fase de processamento paralelo, o *framework* Hadoop distribui a execução das subconsultas (que irão considerar somente um subintervalo dos dados de entrada) como tarefas do tipo *map* a serem processadas pelos *TaskTrackers* em cada nó do cluster. O número de tarefas do tipo *map* simultâneas (*slots* de execução) que um *TaskTracker* executa é um parâmetro configurável do *framework*, que geralmente reflete o número de processadores do nó. Desta forma, o *JobTracker* então envia a um *TaskTracker* um número de tarefas do tipo *map* igual ao número de *slots* de execução disponíveis, e conforme *slots* disponíveis forem aparecendo devido a término de execução, o *JobTracker* envia novas tarefas.

Para compreender como as subconsultas são mapeadas em tarefas do tipo *map*, é importante lembrar que, em uma execução Hadoop padrão, cada tarefa do tipo *map* processa um bloco de dados, denominado *split*, sendo que o *framework* faz uso de um módulo adaptador associado ao *job* para traduzir os dados contidos neste *split* em registros do tipo <chave, valor>, denominados *records*, e repassar cada *record* a uma chamada da função *map* do *job*. Como já foi mencionado, a abordagem proposta por nosso trabalho não faz uso de blocos de dados, mas sim de uma instância de SGBDX em cada um dos nós do cluster, que contém uma réplica completa da base de dados. Ainda, a própria instância do SGBDX é utilizada para executar a subconsulta, aproveitando assim as otimizações nele implementadas. Desta forma, determinou-se que, para o *job* da nossa abordagem, cada subconsulta é mapeada para um *record* do tipo <chave, valor>, sendo que: chave contém a posição inicial do intervalo determinado na fragmentação virtual, e o valor é a própria subconsulta.

Considerando que cada *record* é mapeado para uma subconsulta, é bastante intuitivo que a execução desta subconsulta seja realizada pela função *map* do *job* que compõe a abordagem proposta. O pseudocódigo para esta função *map* está apresentado na Figura 23. Além de realizar a chamada ao SGBDX local para executar a subconsulta, a função *map* também salva o seu resultado em um arquivo no sistema de arquivos distribuído, para que este possa depois ser recuperado e combinado com os resultados das demais subconsultas. Por fim, a função *map* emite um par intermediário <chave, valor> para o *framework*, onde valor é o caminho do arquivo no sistema de arquivos distribuído e a chave é um valor constante, previamente definido.

É importante mencionar que o resultado parcial é salvo em um arquivo cujo nome é gerado de modo a permitir que a ordem entre intervalos considerados nas subconsultas possa se refletir na combinação dos resultados parciais. Isto é necessário pois no formato XML, a ordem dos elementos de um documento é importante. Assim, a resposta de uma consulta deve conter os elementos na mesma ordem em que aparecem no documento de entrada, e não de forma aleatória. Rodrigues, Braganholo e Mattoso (2011) faz a inserção de um elemento específico no resultado parcial para realizar esta ordenação na combinação. Contudo, este trabalho propõe uma melhoria na técnica de fragmentação virtual que consiste na possibilidade de combinar os resultados parciais com concatenação quando aplicável, para melhorar o desempenho. Isto só é possível ser recuperarmos os arquivos de resultados parciais em ordem, e para fazer isso, utiliza-se o próprio nome do arquivo para realizar esta ordenação. O Capítulo 3 apresenta as propostas de melhorias para a técnica de fragmentação virtual, e detalha melhor como a concatenação pode ser aplicada.

```
Função map(par <chave,valor>)
       // chave é o início do intervalo
       // valor é a subconsulta
       resultado ← ExecutaSubconsultaSGBDXLocal(valor)
       se resultado não é vazio
              nomeArquivo ← GeraNomeArquivo(chave)
              SalvaResultadoDFS(resultado, nomeArquivo)
              EmiteIntermediario(<"1", nomeArquivo>)
       fim-se
Fim-Função
```

Figura 23. Pseudocódigo para a função *map*

A Figura 24 apresenta de forma gráfica o processo de execução de uma consulta com a abordagem, até o momento da execução paralela das subconsultas. Inicia-se com o preprocessamento da consulta de entrada, no qual o atributo de fragmentação virtual é determinado. No caso, o elemento escolhido para ser o atributo de fragmentação virtual é o determinado pelo caminho "*/site/people/person*", que, neste exemplo, possui cardinalidade de 200.000.

O número de fragmentos a ser gerado na fragmentação virtual é configurável para cada execução. No exemplo, supõe-se que o número de fragmentos desejado é 200. Desta forma, a intervalo de 200.000 elementos determinado pelo atributo de fragmentação virtual é dividido em 200 subintervalos de tamanho 1.000, como por exemplo, aquele com início no elemento na posição 80.001 e término no elemento na posição 81.000. Com estes subintervalos, por meio da reescrita da consulta de entrada, geram-se 200 subconsultas, inserindo um predicado de seleção no caminho do atributo de fragmentação virtual (como por exemplo, *"/site/people/person[position() >= 80.001 and position() <= 80.000]*").

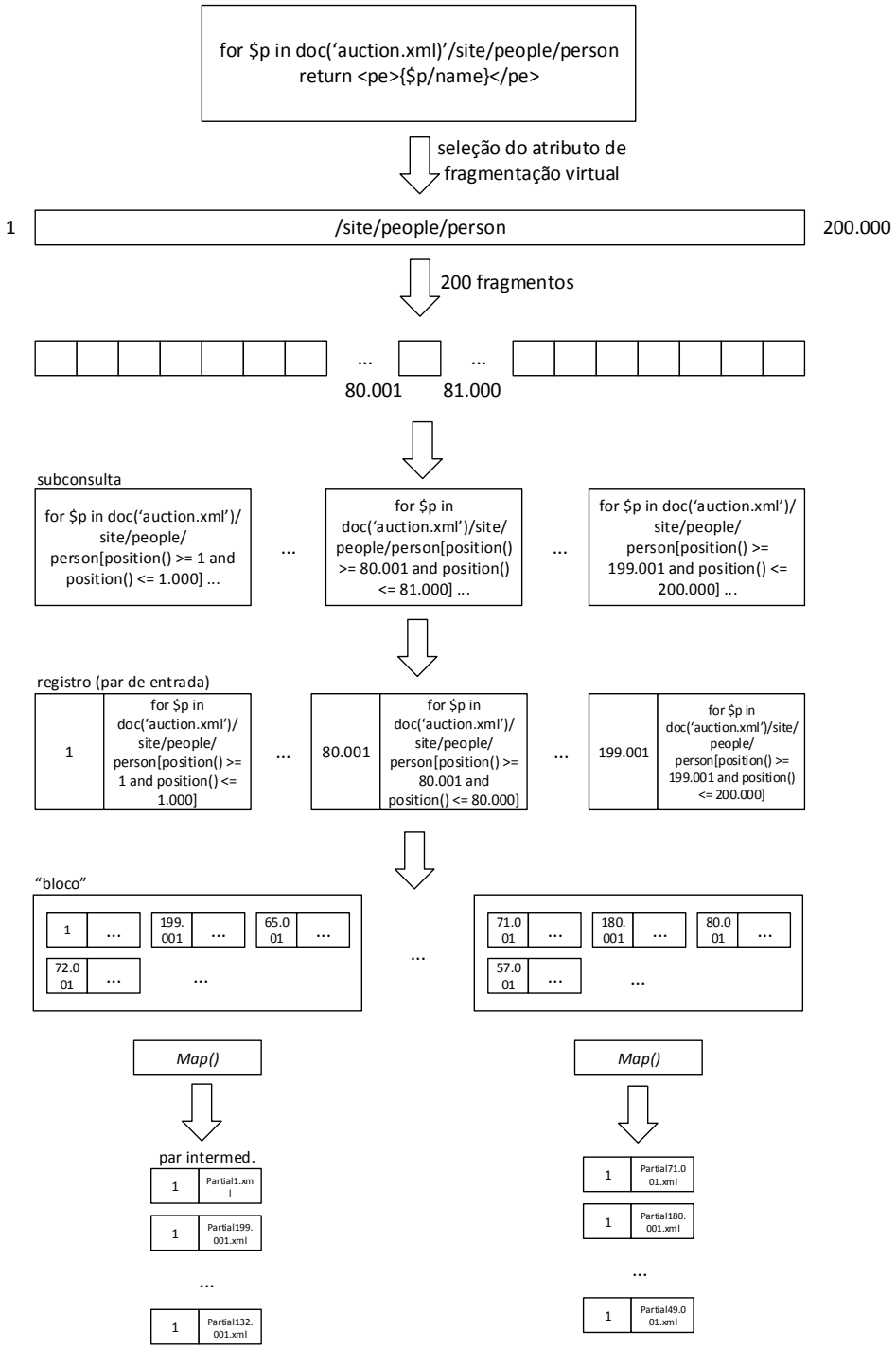


Figura 24. Preprocessamento e execução paralela do VPHadoop

Dentro da abordagem VPHadoop, a execução de uma subconsulta é o que define uma unidade de trabalho passível de distribuição. É preciso, portanto, mapear estas unidades de trabalho para as unidades de trabalho do Hadoop: os registros (*records*) e os blocos (*splits*), sendo que um bloco é um conjunto de registros. Cada subconsulta é mapeada como sendo um registro do Hadoop, sendo que a chave deste registro é o índice que determina o início do subintervalo. Os blocos são, portanto, conjuntos de subconsultas. Como já mencionado, o *framework* Hadoop cria uma tarefa do tipo *map* para cada bloco que precisa ser processado, e

invoca a função *map* para cada registro extraído deste bloco. Percebe-se, portanto, que o mapeamento de 200 subconsultas para registros e blocos pode se dar de várias formas distintas: 200 blocos com 1 registro cada, 100 blocos com 2 registros cada, 50 blocos com 4 registros cada etc. Considerando que o tempo de inicialização de uma tarefa do tipo *map* e a sua distribuição pode não ser desprezível, a forma escolhida pode impactar no tempo total de execução.

Com o exposto, poder-se-ia deduzir que quanto menor o número de tarefas do tipo *map*, menor será o impacto do tempo de inicialização e distribuição no tempo total de execução. Deve-se considerar, contudo, que tal prática pode impactar no balanceamento de carga da solução, uma vez que todos os registros de um bloco serão alocados a um mesmo nó trabalhador, pois serão processados por uma única tarefa do tipo *map*. A Seção 6.5.1 apresenta uma análise detalhada deste parâmetro e do seu impacto no tempo de execução para diferentes tipos de consultas. É importante também notar que os blocos não são compostos por registros contíguos, mas sim aleatórios. Esta forma de compor os blocos permite distribuir a execução de intervalos cujo processamento é mais intenso, e não os manter em um mesmo nó trabalhador, como ocorreria se colocássemos subintervalos contíguos no mesmo bloco.

Seguindo a execução na Figura 24, com a definição dos blocos compostos por subconsultas, o *framework* Hadoop cria as tarefas do tipo *map* necessárias para processar estes blocos e as distribui para execução nos diversos nós trabalhadores do cluster. No nó trabalhador, para cada registro é chamada a função *map* definida para a abordagem VPHadoop, o que faz com que a subconsulta daquele registro seja executada na instância de SGBDX daquele nó. O resultado obtido é escrito em um arquivo no HDFS, tendo o nome uma referência ao subintervalo considerado, de forma a permitir a posterior composição do resultado final. Este nome de arquivo é emitido ao *framework* como resultado intermediário, sendo que a chave deste resultado fica definida como sendo sempre a mesma (no exemplo, o número “1”).

Após a emissão dos pares intermediários <chave, valor>, ainda durante a execução da tarefa do tipo *map*, o *framework* Hadoop realiza a etapa de ordenação e divisão destes pares em partições. No caso, como todos os pares intermediários emitidos contêm a mesma chave, todos compõem uma única partição, que será depois recuperada pela única tarefa do tipo *reduce*.

4.2.3 COMPOSIÇÃO DO RESULTADO FINAL

A tarefa do tipo *reduce* é iniciada após a execução das tarefas do tipo *map* terem atingido um determinado percentual de execução, que também é configurável. O valor padrão para este parâmetro é de 66% da execução de tarefas do tipo *map* completada, e este valor foi mantido na abordagem proposta. A iniciação prematura da tarefa do tipo *reduce* permite que o *framework* já possa iniciar a coleta das partições de pares intermediários resultantes das execuções já completadas de tarefas do tipo *map*. Assim, tão logo a última tarefa do tipo *map* termine, basta finalizar esta última recuperação de resultados intermediários e prosseguir com a execução, que consiste no agrupamento dos pares intermediários resultantes de todas as tarefas do tipo *map*, baseado no valor da chave. Como já mencionado, a abordagem proposta utiliza somente um valor de chave, de maneira que todos os pares intermediários sejam processados por uma única tarefa do tipo *reduce*. Em seguida, o *framework* Hadoop invoca a execução da função *reduce* do *job* que compõe a abordagem proposta. O pseudocódigo para esta função está apresentado na Figura 25. Inicialmente, determina-se qual estratégia de composição pode ser adotada, baseado na consulta de entrada. Em seguida, executa-se esta estratégia de composição, seja ela a de concatenação dos resultados parciais, ou a de criação de coleção temporária no SGBDX local. Conforme já mencionado, esta é uma melhoria proposta por este trabalho na técnica de fragmentação virtual sobre dados XML original, e está descrita em detalhes na Seção 3.1.2.

Após o término da execução do *job* Hadoop, o módulo Cliente da abordagem é notificado e realiza então a cópia do arquivo contendo o resultado final, do sistema de arquivos distribuído do *framework* Hadoop para o sistema de arquivos local do computador que realizou a execução da abordagem, em local indicado pelo usuário.

```
Função reduce(chave, valores[], consulta)
       // valores é uma lista com os caminhos dos
       // arquivos no sistema de arquivos distribuído
       // com resultados parciais
       estratégia ← DeterminaEstratégiaComposição(consulta)
       se estratégia = "concatenação"
              ExecutaConcatenação(valores)
       senão
              ExecutaComposiçãoComColeçãoTemporária(valores, consulta)
       fim-se
Fim-Função
```

Figura 25. Pseudocódigo para a função *reduce*

4.3 CONSIDERAÇÕES FINAIS

Este Capítulo apresentou a abordagem VPHadoop, que faz uso da técnica de fragmentação virtual e do *framework* Hadoop para realizar a execução distribuída de

consultas XQuery em um cluster. A abordagem se propõe a mitigar o problema do desbalanceamento de carga gerado pela distribuição não uniforme dos dados ao aumentar o número de fragmentos (e subconsultas) gerados pela técnica, e fazer uso de um *framework* que já forneça o balanceamento de carga automático, alta escalabilidade e tolerância a falhas, sem a necessidade de reimplementar estas funções. No Capítulo 5 opta-se por não utilizar um *framework* de distribuição pronto, e cria-se uma outra abordagem com mecanismos próprios de distribuição. O Capítulo 6 apresenta a análise experimental realizada sobre a utilização destas abordagens e discute os resultados obtidos.

CAPÍTULO 5 – A ABORDAGEM PARTIX-EVP

Este Capítulo apresenta a segunda abordagem proposta por este trabalho: a Partix-EVP (*Partix – Enhanced Virtual Partitioning*). Esta abordagem utiliza como base a Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), que realiza a fragmentação virtual simples para a execução distribuída de consultas XQuery. No caso da abordagem Partix-EVP, a fragmentação virtual simples sofre uma variação e, assim como na abordagem VPHadoop descrita no Capítulo 4, possibilita que mais fragmentos (subconsultas) sejam gerados a partir da consulta inicial, não ficando limitado ao número de nós processadores. Com um maior número de fragmentos, espera-se que sejam minimizados os efeitos de desbalanceamento de carga ocasionados pela distribuição não uniforme dos dados.

Como o número de fragmentos é maior que o número de processadores, é necessário empregar técnicas de distribuição dinâmica da execução para garantir que todos os fragmentos gerados sejam processados, e também que a utilização do cluster seja otimizada. No caso do Partix-EVP, optou-se por implementar um algoritmo simples, utilizando a arquitetura mestre-escravo, para realizar este trabalho. Com isso, espera-se também poder comparar tempos de execução de consultas desta abordagem com outras que possuem mecanismos mais sofisticados de apoio, como o *framework* Hadoop.

A Seção 5.1 apresenta a arquitetura da abordagem proposta. Em seguida, a Seção 5.2 apresenta detalhes sobre o mediador, e como ocorre a distribuição dinâmica das subconsultas para execução. Por fim, a Seção 5.3 apresenta algumas considerações finais.

5.1 ARQUITETURA

Assim como na abordagem Partix-VP, a arquitetura da abordagem Partix-EVP também é baseada na presença de um nó mediador, que concentra a tarefa de realizar a fragmentação virtual, distribuir a execução entre os nós e compor o resultado final a partir dos resultados parciais. Estas tarefas são realizadas pelos módulos presentes neste nó mediador da arquitetura, conforme se vê na Figura 21. Esta abordagem faz uso da Biblioteca de Fragmentação Virtual, apresentada no Capítulo 3 e, portanto, nota-se que esta arquitetura já contempla a utilização do catálogo da base de dados no **Fragmentador**, que é o responsável pela análise da consulta de entrada e geração das subconsultas (fragmentos). O **Controlador de Execução de Subconsultas** é o módulo responsável por gerenciar a distribuição das subconsultas geradas na fragmentação virtual entre os nós trabalhadores, realizando com eles troca de mensagens MPI para enviar as subconsultas e receber sinalizações de disponibilidade

para execução. O funcionamento deste controlador é melhor detalhado na Seção 5.2. O módulo **Compositor de resultado final** segue o funcionamento já detalhado na Seção 2.3.2 para agregar todos os resultados parciais e compor o resultado final. Este módulo também conta com a melhoria adicionada por este trabalho, conforme discutido no Capítulo 3. Os demais nós da solução (nós trabalhadores) contam com um módulo **Executor de subconsulta**, que é responsável por enviar a subconsulta para execução em um SGBDX, recuperar o resultado parcial e salvá-lo no disco compartilhado, comunicando o fato para o nó mediador.

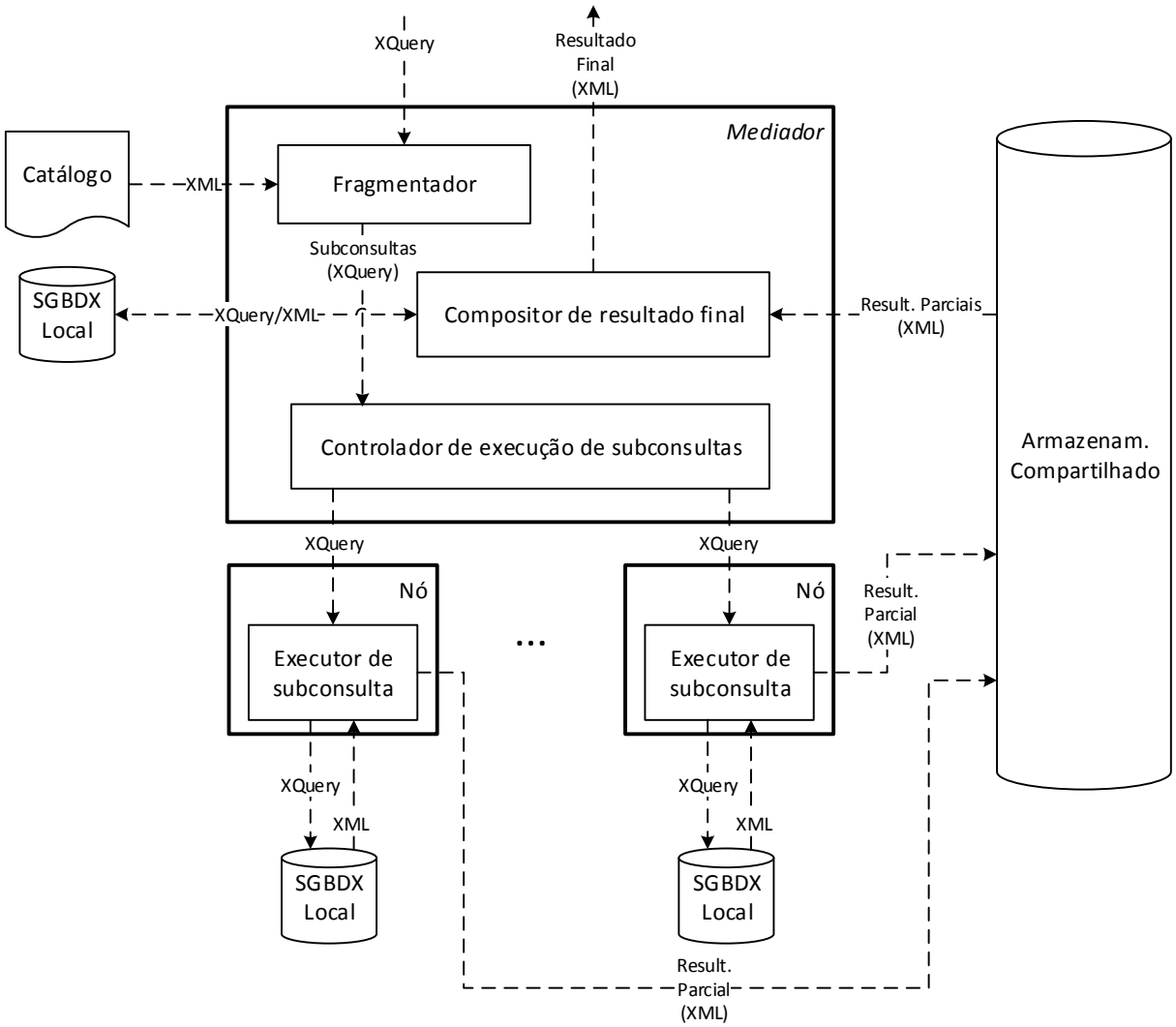


Figura 26. Arquitetura da abordagem Partix-EVP

Para viabilizar o envio ao nó mediador dos resultados parciais de cada subconsulta, executadas em cada nó trabalhador, a abordagem Partix-EVP faz uso de uma solução de armazenamento compartilhada entre os nós, na qual todos os nós podem escrever e ler dados. Idealmente, os tempos de acesso e de escrita dos dados neste armazenamento compartilhado devem ser desprezíveis para que o tempo de execução de consultas com a abordagem não seja

afetado. Contudo, as soluções de armazenamento compartilhadas geralmente adicionam tempos extras de processamento devido à concorrência no acesso. De fato, durante a avaliação experimental, foi notado uma variação nos tempos de execução que pode ter origem nesta concorrência. Este assunto é discutido no Capítulo 6.

É importante mencionar que, diferentemente do Partix-VP, a abordagem Partix-EVP não utiliza o nó mediador para a execução paralela de subconsultas. Com isso, considerando que existam *n* nós de processamento para a abordagem, serão utilizados *n-1* nós para a execução paralela de subconsultas. Na abordagem Partix-VP não há um controle do nó mediador sobre a execução de cada nó trabalhador, o que o libera para que também possa executar uma subconsulta. Já na abordagem Partix-EVP, o nó mediador recebe constantemente informações sobre términos de execução nos demais nós, e atua na distribuição de novas subconsultas. Desta forma, para simplificar a arquitetura e termos somente um fluxo de execução em cada nó, optou-se por não utilizar o nó mediador para execução de subconsultas.

5.2 FUNCIONAMENTO

Assim como nas demais abordagens que utilizam a fragmentação virtual, a abordagem Partix-EVP está focada na execução da consulta. Desta forma, ela não considera a fase da distribuição de dados de entrada, que deve ser realizada previamente. Para que a execução da consulta ocorra, assume-se que cada nó possui sua própria instância de SGBDX local, com uma réplica completa dos dados de entrada. Com isso, a abordagem pode se beneficiar dos diversos recursos disponíveis em um SGBDX (como por exemplo, os índices) para a execução das subconsultas.

A execução de uma consulta XQuery com a abordagem Partix-EVP pode ser dividida em 3 passos: (i) preprocessamento, (ii) execução paralela de subconsultas e (iii) composição do resultado final. Durante o preprocessamento, o nó mediador recebe como entrada a consulta e o número de fragmentos em que se quer dividir o conjunto total de dados. Em seguida, o nó mediador executa a técnica de fragmentação virtual sobre a consulta recebida e deriva as subconsultas, cada uma atuando sobre um intervalo específico dos dados. No segundo passo, o nó mediador inicia a distribuição das subconsultas aos nós trabalhadores, por meio de mensagens MPI. Ao receber a mensagem, um nó trabalhador imediatamente invoca a execução da subconsulta em sua instância local do SGBDX, escreve o resultado obtido em um arquivo no armazenamento compartilhado, e sinaliza o término para o mediador. O nó mediador então verifica se existem ainda subconsultas para serem executadas,

e envia nova consulta ao nó trabalhador. Este ciclo se repete até que todas as subconsultas sejam executadas. O terceiro passo então se inicia, no qual o mediador realiza a leitura de todos os resultados parciais no armazenamento compartilhado, e realiza a composição do resultado final, aplicando as funções de ordenação ou agregação existente na consulta de entrada. O resultado é escrito no disco local do nó mediador.

5.2.1 PREPROCESSAMENTO

Durante esta fase, o nó mediador realiza a análise da consulta de entrada e a geração das subconsultas, armazenando-as em uma lista interna. Os demais nós realizam a sua inicialização e enviam ao nó mediador uma mensagem MPI informando que estão prontos para iniciar a execução de subconsultas.

Sobre a fragmentação virtual, cabe novamente ressaltar que, na abordagem Partix-EVP, o número de fragmentos gerados pela técnica de fragmentação virtual não está ligado ao número de nós processadores existentes no cluster. Este parâmetro é configurável a cada consulta enviada para execução. Um número maior de fragmentos pode diminuir os impactos causados pela distribuição não uniforme de dados. Contudo, este aumento de fragmentos gera também um aumento no processamento extra para distribuição, coleta de resultado parcial e composição do resultado final. Este fato precisa ser considerado quando se está decidindo o número exato de fragmentos. As características da consulta também influenciam o melhor número de fragmentos para a sua execução. A Seção 6.5.2 traz uma análise sobre o número de fragmentos em uma execução com a abordagem Partix-EVP.

5.2.2 PROCESSAMENTO PARALELO DE SUBCONSULTAS

Com a lista de subconsultas, o nó mediador inicia um ciclo de passos que só se encerra quando todas as subconsultas forem executadas. Os passos deste ciclo são os seguintes (ignora-se, para simplificar a descrição, os tratamentos de erros que podem ocorrer):

(1) O nó mediador aguarda receber uma mensagem de qualquer um dos nós trabalhadores.

(2) O nó mediador processa essa mensagem, que pode ser: (a) uma mensagem de pronto para execução, enviada pelo nó trabalhador após a inicialização; ou (b) uma mensagem de término de execução de uma subconsulta.

(3) No caso (b), o nó mediador salva a localização do arquivo contendo o resultado parcial no armazenamento compartilhado, informação esta que foi enviada dentro da mensagem.

(4) Se ainda existem subconsultas para serem processadas, o nó mediador recupera a próxima e envia assincronamente para o nó trabalhador que originou a mensagem recebida. A execução volta para o passo 1.

(5) Se todas as subconsultas tiverem sido processadas, o nó mediador envia ao nó trabalhador uma mensagem sinalizando o término das subconsultas, para que ele possa parar a sua execução.

(6) Se todos os nós já tiverem recebido a mensagem de término das subconsultas, o ciclo se encerra. Senão, a execução volta para o passo 1.

Durante este período, os nós trabalhadores, após a inicialização e o envio da mensagem ao mediador sinalizando a prontidão para execução, também iniciam um ciclo, cujos passos estão abaixo descritos (novamente ignorando os erros possíveis, para simplificar a descrição):

(1) O nó trabalhador espera uma mensagem do nó mediador.

(2) O nó trabalhador processa essa mensagem, que pode ser: (a) uma nova subconsulta para execução; ou (b) uma sinalização de que todas as subconsultas já foram executadas.

(3) Se a mensagem for (a), o nó trabalhador utiliza a sua instância de SGBDX local para executar a subconsulta recebida e salva o resultado em um arquivo do armazenamento compartilhado. Ao término, envia uma mensagem ao nó trabalhador sinalizando este término, contendo o caminho do arquivo com o resultado parcial no armazenamento compartilhado. A execução volta para o passo 1.

(4) Se a mensagem for (b), o nó trabalhador encerra sua execução.

É importante mencionar que o nome do arquivo com resultado parcial é gerado de modo a refletir qual o fragmento que o gerou. Portanto, o nó mediador pode, por meio de uma ordenação dos nomes dos arquivos, recuperar a sequência de origem. Isto é necessário pois no formato XML, a ordem dos elementos de um documento é importante. Assim, a resposta de uma consulta deve conter os elementos na mesma ordem em que aparecem no documento de entrada, e não de forma aleatória.

5.2.3 COMPOSIÇÃO DO RESULTADO FINAL

De posse de todos os caminhos de arquivos com os resultados parciais, o nó mediador então realiza a ordenação destes nomes de arquivos para que os arquivos sejam recuperados mantendo-se a ordem da base de dados de entrada. Em seguida, determina-se qual estratégia

de composição que pode ser adotada baseado na consulta de entrada. A discussão sobre estas estratégias está na Seção 3.1.2, que trata da melhoria adicionada por este trabalho à técnica de fragmentação virtual. A estratégia de composição definida é então executada. O resultado desta composição é escrito em um arquivo do disco local do nó mediador.

5.3 CONSIDERAÇÕES FINAIS

Este Capítulo apresentou a abordagem Partix-EVP para realizar a execução distribuída de consultas XQuery em um cluster. Esta abordagem propõe uma mudança na técnica de fragmentação virtual, que consiste no aumento do número de fragmentos gerados e distribuídos, para mitigar os efeitos da distribuição não uniforme dos dados no tempo de execução de consulta. Esta abordagem faz uso de um controle de execução centralizado no nó mediador para distribuir a execução das subconsultas de maneira a otimizar a utilização dos nós existentes. A técnica de fragmentação virtual utilizada nesta abordagem também conta com todas as melhorias propostas por este trabalho, indicadas no Capítulo 3. O Capítulo 6 apresenta a análise experimental realizada sobre as implementações das abordagens Partix-EVP e VPHadoop e discute os resultados sobre a variação do número de fragmentos e a comparação entre as abordagens.

CAPÍTULO 6 – AVALIAÇÃO EXPERIMENTAL

Os protótipos das abordagens VPHadoop e Partix-EVP foram desenvolvidos e utilizados em uma avaliação experimental para: (i) analisar o impacto que a variação no número de fragmentos utilizados na fragmentação virtual causa no tempo de execução de consultas XQuery; (ii) comparar os tempos de execução das abordagens propostas por este trabalho com aqueles da abordagem centralizada e da abordagem Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) e verificar se houve diminuição no tempo total de execução; e (iii) verificar se o aumento no número de fragmentos é capaz de mitigar possíveis desbalanceamentos de carga entre os processadores, causados por uma distribuição não uniforme dos dados de entrada.

A próxima Seção deste Capítulo traz detalhes sobre o ambiente de execução empregado nesta avaliação. Em seguida, a Seção 6.2 descreve a base de dados considerada, bem como as consultas que foram definidas para serem executadas nesta avaliação. A Seção 6.3 traz detalhes dos protótipos que implementam as abordagens deste trabalho e também da adaptação realizada na abordagem PartiX-VP, para possibilitar a sua execução no novo ambiente. A Seção 6.4 apresenta a metodologia considerada para esta avaliação. A Seção 6.5 traz a análise do número de fragmentos, tanto para a abordagem VPHadoop, quanto para a abordagem PartiX-EVP. Os resultados desta análise do número de fragmentos são então utilizados na comparação entre as abordagens, que está apresentada na Seção 6.6. A Seção 6.7 realiza uma análise sobre o balanceamento de carga das abordagens distribuídas. Por fim, a Seção 6.8 tece algumas conclusões sobre os resultados obtidos.

6.1 AMBIENTE DE EXECUÇÃO

Todos os experimentos aqui descritos e discutidos foram executados no Cluster SUNHPC[7] do Centro Nacional de Processamento de Alto Desempenho (CENAPAD), que é mantido pelo Laboratório Nacional de Computação Científica[8] (LNCC). Este cluster conta com 72 nós de execução, cada um com 2 processadores Intel Xeon E5540 Quad Core 64 bits, totalizando 8 núcleos de processamento em cada máquina, além de 16 GB de memória PC2-5300 DDR2. Todos os nós possuem o sistema operacional Linux, kernel 2.6.18, 64 bits. Os nós deste cluster estão ligados entre si por *switches* DDR com taxa de transferência de 20GB/s, e o *storage* está conectado a uma interface Gigabit Ethernet 1GB/s.

[7] http://www.lncc.br/sunhpc/

[8] http://www.lncc.br/

Quanto ao armazenamento, os nós possuem um disco local, mas apenas o diretório */tmp* destes discos está liberado para escrita. Aos usuários é alocado espaço em um *storage* compartilhado, e este é montado em todos os nós como diretório *home*.

Todas as execuções devem ser submetidas ao cluster por meio do gerenciador de filas *Sun Grid Engine* (SGE), que recebe *jobs* por meio de *scripts* de execução escritos pelos usuários. Os *scripts* de execução deste experimento[9] podem ser divididos em três etapas principais: a inicialização, a execução das consultas, e a finalização. Durante a etapa de inicialização, todos os serviços necessários para aquela execução são levantados. São eles: as instâncias do SGBDX nos diversos nós, os processos do *framework* Hadoop, a configuração de diversas variáveis de ambiente e a criação de diretório para armazenamento dos *logs* de execução. Em seguida, é realizada a etapa de execução das consultas em si, que depende do contexto daquele experimento. Por último, a etapa de finalização é responsável por parar todos os serviços que foram inicializados, apagar eventuais arquivos que foram criados somente para esta execução e coletar mensagens de *log* para análise posterior.

No cluster SUNHPC, os usuários são limitados ao uso simultâneo de, no máximo, 128 núcleos de processamento. Como cada nó do cluster possui 8 destes núcleos, se for considerado o caso em que é necessário o não compartilhamento do nó durante a execução, um usuário está limitado a utilizar, no máximo, 16 nós simultâneos. Contudo, por restrições de tempo, não foi possível realizar todas as execuções, com todas as abordagens, na configuração de 16 nós. Desta forma, as execuções deste trabalho ficaram limitadas a utilização de 8 nós simultâneos, ou seja, 64 núcleos de processamento.

Na avaliação experimental, como a variável que se está medindo é altamente influenciada por outras tarefas que podem estar utilizando a CPU e realizando escritas e leituras nos discos, é necessário que os nós utilizados sejam alocados de forma exclusiva durante a execução. Isto foi viabilizado por meio da criação de um ambiente paralelo exclusivo para estes experimentos, chamado "*hadoop*". Na submissão do *job* para execução, ou no próprio *script* do experimento, deve-se informar a quantidade desejada de processadores, por meio da opção *–pe* (*parallel environment*), e este número de processadores deve ser sempre um múltiplo de 8, para que todos os 8 núcleos de processamento de um nó sejam alocados para a execução. Como exemplo, se for utilizada a opção "*-pe hadoop 16*", isto quer dizer que foi solicitando 16 núcleos de processamento, que resultarão na alocação de 2 nós completos do cluster para a execução.

[9] https://github.com/dew-uff/vphadoop/tree/master/scripts/execucao-lncc/jobs

A versão de Java utilizada é a Oracle JDK 1.7.0_u7, 64 bits, que é carregada durante a fase de inicialização do *script* de execução por meio de um módulo disponibilizado no cluster, acessível por todos os seus nós. Esta versão do Java é compatível com todas as abordagens, assim como o SGBDX empregado e o *framework* Hadoop.

Como SGBDX, foi utilizado o BaseX (GRÜN; HOLUPIREK; SCHOLL, 2007), pois além de ser um software de código aberto, seu desenvolvimento e comunidade de usuários são bastante ativos, além de sua execução atender aos requisitos quanto ao uso da função *position()* (SILVA; MATTOSO; BRAGANHOLO, 2013). A versão utilizada nos experimentos é a 8.2.3. O BaseX foi instalado no diretório *home* compartilhado do usuário, mas na execução do *script*, na etapa de inicialização, os diretórios com os arquivos de dados são copiados para o diretório */tmp* de cada nó envolvido na execução, e cada nó é instruído a utilizar a sua cópia dos dados, tornando local o acesso a disco feito pelo SGBDX.

6.2 BASE DE DADOS E CONSULTAS

Como base de dados para a avaliação experimental, optou-se por gerar e utilizar bases de dados sintéticas, compostas por documentos XML criados com o auxílio da ferramenta disponibilizada pelo *benchmark* XMark (SCHMIDT *et al.*, 2002). A escolha deste *benchmark* se deu por dois motivos. O primeiro e principal deles é a facilidade em se gerar as bases de dados, tanto compostas por um único documento (*single document* - SD) quanto compostas por múltiplos documentos (*multiple document* - MD). Em segundo, esta é uma das bases de dados que foram utilizadas na avaliação do trabalho original de fragmentação virtual para o formato XML, o PartiX-VP, trabalho este que serviu de base para a proposição das abordagens VPHadoop e PartiX-EVP. Desta forma, ao utilizar a mesma base, torna-se possível a utilização das mesmas consultas aplicadas anteriormente na avaliação do PartiX-VP, permitindo comparar as abordagens. Entretanto, identificou-se que algumas consultas não podem ser empregadas nesta nova avaliação, pois o crescimento da base de dados impossibilita a sua execução (a quantidade de memória existente não é suficiente). Estas consultas foram modificadas com a aplicação de seleções e também com modificação do elemento resposta, para que a execução fosse viabilizada.

A base de dados do *benchmark* XMark é composta por um único documento, ou uma coleção deles, todos respeitando o esquema ilustrado na Figura 27. Este esquema foi modelado como uma base de dados de um sistema de leilões online. Várias entidades como pessoa (*person*), leilão aberto (*open_auction*), leilão fechado (*closed_auction*) estão

representadas, além de existir atributos que permitem expressar o relacionamento entre eles. Desta forma, esta base de dados permite realizar consultas com junções.

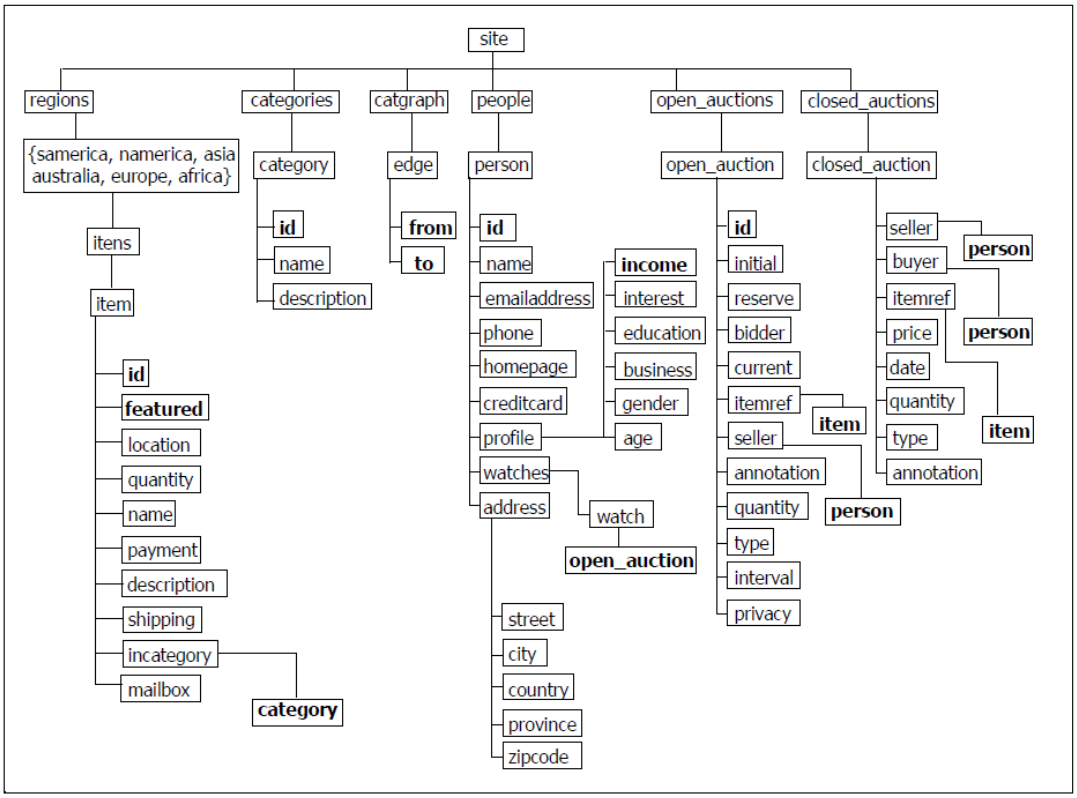


Figura 27. Esquema da base de dados do *benchmark* XMark (RODRIGUES, 2011)

A diferença principal entre a base empregada na avaliação experimental neste trabalho e a base da avaliação experimental do PartiX-VP é o tamanho. Na nova avaliação, optou-se por empregar bases de dados com um volume maior de dados, e para isso foram criadas bases de 1 GB e 10 GB, tanto SD quanto MD, para atender às necessidades de execução distribuída de consultas atuais. A base MD foi formada de modo a possuir o mesmo esquema da base SD, mas composta por 100 documentos. Com isso, foi possível avaliá-la com o mesmo conjunto de consultas utilizado na avaliação da base SD.

Quanto às consultas empregadas, optou-se por utilizar um conjunto que exercitasse diversas características da abordagem proposta: consultas simples, com junções, com caminhos incompletos, funções de agregação e ordenação. A Tabela 1 resume as principais características destas consultas. Na coluna “ID (Rodrigues et al. 2011)”, referencia-se a consulta daquele trabalho que representa esta consulta. O asterisco (“*”) nesta coluna representa que a consulta teve modificações de sua versão original, conforme já descrito. Todas as consultas estão disponíveis no ANEXO A. Cabe lembrar que as mesmas consultas foram utilizadas para avaliar a abordagem na execução sobre base de dados MD, sendo que ao

invés de ser utilizada a função *document()*, que acessa um documento, neste caso utilizou-se a função *collection()*, que acessa uma coleção de documentos.

Tabela 1. Características das consultas utilizadas na avaliação

Cons.	Cardinal. AF (1GB)	Junção	Cardinal. Junção (1GB)	Orden.	Agreg.	Tamanho saída (1GB) (bytes)	ID (RODRIGUES et al. 2011)
C1	10000	Sim	61.220.000	Não	Não	9.832.826	C3*
C2	10000	Sim	398.860.000	Não	Não	38.898.227	C5*
C3	10000	Sim	112.098	Não	Não	231.270	C6*
C4	255000	Não	-	Sim	Não	30.036.313	C8
C5	255000	Não	-	Não	Não	76.256.828	C9
C6	255000	Não	-	Não	Não	11.344.178	C10
C7	5500	Sim	36.952.500	Não	Não	36.437	C11
C8	120000	Não	-	Não	Não	163.660.889	C12*
C9	97500	Não	-	Não	Sim	127	-
C10	10000	Sim	2.550.000.000	Não	Não	1.195.436.660	C3
C11	20000	Sim	5.100.000.000	Não	Não	2.325.832.950	C5
C12	120000	Sim	14.672.445.000	Sim	Não	18.009.015	-

As consultas foram classificadas em 2 categorias diferentes, referentes a seu custo de execução: baixo e alto. Para essa classificação, foram considerados os tempos de execução considerando o cenário centralizado, com a base SD de 1GB de tamanho. As consultas de baixo custo não requerem grande esforço computacional para que seu resultado seja calculado, e o seu tempo de execução é bastante pequeno (< que 20 segundos). As consultas de alto custo, por outro lado, exigem grande esforço computacional. Neste trabalho, foram consideradas consultas de alto custo aquelas que demoraram 20 segundos ou mais para a execução no cenário escolhido. Na Tabela 1, as cores nos indicadores de consulta refletem a categoria da consulta: vermelho para consultas de alto custo, e verde para consultas de baixo custo.

6.3 PROTÓTIPOS

Tanto o protótipo desenvolvido para a abordagem VPHadoop quanto aquele para a abordagem PartiX-EVP tiveram como base o protótipo elaborado para o PartiX-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), e assim como este, foram desenvolvidos em Java. Conforme já detalhado na Seção 3.2, a implementação da técnica de fragmentação virtual do PartiX-VP foi isolada e encapsulada dentro de uma biblioteca, escrita em Java, e desta forma foi incorporada aos protótipos do VPHadoop e do Partix-EVP, já com as melhorias descritas na Seção 3.1.

Para a conexão com o SGBDX, os novos protótipos fazem uso da XQuery API, definida pela JSR 225[10]. A implementação desta API está disponível[11] para diversos SGBDX, na forma de bibliotecas. Foi utilizada a versão 1.4 da API XQJ para BaseX.

As Seções 6.3.1 e 6.3.2 apresentam detalhes específicos dos protótipos criados para as abordagens VPHadoop e PartiX-EVP, respectivamente. A Seção 6.3.3 descreve a adaptação que foi realizada no protótipo da abordagem PartiX-VP para possibilitar a sua execução no ambiente desta avaliação.

6.3.1 VPHADOOP

Conforme detalhado no Capítulo 4, devido ao fato de não serem utilizados dados de entrada oriundos do sistema de arquivos distribuído do Hadoop, foi necessário criar um módulo adaptador específico para o *job* que compõe essa nova abordagem. Isto é feito com a implementação das interfaces *InputFormat*, *InputSplit* e *RecordReader*, que são utilizadas pelo *framework* para realizar a leitura dos dados.

Um objeto que implementa a interface *InputFormat* determina a maneira como o conjunto total de dados do *job* é repartido em unidades menores, para processamento. O *framework* Hadoop o utiliza para obter a lista de blocos (objetos do tipo *InputSplit*) que necessita ser processada, sendo que cada objeto desta lista representa um intervalo dos dados de entrada. Um objeto que implementa a interface *RecordReader* é utilizado pelo *framework* durante o processamento de uma tarefa para saber como ler os diversos registros do intervalo de dados determinado por um *InputSplit*.

No caso do protótipo do VPHadoop, a interface *InputFormat* foi criada de maneira que internamente execute o algoritmo de fragmentação virtual e determine as subconsultas necessárias para a aplicação desta técnica na execução da consulta. Cada subconsulta atua sobre um subintervalo determinado da base de dados de entrada. Deste modo, a implementação da classe *InputSplit* foi feita de forma a encapsular um número configurável de subconsultas, que serão posteriormente lidas como registros pelo *RecordReader*. Assim, a lista de *InputSplit* é formada no *InputFormat*, que a repassa ao *framework* e o *job* está pronto para começar. Cabe mencionar que, ao compor um objeto do tipo *InputSplit* com um número configurável de subconsultas, tem-se o cuidado de tomar subconsultas de forma aleatória, que atuem sobre intervalos não contíguos. Caso contrário, não se mitigaria o problema do desbalanceamento de carga. Um intervalo de dados que possui uma densidade maior de itens

[10] https://jcp.org/en/jsr/detail?id=225

[11] http://xqj.net/

relevantes para a consulta em questão, mesmo que dividido pela técnica de fragmentação virtual, ainda continuaria sendo processado em uma mesma tarefa caso a contiguidade não fosse quebrada, o que resultaria no mesmo desbalanceamento de carga.

Uma característica importante do *framework* Hadoop é que ele considera a localidade dos dados ao alocar as tarefas para execução nos *TaskTrackers*. Com as informações de localização dos dados de entrada, respeitando os limites de execução de cada nó, ele tenta alocar a tarefa que vai atuar sobre um intervalo de dados a um nó que já possua uma cópia daquele intervalo de dados em seu armazenamento local, com o objetivo de minimizar as transferências de dados. Contudo, na abordagem VPHadoop, com a utilização da réplica total da base de dados em cada nó, pode-se considerar que todos os intervalos de dados determinados durante a fragmentação virtual possuem localidade ótima para o *framework*, ou seja, nunca é necessário realizar transferências entre nós para iniciar a execução.

O protótipo do VPHadoop necessita receber como entrada uma série de parâmetros de configuração. O *framework* Hadoop já possui um mecanismo de configuração de *jobs* por arquivos XML seguindo um formato específico. Para a nova implementação, este mecanismo foi também utilizado para definir os parâmetros exclusivos desta nova abordagem. Desta forma, a configuração do VPHadoop é feita passando-se um arquivo XML contendo os parâmetros da abordagem para o *framework*, no envio do *job*. Um exemplo de arquivo de configuração é apresentado no ANEXO B.

Foi preciso trabalhar junto à equipe de suporte do LNCC para que fosse possível configurar o *framework* Hadoop para ser executado no cluster, em modo usuário. Por meio de pesquisas, foi identificado que outros usuários já dedicaram esforços nesta configuração e desenvolveram scripts que automatizam o processo. Um conjunto destes *scripts*, intitulado myHadoop[12], versão 0.30, foi utilizado. Desta forma foi possível instalar o Hadoop no diretório *home* compartilhado, mas durante a inicialização do *framework*, os serviços do Hadoop presentes em cada nó são instruídos a utilizar os discos locais (diretório "*/tmp*") para o processamento. Como já mencionado, a versão do Hadoop utilizada neste trabalho é a 1.2.1, mas o Hadoop atualmente possui uma nova família de versões, intitulada HadoopV2. A utilização desta nova versão não agrega nenhum benefício para a abordagem e por isso optou-se por continuar com a versão 1.2.1, ainda ativa e mantida pela comunidade. Devido à compatibilidade retroativa entre as versões, a migração para utilização da nova versão deve poder ser feita sem maiores dificuldades.

[12] http://www.glennklockwood.com/data-intensive/hadoop/on-hpc.html

É importante mencionar que todas as execuções com o protótipo da abordagem VPHadoop utilizaram o Hadoop configurado para rodar até 8 tarefas simultâneas por nó do cluster. Tal valor foi selecionado devido à presença de 8 núcleos de processamento em cada nó. Quanto ao SGBDX, somente uma instância é utilizada por nó. Contudo, o SGBDX BaseX, escolhido para as execuções desta avaliação, possui paralelismo interconsultas e, portanto, suporta a execução simultânea de várias subconsultas.

6.3.2 PARTIX-EVP

O protótipo da abordagem PartiX-EVP utiliza a biblioteca MPJ Express (SHAFI; CARPENTER; BAKER, 2009), versão 0.44, para possibilitar ao nó mediador distribuir subconsultas para execução nos diversos processadores e controlar cada execução. Esta versão inclui suporte para a execução de aplicações criadas com MPJ em um cluster MPI nativo, o que é o caso do cluster SUNHPC, utilizado nesta avaliação.

Como característica do *framework* MPI, o mesmo executável é distribuído a todos os processadores, e as distinções entre as execuções em cada um, quando necessárias, são feitas por meio do *rank*, um identificador do processador. No caso, a única distinção feita na implementação do PartiX-EVP está no processador de rank “0”, ao qual é atribuído o papel de mediador. Os demais processadores são utilizados como nós trabalhadores. É importante mencionar que, para esta avaliação experimental, o conceito de nó processador descrito no Capítulo 6 é mapeado para um núcleo de processamento do cluster SUNHPC. Com isso, durante a execução, cada nó do cluster se transforma em 8 nós processadores da abordagem. Portanto, ao se mencionar uma execução que envolva 32 processadores, houve a alocação de 4 nós no cluster.

Assim como na abordagem VPHadoop, na utilização da abordagem PartiX-EVP no cluster desta avaliação foi empregado somente uma instância do SGBDX BaseX por nó, compartilhada entre os 8 processadores que executam no mesmo nó. Esta instância do SGBDX BaseX utiliza uma cópia da base de dados, armazenada no disco local.

6.3.3 PARTIX-VP

Como já mencionado, o protótipo da abordagem PartiX-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) foi adaptado para também utilizar a biblioteca de fragmentação virtual criada por este trabalho, e assim, usufruir das melhorias implementadas, tornando factível a sua comparação com as demais abordagens deste trabalho. Durante esta adaptação, a versão da biblioteca MPJ Express utilizada anteriormente foi substituída pela

nova versão 0.44, o que permitiu executar esta abordagem utilizando a infraestrutura MPI nativa do cluster SUNHPC.

Durante as execuções com este protótipo, o conceito de nó processador também está mapeado para um núcleo de processamento de um nó do cluster SUNHPC, de maneira que um mesmo nó do cluster executa 8 nós processadores da abordagem, devido à presença de 8 núcleos. De maneira similar ao das outras abordagens, todos os 8 nós processadores executando em um mesmo nó do cluster compartilham a mesma instância de SGBDX BaseX, com sua base de dados escrita no disco local do nó.

6.4 METODOLOGIA

Sendo um dos objetivos principais deste trabalho a análise do comportamento da fragmentação virtual com a variação no número de fragmentos, a avaliação experimental se inicia com a execução de consultas predeterminadas empregando as duas abordagens propostas por este trabalho, VPHadoop e PartiX-EVP. Nestas execuções, fixa-se todas as variáveis presentes no ambiente e nas abordagens, exceto o número de fragmentos empregado, e analisa-se o tempo total de execução.

O tempo total de execução (TTE) é composto pela combinação de três medições: tempo de execução da fragmentação virtual (TFV), o tempo da execução em paralelo das subconsultas (TES), e o tempo de composição final (TCF), sendo que neste tempo de composição final já está considerado o tempo de escrita do arquivo final no disco local:

$$TTE = TFV + TES + TCF$$

No PartiX-EVP, os tempos de execução acima descritos são todos medidos durante a própria execução e capturados por meio de mensagens de *log* impressas em arquivos. No caso do VPHadoop, foi necessário combinar mensagens de *logs* da aplicação desenvolvida com mensagens de log do *framework* para calcular os tempos, visto que grande parte da execução é controlada e executada por ele. Ao término de cada *job*, o *framework* emite um relatório final contendo *timestamps* de início e término de execução de cada tarefa executada, sendo possível determinar os tempos de execução desta forma.

Muito embora os computadores utilizados nos experimentos tenham alocação exclusiva durante a execução, é possível que alguns fatores, como a execução de serviços de sistema operacional, possam influenciar o tempo de execução das consultas. Desta forma, não é prudente considerar somente o tempo de uma execução da consulta, mas sim uma média aritmética de várias execuções. Neste trabalho utilizou-se estratégia semelhante àquela adotada por Lima (2004), e cada execução é repetida no mínimo por 3 vezes. Ao final, toma-

se a média dos tempos obtidos. O número de repetições considerado é maior que 3 nos casos em que o coeficiente de variação obtido com 3 execuções superou 20%. O coeficiente de variação é definido como a razão entre o desvio padrão e a média (EVERITT, 1998).

Além da análise do comportamento da técnica de fragmentação virtual, os resultados desta primeira fase da avaliação experimental são utilizados em um processo de eleição para se determinar o número de fragmentos ótimo para ser utilizado em cada classe de consultas (baixo custo e alto custo), nas abordagens VPHadoop e PartiX-EVP. Isto é necessário pois é infactível realizar a mesma análise de número de fragmentos em todos os cenários previstos na avaliação (bases SD e MD, 1GB e 10GB). Toma-se o número de fragmentos que proporcionou o melhor tempo de execução para cada subconsulta, contabiliza-se os votos para cada classe de consultas e, ao final, determina-se qual o melhor número de fragmentos para cada classe. Este número é dividido pelo número de processadores utilizado, o que fornece um fator de multiplicação que determina o número de fragmentos ideal. Este fator de multiplicação é então empregado a cada execução destas duas abordagens durante a próxima fase.

Com a análise do número de fragmentos concluída, realiza-se então a comparação entre os tempos de execução de consultas predeterminadas, obtidos com: (i) abordagem centralizada, utilizando somente uma instância de SGBDX e o próprio cliente do SGBDX para executar a consulta; (ii) abordagem distribuída PartiX-VP, proposta original da técnica da fragmentação virtual simples, utilizando o protótipo adaptado para a execução no ambiente definido; (iii) abordagem distribuída VPHadoop, proposta deste trabalho, utilizando o protótipo criado; e (iv) abordagem distribuída PartiX-EVP, utilizando o protótipo também oriundo deste trabalho.

A comparação das abordagens VPHadoop e PartiX-EVP com a abordagem centralizada indica se houve ganhos com a distribuição da execução da consulta. Já a comparação entre as execuções das abordagens distribuídas serve para verificar se as abordagens propostas são capazes de diminuir o tempo de execução das consultas, considerando-se que a execução é realizada no mesmo ambiente. Por fim, a comparação entre as duas abordagens propostas por este trabalho, VPHadoop e PartiX-EVP, indica se, de fato, o *framework* Hadoop adiciona processamento extra durante a execução das consultas, impactando o tempo de execução.

Para a execução das consultas na abordagem centralizada, utilizou-se o próprio aplicativo cliente disponibilizado pelo SGBDX, sendo que o tempo de execução considerado foi capturado pelo utilitário *time* do Linux. Na chamada para execução, instrui-se o cliente a

escrever o resultado da consulta em um arquivo no disco local, e, portanto, o tempo desta escrita também foi considerado no tempo total de execução.

Com os *logs* emitidos pelas abordagens distribuídas também é possível calcular a utilização dos processadores envolvidos na etapa de execução em paralelo das subconsultas. No caso do PartiX-VP, tem-se o *timestamp* do término de execução de cada subconsulta. Considerando que no momento inicial todos os processadores envolvidos são ativados, consegue-se estabelecer de forma cronológica a utilização de processadores pela abordagem. No caso do VPHadoop e do PartiX-EVP, ao se contabilizar a utilização de um processador no início da execução de uma tarefa, e decrementar essa utilização em seu término, também é possível estabelecer uma cronologia de utilização dos processadores. Assim, propõe-se também comparar a utilização dos processadores entre estas duas abordagens, permitindo avaliar o balanceamento de carga.

Com relação ao número de processadores envolvidos na avalição experimental das abordagens de execução distribuída, foram considerados cenários com 16, 32, e 64 processadores, para se avaliar o impacto no tempo ao se alocar mais poder computacional à execução. Como cada nó do cluster SUNHPC possui 8 processadores, cada execução utilizava um mínimo de 2 e um máximo de 8 nós do cluster.

6.5 ANÁLISE DO NÚMERO DE FRAGMENTOS

Tanto a abordagem VPHadoop quanto a PartiX-EVP permitem configurar o número de fragmentos utilizados na fragmentação virtual simples. Desta forma, foi feita uma análise do impacto do número de fragmentos no tempo de execução de consultas, em ambas as abordagens. As Seções 6.5.1 e 6.5.2 abaixo apresentam as análises para cada uma das abordagens.

Durante esta fase de análise do número de fragmentos, notou-se que as execuções de algumas consultas de alto custo possuíam um comportamento bastante discrepante das demais consultas da mesma classe, e diferente também do comportamento esperado. Tal fato deu origem a uma análise aprofundada da execução distribuída destas consultas, e os resultados foram utilizados para determinar as consultas que continuariam no experimento. Desta forma, julga-se importante detalhar esta análise, e ela consta na Seção 6.5.3.

6.5.1 VPHADOOP

Na abordagem VPHadoop, o número de fragmentos que é utilizado na execução da técnica de fragmentação virtual é mapeado para as divisões de dados utilizadas pelo Hadoop:

os *splits* e os *records*. Como apresentado no Capítulo 4, o VPHadoop divide as subconsultas resultantes da fragmentação virtual em *splits* que, por sua vez, são compostos por *records*. Cada subconsulta (e, portanto, cada fragmento) é mapeado para um *record*, e um *split* pode conter um ou mais *records*. Cada *split* é processado por uma tarefa do tipo *map*, que é a unidade de distribuição de trabalho nesta abordagem. Enquanto um número maior de *records* dentro de um *split* pode fazer com que o processamento extra de inicialização de tarefas do tipo *map* possa ser evitado, o fato de se ter menos *splits* pode acarretar uma distribuição não uniforme de trabalho entre os processadores, já que é o número de *splits* (que dita o número de tarefas do tipo *map*) que determina o número de unidades de trabalho distribuíveis pelo *framework*.

Como é inviável executar todas as consultas em todos os cenários variando-se os números de *splits* e *records* utilizados, optou-se por realizar um experimento mais extenso de análise do tempo de execução para o cenário com base de dados SD de 1 GB, utilizando 32 processadores, realizando algumas variações nestes parâmetros. Estas características da base de dados e de número de processadores foram escolhidas, principalmente, para minimizar o tempo total necessário à execução desta análise. Para o número de fragmentos, optou-se por utilizar múltiplos do número de processadores, 1x, 2x, 4x e 8x, o que resultou em cenários com 32, 64, 128 e 256 fragmentos, respectivamente. Esta escolha se deu pois testes preliminares indicaram que os tempos de execução atingem os menores valores quando o número de fragmentos utilizados é igual a um múltiplo do número de nós trabalhadores. Para cada cenário, repetiu-se a execução distribuindo os fragmentos em números de *splits* e *records* possíveis. Para o caso com 32 fragmentos, o único cenário possível que utiliza todos os processadores é aquele com 32 *splits*, sendo que cada *split* possui somente 1 *record*. Já para o cenário com 64 fragmentos, são dois os cenários possíveis. O primeiro, com 64 *splits* de 1 *record* cada, e o segundo, com 32 *splits*, com 2 *records* cada. Os casos com 128 e 256 fragmentos seguiram o mesmo procedimento, sempre respeitando o limite mínimo de 32 *splits*, para que se tenha a utilização de todos os processadores.

Os resultados estão apresentados na Tabela 2 e Tabela 3. As cores nos indicadores de consulta refletem a categoria da consulta, conforme já apresentado na Seção 6.2. O menor TTE para cada consulta está marcado em azul. A Figura 28 apresenta os TTE normalizados de cada consulta nos cenários com 128 fragmentos e 256 fragmentos. Nesta figura, no eixo horizontal, o número que segue a letra “F” indica o número de fragmentos, o número após a letra “S” indica a quantidade de *splits*, e o número após a letra “R” indica o número de *records* utilizado. As consultas de alto custo estão apresentadas com linhas pontilhadas, para

facilitar a diferenciação. Para o cenário com 128 fragmentos, foram utilizadas 3 combinações de *splits* e *records* diferentes, aumentando em cada uma o número de *splits* (e, consequentemente, diminuindo o número de *records*). Já no cenário com 256 fragmentos, foi possível utilizar 4 combinações diferentes. A normalização do tempo de execução de cada consulta se deu com base no seu maior tempo de execução obtido dentre todos os cenários testados (32, 64, 128 e 256 fragmentos), de maneira que se pudesse comparar os comportamentos independente do tempo total de consulta.

Tabela 2. TTE para análise de fragmentos com o VPHadoop C1 a C6 (em s)

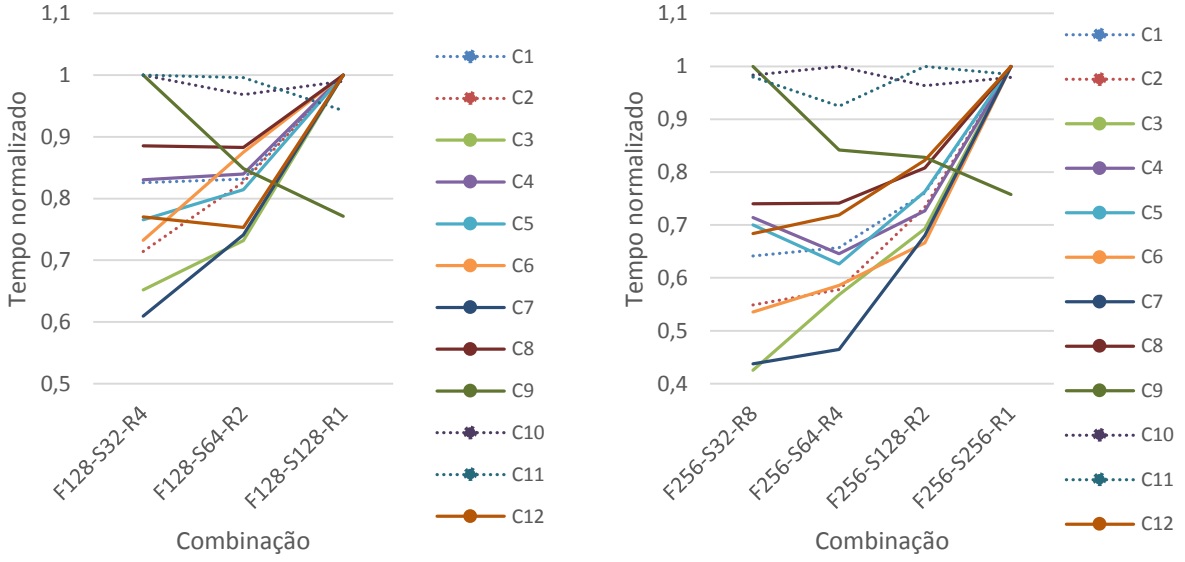
# Frag.	*Splits*	*Records*	C1	C2	C3	C4	C5	C6
32	**32**	**1**	33,194	26,944	22,693	36,943	33,684	24,038
64	**32**	**2**	30,989	26,289	23,536	36,806	36,41	25,256
64	**64**	**1**	34,426	33,155	22,166	41,858	41,876	29,499
128	**32**	**4**	33,783	28,028	22,461	46,001	41,082	29,416
128	**64**	**2**	34,017	32,453	25,212	46,512	43,714	35,15
128	**128**	**1**	40,916	39,244	34,445	55,389	53,672	40,165
256	**32**	**8**	37,25	33,122	23,394	61,093	55,856	38,13
256	**64**	**4**	38,156	34,864	31,253	55,266	49,967	41,699
256	**128**	**2**	44,166	44,315	38,104	62,218	60,871	47,454
256	**256**	**1**	58,064	60,353	55,016	85,592	79,771	71,224

Tabela 3. TTE para análise de fragmentos com o VPHadoop C7 a C12 (em s)

# Frag	*Splits*	*Records*	C7	C8	C9	C10	C11	C12
32	**32**	**1**	21,496	43,357	861,098	183,141	353,557	32,703
64	**32**	**2**	22,666	46,505	1739,212	179,611	359,111	31,023
64	**64**	**1**	28,265	48,816	1419,487	191,711	369,109	35,619
128	**32**	**4**	23,812	53,16	3438,178	186,701	354,585	35,258
128	**64**	**2**	28,948	53,003	2917,843	180,806	353,178	34,461
128	**128**	**1**	39,07	60,056	2652,406	184,865	334,145	45,771
256	**32**	**8**	30,023	68,306	6574,842	192,745	366,193	45,347
256	**64**	**4**	31,879	68,407	5533,103	196,038	345,573	47,681
256	**128**	**2**	46,657	74,547	5442,558	188,826	373,684	54,537
256	**256**	**1**	68,61	92,303	4979,702	191,887	367,945	66,323

É possível notar que quase todas as consultas de baixo custo (C3-C8 e C12) tendem a apresentar uma degradação no tempo de execução com o aumento do número de *splits*. Isto ocorre pois o processamento extra necessário nestes cenários não é insignificante. Quanto menos tarefas forem executadas, melhor. Contudo, consultas com um custo alto de processamento como a C10 e a C11 apresentam um menor tempo de execução quando o número de tarefas (relacionado ao número de *splits*) cresce. Isto porque o *framework* consegue se beneficiar da melhor distribuição do trabalho, e o tempo acrescido para esta

tarefa não é relevante. Foram exceções para este comportamento as consultas C1 e C2, também consideradas de alto custo segundo o critério adotado. Nesta análise, estas consultas se comportaram como consultas de baixo custo. De fato, ao olhar para o tempo de execução destas consultas na Tabela 2, vê-se que eles se aproximam muito dos tempos das consultas de baixo custo, sendo bastante diferentes dos tempos para as consultas C10 e C11. A consulta C9, por sua vez, é uma consulta de baixo custo, mas os tempos apresentados para a sua execução com a fragmentação virtual são bastante elevados. Cabe lembrar que a classificação das consultas em baixo ou alto custo leva em consideração o seu tempo de execução para o caso centralizado. No caso distribuído, a execução das subconsultas no SGBDX pode não se beneficiar das mesmas otimizações que o caso centralizado.



(a) **(b)**

Figura 28. Análise do impacto do número de *splits* e *records* nos TTE para (a) 128 fragmentos e (b) 256 fragmentos

Um outro ponto interessante desta análise do número de fragmentos, *splits* e *records* é verificar se um aumento no número de fragmentos realmente contribui para diminuir o tempo de execução. A Figura 29 apresenta os TTE normalizados das consultas, considerando somente os casos em que se tem 1 *record* por *split*, ou seja, cada fragmento é processado por uma tarefa diferente. O resultado mostra que, novamente, para consultas de baixo custo (C3, C4, C5, C6, C7, C8, C9 e C12), há um incremento no tempo de execução conforme o número de fragmentos aumenta. Contudo, as consultas de alto custo C10 e C11 apresentam os seus melhores tempos de execução em cenários que não são aqueles com o menor número de fragmentos. As consultas C1 e C2, apesar de serem classificadas como consultas de alto custo, apresentam comportamento similar ao daquelas de baixo custo. Ao observar os tempos de

execução, verifica-se que ele muito se assemelha aos tempos de execução das consultas de baixo custo.

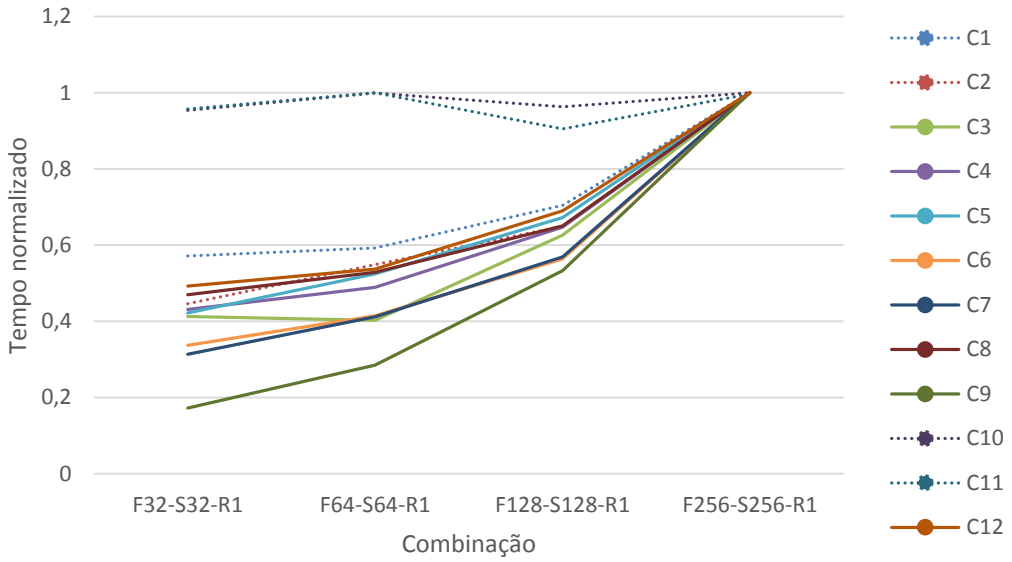


Figura 29. Análise do incremento no número de fragmentos no tempo de execução

Como o número de execuções de cada consulta para este tipo de análise é grande, não é factível realizá-la para os demais cenários, variando o número de processadores e com bases diferentes. Assim, tomou-se o comportamento observado no cenário com 32 processadores e a base de dados SD de 1GB como padrão, e realizou-se um método de eleição para definir qual seria a combinação de *splits* e *records* que seria utilizada nos demais cenários. O processo de eleição consiste em pontuar as combinações que produziram o menor tempo de execução para cada consulta. Para cada categoria de consulta, é então eleita a combinação que tenha tido mais pontos, considerando as consultas que a compõem. Os resultados desta eleição estão resumidos na Tabela 4. É importante mencionar que, para efeitos de execução nos demais cenários, não se considera o número de fragmentos em questão, mas sim o multiplicador sobre o número de processadores que levou àquela combinação. Os multiplicadores também estão indicados na Tabela 4.

Tabela 4. Eleição da combinação de fragmentos, *splits* e *records*

Custo da Consulta	Combinação eleita
Baixo	32 fragmentos, 32 *splits* e 1 *record* por *split* (x1 para fragmentos e x1 para *splits*)
Alto	64 fragmentos, 32 *splits* e 2 *records* por *split* (x2 para fragmentos e x1 para *splits*)

6.5.2 PARTIX-EVP

A abordagem Partix-EVP também possui como parâmetro o número de fragmentos que deve ser gerado pela fragmentação virtual. Como explicado no Capítulo 6, a variação deste número de fragmentos, que nesta abordagem pode ir além do número de nós trabalhadores existentes, pode causar um impacto positivo ou negativo no tempo total de execução de uma consulta. Nesta Seção, descreve-se uma análise deste comportamento. Como é inviável executar todas as consultas em todas as combinações possíveis de cenários e parâmetros, os resultados desta análise são utilizados para definir a fragmentação a ser utilizada nas execuções de consultas nos demais cenários.

Alguns testes indicaram que os tempos de execução atingem utilização ótima da capacidade de processamento do cluster quando o número de fragmentos utilizados é igual a um múltiplo do número de nós trabalhadores. Desta forma, a análise do número de fragmentos para a abordagem PartiX-EVP considerou somente múltiplos do número de processadores trabalhadores envolvidos, mais especificamente, os multiplicadores 1x, 2x, 4x e 8x. Cabe lembrar que o PartiX-EVP faz uso permanente de um processador para servir de mediador e, desta forma, o número de processadores trabalhadores na execução paralela é igual ao número de processadores alocados decrescido de 1.

Para esta análise, optou-se por trabalhar com 16 processadores (2 nós do cluster), com a base de dados SD de 1 GB. Esta escolha ocorreu devido ao tempo necessário para execução, já que existe a limitação no número de processadores simultâneos alocados a cada usuário do cluster SunHPC.

Os resultados obtidos estão resumidos na Tabela 5 e Tabela 6. As cores nos indicadores de consulta refletem a categoria da consulta, conforme já apresentado na Seção 6.2. O menor tempo de execução para cada consulta está marcado em azul.

Tabela 5. Tempos para análise de fragmentos com o PartiX-EVP (em seg.)

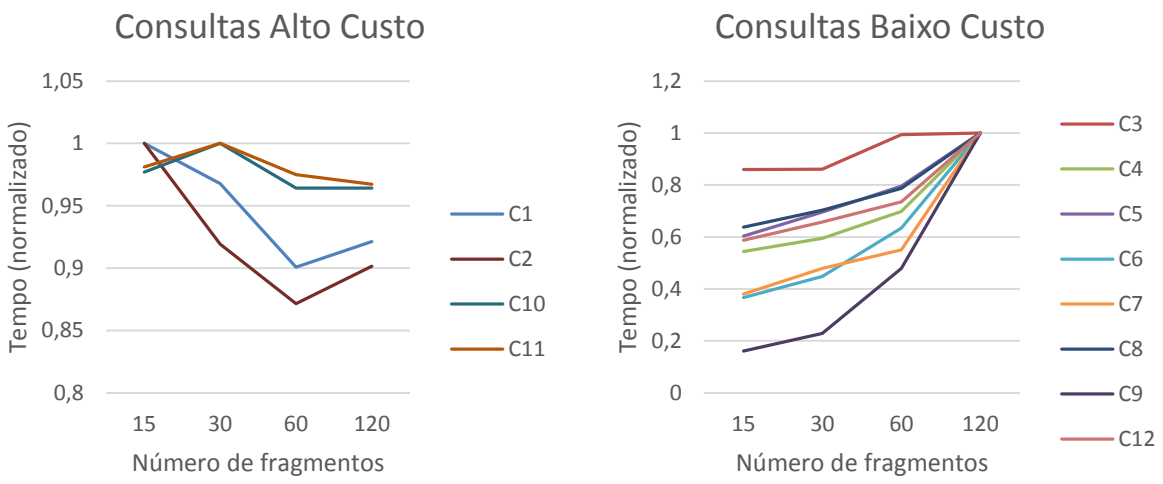
# Fragmentos	C1	C2	C3	C4	C5	C6
15	40,362	27,820	9,165	16,116	16,507	6,207
30	39,059	25,573	9,178	17,630	19,037	7,584
60	36,355	24,244	10,607	20,684	21,793	10,734
120	37,188	25,080	10,669	29,617	27,386	16,927

É possível notar que as consultas C1, C2, C10 e C11 obtêm ganhos no tempo de execução ao se utilizar um número de fragmentos maior que o número de processadores trabalhadores. De uma maneira geral, as consultas de baixo custo não se beneficiam do aumento no número de fragmentos, pois o processamento extra adicionado afeta o tempo total

de execução, que é relativamente baixo. Já as consultas de alto custo tendem a apresentar uma melhora com o aumento no número de fragmentos.

Tabela 6. Tempos para análise de fragmentos com o Partix-EVP (em seg.)

# Fragmentos	C7	C8	C9	C10	C11	C12
15	0,896	24,482	560,597	242,186	490,252	19,373
30	1,127	26,996	794,141	247,888	499,670	21,690
60	1,296	30,200	1664,774	239,001	487,172	24,222
120	2,356	38,401	3475,910	239,043	483,261	32,966



(a) **(b)**

Figura 30. TTE normalizados para as consultas de alto (a) e baixo (b) custo

Com os resultados da Tabela 5 e da Tabela 6, foi feita uma eleição do melhor número de fragmentos para cada categoria de consulta. Os resultados desta eleição estão lançados na Tabela 7.

Tabela 7. Resultado da análise do número de fragmentos para o PartiX-EVP

Baixo custo	15 fragmentos (1x # processadores trabalhadores)
Alto custo	60 fragmentos (4x # processadores trabalhadores)

6.5.3 CONSULTAS DE ALTO CUSTO

Observou-se que, nos casos de algumas consultas extremamente custosas (tempo de processamento > 3 horas), o comportamento diante do aumento no número de fragmentos na técnica de fragmentação virtual é o inverso daquele esperado e observado para outras consultas da mesma categoria. Em geral, fixado um número de processadores, espera-se que o tempo de execução de consultas de alto custo diminua com o aumento do número de fragmentos, mas, ao invés disso, algumas consultas apresentaram aumentos significativos no

tempo de execução. Diante disto, iniciou-se uma investigação mais profunda para analisar este comportamento.

A consulta C2+, constante na Figura 31, é uma variante da consulta C2 utilizada nos demais experimentos. A diferença entre elas está na ordem das cláusulas *for*. Desta forma, a ordem dos elementos no documento resposta muda, mas o tamanho da resposta é exatamente o mesmo, visto que são os mesmos elementos que atendem às condições impostas na cláusula *where*.

```
<results> {
    for $pe in doc('auction.xml')/site/people/person
    for $cat in doc('auction.xml')/site/categories/category
    where count($pe/profile/interest) > 3 and $pe/profile/interest/@category = $cat/@id
    return
        <match>
            <person>{$pe/name}</person>
            <category>{$cat/name}</category>
        </match>
} </results>
```

Figura 31. Consulta C2+, variante da consulta C2

A Tabela 8 apresenta os TTE para as consultas C2 e C2+. A primeira informação que se obtém ao analisar os tempos ali descritos é que a consulta C2+, muito embora também seja considerada uma consulta de alto custo, apresenta o seu melhor TTE com o menor número de fragmentos, comportamento oposto ao da consulta C2 (e de outras consultas de alto custo). Além disso, percebe-se que, muito embora as consultas C2 e C2+ possuem resultado com o mesmo tamanho, o tempo de execução da C2+ é mais de 400 vezes maior que o tempo de execução da consulta C2.

Tabela 8. TTE para as consultas C2 e C2+ (em seg.)

Consulta	Número de fragmentos			
	15	30	60	120
C2	27,820	25,573	24,244	25,080
C2+	11838,36	15327,429	18307,313	28962,475

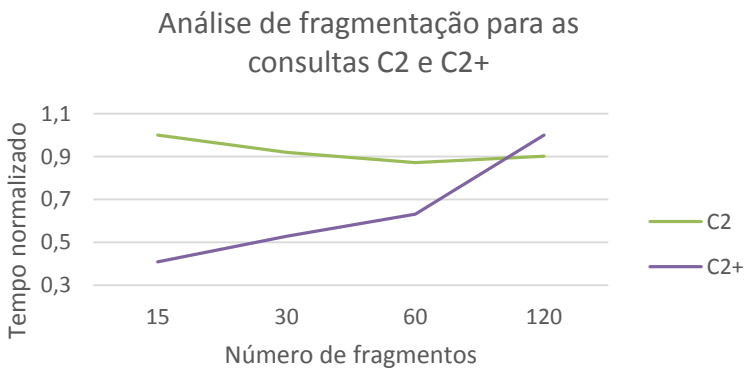


Figura 32. Comportamento do tempo de execução na análise do número de fragmentos

Buscou-se então analisar o processamento de uma subconsulta dentro do SGBDX BaseX. Neste ponto, observou-se que existe uma diferença na otimização de consulta realizada pelo processador XQuery do SGBDX BaseX para as subconsultas resultantes da fragmentação virtual nas consultas C2 e C2+. A Figura 33 mostra a subconsulta C2 (a) e a consulta resultante da otimização do SGBDX BaseX (b). Percebe-se que o SGBDX BaseX consegue aplicar um índice sobre os valores dos atributos para realizar a junção da consulta (marcado em negrito na Figura 33(b)), o que causa uma redução significativa do tempo de execução. Já para a consulta C2+, mostrada na Figura 34, muito embora ela possua estrutura bastante semelhante àquela da consulta C2, o SGBDX BaseX não consegue aplicar os índices que possui. Na consulta C2+, os valores que são atribuídos à variável na segunda cláusula *for* ficam limitados ao filtro de seleção posicional, adicionado pela fragmentação virtual. Logo, não é possível reescrever este caminho utilizando índices. Já na consulta C2, a segunda cláusula *for* não possui nenhuma limitação posicional. Sem utilizar índices, as instâncias do SGBDX BaseX acabam por operar, para cada subconsulta, sobre todo o intervalo de elementos "*person*" que possuem pelo menos 4 interesses registrados, e abrindo, para cada um deles, sequencialmente, todas as categorias que estão no intervalo determinado pela fragmentação virtual. Conforme o número de subconsultas aumenta (decorrente do número de fragmentos gerados na fragmentação virtual), o tempo de execução tende a aumentar. Neste ponto, cabe lembrar que, ao procurar por candidatos a atributo de fragmentação em consultas que possuam junção, a atual implementação do algoritmo da fragmentação virtual escolhe sempre aquele que possui menor cardinalidade. A partir desta análise, consultas como a C2+ foram desconsideradas para a avaliação deste trabalho.

(a)

```
for $cat in doc('expdb/auction.xml')/site/categories/category[position() >= 1 and position() < 334]
for $pe in doc('expdb/auction.xml')/site/people/person
where count($pe/profile/interest) > 3 and $pe/profile/interest/@category = $cat/@id
return
   <match>
      <person>{$pe/name}</person>
      <category>{$cat/name}</category>
   </match>
```

(b)

```
for $cat_0 in db:open-pre("expdb",0)/*:site/*:categories/*:category[position() >= 1 and position() < 334]
for $pe_1 in db:attribute("expdb", $cat_0/@*:id)/self::*:category/parent::*:interest/parent::*:profile/parent::*:person[3.0 <
count(*:profile/*:interest)]
return element match { (element person { ($pe_1/*:name) }, element category { ($cat_0/*:name) }) }
```

Figura 33. Subconsulta de C2 (a) e otimização realizada pelo SGBDX BaseX (b)

(a)

```
for $pe in doc('expdb/auction.xml')/site/people/person
for $cat in doc('expdb/auction.xml')/site/categories/category[position() >= 1 and position() < 334]
where count($pe/profile/interest) > 3 and $pe/profile/interest/@category = $cat/@id
return
      <match>
         <person>{$pe/name}</person>
         <category>{$cat/name}</category>
      </match>
```

(b)

```
for $pe_0 in db:open-pre("expdb",0)/*:site/*:people/*:person[3.0 < count(*:profile/*:interest)]
for $cat_1 in db:open-pre("expdb",0)/*:site/*:categories/*:category[position() = 1 to 333]
                                                    [(@*:id = $pe_0/*:profile/*:interest/@*:category)]
return element match { (element person { ($pe_0/*:name) }, element category { ($cat_1/*:name) }) }
```

Figura 34. Subconsulta de C2+ (a) e otimização realizada pelo SGBDX BaseX (b)

6.6 COMPARAÇÃO ENTRE ABORDAGENS

Todas as abordagens foram executadas utilizando *scripts* que descrevem uma execução no cluster SUNHPC. Estes *scripts* possuem basicamente três seções: inicialização, execução da consulta e finalização. Durante a inicialização, os diretórios que armazenam os logs são criados e as instâncias do SGBDX BaseX são inicializadas em cada nó envolvido na execução, considerando a base de dados que é indicada por um parâmetro. No caso específico da abordagem VPHadoop, além do SGBDX, é também inicializado o *framework* Hadoop.

A execução das consultas depende de variáveis que foram configuradas ao se criar o *job* de execução, que são: os identificadores das consultas a serem executadas e o número de repetições que cada execução deve realizar. Além disso, as abordagens VPHadoop e PartiX-EVP também possuem parâmetros próprios utilizados nesta fase. No caso do VPHadoop, o número de *splits* e número de *records* são parâmetros necessários. Para o PartiX-EVP, o parâmetro adicional é o número de fragmentos aplicado. Na fase de finalização, ocorre a parada de todas as instâncias de SGBDX, e apagam-se arquivos temporários criados. No caso do VPHadoop, para-se também o *framework* Hadoop. Os scripts também possuem tratamento de erros e outros parâmetros, que não possuem relevância para a execução e, portanto, não são detalhados neste trabalho.

Todas as tabelas apresentadas nesta Seção apresentam os TTE em segundos das abordagens para cada uma das bases de dados consideradas: 1 GB SD, 1 GB MD, 10 GB SD e 10 GB MD. Nos casos em que houve algum erro durante a execução, tal fato é apontado com a letra “E” ou “T” no lugar do tempo. A letra “E” refere-se a casos em que houve um erro na execução devido à falta de memória. A letra “T”, por sua vez, aponta casos em que a execução ultrapassou 2 dias (172.800 segundos) sem o retorno da resposta. Este tempo limite foi determinado empiricamente, após se constatar que a maioria das execuções que ultrapassavam essa marca acabavam por apresentar erros de memória posteriormente. O

limite também era necessário para que se pudesse ter o controle sobre o tempo necessário para todas as execuções. Nota-se ainda nestas tabelas que, no caso das abordagens distribuídas, os tempos estão agrupados por número de processadores utilizados na execução, para facilitar a comparação. As cores utilizadas junto ao identificador de cada consulta informam a qual categoria de consulta ela pertence (baixo custo em verde, alto custo em vermelho).

A Tabela 9 e a Tabela 10 apresentam os TTE para todas as execuções que utilizam a base de dados SD de 1 GB, e a base de dados MD de 1 GB, respectivamente. A partir da Tabela 9, uma primeira análise que se pode realizar sobre estes resultados é verificar o ganho obtido com a distribuição da execução das consultas. A Figura 35 apresenta esta comparação de tempos, onde, para cada consulta, toma-se o TTE da abordagem centralizada e o melhor tempo obtido dentre todas as execuções com as abordagens distribuídas, apresentando-os como frações do TTE da abordagem centralizada. As consultas de alto custo C1, C2 e C10 apresentam uma diminuição no tempo de execução, assim como a consulta de baixo custo, C12. No caso da consulta C11, é interessante mencionar que a abordagem centralizada nem mesmo consegue executá-la, enquanto as abordagens distribuídas tiveram sucesso com a execução. Este, portanto, é também considerado um ganho obtido com a distribuição. Uma característica comum a todas estas consultas que obtiveram ganho é a presença de junções, o que geralmente é uma operação bastante custosa para o SGBDX. A consulta C1 foi a que obteve maior ganho com a distribuição: com 64 processadores, reduziu o tempo em aproximadamente 62%.

É interessante notar que, com exceção da C12 (cujo ganho com Partix-EVP em 64 processadores é desprezível), as consultas de baixo custo não obtiveram qualquer ganho com a distribuição. Este é um cenário bastante diferente daquele vivenciado pela abordagem Partix-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011), onde mesmo as consultas de baixo custo apresentavam ganhos com a distribuição. Atribui-se a esta diferença de comportamento o fato de que, na avaliação experimental deste trabalho, utiliza-se o SGBDX BaseX, que automaticamente cria diversos índices para as suas bases, diferentemente do SGBDX Sedna, utilizado no trabalho anterior, que não cria nenhum índice automaticamente. Ao utilizar índices, o SGBDX BaseX diminui consideravelmente o tempo de execução de várias consultas no ambiente centralizado, e isto torna a execução pela abordagem distribuída mais onerosa devido aos custos fixos de processamento inerentes deste tipo de abordagem. No caso da consulta C9, por exemplo, o melhor TTE entre as abordagens distribuídas (498,63 seg) é quase 1000 vezes maior do que o tempo da abordagem centralizada (0,514 seg).

Tabela 9. TTE para a base de dados SD de 1 GB

	1 proc.	16 proc.			32 proc.			64 proc.		
Consulta	**Centralizado**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**
C1	35,013	41,796	46,726	36,355	23,789	30,989	22,678	14,934	27,945	13,403
C2	22,770	26,193	36,36	24,244	20,303	26,289	16,791	14,256	27,463	13,425
C3	4,158	15,734	27,367	9,165	11,772	22,693	7,869	7,591	20,180	5,093
C4	3,447	15,342	33,147	16,116	14,562	36,943	14,042	13,656	38,130	11,768
C5	6,694	17,05	36,466	16,507	14,351	33,684	13,067	13,349	27,391	11,517
C6	2,052	7,267	21,886	6,207	6,705	24,038	5,279	6,424	21,589	4,872
C7	0,380	2,547	20,02	0,896	2,656	21,496	0,935	2,541	19,172	1,069
C8	9,098	25,154	42,151	24,482	21,054	43,357	20,010	19,057	31,619	17,123
C9	0,514	559,736	498,63	560,597	517,886	861,098	560,542	558,92	508,380	585,374
C10	255,456	239,062	246,163	239,001	182,931	179,611	179,395	156,615	144,307	145,993
C11	E	E	469,613	487,172	353,386	359,111	369,813	298,174	259,050	300,555
C12	10,885	20,631	35,433	19,373	15,103	32,703	14,577	12,147	30,032	10,814

Tabela 10. TTE para a base de dados MD de 1GB

	1 proc.	16 proc.			32 proc.			64 proc.		
Consulta	**Centralizado**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**
C1	37,111	14741,079	13020,314	11098,567	7287,778	11638,367	6665,037	3741,759	3499,989	3534,734
C2	30,337	21085,192	19486,987	17081,903	8519,538	16978,379	10597,161	5235,066	5158,983	5306,595
C3	5,260	95,963	100,726	72,283	64,176	79,673	48,706	66,909	41,011	32,391
C4	3,423	17,439	30,839	14,794	15,598	31,448	18,855	15,133	35,452	15,386
C5	5,791	18,288	32,299	18,103	15,967	30,453	13,736	15,291	30,162	13,064
C6	2,083	9,052	20,791	7,941	8,928	21,599	6,703	9,257	23,457	6,447
C7	0,407	325,048	299,349	312,551	175,005	175,351	155,957	98,425	95,373	79,595
C8	9,174	25,857	42,025	24,304	21,635	40,323	20,611	21,951	36,568	18,223
C9	0,610	5,013	21,708	3,750	4,917	21,401	3,384	5,571	22,631	3,386
C10	216,326	63952,468	60281,397	51705,463	33134,823	32213,773	27716,362	16779,784	15760,814	15540,426
C11	E	T	116578,91	101392,189	66082,216	61131,648	58136,893	34435,529	31576,512	31736,255
C12	13,930	T	158843,4	133236,292	88835,471	81911,919	91041,788	42332,576	56041,645	46563,367

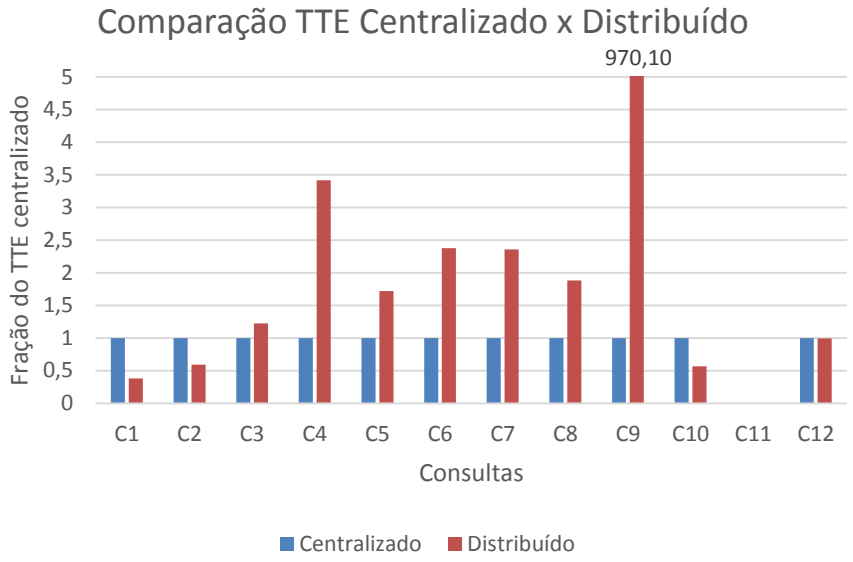


Figura 35. Comparação entre os TTE da abordagem centralizada e da abordagem distribuída (1GB SD)

A Figura 36 apresenta alguns gráficos selecionados de consultas com os TTE das abordagens de acordo com o número de processadores empregados na execução. O ANEXO C apresenta todos os gráficos para o cenário com base de dados SD, de 1 GB. Nestes gráficos, adiciona-se também o TTE obtido com a abordagem centralizada. Portanto, é possível tanto visualizar o comportamento de cada abordagem perante o aumento de processadores quanto comparar os tempos de execuções distribuídas com o tempo obtido da execução centralizada.

A C1 (Figura 36(a)) é uma consulta de alto custo cujo TTE em abordagens distribuídas se apresentou menor que o TTE da abordagem centralizada somente com 32 processadores ou mais. Isto mostra que a abordagem centralizada consegue bons resultados com o emprego dos índices e de suas otimizações, mesmo para consultas de alto custo. Dentre as abordagens distribuídas, a PartiX-EVP foi aquela que apresentou o melhor tempo com os diversos números de processadores. Nota-se que tanto a abordagem VPHadoop quanto a PartiX-VP apresentam um incremento do tempo com 16 processadores, enquanto a Partix-EVP consegue manter o tempo. Atribui-se a isso a melhor distribuição de carga, com o número maior de fragmentos, e o pouco processamento extra adicionado por esta abordagem, em contraste com aquele adicionado pelo *framework* Hadoop na abordagem VPHadoop. Nesta consulta, no cenário com 64 processadores, a PartiX-EVP consegue apresentar um TTE aproximadamente 10% menor que o TTE para a PartiX-VP, e 62% menor que TTE para a abordagem centralizada.

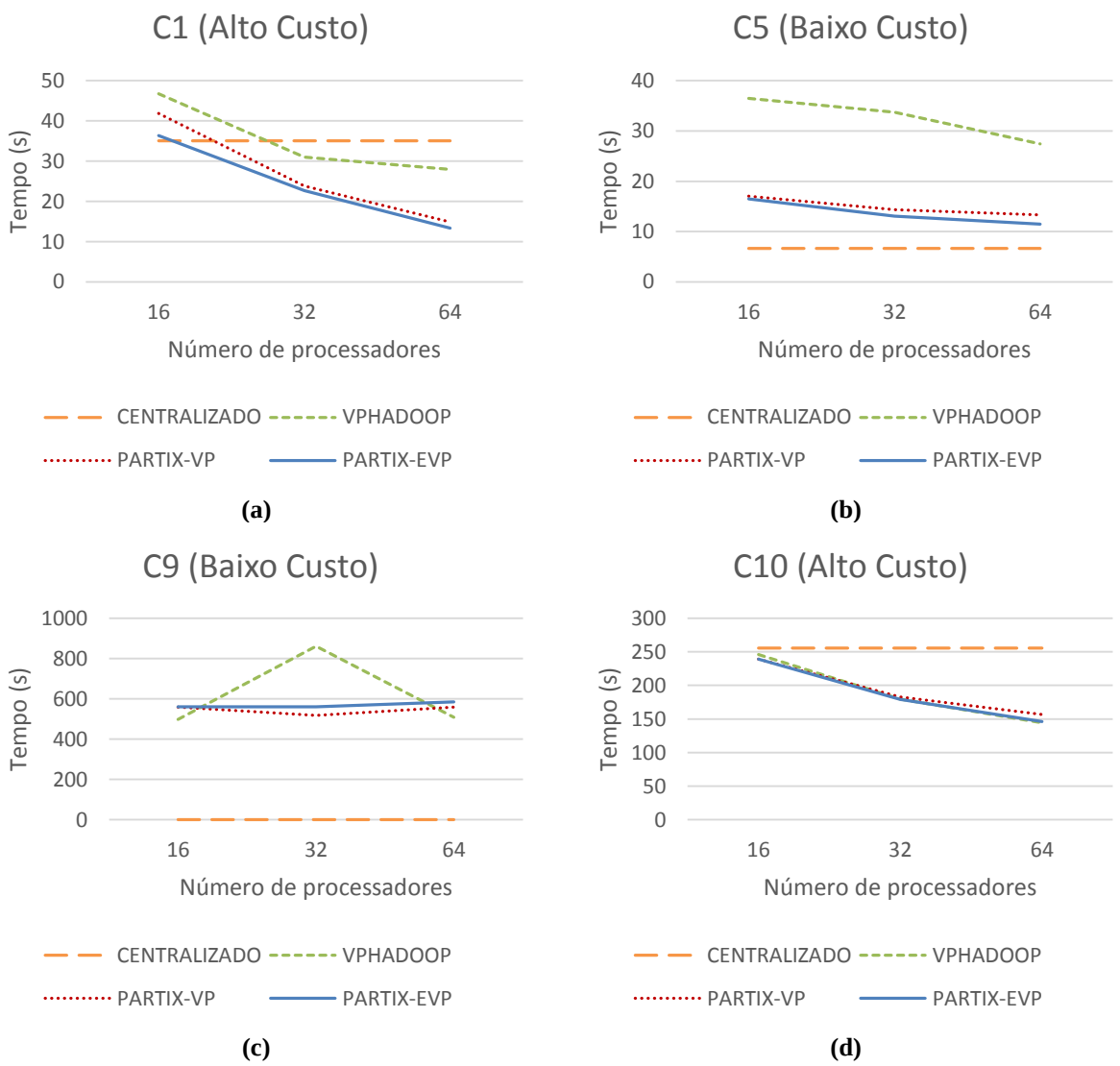


Figura 36. TTE x número de processadores para as consultas C1 (a), C5 (b), C9 (c) e C10 (d) no cenário 1GB SD

A consulta C10 (Figura 36(d)) também é uma consulta de alto custo, mas com a diferença de apresentar um TTE menor que o da abordagem centralizada para todas as abordagens distribuídas. Sendo a execução mais demorada, o impacto do processamento extra devido à distribuição diminui, e todas as abordagens distribuídas apresentam comportamento e TTE bastante similar perante o aumento do número de processadores. Ainda, a distribuição da execução da consulta apresenta benefícios em todos os cenários distribuídos com todas as abordagens, se comparados ao cenário centralizado. Para esta consulta, a abordagem VPHadoop apresentou o menor TTE, com redução de 44% em relação ao cenário centralizado.

De uma maneira geral, as consultas que possuem junções apresentam ganhos com a distribuição de execução e com o aumento do número de processadores. As exceções à regra, neste caso, são as consultas C3, C7 e C12, que também possuem junções, mas são consultas

de baixo custo. Atribui-se o comportamento destas consultas, em específico, ao tamanho reduzido de seus documentos resposta.

A Figura 36(b) e a Figura 36(c) apresentam os resultados para as consultas C5 e C9, de baixo custo. Mesmo utilizando 64 processadores, as abordagens distribuídas não conseguem obter TTE menor que aquele obtido com a abordagem centralizada. Isto acontece porque o TTE destas consultas, nesta abordagem, é pequeno. No caso da consulta C5, ao comparar os TTE das abordagens distribuídas entre si no caso com 64 processadores, nota-se que, novamente, a abordagem PartiX-EVP obtém uma redução de TTE sobre a PartiX-VP, de aproximadamente 14%, enquanto o TTE obtido com a VPHadoop é mais que o dobro. Isto mostra que o *framework* Hadoop não é uma boa alternativa em casos de consulta com TTE baixo. Já na consulta C9, a distribuição da execução, além de aumentar em quase mil vezes o tempo de execução, não apresenta uma diminuição no TTE com o aumento do número de processadores.

Além das consultas mostradas na Figura 36, os gráficos das demais consultas, constantes no ANEXO C, também apresentam a abordagem PartiX-EVP com o menor tempo de execução dentre as abordagens distribuídas em vários casos (9 das 12 consultas). Isto indica que a mudança na técnica de fragmentação virtual proposta por este trabalho, que consiste em aumentar o número de fragmentos gerados, traz um benefício para a execução, sem causar impacto nos casos em que este aumento não ocorreu (consultas de baixo custo).

A Tabela 10 indica que, para o cenário com base de dados MD de 1GB, a abordagem centralizada é sempre mais vantajosa. As abordagens distribuídas fizeram o TTE crescer em todas as consultas, com exceção da C11, a qual a abordagem centralizada não conseguiu executar no tempo estipulado. Percebe-se que os índices e otimizações internas do SGBDX BaseX beneficiam a abordagem centralizada (e prejudicam as abordagens distribuídas, como é explicado adiante). Novamente, é importante mencionar que o comportamento observado, mesmo com a abordagem PartiX-VP, difere do apresentado por Rodrigues, Braganholo e Mattoso (2011) pelo fato do SGBDX empregado naquela avaliação não empregar índices.

Uma característica bastante evidente ao verificarmos os gráficos para estas execuções, constantes no ANEXO D, é que todas as consultas que possuem junções (C1, C2, C3, C7, C10, C11 e C12) possuem comportamentos similares para as abordagens distribuídas. Os TTE são maiores que aqueles da abordagem centralizada, mas diminuem, conforme o número de processadores utilizados aumenta. A Figura 37(b) e a Figura 37(c) apresentam os gráficos para as consultas C7 e C10, que são de baixo e alto custo, respectivamente, e possuem junções. Percebe-se o comportamento bastante similar. Outro ponto interessante destes

gráficos é que a abordagem PartiX-EVP apresentou os melhores resultados dentre as abordagens distribuídas, na maioria dos casos.

No caso das consultas sem junções neste cenário (C4, C5, C6, C8 e C9), as abordagens distribuídas também apresentaram comportamentos parecidos entre si, ora apresentado ligeira queda no TTE com o incremento de processadores, ora apresentando um aumento no tempo. Por exemplo, no gráfico da Figura 37(a) temos o comportamento dos TTE para a consulta C6. A abordagem PartiX-EVP apresenta uma ligeira queda no tempo de execução entre 16 e 64 processadores (18%, aproximadamente), enquanto as abordagens Partix-VP e VPHadoop apresentaram incrementos no tempo (de aproximadamente 20% e 13%, respectivamente). Considerando o tempo com 64 processadores, o Partix-EVP foi 41% mais rápido que o Partix-VP, enquanto o VPHadoop foi 116% mais lento.

Na comparação entre o cenário envolvendo as execuções sobre a base de dados SD e MD de 1 GB, é possível notar que as consultas com junção (C1, C2, C3, C7, C10 e C11) tiveram o seu TTE aumentado de forma bastante significativa no caso MD. O gráfico da Figura 38 apresenta esta comparação. Para cada consulta, é considerado somente o melhor TTE dentre os obtidos pelas abordagens distribuídas, tanto no caso SD quanto no caso MD, sendo que, nos gráficos, estes tempos estão representados como a fração do TTE no caso SD. Com exceção da consulta C3, todas as consultas com junção (Figura 38(b)) apresentam TTE acima de 85 vezes maior para o caso MD, chegando a 384 vezes para a consulta C2. Após uma análise detalhada destes casos, constatou-se que, durante a execução das subconsultas no caso SD, o SGBDX BaseX emprega, como esperado, os seus índices internos para realizar a junção da consulta. No caso MD, devido à reescrita necessária para o emprego da função *position()*, como explicado na seção 2.3.2, o SGBDX além de realizar concatenações de fragmentos para formar o conjunto de dados do lado esquerdo da junção, não consegue empregar o seu índice interno para resolvê-la, requerendo a concatenação e varredura para o lado direito da junção a cada iteração. Isso explica o crescimento substancial no tempo de execução.

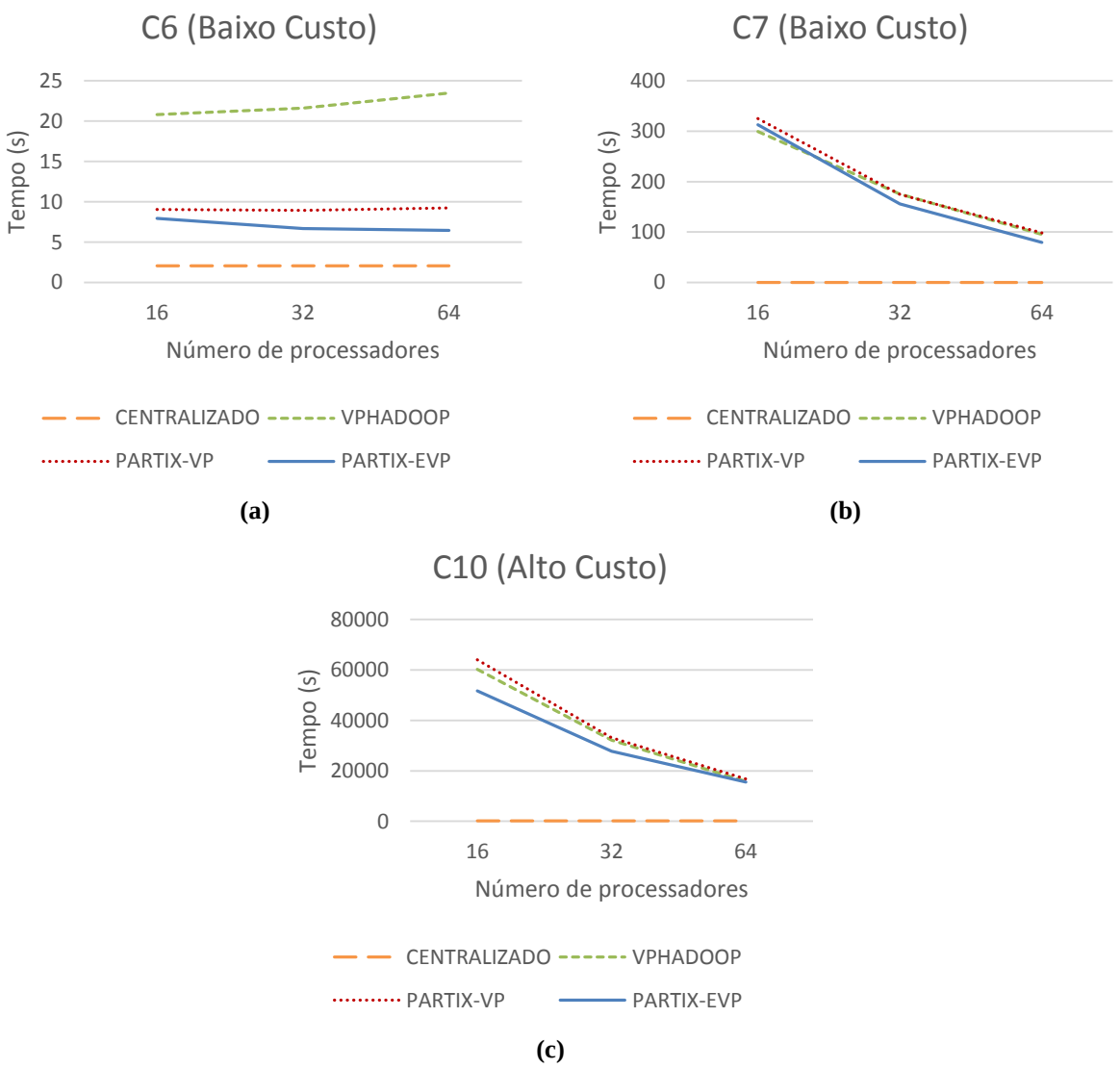
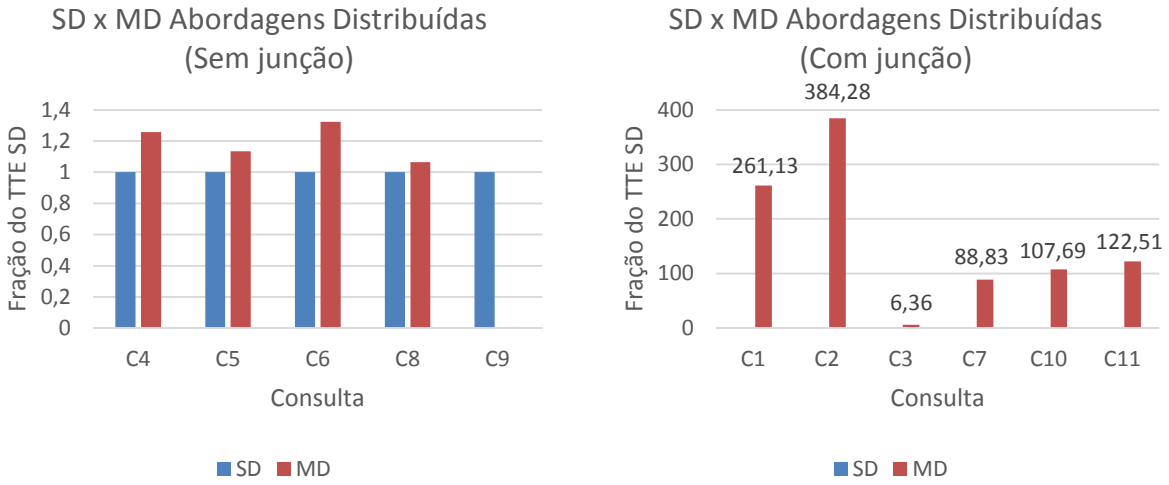


Figura 37. TTE x número de processadores para as consultas no cenário 1GB MD

A Figura 38(a) apresenta o gráfico de comparação entre o caso SD e MD para as consultas que não envolvem junção, considerando somente as abordagens distribuídas. Percebe-se que, com a exceção da consulta C9, todas as consultas apresentam um tempo ligeiramente maior no caso MD. Este incremento é decorrente do processamento extra necessário para executar uma consulta que envolve uma união de sequências de elementos. A execução da consulta C9 nos casos SD e MD foi então analisada mais detalhadamente, para identificar o motivo da diferença entre os seus tempos, que é 99% menor no caso MD. Constatou-se que a subconsulta gerada pela fragmentação virtual no caso SD possui o filtro posicional do atributo de fragmentação virtual adicionado a um caminho incompleto (com "//"), o que não permite ao SGBDX BaseX reescrevê-lo de maneira otimizada. Na subconsulta para o caso MD, o filtro é aplicado somente na união de sequências, conforme

mostrado na Figura 7(b). Desta forma, o BaseX consegue reescrever todos os caminhos para não serem mais incompletos, otimizando a execução.



(a) **(b)**

Figura 38. Comparação entre SD e MD para abordagens distribuídas (1 GB)

A Tabela 11 apresenta os TTE para as execuções utilizando a base de dados SD de 10 GB. A Tabela 12, por sua vez, apresenta os TTE para as execuções de consultas utilizando a base de dados MD de 10 GB. Com o aumento da base, algumas execuções não completaram com sucesso, seja por problemas de falta de memória ou pelo término do tempo estipulado como limite. Estas execuções são, principalmente, das consultas de alto custo e daquelas que possuem junções (no caso de consultas de baixo custo). Contudo, é interessante notar que, nos casos das consultas C2, C10 e C11, a distribuição da execução viabilizou a execução, que não era possível no caso centralizado. No caso da consulta C10, a abordagem distribuída Partix-VP também não conseguia finalizar a execução, enquanto as abordagens propostas por este trabalho foram capazes de obter o resultado final. Isto se deve ao maior número de fragmentos utilizados nestas abordagens, que acaba por diminuir o intervalo de dados considerado em cada subconsulta da fragmentação virtual, tornando a sua execução factível. Na consulta C11, caso com 16 processadores, a abordagem VPHadoop não concluiu a execução, enquanto a abordagem Partix-EVP obteve o resultado final. O número de fragmentos utilizados é a razão para esta diferença: no caso da VPHadoop, pela eleição realizada, o número de fragmentos a ser empregado pela abordagem é de 2 vezes o número de processadores (consulta de alto custo), enquanto para a PartiX-EVP, são utilizados 4 vezes o número de processadores trabalhadores (total de processadores menos 1) para a mesma consulta.

As execuções de consultas nas bases de dados de 10 GB obtiveram um coeficiente de variação bastante elevado. Conforme explicado na metodologia, busca-se mitigar esse

problema com o aumento no número de repetições. Contudo, devido ao tempo requerido para esta tarefa, considerado o TTE requerido para cada repetição da execução nestes cenários, não foi possível atingir o limiar estipulado de 20% de coeficiente de variação em muitos dos casos com as bases de 10 GB. Do exposto, opta-se por não considerar, durante as análises deste trabalho, os TTE obtidos nos cenários de 10 GB que possuam coeficiente de variação maior que o limiar estipulado, embora seus tempos e gráficos (ANEXO E) sejam apresentados. Recomenda-se que tais execuções sejam repetidas em um trabalho futuro para validação das propostas nestes cenários.

As consultas C5, C9 e C11 são as únicas consideradas na análise do cenário SD com 10 GB. Os gráficos destas consultas estão apresentados na Figura 39. A consulta C5 apresenta seu melhor tempo no caso centralizado, o que significa que a distribuição não beneficia a sua execução. Contudo, com o aumento do número de processadores nas abordagens distribuídas, o TTE decai, ficando próximo ao TTE do caso centralizado quando se usa 64 processadores. Todas as abordagens distribuídas tiveram comportamento similar nesta consulta. Na consulta C9 (Figura 39(b)), o TTE nas abordagens distribuídas é, no mínimo, 1000 vezes maior que no caso centralizado. É bastante claro, portanto, que além do processamento extra decorrente da distribuição, a reescrita da consulta na fragmentação virtual interfere drasticamente na otimização de execução de algumas consultas no SGBDX. No caso, esta é uma consulta que utiliza funções de agregação. É provável que o SGBDX já possua em seus registros internos estatísticas sobre a base de dados que permitam a ele responder com extrema rapidez às funções utilizadas, mas que não podem ser utilizadas quando a função está sendo aplicada somente a um subconjunto dos dados. A consulta C11 (Figura 39(c)) não obteve resultados nas abordagens centralizada e Partix-VP devido à falta de memória. No caso da base de dados de 10 GB, o documento resposta desta consulta possui mais de 20 GB, e atribui-se a este fato a impossibilidade de execução com estas abordagens. Contudo, as abordagens Partix-EVP e VPHadoop conseguem realizar a execução, pois o número de fragmentos utilizado é maior. Verifica-se que o VPHadoop possui um ganho superior ao Partix-EVP neste caso. Atribui-se a este ganho o uso de um processador a mais pelo VPHadoop, que no caso do Partix-EVP é utilizado como mediador.

Tabela 11. TTE para a base de dados SD de 10GB

	1 proc.	16 proc.			32 proc.			64 proc.		
Consulta	**Centralizado**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**
C1	1592,406	2973,264	1167,505	1579,495	2773,949	1808,986	3016,620	2489,834	2286,510	1832,514
C2	E	1398,766	1230,868	1430,374	1357,454	1390,34	2166,560	2166,784	1142,450	1907,223
C3	33,860	121,951	77,84	82,945	78,04	62,349	128,792	105,498	82,205	58,200
C4	61,052	137,909	169,057	196,080	124,776	140,404	185,855	111,917	145,155	166,876
C5	100,984	151,4	163,933	147,681	128,916	128,989	122,376	112,697	118,256	109,710
C6	41,846	52,558	58,744	45,963	46,155	50,416	38,522	41,668	45,722	34,885
C7	10,344	21,176	58,201	38,349	10,663	59,758	38,611	8,002	46,988	24,104
C8	179,067	254,918	332,506	242,324	215,882	311,973	195,749	183,367	183,040	169,374
C9	3,475	5775,113	4909,374	5362,793	5952,791	4916,089	5618,992	6486,482	7027,515	5605,607
C10	E	E	5504,339	6067,936	E	3870,115	5429,344	E	3512,067	5772,842
C11	E	E	E	9947,107	E	8419,134	9628,711	E	6678,405	9875,753
C12	341,893	295,436	236,449	319,070	225,832	231,044	242,243	304,164	139,873	339,811

Tabela 12. TTE para a base de dados MD de 10GB

	1 proc.	16 proc.			32 proc.			64 proc.		
Consulta	**Centralizado**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**	**PartiX-VP**	**VPHadoop**	**Partix-EVP**
C1	578,743	T	T	T	T	T	T	T	T	T
C2	E	T	T	T	T	T	T	T	T	T
C3	41,883	539,541	632,777	344,609	388,502	463,884	292,404	253,649	225,779	160,183
C4	57,027	166,714	171,927	168,725	149,379	189,21	179,872	159,6	142,619	147,777
C5	84,144	155,36	183,734	156,208	170,882	168,081	124,109	165,727	134,321	106,855
C6	32,847	62,412	72,411	56,055	60,114	75,012	45,579	91,395	70,470	39,822
C7	22,511	31703,24	29564,226	31952,785	16866,946	22027,351	15738,448	8307,606	11500,024	7862,163
C8	139,728	260,281	285,981	258,220	199,844	239,443	258,220	227,333	217,993	195,226
C9	16,741	26,075	34,221	24,692	31,315	32,807	22,116	32,558	33,340	25,580
C10	E	T	T	T	T	T	T	T	T	T
C11	E	T	T	T	T	T	T	T	T	T
C12	342,290	T	T	T	T	T	T	T	T	T

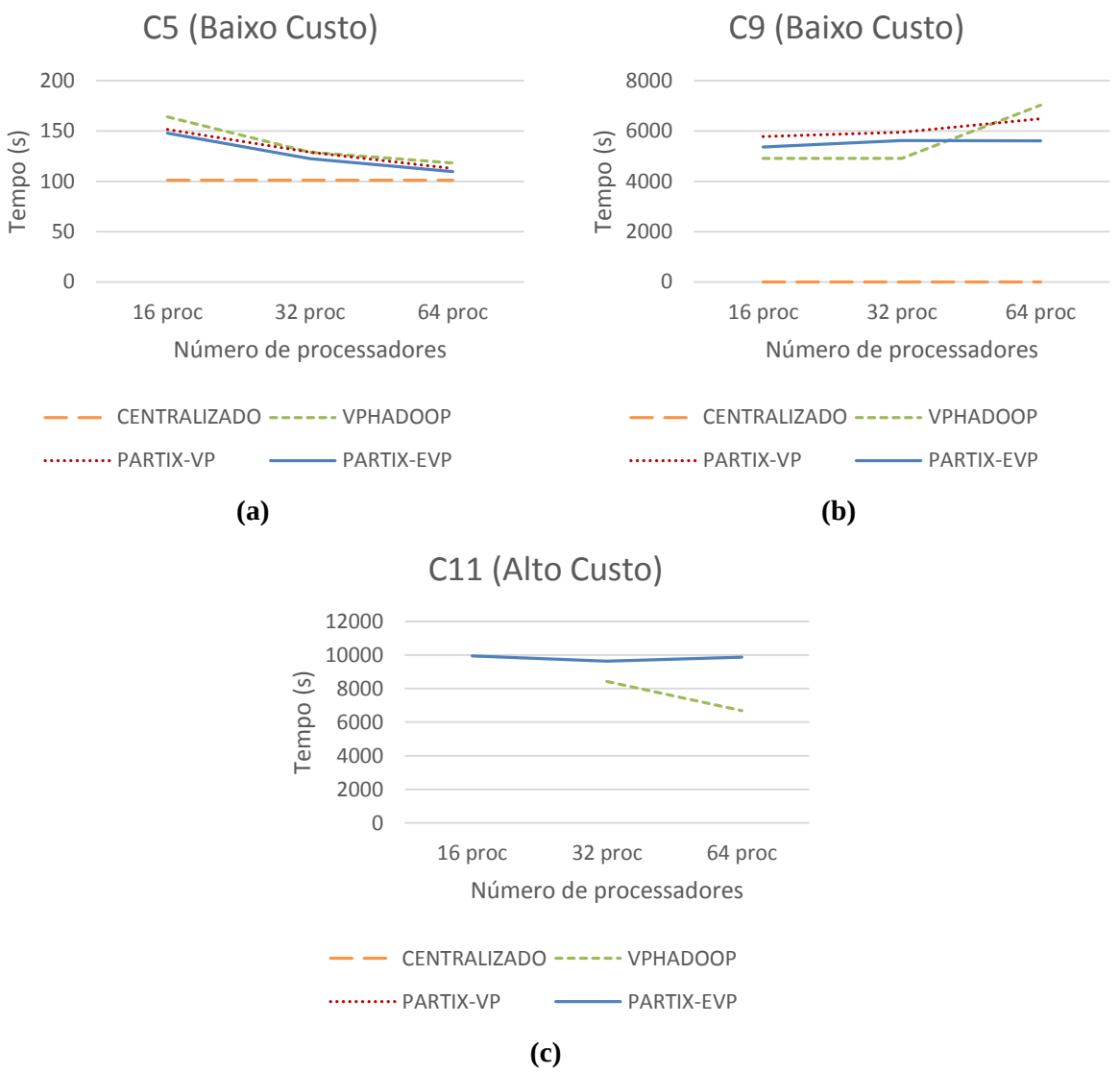


Figura 39. TTE x Número de processadores para consultas no cenário SD com 10 GB

Com os tempos de execução definidos, é também interessante estabelecer como se comporta o TTE em decorrência do crescimento da base de dados. O gráfico da Figura 40 apresenta o TTE das consultas para o caso SD de 10 GB como fração do TTE das mesmas consultas para o caso SD de 1GB. Com isso, podemos verificar em quantas vezes o tempo cresceu para cada consulta. No caso da consulta de alto custo C1, percebe-se que o TTE para o caso distribuído cresceu de forma muito mais significativa que o TTE para o caso centralizado. Para as consultas C2, C10 e C11, o caso centralizado não foi concluído devido à falta de memória, então não é possível realizar a comparação. No caso das consultas de baixo custo C4, C5, C6, C7, C8 e C12, o crescimento no caso distribuído foi menor, o que indica que, muito embora o TTE com estas abordagens seja maior que o TTE do caso centralizado, o impacto do crescimento da base nestas abordagens, para estas consultas, é menor. Note que os

coeficientes de variação nas execuções nas bases de 10GB foram altos, o que pode impactar nos resultados dessa análise.

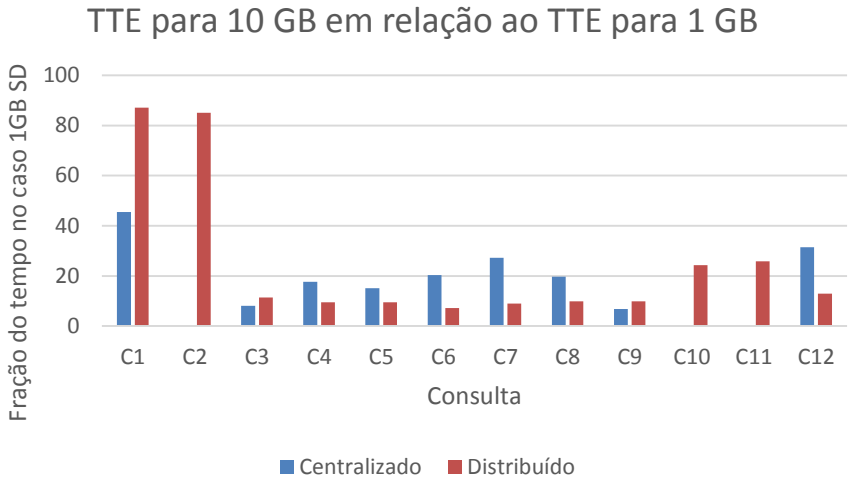


Figura 40. Comportamento do TTE com o crescimento da base

Assim como no caso da base de dados MD de 1 GB, várias execuções das consultas no caso MD de 10 GB não concluíram devido ao tempo limite estabelecido, pois a reescrita da consulta com a função *position()* afeta as otimizações do SGBDX durante a execução das subconsultas, conforme já explicado. Além disso, as execuções que concluíram também apresentaram grande coeficiente de variação, não havendo tempo hábil para a reexecução com mais repetições. Desta forma, opta-se por não analisar estes resultados e propõe-se como trabalho futuro a reexecução e subsequente análise.

6.7 DISTRIBUIÇÃO DE CARGA

Um dos objetivos deste trabalho é verificar se a mudança na técnica de fragmentação proposta é capaz de mitigar o problema da distribuição não uniforme dos dados e seus efeitos na distribuição de carga por meio do incremento no número de fragmentos virtuais utilizados na técnica. Esta Seção analisa se a abordagem escolhida consegue, de fato, mitigar este problema.

Para medir a distribuição de carga, cada abordagem foi instrumentada de forma a se obter, durante a fase de execução paralela de subconsultas, quantos processadores estavam em uso a cada momento. No caso do PartiX-VP, a abordagem indica, eu seu *log* de execução, o tempo que cada subconsulta (e, portanto, cada processador) levou para executar. Considerando que as consultas iniciaram sua execução simultaneamente no tempo t_0, esses dados nos indicam os momentos em que cada processador termina sua execução, sendo possível, portanto, estabelecer uma curva de utilização de processadores ao longo do tempo.

Para a abordagem VPHadoop, tem-se nos *logs* de execução do *framework* o momento exato (*timestamp*) em que cada tarefa do tipo *map* iniciou e terminou. Desta forma, também é possível determinar o perfil de utilização de processadores da abordagem, incrementando o contador de processadores quando uma tarefa começa, e decrementando-o quando a tarefa termina. Na abordagem PartiX-EVP, também tem-se o momento (*timestamp*) em que cada processador inicia e termina o processamento de uma subconsulta, sendo seu perfil de utilização de processadores calculado da mesma forma. Cabe lembrar que a abordagem PartiX-EVP sempre utiliza um processador como mediador e, portanto, este processador não executa subconsultas durante toda a fase de execução paralela. Logo, os gráficos desta abordagem sempre estarão com um processador a menos. Como a análise, neste caso, é para cada execução de consulta, opta-se por considerar o TTE da repetição que representa a mediana dentre todos os TTE para a referida consulta, naquele cenário. A escolha da consulta é feita com base na comparação entre os tempos obtidos com as várias abordagens. Uma diminuição no tempo de execução pode significar um melhor balanceamento de carga entre os processadores.

A Figura 41 apresenta as curvas de utilização de processadores ao longo do tempo para a execução da consulta C10 sobre a base de dados SD de 1GB, utilizando uma configuração com 64 processadores. Neste caso, a abordagem VPHadoop está executando a fragmentação virtual com 128 fragmentos, sendo estes divididos em 64 *splits*. Já a abordagem Partix-EVP utiliza 252 fragmentos. Nota-se que tanto a abordagem VPHadoop quanto a Partix-VP apresentaram “degraus” bastante significativos no decaimento do número de processadores, o que significa que alguns processadores passaram mais tempo executando do que outros. Já a abordagem Partix-EVP apresentou um decaimento mais acentuado, o que indica que seus processadores acabaram seus processamentos sem grandes variações no tempo de execução. Os gráficos da Figura 42 corroboram esta observação. Neles, os tempos acumulados de execução em cada processador são mostrados em ordem crescente, para facilitar a visualização do comportamento. Percebe-se que o tempo acumulado de execução em cada processador é bem mais uniforme no caso da abordagem Partix-EVP (Figura 42(b)), se comparado ao da abordagem Partix-VP (Figura 42(a)). No caso da abordagem VPHadoop, tal gráfico não é possível de ser feito pois nos *logs* de execução do *framework* Hadoop não se tem a identificação de qual processador que o executa, somente o nó que o executou (e cada nó possui 8 processadores).

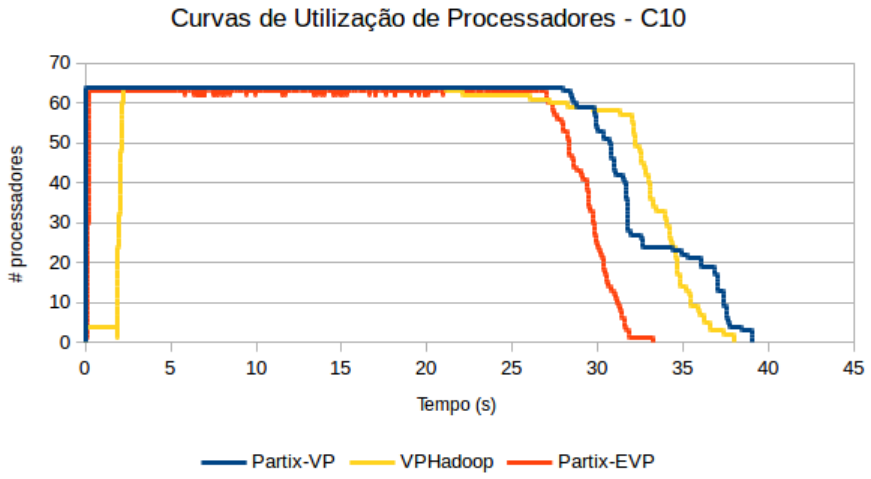
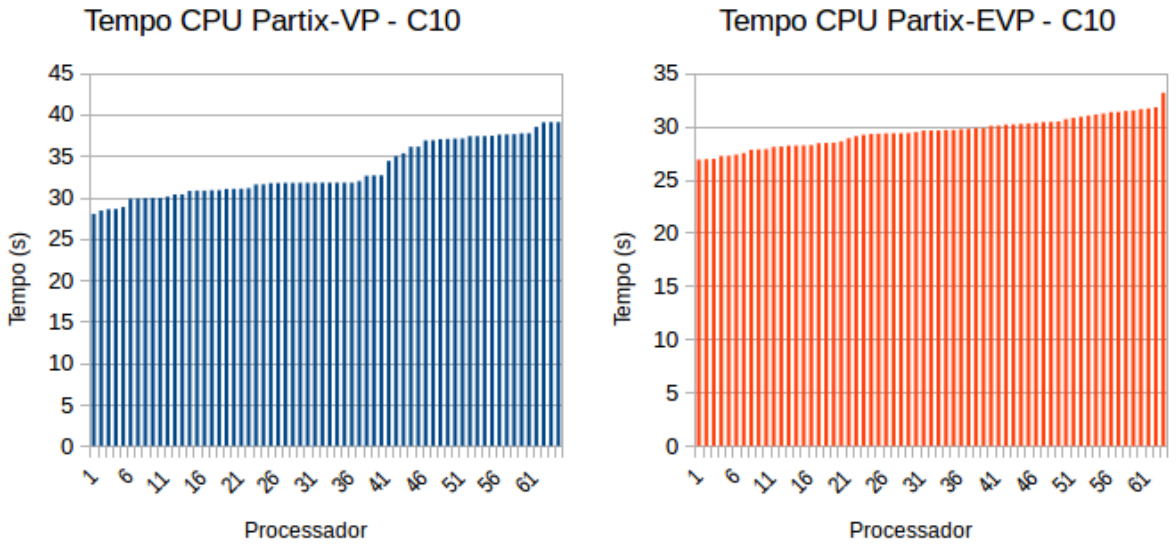


Figura 41. Curvas de utilização de processadores para C10, caso SD de 1GB, 64 processadores



(a) **(b)**

Figura 42. Tempo de processamento dos processadores para C10, caso SD de 1GB, com 64 processadores

É importante mencionar, contudo, que nem todas as execuções obtiveram um melhor balanceamento com a utilização da abordagem Partix-EVP. A Figura 43 apresenta as curvas de utilização de processadores para a consulta C1, no cenário com base de dados SD de 1GB, com 16 processadores. Nesta consulta, percebe-se que a abordagem Partix-VP e a abordagem Partix-EVP, muito embora tenham número de processadores diferentes, apresentam comportamento bastante similar, ficando o decaimento do número de processadores na abordagem Partix-EVP um pouco mais lento do que o decaimento do Partix-VP. No caso do VPHadop, o tempo de inicialização do *job* e das tarefas influencia o início do processamento,

que começa mais tarde. Além disso, o decaimento do número de processadores nesta abordagem possui um “degrau” ao atingir 6 processadores, o que indica um desbalanceamento ainda maior. Os gráficos da Figura 44 corroboram o apresentado no gráfico da Figura 43, pois indicam que há uma maior variabilidade nos tempos acumulados de execução na abordagem Partix-EVP.

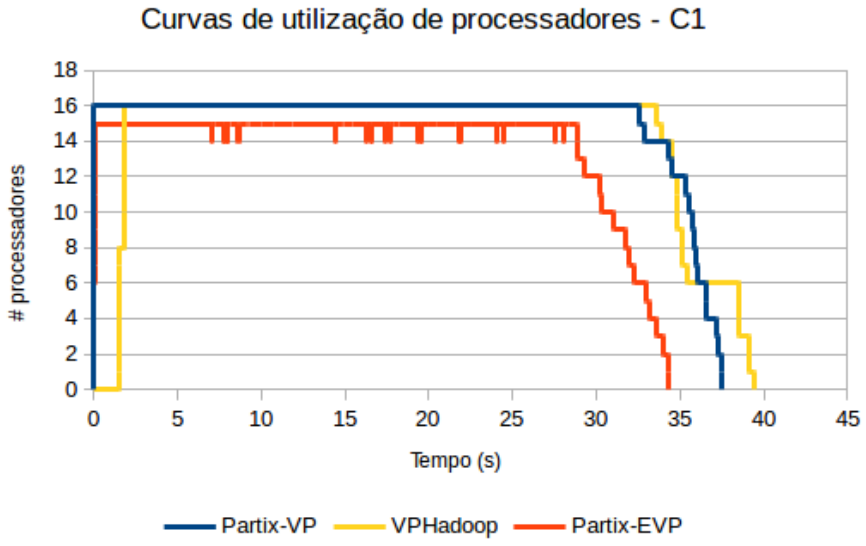
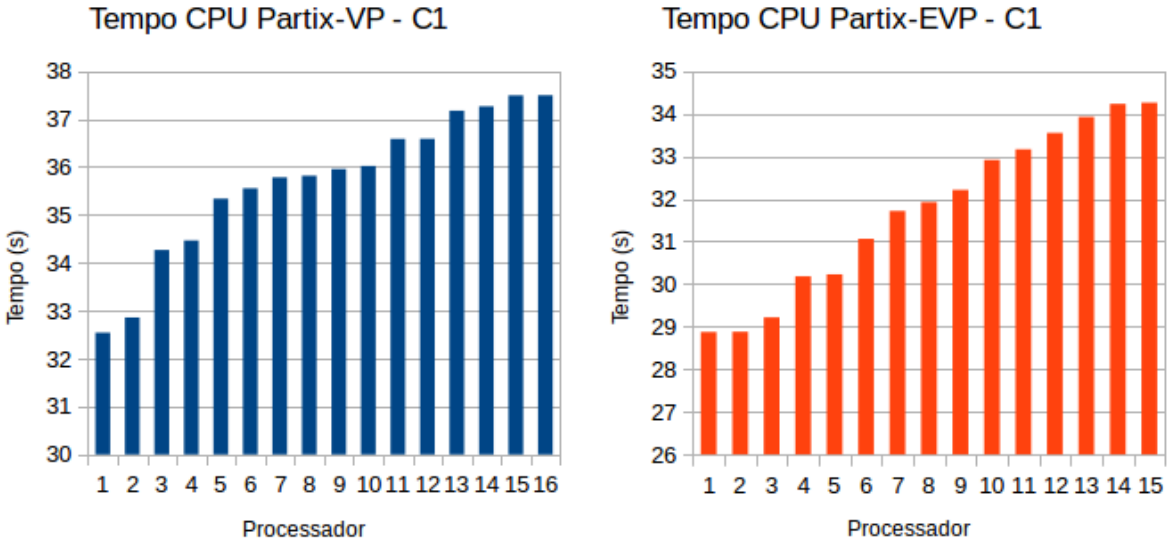


Figura 43. Curvas de utilização de processadores para C1



(a) **(b)**

Figura 44. Tempo de processamento dos processadores para C1

Salienta-se que, no caso da consulta C1, apresentado na Figura 43, o número de processadores utilizados é menor do que aquele utilizado no caso da consulta C10, apresentado na Figura 41. O fato de ter menos processadores influencia no número de fragmentos que se aplicará na execução paralela, que por sua vez, pode influenciar a

distribuição da execução nos nós. Idealmente, a utilização de mais fragmentos resultaria em uma melhor distribuição do processamento entre os processadores. Mas é preciso ter em mente que, em algumas consultas, aumentar o número de fragmentos acarreta também um aumento no tempo de execução, conforme se vê na Seção 6.5, devido ao aumento no processamento extra requerido.

6.8 CONSIDERAÇÕES FINAIS

Os protótipos VPHadoop e o PartiX-EVP, criados para viabilizar o estudo sobre a variação no número de fragmentos, demonstram que ao aumentar o número de fragmentos empregados na fragmentação virtual, é possível diminuir o TTE de consultas de alto custo, em comparação ao TTE da abordagem Partix-VP, que utiliza um número de fragmentos igual ao número de processadores empregados. Observa-se também que consultas de alto custo que produzem respostas de maior tamanho, como as consultas C10 e C11 obtêm ganhos adicionais com o aumento da fragmentação. As consultas de baixo custo geralmente não se beneficiam com este aumento de fragmentos, uma vez que isso gera um aumento no processamento extra requerido para distribuição e consolidação, o que no caso de consultas de baixo custo, é bastante significativo. Ainda, ao aumentarmos muito o número de fragmentos, constata-se que o processamento extra adicionado pode afetar também as consultas de alto custo. É necessário, portanto, cautela na escolha do número de fragmentos. Recomenda-se, em um trabalho futuro, automatizar esta escolha com a utilização de heurísticas baseadas em características da consulta de entrada, tais como presença de junção e cardinalidade dos elementos, dentre outras.

O *framework* Hadoop é capaz de prover um ambiente de execução distribuída com bastante transparência ao usuário. Ou seja, o usuário não necessita lidar com a fragmentação do trabalho, nem com a distribuição e controle da execução, o que pode ser bastante conveniente. Entretanto, pelos resultados obtidos, percebe-se que o *framework* pode adicionar um processamento extra considerável no processamento de consultas relativamente rápidas, como no caso das consultas de baixo custo da avaliação. Ao simplificarmos a distribuição da execução por meio do emprego de um mediador ativo durante a fase de execução paralela no PartiX-EVP, pôde-se comprovar que, para os casos em que a execução é relativamente rápida, o gargalo da abordagem não está na execução das subconsultas em si, mas sim no *framework* escolhido para realizar a distribuição, no caso, o Hadoop.

A avaliação do PartiX-VP em seu trabalho original (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) apresentou resultados bastante expressivos quanto a diminuição no tempo

total de execução com o emprego da fragmentação virtual. Todavia, a quantidade de dados utilizada naquela avaliação foi bastante limitada (100 MB), o que não ocasionou problemas na utilização de memória da solução. Ao realizar um aumento na quantidade de dados neste trabalho, foi possível perceber que, embora a abordagem não apresente limitações teóricas quanto ao tamanho da base, na prática, algumas combinações de consultas e bases tornam-se infactíveis, devido ao alto consumo de memória, principalmente por parte do SGBDX.

Ainda sobre a avaliação da fragmentação virtual em seu trabalho original, pode-se tecer alguns comentários quanto ao SGBDX empregado naquela e nesta avaliação experimental. Conforme já discutido no Capítulo 2, em um trabalho posterior à proposta do PartiX-VP, Silva, Mattoso e Braganholo (2013) identificaram que o SGBDX Sedna não apresenta um comportamento uniforme na utilização da função *position()*. Além disso, durante as investigações iniciais deste trabalho, identificou-se também que o SGBDX Sedna, por padrão, não utiliza qualquer índice interno para otimizar as consultas. Desta forma, as execuções da avaliação original do PartiX-VP por Rodrigues, Braganholo e Mattoso (2011) não se beneficiaram de qualquer redução no tempo de execução que o emprego de índices propicia, além de estarem sujeitas às variações observadas na utilização da função *position()*. Tais fatos contribuíram para a decisão de, neste trabalho, realizar-se a troca do Sedna para o BaseX, um SGBDX que emprega índices em suas consultas, e que apresenta comportamento uniforme na utilização da função *position()*. Mas, ao realizar esta mudança, constatou-se que, em muitos casos, a solução centralizada é bem mais rápida que as soluções distribuídas. Isto não ocorreu em nenhum caso na avaliação anterior. Ainda, constatou-se também que, em casos como o de consultas com junção em bases de dados MD, a fragmentação virtual interfere nas otimizações internas de consultas do SGBDX, o que faz com que a execução distribuída sofra grandes impactos. Isso posto, deve-se analisar cuidadosamente os casos em que a fragmentação virtual pode impactar negativamente o tempo de execução de consultas.

Finalmente, a avaliação da distribuição de carga no emprego das novas abordagens mostra que o aumento no número de fragmentos pode, de fato, melhorar a distribuição. Contudo, existe um ponto que deve ser considerado: ao aumentar o número de fragmentos, aumenta-se também o processamento extra necessário para distribuição e consolidação, o que aumenta o tempo total de execução. Se o objetivo inicial ao se mitigar o problema do balanceamento de carga é exatamente o de diminuir o tempo de execução, é preciso determinar o ponto de equilíbrio entre estas duas variáveis. Conforme já mencionado, um trabalho futuro que determine automaticamente o número de fragmentos pode, também auxiliar neste equilíbrio.

CAPÍTULO 7 – TRABALHOS RELACIONADOS

São muitos os trabalhos que utilizam a distribuição da execução de processamento em um cluster de computadores. Neste Capítulo são apresentadas algumas técnicas que focam na diminuição do tempo total de execução, o que motiva o estudo realizado neste trabalho. São elencados e discutidos, primeiramente, aqueles que, de alguma forma embasaram as abordagens VPHadoop e Partix-EVP, propostas deste trabalho. Em seguida, discute-se outros trabalhos que também abordam o processamento de dados XML em ambientes distribuídos com o modelo MapReduce. Na apresentação de cada técnica, busca-se salientar os seus pontos fortes e comentar os resultados. Além disso, são também apresentados os seus principais problemas e diferenças em relação à abordagem deste trabalho. São também apresentados trabalhos que apresentam propostas para a mitigação de problemas encontrados pelo trabalho aqui proposto, que podem vir a contribuir com a evolução da técnica e com pesquisas sobre a execução distribuída de consultas.

7.1 TRABALHOS DE BASE

A fragmentação virtual para processar documentos XML proposta por Rodrigues, Braganholo e Mattoso (2011) é uma das abordagens que embasa o trabalho aqui proposto. Esta técnica utiliza um cluster MPI de computadores para distribuir a execução de consultas XQuery em vários nós, fazendo uso também de instâncias de um SGBDX instaladas em cada nó contendo uma réplica total dos dados de entrada. Por meio da escolha criteriosa de um atributo de fragmentação e da aplicação da função XPath *position()* sobre um caminho presente na consulta de entrada, é possível derivar subconsultas que operam sobre intervalos de dados disjuntos, definindo assim fragmentos virtuais sobre os quais cada processador executa, de forma paralela. Em seguida, faz-se a combinação dos resultados parciais obtidos com a execução paralela, e obtém-se o resultado final da consulta. Esta técnica está apresentada em detalhes na Seção 2.3.2, e conforme discutido no Capítulo 4 e no Capítulo 5, foi também empregada nas propostas deste trabalho, com algumas melhorias (conforme relata o Capítulo 3). Os resultados obtidos por esta técnica são expressivos (95% de redução no tempo de execução), mas a massa de dados considerada na sua avaliação foi relativamente pequena (100 MB). Além disso, a abordagem proposta por Rodrigues, Braganholo e Mattoso (2011) não trata o problema da distribuição não uniforme dos dados, o que acarreta o desbalanceamento de carga durante a fase de processamento paralelo.

A abordagem MapReduce (DEAN; GHEMAWAT, 2004) para processamento de dados em cluster de computadores é outra abordagem que embasa o trabalho aqui proposto. Esta abordagem permite que a tarefa de processamento de dados em paralelo seja resumida à escrita de duas funções, a *map* e a *reduce*, que são aplicadas paralelamente por vários computadores de um cluster para se obter o resultado desejado. O MapReduce facilita o processamento de grandes massas de dados, disponibilizando a tolerância a falhas e a escalabilidade que caracterizam a sua flexibilidade para diversos cenários. Sendo assim, vários trabalhos que seguiram também o utilizaram como base para a realização da distribuição de execução, como veremos a seguir. Contudo, a facilidade proporcionada pelo modelo vem com um preço: o processamento extra realizado pelos *frameworks* que o implementam podem anular os ganhos obtidos com a distribuição do processamento.

7.2 PROCESSAMENTO DE CONSULTAS DISTRIBUÍDAS SOBRE DOCUMENTOS XML COM MAPREDUCE

7.2.1 CHUQL

ChuQL (KHATCHADOURIAN; CONSENS; SIMÉON, 2011) é uma proposta de extensão à linguagem XQuery para processar dados XML de forma nativa usando Hadoop. Além de estender o modelo de dados da XQuery (FERNÁNDEZ *et al.*, 2004) com registros do tipo MapReduce (pares <chave, valor>), a proposta também estende a linguagem XQuery para poder expressar as tarefas de *jobs* MapReduce. De uma forma geral, a ChuQL permite a um desenvolvedor habituado à linguagem XQuery expressar os passos básicos de um *job* MapReduce utilizando XQuery dentro de cada um deles. São eles: (1) *input*, que indica como ler os documentos a partir do HDFS; (2) *RecordReader(rr)*, que é responsável pela derivação dos registros do tipo par <chave, valor> a partir dos fragmentos XML da entrada; (3) *map*, que processa os registros disponibilizados pelo *rr* e emite novos registros do tipo par <chave, valor>, intermediários; (4) *reduce*, que processa os registros emitidos pelo *map*, agrupados pela chave; (5) *RecordWriter(rw)*, que define como o resultado da tarefa de *reduce* é convertido para um registro do tipo par <chave, valor>; e (6) *output*, que define como cada registro é escrito no HDFS.

A Figura 45 apresenta um programa em ChuQL para o exemplo clássico de processamento MapReduce, o *WordCount*, que realiza a contagem de palavras em um conjunto de documentos XML compostos por diversos elementos <linha>, sendo que cada elemento <linha> contém uma linha de texto do documento original. A expressão *mapreduce*

define o *job* MapReduce em ChuQL. A expressão *input* define a entrada de dados, que é oriunda de uma pasta do HDFS composta pelos vários arquivos XML, que se transforma em uma coleção de documentos XML. Cada documento em *input* será mapeado para uma tarefa do tipo *map* de um *job* MapReduce. A expressão *rr* define como se dará a criação de registros do tipo par <chave, valor> a partir dos dados de entrada, que no caso deste exemplo é <identificador, texto da linha>. Na expressão *map*, o valor recebido do registro é dividido utilizando como base espaços, e para cada palavra obtida, é emitido um registro do tipo <chave, valor> onde chave é a palavra e valor contém o número 1, sinalizando uma ocorrência daquela palavra. Na expressão *reduce*, são simplesmente contadas as ocorrências de valores para cada chave, e um novo registro é emitido contendo como chave a palavra e como valor o número de ocorrências daquela palavra. A expressão *rw* define como o registro emitido pelo *reduce* se traduz para um arquivo de resultado. Finalmente, na expressão *output*, o desenvolvedor define como se dará a escrita do registro, que no caso utiliza a função *put* da XQuery (MALHOTRA *et al.*, 2010) para criar um documento XML.

```
mapreduce {
  input { fn:collection("hdfs://input/") }
  rr { for $line at $i in $in//line return { key: $i, val: $line } }
  map { for $word in fn:tokenize($in=>val, " ")
        return { key: $word, val: 1 } }
  reduce { { key: $in=>key, val: fn:count($in=>val) } }
  rw { <word text="{$in=>key}" count="{$in=>val}"/> }
  output { fn:put($in,"hdfs://output/") }
}
```

Figura 45. Programa WordCount em ChuQL(KHATCHADOURIAN; CONSENS; SIMÉON, 2011, adaptado)

Em termos de arquitetura e funcionamento interno, o ChuQL funciona com um processador XQuery modificado para suportar a gramática estendida e os novos tipos de dados adicionados, de maneira que ele entenda as funções criadas. O *job* ChuQL é um *job* MapReduce com classes customizadas para *RecordReader*, *Mapper*, *Reducer* e *RecordWriter*, sendo que cada tarefa do tipo *map* e do tipo *reduce* também faz uso de um processador XQuery para processar os dados.

Durante os experimentos, os autores explicam que tiveram que dividir a base XML escolhida em diversos arquivos para que coubessem individualmente na memória do processo do processador XQuery. Caso contrário, teriam que lidar com a fragmentação XML, o que não foi feito. Entende-se, portanto, que esta é uma limitação da abordagem proposta. Também não são apresentados resultados de desempenho da abordagem, de maneira que não se consegue estabelecer medidas de comparação.

7.2.2 HADOOPXML

O HadoopXML (CHOI *et al.*, 2012) é uma solução para processamento distribuído de consultas XPath sobre documentos XML, fazendo uso do modelo MapReduce (DEAN; GHEMAWAT, 2004) ao empregar sua implementação de código aberto, o Hadoop.

Esta abordagem utiliza como entrada um arquivo XML contendo grande quantidade de dados armazenada no sistema de arquivos distribuído do Hadoop, o HDFS. Este armazenamento, contudo, não se dá de forma padrão. O documento XML é antes pre-processado para assim obter informações sobre a sua estrutura e seus elementos (rótulos). Estas informações são armazenadas em blocos separados no HDFS, mas no mesmo local que o bloco. Ainda, para cada bloco de dados, a solução armazena também o contexto em que o fragmento ali armazenado está (caminho completo do elemento inicial), possibilitando a recuperação deste contexto ao iniciar o processamento de cada bloco. É também no preprocessamento que as consultas são decompostas em padrões de caminho linear (*linear path patterns*), que são indexados em uma tabela com o mapeamento entre as consultas e as respectivas decomposições.

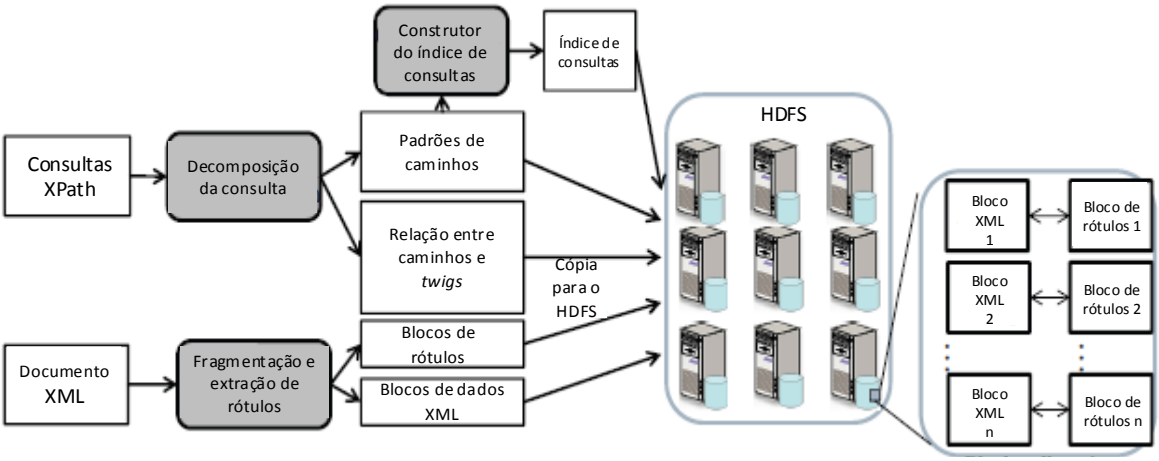


Figura 46. Preprocessamento do HadoopXML (CHOI *et al.*, 2012, adaptado)

Além do preprocessamento descrito, esta abordagem faz uso de 2 *jobs* MapReduce para processar as consultas de entrada. A Figura 47 e a Figura 48 ilustram o primeiro e o segundo *job*, respectivamente. No primeiro *job*, os *mappers* leem os blocos XML utilizando SAX e filtram os rótulos que possuem relação com as consultas. Então, os *reducers* agrupam os rótulos dos blocos XML de acordo com um identificador da subconsulta (*pathID*), contando a quantidade de rótulos para cada *pathID*. O resultado do primeiro *job* é um conjunto de pares <*pathID*, lista de rótulos>.

Na segunda etapa de MapReduce, os *mappers* leem as saídas da etapa anterior agrupando-as de acordo com o identificador de cada *reducer*. Após isso, as saídas agrupadas são encaminhadas para os respectivos *reducers*.

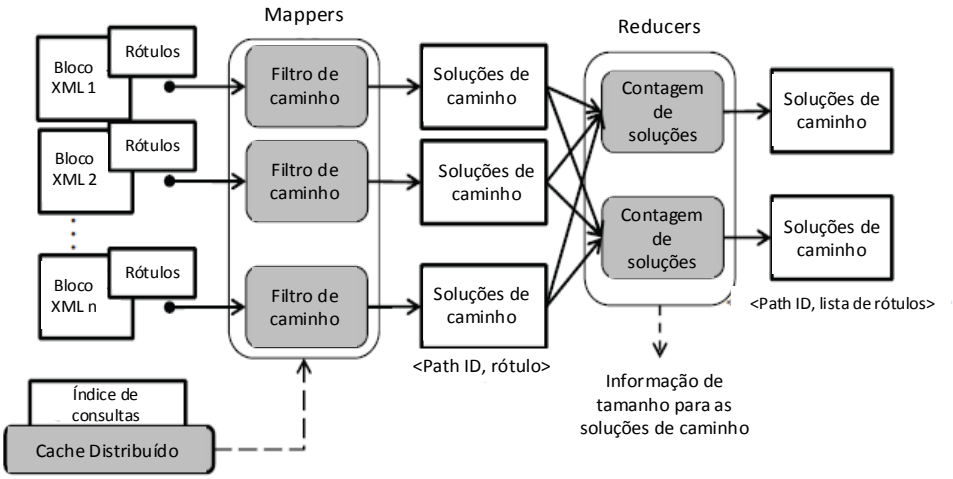


Figura 47. Primeiro *job* MapReduce do processamento HadoopXML (CHOI *et al.*, 2012, adaptado)

Por fim, os *reducers* executam as consultas com base no algoritmo TwigStack (BRUNO; KOUDAS; SRIVASTAVA, 2002), que combina consultas XPath do tipo *twig pattern* para formar junções, e consolidam o resultado final no sistema de arquivos. Essa etapa conta com um módulo de otimização de consultas, que leva em consideração o tamanho das saídas da primeira etapa de MapReduce e a tabela de mapeamento criada na etapa de pré-processamento, para fornecer um mecanismo de balanceamento de carga em tempo de execução. Na Figura 3.11 é apresentada uma visão geral da segunda etapa de MapReduce do HadoopXML.

Uma das limitações do HadoopXML é o custo de processamento que se tem ao executar duas etapas diferentes de MapReduce a cada nova consulta recebida, o que prejudica o desempenho do processamento de consultas com muitas junções. Além disso, esta abordagem trabalha somente com consultas XPath específicas, e não com formatos de consultas mais abrangentes como a XQuery. Por fim, a solução não foca em eficiência de processamento de uma consulta, mas sim em escalabilidade e tolerância a falhas na execução de múltiplas consultas XPath simultâneas.

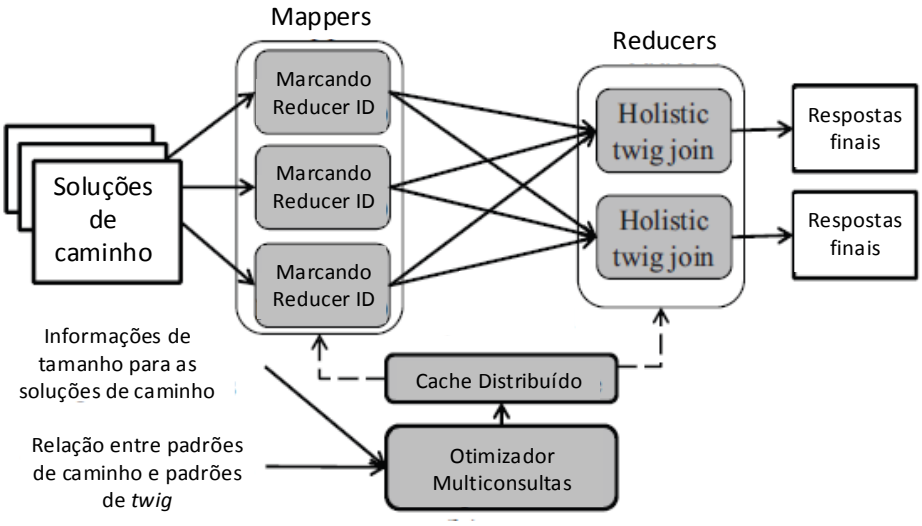


Figura 48. Segundo *job* MapReduce do processamento HadoopXML (CHOI *et al.*, 2012, adaptado)

7.2.3 BRACKITMR

O BrackitMR (SAUER; BÄCHLE; HÄRDER, 2013) é um sistema de processamento distribuído de consultas XQuery, baseado no Brackit, um *processador* XQuery para um único núcleo de processamento. O Brackit tem como principal objetivo prover um *framework* padrão para a compilação, otimização e execução de consultas, isolando estas tarefas do armazenamento, que é abstraído e implementado conforme a solução adotada. Com isso, este processador permite o processamento de dados oriundos de diversas fontes, o que o torna flexível e reutilizável.

Em um trabalho anterior, Sauer e Härder (2013) propuseram uma descrição abstrata da compilação de linguagens de consulta para o modelo de processamento MapReduce (DEAN; GHEMAWAT, 2008). Segundo eles, o processo de compilação é essencialmente o mesmo para quase todas as linguagens de consulta, dependendo somente da definição abstrata dos operadores e de suas composições, assim como na classificação deles em bloqueantes e não bloqueantes. Ainda neste trabalho, os autores apresentaram uma generalização do modelo MapReduce de processamento. Nela, um *job* MapReduce consiste em uma sequência de estágios *Mapper-Shuffle-Mapper* e o autor generaliza as funções *map* e *reduce* (que atuam em cada par <chave, valor>) para uma única função *task* (que atua em todo o subintervalo alocado para aquela tarefa). Um *job* é, portanto, composto de um par de funções *task*, uma executada antes do *shuffle*, outra depois. Uma consulta, neste modelo, é uma árvore de funções *tasks*, separadas por *shuffles*. A tradução utilizando a generalização é bastante

simples: operadores bloqueantes são convertidos em operações *shuffle* do MapReduce, enquanto que as operações não bloqueantes são empacotadas em *tasks*.

No BrackitMR, uma consulta é compilada em uma árvore de expressões que são avaliadas de forma *bottom-up*, formando algo semelhante a um plano de consulta com os operadores definidos na XQuery. A Figura 49 apresenta um exemplo desta tradução, em que uma árvore de operadores é derivada a partir de uma consulta XQuery. Esta árvore de operadores serve então de base para criar a árvore de operações MapReduce abstrata, que posteriormente será transformada em *jobs* MapReduce tradicionais.

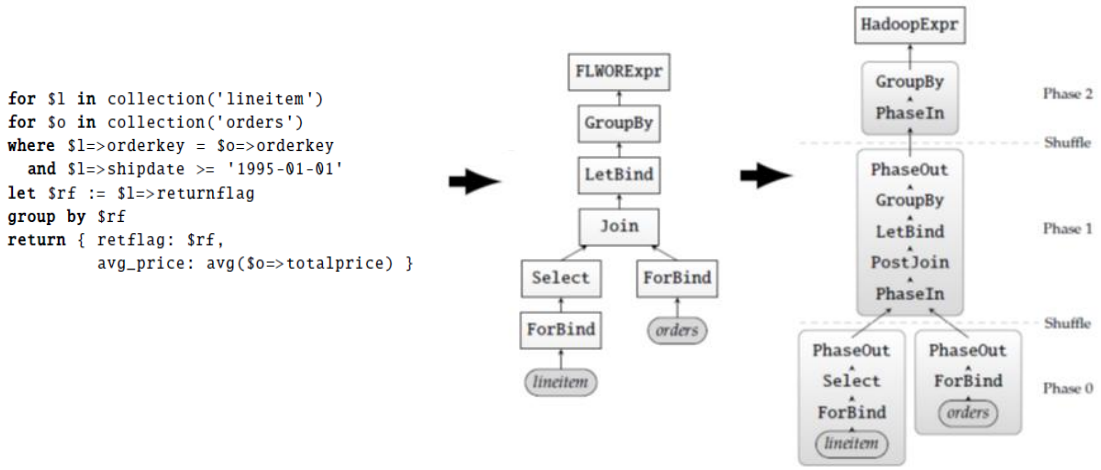


Figura 49. Tradução de XQuery para o modelo abstrato de MapReduce (SAUER; BÄCHLE; HÄRDER, 2013, adaptado)

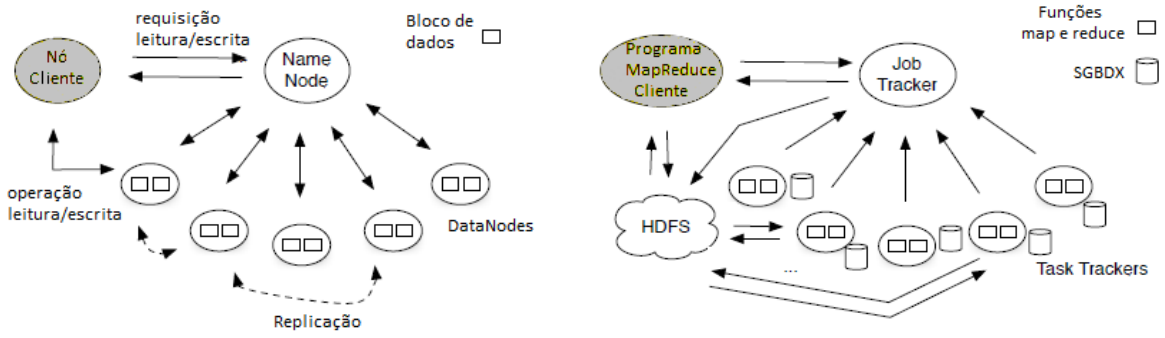
A abordagem possui uma limitação para o caso de processamento de dados XML: o sistema só trabalha com coleções de documentos, uma vez que o processamento de documentos muito grandes culminaria na necessidade de munir o processador com informações sobre a fragmentação dos dados nos blocos, e dados sobre o esquema utilizado, o que nem sempre existe. Desta forma, este trabalho não consegue manipular documentos XML grandes, que necessitariam de fragmentação. Outro ponto a considerar sobre esta abordagem é quanto ao desempenho, que foi similar ao de outras abordagens, como o Hive, mas não obteve ganhos.

7.2.4 BASEX MAPREDUCE

Lewandowski (2012) propôs uma solução para processamento de consultas XQuery em ambiente distribuído que estende a funcionalidade de processamento do SGBDX BaseX (GRÜN; HOLUPIREK; SCHOLL, 2007) com a capacidade de distribuição do *framework* Hadoop, que emprega o modelo MapReduce (DEAN; GHEMAWAT, 2008). No caso, o

processamento mencionado considera que os dados a serem processados são grandes coleções de pequenos documentos XML.

O carregamento dos dados para processamento é realizado em duas fases. Na primeira, os arquivos XML de entrada são distribuídos pelos nós do cluster. Realizando interações com o NameNode do *framework* Hadoop, a abordagem requisita a alocação de blocos livres e realiza o armazenamento dos documentos XML nestes blocos. Os blocos de dados criados são automaticamente replicados devido à configuração padrão do *framework*, o que proporciona a tolerância a falhas da solução. Na segunda fase, um *job* MapReduce é iniciado para cada arquivo de entrada. A função *map* recebe o conteúdo do arquivo XML e o armazena na instância do SGBDX local, enquanto a função *reduce* é responsável por escrever uma lista com todos os arquivos XML armazenados. As fases 1 e 2 do carregamento de dados estão ilustradas na Figura 50.



(a) **(b)**

Figura 50. Fase 1 (a) e Fase 2 (b) do carregamento de dados (LEWANDOWSKI, 2012, adaptado)

Para efetivamente executar a consulta, a solução faz uso de outro *job* MapReduce, no qual a função *map* realiza a consulta no SGBDX local e salva o resultado intermediário em um arquivo temporário. A função *reduce* se encarrega de consolidar os resultados intermediários. No entanto, o trabalho não apresenta detalhes sobre como esta consolidação é realizada.

Uma desvantagem da solução proposta é que ela somente se aplica a coleções de pequenos documentos, uma vez que não executa a fragmentação física do documento XML, mas somente os distribui em diferentes blocos de dados do HDFS. Além disso, esta prática, aliada à necessidade que a solução tem de executar um *job* MapReduce para cada arquivo, torna o processamento bastante oneroso, sendo este fato comentado pelo autor.

7.2.5 MRPARBOX

O MRParBoX (CONG *et al.*, 2012) é um algoritmo proposto para a avaliação de consultas XPath booleanas em um ambiente distribuído, utilizando o modelo MapReduce (DEAN; GHEMAWAT, 2008) e a técnica de avaliação parcial (JONES, 1996). Esta técnica consiste em enviar a consulta completa para todos os nós (com fragmentos da base de dados completa), os quais parcialmente a avaliam e enviam funções compactas a um coordenador. Este coordenador, por sua vez, avalia estas funções e as combina, produzindo um resultado final.

O processamento no MRParBoX ocorre em três etapas, sendo a primeira uma fase de configuração. Nesta fase, a consulta é lida e as informações sobre a fragmentação da árvore XML são carregadas. A segunda etapa é a fase de *map*. A função *map* recebe como entrada um par <chave, valor> em que a chave é o identificador de um fragmento XML e o valor é o seu próprio conteúdo. Realiza-se então o processamento paralelo da consulta XPath sobre cada fragmento, utilizando a estratégia de avaliação parcial. É importante mencionar que caso existam nós folha virtuais neste fragmento (que representam outro fragmento), a abordagem utiliza variáveis para que o resultado final seja montado na fase de *reduce*. Como cada fragmento contém parte da árvore, a avaliação parcial da consulta sobre esses fragmentos consiste no seu resultado parcial.

A função *reduce*, por sua vez, recebe como entrada um par formado por uma chave e uma lista dos resultados parciais associados àquela chave. Esta lista é obtida ao agrupar a saída da função de *map* pela chave intermediária. A função *reduce* extrai o conteúdo de cada resultado parcial como uma expressão e a avalia para compor o resultado final da consulta.

O algoritmo MRParBoX possui como principal limitação a restrição a consultas XPath booleanas, o que torna a sua aplicação bastante limitada. Os resultados dos experimentos também demonstram que a solução possui problemas de escalabilidade, pois: (1) com o aumento no número de nós, aumenta-se o custo de gerenciamento entre eles; (2) a solução emprega somente um *reducer* para consolidação do resultado final, o que limita o ganho com o paralelismo.

7.3 PAXQUERY

O PAXQuery (CAMACHO-RODRÍGUEZ *et al.*, 2015; CAMACHO-RODRÍGUEZ; COLAZZO; MANOLESCU, 2014) paraleliza a execução de consultas XQuery sobre grandes coleções de documentos XML. Este trabalho propõe uma arquitetura em camadas que transforma de forma eficiente um subconjunto da linguagem XQuery em planos de execução

do modelo de programação PACT (*Parallelization Contracts*) (BATTRÉ *et al.*, 2010), e utiliza o *framework* Apache Flink[13] para a execução do plano PACT sobre um cluster de computadores, de forma tolerante a falhas e escalável.

A Figura 51 apresenta a arquitetura em alto nível da abordagem. Inicialmente, um *Parser* XQuery é responsável por transformar a consulta de entrada em uma expressão algébrica equivalente, e criar o seu plano lógico. Este plano lógico, por sua vez, passa por otimizações como *push down projections* e *push down selections*, por meio da reescrita do plano de consulta para um plano alternativo equivalente. A tradução para o PACT emprega um mapeamento entre os operadores definidos pelo modelo PACT e os operadores algébricos, com o intuito de criar um plano PACT compilado. O PACT é uma generalização do modelo MapReduce, ampliando a gama de operadores, que antes era restrita a somente Map e Reduce. Um plano PACT é um grafo direcionado acíclico de operadores paralelos implícitos, que são otimizados e depois traduzidos para fluxos de dados paralelos explícitos pelo *framework* Flink.

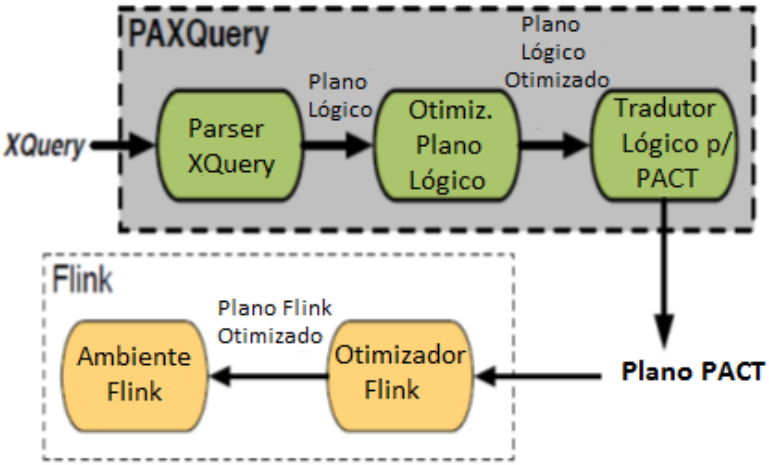


Figura 51. Arquitetura do PAXQuery (CAMACHO-RODRÍGUEZ *et al.*, 2015, adaptado)

Os resultados obtidos pelos autores demonstram que a abordagem é escalável para várias consultas experimentadas. Além disso, a comparação com uma implementação MapReduce de execução das consultas mostra que os tempos de execução são sempre menores no PAXQuery. Uma observação interessante feita pelo autor é a de que o BaseX, utilizado em uma implementação alternativa, funcionou de 2 a 5 vezes mais rápido ao ser integrado ao Flink ao invés do Hadoop, e que isso se deu pela escrita periódica de *checkpoints*

[13] https://flink.apache.org/

pelo *framework* Hadoop. Desta forma, entende-se que a utilização do Flink pode ser algo a ser verificado em futuras pesquisas sobre a execução distribuída de consultas XQuery.

7.4 VXQUERY

O VxQuery (CARMAN JR. *et al.*, 2015) é um processador XQuery escalável e *open-source*, construído sobre outros dois *frameworks open-source*: o Hyracks (BORKAR, V. *et al.*, 2011), um *framework* de execução paralela, e o Algebricks (BORKAR, VINAYAK *et al.*, 2015), um conjunto de ferramentas para compilação de linguagens agnóstico.

A arquitetura do VxQuery pode ser representada em 3 camadas, conforme ilustra a Figura 52. A camada superior recebe a consulta XQuery e constrói um plano lógico do Algebricks. Este plano Algebricks inicial é então otimizado localmente e traduzido para um plano físico do Algebricks, que é mapeado diretamente para um *job* Hyracks para ser executado.

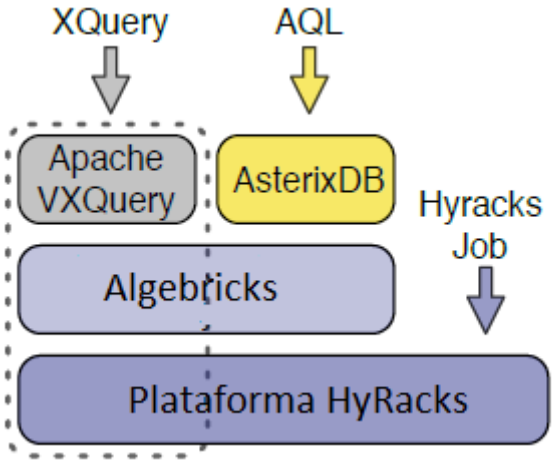


Figura 52. Arquitetura do VxQuery (CARMAN JR. *et al.*, 2015, adaptado)

O Hyracks é uma plataforma de execução paralela que permite executar fluxos de trabalho em paralelo sobre um cluster sem compartilhamento (*shared-nothing*). Ele foi concebido para ser independente de qualquer modelo de dados em particular, e processa dados em partições de bytes contíguos, movendo dados em janelas de tamanho fixo que contêm registros físicos. Ainda, o Hyracks define interfaces que permitem a um usuário especificar detalhes dos tipos de dados para comparação, *hashing*, serialização e de-serialização. Um *job* Hyracks é definido por um grafo direcionado acíclico de fluxo de dados, com operadores e conectores. Durante a execução, os operadores permitem que o cálculo consuma uma partição de entrada e produza uma partição de saída, enquanto os conectores redistribuem os dados entre as partições.

O Algebricks provê uma álgebra abstrata para a tradução e otimização de consultas paralelas, e complementa a plataforma Hyracks no baixo nível. As implementações de linguagens podem estender o modelo de sua camada algébrica para criar processadores paralelos sobre a plataforma Hyracks. Um sistema que utiliza o Algebricks para seu processamento de consultas começa com uma análise léxica da consulta. A consulta é então transformada em um grafo acíclico direcionado de fluxo de dados, ou plano de consulta, usando os operadores lógicos do Algebricks como uma linguagem ou álgebra intermediária. O otimizador então atua sobre o plano de consulta lógico, que é traduzido para o plano de consulta físico, que por sua vez, é também otimizado, considerando as características dos operadores, as propriedades das partições e a localização dos dados. A partir do plano físico, o *job* Hyracks é gerado para execução, que faz uso de funções especificamente definidas pelo VxQuery para o processamento da consulta XQuery. É importante mencionar que o VxQuery não lida com a fragmentação de documentos XML, considerando que a base de dados é formada por documentos XML não fragmentados, igualmente distribuídos pelo cluster.

Para avaliar a abordagem proposta, os autores realizaram experimentos de comparação tanto em um único nó quanto em um cluster. Os experimentos em um único nó foram realizados para atestar a capacidade de utilização de múltiplos núcleos, por meio da comparação dos tempos de execução de um processador XQuery bastante conhecido, o Saxon (KAY, 2001). Verificou-se que a abordagem conseguiu superar o conhecido processador, ao utilizar todos os núcleos presentes. Para os experimentos com múltiplos núcleos, observou-se também que a abordagem apresentou bons resultados de *speedup*. Na comparação com outras abordagens que executam sobre cluster MapReduce, observou-se uma melhora de 2,5 vezes no tempo de execução.

Uma das grandes desvantagens da abordagem é o fato desta não trabalhar com a fragmentação de documentos XML, o que torna impossível a sua aplicação para bases com documentos muito grandes, que necessariamente são fragmentados no armazenamento. Contudo, as melhoras obtidas pelos autores colocam o *framework* Hyracks como opção para novas abordagens de execução distribuída de consultas XQuery.

7.5 OUTRAS MELHORIAS PARA O PROCESSAMENTO MAPREDUCE

7.5.1 SKEWTUNE

O SkewTune (KWON *et al.*, 2012) é uma proposta para mitigar problemas de balanceamento de carga em *jobs* MapReduce. O objetivo do trabalho é mitigar o problema de

forma a não necessitar de nenhuma entrada extra do usuário, ser completamente transparente e causar um aumento mínimo do processamento regular do *job*. Isto é feito ao se substituir módulos centrais da implementação do MapReduce para lidar com a redistribuição dinâmica de trabalho entre os nós do cluster, sem exigir qualquer adequação dos *jobs* já existentes ou novos.

Em vários *jobs* MapReduce, o desbalanceamento de carga é bastante evidente, como mostra a Figura 53. Nela, vemos a execução do algoritmo PageRank em um *job* MapReduce, no qual algumas tarefas do tipo *map* necessitam muito mais tempo para completar a execução, o que acarreta um atraso em toda a execução do *job*, dado que as tarefas *reduce* só podem concluir a sua fase de *shuffle* após o término de todas as tarefas do tipo *map*. Com isso, temos um grande poder de processamento ocioso.

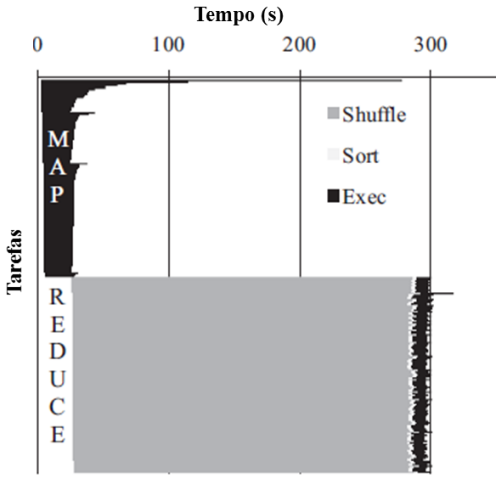


Figura 53. Execução do algoritmo PageRank em MapReduce (KWON *et al.*, 2012, adaptado)

A abordagem funciona da seguinte forma: quando um nó no cluster fica disponível, a tarefa com o maior tempo de término esperado é identificada, e os dados ainda não processados desta tarefa são então refragmentados dinamicamente em um modo que utiliza todos os nós disponíveis e preserva a ordenação dos dados de entrada, para que os dados de saída possam ser reconstruídos por concatenação. A Figura 54 apresenta a arquitetura proposta, construída a partir da arquitetura do Hadoop, implementação *open-source* do MapReduce. Ela é consistida por um *JobTracker* e um *TaskTracker*, análogos aos presentes no Hadoop, com a diferença que os processos criados para executar as tarefas reportam não só para o *TaskTracker*, mas também para um novo módulo de monitoramento de tarefas do *SkewTune*. O *JobTracker* do *SkewTune* serve como um coordenador e é responsável por detectar e mitigar os desbalanceamentos nos *jobs* enviados por ele.

Quando um desbalanceamento é detectado em uma tarefa do tipo *map*, a abordagem realiza uma decisão baseada em custo, com heurísticas lineares para decidir se é necessária a redistribuição do trabalho, e pede a esta tarefa para pausar. É então feita uma divisão dos dados ainda não processados, por meio da repartição de intervalos de chaves, utilizando o número de processadores prontamente disponíveis para execução. Definidos estes novos fragmentos, a abordagem lança novas tarefas, chamadas de mitigadoras para realizar o processamento. Para redistribuir uma tarefa do tipo *reduce*, a abordagem substitui o comportamento padrão de leitura dos resultados intermediários para manter controle sobre quais itens ainda não foram lidos, de maneira a reparti-los entre novas tarefas do tipo *reduce* (caso possível).

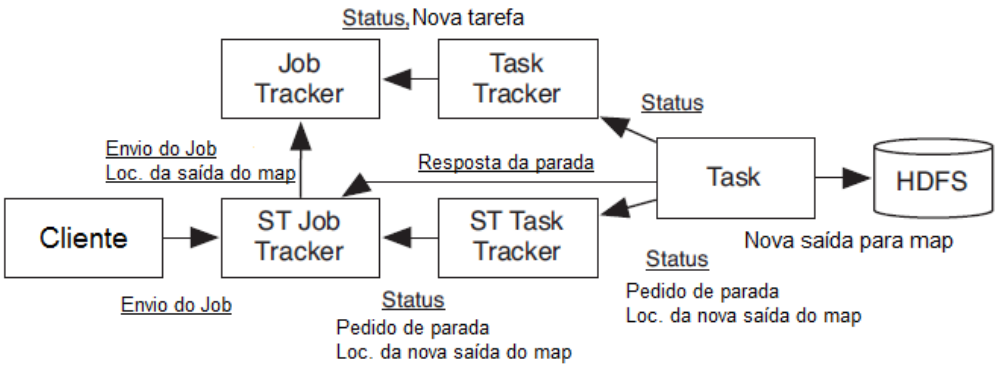


Figura 54. Arquitetura do SkewTune (KWON *et al.*, 2012, adaptado)

Os experimentos conduzidos no trabalho demonstram que a abordagem pode ajudar a reduzir o tempo de execução de *jobs* MapReduce em até 4 vezes, além de reduzir a variabilidade de execução. Nos casos em que o *job* não apresenta desbalanceamento, os autores também demonstram que a abordagem quase não acrescenta processamento extra. Uma das limitações da abordagem, entretanto, é que ela se aplica somente a *jobs* que processam dados de arquivos presentes no HDFS, ou seja, *jobs* que sobrescrevem o comportamento padrão do Hadoop de ler dados do HDFS não podem se beneficiar com a técnica.

7.5.2 FP-HADOOP

O FP-Hadoop (LIROZ-GISTAU; AKBARINIA; VALDURIEZ, 2015) é um sistema baseado no *framework* Hadoop que tem por objetivo otimizar o tempo de execução de *jobs* que em uma execução Hadoop padrão apresentariam problema de distribuição de carga entre os nós do cluster, devido à distribuição não uniforme dos dados. Os autores propõem uma nova fase de processamento, executada entre a fase de *Map* e a fase de *Reduce*, chamada de

Intermediate Reduce (IR). Nesta nova fase, blocos de valores intermediários, construídos dinamicamente, são processados paralelamente por tarefas constituídas por uma função de *reduce* intermediário, definida pelo usuário. Desta forma, mesmo se todos os valores intermediários gerados pelas tarefas do tipo *map* estiverem associados a um único valor de chave, a parte principal do trabalho de *reduce* pode ser feita em paralelo usando os recursos computacionais de todos os nós disponíveis.

A fase de *Map* do FP-Hadoop é quase idêntica àquela do Hadoop tradicional. A única diferença é que no FP-Hadoop a saída da tarefa do tipo *map* é manipulada e a combinação (*merge*) dos pares <chave, valor> gerados não ocorre. Ao invés disso, estes pares <chave, valor> são agrupados, gerando fragmentos de resultados intermediários. Informações sobre estes fragmentos são enviadas ao nó principal do cluster, que as utiliza para criar partições de resultados intermediários, que são então processadas por tarefas de *reduce* intermediário. Estas tarefas são criadas pelo sistema FP-Hadoop utilizando uma estratégia de escalonamento, e funcionam de maneira iterativa, ou seja, os pares <chave, valor> resultantes das tarefas de *reduce* intermediário são utilizados para criar novos fragmentos e partições de resultado intermediário, que poderão ou não serem reprocessadas por uma nova tarefa de *reduce* intermediário. No caso, o usuário pode definir o número máximo de iterações, ou deixá-lo variável, utilizando como critério de parada a razão entre o tamanho da entrada e saída da função de *reduce* intermediário. Percebe-se que uma tarefa de *reduce* intermediário tem papel semelhante àquele do *Combiner* existente no *framework* Hadoop, com a diferença de trabalhar com valores oriundos de diferentes tarefas do tipo *map*, e não somente de uma. O processamento segue então para a fase de *Reduce* Final, em que a saída da fase anterior é fragmentada, e tarefas de *reduce* final recuperam as partições intermediárias correspondentes, combinando-as, e gerando o resultado final. Como no FP-Hadoop as tarefas de *reduce* final recebem os valores sobre os quais as tarefas de *reduce* intermediários já trabalharam, a carga de processamento nas tarefas de *reduce* final são geralmente muito menores do que aquelas das tarefas *reduce* do Hadoop.

Os autores fizeram uma comparação entre a execução de um *job* Hadoop e a execução de um *job* FP-Hadoop equivalente, e obtiveram reduções de até 5 vezes no tempo total de execução. Se comparado somente o tempo de execução da fase de *reduce*, a redução foi de até 10 vezes. Entretanto, os autores não fornecem muitos detalhes sobre os experimentos. Outro ponto a se considerar é que as modificações realizadas no Hadoop exigem que o usuário escreva uma nova função ao criar o seu *job* de processamento. Com isso, os *jobs* Hadoop já existentes não são compatíveis com a nova abordagem.

7.6 CONSIDERAÇÕES FINAIS

O processamento distribuído de dados é um campo de estudos vasto e altamente ativo, e isto se deve aos avanços na disponibilização e manutenção da infraestrutura necessária, (grandes clusters e grids, compostos por milhares de computadores), bem como na aquisição e disponibilização cada vez maior de grandes massas de dados. É também possível notar a grande atividade na área por meio do surgimento de inúmeros *frameworks* de processamento distribuído de dados (Hadoop, Spark, Hyracks e Flink, como exemplo).

No caso específico do processamento distribuído de dados XML, é interessante notar o desafio que tal modelo impõe às técnicas existentes. Devido à sua natureza semiestruturada, a divisão do esforço de processamento pela fragmentação de dados é bastante complexa, ao ponto de diversas abordagens propostas simplesmente não a considerarem. É o caso de técnicas como a recente VxQuery (CARMAN JR. *et al.*, 2015) e da proposta de Lewandowski (2012). Neste sentido, o PartiX-VP (RODRIGUES; BRAGANHOLO; MATTOSO, 2011) apresenta a fragmentação virtual, uma proposta viável que se aplica tanto para bases de dados formadas por coleções de documentos quanto para bases de dados formadas por somente um documento.

A Tabela 13 apresenta um resumo dos trabalhos relacionados e de suas características, juntamente com a abordagem proposta deste trabalho, para facilitar a comparação. A linguagem de consulta disponibilizada aos usuários é uma destas características. No VPHadoop e no Partix-EVP, a linguagem oferecida é a XQuery, linguagem padrão e amplamente conhecida para consultas sobre dados no formato XML. Outras abordagens empregam linguagens menos expressivas (XPath) ou determinam a sua própria linguagem (ChuQL), o que pode não ser desejável. O tipo de coleção de documentos XML suportado (SD ou MD) está intimamente ligado com a capacidade da abordagem em lidar com a fragmentação do modelo XML. Muitas abordagens não lidam com a fragmentação dos documentos, e ficam, portanto, limitadas a utilizar somente coleções de documentos. Estas abordagens impõem limites no tamanho do arquivo para garantir que ele seja armazenado em somente um bloco do HDFS, ou que ele caiba inteiramente na memória. Neste ponto, as abordagens que utilizam fragmentação virtual se destacam pela capacidade de lidar com ambos os tipos.

O armazenamento no SGBDX e seu emprego durante a execução da consulta são outros dois fatores considerados. O armazenamento é feito no SGBDX somente nas abordagens que utilizam a fragmentação virtual. A abordagem BaseX MapReduce também

faz uso deste tipo de sistema para realizar a execução, mas considera o carregamento dos arquivos como etapa da execução, de maneira que os dados não são reaproveitados entre as execuções.

Tabela 13. Características das abordagens

Característica	**PartiX-VP**	**ChuQL**	**HadoopXML**	**BrackitMR**	**BaseX MapReduce**	**MRParBox**	**PAXQuery**	**VxQuery**	**VPHadoop**	**Partix-EVP**
Linguagem de consulta	XQuery	ChuQL	XPath	XQuery	XQuery	XPath bool.	XQuery	XQuery	XQuery	XQuery
Tipo de coleção	SD e MD	MD	SD	MD	MD	SD ou MD	MD	MD	SD ou MD	SD ou MD
Armazenamento dos dados de entrada	SGBDX	HDFS	HDFS	HDFS	HDFS	HDFS	HDFS	Hyracks	SGBDX	SGBDX
Emprega SGBDX	Sim	Não	Não	Não	Sim	Não	Não	Não	Sim	Sim
Fragmentação XML	Virtual	Não	Física	Não	Não	Física	Não	Não	Virtual	Virtual
Framework de distribuição	MPI	Hadoop	Hadoop	Hadoop	Hadoop	Hadoop	Flink	HyRacks	Hadoop	MPI
Balanceamento de carga	Não	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Quanto ao balanceamento de carga, devido ao fato de não empregar qualquer *framework* de execução distribuída, o PartiX-VP é o único que não tem essa característica presente. Neste ponto, cabe mencionar que, mesmo tendo a distribuição entre os nós realizada e coordenada pelo *framework*, dependendo dos dados de entrada e da consulta realizada, ainda pode haver distorções, o que é insumo para novas pesquisas, como vimos com o SkewTune e o FP-Hadoop.

No que diz respeito ao *framework* para distribuição de processamento, percebe-se cada vez mais a tendência a abstrair os controles e falhas inerentes da distribuição para prover um formato de processamento mais simples e inerentemente tolerante a falhas. É o caso de *frameworks* como Hadoop, Spark e Hyracks. Neles, o usuário se beneficia automaticamente de uma camada de distribuição tolerante a falhas, e que já provê o balanceamento de carga entre os nós envolvidos, desde que o usuário respeite o modelo de programação empregado. Este é o caso da escolha do modelo MapReduce e do *framework* Hadoop na proposta VPHadoop. Contudo, ao mesmo tempo em que permitem usuários estruturar um processamento paralelo e executá-lo com certa facilidade, tais abstrações providas por estes *frameworks* adicionam um processamento extra que pode causar um impacto significativo no tempo de execução, seja porque não houve um correto emprego do modelo, ou simplesmente porque o tamanho do problema não justifica o seu uso.

Finalmente, percebe-se que atualmente existem muitos outros *frameworks* e modelos que podem ser empregados na execução distribuída de consultas XQuery. Em alguns casos, os autores alegam ganhos de até 100 vezes sobre o tempo de execução de um *job* no *framework* Hadoop, como é o caso do Spark. Propõe-se que, como trabalho futuro, a integração da fragmentação virtual com este novo *framework* seja avaliada. Ainda, mesmo o Hadoop possui algumas adaptações que merecem um estudo detalhado, principalmente em casos como o do VPHadoop, que faz uso de somente uma tarefa *reduce*. Alternativas como o SkewTune e o FP-Hadoop podem auxiliar a diminuir o tempo de processamento dos *jobs* Hadoop que empregam a fragmentação virtual para executar consultas XQuery de forma distribuída.

CAPÍTULO 8 – CONCLUSÕES E TRABALHOS FUTUROS

8.1 CONSIDERAÇÕES FINAIS

A literatura indica que a distribuição da execução de consultas XQuery entre diversos nós de um cluster é capaz de diminuir o tempo total de execução (ANDRADE *et al.*, 2006; CAMACHO-RODRÍGUEZ *et al.*, 2015; CARMAN JR. *et al.*, 2015; FIGUEIREDO; BRAGANHOLO; MATTOSO, 2010; RODRIGUES; BRAGANHOLO; MATTOSO, 2011). Em especial, a fragmentação virtual simples torna viável esta distribuição para o formato XML em cenários *ad-hoc*, nos quais não se tem conhecimento prévio das consultas de entrada, e apresenta bons resultados. Contudo, a distribuição não uniforme dos dados de entrada pode causar um desbalanceamento de carga nesta abordagem, que prevê somente uma distribuição estática da execução. Este desbalanceamento pode, por sua vez, impactar o tempo total de execução.

Neste trabalho, apresenta-se um estudo sobre o impacto da variação no número de fragmentos empregados pela técnica de fragmentação virtual simples sobre o formato XML, com a proposta de duas novas abordagens que utilizam essa variação. Nestas novas propostas, o número de fragmentos gerados pela fragmentação virtual é maior que o número de nós processadores envolvidos e há uma distribuição dinâmica de execução das subconsultas que agem sobre cada um destes fragmentos.

A abordagem VPHadoop integra a nova proposta de fragmentação virtual simples com o modelo MapReduce de processamento, e faz uso do *framework* Hadoop para distribuir a execução de subconsultas e consolidar os resultados. Cada subconsulta é executada por uma tarefa do tipo *map*, e a consolidação do resultado final ocorre na etapa de *reduce*, que é composta de uma única tarefa. Desta forma, a distribuição da execução fica a cargo do *framework*.

Já no caso da abordagem Partix-EVP, opta-se por implementar a distribuição da execução diretamente sobre MPI, com um algoritmo mestre-escravo. Um dos nós é eleito o controlador, que fica dedicado em distribuir as subconsultas para execução nos demais nós processadores, conforme a disponibilidade destes. Ao ser notificado do término de uma execução em um nó, caso ainda haja subconsultas para executar, o nó controlador imediatamente envia uma subconsulta para processamento, evitando assim que este fique ocioso.

Ambas as abordagens se beneficiam de melhorias realizadas na técnica de fragmentação virtual simples e do seu encapsulamento em uma biblioteca, o que facilita o seu uso. A utilização de um catálogo da base dados para eliminar a necessidade de consultas ao SGBDX durante a fragmentação, e a determinação dinâmica da estratégia de composição do resultado final compõem estas melhorias. Com a utilização do catálogo proposto, é possível diminuir em até 99% o tempo de fragmentação. Já a flexibilização da estratégia de composição, que permite a aplicação de concatenação nos casos aplicáveis, chega a diminuir até 78% do tempo total de execução de uma consulta, sem causar impacto nos casos em que a coleção temporária é necessária.

Considerando a primeira questão de pesquisa apresentada no Capítulo 1, na qual se desejava saber como a variação do número de fragmentos impacta a execução de consultas utilizando a técnica de fragmentação virtual, a análise conduzida por este trabalho indicou que é possível diminuir o tempo total de execução de consultas de alto custo ao se aumentar o número de fragmentos. Em contrapartida, consultas de baixo custo tendem a apresentar um alto impacto no tempo de execução com esse aumento, em decorrência do processamento extra requerido para se distribuir a execução e consolidar o resultado final. É necessário, portanto, escolher com cautela o número de fragmentos que vai se aplicar na execução de cada consulta.

A segunda questão de pesquisa deste trabalho indagava se era melhor utilizar a distribuição de um *framework* específico e deixá-la fora da abordagem, ou implementar um mecanismo próprio de distribuição. Neste aspecto, foi possível perceber que o *framework* Hadoop, apesar de prover automaticamente uma forma de distribuição da execução, acrescenta um processamento extra bastante considerável no caso da integração proposta com a fragmentação virtual simples. Em casos de consultas de alto custo, o processamento extra adicionado à abordagem VPHadoop pelo *framework* dilui-se e os tempos ficam similares àqueles obtidos com a abordagem Partix-EVP. Já em consultas de baixo custo, o peso do *framework* fica evidente. A abordagem Partix-EVP, por sua vez, foi capaz de apresentar um processamento extra bastante reduzido mesmo em casos de consultas de baixo custo, o que a coloca como opção mais viável. Com isso, conclui-se que, se o que se deseja é a diminuição no tempo total de execução, uma implementação mais enxuta da distribuição é a mais recomendada.

Na comparação entre abordagens, percebeu-se claramente a diferença ao se empregar um SGBDX que crie automaticamente seus índices. Em geral, a abordagem centralizada é bastante beneficiada e se mostra superior nos casos em que a quantidade de dados envolvida

não é muito grande. Foi também possível verificar que o emprego destes índices em otimizações internas pode entrar em conflito com a execução das subconsultas da fragmentação virtual, causando impactos severos no tempo total de execução, como é o caso de consultas sobre de bases de dados MD e consultas com junções cujo atributo de fragmentação virtual não esteja na primeira cláusula *for*.

O coeficiente de variação elevado obtido em várias execuções de consultas para os casos de bases de dados maiores indica que pode haver outros fatores que interfiram na execução da consulta além daqueles previstos pelas abordagens e metodologia utilizada nos experimentos, tais como a concorrência na utilização do armazenamento compartilhado para salvar os resultados parciais e o final.

Finalmente, a verificação da distribuição de carga durante a execução paralela mostrou que a abordagem Partix-EVP é capaz de melhorar a distribuição da execução. O maior número de fragmentos permite uma melhor utilização dos processadores, muito embora deva-se levar em consideração com um aumento no número de fragmentos pode elevar o tempo total de execução.

Os experimentos com a fragmentação virtual neste trabalho e as mudanças empregadas nas abordagens propostas mostram que esta técnica possui um grande potencial teórico de reduzir o tempo de execução de consultas *ad hoc* complexas e custosas, principalmente aquelas com junções e que envolvem uma grande quantidade de dados nas respostas. Contudo, na prática, a técnica pode enfrentar problemas com o alto consumo de memória do SGBDX, assim como conflitos com as otimizações internas que os SGBDX podem aplicar nas consultas XQuery.

8.2 LIMITAÇÕES

É importante mencionar algumas limitações observadas no decorrer do estudo realizado por este trabalho. A primeira delas se refere à reutilização da implementação original da técnica de fragmentação virtual simples. Como já mencionado no Capítulo 3, duas melhorias e várias correções foram feitas. Contudo, não houve tempo hábil para ampliar a gramática XQuery suportada pela solução. Com isto, consultas que utilizem algumas construções como filtros, cláusula *group by* e funções mais avançadas da XQuery não são suportadas. Entende-se que esta limitação precisa ser revista em um trabalho futuro.

Outra limitação que foi observada refere-se ao uso de armazenamento compartilhado nas arquiteturas propostas. Muito embora este armazenamento geralmente utilize soluções com taxas de transferências bastante altas, é bastante difícil afirmar que os tempos não são

afetados por outras tarefas em execução nos nós de processamento. Um estudo comparativo que considere abordagens sem compartilhamento (*shared-nothing*) seria importante.

No que se refere aos experimentos, a utilização do cluster SunHPC propiciou uma grande capacidade de paralelismo na execução, contudo o armazenamento local dos nós é compartilhado entre os 8 núcleos de processamento utilizados em cada nó. Entende-se que esta é uma limitação da avaliação experimental realizada, pois a execução de operações de escrita e leitura de um nó processador pode interferir nas operações de escrita e leitura do outro, muito embora o SGBDX utilizado já tenha embutido mecanismos de paralelismo inter-consultas. Novamente, ambientes sem compartilhamento (*shared-nothing*) podem ser utilizados para avaliar este impacto.

Finalmente, não houve tempo suficiente para reexecutar as consultas que tiveram coeficiente de variação acima de 20% para as bases de 10GB, o que limitou as análises de execução de consultas sobre este tamanho de base. Entende-se que é necessária uma reavaliação destes tempos, o que permitiria que as conclusões possam ser tiradas com maior fidelidade.

8.3 TRABALHOS FUTUROS

Como citado na Seção 6.5, as abordagens propostas neste trabalho utilizam um número de fragmentos flexível para a fragmentação virtual, contudo este é um parâmetro que deve ser inserido na execução de cada consulta. Julga-se possível definir heurísticas que possam classificar automaticamente uma consulta e aplicar um número de fragmentos baseado em suas características e no número de processadores existentes. É, portanto, um trabalho futuro proposto definir estas heurísticas.

A solução de catálogo proposta neste trabalho na Seção 3.1.1 toma como premissa a base de dados ser somente leitura, ou seja, atualizações nesta base invalidam o catálogo criado e impossibilitam a sua utilização. É possível, contudo, definir mecanismos que permitam a identificação de mudanças e a atualização do catálogo de forma automática, sendo este outro trabalho proposto.

As abordagens propostas neste trabalho usem a fragmentação virtual, que se baseiam na replicação total dos dados, que podem apresentar um alto custo de armazenamento. No mundo relacional, uma estratégia que combina a fragmentação virtual com a fragmentação física já foi proposta (FURTADO *et al.*, 2005), e apresentou resultados semelhantes ao da fragmentação virtual, com uma grande redução no armazenamento necessário. Esta estratégia,

denominada fragmentação híbrida, ainda não foi aplicada ao modelo XML, sendo este um trabalho futuro que merece atenção.

Muito embora a ordem dos elementos em um documento XML seja relevante, em algumas aplicações pode ser possível desprezar essa restrição durante a execução das consultas. Desta forma, poder-se-ia paralelizar também a composição do resultado final. Esta e outras simplificações no modelo XML podem ser analisadas, e novas abordagens poderiam obter grandes ganhos de desempenho. Este análise é, portanto, outro trabalho futuro proposto.

Nas execuções realizadas para este trabalho, nota-se o alto coeficiente de variação observado nos cenários com bases de dados maiores. Tal fato pode ser melhor avaliado em um trabalho futuro para identificar possíveis problemas nas abordagens propostas, bem como possíveis soluções.

Também nas execuções, foi identificado que o SGBDX BaseX apresentou problemas quando a quantidade de dados envolvida era grande. Ainda, algumas otimizações internas desse SGBDX se mostraram conflitantes com a técnica de fragmentação virtual utilizada neste trabalho. Recomenda-se um estudo mais detalhado destes conflitos, utilizando também outros SGBDX para verificar se é possível identificar outros sistemas que possam ser empregados com a fragmentação virtual. É importante mencionar, contudo, que se busque a utilização de todas as técnicas de otimização e índices na execução disponíveis naqueles sistemas, tanto nas execuções centralizadas quanto nas execuções distribuídas, para que se faça uma comparação justa.

Com o surgimento de novos *frameworks* que possibilitam o processamento paralelo de grandes quantidades de dados, é possível que a integração da fragmentação virtual sobre estes venha a possibilitar novos ganhos no tempo de execução. É proposto, portanto, um estudo mais abrangente que verifique a viabilidade desta integração com outros *frameworks* e avalie os resultados.

REFERÊNCIAS

ABITEBOUL, S.; BUNEMAN, P.; SUCIU, D. *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

AKAL, F.; BÖHM, K.; SCHEK, H.-J. OLAP Query Evaluation in a Database Cluster: a performance study on intra-query parallelism. In: EAST EUROPEAN CONFERENCE ON ADVANCES IN DATABASES AND INFORMATION SYSTEMS (ADBIS), ADBIS ’02, 2002, London, UK. *Anais*... London, UK: Springer-Verlag, 2002. p. 218–231.

ANDRADE, A. *et al.* Efficiently Processing XML Queries over Fragmented Repositories with PartiX. In: INTERNATIONAL WORKSHOP ON DATABASE TECHNOLOGIES FOR HANDLING XML INFORMATION ON THE WEB (DATAX), 2006, Munich, Germany. *Anais*... Munich, Germany: [s.n.], 2006. p. 150–163.

ANDRADE, A. *et al. PartiX: processing XQuery queries over fragmented XML repositories*. Technical Report, nº ES-691. Rio de Janeiro, Brazil: Universidade Federal do Rio de Janeiro, 2005.

BATTRÉ, D. *et al.* Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In: 1ST ACM SYMPOSIUM ON CLOUD COMPUTING, SoCC ’10, 2010, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2010. p. 119–130.

BORKAR, V. *et al.* Algebricks: A Data Model-agnostic Compiler Backend for Big Data Languages. In: 6TH ACM SYMPOSIUM ON CLOUD COMPUTING, SoCC ’15, 2015, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2015. p. 422–433.

BORKAR, V. *et al.* Hyracks: A flexible and extensible foundation for data-intensive computing. In: 2011 IEEE 27TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), abr. 2011, [S.l: s.n.], abr. 2011. p. 1151–1162.

BRAGANHOLO, V.; MATTOSO, M. A Survey on XML Fragmentation. *SIGMOD Record*, v. 43, n. 3, p. 24–35, dez. 2014.

BRUNO, N.; KOUDAS, N.; SRIVASTAVA, D. Holistic twig joins: optimal XML pattern matching. In: SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2002, [S.l.]: ACM, 2002. p. 310–321.

CAMACHO-RODRÍGUEZ, J. *et al.* PAXQuery: Parallel Analytical XML Processing. SIGMOD ’15, 2015, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2015. p. 1117–1122.

CAMACHO-RODRÍGUEZ, J.; COLAZZO, D.; MANOLESCU, I. PAXQuery: A Massively Parallel XQuery Processor. DanaC’14, 2014, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2014. p. 6:1–6:4.

CARMAN JR., E. P. *et al.* Apache VXQuery: A Scalable XQuery Implementation. *arXiv:1504.00331 [cs]*, arXiv: 1504.00331, 1 abr. 2015.

CHOI, H. *et al.* HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT (CIKM), 2012, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2012. p. 2737–2739.

CONG, G. *et al.* Partial Evaluation for Distributed XPath Query Processing and Beyond. *ACM Trans. Database Syst.*, v. 37, n. 4, p. 32:1–32:43, dez. 2012.

DAMIGOS, M.; GERGATSOULIS, M.; PLITSOS, S. Distributed Processing of XPath Queries Using MapReduce. *New Trends in Databases and Information Systems*. Advances in Intelligent Systems and Computing. [S.l.]: Springer International Publishing, 2014. p. 69–77.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: SYMPOSIUM ON OPERATING SYSTEMS DESIGN & IMPLEMENTATION (OSDI), OSDI'04, 2004, Berkeley, CA, USA. *Anais*... Berkeley, CA, USA: USENIX Association, 2004. p. 137–150.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, v. 51, n. 1, p. 107–113, jan. 2008.

EVERITT, B. S. *Cambridge Dictionary of Statistics*. [S.l.]: Cambridge University Press, 1998.

FADIKA, Z.; HEAD, M. R.; GOVINDARAJU, M. Parallel and distributed approach for processing large-scale XML datasets. In: 2009 10TH IEEE/ACM INTERNATIONAL CONFERENCE ON GRID COMPUTING, 2009, [S.l: s.n.], 2009. p. 105–112.

FERNÁNDEZ, M. *et al. XQuery 1.0 and XPath 2.0 Data Model. W3C Recommendation.* [S.l: s.n.], 2004.

FIGUEIREDO, G.; BRAGANHOLO, V.; MATTOSO, M. Processing Queries over Distributed XML Databases. *Journal of Information and Data Management (JIDM)*, v. 1, n. 3, p. 455–470, 2010.

FOMICHEV, A.; GRINEV, M.; KUZNETSOV, S. Sedna: A Native XML DBMS. In: INTERNATIONAL CONFERENCE ON CURRENT TRENDS IN THEORY AND PRACTICE OF COMPUTER SCIENCE (SOFSEM), 2006, Merin, Czech Republic. *Anais*... Merin, Czech Republic: [s.n.], 2006. p. 272–281.

FURTADO, C. *et al.* Physical and virtual partitioning in OLAP database clusters. In: 17TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2005. SBAC-PAD 2005, 2005, [S.l: s.n.], 2005. p. 143–150.

GOLDMAN, R.; WIDOM, J. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES (VLDB), 1997, Athens, Greece. *Anais*... Athens, Greece: [s.n.], 1997. p. 436–445.

GRÜN, C.; HOLUPIREK, A.; SCHOLL, M. H. Visually Exploring and Querying XML with BaseX. 2007, [S.l: s.n.], 2007. p. 629–632.

JANSEN, B. J. Search log analysis: What it is, what's been done, how to do it. *Library & Information Science Research*, v. 28, n. 3, p. 407–432, 2006.

JONES, N. D. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, v. 28, n. 3, p. 480–503, 1996.

KAY, M. *Saxon XSLT and XQuery Processor*. Disponível em: <http://sourceforge.net/projects/saxon>. Acesso em: 11 jun. 2016.

KHATCHADOURIAN, S.; CONSENS, M. P.; SIMÉON, J. Having a ChuQL at XML on the Cloud. 2011, [S.l: s.n.], 2011.

KIDO, K.; AMAGASA, T.; KITAGAWA, H. Processing XPath Queries in PC-clusters Using XML Data Partitioning. 2006, [S.l.]: IEEE, 2006. p. 11–16.

KWON, Y. *et al.* SkewTune: Mitigating Skew in Mapreduce Applications. SIGMOD '12, 2012, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2012. p. 25–36.

LEE, K.-H. *et al.* Parallel data processing with MapReduce: a survey. *SIGMOD Record*, v. 40, n. 4, p. 11–20, jan. 2012.

LEWANDOWSKI, L. *Using Map and Reduce for Querying Distributed XML Data*. 2012. Master's Thesis – Master's Thesis. University of Konstanz, 2012.

LIMA, A. *Paralelismo Intra-consulta em Clusters de Bancos de Dados*. 2004. 105 f. Tese – Universidade Federal do Rio de Janeiro, COPPE, Programa de Pós-graduação de Engenharia de Sistemas e Computação, Rio de Janeiro, 2004.

LIMA, A. *et al.* Parallel OLAP query processing in database clusters with data replication. *Distributed Parallel Databases*, v. 25, n. 1–2, p. 97–123, abr. 2009.

LIMA, A.; MATTOSO, M.; VALDURIEZ, P. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. *Journal of Information and Data Management (JIDM)*, v. 1, n. 1, p. 75–88, 2010.

LIROZ-GISTAU, M.; AKBARINIA, R.; VALDURIEZ, P. FP-Hadoop: Efficient Execution of Parallel Jobs over Skewed Data. *Proc. VLDB Endow.*, v. 8, n. 12, p. 1856–1859, Agosto 2015.

MALHOTRA, A. *et al. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. . [S.l: s.n.]. , 2010

MAYER-SCHNBERGER, V.; CUKIER, K. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. [S.l.]: Eamon Dolan/Houghton Mifflin Harcourt, 2014.

ÖZSU, M. T.; VALDURIEZ, P. *Principles of Distributed Database Systems*. 3. ed. [S.l.]: Prentice Hall, 2011.

PAPARIZOS, S. *et al.* Tree logical classes for efficient evaluation of XQuery. 2004, Paris, France. *Anais*... Paris, France: ACM, 2004. p. 71–82.

RANGANATHAN, P. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer*, v. 44, n. 1, p. 39–48, jan. 2011.

RODRIGUES, C. *Processamento Paralelo de Consultas sobre Fragmentos Virtuais de Documentos XML*. 2011. 105 f. Dissertação de Mestrado – Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2011.

RODRIGUES, C.; BRAGANHOLO, V.; MATTOSO, M. Virtual Partitioning ad-hoc Queries over Distributed XML Databases. *Journal of Information and Data Management (JIDM)*, v. 2, n. 3, p. 495–510, 2011.

SAUER, C.; BÄCHLE, S.; HÄRDER, T. BrackitMR: flexible XQuery processing in MapReduce. *Web-Age Information Management*. [S.l.]: Springer, 2013. p. 806–808.

SAUER, C.; HÄRDER, T. Compilation of Query Languages into MapReduce. *Datenbank-Spektrum*, p. 1–11, 2013.

SAX. Disponível em: <http://www.saxproject.org/>. Acesso em: 11 jun. 2016.

SCHMIDT, A. *et al.* XMark: A Benchmark for XML Data Management. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES (VLDB), 2002, Hong Kong, China. *Anais*... Hong Kong, China: [s.n.], 2002. p. 974–985.

SHAFI, A.; CARPENTER, B.; BAKER, M. Nested parallelism for multi-core HPC systems using Java. *Journal of Parallel and Distributed Computing*, v. 69, n. 6, p. 532–545, jun. 2009.

SHIM, J. P. *et al.* Past, present, and future of decision support technology. *Decision Support Systems*, Decision Support System: Directions for the Nest Decade. v. 33, n. 2, p. 111–126, jun. 2002.

SILVA, L. *PROCESSAMENTO DE CONSULTAS XML AD HOC EM AMBIENTES DISTRIBUÍDOS*. 2015. Exame de Qualificação – Universidade Federal Fluminense, Niterói, RJ - Brasil, 2015.

SILVA, L.; MATTOSO, M.; BRAGANHOLO, V. On the Performance of the Position() XPath Function. In: ACM SYMPOSIUM ON DOCUMENT ENGINEERING (DOCENG), 2013, Florence, Italy. *Anais*... Florence, Italy: [s.n.], 2013. p. 229–230.

VIRTUAL PARTITIONING. In: MATTOSO, M. *Encyclopedia of Database Systems*. [S.l.]: Springer, 2009. p. 3340–3341.

ANEXO A – CONSULTAS XQUERY UTILIZADAS NOS EXPERIMENTOS

Identificador	Consulta

C1	`<results> {`
	`  for $it in doc('auction.xml')/site/regions/samerica/item`
	`  for $pe in doc('auction.xml')/site/people/person`
	`  where $pe/profile/@income > 90000 and $pe/profile/interest/@category = $it/incategory/@category`
	`  return`
	`  <match>`
	`  <person>{$pe/name}</person>`
	`  <item>{$it/name}</item>`
	`  </match>`
	`} </results>`

C2	`<results> {`
	`  for $cat in doc('auction.xml')/site/categories/category`
	`  for $pe in doc('auction.xml')/site/people/person`
	`  where count($pe/profile/interest) > 3 and $pe/profile/interest/@category = $cat/@id`
	`  return`
	`  <match>`
	`  <person>{$pe/name}</person>`
	`  <category>{$cat/name}</category>`
	`  </match>`
	`} </results>`

C3	`<results> {`
	`  for $it in doc('auction.xml')/site/regions/samerica/item`
	`  for $pe in doc('auction.xml')/site/people/person`
	`  where $it/incategory/@category = "category250" and $pe/profile/interest/@category = $it/incategory/@category and $pe/profile/education = "Graduate School"`
	`  return`
	`  <match_cat_250>`
	`  {$pe}`
	`  </match cat 250>`
	`} </results>`

C4	`<results> {`
	`  for $pe in doc('auction.xml')/site/people/person`
	`  let $int := $pe/profile/interest`
	`  where $pe/profile/business = "Yes" and count($int) > 1`
	`  order by $pe/name`
	`  return`
	`  <person>`
	`  {$pe}`
	`  </person>`
	`} </results>`

C5	`<results> {`
	`  for $p in doc('auction.xml')/site/people/person`
	`  where $p/profile/@income > 10000`
	`  return`
	`  <people>`
	`  {$p}`
	`  </people>`
	`} </results>`

C6	`<results> {`
	`  for $p in doc('auction.xml')/site/people/person`
	`  let $e := $p/homepage`
	`  where count($e) = 0`
	`  return`
	`  <person_without_homepage>`
	`  {$p/name}`
	`  </person_without_homepage>`
	`} </results>`

C7	`<results> {`
	`  for $it in doc('auction.xml')/site/regions/africa/item`
	`  for $co in doc('auction.xml')/site/closed_auctions/closed_auction`
	`  where $co/itemref/@item = $it/@id and $it/payment = "Cash"`
	`  return`
	`  <item_cash>`

	{$co/price} {$co/date} {$co/quantity} {$co/type} {$it/payment} {$it/location} {$it/from} {$it/to} </item cash> } </results>
C8	<results> { for $op in doc('auction.xml')/site/open_auctions/open_auction where count($op/bidder) > 5 return <open_auctions_with_more_than_5_bidders> <auction> {$op} </auction> </open_auctions_with_more_than_5_bidders> } </results>

C9	<results> {
	let $p := doc('auction.xml')//closed auction
	return
	<summary>
	<total items>{count($p)}</total items>
	<avg price>{avg($p/price)}</avg price>
	</summary>
	} </results>

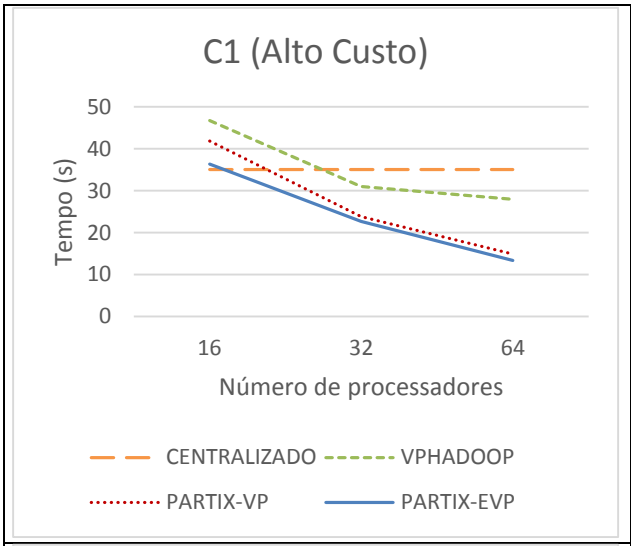
C10	<results> { for $it in doc('auction.xml')/site/regions/samerica/item for $pe in doc('auction.xml')/site/people/person where $pe/profile/interest/@category = $it/incategory/@category return <people> {$pe} </people> } </results>
C11	<results> { for $it in doc('auction.xml')/site/regions/asia/item for $pe in doc('auction.xml')/site/people/person where $pe/profile/interest/@category = $it/incategory/@category return <people> {$pe} </people> } </results>
C12	<results>{ for $oa in doc('auction.xml')/site/open_auctions/open_auction for $pe in doc('auction.xml')/site/people/person where $oa/type = "Featured" and $pe/@id = $oa/seller/@person order by $pe/address/country return <seller> <country>{ $pe/address/country }</country> <person>{ $pe/name }</person> <item id='{$oa/itemref/@item}'> <initial_bid>{$oa/initial}</initial_bid> <current_bid>{$oa/current}</current_bid> </item> </seller> }</results>

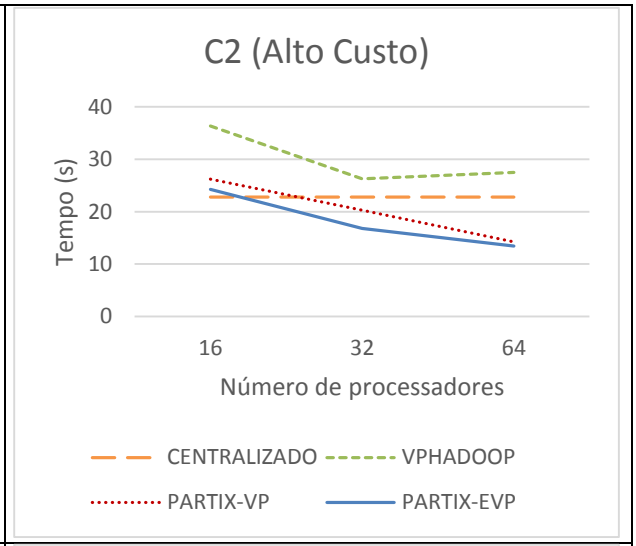
ANEXO B – ARQUIVO DE CONFIGURAÇÃO DO PROTÓTIPO VPHADOOP

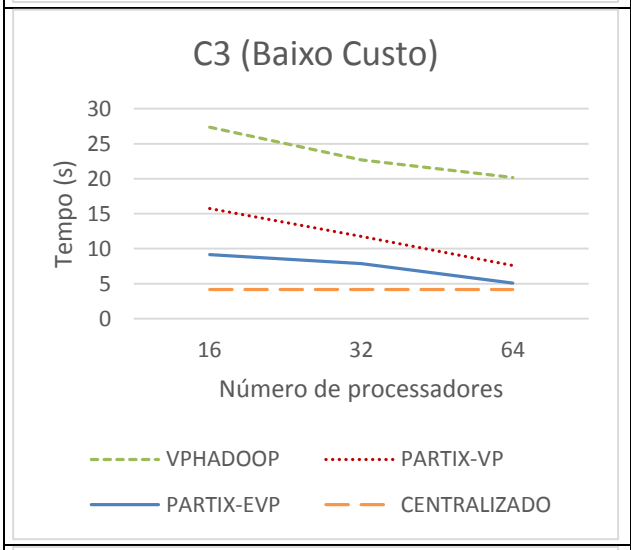
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>

  <!-- Configuração framework -->
  <property>
      <name>mapred.task.timeout</name>
      <value>0</value>
      <description>
        Configura o framework para não considerar um timeout de tarefas.
        Necessário pois algumas consultas demoram a retornar do SGBDX, dando a
        falsa impressão ao framework de que a execução travou.
      </description>
  </property>

  <!-- Configuração VPHADOOP -->
  <property>
      <name>vphadoop.svp.numsplits</name>
      <value>32</value>
      <description>
        Número de 'blocos de dados' (splits) que a abordagem VPHadoop deve
        criar.
      </description>
  </property>

  <property>
      <name>vphadoop.compress</name>
      <value>true</value>
      <description>
        Liga ou desliga a compressão dos dados durante a escrita dos
        resultados parciais no HDFS.
      </description>
  </property>

  <property>
      <name>vphadoop.svp.numrecords</name>
      <value>2</value>
      <description>
        Número de subconsultas que serão colocadas em um mesmo 'bloco de
        dados' (split)
      </description>
  </property>

  <property>
      <name>vphadoop.db.hostname</name>
      <value>localhost</value>
      <description>
        IP ou nome do host que está rodando a instância do SGBDX. Em nossa
        abordagem, é sempre localhost.
      </description>
  </property>

  <property>
      <name>vphadoop.db.port</name>
      <value>1984</value>
      <description>
        Porta em que o SGBDX está esperando conexões para realizar as
        consultas.
      </description>
  </property>

  <property>
      <name>vphadoop.db.username</name>
      <value>admin</value>
      <description>
        Nome de usuário utilizado para se conectar ao SGBDX
      </description>
  </property>
```

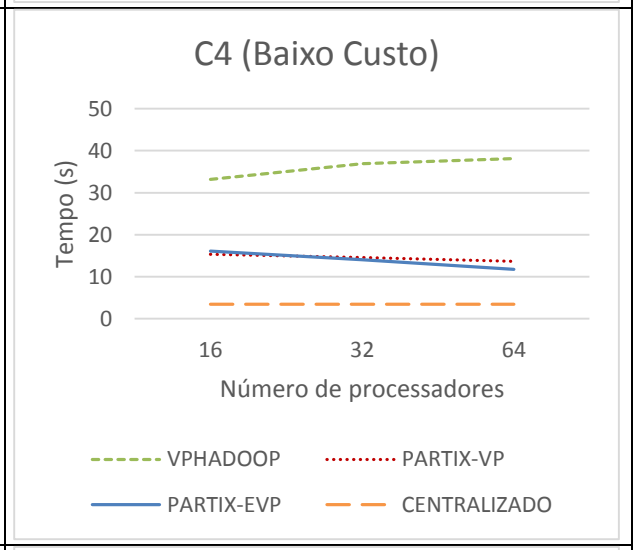
```
  <property>
      <name>vphadoop.db.password</name>
      <value>admin</value>
      <description>
        Senha do usuário utilizado para se conectar ao SGBDX
      </description>
  </property>

  <property>
      <name>vphadoop.db.database</name>
      <value>expdb</value>
      <description>
        Nome da base de dados no SGBDX sendo considerada na abordagem
      </description>
  </property>

  <property>
      <name>vphadoop.db.type</name>
      <value>BASEX</value>
      <description>
        Qual o SGBDX está sendo considerado. São suportados atualmente o
       'SEDNA' e o 'BASEX'
      </description>
  </property>

</configuration>
```

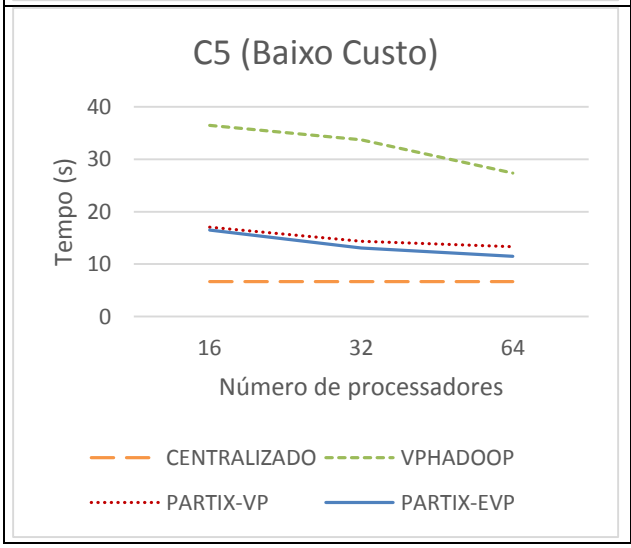
ANEXO C – GRÁFICOS DAS EXECUÇÕES PARA A BASE DE DADOS SD DE 1 GB

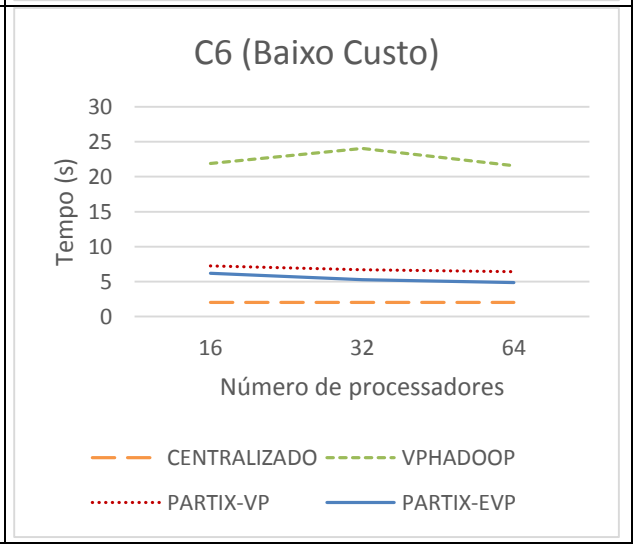


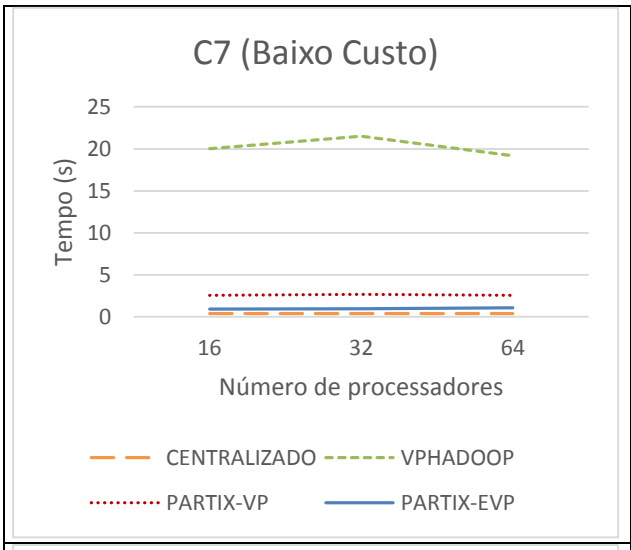


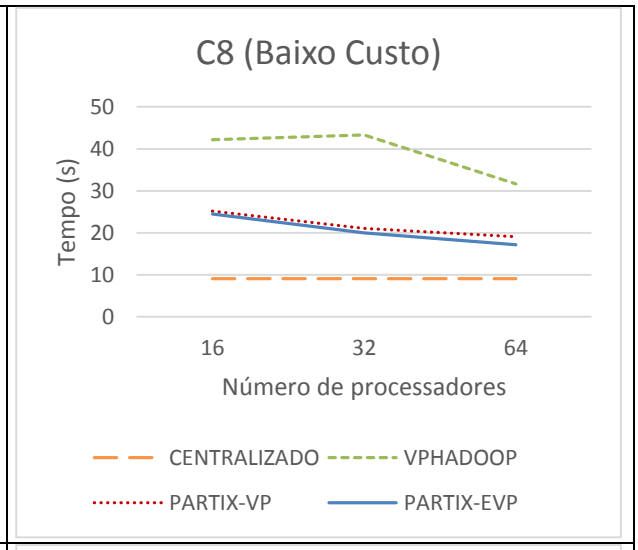


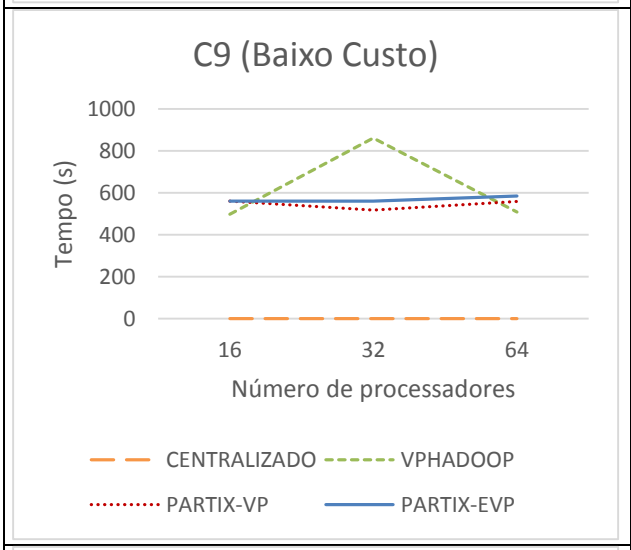


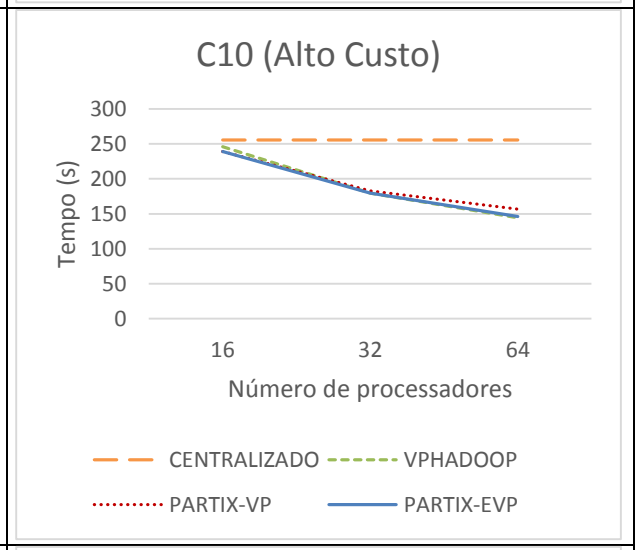


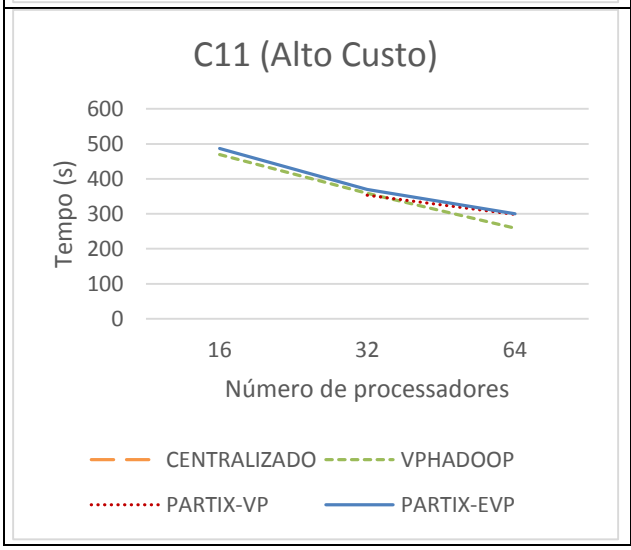


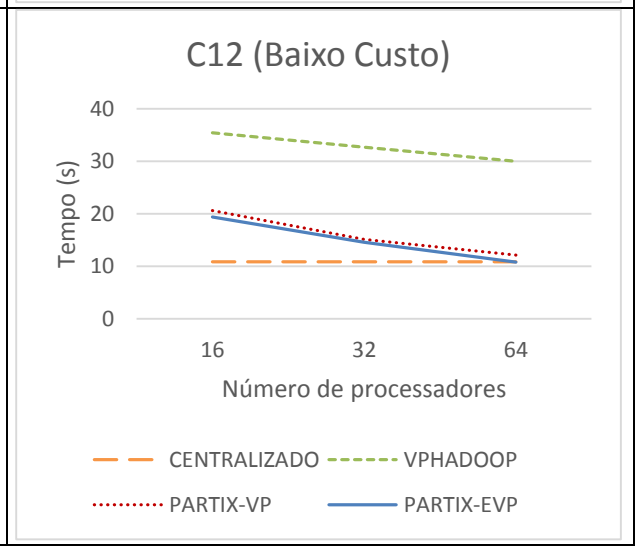

C7 (Baixo Custo)
Tempo (s)
25
20
15
10
5
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP


C8 (Baixo Custo)
Tempo (s)
50
40
30
20
10
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

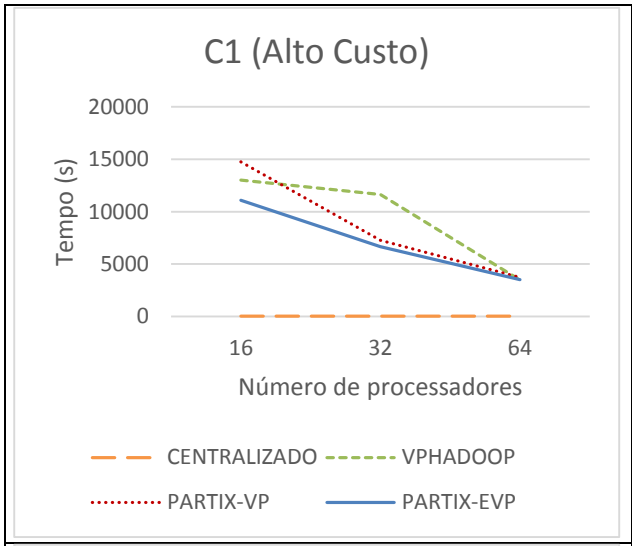

C9 (Baixo Custo)
Tempo (s)
1000
800
600
400
200
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

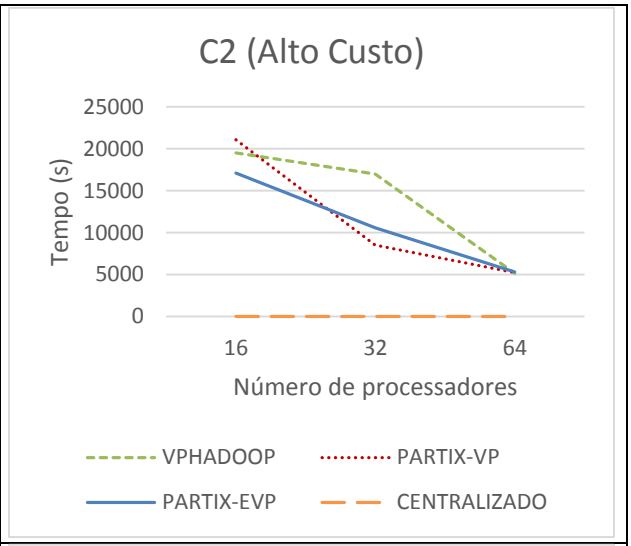

C10 (Alto Custo)
Tempo (s)
300
250
200
150
100
50
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

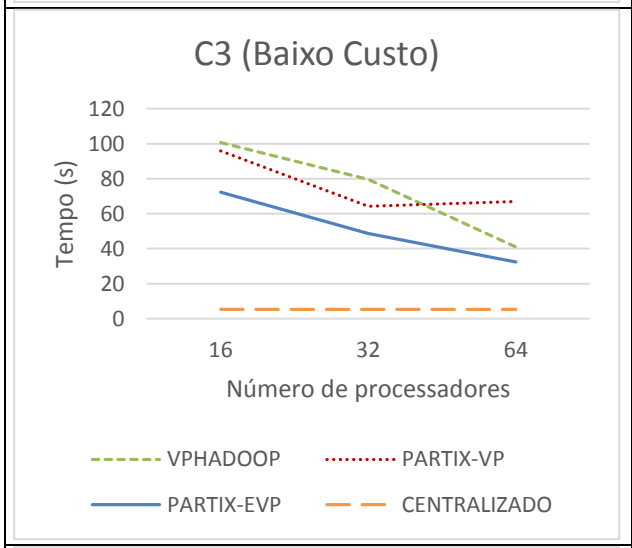

C11 (Alto Custo)
Tempo (s)
600
500
400
300
200
100
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

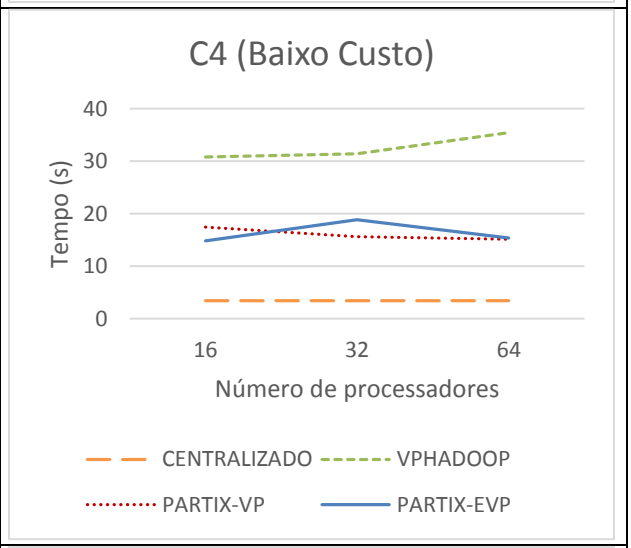

C12 (Baixo Custo)
Tempo (s)
40
30
20
10
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

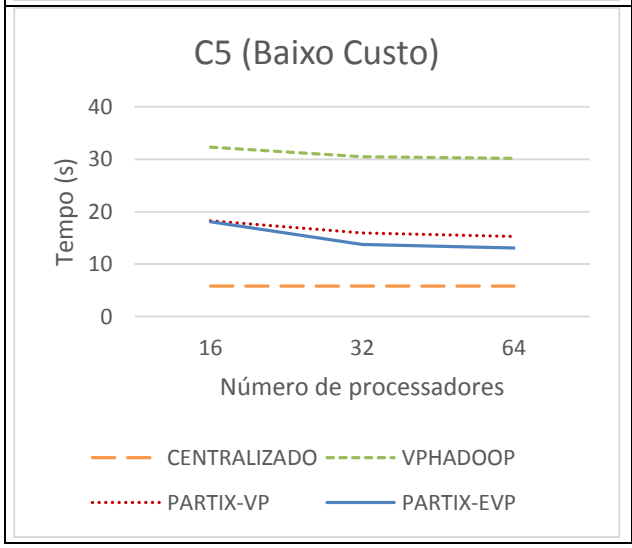
ANEXO D – GRÁFICOS DAS EXECUÇÕES PARA A BASE DE DADOS MD DE 1 GB

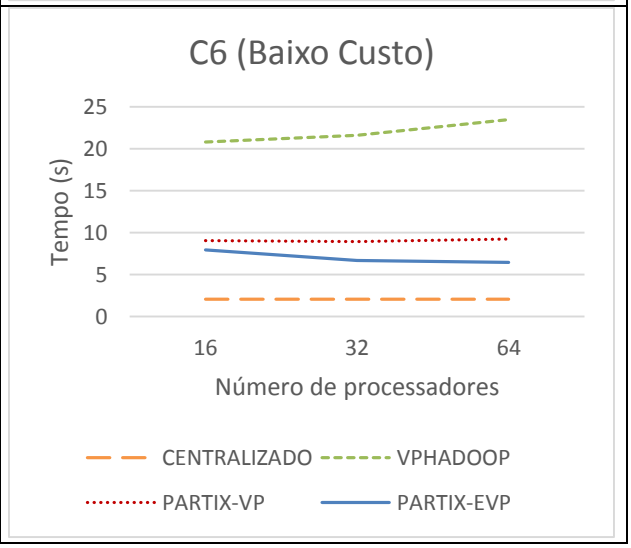


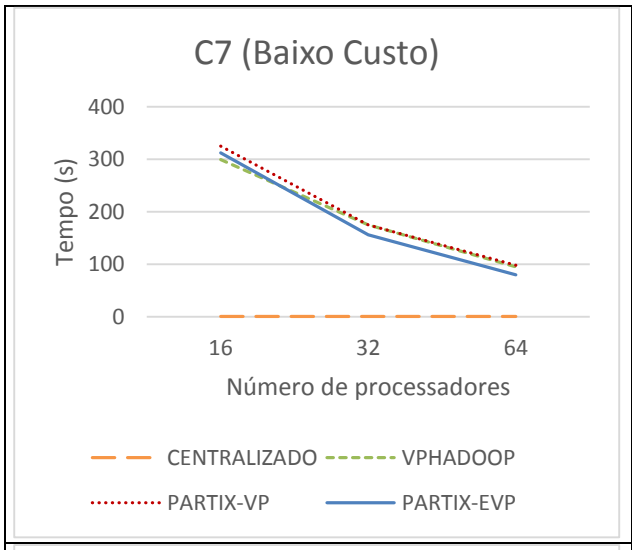


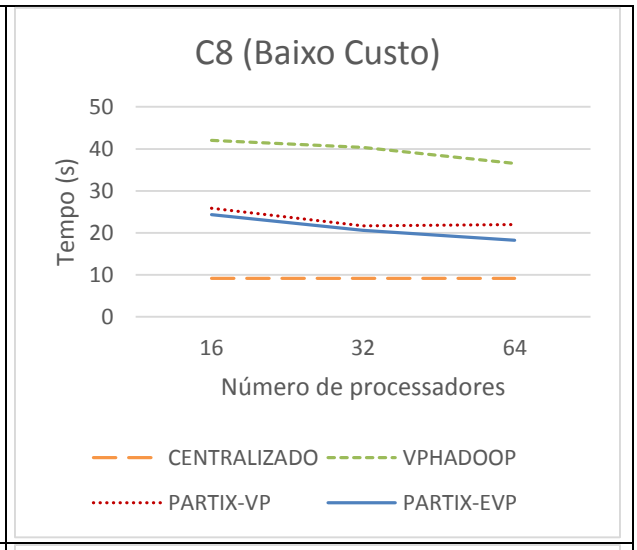


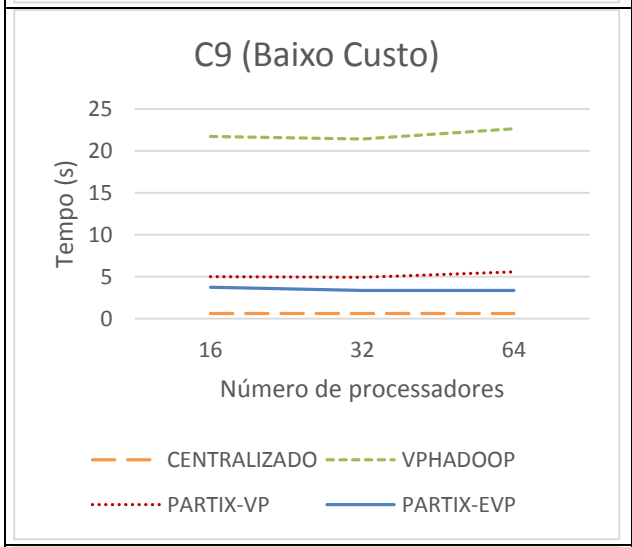


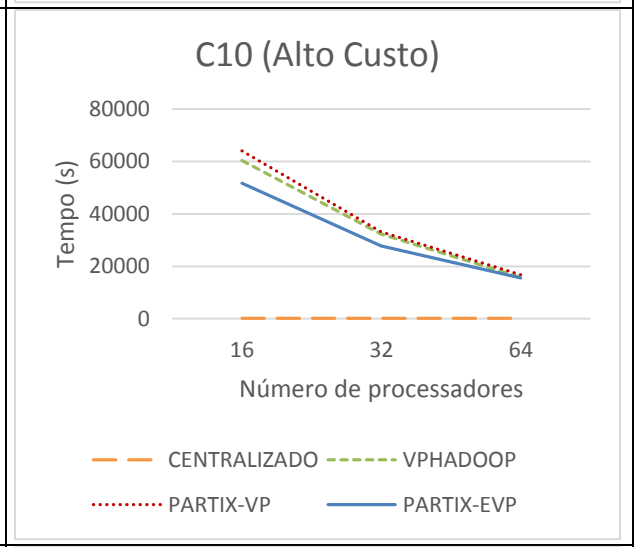


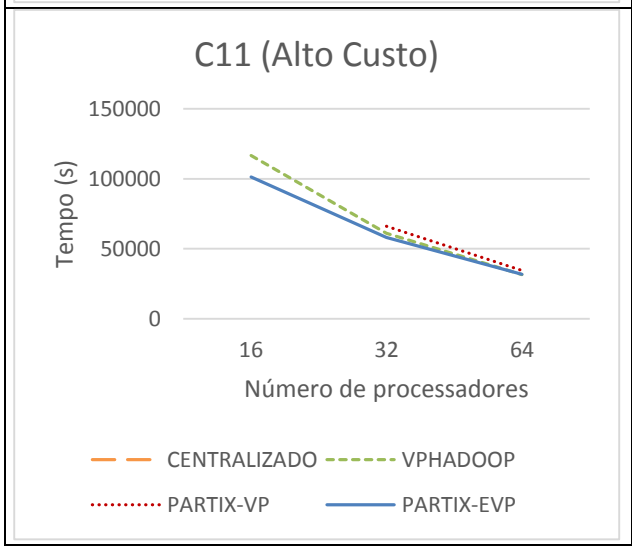


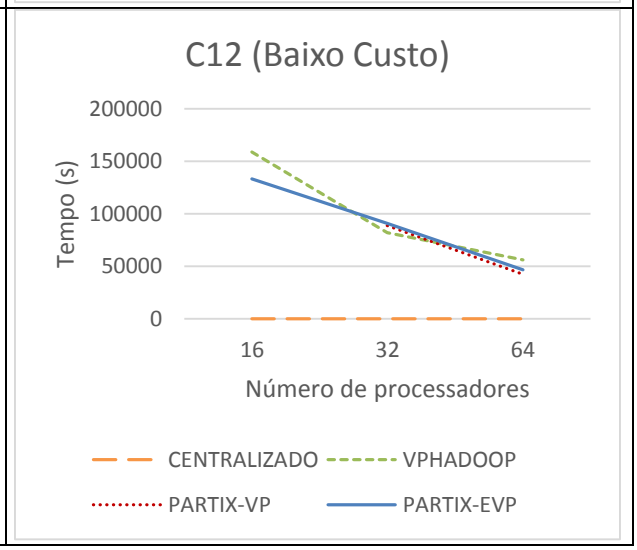

C7 (Baixo Custo)
Tempo (s)
400
300
200
100
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP


C8 (Baixo Custo)
Tempo (s)
50
40
30
20
10
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

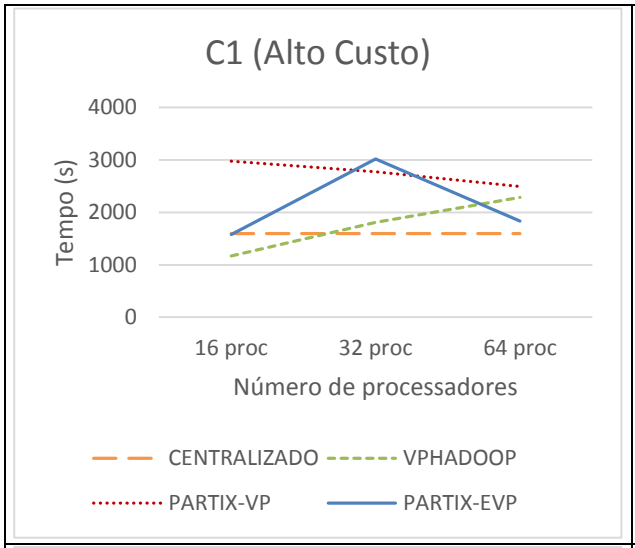

C9 (Baixo Custo)
Tempo (s)
25
20
15
10
5
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

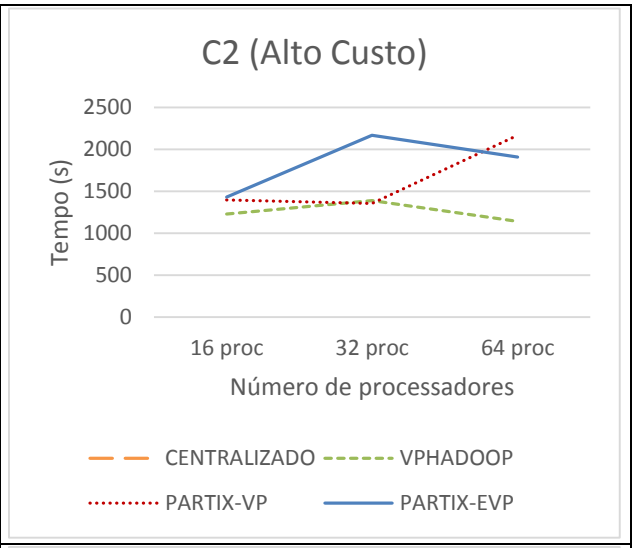

C10 (Alto Custo)
Tempo (s)
80000
60000
40000
20000
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

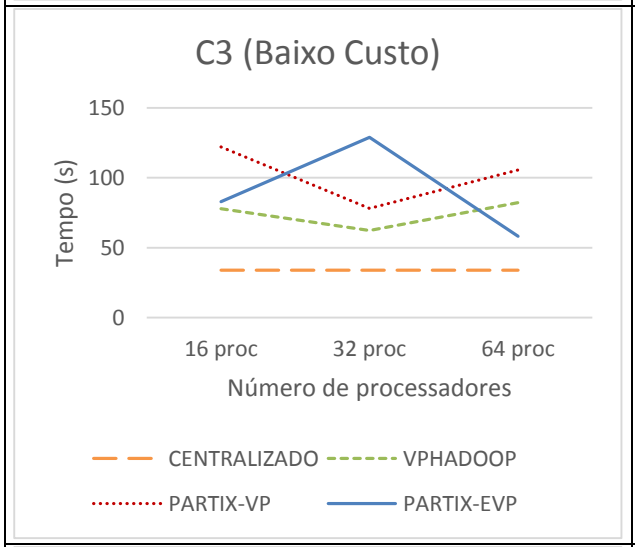

C11 (Alto Custo)
Tempo (s)
150000
100000
50000
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

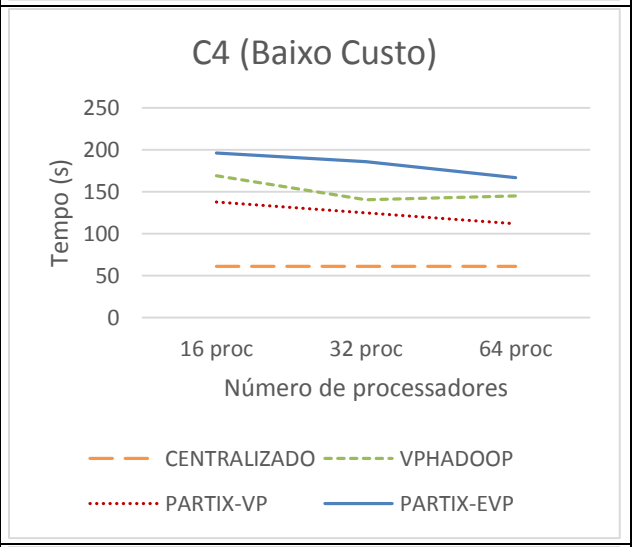

C12 (Baixo Custo)
Tempo (s)
200000
150000
100000
50000
0
16
32
64
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

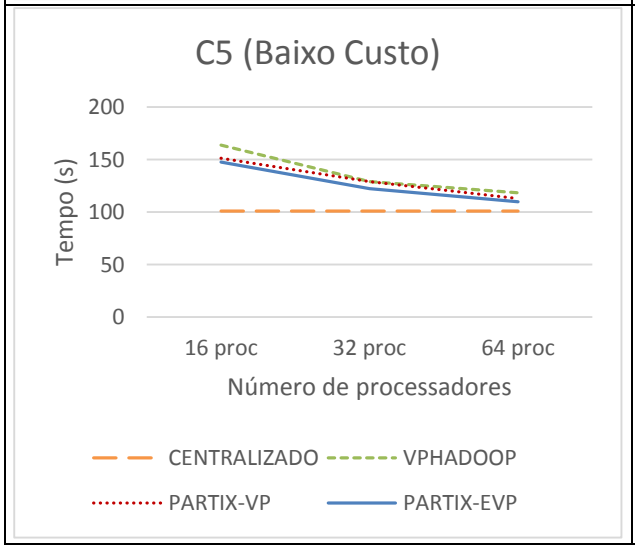
ANEXO E – GRÁFICOS DAS EXECUÇÕES PARA A BASE DE DADOS SD DE 10 GB

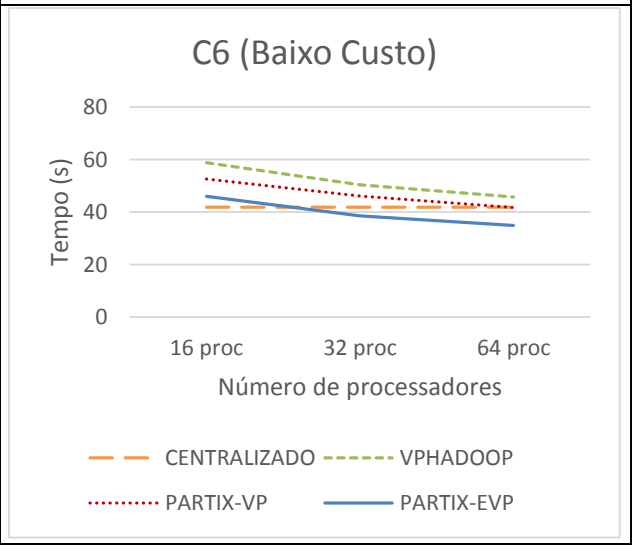


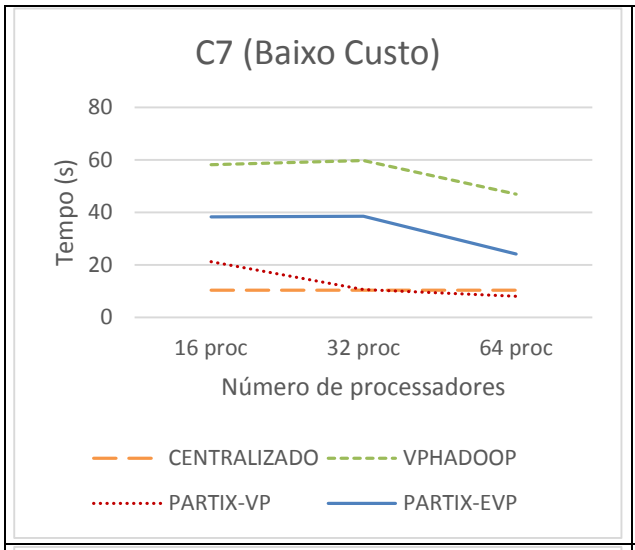


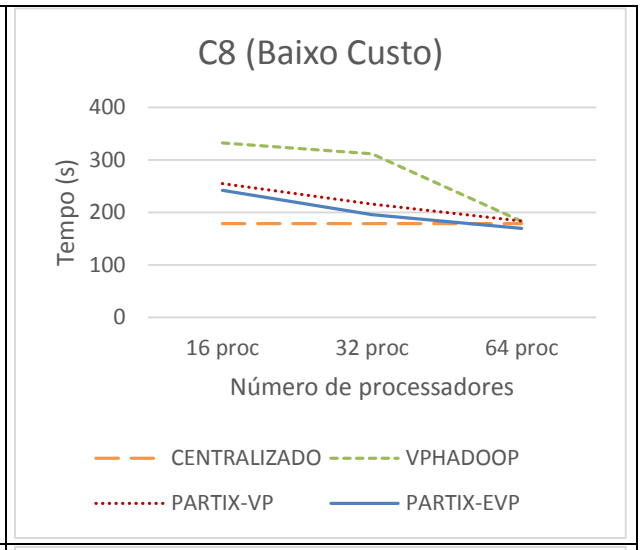


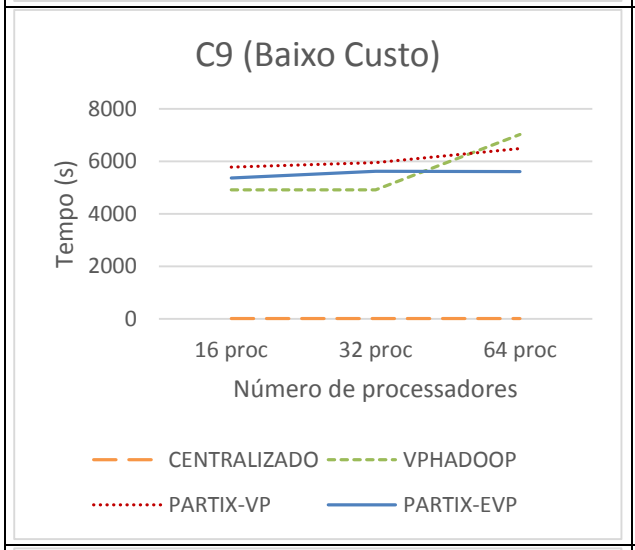


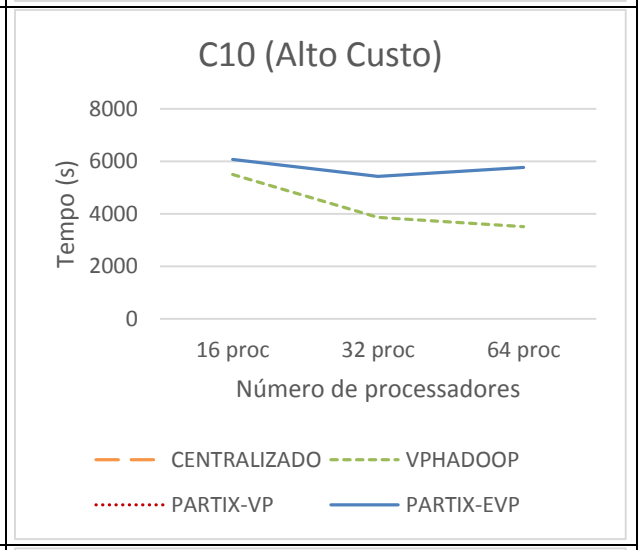


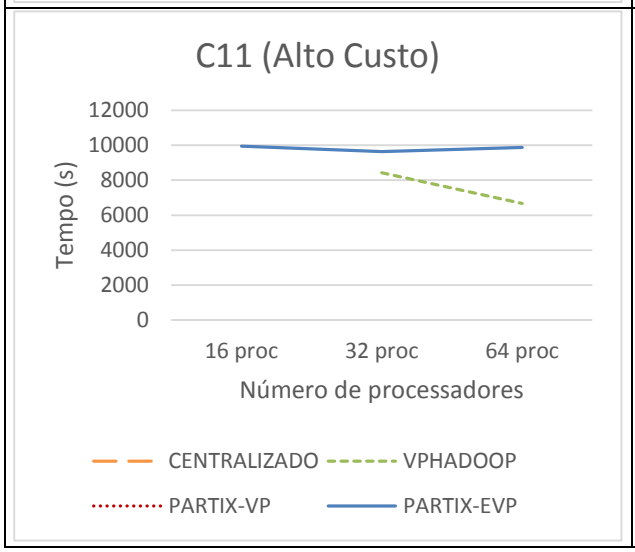


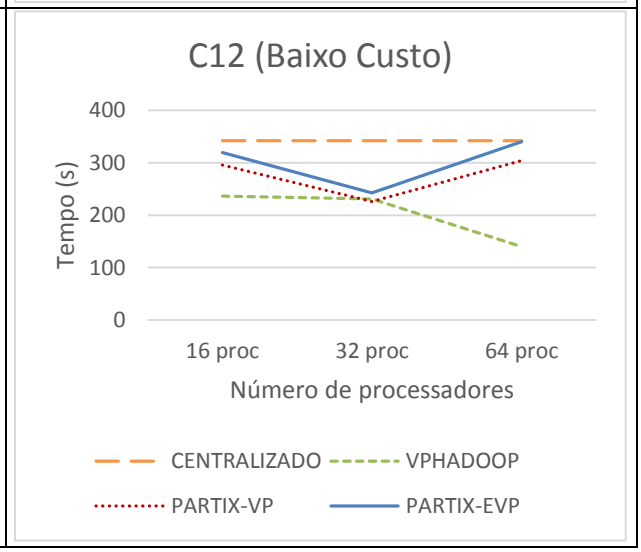

C7 (Baixo Custo)
Tempo (s)
80
60
40
20
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP


C8 (Baixo Custo)
Tempo (s)
400
300
200
100
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

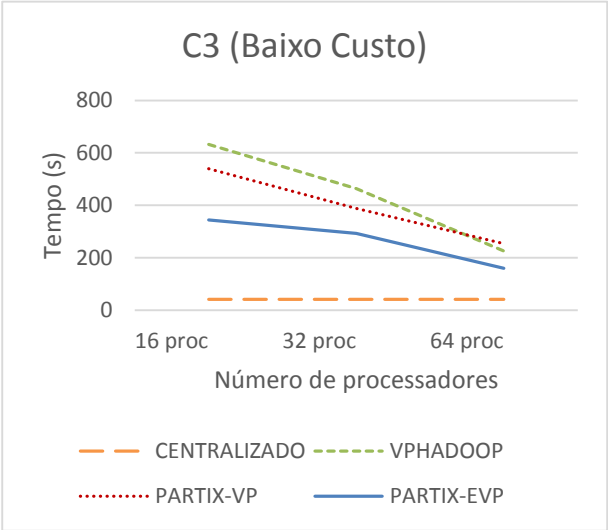
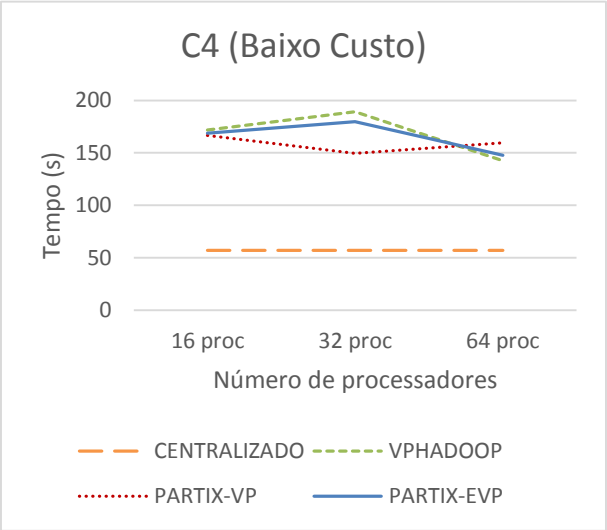
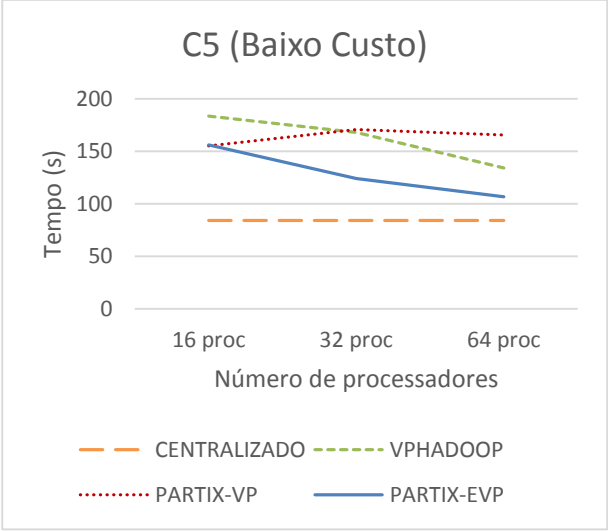
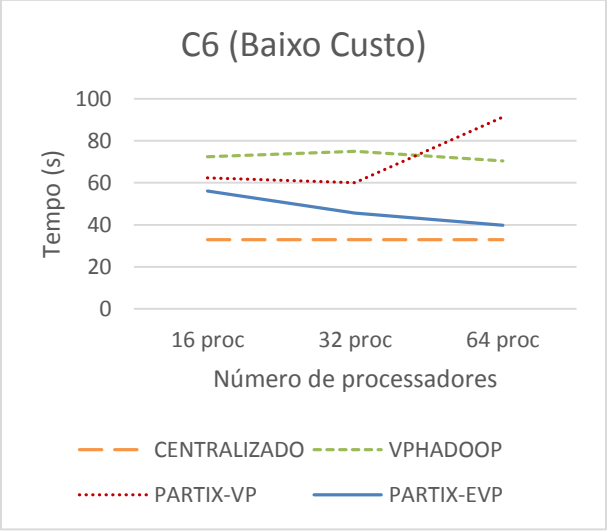

C9 (Baixo Custo)
Tempo (s)
8000
6000
4000
2000
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

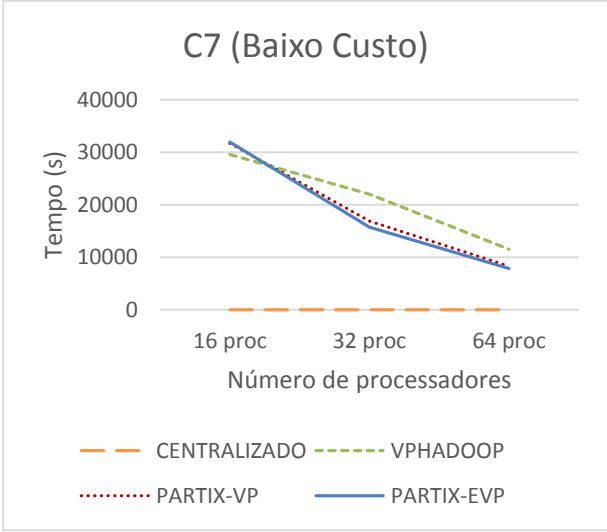

C10 (Alto Custo)
Tempo (s)
8000
6000
4000
2000
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

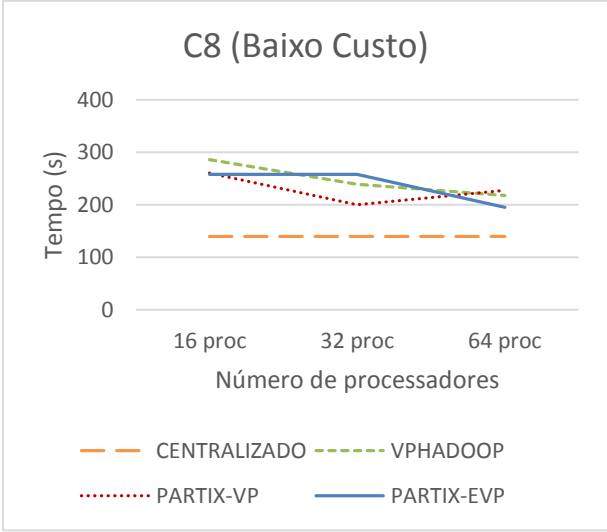

C11 (Alto Custo)
Tempo (s)
12000
10000
8000
6000
4000
2000
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

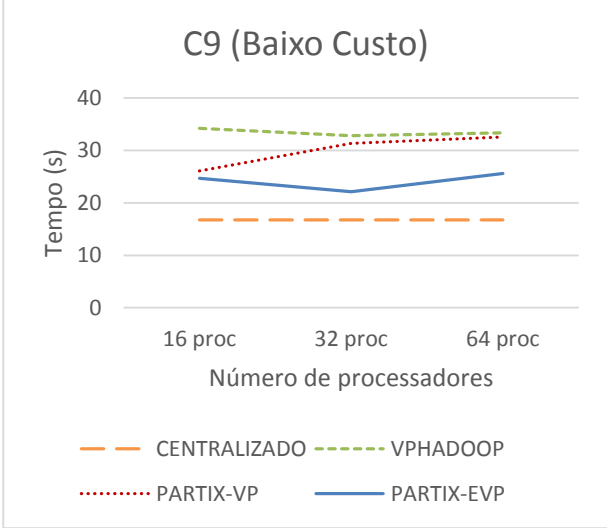

C12 (Baixo Custo)
Tempo (s)
400
300
200
100
0
16 proc
32 proc
64 proc
Número de processadores
CENTRALIZADO
VPHADOOP
PARTIX-VP
PARTIX-EVP

ANEXO F – GRÁFICOS DAS EXECUÇÕES PARA A BASE DE DADOS MD DE 10 GB

A consulta C1 não retornou resultados na execução com as abordagens distribuídas (*timeout*)	A consulta C2 não retornou resultados na execução com as abordagens distribuídas (*timeout*)
	
	







A consulta C10 não retornou resultados na execução com as abordagens distribuídas (*timeout*)

A consulta C11 não retornou resultados na execução com as abordagens distribuídas (*timeout*)

A consulta C12 não retornou resultados na execução com as abordagens distribuídas (*timeout*)