ESTUDO COMPARATIVO ENTRE SISTEMAS DE BANCOS DE DADOS DE GRAFOS E RELACIONAIS PARA A GERÊNCIA DE DADOS DE PROVENIÊNCIA EM WORKFLOWS CIENTÍFICOS

Ricardo Rocha Soares

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador(es): Alexandre de Assis Bento Lima

Rio de Janeiro

Setembro de 2013

ESTUDO COMPARATIVO ENTRE SISTEMAS DE BANCOS DE DADOS DE GRAFOS E RELACIONAIS PARA A GERÊNCIA DE DADOS DE PROVENIÊNCIA EM WORKFLOWS CIENTÍFICOS

Ricardo Rocha Soares

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____
Prof. Alexandre de Assis Bento Lima, D.Sc.

_____
Prof[a.] Marta Lima de Queirós Mattoso, D.Sc.

_____
Prof. Leonardo Guerreiro Azevedo, D.Sc.

_____
Prof. Daniel Cardoso Moraes de Oliveira, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2013

Soares, Ricardo Rocha

Estudo Comparativo entre Sistemas de Bancos de Dados de Grafos e Relacionais para a Gerência de Dados de Proveniência em Workflows Científicos/ Ricardo Rocha Soares. – Rio de Janeiro: UFRJ/COPPE, 2013.

XII, 130 p.: il.; 29,7 cm.

Orientador: Alexandre de Assis Bento Lima

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 56-62.

1. Proveniência. 2. Bancos de Dados de Grafos. 3. Workflows Científicos. I. Lima, Alexandre de Assis Bento. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Ao meu pai Antonio (In memoriam),

à minha mãe Maria Isabel,

ao meu irmão Marcos,

à Danielle e aos meus amigos,

por tudo o que representam em minha vida.

Agradecimentos

A Deus, por me dar forças e não desistir dessa longa jornada.

À minha família, por todo apoio e incentivo.

À Danielle, pela paciência e compreensão.

Ao Alexandre de Assis, por sua paciência e orientação na minha vida acadêmica.

Aos professores Marta Mattoso, Leonardo Azevedo e Daniel Oliveira, por terem aceitado participar da banca avaliadora.

Ao Carlos Santanna, Vanius, Katia, Carlos André e Carlos Gomes pela amizade.

À CAPES, pela concessão da bolsa de mestrado.

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

ESTUDO COMPARATIVO ENTRE SISTEMAS DE BANCOS DE DADOS DE GRAFOS E RELACIONAIS PARA A GERÊNCIA DE DADOS DE PROVENIÊNCIA EM WORKFLOWS CIENTÍFICOS

Ricardo Rocha Soares

Setembro/2013

Orientador: Alexandre de Assis Bento Lima

Programa: Engenharia de Sistemas e Computação

Avanços científicos estão cada vez mais sendo alcançados através de conjuntos complexos de cálculos e análises de dados, que podem compreender milhares de atividades encadeadas e representadas por meio de *workflows* científicos. Um *workflow* pode ser executado inúmeras vezes, sendo que os dados de proveniência destas execuções são uma fonte de informação cada vez mais relevante, além de um componente crucial para os Sistemas Gerência de *Workflows* Científicos (SGWfC). Vários sistemas que lidam com dados de proveniência os armazenam em Bancos de Dados Relacionais (BDR). No entanto, estes dados são naturalmente modelados como grafos direcionados acíclicos, o que faz parecer que a utilização de Bancos de Dados de Grafos (BDG) para sua gerência seria mais adequada. Esta dissertação tem como objetivo investigar, considerando diferentes cenários, qual das alternativas para gerência de dados de proveniência é a melhor, avaliando principalmente o desempenho de execuções de consultas típicas de proveniência. Além de comparar o desempenho, realizou-se uma análise em termos de uso de processador e memória, bem como do espaço ocupado em disco por cada base experimental. Nenhuma solução foi superior em todos os aspectos testados, porém, em termos gerais, o BDG superou o BDR.

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

COMPARATIVE STUDY BETWEEN GRAPH AND RELATIONAL DATABASE SYSTEMS FOR PROVENANCE DATA MANAGEMENT IN SCIENTIFIC WORKFLOWS

Ricardo Rocha Soares

September/2013

Advisor: Alexandre de Assis Bento Lima

Department: Systems and Computer Engineering

Scientific advances are increasingly being achieved through sets of complex calculations and data analysis, which may include thousands of linked activities represented by scientific workflows. A workflow can be executed numerous times, and provenance data coming from these runs are an increasingly important source of information for the analysis and reproduction of such executions. Many systems that deal with provenance data store them in Relational Databases (RDB). However, these data are naturally modeled as directed acyclic graphs, which makes it seem that the use of Graph Databases (GDB) to its management would be more appropriate. This dissertation aims to investigate, considering different scenarios, which of the alternatives for provenance data management is the best by evaluating the performance of typical provenance query executions. In addition to performance evaluation, an analysis was carried out in terms of processor and memory consumption, and disk space needed for storing each experimental database. No solution was superior in all aspects but, in general, GDB performed better than RDB.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1 – Introdução

Avanços científicos significativos estão cada vez mais sendo alcançados através de conjuntos complexos de cálculos e análise de dados (GIL *et al*., 2007) realizados em ambientes computacionais. Estes cálculos e análises podem compreender milhares de atividades (processos), onde cada atividade pode integrar diversos modelos e fontes de dados desenvolvidos por diferentes grupos. As aplicações e dados podem estar distribuídos no ambiente de execução. Cada atividade pode produzir uma coleção de dados com determinada sintaxe e semântica, e estes dados podem ser utilizados como entrada para a próxima atividade a ser executada no fluxo (TAYLOR *et al*., 2007). Este encadeamento de atividades é normalmente representado por meio de *workflows* científicos. Em geral, um w*orkflow* pode ser pensado como um grafo, onde os vértices representam os módulos de análise e as arestas capturam o fluxo de dados entre os módulos (DAVIDSON *et al*., 2007). Já o termo *workflow* científico é utilizado para descrever *workflows* em qualquer área da ciência, e geralmente são utilizados em experimentos científicos de larga escala (COSTA *et al*., 2013).

Proveniência, no contexto de bibliotecas digitais, refere-se à documentação de processos executados durante o ciclo de vida de um objeto digital (GROUP, 2005). Existem duas formas de proveniência: prospectiva e retrospectiva. Proveniência prospectiva captura a especificação de uma tarefa computacional (seja um *script* ou *workflow*) e relaciona os passos que devem ser seguidos para gerar o resultado do experimento. Proveniência retrospectiva captura as atividades executadas, bem como informações sobre o ambiente usado para a produção dos resultados, ou seja, é um registro detalhado de uma execução de tarefa computacional (FREIRE *et al*., 2008). A proveniência prospectiva é utilizada, por exemplo, para visualização da cadeia de atividades a ser executada por um *workflow* científico. A partir da proveniência retrospectiva, que é tratada nesta dissertação, pode-se obter informações sobre a produção, a interpretação e a validação do resultado científico alcançado em um experimento (MARINHO *et al*., 2009, 2010a, 2010b).

Os dados de proveniência retrospectiva podem ser referentes a um arquivo gerado por algum processo, como, por exemplo, tamanho do arquivo, diretório no qual está armazenado, última vez em que foi acessado, entre outros. Podem ser referentes a um processo, como nome, identificador, tempo de execução, entre outros. Podem

também se referir às dependências causais, por exemplo, o processo X usou o arquivo Y como entrada, ou o arquivo Y foi gerado pelo processo W. Apenas este tipo de proveniência é considerado nesta dissertação, portanto, quando se menciona proveniência, considera-se unicamente a proveniência retrospectiva.

É comum a proveniência ser representada em forma de grafos acíclicos direcionados (GAD) (MOREAU *et al*. 2010), ilustrados na Figura 1. Geralmente, consultas sobre dados de proveniência buscam não só as causas diretas de uma entidade ou ação mas também as indiretas. Para a determinação de causas indiretas, é necessária a execução de uma operação de travessia no grafo.

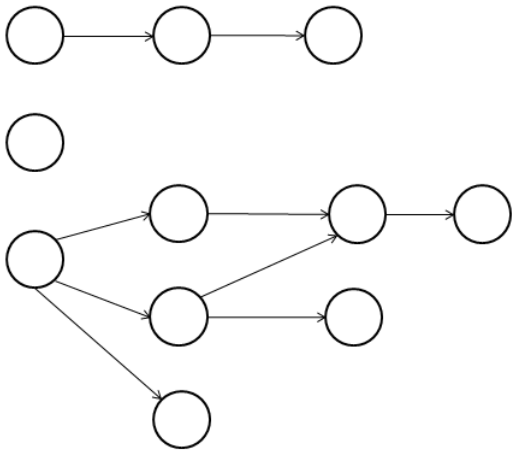

Figura 1 - Grafos acíclicos direcionados (GAD)

Vários sistemas que lidam com dados de proveniência utilizam Sistemas de Gerência de Bancos de Dados Relacionais (SGBDR), incluindo os Sistemas de Gerência de *Workflows* Científicos (SGWfC). Alguns dos SGWfC que armazenam proveniência em um modelo Relacional são Chiron (OGASAWARA, 2011) e Swift (ZHAO *et al*., 2007). Apesar de ser a opção mais tradicional, não há um esquema Relacional padrão, ou seja, cada um dos sistemas citados define seu próprio esquema. Apesar disso, existem modelos de representação de dados de proveniência, como o *Open Provenance Model* (OPM, 2007), que foi referência para a criação do OPMProv (LIM *et al*., 2011), um sistema capaz de armazenar dados de proveniência gerados por qualquer SGWfC que siga o OPM. Mais recentemente, surgiu o PROV (GIL, MILES, 2013), uma evolução do OPM e que é a proposta atual para um modelo padrão de proveniência. Embora o uso de SGBDR seja a opção mais comum para armazenamento de dados de proveniência, ele apresenta algumas limitações, dentre elas, responder

consultas que necessitem executar travessias em grafos, pois requer um número arbitrário de junções entre tabelas, e, no contexto de proveniência, este tipo de consulta é bem comum (WOODMAN *et al*., 2011).

Recentemente, o termo “NoSQL” (que significa “não apenas SQL”) foi cunhado para descrever uma classe de bancos de dados que não têm propriedades de Bancos de Dados Relacionais (BDR) tradicionais (JATANA *et al.*, 2012). Estes bancos se caracterizam por não utilizar um modelo Relacional; funcionar de forma eficiente em agregados de computadores; ter consistência de dados tardia; manipular grafos de forma eficiente; e/ou agregar dados, ou seja, tratar uma coleção de dados como uma unidade (SADALAGE, FOWLER, 2013). Cada ferramenta pode apresentar todas estas características ou apenas um subconjunto delas. Ao longo dos anos, várias soluções de bancos de dados NoSQL vêm sendo desenvolvidas, geralmente para atender a necessidades específicas, onde a solução tradicional, ou seja, o uso de um BDR, não atende aos requisitos de uma determinada aplicação. Os motivos podem ser os mais variados: crescimento exponencial do volume de dados, baixo desempenho na execução de consultas, necessidade de navegação entre sequências de objetos, baixa escalabilidade, entre outros (LAKSHMAN, MALIK, 2010, CHANG *et al*., 2008).

Amazon, Facebook e Google, empresas conhecidas em nível mundial, caracterizadas por terem de manipular grandes volume de dados e por operarem *online* 24 horas por dia, baseadas em suas necessidades, desenvolveram suas próprias soluções de banco de dados NoSQL: DynamoDB (DECANDIA *et al*., 2007, DYNAMODB, 2013), Cassandra (CASSANDRA, 2013) e BigTable (CHANG *et al*., 2008), respectivamente.

Entre os projetos NoSQL, alguns autores incluem os Bancos de Dados de Grafos (BDG) (SADALAGE, FOWLER, 2013, ROBINSON *et al*., 2013). Diferentemente dos demais, eles não agregam dados. Além disso, possuem um suporte nativo a grafos. Considerando estas características, os BDG podem ser uma melhor solução para gerência de dados de proveniência do que os BDR, uma vez que os dados de proveniência formam grafos, mais especificamente GAD. Gerenciados por um Sistema de Gerência de Bancos de Dados de Grafos (SGDBG), eles seriam armazenados e tratados em sua forma nativa, pois a maioria destas ferramentas possui suporte à execução de operações tradicionais de grafos.

VICKNAIR *et al.* (2010) comparam o desempenho de Sistemas de Gerência de Bancos de Dados Relacionais e de Grafos para armazenamento, busca e recuperação de

dados de proveniência. Para isto, os autores criaram consultas e mediram seus tempos de execução em bancos de dados gerenciados por ambos os tipos de ferramentas. As consultas que necessitavam executar travessias no grafo foram limitadas a uma quantidade máxima de passos pré-determinada e implementadas através de códigos procedimentais em Java que, nos testes envolvendo BDR, executavam várias consultas em SQL. Eles concluíram que em termos de desempenho, o BDG foi melhor que o BDR para este tipo de consultas.

Na mesma linha de trabalho de VICKNAIR *et al*. (2010), esta dissertação analisa e compara qual das alternativas (SGBDR e SGBDG) é a melhor para gerência dos dados de proveniência. Porém, são utilizadas outras abordagens, ainda não utilizadas em estudos comparativos, para implementar consultas típicas de proveniência, como por exemplo, visões recursivas e procedimentos armazenados (*stored procedures*). Estas abordagens, em teoria, são mais eficientes e permitem executar travessias completas, ou seja, não necessitam limitar a profundidade de busca conforme VICKNAIR *et al*. (2010) fizeram. Além de comparar o desempenho, realizou-se novas análises, não presentes em VICKNAIR *et al*. (2010), como o uso de processador e memória. Enquanto o trabalho citado restringe sua análise ao desempenho obtido durante a execução de consultas, esta dissertação também analisa estas métricas no contexto do processo de carga das bases de dados experimentais.

Nesta dissertação, optou-se pela utilização de ferramentas de software livre para gerência das bases de dados. No contexto Relacional foi escolhido o SGBDR PostgreSQL (POSTGRESQL, 2013), o qual suporta visões recursivas e procedimentos armazenados. E no contexto de Grafos foi utilizado o Neo4j (NEO4J, 2013).

Os experimentos utilizaram, em diferentes cenários, 14 consultas típicas de proveniência, as quais buscavam e recuperavam dados relativos a execuções reais de *workflows* científicos, totalizando 13440 execuções. Os dados utilizados no experimento estão no formato OPM (MOREAU *et al*., 2010), e, por isso, os modelos de armazenamento se baseiam neste padrão e não no PROV. Independentemente do padrão utilizado, o resultado não mudaria, uma vez que as características avaliadas são relativas a dados de proveniência em geral. O SGBDG foi mais rápido na execução das consultas, e o SGBDR exigiu menos recursos computacionais, ou seja, nenhuma solução foi superior em todos os aspectos testados. Apesar disso, em termos gerais, o SGBDG superou o SGBDR.

Nas próximas seções, é discorrido a respeito da motivação e do objetivo desta dissertação. Por fim, sua estrutura é apresentada.

1.1 Motivação

Os Sistemas de Gerência de Bancos de Dados Relacionais, durante anos, foram as principais e mais populares ferramentas para gerência de dados, incluindo dados de proveniência. Mas, em alguns casos, soluções NoSQL, como os Sistemas de Gerência de Bancos de Dados de Grafos, vêm substituindo os SGBDR. Ambas as ferramentas possuem vantagens e desvantagens. Os SGBDG oferecem suporte nativo a operações sobre grafos. Já os SGBDR, por existirem há mais tempo, estão mais amadurecidos e, consequentemente, otimizados em diversos aspectos. Portanto, considerando as vantagens e desvantagens, esta dissertação tem como motivação investigar, considerando diferentes cenários, qual das alternativas para gerência de dados de proveniência é a melhor, avaliando principalmente o desempenho de execuções de consultas típicas de proveniência.

1.2 Objetivo

O objetivo desta dissertação é analisar o uso de Sistemas de Gerência de Bancos de Dados Relacionais e de Grafos para a gerência de dados de proveniência, especialmente no que diz respeito ao processamento de consultas típicas de proveniência. Uma comparação já foi feita por VICKNAIR *et al.* (2010), porém, esta dissertação analisa outras técnicas para a implementação e execução de consultas, como a utilização de visões recursivas e procedimentos armazenados no contexto Relacional, e a linguagem declarativa Cypher do Neo4j no contexto de Grafos, além da API Java desta ferramenta (já utilizada no referido trabalho). Foram analisados e comparados os desempenhos obtidos durante a geração das bases experimentais e a execução de 14 consultas, que abordam a maioria dos padrões de consultas de proveniência, em um banco Relacional e em um Banco de Grafos.

1.3 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta os principais conceitos relacionados ao nosso tema de pesquisa. O Capítulo 3 apresenta as alternativas de armazenamento de dados de proveniência aqui analisadas. O Capítulo 4 detalha a metodologia utilizada para o trabalho de comparação, especificando os padrões de consultas de proveniência empregados, detalhes de implementação para cada estratégia de gerência de dados, os tipos de comparações feitas, os critérios utilizados para medição, programas, bibliotecas e *hardware* utilizados. O Capítulo 5 apresenta os resultados da avaliação experimental. Finalmente, o Capítulo 6 conclui a dissertação apresentando os principais resultados e os desdobramentos para trabalhos futuros.

Capítulo 2 – SGWfC, Proveniência e Bancos de Dados de Grafos

Este capítulo aborda os principais conceitos relacionados a esta dissertação: os Sistemas de Gerência de *Workflows* Científicos (SGWfC), para os quais é contextualizada a importância da proveniência; algumas formas de representação de dados de proveniência, como, por exemplo, o *Open Provenance Model* (OPM); e, por último, os Sistemas de Gerência de Banco de Dados baseados em Grafos, mais especificamente, o Neo4j.

2.1 Sistemas de Gerência de Workflows Científicos

Segundo NARDI (2009) e DEELMAN *et al.* (2009), *workflows* científicos são geralmente utilizados em experimentos com as seguintes características:

- Fluxos com atividades complexas, comum no processamento de tarefas em áreas como Bioinformática (COUTINHO *et al.*, 2010, OCAÑA *et al.*, 2013, OLIVEIRA, *et al.*, 2013), Química (NÉMETH, 2005), Agricultura (FILETO *et al.*, 2003) e prospecção de petróleo em águas profundas (CARVALHO, 2009, MARTINHO *et al.*, 2009, OGASAWARA *et al.*, 2011, OLIVEIRA *et al.*, 2009);
- Fluxos voláteis que podem sofrer alterações frequentes na avaliação de hipóteses científicas;
- Necessidade de parametrização para um grande número de atividades, por exemplo, em buscas por padrões no genoma;
- Monitoramento da execução, que deve levar em conta a dinamicidade dos fluxos;
- Execução em ambientes nos quais os recursos sejam desconhecidos a princípio; e/ou
- Necessidade de manipulação de grandes volumes de dados e demanda de alto poder computacional.

Os *workflows* científicos têm a característica de serem dinâmicos e frequentemente se tem a necessidade de adaptá-los para explorar novas técnicas, métodos ou modelos científicos (BARGA, GANNON, 2007). Visando ao apoio computacional para elaboração e execução desses *workflows* científicos, surgiram os

SGWfC, cujo objetivo principal é propiciar a orquestração de vários algoritmos e processos computacionais, fazendo-se valer de processamento paralelo e distribuído, bancos de dados, inteligência artificial, dentre outros recursos, construindo, assim, um arcabouço para experimentação através de simulação (Taylor *et al*., 2007).

Ao longo do tempo, vários SGWfC foram criados, entre eles, o VisTrails (CALLAHAN *et al*., 2006), Kepler (ALTINTAS *et al*., 2004), Taverna (HULL *et al*., 2006), Swift (ZHAO *et al*., 2007), E-BioFlow (WASSINK *et al*., 2008), SciCumulus (OLIVEIRA *et al.*, 2010, OLIVEIRA *et al.*, 2012) e o Chiron (OGASAWARA, 2011, OGASAWARA *et al.*, 2013). Cada SGWfC foi desenvolvido atendendo determinados tipos de aplicações e domínios, com uma forma individual para representação de *workflows*. Alguns oferecem uma interface visual para apoio a criação, alteração e execução dos *workflows*.

Na próxima seção, é discorrido a respeito de proveniência e sua importância para os cientistas.

2.2 Proveniência

Um *workflow* pode ser executado inúmeras vezes no contexto de um único projeto, gerando uma grande quantidade de dados intermediários e finais de interesse do usuário (BOWERS *et al*., 2007). Dados de proveniência destas execuções são uma fonte de informação cada vez mais relevante para análise e reprodução das execuções assim como um componente crucial para os sistemas de *workflows* científicos (GIL *et al*., 2007). Em outras palavras, em experimentos científicos, os dados de proveniência ajudam os cientistas a: interpretar e entender os resultados através de uma análise da sequência de passos que levaram a um resultado; obter conhecimento sobre a cadeia de raciocínio utilizado na sua produção; verificar se o experimento foi realizado de acordo com procedimentos aceitos; e identificar as entradas do experimento (FREIRE *et al*., 2008).

Capturar proveniência acompanhando o fluxo de dados requer que o sistema identifique as fontes e os consumidores de cada dado, e definia a granularidade na qual um dado será rastreado (MALIK *et al*., 2010). Dados de proveniência podem ser capturados em diversos níveis e para diferentes finalidades, entre elas obter informações sobre a produção, a interpretação e a validação do resultado científico alcançado em um

experimento (MARINHO *et al*., 2009, 2010a, 2010b), como também solucionar problemas e otimizar a eficiência (DAVIDSON *et al*., 2007). Dependendo do nível de granularidade dos dados de proveniência, podemos responder, por exemplo, a questões como:

- Qual programa foi utilizado para criar um determinado arquivo?
- Quais arquivos foram lidos pelo programa?
- Qual foi a sequência de programas executados pelo *workflow*?
- Qual foi o tempo de execução do primeiro programa?
- Qual usuário executou o *workflow*?
- O *workflow* foi executado com sucesso?
- Qual foi a quantidade de memória utilizada pelo programa?

FREIRE *et al*. (2008) cita que uma solução para gerência de proveniência é composta de três componentes principais: um mecanismo de captura (baseado em *workflows*, em processos ou no sistema operacional), um modelo de representação, e uma infraestrutura para armazenamento, acesso e consulta. Estes componentes são abordados ao longo desta dissertação.

Na próxima seção é apresentado um modelo de representação de dados de proveniência desenvolvido a partir de diversas competições de proveniência.

2.2.1 Open Provenance Model

Conforme a importância da proveniência aumenta, se faz necessário um modelo de representação de seus dados que permita a interoperabilidade entre diferentes SGWfC. Para suprir esta necessidade, várias desafios acadêmicos sobre proveniência foram iniciados, sendo o primeiro deles em 2006, durante o primeiro *International Provenance and Annotation Workshop* (IPAW, 2013). O resultado deste esforço foi a criação de um modelo de representação de dados de proveniência chamado de *Open Provenance Model* (OPM). Originalmente apresentado em um encontro realizado em agosto de 2007, em Salt Lake City, e aperfeiçoado nos anos seguintes, o OPM se encontra em sua versão final (1.1) (OPM, 2007). Ele é base para a construção de um novo modelo, o PROV (GIL, MILES, 2013), que se encontra em desenvolvimento. As bases experimentais implementadas nesta dissertação a última versão do OPM.

O OPM foi projetado para atender aos seguintes requisitos: permitir que as informações de proveniência sejam trocadas entre diversos sistemas; permitir aos desenvolvedores construir e compartilhar ferramentas que operam sobre um modelo de proveniência; definir proveniência de forma precisa; apoiar uma representação digital de proveniência para qualquer fim, produzida por sistemas de computador ou não; possibilitar a coexistência de múltiplos níveis de descrição para uma única execução de um *workflow*; e definir um conjunto de regras que identificam as inferências válidas que podem ser feitas em uma representação de proveniência (MOREAU *et al*., 2010).

Através de dependências causais entre nós, dados modelados segundo o OPM formam um GAD com três tipos de nós e cinco tipos de arestas (dependências). Conforme MOREAU *et al*. (2010) definem, os três tipos de nós são:

- Artefato (representado como um círculo): objeto físico ou uma representação digital imutável.
- Processo (representado como um quadrado): ação ou série de ações realizadas em ou causadas por artefatos, e resultando em novos artefatos.
- Agente (representado como um octógono): entidade contextual que age como um catalisador de um processo, permitindo, facilitando, controlando ou afetando sua execução.

Além destes nós, MOREAU *et al*. (2010) definem também os cinco tipos de dependências do OPM. São elas:

- *Used*: entre um processo que usa um artefato;
- *WasGeneratedBy*: entre um artefato que é gerado por um processo;
- *WasControlledBy*: entre um processo que é controlado por um agente;
- *WasTriggeredBy*: entre um processo que é desencadeado por outro processo; e
- *WasDerivedFrom*: entre um artefato que é derivado de um outro artefato.

Uma aresta representa uma dependência causal entre sua origem, denotando o efeito, e seu destino, denotando a causa. Os tipos de arestas e nós podem ser vistos na Figura 2.

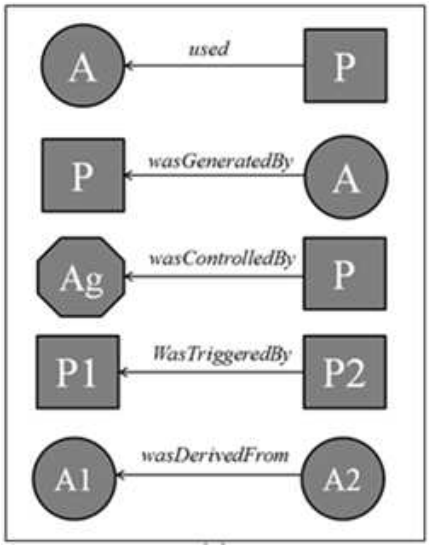

Figura 2 - Dependências causais entre nós (MOREAU *et al*., 2010)

Segundo as regras do OPM, pode-se inferir outras dependências causais a partir de dependências *wasDerivedFrom* e *wasTriggeredBy* (Figura 3). A partir destas dependências e conversões, pode-se inferir quatro tipos de dependências de múltiplos passos: *Muti-Step wasDerivedFrom*, *Muti-Step used*, *Muti-Step wasTriggeredBy* e *Muti-Step wasGeneratedBy*. Estas dependências são úteis quando queremos saber não só as causas diretas de um artefato ou processo, mas também as indiretas.

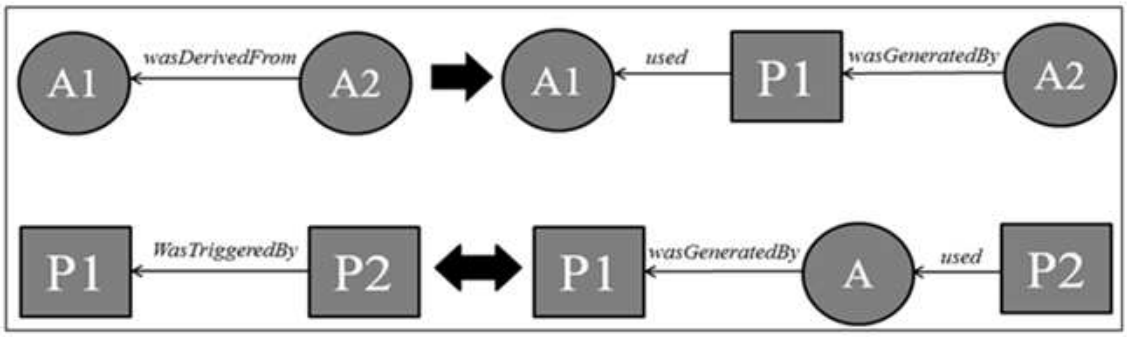

Figura 3 – Inferências realizadas a partir de dependências *wasDerivedFrom* e *wasTriggeredBy* (MOREAU *et al*., 2010)

A Figura 4 mostra um exemplo de subgrafo no qual podemos inferir que o processo P1 “usa” o artefato A1, “eliminando” os artefatos que aparecem no caminho (apenas artefatos podem ser “eliminados”) (MOREAU *et al*., 2010).

Dentre os sistemas de workflows científicos que implementam o OPM estão: Taverna, Swift, Kepler, E-BioFlow e VisTrails.

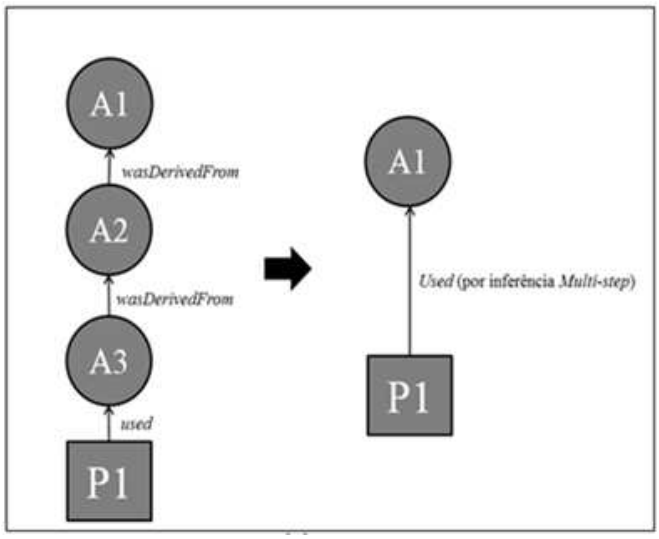

Figura 4 - Inferência de múltiplos passos ***used*** **(MOREAU *et al.*, 2010)**

Com a criação de um modelo de representação de dados de proveniência, o OPM, surgiram novas frentes de trabalhos científicos baseados no mesmo. Uma destas frentes é apresentada a seguir.

2.2.2 OPMProv

O OPMProv (LIM *et al.*, 2011) é um sistema capaz de armazenar qualquer dado de proveniência gerado por qualquer SGWfC que siga o OPM. Utilizando o OPM como modelo de dados conceitual, o OPMProv tem como características:

- Possui suporte a inferências diretas e indiretas definidas no OPM (v.1.1) utilizando a linguagem SQL com visões recursivas, sem necessidade de qualquer outro mecanismo adicional, como o uso de visões materializadas ou procedimentos armazenados.
- Foi construído no DB2 (CHAMBERLIN, 1996), mas pode ser utilizado em outros SGBDR, incluindo Oracle (ORACLE,2013), SQLServer (SQLSERVER, 2013) e PostgreSQL (POSTGRESQL, 2013).
- Possibilita que dados de proveniência representados em documentos XML que estejam em conformidade com a especificação XML Schema para o modelo OPM (v.1.1) podem ser inseridos no OPMProv usando um procedimento de

mapeamento de dados que fragmenta os documentos XML em tuplas e as armaneza em relações correspondentes do OPMProv.

Basicamente, o OPMProv traduz um modelo de representação de dados de proveniência orientado a grafo para um esquema Relacional. As bases de dados Relacionais utilizadas nesta dissertação foram implementadas de acordo com esse esquema.

Na próxima seção, é apresentado um novo modelo de dados de proveniência baseado no OPM: o PROV.

2.2.3 PROV

A evolução do OPM para representação de dados de proveniência é o PROV (GIL, MILES, 2013), um modelo de dados de proveniência que descreve as entidades, pessoas e atividades envolvidas na produção de um dado ou objeto no mundo. Em abril de 2013, o PROV passou a ser uma recomendação do *World Wide Web Consortium* (W3C) (W3C, 2013).

O PROV define uma entidade como algo físico, digital ou conceitual, podendo ser real ou imaginário. Uma atividade é definida como algo que ocorre durante um período de tempo e age sobre ou com entidades, pode transformar, consumir, modificar, utilizar, gerar, processar ou realocá-las. O PROV define um agente como algo que tem alguma responsabilidade sobre a ocorrência de uma atividade ou a existência de uma entidade. Um grafo que segue o padrão OPM pode ser mapeado para o PROV, conforme BIVAR *et al*. (2013) demonstram.

O Modelo de Dados do PROV (PROV-DM, sigla em inglês para PROV *Data Model*) (MOREAU, MISSIER, 2013) é constituído de seis componentes ilustrados na figura 5 e descritos logo abaixo. Um componente pode se basear em conceitos definidos em outro que esteja posicionado abaixo na figura.

- Componente 1: Entidades/Atividades – Consiste em entidades, atividades e conceitos ligando-os, como geração, uso, início e fim. Este é o único componente comprometido com conceitos relacionados ao tempo.

- Componente 2: Derivações – Este componente é formado por derivações, ou seja, transformações de entidades em outras, e subtipos de derivações, como, por exemplo, *WasRevisionOf*, *WasQuotedFrom* e *HasPrimarySource*.
- Componente 3: Agentes, Responsabilidade e Influência – O terceiro componente consiste em agentes e conceitos que atribuem responsabilidade aos agentes.
- Componente 4: Pacotes – O quarto componente se refere a pacotes, um mecanismo de apoio a proveniência da proveniência.
- Componente 5: Alternativo – Consiste em relações que ligam entidades que se referem a um mesmo objeto.
- Componente 6: Coleções – Por último, o sexto componente é sobre coleções, uma entidade que possui membros, os quais também são entidades.

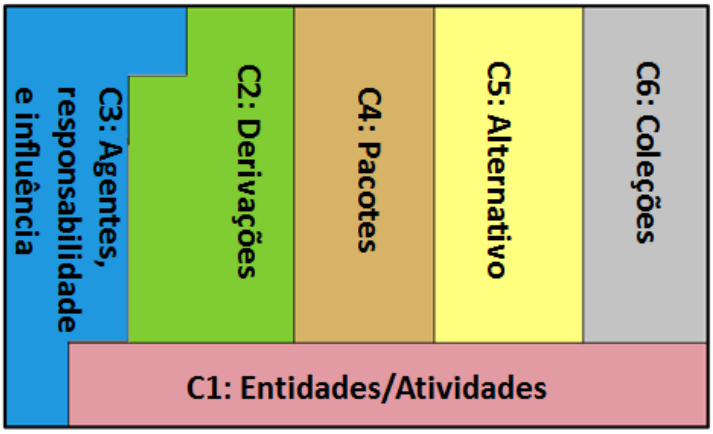

Figura 5 - Componentes do PROV-DM (Moreau, Missier, 2013)

Na próxima seção, é abordado como os principais SGWfC se integram aos modelos de representação de dados de proveniência.

2.2.4 OPM, PROV e os Principais SGWfC

Por ser muito recente e ter se tornado uma recomendação do W3C há pouco tempo, o PROV ainda é pouco utilizado. A maioria dos principais SGWfC adotam a modelagem OPM em suas implementações, em alguns, de maneira mais integrada. Apesar disso, alguns trabalhos começam a surgir na linha do PROV, como o de COSTA *et al*. (2013), que utiliza este modelo para capturar e consultar proveniência de *workflows*.

Os SGWfC VisTrails, Swift, Taverna e Kepler possuem a opção de exportar dados de proveniência para um arquivo no formato XML em conformidade com *The Open Provenance Model XML Schema* (OPMX). O E-BioFlow, um pouco mais integrado que os demais, usa o OPM para capturar a proveniência gerada pela execução do workflow e inclui uma visualização dos dados baseada no OPM.

A seguir, são discutidos alguns dos principais sistemas de proveniência.

2.2.5 Sistemas de Proveniência

Existem diversas ferramentas que capturam proveniência em ambientes centralizados ou distribuídos. Algumas capturam no nível de Sistema Operacional, outros podem ser integrados aos SGWfC, e geralmente armazenam os dados em um BDR. Estas ferramentas são os Sistemas de Proveniência. Vários destes sistemas foram aplicados a dados científicos reais (BOSE, FREW, 2005, SIMMHAN *et al*., 2005). CHAPMAN e JAGADISH (2007) citam algumas das principais questões em se construir sistemas de proveniência, como, por exemplo, escolher a granularidade dos dados de proveniência e o tamanho do armazenamento (visto que os dados de proveniência podem crescer imensamente), encontrar erros e consertá-los, e ser viável executar consultas aos dados de proveniência. A seguir, são citados alguns exemplos de sistemas de proveniência tanto no âmbito centralizado quanto no distribuído.

Lineage File System (LFS) é um sistema de arquivos que rastreia automaticamente a proveniência em nível de sistema de arquivos, tendo executáveis, linhas de comando e arquivos de entrada como as únicas fontes de proveniência, ignorando o ambiente de *hardware* e *software* em que tais processos foram executados. Além de rastrear, o LFS também armazena periodicamente os dados de proveniência em um banco Relacional. Uma de suas desvantagens é que não elimina os dados redundantes (LFS, 2013).

O *Harvard's Provenance-Aware Storage System* (PASS) audita um conjunto de eventos monitorado pelo LFS, incorporando um registro de ambiente de *software* e *hardware* onde o processo foi executado (MUNISWAMY-REDDY *et al*., 2006). Ele fornece uma maior integração entre os dados e metadados, armazenando seus registros no Berkeley DB (BERKELEY DB, 2013). Tanto o LFS quanto o PASS foram projetados para uso em um único nó, porém, podem ser estendidos passando os registros

de proveniência (ou consultas sobre eles) e metadados dos clientes para o servidor (GEHANI *et al*., 2009). Esta arquitetura também é empregada pelo projeto *Provenance-Aware Service Oriented Architecture* (PASOA).

O *Earth System Science Server* (ES3) extrai informações de proveniência automaticamente de aplicações arbitrárias, monitorando suas interações com o ambiente de execução e armazenando as informações em um banco de dados (FREW *et al*., 2008). Enquanto o ES3 registra a proveniência em uma granularidade muito maior do que o PASS, ele segue o mesmo modelo centralizado de registro de metadados.

Outro sistema de proveniência, desta vez destinado a ambientes distribuídos, é o *Distributed Provenance Aware Storage System* (DPASS). Este é um sistema de distribuído que armazena a proveniência de quaisquer arquivos contidos em um Sistema de Arquivos, permitindo ao usuário acessar a proveniência remotamente (BADOIU *et al*., 2006). Esta descentralização torna mais complexas algumas questões como a recuperação de dados de proveniência e a execução de consultas eficientes sobre os mesmos.

Outro sistema distribuído é o SPADE (GEHANI, TARIQ, 2012). Este sistema possui um modelo de armazenamento de dados de proveniência totalmente descentralizado. Para cada nó, a proveniência coletada é armazenada em um banco de dados Relacional local. SPADE armazena dados de proveniência em granularidade "fina" em ambientes distribuídos. Algumas perguntas que o SPADE é capaz de responder são:

- Qual programa foi usado para criar determinado arquivo?
- Quais arquivos foram lidos por um determinado programa?
- Quando um determinado programa foi executado, em quais arquivos ele escreveu?
- Poderia algum dado fluir de um determinado arquivo para outro?

O SPADE se utiliza de dois artifícios para diminuir a latência de rede para unificar consultas de proveniência: primeiro, ele otimiza a execução da consulta através do uso de estruturas de resumo, então, ele emprega um pré-*cache* de registros de proveniência de nós remotos quando os arquivos são transferidos (GEHANI, TARIQ, 2012).

O ProvManager (MARINHO *et al*., 2013), diferentemente dos demais sistemas citados, captura proveniência prospectiva e retrospectiva a partir de *workflows*

científicos orquestrados pelos SGWfC. Este sistema gerencia proveniência em ambientes distribuídos e heterogêneos, onde mais de um SGWfC é utilizado para executar um único experimento científico. Além disso, seu modelo segue o OPM. Ele possui características que o fazem ser compatível com os SGWfC VisTrails, Kepler e Taverna.

2.3 Bancos de Dados de Grafo

Um grafo pode ser definido como uma coleção de vértices e arestas, representando entidades na forma de vértices, e a maneira como elas se relacionam no mundo na forma de arestas (ROBINSON *et al.*, 2013). Os BDG são considerados por muitos autores como soluções NoSQL. Apesar de não agregarem dados, uma característica comum a estas soluções, eles possuem outras semelhanças como, por exemplo, não utilizam o modelo Relacional, funcionam de forma eficiente em *clusters* e geralmente possuem código aberto (SADALAGE, FOWLER, 2013).

ROBINSON *et al*. (2013) definem os Sistemas de Gerência de Bancos de Dados de Grafos (SGBDG) como "sistemas *online* de gerência de bancos de dados com métodos que expõem um modelo de dados de grafo através de operações de criação, leitura, atualização e remoção".

Nesta dissertação, foi utilizado o Neo4j (NEO4J, 2013), um Sistema de Gerência de Bancos de Dados de Grafos. Este SGBDG foi o escolhido por ser de código aberto; possuir uma estratégia nativa para armazenamento físico de dados; suportar mais de uma forma de acesso; e ser o mais popular dentre os Bancos de Dados de Grafos.

O Neo4j utiliza o modelo de grafos de propriedade. Mais especificamente, o grafo de propriedades é um multigrafo atribuído direcionado e rotulado. Rotulado porque cada aresta tem um rótulo que é usado para especificar o seu tipo. É direcionado porque permite arestas com uma orientação fixa, a partir da cauda ou do nó origem para a cabeça ou nó destino. É atribuído porque permite uma lista variável de atributos para cada nó e aresta, onde um atributo é um valor associado a um nome. E também é multigrafo porque permite múltiplas arestas entre dois nós (RODRIGUEZ, NEUBAUER, 2011). A figura 6 demonstra um grafo de proveniência criado no banco de dados Neo4j.

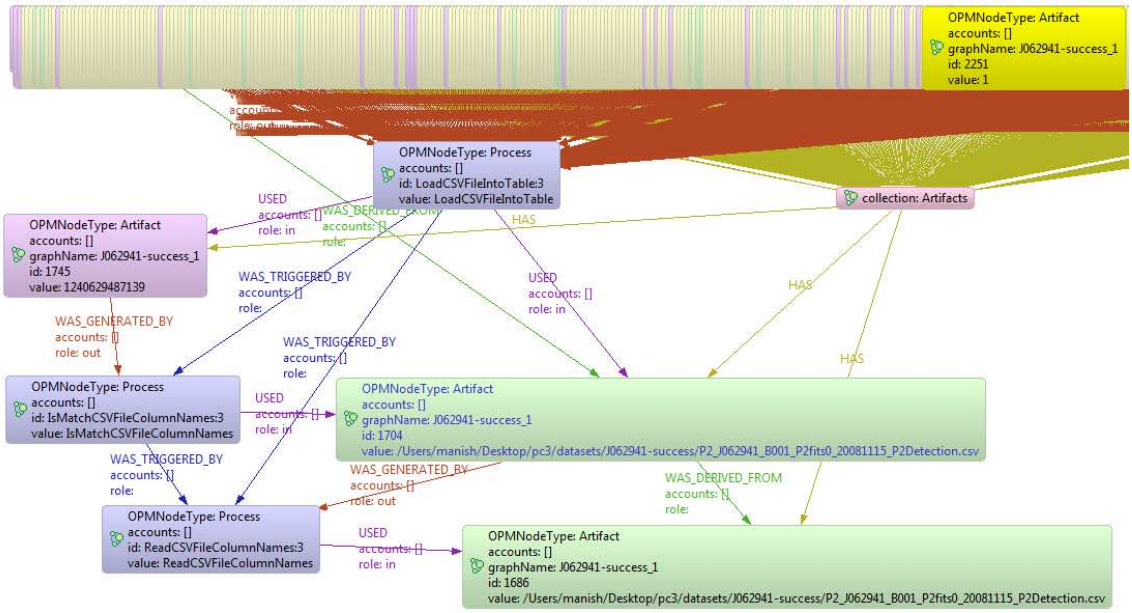

Figura 6 - Exemplo de um grafo de proveniência

O Neo4j implementa transações com propriedades ACID e pode ser usado tanto como um banco de dados embutido, quanto como um servidor independente. Hoje, o Neo4j conta com duas licenças, GPLv3 e AGPLv3/comercial (Neo4j, 2013). Além disso, o Neo4j, desenvolvido na linguagem Java, é altamente escalável, podendo suportar um grafo com um número grande de nós/arestas/propriedades em uma única máquina e sem a necessidade de que o grafo possa ser inteiramente alocado na memória. Para executar uma consulta sobre o grafo, algumas das principais opções são:

- Cypher (CYPHER, 2013): É uma linguagem declarativa de consulta a grafos desenvolvida pelos mesmos criadores do Neo4j.

- Gremlin (GREMLIN, 2013): É uma linguagem de código aberto suportada por diversos SGBDG como, por exemplo, Neo4j, DEX (DEX, 2013), TinkerGraph (TINKERGRAPH, 2013), OrientDB (ORIENTDB, 2013), InfiniteGraph (INFINITEGRAPH, 2013) e Rexter (REXTER, 2013) .

- API Java do Neo4j: É uma API para desenvolvimento de aplicações na linguagem Java.

No próximo capítulo, são avaliadas as principais formas de armazenamento de dados de proveniência, seus pontos fortes e fracos.

Capítulo 3 – Armazenamento de Dados de Proveniência

A maioria dos sistemas de gerência de proveniência utilizam Sistemas de Gerência de Bancos de Dados Relacionais, XML ou RDF para armazenamento de dados, bem como suas respectivas linguagens para a formulação de consultas (MOREAU *et al*., 2008). Neste capítulo, cada uma destas formas de armazenamento é discutida, além da alternativa de se utilizar Bancos de Dados de Grafos. Ao final do capítulo, uma comparação entre elas é feita.

3.1 Armazenamento em Bancos de Dados Relacionais

Em desenvolvimento há décadas, os Sistemas de Gerência de Bancos de Dados Relacionais (SGBDR) são uma das opções mais utilizadas pelos SGWfC para armazenar os dados de proveniência. Alguns dos SGWfC que armazenam proveniência em um Banco de Dados Relacional são Chiron e Swift. Apesar de ser a opção mais tradicional, não há um esquema Relacional padrão, ou seja, cada um dos sistemas citados implementa sua própria solução.

O uso de SGBDR para armazenar dados de proveniência foi avaliado por VICKNAIR *et al*. (2010). Naquele trabalho, os dados de proveniência são armazenados em um BDR, cujo esquema consiste de apenas duas tabelas:

- Node (nodeid, payload) – Armazena os nós do grafo, sendo a coluna “nodeid” utilizada para armazenar um identificador para cada nó e a coluna “payload” para armazenar os dados que são associados ao nó.
- Edge (source, sink) – Armazena as arestas entre os nós, sendo que a coluna “source” armazena o identificador do nó origem, e a coluna “sink”, o identificador do nó destino.

Como a maioria dos SGBDR não possui um suporte nativo a grafos, a única maneira de armazenar os dados de proveniência é tratar os nós e arestas do grafo como listas, conforme VICKNAIR *et al*. (2010) implementaram. Desta forma, para executar travessias, ou seja, caminhar no grafo, são necessárias múltiplas junções entre as listas (HOLLAND *et al*., 2008).

Para execução de consultas em BDR, utiliza-se a linguagem declarativa de Banco de Dados Relacionais, a SQL. Todavia, ela é limitada em alguns casos como, por exemplo, executar travessias em grafos. Uma das consultas que VICKNAIR *et al*. (2010) utilizaram para o experimento refere-se a executar uma travessia no grafo até uma profundidade de 128 níveis e contar quantos nós foram alcançados. Esta consulta pode ser implementada em SQL executando 128 junções entre duplicações da tabela "Edge" (o que seria muito trabalhoso de se implementar e para o SGBDR executar), ou conforme feito por VICKNAIR *et al*. (2010), através de um código procedimental escrito em Java, que implementa uma busca em largura, e executa várias consultas em SQL (sem junções, pois só a tabela que armazenava as ligações entre os nós é acessada). Outra alternativa para implementar esta consulta é utilizar um procedimento armazenado. Desta forma, evita-se o tráfego de dados entre a aplicação Java e o SGBDR e toda a navegação é feita no próprio servidor. Pode-se também implementar procedimentos armazenados recursivos, de forma que não seja necessário limitar a profundidade de busca. Esta é uma das alternativas que esta dissertação visa investigar.

Um recurso da SQL relativamente recente, e também utilizado nesta dissertação, que pode ser usado para facilitar o percurso de caminhos em um grafo sem a necessidade de restringir a profundidade, são as visões recursivas. Este recurso é utilizado no OPMProv (LIM *et al*., 2011). Nesse sistema, pode-se formular qualquer consulta de proveniência utilizando-se SQL. Até mesmo as consultas onde desejamos saber não só as causas diretas de uma entidade ou ação, como também as indiretas, ou seja, consultas que necessitam fazer travessias em grafos, podem ser respondidas utilizando visões recursivas que implementam as inferências de múltiplos passos do OPM.

A Figura 7 apresenta o diagrama Entidade-Relacionamento do OPMProv. A partir dele, pode-se observar três tipos de entidades: Processo, Artefato e Agente; e cinco tipos de relacionamento: *Used*, *WasGeneratedBy*, *WasControlledBy*, *WasTriggeredBy*, e *WasDerivedFrom*. Este diagrama é mapeado em um esquema Relacional com 24 tabelas e 5 visões, conforme listado na Figura 8. As tabelas Artifact, Process, Agent, Used, WasGeneratedBy, WasControlledBy, WasDerivedFrom e ExplicitWasTriggeredBy estão diretamente derivadas do diagrama E-R. No entanto, para manipular o conjunto de valores relacionados a Account e Annotation, algumas tabelas adicionais são introduzidas, como o "xxxHasAccount" e "xxxAnnotation", onde o "xxx" correspondente a cada tabela citada acima. Além disso, cada tabela apresenta

um identificador do grafo (o atributo OPMGraphId) para diferenciar cada execução de *workflow* (LIM *et al*., 2011) .

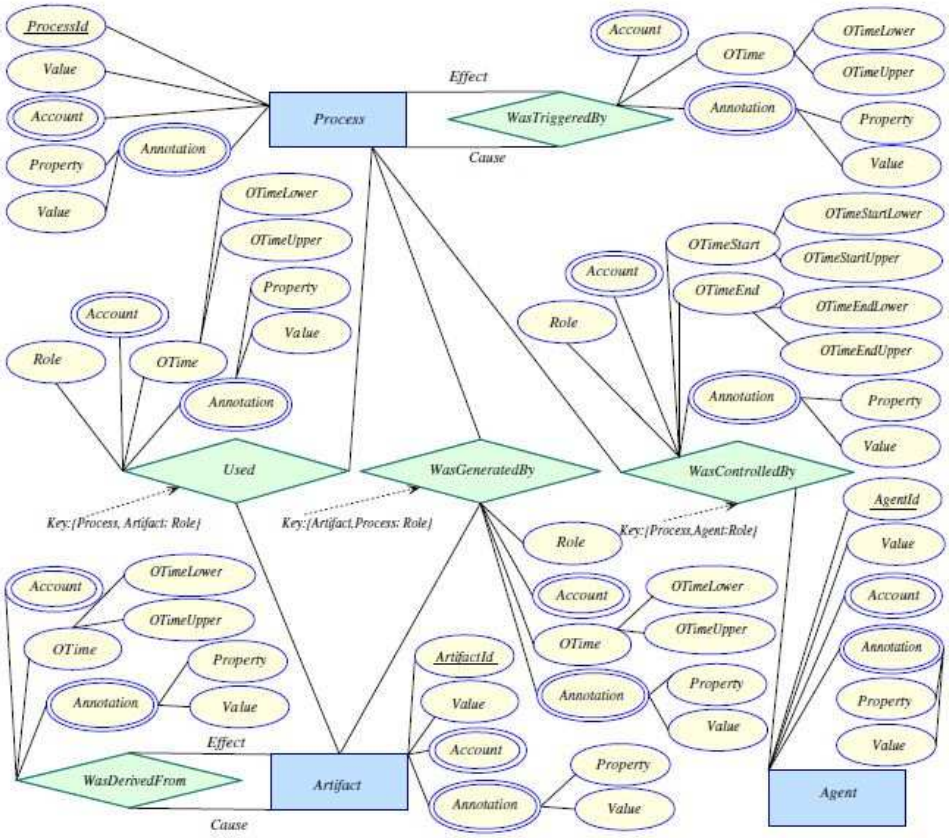

Figura 7 - Diagrama E-R do OPMProv (LIM *et al*., 2011)

```
1.Artifact (OPMgraphId, ArtifactId, Value)                                          //key = {OPMgraphId, ArtifactId}
2.Process (OPMgraphId, ProcessId, Value)                                            //key = {OPMgraphId, ProcessId}
3.Agent (OPMgraphId, AgentId, Value)                                                //key = {OPMgraphId, AgentId}
4.Used (OPMgraphId, ProcessId, Role, ArtifactId, OTimeLower, OTimeUpper)            //key = {OPMgraphId, ProcessId, ArtifactId, Role}
5.WasGeneratedBy (OPMgraphId, ArtifactId, Role, ProcessId, OTimeLower, OTimeUpper)  //key = {OPMgraphId, ArtifactId, ProcessId, Role}
6.WasControlledBy (OPMgraphId, ProcessId, Role, AgentId, OTimeStartLower,           //key = {OPMgraphId, ProcessId, ArtifactId, Role}
                OTimeStartUpper, OTimeEndUpper)
7.WasDerivedFrom (OPMgraphId, EffectArtifactId, CauseArtifactId,
                OTimeLower, OTimeUpper)                                     //key = {OPMgraphId, EffectArtifactId, CauseArtifactId}
8.ExplicitWasTriggeredBy (OPMgraphId, EffectProcessId, CauseProcessId,
                OTimeLower, OTimeUpper)                                     //key = {OPMgraphId, EffectArtifactId, CauseArtifactId}
9.ArtifactHasAccount (OPMgraphId, AgentId, Account)                         //key = {OPMgraphId, AgentId, Account}
10.ProcessHasAccount (OPMgraphId, ProcessId, Account)                       //key = {OPMgraphId, ProcessId, Account}
11.AgentHasAccount (OPMgraphId, AgentId, Account)                           //key = {OPMgraphId, AgentId, Account}
12.UsedHasAccount (OPMgraphId, ProcessId, Role, ArtifactId, Account)        //key = {OPMgraphId, ProcessId, Role, ArtifactId, Account}
13.WasGeneratedByHasAccount (OPMgraphId, ArtifactId, Role, ProcessId, Account)  //key = {OPMgraphId, ArtifactId, Role, ProcessId, Account}
14.WasControlledByHasAccount (OPMgraphId, ProcessId, Role, AgentId, Account)    //key = {OPMgraphId, ProcessId, Role, AgentId, Account}
15.WasDerivedFromHasAccount (OPMgraphId, EffectArtifactId,
                CauseArtifactId, Account)                                   //key = {OPMgraphId, EffectArtifactId, CauseArtifactId, Account}
16.ExplicitWasTriggeredByHasAccount (OPMgraphId, EffectProcesId,
                CauseProcessId, Account)                                    //key = {OPMgraphId, EffectProcesId, CauseProcessId, Account}
17.ProcessAnnotation (OPMgraphId, ProcessId, Property, Value)               //key = {OPMgraphId, ProcessId, Property, Value}
18.ArtifactAnnotation (OPMgraphId, ArtifactId, Property, Value)             //key = {OPMgraphId, ArtifactId, Property, Value}
19.AgentAnnotation (OPMgraphId, AgentId, Property, Value)                   //key = {OPMgraphId, AgentId, Property, Value}
20.UsedAnnotation (OPMgraphId, ProcessId, Role, ArtifactId, Property, Value)    //key = {OPMgraphId, ProcessId, Role, ArtifactId, Property, Value}
21.WasGeneratedByAnnotation (OPMgraphId, ArtifactId, Role, ProcessId,
                Property, Value)                                            //key = {OPMgraphId, ArtifactId, Role, ProcessId, Property, Value}
22.WasControlledByAnnotation (OPMgraphId, ProcessId, Role, AgentId,
                Property, Value)                                            //key = {OPMgraphId, ProcessId, Role, AgentId, Property, Value}
23.WasDerivedFromAnnotation (OPMgraphId, EffectArtifactId, CauseArtifactId,
                Property, Value)                                            //key = {OPMgraphId, EffectArtifactId, CauseArtifactId, Property, Value}
24.ExplicitWasTriggeredByAnnotation (OPMgraphId, EffectProcessId,
                CauseProcessId, Property, Value)                            //key = {OPMgraphId, EffectProcessId, CauseProcessId, Property, Value}
25.WasTriggeredBy (OPMgraphId, EffectProcessId, CauseProcessId, Account,
                OTimeLower, OTimeUpper)                                     //view
26.MultiStepWasDerivedFrom (OPMgraphId, EffectArtifactId, CauseArtifactId, Account)  //view
27.MultiStepWasTriggeredBy (OPMgraphId, EffectProcessId, CauseProcessId, Account)    //view
28.MultiStepUsed (OPMgraphId, ProcessId, ArtifactId, Account)               //view
29.MultiStepWasGeneratedBy (OPMgraphId, ArtifactId, ProcessId, Account)     //view
```

Figura 8 - Tabelas e visões do OPMProv (LIM *et al*., 2011)

O OPMProv não fornece um suporte nativo para grafos e armazena os dados de proveniência tratando os nós e arestas do grafo como listas. Como exemplo, a Figura 9 demonstra o código da visão recursiva MultiStepWasDerivedFrom do OPMProv (implementada no PostgreSQL), que torna possível a realização de todas as inferências de múltiplos passos da dependência *WasDerivedFrom* do OPM.

```
CREATE OR REPLACE VIEW multistepwasderivedfrom AS
         SELECT multistepwasderivedfrom1.opmgraphid,
         multistepwasderivedfrom1.effectartifactid,
         multistepwasderivedfrom1.causeartifactid,
         multistepwasderivedfrom1.account
         FROM multistepwasderivedfrom1
         UNION
         SELECT multistepwasderivedfrom2.opmgraphid,
         multistepwasderivedfrom2.effectartifactid,
         multistepwasderivedfrom2.causeartifactid,
         multistepwasderivedfrom2.account
         FROM multistepwasderivedfrom2;

CREATE OR REPLACE VIEW multistepwasderivedfrom1
        AS WITH RECURSIVE rec(opmgraphid, effectartifactid,
        causeartifactid, account) AS (
         SELECT da1.opmgraphid, da1.effectartifactid, da1.causeartifactid,
                da1.account
         FROM wasderivedfromhasaccount da1
         UNION ALL
         SELECT da2.opmgraphid, da2.effectartifactid,
                rec.causeartifactid, da2.account
         FROM wasderivedfromhasaccount da2, rec
         WHERE da2.opmgraphid = rec.opmgraphid AND
                da2.causeartifactid = rec.effectartifactid
)
SELECT rec.opmgraphid, rec.effectartifactid, rec.causeartifactid,
        rec.account
FROM rec;

CREATE OR REPLACE VIEW multistepwasderivedfrom2
        AS WITH RECURSIVE rec(opmgraphid, effectartifactid,
        causeartifactid, account) AS (
         SELECT da1.opmgraphid, da1.effectartifactid,
                da1.causeartifactid, da1.account
         FROM wasderivedfromhasaccount da1
         UNION ALL
         SELECT da2.opmgraphid, da2.effectartifactid,
                rec.causeartifactid, rec.account
         FROM wasderivedfromhasaccount da2, rec
         WHERE da2.opmgraphid = rec.opmgraphid AND
                da2.causeartifactid = rec.effectartifactid
)
SELECT rec.opmgraphid, rec.effectartifactid, rec.causeartifactid,
        rec.account
FROM rec;
```

Figura 9 - Visão recursiva MultiStepWasDerivedFrom implementada no PostgreSQL

Considerando o uso da visão recursiva MultiStepWasDerivedFrom, observa-se que esta visão retorna todos os possíveis caminhos seguindo as devidas regras definidas pelo OPM para todas as execuções armazenadas no BDR. Isto pode ser muito demorado, principalmente se o banco armazenar muitas execuções, e desnecessário, caso a consulta deva retornar dados de apenas uma determinada execução de *workflow*, o que é o cenário mais comum. Este problema se repete não só para esta visão, mas para todas as que implementam inferências de múltiplos passos. Uma solução para este problema é reproduzir estas visões na forma de procedimentos armazenados onde pode-se passar parâmetros, no caso, o identificador da execução do *workflow* e o nó inicial ou final do caminho. Deste modo, apenas os caminhos necessários para responder à consulta são percorridos. Logo, quanto maior a quantidade de execuções de *workflows* armazenadas no BDR, maior será a redução de junções desnecessárias executadas, e por consequência, menor será o tempo relativo de execução da consulta.

3.2 Armazenamento em Documentos XML

Nenhum dos sistemas pesquisados nesta dissertação armazena a proveniência no formato XML. Porém, Kepler, VisTrails e Taverna, são alguns dos SGWfC capazes de exportar dados de proveniência para um documento XML no formato do OPM. A Figura 10 apresenta um trecho de um documento XML utilizado nesta dissertação gerado pela equipe UCDGC, a qual participou do *Third Provenance Challenge* e utilizou o Kepler para exportação dos dados de proveniência para um documento XML. Nesta figura, pode-se observar uma lista de processos e artefatos com seus respectivos atributos (identificador e valor) e, logo abaixo, a listagem das dependências causais.

A dificuldade em usar esse modelo de armazenamento para representar dados de proveniência está no fato de que ele é hierárquico e não representa, naturalmente, objetos com vários “pais”. Ou seja, como a XML representa os dados como uma árvore, esta estrutura limita cada objeto a ter somente um “pai”. Caso um determinado objeto, na prática, tenha mais de um “pai”, para ser representado em formato XML, este objeto deve ser replicado para cada “pai”.

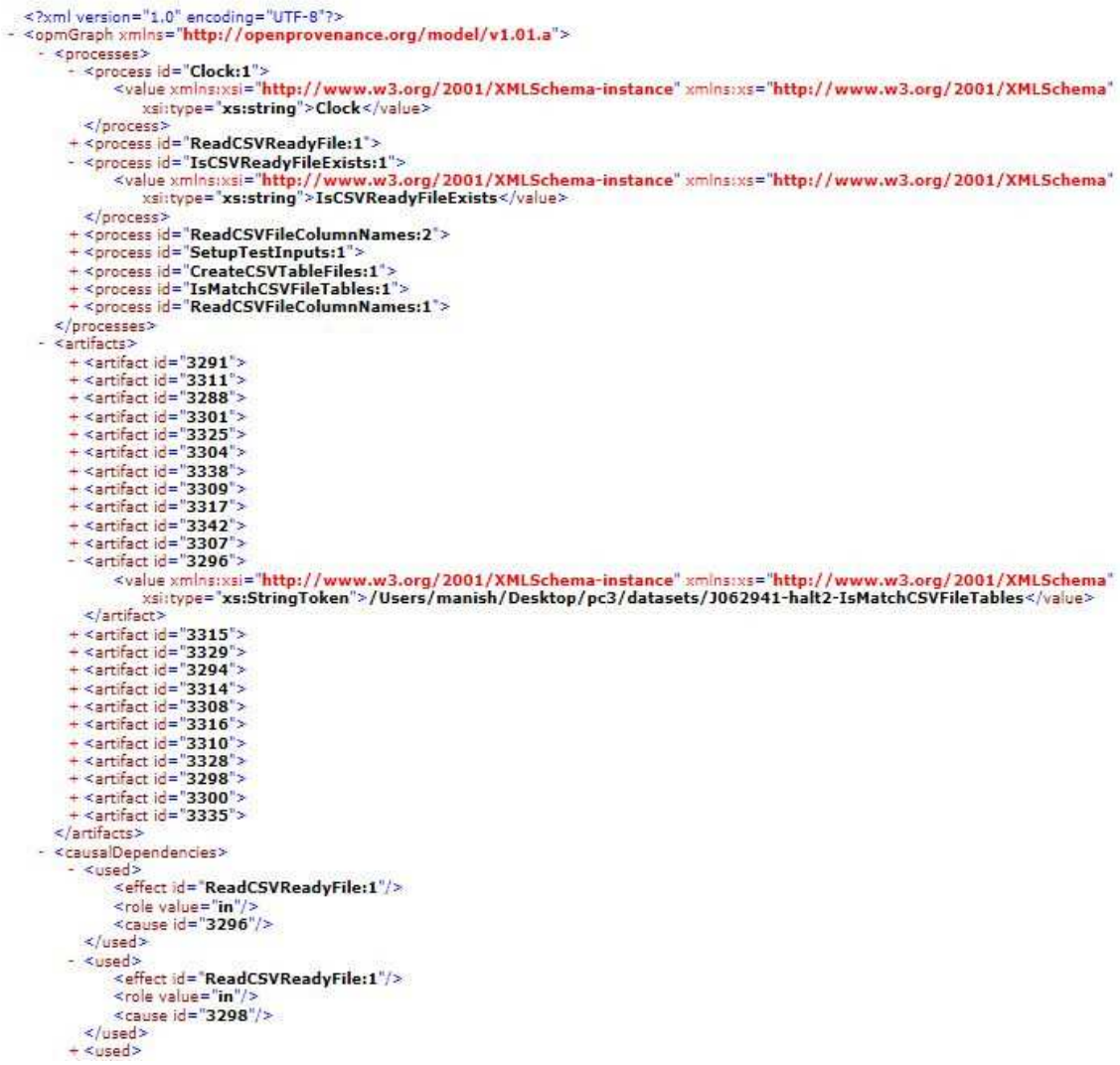

Figura 10 - Exemplo de dados de proveniência exportados para um documento XML

Enquanto a XML representa os dados como uma árvore, a proveniência é inerentemente orientada a grafos (HOLLAND *et al*., 2008). Porém, ao contrário do modelo de armazenamento Relacional, o XML suporta consultas por caminhos através das linguagens XQuery (XQUERY, 2013) e XPath (XPATH, 2013), permitindo inserir predicados no caminho para restringir objetos intermediários. Contudo, a forma com que os dados são armazenados (em árvore) dificulta percorrer caminhos típicos de consultas de proveniência, como "descobrir quais programas dependiam diretamente e indiretamente da criação do artefato A_x para começarem a sua execução". Em XQuery, por exemplo, é impossível extrair os objetos intermediários para posterior inspeção ou comparação com objetos de outros caminhos, o que seria necessário para resolver a

consulta acima. Além disso, um caminho XPath tem a restrição de sempre ter que começar na raiz do documento XML.

HOLLAND *et al*. (2008) consideram que utilizar XPath e XQuery não são apropriados para dados de proveniência. Conforme citado pelos autores, todas as equipes do *First Provenance Challenge* (FIRST CHALLENGE, 2013) que utilizaram XML como forma de armazenamento foram obrigadas a escrever códigos procedimentais para manipular as consultas do desafio que necessitavam da execução de travessias no grafo.

3.3 Armazenamento em Arquivos RDF

Recomendado desde 1999 pelo W3C, os arquivos RDF possuem um formato de dados para grafos rotulados direcionados (HOLLAND *et al*., 2008), apresentando três componentes básicos: recurso, propriedade e indicação. Assim como o formato XML, nenhum sistema pesquisado nesta dissertação utiliza o RDF como forma de armazenamento de dados de proveniência. Porém, alguns SGWfC são capazes de exportar os dados no formato RDF, entre eles, o Taverna.

A linguagem de consulta SPARQL (SPARQL, 2013a), também recomendada pelo W3C, é capaz de recuperar e manipular dados armazenados em RDF. Porém, conforme HOLLAND *et al*. (2008) citam, o SPARQL tem várias deficiências, dentre elas, carecer de suporte para sub-consultas, funções de agregação e expressões na cláusula *select*.

Em março de 2013, o W3C passou a recomendar a versão 1.1 do SPARQL (SPARQL, 2013b) que acrescenta diversas características cuja ausência na versão anterior era citada por HOLLAND *et al*. (2008) como uma deficiência. Entre as novidades estão inclusão de sub-consultas, atribuição de valores, expressões de caminho e agregação, e suporte a caminhos de comprimentos arbitrários. Este último era um dos problemas que mais impactavam no contexto de proveniência. No meio acadêmico, por ter se tornado recomendação do W3C há pouco tempo, o SPARQL 1.1 foi pouco explorado.

3.4 Armazenamento em Bancos de Dados de Grafos

Considerando a natureza dos dados de proveniência ser orientada a grafos, parece apropriado que estes sejam armazenados em um banco de dados orientado a grafos. Existem diversos SGBDG: DEX (DEX, 2013), HyperGraphDB (HYPERGRAPHDB, 2013), InfiniteGraph (INFINITEGRAPH, 2013), e, conforme citado anteriormente, o Neo4j. Além disso, possuem suporte nativo a operações de travessia em grafos.

Um dos sistemas de proveniência que utilizam o Neo4j para gerência de dados é o SPADEv2 (GEHANI A., TARIQ D., 2012). Além disso, o Neo4j foi utilizado para armazenamento de dados de proveniência em alguns trabalhos acadêmicos. ZHAO *et al*. (2011) propõem uma solução para unificar e consultar grafos de proveniência de repositórios de proveniência distribuídos utilizando o Neo4j. WOODMAN *et al*. (2011) descrevem um sistema capaz de armazenar e executar versões antigas e a atual de *workflows* e serviços, armazenando os dados de proveniência no Neo4j. Nenhum destes trabalhos demonstram o modelo de armazenamento utilizado.

Foi desenvolvido especificamente para esta dissertação, um modelo de armazenamento de dados de proveniência em Bancos de Dados de Grafos, de forma que seguisse a representação do OPM em sua forma natural de grafo. Portanto, este modelo possui os mesmos tipos de dependências e nós definidos no OPM. Além disso, foram acrescentados ao modelo nós e arestas auxiliares (Figura 11). O "Nó Referência" presente na Figura 11 é criado obrigatoriamente pelo próprio Neo4j e pode ser utilizado como ponto de entrada principal do grafo.

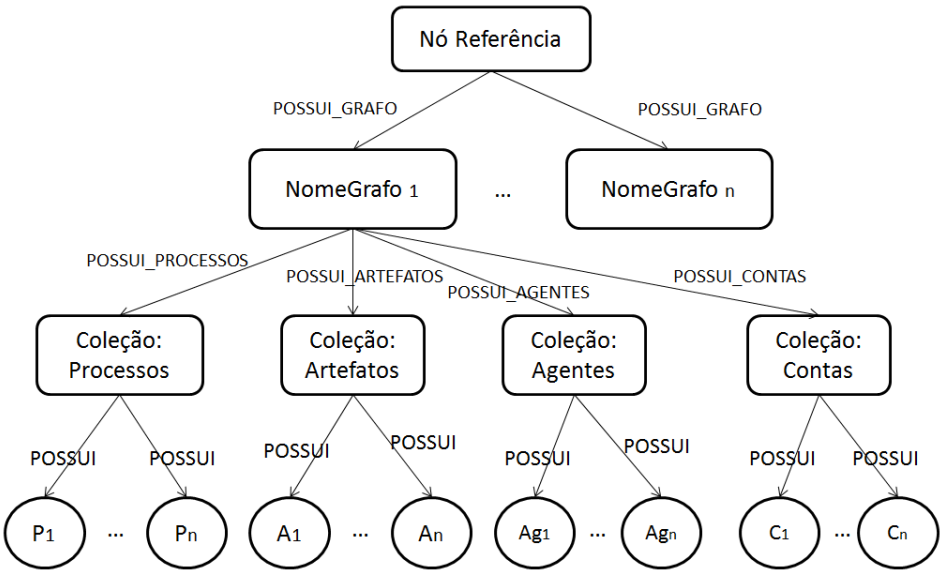

Figura 11 - Nós auxiliares

Até sua versão 1.9, um banco de dados criado com o Neo4j possui como ponto de entrada o "Nó Referência". Ao utilizá-lo para armazenamento de diferentes grafos, é necessário fazer com que cada um deles tenha pelo menos um nó com uma aresta para o "Nó Referência". Por esta razão, e para facilitar o acesso a grafos individuais, foram criados os nós do tipo "NomeGrafo$_x$". Cada um deles marca o início de um grafo de proveniência e está conectado ao "Nó Referência" por uma aresta do tipo POSSUI_GRAFO. Além disso, algumas consultas exigem acesso a apenas um tipo de nó como, por exemplo, nós do tipo Processo ou do tipo Artefato. Dada a ausência de coleções no Neo4j, foram criados quatro tipos de nós auxiliares para representar as coleções desejadas: "Coleção:Processos", "Coleção:Artefatos", "Coleção:Agentes" e "Coleção:Contas". Cada grafo possui um, e apenas um, nó de cada um destes tipos. Um nó do tipo "Coleção:Y" é ligado ao nó "NomeGrafo$_x$" relativo ao seu grafo através de uma aresta do tipo POSSUI_Y, onde Y ∈ {"Processos", "Artefatos", "Agentes", "Contas"}. Cada nó do tipo "Coleção:Y" está ainda ligado a todos os nós do tipo Y pertencentes ao seu grafo através arestas do tipo POSSUI. Por exemplo, o nó "Coleção:Processos" do grafo "NomeGrafo$_1$" possui uma aresta possui ligando-o a cada no do tipo Processo daquele grafo.

Com a criação dos nós auxiliares, pode-se facilmente visualizar todos os processos, artefatos, agentes e contas de cada execução de *workflow*, sem a necessidade de percorrer o grafo inteiro.

Tabela 1 - Arestas auxiliares

Arestas auxiliares	**Nó origem**	**Nó destino**
POSSUI_GRAFO	Nó Referencia	NomeGrafo$_n$
POSSUI_CONTAS	NomeGrafo$_n$	Coleção: Contas
POSSUI_ARTEFATOS	NomeGrafo$_n$	Coleção: Artefatos
POSSUI_PROCESSOS	NomeGrafo$_n$	Coleção: Processos
POSSUI_AGENTES	NomeGrafo$_n$	Coleção: Agentes
POSSUI	Coleção: Contas	Conta
POSSUI	Coleção: Artefatos	Artefato*
POSSUI	Coleção: Processos	Processo*
POSSUI	Coleção: Agentes	Agente*

*Nós definidos pelo OPM

A Figura 12 demostra um subgrafo onde se tem o nó auxiliar "Coleção: Processos" relacionado ao nó Processo, e este, conectado ao nó Artefato através da

dependência USED, e também, ligado a outro nó Artefato através da dependência WAS_GENERATED_BY, sendo estes três últimos, nós e dependências do OPM.

As outras três formas de armazenamento (Relacional, XML e RDF) apresentam alguma dificuldade, seja na consulta ou no armazenamento de grafos, pois necessitam de adaptações, já que não possuem, em geral, suporte nativo a grafos. Nesta forma, não existe estes problemas. Outras considerações a respeito destas formas são apresentadas na próxima seção.

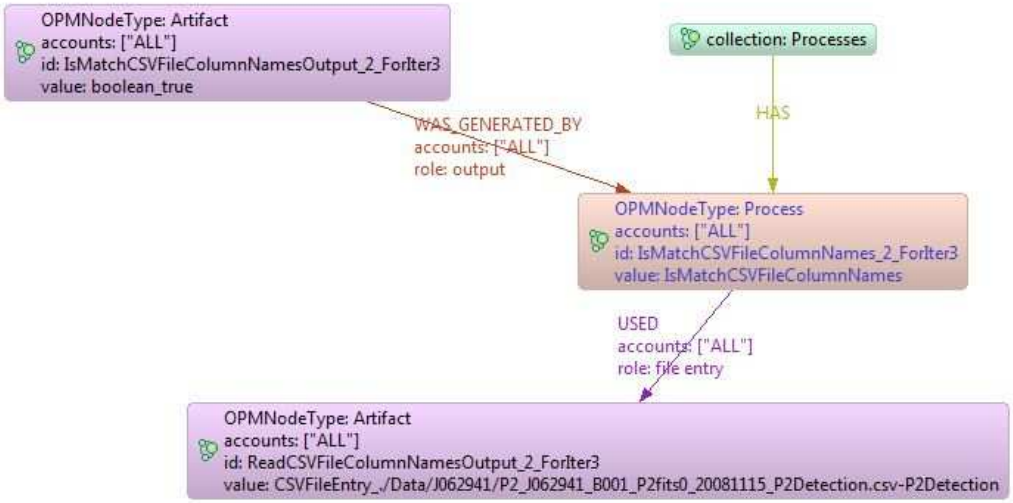

Figura 12 - Subgrafo armazenado no Neo4j

3.5 Considerações Finais

Ao analisar cada forma de armazenamento, torna-se necessário, também, avaliar as opções de linguagens de consulta suportadas por cada uma, pois não só um armazenamento eficiente é importante, como também a busca e recuperação dos dados armazenados. Logo, a linguagem de consulta pode representar um dos fatores determinantes para a escolha do SGBD.

Os SGBDR, geralmente, não oferecem um suporte nativo para grafos, e armazenam os nós e arestas em forma de listas. Documentos XML são hierárquicos e não representam objetos com vários “pais”. RDF possui uma linguagem de consulta que até pouco tempo atrás era muito deficiente para busca e recuperação de dados de proveniência (HOLLAND *et al.*, 2008). O armazenamento gerenciado por SGBDG

pode ser considerado o mais apropriado para dados de proveniência, pois os representa em sua forma nativa, o que torna, por exemplo, sua visualização intuitiva Além disso, o suporte a linguagens que suportam travessias em grafos são uma característica importante para consulta a dados de proveniência.

Dentre as quatro formas de armazenamento descritas neste capítulo, as que demonstram o maior potencial para a gerência dos dados de proveniência, é a de grafos e a Relacional. As soluções de armazenamento de grafos ainda são bem recentes. Portanto, na prática, não necessariamente irão apresentar um melhor desempenho que os SGBDR, os quais possuem décadas de desenvolvimento e estão mais amadurecidos, o que os torna ainda, apesar de suas deficiências apresentadas, fortes candidatos para a gerência de dados de proveniência.

Comparações de desempenho de consultas de proveniência entre um Banco de Dados de Grafo e um Banco Relacional já foram estudadas. VICKNAIR *et al*. (2010) fazem uma comparação entre o Neo4j e o SGBDR MySQL (MYSQL, 2013), porém, as consultas que envolviam travessias foram limitadas a uma determinada profundidade, o que não representa consultas típicas de proveniência. Além disso, o modelo Relacional utilizado para armazenamento dos dados é bem simples, consistindo de apenas duas tabelas, uma para armazenar os nós, e outra para as ligações entre eles. Existem alternativas para implementação destas consultas que podem ser mais eficazes, como o uso de visões recursivas e de procedimentos armazenados, as quais são investigadas nesta dissertação, e que eliminam a necessidade de limitar a profundidade da busca.

A fim de verificar e comparar na prática o desempenho entre os Bancos de Dados Relacionais e de Grafos para gerência de dados de proveniência, nos próximos capítulos, é apresentada a metodologia utilizada para análise e comparação de ambas as formas, e seus resultados.

Capítulo 4 – Estudo Comparativo: Metodologia

Neste capítulo, é apresentada a metodologia utilizada para fazer o estudo comparativo entre Banco de Dados Relacionais e de Grafos para armazenamento e manipulação de dados de proveniência. Primeiramente, são descritos os principais padrões de consultas de proveniência, e, em seguida, são descritas as consultas escolhidas para realização dos experimentos comparativos. Foram escolhidas consultas do *Third Provenance Challenge*, ou seja, consultas típicas de proveniência para realizar o experimento. Elas representam a maioria dos padrões descritos. Em seguida, são apresentadas as quatro formas utilizadas para a implementação das consultas e os tipos de comparações realizadas, tanto no contexto de carga de dados quanto no de execução das consultas. Por fim, são apresentados os critérios de medição, os programas, as bibliotecas de software e a configuração do computador utilizado nos experimentos.

4.1 Padrões de Consultas de Proveniência

Alguns trabalhos abordam os principais padrões de consultas de proveniência. WOODMAN *et al*. (2011) consideram que as consultas mais usuais de proveniência são as que retornam todas as dependências de dados diretas e indiretas, ou seja, que retornam um fragmento do grafo de proveniência que inclui todos os caminhos partindo de ou chegando a um determinado objeto, como, por exemplo: “Encontre todas as dependências diretas e indiretas de um objeto X”. Esta consulta pode ter duas variações: a busca por objetos anteriores (exemplo: “Encontre todos os objetos usado para gerar um objeto X”), ou por objetos posteriores (exemplo: “Encontre todos os objetos que são derivados a partir de X”. Ainda segundo WOODMAN *et al*. (2011), isto é geralmente considerado como o patamar de consulta que todos sistemas de gerenciamento de proveniência devem apoiar.

GADELHA *et al*. (2011), mapearam oito padrões principais de consultas de proveniência baseando-se nas consultas propostas nos três primeiros *Provenance Challenges* (CHALLENGES, 2013), nas consultas de proveniência encontradas na literatura acadêmica, e nas consultas concebidas em colaboração com os usuários científicos do SGWfC Swift (entre estes padrões estão incluídos o que WOODMAN *et al*. (2011) propõe). São eles:

- Atributo de entidade: Consultas baseadas em atributos de uma entidade do modelo de dados;
- Relacionamento de um passo: Consultas a entidades envolvidas em um relacionamento do modelo de dados, como, por exemplo, consumo ou produção de um arquivo por um processo;
- Relacionamento de múltiplos passos: São consultas a entidades envolvidas em um número arbitrário de relacionamentos no modelo de dados. É o caso onde a consulta geralmente retorna todas as dependências de dados diretas e indiretas de uma entidade ou ação, e, para isto, é necessário executar uma operação de travessia;
- Correspondência entre grafos de proveniência: Consultas para determinar similaridade entre grafos de linhagem. Elas podem incluir uma combinação dos outros três padrões citados acima;
- Executar resumos: São consultas que retornam atributos específicos de uma execução (por exemplo, memória utilizada) ou das entidades que o *workflow* contém (processos e seus respectivos conjuntos de dados de entrada e saída);
- Executar comparações: Essas consultas comparam múltiplas execuções de *workflows* em relação a algum atributo para analisar como o mesmo variou entre elas;
- Executar correlações: São consultas que correlacionam atributos de várias execuções de *workflows* diferentes;
- Resumo de execuções do mesmo *workflow*: Estas consultas agregam dados de um conjunto de execuções de um mesmo *workflow*. Um exemplo de consulta neste padrão seria extrair a média da precisão de um modelo computacional após um determinado número de execuções.

Para uma comparação entre SBDR e SBDG no contexto de gerência de dados de proveniência, e como parte da metodologia utilizada nesta dissertação, foi escolhida uma lista de consultas que abordam a maioria dos padrões citados acima. Elas são descritas na próxima seção.

4.2 Consultas do *Third Provenance Challenge*

Nesta seção, são apresentadas as consultas propostas pelo *Third Provenance Challenge* utilizadas nesta dissertação e suas classificações de acordo com os padrões de consultas de proveniência propostos por GADELHA *et al*. (2011).

O *Third Provenance Challenge* teve como principais objetivos: identificar os pontos fortes e fracos do OPM; determinar o quanto o OPM pode representar proveniência em uma variedade de tecnologias; e reunir a comunidade para discutir a interoperabilidade dos sistemas de proveniência (THIRD CHALLENGE, 2013). Para isto, o desafio propôs às equipes participantes a execução de um *workflow* cujo fluxo de atividades pode ser visto na Figura 13.

Este *workflow* representa uma parte do projeto Pan-STARRS, que se propõe a fazer pesquisas astronômicas, entre elas, analisar continuamente o céu através de um telescópio localizado no Havaí, e construir uma série temporal de dados. Isso ajuda a detectar objetos em movimento que podem sofrer impacto com a Terra, e a construir um catálogo do sistema solar e das estrelas visíveis no hemisfério norte (PAN-STARRS, 2013). Neste projeto, o *workflow* utilizado no desafio aparece entre o condutor de imagens geradas pelo telescópio e o banco de dados que armazena informações sobre elas. Mais especificamente, o *workflow* utiliza arquivos CSV como entrada de dados e armazena estes dados em tabelas de um BDR, executando uma série de verificações de consistência.

Além de propor o *workflow*, o desafio também sugere três consultas obrigatórias e treze consultas opcionais relacionadas à proveniência dos dados gerados pelo *workflow*. Desta lista de consultas, esta dissertação descartou as seguintes consultas opcionais: OQ2 e OQ9, pelos mesmos motivos apresentados por LIM *et al*. (2011), pois, para respondê-las, é necessário não apenas os dados de proveniência retrospectiva, mas também, uma consulta à definição do *workflow*, o que está fora do escopo desta dissertação, por se tratar de proveniência prospectiva. As consultas utilizadas nesta dissertação são descritas na Tabela 2 (as três primeiras – CQ1 a CQ3 – são obrigatórias e as demais – OQ1 a OQ13 – são opcionais).

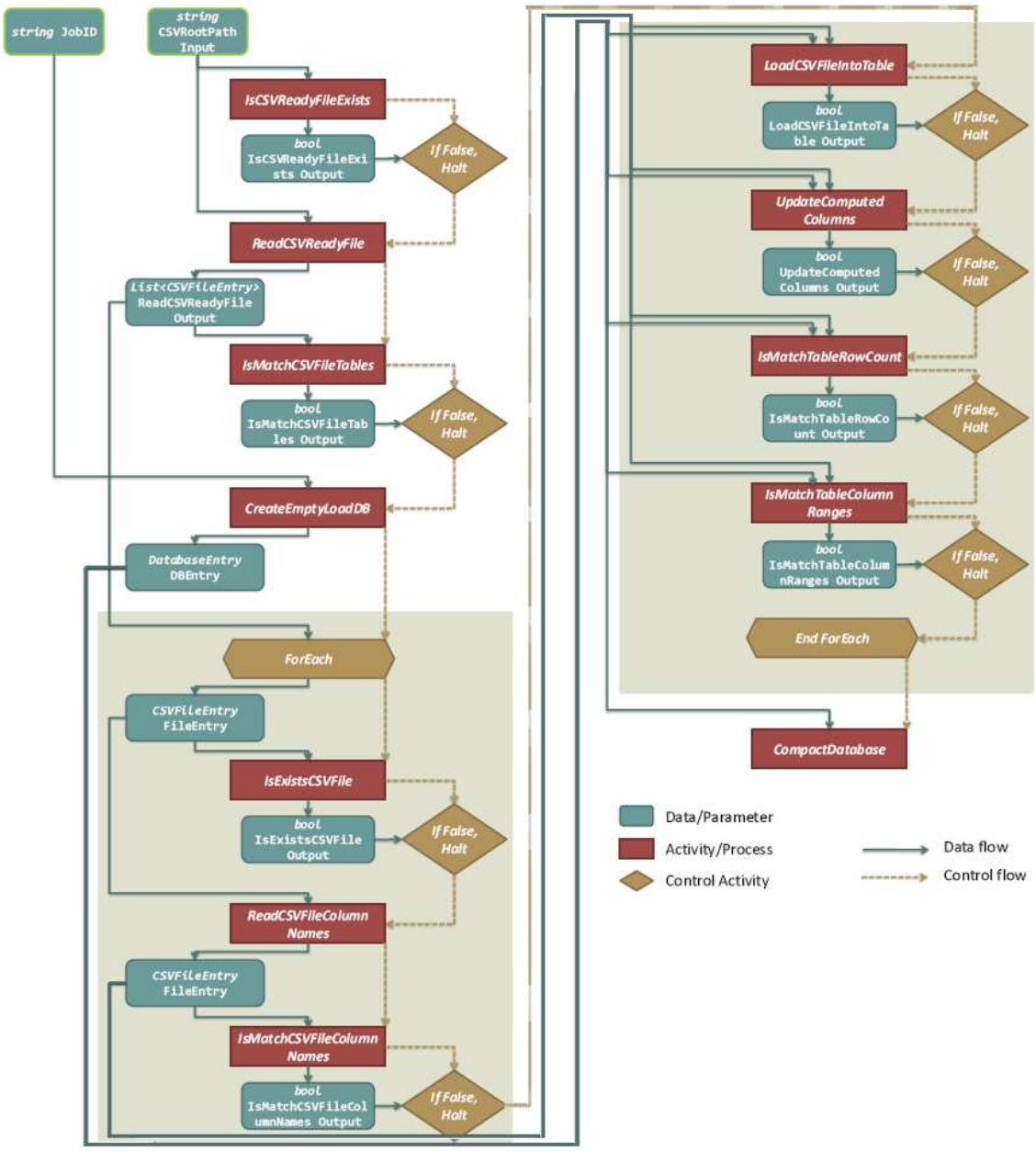

Figura 13 - Fluxo de atividades executadas pelo *workflow* proposto no *Third Provenance Challenge* (THIRD CHALLENGE, 2013)

O motivo da escolha destas consultas para realizar as comparações de busca e recuperação de dados de proveniência consiste no fato de que elas abordam a maioria dos padrões de consultas de proveniência mapeados por GADELHA *et al*. (2011). Logo, como parte da metodologia utilizada nesta dissertação consiste em utilizar consultas típicas de proveniência, as consultas propostas pelo desafio se mostram adequadas. A Tabela 3 demonstra como as consultas utilizadas nesta dissertação se enquadram nos padrões abordados por GADELHA *et al*. (2011). Na próxima seção, são discutidos os detalhes relacionados à implementação das consultas.

Tabela 2 - Lista de consultas de proveniência propostas pelo *Third Provenance Challenge* utilizadas nos experimentos comparativos

Consulta	Descrição
CQ1	Dada uma detecção, quais arquivos CSV contribuíram para ela?
CQ2	O usuário considera uma tabela que contém valores não esperados. A verificação de intervalo (IsMatchTableColumnRanges) foi realizada para esta tabela?
CQ3	Quais operações eram estritamente necessárias para a tabela Image conter um valor (não computado) em particular?
OQ1	A execução do *workflow* parou devido a uma falha na checagem "IsMatchTableColumnRanges". Quantas tabelas foram carregadas com sucesso antes do *workflow* falhar*?*
OQ3	Um arquivo CSV ou "cabeçalho" é eliminado durante a execução do *workflow*. Quanto tempo se passou entre um teste "IsMatchCSVFileTables" (quando o arquivo existia) bem sucedido e um teste "IsExistsCSVFile" sem sucesso (quando o arquivo foi excluído)?
OQ4	Por que esta entrada está no banco de dados?
OQ5	Um usuário executa o *workflow* muitas vezes em diferentes conjuntos de dados. Ele quer determinar quais execuções falharam.
OQ6	Determinar o passo onde ocorreu falha.
OQ7	Determinar os dados e granularidades associadas dos dados processados, quando uma falha ocorreu.
OQ8	Quais passos foram concluídos com êxito antes de ocorrer a falha?
OQ10	Para uma execução do *workflow* , determinar as entradas de usuário.
OQ11	Para uma execução do *workflow* , determinar os passos que exigiram entradas do usuário.
OQ12	Para uma execução do *workflow* que falhou, quais arquivos foram processados com sucesso?
OQ13	Para uma execução do *workflow* , apresentar as seguintes visões de proveniência: visões de dependência de dados e visões de dependência de passos.

Tabela 3 – Enquadramento das consultas por padrões

Padrões	Consultas Obrigatórias (CQ)			Consultas Opcionais (OQ)										
	1	2	3	1	3	4	5	6	7	8	10	11	12	13
Atributo de entidade	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Relacionamento de um passo	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Relacionamento de múltiplos passos	X		X			X				X				X
Correspondência de grafos de proveniência														
Executar resumos	X	X	X	X	X	X		X	X	X	X	X	X	X
Executar comparações							X							
Executar correlações														
Resumo de execuções do mesmo workflow														

4.3 Implementação das Consultas

As quatorze consultas do *Third Provenance Challenge* escolhidas para o estudo comparativo foram implementadas de quatro formas diferentes nesta dissertação, sendo duas para cada uma das formas de gerência de dados em análise: em SQL (usando visões recursivas); como procedimentos armazenados (alguns reproduzindo as inferências de múltiplos passos do OPM); em Java, utilizando a API do Neo4j; e na linguagem Cypher, sendo as duas primeiras executadas em um Banco de Dados Relacional e as demais, em um Banco de Dados de Grafos (Neo4j). O objetivo de se implementar as consultas nestas quatro formas é comparar e avaliar o desempenho de cada uma delas. A implementação das consultas nas quatro formas é apresentada no Apêndice A. O uso da recursividade é necessário para implementar as regras inferência de múltiplos passos definidas no OPM, constituindo assim uma solução que pode ser facilmente reproduzida em SGBD com suporte a este recurso.

Após o entendimento das consultas propostas pelo desafio, foi estudada a API Java do Neo4j, cada uma das linguagens utilizadas nesta dissertação e a melhor forma de otimização para as consultas implementadas em suas respectivas formas. Na próxima seção, são apresentadas as comparações realizadas no experimento, tanto no contexto de carga de dados, quanto no de execução das consultas.

4.4 Comparações Realizadas

O experimento realizado é composto por duas partes. Na primeira, foram carregados três conjuntos de dados de proveniência em três bancos de dados distintos, utilizando tanto o PostgreSQL, quanto o Neo4j. Para cada conjunto, foram medidos o tempo de carga, o uso do processador, o consumo de memória e o espaço ocupado em disco.

Na segunda parte do experimento, as consultas selecionadas, implementadas das quatro formas descritas, foram executadas sobre cada conjunto de dados armazenados na primeira parte do experimento. Ao final, foram feitas comparações considerando o tempo de execução, o uso do processador e o consumo de memória.

4.5 Ambiente de Testes

Para a realização dos experimentos, foram preparados três conjuntos de dados diferentes baseados em documentos XML que seguem o formato OPM. Estes documentos foram gerados a partir de execuções de *workflows* em diferentes SGWfC, por oito equipes participantes do *Third Provenance Challenge*: Karma3, KCL, NCSA, PASS3, SDSCPc3, TetherlessPC3, UCDGC, VisTrails3 (PARTICIPATING TEAMS, 2013). Algumas equipes não forneciam dados de proveniência relativos a execuções do *workflow* com falhas. Para estas, foram criados documentos similares aos gerados pelas respectivas equipes, mas modificados de forma a simular um erro de execução.

A Tabela 4 mostra características de cada documento gerado por cada participante do desafio, como tamanho e quantidade de nós e relacionamentos no padrão OPM explícitos em cada documento. Além disso, esta tabela apresenta os sistemas utilizados por cada equipe para captura dos dados de proveniência. A partir destes documentos, três conjuntos de dados foram criados:

- CD1 – conjunto composto por uma cópia de cada documento mostrado na Tabela 4, totalizando 31 documentos.
- CD50 – conjunto composto por 50 cópias dos mesmos documentos do conjunto CD1, totalizando 1550 documentos.
- CD100 – conjunto composto por 100 cópias de cada documento de CD1, totalizando 3100 documentos.

Tabela 4 - Dados dos Documentos XML utilizados

Participantes/ Sistema utilizado	**Documentos XML**	**Tamanho (KB)**	**Quantidade de nós**	**Quantidade de relacionamentos**
Karma3/Karma (KARMA, 2013)	Karma3_J062941-opm	100	60	144
	Karma3_J062941-opm-error*	100	60	144
	Karma3_J062942-opm	103	60	144
	Karma3_J062943-opm	137	60	141
	Karma3_J062944-opm	98	59	141
	Karma3_J062945-opm	98	59	141
KCL/ SourceSource (SOURCESOURCE, 2013)	kcl-opm	351	436	460
	kcl-opm-error*	351	436	460
NCSA/Tupelo (TUPELO, 2013)	NCSA_J609241_output	50	86	119
	NCSA_J609241_output_error*	50	86	119
	NCSA_J609242_output	49	86	118
	NCSA_J609243_output	49	86	118

	NCSA_J609244_output	49	86	118
	NCSA_J609245_output	49	86	118
PASS3/PASS	PASS3_opm	5619	14707	14758
	PASS3_opm_error*	5720	14707	14758
SDSCPc3/Kepler	SDSC_pc3-J062941.opt-query3*	23	31	31
	SDSC_pc3-J062941.good	97	130	158
TetherlessPC3/ ProtoProv (PROTOPROV, 2013)	TetherlessPC3_opm	39	63	94
	TetherlessPC3_opm_error*	40	63	94
UCDGC/Kepler	J062941-halt1-IsCSVReadyFileExists_1*	3	10	7
	J062941-halt2-IsMatchCSVFileTables_1*	13	31	53
	J062941-halt3-IsExistsCSVFile_1*	20	48	93
	J062941-halt4-IsMatchCSVFileColumnNames_1*	34	85	145
	J062941-halt5-LoadCSVFileIntoTable_1*	115	353	419
	J062941-halt6-UpdateComputedColumns_1*	426	1293	1620
	J062941-halt7-IsMatchTableRowCount_1*	420	1281	1591
	J062941-halt8-IsMatchTableColumnRanges_1*	424	1291	1610
	J062941-success_1	426	1293	1620
VisTrails3/ VisTrails	Vistrails3_workflow_opm	56	78	100
	Vistrails3_workflow_opm_err*	34	45	53

* Execuções com falha.

Um banco de dados foi criado para cada conjunto de dados, tanto no Neo4j (versão community-2.0.0-M03) quanto no SGBDR PostgreSQL (versão 9.1). A carga de cada conjunto de dados tanto no PostgreSQL quanto no Neo4j foi executada utilizando programas feito em Java. Ambos os programas são similares no modo de capturar um nó ou relacionamento dos documentos XML. A diferença entre eles está no modo de inserção destes dados, que deve obedecer ao modelo Relacional do OPMProv (no caso do PostgreSQL), e ao modelo de armazenamento criado para esta dissertação para armazenar dados de proveniência em Banco de Dados de Grafos (descrito no Capítulo 3). Neste experimento foram utilizados o Neo4j embutido ao Java, métodos da API Java do Neo4j e o *driver* JDBC do PostgreSQL. Em relação às transações, os programas abriam uma nova a cada lote de documentos XML iguais e fechavam ao final da carga deste lote. Portanto, em todos os conjuntos foi utilizado o mesmo número de transações, sendo que no CD1 cada transação era responsável pela carga de apenas um documento, e no CD50 e CD100 de 50 e 100 documentos, respectivamente.

Índices foram utilizados para minimização do tempo de processamento das consultas tanto no Neo4j quanto no PostgreSQL. No total, foram criados 27 índices no PostgreSQL e 8 no Neo4j. Além disso, foram mantidas suas respectivas políticas de *cache* padrão e foi evitada a utilização de *caches* de consulta.

Cada uma das quatorze consultas oriundas do *Third Provenance Challenge* escolhidas para este experimento foi executada dez vezes para cada uma das quatro formas em que foram implementadas; para cada uma das oito equipes participantes do desafio; e para cada um dos três conjuntos de dados; totalizando 13440 execuções. Para a medição do tempo de execução, uso do processador e da memória de cada consulta, foi utilizada a biblioteca SIGAR (SIGAR, 2013), a qual, além de ser um *software* livre, captura métricas referentes ao Sistema Operacional e não apenas à máquina virtual Java. Para cada cenário, foram descartados o pior e melhor resultado do tempo de execução, uso do processador e memória e a média entre os resultados restantes foi calculada.

As consultas foram executadas sobre os dados de apenas um documento XML de cada equipe participante nas dez execuções, exceto para a consulta OQ5, que foi a única que exigia uma busca em todas as execuções de *workflow* armazenadas no banco. As consultas OQ1, OQ3, OQ6, OQ7, OQ8 e OQ12 foram executadas sobre dados de proveniência relativos a uma execução de *workflow* que falhou. Referente à equipe UCDGC, estas consultas foram executadas sobre uma cópia do documento XML J062941-halt8-IsMatchTableColumnRanges_1. As demais consultas foram executadas sobre cópias de execuções de *workflow* bem sucedidas. Para as equipes Karma3 e NCSA, estas consultas foram executadas sobre uma cópia do documento Karma3_J062941-opm e NCSA_J609241_output respectivamente.

Todo o experimento foi executado em um computador com o Sistema Operacional Windows 7 Professional Service Pack 1 64-bit, 8 GB de RAM, processador Intel Core i5-2310 2.90GHz e 1 TB de espaço em disco rígido. No próximo capítulo, são apresentados os resultados do estudo comparativo.

Capítulo 5 – Resultados Experimentais

Neste capítulo, são apresentados os resultados experimentais relativos ao armazenamento e à busca e recuperação de dados de proveniência utilizando Bancos de Dados Relacional e de Grafos para as bases criadas. Além disso, são feitas análises comparativas entre os resultados obtidos.

5.1 Carga de Dados

Com base nas medições capturadas durante a criação dos bancos de dados de proveniência, nas próximas seções é feita uma análise comparativa dos processos de carga de dados no Neo4j e no PostgreSQL, considerando os seguintes fatores: tempo de execução, uso de processador, consumo de memória e espaço ocupado em disco. As medições referentes às cargas dos conjuntos CD1, CD50 e CD100 podem ser vistas no Apêndice B. No PostgreSQL, a fim de minimizar o tempo de carga, os índices foram criados no final da carga. Para o Neo4j, no entanto, essa estratégia não se mostrou eficiente e, por isso, os índices foram criados durante a carga.

A figura 14 demonstra o tempo total de carga de cada conjunto de dados. Podemos ver que, no CD1, o PostgreSQL foi mais rápido, sendo o tempo 1,35 vezes menor, enquanto que nos maiores conjuntos, a vantagem do PostgreSQL foi maior, alcançando um tempo de 1,59 vezes menor no CD50 e 1,8 vezes menor no CD100. Considerando que os conjuntos CD50 e CD100 são 50 e 100 vezes maiores do que o CD1, respectivamente, pode-se observar a partir dos tempos de carga destes conjuntos que houve uma perda de desempenho no Neo4j, pois seus tempos foram maiores que o tempo do CD1 multiplicado por 50 e 100. Esta redução de desempenho na inserção se deve, em parte, aos índices, pois a cada inserção, cada registro precisa ser reposicionado em todos os índices que o utilizam. Este desempenho foi ainda pior em experimentos que criavam os índices posteriormente, cujos resultados não estão descritos nesta dissertação.

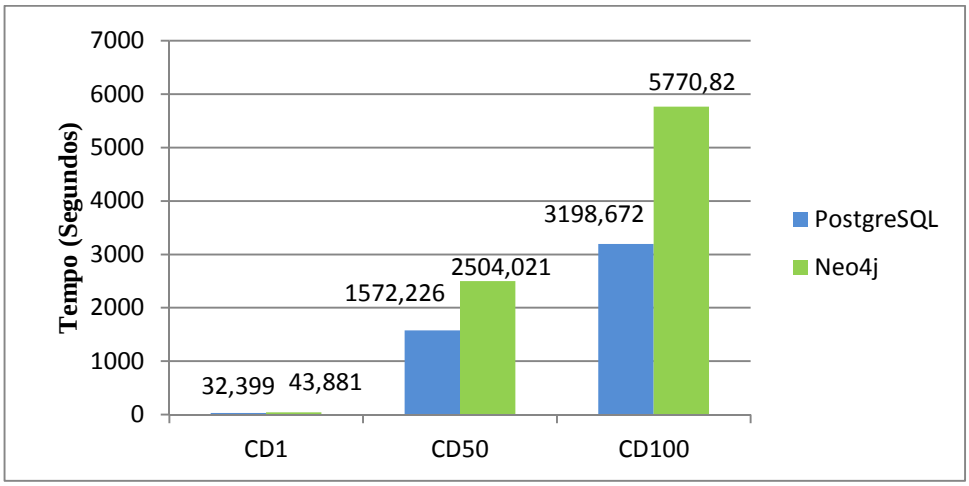

Figura 14 - Tempo total de carga por base

Em relação ao uso do processador, conforme mostra a Figura 15, as cargas dos conjuntos de dados no Neo4j apresentaram maior uso médio do processador pois foi gasto menos tempo em espera por operações de disco. Na carga no CD1, o uso do processador obteve a maior diferença, chegando a 68% a mais no Neo4j. Esta diferença diminuiu no CD50 (37%) e mais ainda no CD100 (26%).

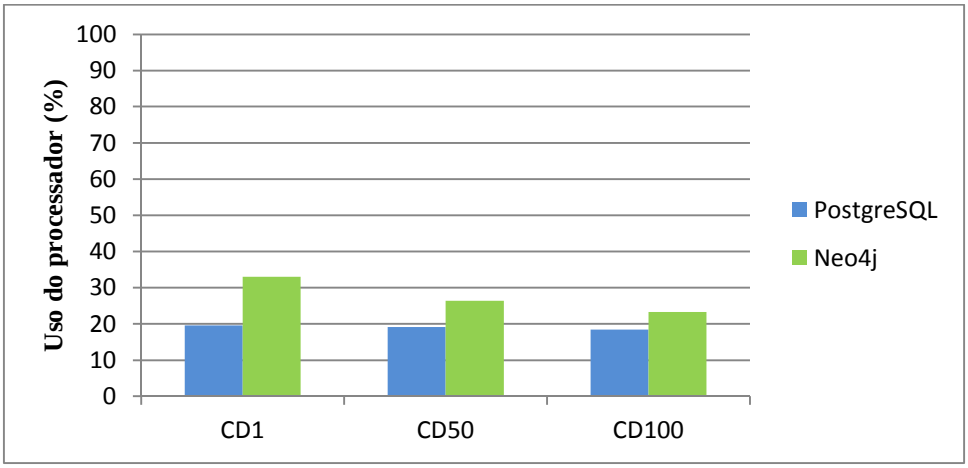

Figura 15 - Uso médio do processador

Em relação ao consumo médio de memória, os três conjuntos de dados tiveram um consumo substancialmente maior no Neo4j (variando entre 7 e 22 vezes) conforme apresentado na Figura 16. Este alto consumo de memória do Neo4j é devido à política de *cache* de objetos do Neo4j. O Neo4j possui dois tipos de *cache*: um cache de arquivo (de tamanho fixo) e um *cache* de objetos (cujo tamanho depende da memória alocada para a memória virtual Java). O PostgreSQL, por sua vez, possui apenas um *cache* para

arquivos, cujo tamanho não varia ao longo do tempo. O *cache* de objetos do Neo4j se expande até o limite permitido pela máquina virtual Java. Quanto mais objetos são criados, mais memória é alocada, pois todo objeto criado ou acessado permanece neste *cache* até que tenha que ser substituído por necessidade de espaço para outro objeto. O *cache* de arquivos do Neo4j tem tamanho fixo, como o do PostgreSQL.

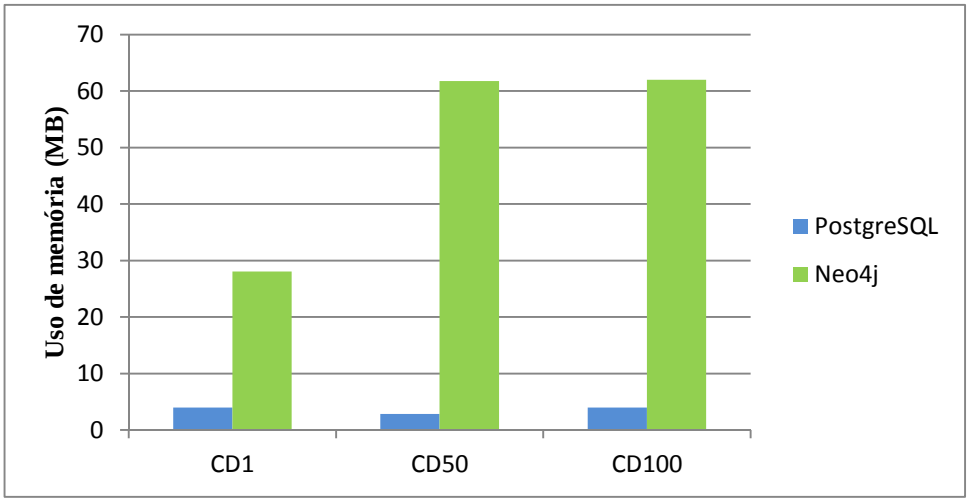

Figura 16 - Uso médio de memória

Finalizada a carga dos conjuntos de dados, foi medido o espaço em disco ocupado por cada base. Conforme mostra a figura 17, em termos de utilização de espaço em disco, o Neo4j apresentou o pior resultado em todos os conjuntos de dados. A diferença variou entre 2,23 (no CD1) a 2,39 (no CD100) vezes a mais o espaço ocupado pelos mesmos conjuntos de dados carregados no PostgreSQL. Esta diferença pode ser explicada em parte devido ao fato de o serviço de indexação do Lucene (utilizado pelo Neo4j) armazenar o nome de cada atributo juntamente com seu valor nos índices. Isso não ocorre em Bancos de Dados Relacionais. Apesar do uso de apenas 8 índices, sendo quatro que indexam dois atributos e outros quatro que indexam todos os atributos de um determinado tipo de nó, o espaço ocupado por eles é maior do que o ocupado pelos 27 índices do PostgreSQL. No CD100, por exemplo, o espaço ocupado pelos índices é 3,76 GB no Neo4j, e 2,58 GB no PostgreSQL (1,18 GB a menos), em outras palavras, os índices ocupam cerca de 40% do espaço em disco do Neo4j e 67,5% no PostgreSQL. Portanto, além do espaço relativo ocupado pelos índices ser maior no Neo4j, o espaço relativo ocupado pelos dados é ainda maior se comparado ao PostgreSQL (60% da base é ocupado pelos dados no Neo4j e 32,5% no PostgreSQL). Cenário semelhante ocorre nos conjuntos CD1 e CD50. Outro motivo que justifica o maior uso de espaço ocupado

pelo Neo4j está relacionado ao fato de que todos os elementos (nós, arestas e atributos) formam listas encadeadas (criadas pelo próprio Neo4j). Cada atributo de um nó contém ponteiros para outros atributos do mesmo nó. Cada aresta contém ponteiros para a anterior e a próxima. Isto não ocorre no BDR.

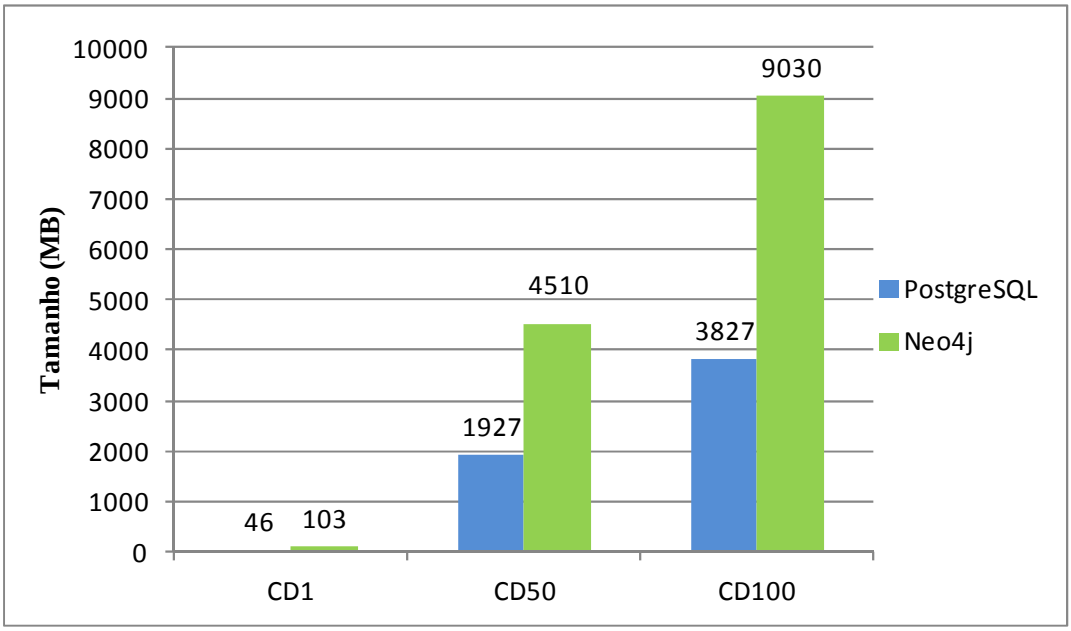

Figura 17 – Tamanho das Bases pós carga

Os resultados obtidos mostram, portanto, que, em termos de recursos computacionais (memória, processador e espaço em disco) e tempo de carga, o Neo4j exigiu muito mais do que o PostgreSQL.

A seguir, são apresentadas as análises comparativas em relação ao desempenho verificado durante as execuções das consultas.

5.2 Desempenho durante as Execuções das Consultas

A partir das medições capturadas durante as execuções das consultas, nas próximas seções são feitas análises comparativas entre as alternativas de gerência de dados consideradas nesta dissertação. Para tanto, são considerados tempo de execução, uso de processador e consumo de memória. Como cada consulta foi executada dez vezes para cada uma das oito equipes, as comparações apresentadas nas próximas seções representam a média destas execuções (descartanto o pior e melhor resultado). As

medições referentes às execuções de cada consulta no CD1, CD50 e CD100 podem ser vistas no Apêndice C.

Para analisar e comparar o tempo médio de execução das consultas, elas foram separadas em dois grupos: as que necessitam executar travessias no grafo, e as que não necessitam. Na próxima seção, é abordado o primeiro grupo.

5.2.1 Consultas que Envolvem Travessias no Grafo

A figura 18 mostra o tempo médio de execução das consultas que envolvem travessias no grafo de proveniência para cada alternativa de implementação considerada, ou seja, em SQL (utilizando visões recursivas), procedimentos armazenados, API Java do Neo4j e Cypher, considerando cada conjunto de dados. Para facilitar a visualização, os gráficos mostrados nesta figura, apresentam a dimensão “Tempo” em escala logarítmica (base 2).

A partir da Figura 18, pode-se observar que o uso de SQL com visões recursivas foi substancialmente a pior abordagem em termos de tempo de execução. Como era de se esperar, como as visões não suportam a passagem de parâmetros, várias travessias foram executadas desnecessariamente sobre todas as execuções de *workflow* armazenadas nos bancos Relacionais. O uso de procedimentos armazenados, por outro lado, suporta a passagem de parâmetros, e, por isso, teve um desempenho bem melhor do que o obtido com SQL e visões recursivas. Em geral, a linguagem Cypher e a API Java do Neo4j apresentaram os melhores resultados, sendo a API Java a solução que executou as consultas no menor tempo. Em alguns casos, para o CD1, a linguagem Cypher teve desempenho pior do que o obtido com procedimentos armazenados, e em todos os conjuntos de dados foi pior que a API Java. Isto pode ser explicado pelo fato do plano de execução das consultas implementadas em Cypher ser gerado pelo Neo4j, e na API Java termos condições de adotar estratégias de execução diferentes e mais eficientes, mostrando a necessidade de melhorias no otimizador de consultas utilizado pelo Neo4j para processamento de consultas em Cypher.

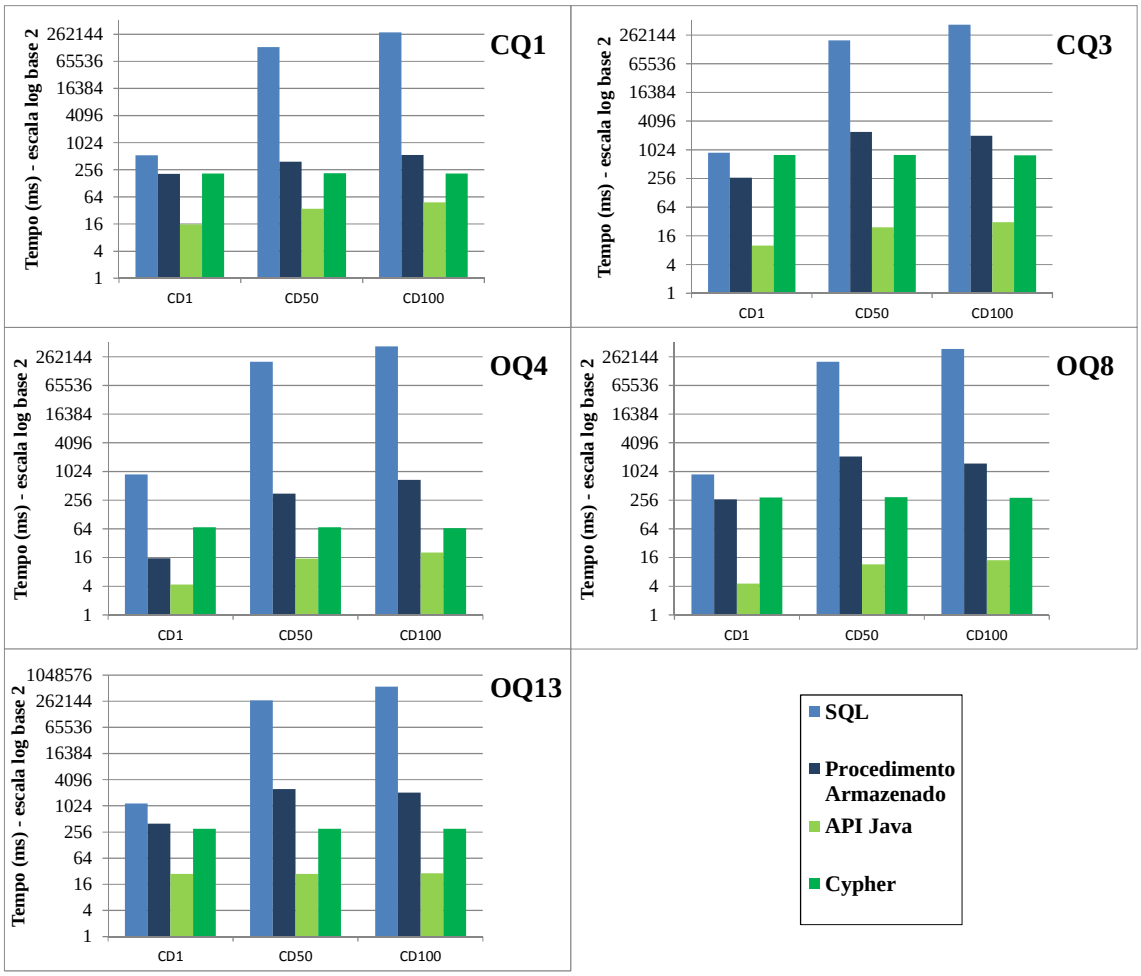

Figura 18 - Tempo médio de execução das consultas que envolvem travessias

A Figura 19 demonstra a média do uso do processador das consultas que envolvem travessias no grafo de proveniência. Em todas as consultas, os procedimentos armazenados foram a forma que menos utilizou o processador, enquanto que as outras três formas tiveram uma média bem similar. Todas as formas mantiveram um uso médio entre 20% e 26%, exceto para a CQ1 com procedimentos armazenados, que teve um uso médio em torno de 15% nos três conjuntos de dados. Esta diferença pode ser explicada pelo fato de que a CQ1 é a única que utiliza as regras de inferência de múltiplos passos *Muti-Step wasGeneratedBy*, definidas pelo OPM, enquanto que as demais (CQ3, OQ4, OQ8 e OQ13) implementam as regras baseadas no *Muti-Step wasTriggeredBy* para executar travessias.

Pode-se observar ainda que independentemente do volume de dados armazenado em cada banco de dados, o uso do processador não se altera consideravelmente.

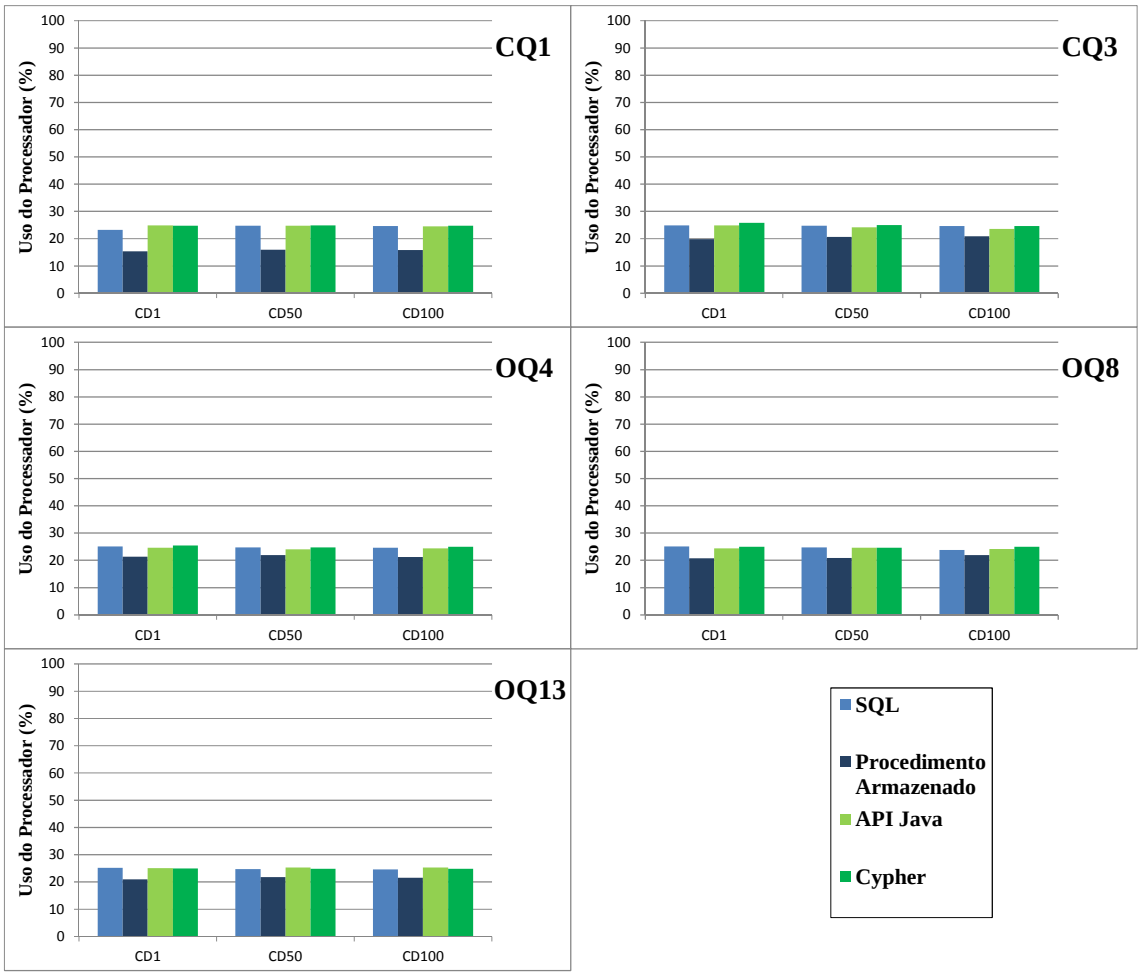

Figura 19 - Uso médio do processador na execução das consultas que envolvem travessias

Em termos de memória, pode-se observar, através da Figura 20, que no geral, a API Java teve o maior consumo médio. Isso se deve ao *cache* de objetos. Nas consultas CQ3 e OQ8, a alternativa que utiliza SQL chegou a ter consumo mais alto devido às junções desnecessárias. O Cypher consumiu menos memória do que a API Java na maioria dos casos, porém, não se aproximou dos procedimentos armazenados, os quais, nos conjuntos de dados 50 e 100, foram a forma que menos consumiu, perdendo apenas para a SQL na maioria das consultas no CD1.

Quanto aos procedimentos armazenados, observa-se um comportamento diferenciado na CQ1, apresentando um consumo menor em relação às demais consultas. O mesmo comportamento foi verificado no uso médio do processador e deve-se à utilização de outras regras de inferência de múltiplos passos definidas no OPM.

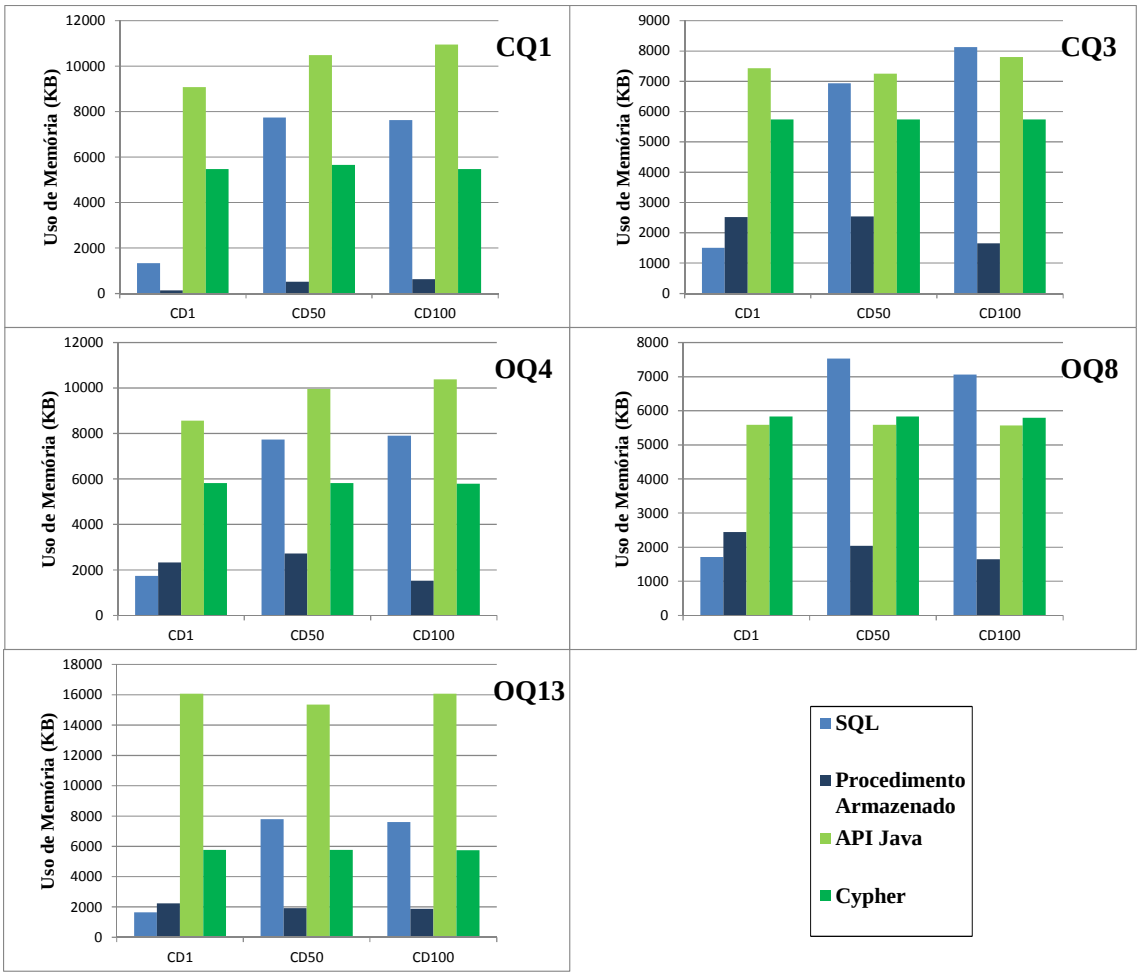

Figura 20 - Uso médio de memória na execução das consultas que envolvem travessias

5.2.2 Consultas que Não Envolvem Travessias no Grafo

Para as consultas que não envolvem travessias no grafo, ou seja, consultas que no geral são mais simples para os SGBD Relacionais, pois requerem poucas junções entre tabelas, as médias dos tempos de execução podem ser vistas na Figura 21. Nesta figura, apenas o gráfico que monstra a consulta OQ1 possui a dimensão “Tempo” na escala logarítmica (base 2). Além disso, como estas consultas não envolvem travessias, não foram utilizadas as visões recursivas na implementação em SQL.

Conforme mostra a Figura 21, chama a atenção o desempenho inferior obtido com o uso do Cypher. Os mesmos índices foram utilizados na solução com API Java, o que reforça que o problema está na implementação dos planos de execução da linguagem Cypher. Além disso, pode-se observar que ao se comparar as formas de executar as consultas no contexto Relacional, os procedimentos armazenados foram ligeiramente mais rápidos que as consultas executadas em SQL. Isto acontece porque os

procedimentos armazenados são compilados previamente pelo SGBDR e, portanto, seus planos de execução já estão prontos no momento da execução.

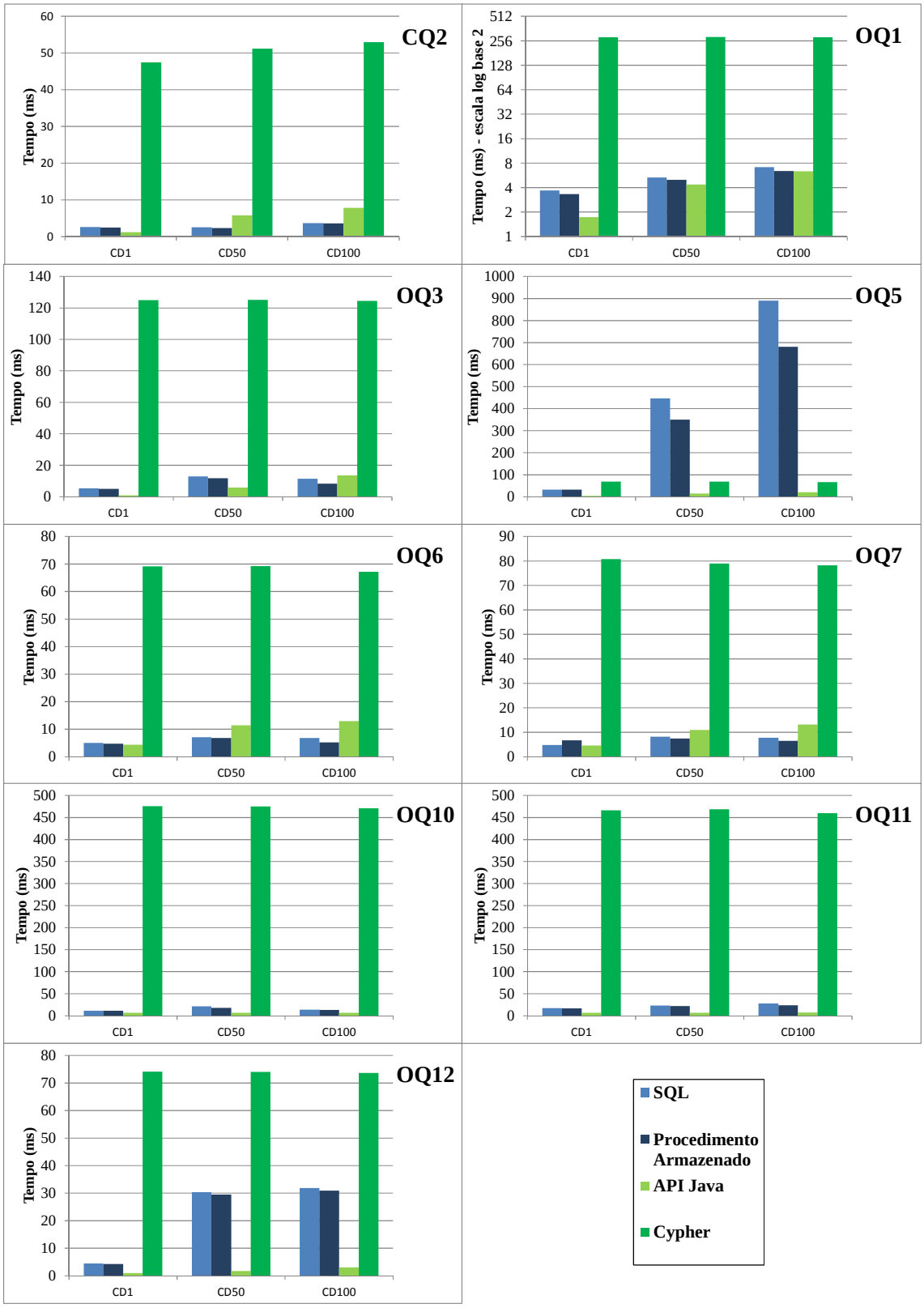

Figura 21 - Tempo médio de execução das consultas que não envolvem travessias

No geral, a API Java do Neo4j e os procedimentos armazenados se revezaram como a melhor solução, com uma pequena diferença de tempo entre elas, diferentemente do primeiro grupo de consultas, onde a diferença era grande. Contudo, uma consulta saiu do padrão das demais. A OQ5, executada em SQL e procedimento armazenado, teve uma média muito alta comparada às formas de consulta no contexto de grafos. Esta diferença pode ser explicada pelo fato de que ela possui um padrão diferente das demais. Ela é a única que compara múltiplas execuções de *workflows*, e, como retorna vários registros de uma determinada tabela, principalmente nos maiores conjuntos de dados, e para cada registro retornado é feita uma busca em outra tabela, o que aumenta o tempo de execução.

Em termos de uso de processador, conforme mostra a Figura 22, as alternativas que utilizam Cypher e API Java do Neo4j mantiveram o mesmo comportamento apresentado nas consultas que envolviam travessias, e os procedimentos armazenados continuam sendo a forma que menos utiliza o processador. A maior diferença está nas consultas em SQL, que mostraram uma queda na utilização do processador, variando entre 10% e 20%. Isto acontece porque para estas consultas não são feitas junções desnecessárias, o que acontecia nas consultas analisadas anteriormente.

Conforme mostra a Figura 23 (com a dimensão “Uso de memória” em escala logarítmica na base 2), pode-se observar um padrão de consumo médio de memóra bem definido em relação às alternativas com Cypher e API Java. Ambas possuem um consumo alto e bastante semelhante, em parte, por causa da política do *cache* de objetos do Neo4j. Nota-se, no entanto, que, em termos absolutos, o consumo de memória foi menor do que o ocorrido nas consultas com travessias. Isto se dá porque o número de objetos instanciados em uma travessia é muito maior do que no caso aqui considerado, evidenciando o papel do *cache* de objetos na explicação deste comportamento. Já as alternativas que utilizam SQL e procedimentos armazenados apresentaram um consumo bem menor, se revezando como a forma que menos consome. Na seção a seguir, são apresentas as conclusões gerais do experimento.

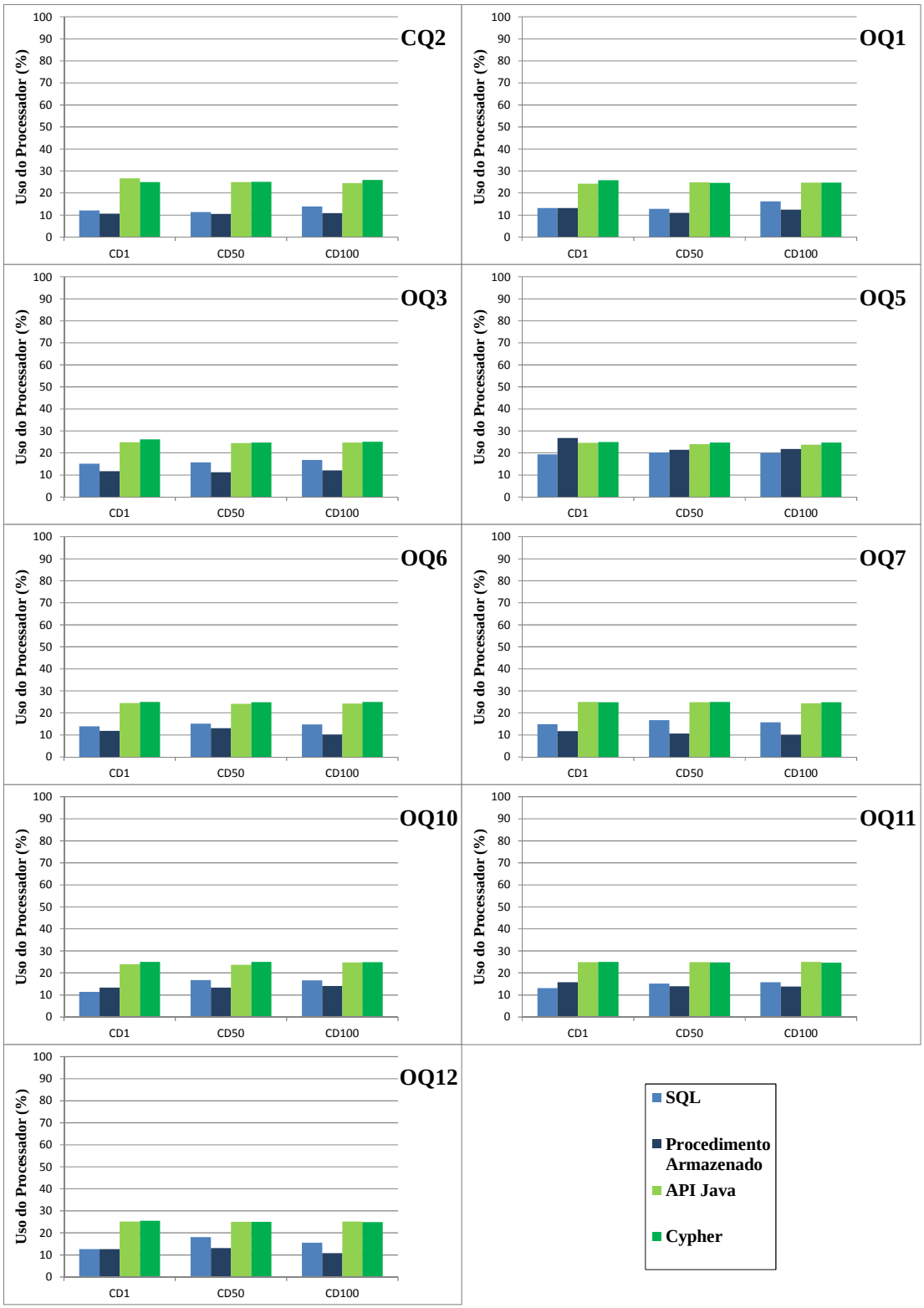

Figura 22 - Uso médio do processador na execução das consultas que não envolvem travessias

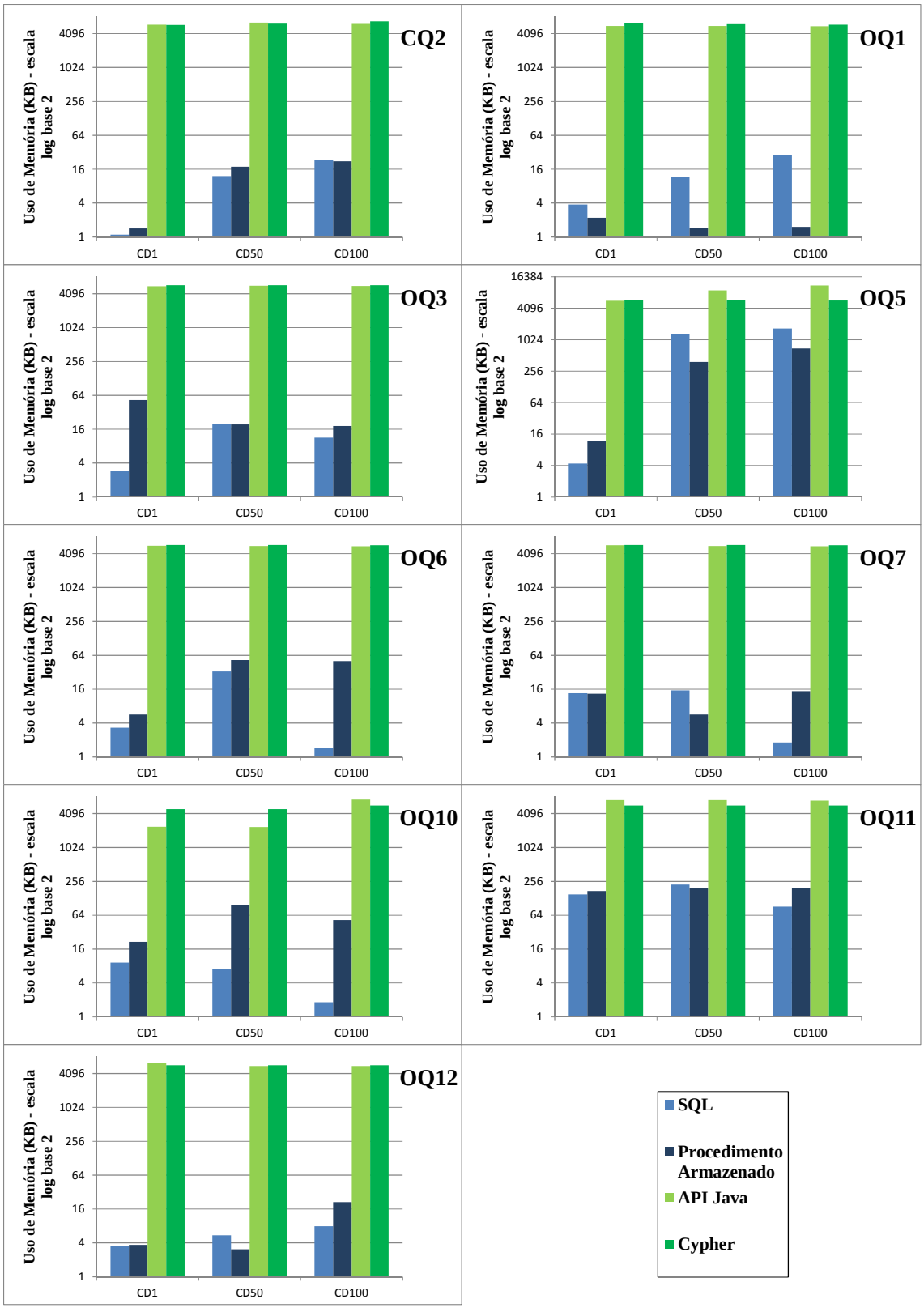

Figura 23 - Consumo médio de memória na execução das consultas que não envolvem travessias

5.3 Considerações Finais do Experimento

A partir dos resultados obtidos, verificamos que o experimento obteve êxito no que se propôs a analisar. Todos os ambientes considerados foram capazes de armazenar os dados de proveniência seguindo o padrão OPM e todas as alternativas de implementação das consultas foram capazes de responder a todas as consultas selecionadas. Quanto aos resultados, podemos observar que nenhuma solução apresentou superioridade em todos os aspectos analisados.

Em relação ao uso de recursos computacionais (processador, memória e espaço em disco), tanto na carga quanto na execução das consultas, o Neo4j exigiu muito mais do que o PostgreSQL, independentemente da forma com que as consultas foram implementadas.

Em termos de tempo de execução, o Neo4j foi substancialmente melhor que o PostgreSQL em todas as consultas que envolviam travessias. Na OQ13, por exemplo, para o CD100, a média do tempo de execução da consulta implementada em API Java foi de 28 milisegundos, e em SQL foi superior a nove minutos, ou seja, a API Java foi 20.079,87 vezes mais rápida, e em relação aos procedimentos armazenados (que executaram a mesma consulta em dois segundos) foi 72,92 vezes mais rápida.

Para as consultas que não envolviam travessias, a API Java obteve o melhor tempo médio na maioria das consultas. Considerando que neste grupo há nove consultas, e que em cada consulta foi medido o tempo em três conjuntos de dados (totalizando 27 médias), a API Java obteve o melhor tempo em 20 médias e perdeu em 7 casos, sendo que nestes a diferença não passou de 8 milisegundos. As tabelas 5, 6 e 7 demonstram as quatros formas implementadas ordenando-as pelo tempo médio por consulta. Conforme mostra a Tabela 5, a API Java obteve o melhor tempo médio em todas as consultas no CD1. Para o CD50, apresentado na Tabela 6, a API perdeu em 3 casos, e para a CD100 (Tabela 7), a API perdeu em 4 casos.

Tabela 5 - Classificação das alternativas utilizadas para implementação das consultas segundo o tempo médio de execução no CD1

Forma de Implementação	Consultas Obrigatórias			Consultas Opcionais										
	1	**2**	**3**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**11**	**12**	**13**
SQL	4	3	4	3	3	4	2	3	2	4	3	3	3	4
Proc. Arm.	2	2	2	2	2	2	3	2	3	2	2	2	2	3
API	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Cypher	3	4	3	4	4	3	4	4	4	3	4	4	4	2

Tabela 6 - Classificação das alternativas utilizadas para implementação das consultas segundo o tempo médio de execução no CD50

Forma de Implementação	Consultas Obrigatórias			Consultas Opcionais										
	1	**2**	**3**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**11**	**12**	**13**
SQL	4	2	4	3	3	4	4	2	2	4	3	2	3	4
Proc. Arm.	3	1	3	2	2	3	3	1	1	3	2	2	2	3
API	1	3	1	1	1	1	1	3	3	1	1	1	1	1
Cypher	2	4	2	4	4	2	2	4	4	2	4	4	4	2

Tabela 7 - Classificação das alternativas utilizadas para implementação das consultas segundo o tempo médio de execução no CD100

Forma de Implementação	Consultas Obrigatórias			Consultas Opcionais										
	1	**2**	**3**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**11**	**12**	**13**
SQL	4	2	4	3	2	4	4	2	2	4	3	3	3	4
Proc. Arm.	3	1	3	2	1	3	3	1	1	3	2	2	2	3
API	1	3	1	1	3	1	1	3	3	1	1	1	1	1
Cypher	2	4	2	4	4	2	2	4	4	2	4	4	4	2

Capítulo 6 – Conclusões

O objetivo desta dissertação é avaliar a utilização de bancos Relacionais e de Grafos para o armazenamento e acesso a dados de proveniência, na tentativa de descobrir a mais adequada. Estas alternativas foram escolhidas porque vários sistemas que lidam com proveniência utilizam BDR. No entanto, estes dados são naturalmente modelados como grafos direcionados acíclicos, o que faz parecer que a utilização de BDG para sua gerência seria mais adequada.

Parte da metodologia utilizada nesta dissertação consistiu em escolher consultas que seguissem a maior quantidade possível dos padrões típicos de consultas de proveniência. Para atender a esta necessidade, foram utilizadas quatorze consultas do *Third Provenance Challenge*, que se adequam à maioria destes padrões. Para avaliar o desempenho das consultas, utilizou-se diferentes abordagens das já testadas no meio acadêmico, como o uso de visões recursivas e procedimentos armazenados para um BDR (PostgreSQL), além da linguagem Cypher e da implementação de consultas em Java para um BDG (Neo4j). Além do tempo de processamento das consultas, foi analisado o consumo de recursos computacionais (processador e memória) durante sua execução. Outro aspecto avaliado foi o desempenho de cada alternativa de gerência de dados durante o processo de carga dos conjuntos de dados de teste. Neste caso, além do tempo de execução e do consumo de memória e processador, foi considerado também o espaço ocupado em disco pelas bases experimentais.

As quatro alternativas utilizadas para implementação das consultas foram capazes de responder a todas as consultas escolhidas para o experimento, o que indica que estas formas, assim como os modelos de armazenamento utilizados, suportam consultas típicas sobre dados de proveniência.

A partir das medições e análises feitas, pode-se concluir que o Neo4j exige muito mais recursos computacionais para executar uma carga ou uma consulta do que o PostgreSQL, independentemente da forma como foi implementada a consulta. Conclui-se também que, considerando o mesmo conjunto de dados de proveniência, o Neo4j necessita mais do que o dobro do espaço em disco do que o PostgreSQL para armazenamento.

Em termos de tempo de execução das consultas que envolvem travessias em grafos, conclui-se que as consultas implementadas em API Java do Neo4j e até mesmo em Cypher, tiveram um desempenho muito superior às formas implementadas para o

PostgreSQL. O uso de visões recursivas deve ser evitado, principalmente se o BDR armazenar diversas execuções de *workflow*. Em um determinado caso, o tempo médio de execução em SQL ultrapassou nove minutos, enquanto que a API Java do Neo4j obteve uma média de 28 milisegundos para a mesma consulta, ou seja, esta alternativa foi 20.079,87 vezes mais rápida que a primeira. O uso de procedimentos armazenados pode ser considerado uma alternativa aceitável, pois melhora consideravelmente o desempenho. Apesar disso, ainda fica bem abaixo do desempenho apresentado pelas consultas implementadas em API Java. Considerando o mesmo exemplo acima, o uso de procedimentos armazenados apresentou um tempo médio de execução de dois segundos.

Para as consultas mais simples, que não envolvem travessias em grafos, as alternativas que utilizam SQL, procedimentos armazenados e API Java apresentaram desempenho bem similar. No entanto, a API Java obteve o melhor desempenho na maioria dos casos, e nos casos em que perdeu foi por uma diferença substancialmente pequena (menor que 8 milisegundos). Já o Cypher apresentou o pior desempenho devido ao plano de execução implementado pelo próprio Neo4j não ser eficiente.

Por fim, pode-se observar que nenhuma solução se apresentou superior em todos os aspectos aqui considerados. Apesar disso, considerando que: os Bancos de Dados de Grafos representam os dados de proveniência em sua forma nativa; e que o desempenho das consultas que envolvem travessias foi muito superior em relação ao contexto Relacional, e, para as outras consultas, foi ligeiramente melhor, conclui-se que a forma ideal para gerenciar dados de proveniência que se destinam a responder a consultas típicas é armazená-los em um Banco de Dados de Grafos e implementar consultas via API Java (caso o SGBD seja o Neo4j). Isto, sem considerar aspectos como segurança, flexibilidade, documentação, entre outros, os quais não foram objetos de estudo desta dissertação e que podem ser requisitos importantes na escolha de um Sistema de Gerência Banco de Dados.

6.1 Contribuições

A contribuição principal desta dissertação consiste no mapeamento dos pontos fortes e fracos de cada solução estudada para gerência de dados de proveniência, comparando quatro formas diferentes de executar consultas típicas de proveniência nunca antes avaliadas em conjunto. Além disso, para o armazenamento de dados no banco de dados

de Grafos, foi desenvolvida uma modelagem que segue o OPM e um programa implementado em Java para fazer a carga respeitando esta modelagem.

6.2 Trabalhos Futuros

Como proposta para trabalhos futuros, baseando-se no fato desta pesquisa concluir que um banco de Grafos seria a melhor solução para gerência de dados de proveniência, e considerando que o Neo4j é um sistema de código livre, seria interessante avaliar a possibilidade de otimizar os aspectos onde o Neo4j apresentou um desempenho pior como, por exemplo, o consumo de memória, espaço ocupado em disco e a execução de consultas simples, onde geralmente necessita-se buscar apenas um nó considerando alguma restrição.

As consultas utilizadas nesta dissertação cobriam cinco dos oito padrões de consultas de proveniência mapeados por GADELHA *et al*. (2011). Outro trabalho futuro seria a implementação e avaliação do desempenho de consultas que atendam aos outros três padrões. Além disso, uma terceira forma de execução de consultas em bancos de Grafos poderia ser avaliada utilizando a linguagem Gremlin.

Em relação ao espaço ocupado em disco pelo Neo4j, foi verificado que cerca de 40% deste espaço foi ocupado pelos índices. Um trabalho futuro seria avaliar a possibilidade de utilizar outro serviço de indexação que não fosse o Lucene ou otimizar este serviço de forma que ocupe menos espaço sem que ocorra perda de desempenho das consultas.

As 14 consultas utilizadas não foram tratadas como consideradas *ad-hoc*. Outra proposta de um trabalho futuro seria tratá-las desta forma, pois isso influenciaria nos índices criados.

Referências Bibliográficas

ALTINTAS, I., BERKLEY, C., JAEGER, E., *et al*., 2004, "Kepler: an extensible system for design and execution of scientific workflows". In: *Scientific and Statistical Database Management*, pp. 423-424, Greece.

BADOIU, M., MUNISWAMY-REDDY, K., SIDIROPOULOS, A. *A Distributed Provenance Aware Storage System*. 2006. Disponível em: <http://pdos.csail.mit.edu/6.824-2005/reports/kiran.pdf>. Acesso em: 02 ago. 2013, 14:21:08.

BARGA, R., GANNON, D., "Scientific versus Business Workflows", In: *Workflows for e-Science*, Springer, pp. 9-16, 2007.

BERKELEY DB, 2013. Disponível em: <http://docs.oracle.com/cd/E17076_03/html/index.html>. Acesso em: 23 maio 2013, 19:24:12.

BIVAR, B., SANTOS, L., KOHWALTER, T.C., et al., "Uma Comparação entre os Modelos de Proveniência OPM e PROV". In: *CSBC 2013*, Maceió, Alagoas, Brasil, Jul. 2013.

BOSE, R., FREW, J., "Lineage retrieval for scientific data processing: a survey", *ACM Computing Surveys*, v.37, n.1, pp. 1-28, Mar. 2005.

BOWERS, S., MCPHILLIPS, T., WU, M., LUDÄSCHER, B., "Project histories: Managing provenance across collection oriented scientific workflow runs". In: *Data Integration in the Life Sciences*, *Lecture Notes in Computer Science*, Springer, pp. 122-138, 2007.

CALLAHAN, S.P., FREIRE, J., SANTOS, E., *et al.*, "VisTrails: visualization meets data management". In: *ACM SIGMOD*, pp. 745-747, Chicago, Illinois, USA, Jun. 2006.

CARVALHO, L.O.M., 2009, *Aplicação de Workflows científicos a projetos de sistemas de produção de petróleo offshore*. Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

CASSANDRA, 2013. Disponível em: <http://cassandra.apache.org/>. Acesso em: 08 ago. 2013, 17:16:45.

CHALLENGES, 2013. Disponível em: <http://twiki.ipaw.info/bin/view/Challenge/>. Acesso em: 18 ago. 2013, 17:05:23.

CHAMBERLIN, D., *Using the new DB2: IBM's object-relational database system*. San Francisco, CA, USA, AP Professional, 1996.

CHANG, F., DEAN, J., GHEMAWAT, S., *et al*., "Bigtable: A Distributed Storage System for Structured Data", In: *ACM Transactions on Computer Systems*, v. 26, n.2, 2008.

CHAPMAN, A., JAGADISH, H.V., “Issues in Building Practical Provenance Systems”. In: *IEEE Data Eng. Bull*., p. 38-43, 2007.

COSTA, F., SILVA, V., OLIVEIRA, D., *et al*., “Capturing and Querying Workflow Runtime Provenance with PROV: a Practical Approach”. In: *EDBT/ICDT 2013*, Genoa, Italy, Mar. 2013.

COUTINHO, F., OGASAWARA, E., DE OLIVEIRA, D., *et al*., “Data parallelism in bioinformatics workflows using Hydra”. In: *19th ACM International Symposium on High Performance Distributed Computing*, pp. 507–515, Chicago, Ilinois, USA, Jun. 2010.

CYPHER, 2013. Disponível em: <http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html>. Acesso em: 09 ago. 2013, 17:07:31.

DAVIDSON, S., COHEN-BOULAKIA, S., EYAL, A., *et al*., “Provenance in Scientific Workflow Systems”, In: *ACM Sigmod*, pp. 1345-1350, Vancouver, Canada, Jun. 2008.

DECANDIA, G., HASTORUN, D., JAMPANI, M., *et al*., “Dynamo: Amazon’s Highly Available Key-value Store”. In: *21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, USA, Out. 2007.

DEELMAN, E., GANNON, D., SHIELDS, M., *et al*., “Workflows and e-Science: Anoverview of workflow system features and capabilities”, *Future Generation Computer Systems*, v. 25, n. 5, p. 528-540, Maio 2009.

DEX, 2013. Disponível em: <http://www.sparsity-technologies.com/dex.php>. Acesso em: 16 ago. 2013, 18:58:12.

DYNAMODB, 2013. Disponível em: <http://aws.amazon.com/en/dynamodb/>. Acesso em: 08 ago. 2013, 19:01:14.

FILETO, R., LIU, L., PU, C., *et al*., “POESIA: An ontological workflow approach for composing Web services in agriculture”, *The VLDB Journal*, v. 12, n. 4, pp. 352–367, Nov. 2003.

FIRST CHALLENGE, 2013. Disponível em: <http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge>. Acesso em: 15 ago. 2013, 18:07:47.

FREIRE, J., KOOP, D., SANTOS, E., *et al*., “Provenance for Computational Tasks: A Survey”, *Computing in Science and Engineering*, v. 10, n. 3, pp. 11-21., Maio 2008.

FREW, J., METZGER, D., SLAUGHTER, P., “Automatic capture and reconstruction of computational provenance”, *Concurrency and Computation: Practice and Experience*, v. 20, n.5, pp. 485-496, Abr. 2008.

GADELHA, L.M.R., MATTOSO, M., WILDE, M., *et al*., "Provenance Query Patterns for Many-Task Scientific Computing". *TaPP'11*, Heraklion, Crete, Greece, Jun. 2011.

GEHANI, A., KIM, M., ZHANG, J., "Steps toward managing lineage metadata in grid clusters". *First workshop on on Theory and practice of provenance*, 7, San Francisco, CA, USA, Fev. 2009.

GEHANI, A., TARIQ, D., "SPADE: Support for Provenance Auditing in Distributed Environments". *13th International Middleware Conference*, p. 101-120, Montreal, Quebec, Canada, Dez. 2012.

GIL, Y., DELLMAN, E., ELLISMAN, M., *et al.*, "Examining the Challenges of Scientific Workflows", *IEEE Computer*, vol. 40, n. 12, pp. 24-32, 2007.

GIL, Y., MILES, S. PROV, 2013. Disponível em: <http://www.w3.org/TR/prov-primer/>. Acesso em: 23 out. 2013, 15:01:10.

GREMLIN, 2013. Disponível em: <https://github.com/tinkerpop/gremlin/wiki>. Acesso em: 19 ago. 2013, 17:27:12.

GROUP, P.W., 2005, *Data Dictionary for Preservation Metadata*, Ohio, USA, OCLC e RGL.

HOLLAND, D.A., BRAUN, U., MACLEAN, D., *et al*., "Choosing a Data Model and Query Language for Provenance". *Proceedings of the 2nd International Provenance and Annotation Workshop*, Salt Lake City, Utah, USA, Jun. 2008.

HULL, D., WOLSTENCROFT, K., STEVENS, R., *et al*., "Taverna: a tool for building and running workflows of services", *Nucleic Acids Research*, v. 34, n. 2, pp. 729-732, Jul. 2006.

HYPERGRAPHDB, 2013. Disponível em: <http://www.hypergraphdb.org/index>. Acesso em: 16 ago. 2013, 18:58:32.

IPAW, 2013. Disponível em: <http://www.ipaw.info/>. Acesso em: 22 ago. 2013, 14:21:32.

INFINITEGRAPH, 2013. Disponível em: <http://www.objectivity.com/infinitegraph>. Acesso em: 16 ago. 2013, 18:59:22.

JATANA, N., PURI, S., AHUJA, M., *et al*., "A Survey and Comparison of Relational and Non-Relational Database", *International Journal of Engineering Research & Technology*, v. 1, n. 6, Ago. 2012.

KARMA, 2013. Disponível em: <http://sourceforge.net/projects/karmatool/>. Acesso em: 22 out. 2013, 18:59:22.

LAKSHMAN, A., MALIK, P., "Cassandra - A Decentralized Structured Storage System", In: *ACM SIGOPS Operating Systems Review*, v.44, n. 2, pp. 35-40, Abr. 2010.

LFS, 2013. Disponível em: <http://crypto.stanford.edu/~cao/lineage.html>. Acesso em: 22 maio 2013, 18:31:10.

LIM, C., LU, S., CHEBOTKO, A., *et al*., "Storing, reasoning, and querying OPM compliant scientific workflow provenance using relational databases", *Journal of Future Generation Computer Systems*, v.27, n.6, pp. 781-789, Jun. 2011.

MALIK, T., NISTOR, L., GEHANI, A., "Tracking and Sketching Distributed Data Provenance", *IEEE Sixth International Conference on ESCIENCE*, pp. 190-197, Brisbane, Queensland, Australia, Dez. 2010.

MARINHO, A., MURTA, L., WERNER, C., *et al*., "A Strategy for Provenance Gathering in Distributed Scientific Workflows", *2009 World Conference on Services*, pp. 344-347, Los Angeles, California, Jul. 2009.

MARINHO, A., MURTA, L., WERNER, C., *et al*., "Managing Provenance in Scientific Workflows with ProvManager", *International Workshop on Challenges in e-Science - SBAC*, pp. 17-24, Petrópolis, Rio de Janeiro, Brasil, Out. 2010a.

MARINHO, A., MURTA, L., WERNER, C., *et al*., "Integrating Provenance Data from Distributed Workflow Systems with ProvManager", *Provenance and Annotation of Data and Processes*, pp. 286-288, Troy, New York, Jun. 2010b.

MARINHO, A., MURTA, L., WERNER, C., *et al*., "ProvManager: a provenance management system for scientific workflows", *Concurrency and Computation: Practice & Experience*, v. 24, n. 13, pp. 1513-1530, Set. 2013.

MARTINHO, W., OGASAWARA, E., OLIVEIRA, D., *et al*., "A Conception Process for Abstract Workflows: An Example on Deep Water Oil Exploitation Domain". *5th IEEE International Conference on e-Science*, Oxford, UK, 2009.

MCBRIDE, B, 2002, "Jena: A semantic web toolkit", *Journal of IEEE Internet Computing*, v. 6, n. 6, pp. 55-59, Nov. 2002.

MOREAU, L. *et al*., "The First Provenance Challenge". In: *Concurrency and Computation: Practice and Experience*, v. 20, n.5, Wiley, pp. 409-418, 2008.

MOREAU, L., CLIFFORD, B., FREIRE, J., *et al*., "The Open Provenance Model Core Specification (v1.1)", *Future Generation Computer Systems*, v. 27, n. 6, pp. 743-756, Jul. 2010.

MOREAU, L., MISSIER, P. *PROV-DM*, 2013. Disponível em: <http://www.w3.org/TR/prov-dm/>. Acesso em: 26 abr. 2013, 15:04:30.

MUNISWAMY-REDDY, K., HOLLAND, D.A., BRAUN, U., *et al*., "Provenance-Aware Storage Systems". *USENIX Annual Technical Conference*, Boston, MA, Jun. 2006.

MYSQL, 2013. Disponível em: <http://www.mysql.com/>. Acesso em: 21 ago. 2013, 18:24:43.

NARDI, A.R., 2009, *Uma arquitetura de baixo acoplamento para execução de padrões de controle de fluxo em grades*, Tese D.Sc., Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo, Brasil.

NÉMETH, Z., "Workflow enactment based on a chemical metaphor". *3rd IEEE International Conference on Software Engineering and Formal Methods*, pp. 127-136, Koblenz, Germany, Set. 2005.

NEO4J, 2013. Disponível em: <http://neo4j.org>. Acesso em: 8 ago. 2013, 19:21:56.

OCAÑA, K.A.C.S., OLIVEIRA, D., DIAS, D., *et al*., "Designing a parallel cloud based comparative genomics workflow to improve phylogenetic analyses", *Future Generation Computer Systems*, Abr. 2013.

OGASAWARA, E., DIAS, J., OLIVEIRA, D., *et al*., 2011, "An Algebraic Approach for Data-Centric Scientific Workflows", *Proceedings of the VLDB Endowment*, v. 4, n. 12, p. 1328-1339, Seattle, WA, Ago. 2011.

OGASAWARA, E.S., 2011, *Uma Abordagem Algébrica para Workflows Científicos com Dados em Larga Escala*, Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

OGASAWARA, E., DIAS, J., SILVA, V., *et al*., "Chiron: a parallel engine for algebraic scientific workflows", *Concurrency and Computation: Practice and Experience*, 10.1002/cpe.3032, Mai. 2013.

OLIVEIRA, D., CUNHA, L., TOMAZ, L., *et al*., "Using Ontologies to Support Deep Water Oil Exploration Scientific Workflows". *Services I*, pp. 364-367, Los Angeles, CA, United States, Jul 2009.

OLIVEIRA, D., OGASAWARA, E., BAIÃO, F., "SciCumulus: A Lightweight Cloud Middleware to Explore Many Task Computing Paradigm in Scientific Workflows". In: *3rd International Conference on Cloud Computing*, pp. 378-385, Washington, DC, USA, 2010.

OLIVEIRA, D., OCAÑA, K.A.C.S., BAIÃO, F., *et al*., "A Provenance-based Adaptive Scheduling Heuristic for Parallel Scientific Workflows in Clouds", *Journal of Grid Computing*, v. 10, n. 3, pp. 521-552, Set. 2012.

OLIVEIRA, D., OCAÑA, K.A.C.S., OGASAWARA, E., *et al*., "Performance evaluation of parallel strategies in public clouds: A study with phylogenomic workflows", *Future Generation Computer Systems*, v. 29, n. 7, pp. 1816-1825, Set. 2013.

OPM, 2007. Disponível em: <http://openprovenance.org/>. Acesso em: 26 abr. 2013, 15:27:47.

ORACLE, 2013. Disponível em: <http://www.oracle.com/>. Acesso em: 4 maio 2013, 15:59:35.

ORIENTDB, 2013. Disponível em: <http://www.orientdb.org/>. Acesso em: 19 ago. 2013, 18:50:10.

PAN-STARRS, 2013. Disponível em: <http://pan-starrs.ifa.hawaii.edu/public/home.html>. Acesso em: 5 set. 2013, 16:07:32.

PARTICIPATING TEAMS, 2013. Disponível em: <http://twiki.ipaw.info/bin/view/Challenge/ParticipatingTeams3>. Acesso em: 26 ago. 2013, 16:46:23.

POSTGRESQL, 2013. Disponível em: <http://www.postgresql.org/>. Acesso em: 4 maio 2013, 15:49:13.

PROTOPROV, 2013. Disponível em: <https://code.google.com/p/tetherlesspc3/source/browse/trunk/src/edu/rpi/tw/provenance/protoprov/?r=24>. Acesso em: 22 out. 2013, 18:35:33.

REXTER, 2013. Disponível em: <https://github.com/tinkerpop/rexster/wiki>. Acesso em 19 ago. 2013, 18:52:43.

ROBINSON, I., WEBBER, J., EIFREM, E., 2013, *Graph Databases*, 1 ed. California, USA, O'Reilly.

RODRIGUEZ, M.A., NEUBAUER, P., "The Graph Traversal Pattern", *Graph Data Management: Techniques and Applications*, eds. S. Sakr, E. Pardede, 2011.

SADALAGE, P.J., FOWLER, M., 2013, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 2 ed. New Jersey, USA, Pearson.

SIGAR, 2013. Disponível em: <https://support.hyperic.com/display/SIGAR/Home>. Acesso em: 26 ago. 2013, 17:13:31.

SIMMHAN, Y., PLALE, B., GANNON, D., "A survey of data provenance in e-science", *SIGMOD Record*, v. 34, n. 2, pp.31-36. Set. 2005.

SOURCESOURCE, 2013. Disponível em: <https://github.com/drsimonmiles/sourcesource>. Acesso em: 22 out. 2013, 18:23:13.

SPARQL, 2013a. Disponível em: <http://www.w3.org/TR/rdf-sparql-query/>. Acesso em: 16 ago. 2013, 16:25:22.

SPARQL, 2013b. Disponível em: <http://www.w3.org/TR/sparql11-overview/>. Acesso em: 16 ago. 2013, 16:27:42.

SQLSERVER, 2013. Disponível em: <http://www.microsoft.com/en-us/sqlserver/default.aspx>. Acesso em: 4 maio 2013, 16:02:10.

TAYLOR, I.J., DEELMAN, E., GANNON, D.B., *et al*., *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007.

THIRD CHALLENGE, 2013. Disponível em: <http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge>. Acesso em: 8 ago. 2013, 19:28:10.

TINKERGRAPH, 2013. Disponível em: <https://github.com/tinkerpop/blueprints/wiki/TinkerGraph>. Acesso em: 19 ago. 2013, 18:49:10.

TUPELO, 2013. Disponível em: <http://tupeloproject.ncsa.uiuc.edu/node/2>. Acesso em: 22 out. 2013, 18:29:19.

VICKNAIR, C., MACIAS, M., ZHAO, Z., *et al*., "A Comparison of a Graph Database and a Relational Database". *48th Annual Southeast Regional Conference*, Mississipi, USA, Abr. 2010.

W3C, 2013. Disponível em: <http://www.w3.org/>. Acesso em: 21 out. 2013, 19:08:33.

WASSINK, I., RAUWERDA, H., VET, P., *et al*., "E-BioFlow: Different Perspectives on Scientific Workflows". *Second International Conference, BIRD*, pp. 243-257, Vienna, Austria, Jul. 2008.

WOODMAN, S., HIDEN, H., WATSON, P., *et al*., "Achieving reproducibility by combining provenance with service and workflow versioning". *Proceedings of the 6th workshop on Workflows in support of large-scale science*, pp. 127-136, Seattle, WA, Nov. 2011.

XPATH, 2013. Disponível em: <http://www.w3schools.com/xpath/>. Acesso em: 15 ago. 2013, 18:05:33.

XQUERY, 2013. Disponível em: <http://www.w3schools.com/xquery/>. Acesso em: 15 ago. 2013, 18:06:01.

ZHAO, Y., HATEGAN, M., CLIFFORD, B., *et al*., "Swift: Fast, Reliable, Loosely Coupled Parallel Computation". *IEEE Congress on Services*, pp. 199-206, Salt Lake City, UT, Jul. 2007.

ZHAO, J., SIMMHAN, Y., GOMADAM, K., *et al*., "Querying Provenance Information in Distributed Environments", *Journal of I. J. Comput. Appl*., v. 18, n. 3, pp. 196-215, 2011.

APÊNDICE A

A.1 Implementação da CQ1

SQL:

```
select distinct a2. Value from Artifact a2, Used u,
(select distinct tg. OPMGraphId , tg. ProcessId
from MultiStepWasGeneratedBy tg, Artifact a
where tg. OPMGraphId =
a. OPMGraphId and tg. ArtifactId = a. ArtifactId and a. Value like $pc1 )
as pv where u. OPMGraphId = pv. OPMGraphId
and u. ProcessId = pv. ProcessId and u. OPMGraphId = a2. OPMGraphId
and u. ArtifactId = a2. ArtifactId and u. OPMGraphId = $idSuc and a2. Value like
$pc12
union all
select distinct a2. Value from Artifact a2, Used u
where u. OPMGraphId = a2. OPMGraphId
and u. ArtifactId = a2. ArtifactId and u. OPMGraphId = $idSuc and a2. Value like
$pc12;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_cq1_result(idgraph integer, artifactvalue text, parameter2 text)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in select distinct * from (
                select * from (
                select artifactid from artifact where opmgraphid = idgraph and value like artifactValue
        loop
                for r in select distinct a2. Value from Artifact a2, Used u,
                        (select distinct tg. OPMGraphId , tg. ProcessId
                        from stp_MultiStepWasGeneratedBy(idgraph, x) tg, Artifact a
                        where tg. OPMGraphId =
                        a. OPMGraphId and tg. ArtifactId = a. ArtifactId
                        and a. Value like artifactValue )
                        as pv where u. OPMGraphId = pv. OPMGraphId
                        and u. ProcessId = pv. ProcessId and u. OPMGraphId = a2. OPMGraphId
                        and u. ArtifactId = a2. ArtifactId and u. OPMGraphId = idgraph
                        and a2. Value like
                        parameter2)
                        union all
                        select distinct a2. Value from Artifact a2, Used u
                        where u. OPMGraphId = a2. OPMGraphId
                        and u. ArtifactId = a2. ArtifactId and u. OPMGraphId = idgraph
                        and a2. Value like
                        parameter2 ) t
                        group by 1
                loop

                        return next r;
                end loop;
        end loop;
end;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_cq1_result(integer, text, text)
  OWNER TO postgres;
```

API Java:

```
IndexHits<Node> detections = idx.query("value:"
                + detectionArtifactValue + " AND graphname:" + graphName);

for (Node detection : detections)
        if (detection != null) {
                AccountPredicate validAccounts = new AccountPredicate(
                                (String[]) detection.getProperty("accounts",
                                                new String[0]));
                for (Relationship r : detection.getRelationships(
                                RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                        if (validAccounts.accept(r))
                                processesToCheck.addLast(r.getEndNode());
                }
                for (Relationship r : detection.getRelationships(
                                RelTypes.WAS_DERIVED_FROM, Direction.OUTGOING)) {
                        if (validAccounts.accept(r))
                                artifactsToCheck.addLast(r.getEndNode());
                }
                while (!artifactsToCheck.isEmpty()
                                || !processesToCheck.isEmpty()) {
                        if (!artifactsToCheck.isEmpty()) {
                                Node artifact = artifactsToCheck.removeFirst();
                                if (!artifactsChecked.contains(artifact)) {
                                        for (Relationship r : artifact.getRelationships(
                                                                        RelTypes.WAS_GENERATED_BY,
                                                                        Direction.OUTGOING)) {
                                                if (validAccounts.accept(r)) {
                                                        Node generatorProcess = r.getEndNode();
                                                        if (!processesChecked
                                                                        .contains(generatorProcess)
                                                                        && !processesToCheck.contains(generatorProcess))
                                                                processesToCheck.addLast(generatorProcess);
                                                }
                                        }
                                        for (Relationship r : artifact
                                                        .getRelationships(
                                                                        RelTypes.WAS_DERIVED_FROM,
                                                                        Direction.OUTGOING)) {
                                                if (validAccounts.accept(r)) {
                                                        Node deriverArtifact = r.getEndNode();
                                                        if (!artifactsChecked.contains(deriverArtifact)
                                                                        && !artifactsToCheck.contains(deriverArtifact))
                                                                artifactsToCheck.addLast(deriverArtifact);
                                                }
                                        }
                                        artifactsChecked.add(artifact);
                                }
                        }
                        if (!processesToCheck.isEmpty()) {
                                Node process = processesToCheck.removeFirst();
                                if (!processesChecked.contains(process)) {
                                        for (Relationship r : process.getRelationships(
                                                        RelTypes.USED, Direction.OUTGOING)) {
                                                if (validAccounts.accept(r)) {
                                                        Node usedArtifact = r.getEndNode();
                                                        String artifactValue = (String) usedArtifact
                                                                        .getProperty("value", null);
                                                        if (artifactValue != null)
                                                                if (artifactValue.contains(detection2)
                                                                                && !fileNames.contains(artifactValue))
                                                                        fileNames.add(artifactValue);
                                                }
                                        }
                                        processesChecked.add(process);
                                }
                        }
                }
        }
```

Cypher:

```
START n=node: artifactsAtt(value: $detectionArtifactValue AND graphname: $graphName)
MATCH (n)-[?:WAS_DERIVED_FROM*1..9999999]->(a)
WITH n, a
MATCH (n)-[:WAS_GENERATED_BY]->(p)-[:USED]->(x)<-[:USED]-()<-[:WAS_GENERATED_BY]-(a)
where x.value =~ '.*$detection2.*'
return distinct x.value
```

A.2 Implementação da CQ2

SQL:

```
select 'YES' as answer
from (select count(g.artifactid) as number from wasgeneratedby g, process p
where p.value like $pc2 and g.opmgraphid =  $idSuc
and p.processid = g.processid and p.opmgraphid = g.opmgraphid)
as output where output.number > 0;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_cq2(idgraph integer, pvalue text)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select 'YES' as answer
                from (select count(g.artifactid) as number from wasgeneratedby g, process p
                where p.value like pValue and g.opmgraphid = idgraph
                and p.processid = g.processid and p.opmgraphid = g.opmgraphid)
                as output where output.number > 0
         loop
                return next r;
         end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_cq2(integer, text)
  OWNER TO postgres;
```

API Java:

```
for(Node process: idx.query("value:" + processValue + " AND graphname:" + graphName))
{
            if (process != null ) {

                    for(Relationship r : process.getRelationships(RelTypes.WAS_GENERATED_BY,
                        Direction.INCOMING)) {
                                count++;
                    }

        }
}
```

Cypher:

```
START n=node:processesAtt(value: $processValue AND graphname: $graphName)
MATCH (n)<-[:WAS_GENERATED_BY]-(x)
return case when count(x.value)>0
then 'YES' else 'NO'
```

A.3 Implementação da CQ3

SQL:

```
select cp.value as operation, sum(cp.number) as count
from (select tt.opmgraphid, tt.causeprocessid, p.value, 1 as number
from  multistepwastriggeredby tt, artifact a, process p, wasgeneratedby g
where tt.opmgraphid = g.opmgraphid and g.opmgraphid = a.opmgraphid
and tt.effectprocessid =g.processid
and g.artifactid = a.artifactid and p.opmgraphid =  $idSuc
and a.value like $pc3 and tt.opmgraphid = p.opmgraphid and tt.causeprocessid = p.processid
and p.value not like 'Is%'
and p.value not like 'Update%'
and p.value not like 'Compact%'
and p.value not like 'DirectAssertion%'
and p.value not like 'ForEach%'
union all
select p.opmgraphid, p.processid, p.value, 1 as number
from wasgeneratedby w, artifact a, process p
where w.artifactid = a.artifactid
and a.value like $pc3
and p.processid = w.processid and p.opmgraphid =  $idSuc
and p.opmgraphid = w.opmgraphid and a.opmgraphid = w.opmgraphid
) as cp where cp.opmgraphid =  $idSuc group by cp.value;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_cq3(idgraph integer, artifactvalue text)
RETURNS SETOF oq8_result AS
$BODY$
declare
r oq8_result%rowtype;
begin
        for r in
                select cp.value as operation, sum(cp.number) as count
                from (select tt.opmgraphid, tt.causeprocessid, p.value, 1 as number
                from  (select distinct * from stp_multistepwastriggeredbyeffect2(idgraph)) tt,
                artifact a, process p, wasgeneratedby g
                where tt.opmgraphid = g.opmgraphid and g.opmgraphid = a.opmgraphid
                and tt.effectprocessid =g.processid
                and g.artifactid = a.artifactid and p.opmgraphid =  idgraph  and
                a.value like artifactvalue and tt.opmgraphid = p.opmgraphid
                and tt.causeprocessid = p.processid
                and p.value not like 'Is%'
                and p.value not like 'Update%'
                and p.value not like 'Compact%'
                and p.value not like 'DirectAssertion%'
                and p.value not like 'ForEach%'
                union all
                select p.opmgraphid, p.processid, p.value, 1 as number
                from wasgeneratedby w, artifact a, process p
                where w.artifactid = a.artifactid
                and a.value like artifactvalue
                and p.processid = w.processid and p.opmgraphid =  idgraph
                and p.opmgraphid = w.opmgraphid
                and a.opmgraphid = w.opmgraphid
                ) as cp where cp.opmgraphid =  idgraph  group by cp.value
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_cq3 (integer, text)
  OWNER TO postgres;
```

API Java:

```
StringCounter processNames = new StringCounter();
IndexHits<Node> images = idx.query("value:" + imageArtifactValue
                + " AND graphname:" + graphName);
for (Node image : images) {
        AccountPredicate validAccounts = new AccountPredicate(
                        (String[]) image.getProperty("accounts", new String[0]));
        for (Relationship wgb : image.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                if (validAccounts.accept(wgb)) {
                        Node generatorProcess = wgb.getEndNode();
                        String value = (String) generatorProcess.getProperty(
                                        "value", "");
                        if (validProcessValue(value)) {
                                processNames.add(value);

                        }
                        for (Node process : OPM_Inferences
                                        .multiStepWasTriggeredBy(generatorProcess)) {
                                value = (String) process.getProperty("value", "");
                                if (validProcessValue(value)) {
                                        processNames.add(value);

                                }
                        }
                }
        }
}
```

Cypher:

```
START n=node: artifactsAtt(value: $imageArtifactValue AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->(y)
where y.value =~ '[^Is].*' and y.value =~ '[^Update].*' and y.value =~ '[^Compact].*'
and y.value =~ '[^DirectAssertion].*' and y.value =~ '[^ForEach].*'
WITH y
MATCH (a)<-[?:USED]-(y)-[?:WAS_TRIGGERED_BY]->(z)
where z.value =~ '[^Is].*' and z.value =~ '[^Update].*' and z.value =~ '[^Compact].*'
and z.value =~ '[^DirectAssertion].*' and z.value =~ '[^ForEach].*'
WITH y,z,a
MATCH (w)<-[?:WAS_GENERATED_BY]-()<-[?:WAS_DERIVED_FROM*1..999999]-(a)
where w.value =~ '[^Is].*' and w.value =~ '[^Update].*' and w.value =~ '[^Compact].*'
and w.value =~ '[^DirectAssertion].*' and w.value =~ '[^ForEach].*'
return distinct y.value,w.value, z.value
```

A.4 Implementação da OQ1

SQL:

```
select count(*) as Count
from artifact a, wasgeneratedby g, process p
where a.opmgraphid = g.opmgraphid and a.artifactid = g.artifactid and p.value like $po11
and p.processid = g.processid and p.opmgraphid = g.opmgraphid
and a.value like $po12
and a.opmgraphid =  $idHalt;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq1(idgraph integer, pvalue text, avalue text)
RETURNS SETOF integer AS
$BODY$
declare
r text;
begin
        for r in
                select count(*) as Count
                from artifact a, wasgeneratedby g, process p
                where a.opmgraphid = g.opmgraphid and a.artifactid = g.artifactid
                and p.value like pvalue
                and p.processid = g.processid and p.opmgraphid = g.opmgraphid
                and a.value like avalue
                and a.opmgraphid = idgraph
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq1(integer, text, text)
  OWNER TO postgres;
```

API Java:

```
for (Node process : idx.query("value:" + processValue
                + " AND graphname:" + graphName)) {
        for (Relationship r : process.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.INCOMING)) {
                Node a = r.getStartNode();
                if (a.getProperty("value").toString()
                                .contains(artifactValue))
                        count++;
        }
}
```

Cypher:

```
START n=node: processesAtt(value: $processValue AND graphname: $graphName)
MATCH (n)<-[:WAS_GENERATED_BY]-(y)
where y.value =~ '.*$artifactValue.*'
return count(*)
```

A.5 Implementação da OQ3

SQL:

```
select t2.otimeupper as startTime, t1.otimeupper as endtime from
(select w1.otimeupper from artifact a1, wasgeneratedby w1, process p
where a1.opmgraphid = w1.opmgraphid and A1.artifactid = w1.artifactid and p.value like $po31
and p.processid = w1.processid and p.opmgraphid = w1.opmgraphid
and a1.value like $po33 and w1.opmgraphid =  $idHalt ) as t1,
(select w2.otimeupper from artifact a2, wasgeneratedby w2, process p
where a2.opmgraphid = w2.opmgraphid and A2.artifactid = w2.artifactid and p.value like $po32
and p.processid = w2.processid and p.opmgraphid = w2.opmgraphid
and a2.value like $po34+ and w2.opmgraphid =  $idHalt  ) as t2;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq3(idgraph integer, processidexists text,
processidmatch text, artifactvaluefalse text, artifactvaluetrue text)
RETURNS SETOF oq3result AS
$BODY$
declare
r oq3result;
begin
        for r in
                select t2.otimeupper as startTime, t1.otimeupper as endtime from
                (select w1.otimeupper from artifact a1, wasgeneratedby w1, process p
                where a1.opmgraphid = w1.opmgraphid and A1.artifactid = w1.artifactid
                and p.value like processidexists
                and p.processid = w1.processid and p.opmgraphid = w1.opmgraphid
                and a1.value like artifactvalueFalse and w1.opmgraphid = idgraph) as t1,
                (select w2.otimeupper from artifact a2, wasgeneratedby w2, process p
                where a2.opmgraphid = w2.opmgraphid and A2.artifactid = w2.artifactid
                and p.value like processidmatch
                and p.processid = w2.processid and p.opmgraphid = w2.opmgraphid
                and a2.value like artifactvalueTrue and w2.opmgraphid = idgraph) as t2
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq3(integer, text, text, text, text)
  OWNER TO postgres;
```

API Java:

```
Node p1 = idx.query(
                "value:" + successProcessValue + " AND graphname:"
                                + graphName).getSingle();
if (p1 != null) {
        for (Relationship r1 : p1.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.INCOMING)) {

                if (r1.getStartNode().getProperty("value").toString()
                                .contains((successArtifactValue))) {
                        for (String prop : r1.getPropertyKeys()) {
                                if (prop.equals("time.noLaterThan")) {
                                        startTime = r1.getProperty("time.noLaterThan")
                                                        .toString();

                                }
                                break;
                        }
                }
                break;
        }
}

for (Node p2 : idx.query("value:" + haltProcessValue
                + " AND graphname:" + graphName)) {
        for (Relationship r2 : p2.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.INCOMING)) {
                if (r2.getStartNode().getProperty("value").toString()
                                .contains((haltArtifactValue))) {
                        for (String prop : r2.getPropertyKeys()) {
                                if (prop.equals("time.noLaterThan")) {
                                        times.add(new aprop(startTime, r2.getProperty(
                                                        "time.noLaterThan").toString()));
                                }
                                break;
                        }
                }
        }
}
```

Cypher:

```
START n=node: processesAtt(value: $successProcessValue AND graphname: $graphName)
MATCH (n)<-[:WAS_GENERATED_BY]-(y)
where y.value =~ '.*$successArtifactValue.*'
return y.time
START n=node: processesAtt(value: $haltProcessValue AND graphname: $graphName)
MATCH (n)<-[:WAS_GENERATED_BY]-(y)
where y.value =~ '.*$haltArtifactValue.*'
return y.time
```

A.6 Implementação da OQ4

SQL:

```
select pv.value, sum(pv.number) as count
from (select tt.opmgraphid, tt.causeprocessid, p.value, 1 as number
from multistepwastriggeredby tt, artifact a, wasgeneratedby g, process p
where tt.opmgraphid = g.opmgraphid and tt.effectprocessid
= g.processid and tt.opmgraphid = p.opmgraphid and tt.causeprocessid = p.processid
and a.opmgraphid = g.opmgraphid and a.artifactid =
g.artifactid and a.value like $po4+ and p.opmgraphid =  $idSuc
union all
select p.opmgraphid, p.processid, p.value, 1 as number from wasgeneratedby w, artifact a, process p
where w.artifactid = a.artifactid and a.value like $po4+
and p.processid = w.processid and p.opmgraphid =   $idSuc and p.opmgraphid = w.opmgraphid
and a.opmgraphid = w.opmgraphid
) as  pv group by pv.value;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq4(idgraph integer, artifactvalue text)
RETURNS SETOF oq8_result AS
$BODY$
declare
r oq8_result%rowtype;
begin
        for r in select pv.value, sum(pv.number) as count
                from (select tt.opmgraphid, tt.causeprocessid, p.value, 1 as number
                from (select distinct * from stp_multistepwastriggeredbyeffect2(idgraph)) tt,
                 artifact a, wasgeneratedby g, process p where tt.opmgraphid = g.opmgraphid
                 and tt.effectprocessid
                = g.processid and tt.opmgraphid = p.opmgraphid and tt.causeprocessid = p.processid
                and a.opmgraphid = g.opmgraphid and a.artifactid =
                g.artifactid and a.value like artifactvalue and p.opmgraphid =  idgraph
                union all
                select p.opmgraphid, p.processid, p.value, 1 as number from wasgeneratedby w,
                artifact a, process p where w.artifactid = a.artifactid and a.value like artifactvalue
                and p.processid = w.processid and p.opmgraphid =   idgraph
                and p.opmgraphid = w.opmgraphid and a.opmgraphid = w.opmgraphid
                ) as  pv group by pv.value
        loop
                return next r;
        end loop;

end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq4 (integer, text)
  OWNER TO postgres;
```

API Java:

```
IndexHits<Node> entries = idx.query("value:" + dbEntryValue + " AND graphname:" + graphName);
for (Node entry : entries) {

        AccountPredicate validAccounts = new AccountPredicate((String[])entry.
                        getProperty("accounts", new String[0]));
        for(Relationship wgb : entry.getRelationships(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                if (validAccounts.accept(wgb)) {
                        Node generatorProcess = wgb.getEndNode();
                        processNames.add((String)generatorProcess.getProperty("value", ""));
                        for (Node process : OPM_Inferences.multiStepWasTriggeredBy(generatorProcess)) {
                                processNames.add((String)process.getProperty("value", ""));
                        }
                }
        }
}
```

Cypher:

```
START n=node: artifactsAtt(value: $dbEntryValue AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->(y)
WITH y
MATCH (a)<-[?:USED]-(y)-[?:WAS_TRIGGERED_BY]->(z)
WITH y,z,a
MATCH (w)<-[?:WAS_GENERATED_BY]-()<-[?:WAS_DERIVED_FROM*1..999999]-(a)
return distinct y.value,w.value, z.value
```

A.7 Implementação da OQ5

SQL:

```
select 'opmgraphid: '||a.opmgraphid|| ' nameOfDatabase: ' ||a.value as nameOfDatabase
from artifact a, wasgeneratedby w
where a.opmgraphid = w.opmgraphid and a.artifactid = w.artifactid and a.value like $error;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq5(artifactvalue text)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select 'opmgraphid: '||a.opmgraphid|| ' nameOfDatabase: ' ||a.value
                as nameOfDatabase
                from artifact a, wasgeneratedby w
                where a.opmgraphid = w.opmgraphid and a.artifactid = w.artifactid
                and a.value like artifactvalue
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq5(text)
  OWNER TO postgres;
```

API Java:

```
for(Node artifact: idx.query("value:" +haltValue))
{
        if(artifact.hasRelationship(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING) )
        {

                artifacts.add(new aprop(artifact.getProperty("graphName").toString()
                                , artifact.getProperty("value").toString()));

        }
}
```

Cypher:

```
START n=node: artifactsAtt (value: $haltValue)
MATCH (n)-[:WAS_GENERATED_BY]->(x)
return n.graphName, n.value
```

A.8 Implementação da OQ6

SQL:

```
select hp.value as HaltStep, sum(hp.number) as count
from (select distinct p.opmgraphid, p.processid, p.value, 1 as number
from artifact a, wasgeneratedby g, process p
where a.value like $error and a.opmgraphid = g.opmgraphid and
a.artifactid = g.artifactid and g.opmgraphid = p.opmgraphid and g.processid = p.processid
and p.opmgraphid =   $idHalt ) as hp
group by hp.value;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq6(idgraph integer, artifactvalue text)
RETURNS SETOF oq8_result AS
$BODY$
declare
r oq8_result%rowtype;
begin
        for r in
                select hp.value as HaltStep, sum(hp.number) as count
                from (select distinct p.opmgraphid, p.processid, p.value, 1 as number
                from artifact a, wasgeneratedby g, process p
                where a.value like artifactvalue and a.opmgraphid = g.opmgraphid and
                a.artifactid = g.artifactid and g.opmgraphid = p.opmgraphid
                and g.processid = p.processid
                ) as hp where hp.opmgraphid = idgraph
                group by hp.value
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq6(integer, text)
  OWNER TO postgres;
```

API Java:

```
IndexHits<Node> artifacts = idx.query("value:" + haltValue
                + " AND graphname:" + graphName);
for (Node a : artifacts) {
        for (Relationship wgb : a.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                Node generatorProcess = wgb.getEndNode();
                String value = "";
                if (generatorProcess.hasProperty("value"))
                        value = (String) generatorProcess.getProperty("value");
                if (!prs.contains((String) generatorProcess
                                .getProperty("id") + value)) {
                        processNames.add(value);
                        prs.add((String) generatorProcess.getProperty("id")
                                        + value);
                }
        }
}
```

Cypher:

```
START n=node: artifactsAtt(value: $haltValue AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->(y)
return distinct n.id, count(n.value)
```

A.9 Implementação da OQ7

SQL:

```
select distinct 'artifactid: '||a2.artifactid ||' value: ' ||a2.value
from artifact a1, wasgeneratedby g, artifact a2, used u
where a1.value like $error
and a1.opmgraphid = g.opmgraphid and a1.artifactid = g.artifactid
and u.opmgraphid = g.opmgraphid and u.processid = g.processid and a2.opmgraphid = u.opmgraphid
and a2.artifactid = u.artifactid and a2.opmgraphid = $idHalt;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq7(idgraph integer, artifactvaluehalt text)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select distinct 'artifactid: '||a2.artifactid ||' value: ' ||a2.value
                from artifact a1, wasgeneratedby g, artifact a2, used u
                where a1.value like artifactValueHalt
                and a1.opmgraphid = g.opmgraphid and a1.artifactid = g.artifactid
                and u.opmgraphid = g.opmgraphid and u.processid = g.processid
                and a2.opmgraphid = u.opmgraphid
                and a2.artifactid = u.artifactid and a2.opmgraphid = idgraph
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq7(integer, text)
  OWNER TO postgres;
```

API Java:

```
for(Node artifact: idx.query("value:" + haltArtifact + " AND graphname:" + graphName))
{
        if (artifact != null) {
                for(Relationship r: artifact.getRelationships(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING))
                {
                        Node n = r.getEndNode();
                        for(Relationship rr: n.getRelationships(RelTypes.USED, Direction.OUTGOING))
                        {
                                Node nn = rr.getEndNode();
                                String value = "";
                                try
                                {
                                        value = nn.getProperty("value").toString();
                                }
                                catch(NotFoundException e)
                                {

                                        value = "";
                                }
                                artifacts.put(nn.getProperty("id").toString(),
                                        new pprop(nn.getProperty("id").toString(),value));
                        }

                 }
        }

}
```

Cypher:

```
START n=node: artifactsAtt(value: $haltArtifact AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->()-[:USED]->(y)
return distinct y.id,y.value
```

A.10 Implementação da OQ8

SQL:

```
select p.value,  count(*)
from multistepwastriggeredby w, process p, wasgeneratedby ww, artifact a
where w.effectprocessid = ww.processid and ww.artifactid = a.artifactid and a.value like $error
and w.causeprocessid = p.processid and p.opmgraphid =  $idHalt
and p.opmgraphid = w.opmgraphid and a.opmgraphid = ww.opmgraphid
and w.opmgraphid = ww.opmgraphid
group by p.value
union
select p.value, 1 from wasgeneratedby w, artifact a, process p
where w.artifactid = a.artifactid and a.value like $error
and p.processid = w.processid and p.opmgraphid =   $idHalt
and p.opmgraphid = w.opmgraphid and a.opmgraphid = w.opmgraphid;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq8(idgraph integer, artifactvalue text)
RETURNS SETOF oq8_result AS
$BODY$
declare
r oq8_result%rowtype;
begin
        for r in
                select p.value,  count(*) from stp_multistepwastriggeredbyeffect2(idgraph) w,
                 process p, wasgeneratedby ww, artifact a
                where w.effectprocessid = ww.processid and ww.artifactid = a.artifactid
                and a.value like artifactvalue
                and w.causeprocessid = p.processid and p.opmgraphid = idgraph
                and p.opmgraphid = w.opmgraphid and a.opmgraphid = ww.opmgraphid
                and w.opmgraphid = ww.opmgraphid
                group by p.value
                union
                select p.value, 1 from wasgeneratedby w, artifact a, process p
                where w.artifactid = a.artifactid and a.value like artifactvalue
                and p.processid = w.processid and p.opmgraphid =   idgraph
                and p.opmgraphid = w.opmgraphid and a.opmgraphid = w.opmgraphid
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq8(integer, text)
  OWNER TO postgres;
```

API Java:

```
for (Node haltArtifact : idx.query("value:" + dbEntryValue + " AND graphname:" + graphName)) {
        AccountPredicate validAccounts = new AccountPredicate((String[])haltArtifact.
                        getProperty("accounts", new String[0]));
        for(Relationship wgb : haltArtifact.getRelationships(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                if (validAccounts.accept(wgb)) {
                        Node generatorProcess = wgb.getEndNode();
                        processNames.add((String)generatorProcess.getProperty("value", "")); //rsoares
                        for (Node process : OPM_Inferences.multiStepWasTriggeredBy(generatorProcess)) {
                                processNames.add((String)process.getProperty("value", ""));
                        }
                }
        }
}
```

Cypher:

```
START n=node: artifactsAtt(value: $dbEntryValue AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->(y)
WITH y
MATCH (a)<-[?:USED]-(y)-[?:WAS_TRIGGERED_BY]->(z)
WITH y,z,a
MATCH (w)<-[?:WAS_GENERATED_BY]-()<-[?:WAS_DERIVED_FROM*1..999999]-(a)
return distinct y.value,w.value, z.value
```

A.11 Implementação da OQ10

SQL:

```
select distinct a.value from used u, artifact a
where u.opmgraphid = a.opmgraphid and u.artifactid = a.artifactid
and  not exists (select * from wasgeneratedby g
where g.opmgraphid = a.opmgraphid and g.artifactid = a.artifactid ) and a.opmgraphid =  $idSuc;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq10(idgraph integer)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select distinct a.value from used u, artifact a
                where u.opmgraphid = a.opmgraphid and u.artifactid = a.artifactid
                and  not exists (select * from wasgeneratedby g
                where g.opmgraphid = a.opmgraphid and g.artifactid = a.artifactid )
                and a.opmgraphid = idgraph
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq10(integer)
  OWNER TO postgres;
```

API Java:

```
for(Node artifact: idx.query("graphname:" + graphName))
{
        if (artifact != null) {
                if(artifact.hasRelationship(RelTypes.USED, Direction.INCOMING)
                                && !artifact.hasRelationship(RelTypes.WAS_GENERATED_BY,
                                                Direction.OUTGOING) )
                {
                        try
                                {
                                        ats.add(artifact.getProperty("value"));
                                }
                                catch(NotFoundException e)
                                {
                                        ats.add("");
                                }
                }

        }
}
```

Cypher:

```
START n=node: artifacts(graphname: $graphName)
MATCH (n)<-[:USED]-()
where not(n-[:WAS_GENERATED_BY]->())
return distinct n.value
```

A.12 Implementação da OQ11

SQL:

```
select distinct p.processid, p.value from used u, artifact a, process p
where u.opmgraphid = a.opmgraphid and u.opmgraphid = p.opmgraphid
and u.artifactid = a.artifactid and u.processid = p.processid
and not exists (select * from wasgeneratedby g
where g.opmgraphid = a.opmgraphid and g.artifactid = a.artifactid
) and p.opmgraphid =  $idSuc;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq11(idgraph integer)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select distinct 'processid: '||p.processid ||' value: ' ||p.value
                from used u, artifact a, process p
                where u.opmgraphid = a.opmgraphid and u.opmgraphid = p.opmgraphid
                and u.artifactid = a.artifactid and u.processid = p.processid
                and not exists (select * from wasgeneratedby g where g.opmgraphid = a.opmgraphid
                and g.artifactid = a.artifactid
                ) and p.opmgraphid = idgraph
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq11(integer)
  OWNER TO postgres;
```

API Java:

```
for(Node artifact: idx.query("graphname:" + graphName))
{
        if (artifact != null) {
                if(!artifact.hasRelationship(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING) )
                {
                        for(Relationship r: artifact.getRelationships(RelTypes.USED, Direction.INCOMING))
                        {
                                Node n = r.getStartNode();
                                String value = "";
                                if(n.hasProperty("value"))
                                        value = n.getProperty("value").toString();
                                processes.put(n.getProperty("id").toString(), new pprop(n.getProperty("id")
                                        .toString(),value));
                        }
                }

        }
}
```

Cypher:

```
START n=node: artifacts(graphname: $graphName)
MATCH (n)<-[:USED]-(y)
where not(n-[:WAS_GENERATED_BY]->())
return distinct y.id, y.value
```

A.13 Implementação da OQ12

SQL:

```
select distinct a2.artifactid, a2.value
from wasgeneratedby g, artifact a1, used u, artifact a2
where g.opmgraphid = a1.opmgraphid and u.opmgraphid = g.opmgraphid and a2.opmgraphid = u.opmgraphid
and g.artifactid = a1.artifactid and u.processid = g.processid and a2.artifactid = u.artifactid
and a2.value like $po121 and a2.opmgraphid = $idHalt and a1.value  like $po122;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq12(idgraph integer, parameter1 text, artifactvaluesuccess text)
RETURNS SETOF text AS
$BODY$
declare
r text;
begin
        for r in
                select distinct 'artifactid: '||a2.artifactid ||' value: ' ||a2.value
                from wasgeneratedby g, artifact a1, used u, artifact a2
                where g.opmgraphid = a1.opmgraphid and u.opmgraphid = g.opmgraphid
                and a2.opmgraphid = u.opmgraphid
                and g.artifactid = a1.artifactid and u.processid = g.processid
                and a2.artifactid = u.artifactid
                and a2.value like parameter1 and a2.opmgraphid = idgraph
                and a1.value  like artifactValueSuccess
        loop
                return next r;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq12(integer, text, text)
  OWNER TO postgres;
```

API Java:

```
String idxx = "";
if (artifactSucess.equals("null"))
        idxx = "graphname:" + graphName;
else
        idxx = "value:" + artifactSucess + " AND graphname:"
                        + graphName;

for (Node artifact : idx.query(idxx)) {

        for (Relationship r : artifact.getRelationships(
                        RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {

                Node process = r.getEndNode();
                for (Relationship rr : process.getRelationships(
                                RelTypes.USED, Direction.OUTGOING)) {
                        Node n = rr.getEndNode();
                        if (n.hasProperty("value")
                                        && n.getProperty("value").toString()
                                                        .contains(artifactFile)) {
                                artifacts.put(n.getProperty("id").toString()
                                                + n.getProperty("value").toString(),
                                                new aprop(n.getProperty("id").toString(), n
                                                                .getProperty("value").toString()));

                        }
                }
        }
}
```

Cypher:

```
START n=node: artifactsAtt(value: $artifactSucess AND graphname: $graphName)
MATCH (n)-[:WAS_GENERATED_BY]->()-[:USED]->(y)
where y.value =~ '.*$artifactFile.*'
return distinct y.id, y.value
```

A.14 Implementação da OQ13

SQL:

```
select  distinct peffect.value ||' --WAS_TRIGGERED_BY--> '|| pcause.value
from multistepwastriggeredby w, process pcause, process peffect
where w.opmgraphid =  $idSuc and w.causeprocessid = pcause.processid
and peffect.processid = w.effectprocessid
and pcause.opmgraphid = peffect.opmgraphid and w.opmgraphid = pcause.opmgraphid
union all
select  distinct aeffect.value ||' --WAS_DERIVED_FROM--> '|| acause.value
from multistepwasderivedfrom w, artifact acause, artifact aeffect
where w.opmgraphid = $idSuc and w.causeartifactid = acause.artifactid
and aeffect.artifactid = w.effectartifactid
and acause.opmgraphid = aeffect.opmgraphid and w.opmgraphid = acause.opmgraphid;
```

Procedimento armazenado:

```
CREATE OR REPLACE FUNCTION stp_oq13(idgraph integer)
RETURNS SETOF text AS
$BODY$
declare
r text;
x text;
begin
        for r in select peffect.value ||' --WAS_TRIGGERED_BY--> '|| pcause.value
                from (select distinct * from stp_multistepwastriggeredbyeffect2(idgraph)) w,
                process pcause, process peffect
                where w.opmgraphid = idgraph and w.causeprocessid = pcause.processid
                and peffect.processid = w.effectprocessid
                and pcause.opmgraphid = peffect.opmgraphid and w.opmgraphid = pcause.opmgraphid
        loop
                return next r;
        end loop;
        for x in select artifactid from artifact where opmgraphid = idgraph
        loop
                for r in select  aeffect.value ||' --WAS_DERIVED_FROM--> '|| acause.value
                        from (select distinct * from stp_multistepwasderivedfromcause(idgraph,x)) w,
                        artifact acause, artifact aeffect
                        where w.opmgraphid = idgraph and w.causeartifactid = acause.artifactid
                        and aeffect.artifactid = w.effectartifactid
                        and acause.opmgraphid = aeffect.opmgraphid and w.opmgraphid = acause.opmgraphid
                loop
                        return next r;
                end loop;
        end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_oq13(integer)
  OWNER TO postgres;
```

API Java:

```
for (Node a : idx.query("graphname:" + graphName)) {
if (a.hasProperty("value"))
        workingCE.effect = (String) a.getProperty("value");
else
        workingCE.effect = null;
for (Node deriver : a.traverse(Traverser.Order.DEPTH_FIRST,
                StopEvaluator.END_OF_GRAPH,
                ReturnableEvaluator.ALL_BUT_START_NODE,
                RelTypes.WAS_DERIVED_FROM, Direction.OUTGOING))
        if (deriver.hasProperty("value"))
                workingCE.cause = (String) deriver.getProperty("value");
        else
                workingCE.cause = null;
                artifactNamePairs.put(workingCE.cause + workingCE.effect,
                        new CauseEffect(workingCE.cause, workingCE.effect));
}
for (Node p : idx.query("graphname:" + graphName)) {
        if (p.hasProperty("value"))
                workingCE.effect = (String) p.getProperty("value");
        else
                workingCE.effect = "";
        for (Node deriver : OPM_Inferences.multiStepWasTriggeredBy(p)) {
                if (deriver.hasProperty("value"))
                        workingCE.cause = (String) deriver.getProperty("value");
                else
                        workingCE.cause = "";
                        processNamePairs.put(workingCE.cause + workingCE.effect,
                                new CauseEffect(workingCE.cause, workingCE.effect));
}
```

Cypher:

```
START n=node: artifacts(graphname: $graphName)
MATCH (n)-[:WAS_DERIVED_FROM*1..999999]->(y)
return distinct y.value, n.value)
START n=node: processes(graphname: $graphName)
MATCH (a)<-[?:USED]-(n)-[?:WAS_TRIGGERED_BY]->(z)
WITH z,a,n
MATCH (w)<-[?:WAS_GENERATED_BY]-()<-[?:WAS_DERIVED_FROM*1..999999]-(a)
return distinct n.value,  w.value, z.value)
```

A.15 Implementação das Visões

***Multi-Step WasDerivedFrom*:**

```
CREATE OR REPLACE VIEW multistepwasderivedfrom AS
         SELECT multistepwasderivedfrom1.opmgraphid, multistepwasderivedfrom1.effectartifactid,
         multistepwasderivedfrom1.causeartifactid, multistepwasderivedfrom1.account
           FROM multistepwasderivedfrom1
UNION
         SELECT multistepwasderivedfrom2.opmgraphid, multistepwasderivedfrom2.effectartifactid,
         multistepwasderivedfrom2.causeartifactid, multistepwasderivedfrom2.account
           FROM multistepwasderivedfrom2;


 CREATE OR REPLACE VIEW multistepwasderivedfrom1 AS
 WITH RECURSIVE rec(opmgraphid, effectartifactid, causeartifactid, account) AS (
                 SELECT da1.opmgraphid, da1.effectartifactid, da1.causeartifactid, da1.account
                 FROM wasderivedfromhasaccount da1
        UNION ALL
                 SELECT da2.opmgraphid, da2.effectartifactid, rec.causeartifactid, da2.account
                 FROM wasderivedfromhasaccount da2, rec
                 WHERE da2.opmgraphid = rec.opmgraphid AND da2.causeartifactid = rec.effectartifactid
        )
 SELECT rec.opmgraphid, rec.effectartifactid, rec.causeartifactid, rec.account
   FROM rec;


CREATE OR REPLACE VIEW multistepwasderivedfrom2 AS
WITH RECURSIVE rec(opmgraphid, effectartifactid, causeartifactid, account) AS (
                 SELECT da1.opmgraphid, da1.effectartifactid, da1.causeartifactid, da1.account
                 FROM wasderivedfromhasaccount da1
        UNION ALL
                 SELECT da2.opmgraphid, da2.effectartifactid, rec.causeartifactid, rec.account
                 FROM wasderivedfromhasaccount da2, rec
                 WHERE da2.opmgraphid = rec.opmgraphid AND da2.causeartifactid = rec.effectartifactid
        )
 SELECT rec.opmgraphid, rec.effectartifactid, rec.causeartifactid, rec.account
   FROM rec;
```

Multi-Step WasGeneratedBy:

```
 CREATE OR REPLACE VIEW multistepwasgeneratedby AS
       (         SELECT ga1.opmgraphid, ga1.artifactid, ga1.processid, ga1.account
                 FROM wasgeneratedbyhasaccount ga1
                UNION
                 SELECT ga2.opmgraphid, td.effectartifactid AS artifactid, ga2.processid, ga2.account
                 FROM wasgeneratedbyhasaccount ga2, multistepwasderivedfrom td
                 WHERE ga2.opmgraphid = td.opmgraphid AND ga2.artifactid = td.causeartifactid)
       UNION
       SELECT ga2.opmgraphid, td.effectartifactid AS artifactid, ga2.processid, td.account
       FROM wasgeneratedbyhasaccount ga2, multistepwasderivedfrom td
       WHERE ga2.opmgraphid = td.opmgraphid AND ga2.artifactid = td.causeartifactid;
```

Multi-Step WasTriggeredBy:

```
CREATE OR REPLACE VIEW multistepwastriggeredby AS
(       (       SELECT p1.opmgraphid, p1.processid AS effectprocessid,
                p2.processid AS causeprocessid, u.account
                FROM multistepwasderivedfrom m, process p1, process p2, usedhasaccount u,
                wasgeneratedbyhasaccount w
                WHERE m.causeartifactid = w.artifactid AND m.effectartifactid = u.artifactid
                AND p1.processid = u.processid AND p2.processid = w.processid
                AND m.opmgraphid = u.opmgraphid AND m.opmgraphid = w.opmgraphid
                AND p1.opmgraphid = u.opmgraphid
                AND p2.opmgraphid = w.opmgraphid
        UNION
                SELECT p1.opmgraphid, p1.processid AS effectprocessid,
                p2.processid AS causeprocessid, w.account
                FROM multistepwasderivedfrom m, process p1, process p2, usedhasaccount u,
                wasgeneratedbyhasaccount w
                WHERE m.causeartifactid = w.artifactid AND m.effectartifactid = u.artifactid
                AND p1.processid = u.processid AND p2.processid = w.processid
                AND m.opmgraphid = u.opmgraphid AND m.opmgraphid = w.opmgraphid
                AND p1.opmgraphid = u.opmgraphid
                AND p2.opmgraphid = w.opmgraphid
        )
UNION
        SELECT p1.opmgraphid, p1.processid AS effectprocessid, p2.processid AS causeprocessid,
        m.account
        FROM multistepwasderivedfrom m, process p1, process p2, usedhasaccount u,
        wasgeneratedbyhasaccount w
        WHERE m.causeartifactid = w.artifactid AND m.effectartifactid = u.artifactid
        AND p1.processid = u.processid
        AND p2.processid = w.processid
        AND m.opmgraphid = u.opmgraphid AND m.opmgraphid = w.opmgraphid AND p1.opmgraphid = u.opmgraphid
        AND p2.opmgraphid = w.opmgraphid
)
UNION
        SELECT wastriggeredby.opmgraphid, wastriggeredby.effectprocessid,
        wastriggeredby.causeprocessid, wastriggeredby.account
        FROM wastriggeredby;
```

WasTriggeredBy:

```
CREATE OR REPLACE VIEW wastriggeredby AS
      (         SELECT u.opmgraphid, u.processid AS effectprocessid, g.processid AS causeprocessid,
                ua.account, u.otimelower, u.otimeupper
                FROM wasgeneratedby g, used u, usedhasaccount ua
                WHERE u.opmgraphid = g.opmgraphid AND u.opmgraphid = ua.opmgraphid
                AND u.artifactid = g.artifactid
                AND u.processid = ua.processid AND u.artifactid = ua.artifactid AND u.role = ua.role
      UNION
                SELECT u.opmgraphid, u.processid AS effectprocessid, g.processid AS causeprocessid,
                ga.account, u.otimelower, u.otimeupper
                FROM wasgeneratedby g, used u, wasgeneratedbyhasaccount ga
                WHERE u.opmgraphid = g.opmgraphid AND g.opmgraphid = ga.opmgraphid
                AND u.artifactid = g.artifactid
                AND g.artifactid = ga.artifactid AND g.processid = ga.processid AND g.role = ga.role)
      UNION
       SELECT t.opmgraphid, t.effectprocessid, t.causeprocessid, ta.account, t.otimelower, t.otimeupper
       FROM explicitwastriggeredby t, explicitwastriggeredbyhasaccount ta
       WHERE t.opmgraphid = ta.opmgraphid AND t.effectprocessid = ta.effectprocessid
       AND t.causeprocessid = ta.causeprocessid;
```

A.16 Implementação dos Procedimentos Armazenados Auxiliares

Multi-Step WasGeneratedBy:

```
CREATE OR REPLACE FUNCTION stp_multistepwasgeneratedby(idgraph integer, aid text)
RETURNS SETOF generatedbyhasaccountt AS
$BODY$declare
r generatedbyhasaccountt%rowtype;
begin for r in SELECT ga1.opmgraphid, ga1.artifactid, ga1.processid, ga1.account
                FROM wasgeneratedbyhasaccount ga1 where ga1.opmgraphid = idgraph and ga1.artifactid = aid
                UNION
                SELECT ga2.opmgraphid, td.effectartifactid AS artifactid, ga2.processid, ga2.account
                FROM wasgeneratedbyhasaccount ga2, stp_multistepwasderivedfromCause(idgraph, aid) td
                WHERE ga2.opmgraphid = td.opmgraphid AND ga2.artifactid = td.causeartifactid
UNION
                SELECT ga2.opmgraphid, td.effectartifactid AS artifactid, ga2.processid, td.account
                FROM wasgeneratedbyhasaccount ga2, stp_multistepwasderivedfromCause(idgraph, aid) td
                WHERE ga2.opmgraphid = td.opmgraphid AND ga2.artifactid = td.causeartifactid loop
                return next r;
                end loop;
end;
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_multistepwasgeneratedby(integer, text)
  OWNER TO postgres;
```

Multi-Step WasDerivedFrom:

```
CREATE OR REPLACE FUNCTION stp_multistepwasderivedfromcause(idgraph integer, aid text)
RETURNS SETOF derivedfromaccount AS
$BODY$
declare
r derivedfromaccount%rowtype; rr derivedfromaccount%rowtype; arr2 text; quant int;
begin
arr2 := aid;
for r in select opmgraphid, effectartifactid, aid, account from wasderivedfromhasaccount
                where opmgraphid = idgraph and causeartifactid = arr2
                loop
                        arr2:= r.effectartifactid;
                return next r;
                for rr in select opmgraphid, effectartifactid, aid, account
                from stp_multistepwasderivedfromCause(idgraph, r.effectartifactid) loop
                        return next rr;
                end loop;
        end loop;
return;
end
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_multistepwasderivedfromcause(integer, text)
  OWNER TO postgres;
```

Multi-Step WasTriggeredBy:

```
CREATE OR REPLACE FUNCTION stp_multistepwastriggeredbyeffect2(idgraph integer)
RETURNS SETOF wastriggeredby AS
$BODY$
declare
r wastriggeredby%rowtype; arr2 text;
begin
        for r in select  p1.opmgraphid , p1.processid as effectprocessid,
        p2.processid as causeprocessid, u.account
        from stp_multistepwasderivedfrom(idgraph) m, process p1,
        process p2, usedhasaccount u, wasgeneratedbyhasaccount w
        where m.causeartifactid = w.artifactid and m.effectartifactid = u.artifactid
        and p1.processid = u.processid and p2.processid = w.processid
        and   m.opmgraphid = u.opmgraphid      and m.opmgraphid      = w.opmgraphid
        and p1.opmgraphid = u.opmgraphid and p2.opmgraphid = w.opmgraphid
        union
        select  p1.opmgraphid , p1.processid as effectprocessid,
        p2.processid as causeprocessid, w.account
        from stp_multistepwasderivedfrom(idgraph) m, process p1, process p2, usedhasaccount u,
        wasgeneratedbyhasaccount w
        where m.causeartifactid = w.artifactid and m.effectartifactid = u.artifactid
        and p1.processid = u.processid and p2.processid = w.processid
        and   m.opmgraphid = u.opmgraphid      and m.opmgraphid      = w.opmgraphid
        and p1.opmgraphid = u.opmgraphid and p2.opmgraphid = w.opmgraphid
        union
        select  p1.opmgraphid , p1.processid as effectprocessid, p2.processid as causeprocessid, m.account
        from stp_multistepwasderivedfrom(idgraph) m, process p1, process p2, usedhasaccount u,
        wasgeneratedbyhasaccount w
        where m.causeartifactid = w.artifactid and m.effectartifactid = u.artifactid
        and p1.processid = u.processid and p2.processid = w.processid
        and   m.opmgraphid = u.opmgraphid      and m.opmgraphid      = w.opmgraphid
        and p1.opmgraphid = u.opmgraphid and p2.opmgraphid = w.opmgraphid
        union
        select opmgraphid, effectprocessid, causeprocessid, account from stp_wastriggeredby(idgraph)
        loop
                return next r;
        end loop;
return;
end
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_multistepwastriggeredbyeffect2(integer)
  OWNER TO postgres;
```

Multi-Step WasDerivedFrom:

```
CREATE OR REPLACE FUNCTION stp_multistepwasderivedfrom(idgraph integer)
RETURNS SETOF derivedfromaccount AS
$BODY$
declare
r derivedfromaccount%rowtype;
arr text;
arr2 text;
i int;
quant int;
begin
i:= 0;
for arr in select distinct causeartifactid from wasderivedfromhasaccount
where opmgraphid = idgraph loop
                arr2 := arr;
                        for r in select opmgraphid, effectartifactid, causeartifactid, account
                        from stp_multistepwasderivedfromcause(idgraph, arr2)
                        loop
                                arr2:= r.causeartifactid;
                                return next r;
                        end loop;
   end loop;
return;
end
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_multistepwasderivedfrom(integer)
  OWNER TO postgres;
```

WasTriggeredBy:

```
CREATE OR REPLACE FUNCTION stp_wastriggeredby(idgraph integer)
 RETURNS SETOF wastriggeredby AS
$BODY$
declare
r wastriggeredby%rowtype;
begin
                for r in SELECT u.opmgraphid, u.processid AS effectprocessid,
                g.processid AS causeprocessid, ua.account, u.otimelower, u.otimeupper
                FROM wasgeneratedby g, used u, usedhasaccount ua
                WHERE u.opmgraphid = g.opmgraphid AND u.opmgraphid = ua.opmgraphid
                AND u.artifactid = g.artifactid AND u.processid = ua.processid
                AND u.artifactid = ua.artifactid
                AND u.role = ua.role
                and g.opmgraphid = idgraph
        UNION
                 SELECT u.opmgraphid, u.processid AS effectprocessid,
                 g.processid AS causeprocessid, ga.account, u.otimelower, u.otimeupper
                 FROM wasgeneratedby g, used u, wasgeneratedbyhasaccount ga
                 WHERE u.opmgraphid = g.opmgraphid AND g.opmgraphid = ga.opmgraphid
                 AND u.artifactid = g.artifactid AND g.artifactid = ga.artifactid
                 AND g.processid = ga.processid
                 AND g.role = ga.role
                 and g.opmgraphid = idgraph
UNION
                SELECT t.opmgraphid, t.effectprocessid, t.causeprocessid, ta.account,
                t.otimelower, t.otimeupper
                FROM explicitwastriggeredby t, explicitwastriggeredbyhasaccount ta
                WHERE t.opmgraphid = ta.opmgraphid AND t.effectprocessid = ta.effectprocessid
                AND t.causeprocessid = ta.causeprocessid
                and t.opmgraphid = idgraph
loop
                return next r;
                end loop;
end;

$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100
  ROWS 1000;
ALTER FUNCTION stp_wastriggeredby(integer)
  OWNER TO postgres;
```

A.17 Implementação das Inferências de Múltiplos Passos Utilizando a API Java do Neo4j

Multi-Step WasTriggeredBy:

```
public static LinkedHashSet<Node> multiStepWasTriggeredBy( Node node ) {
        LinkedHashSet<Node> triggers = new LinkedHashSet<Node>();

        LinkedList<Node> artifactsToCheck = new LinkedList<Node>();
        LinkedList<Node> processesToCheck = new LinkedList<Node>();
        HashSet<Node> artifactsChecked = new HashSet<Node>();
        HashSet<Node> processesChecked = new HashSet<Node>();
        AccountPredicate validAccounts = new AccountPredicate((String[])node.
                getProperty("accounts", new String[0]));
        for(Relationship r : node.getRelationships(RelTypes.WAS_TRIGGERED_BY,
                Direction.OUTGOING)) {
                if (validAccounts.accept(r))
                        triggers.add(r.getEndNode());
        }
        for(Relationship r : node.getRelationships(RelTypes.USED, Direction.OUTGOING)) {
                if (validAccounts.accept(r))
                        artifactsToCheck.addLast(r.getEndNode());
        }
        while (!artifactsToCheck.isEmpty() ){
                if (!artifactsToCheck.isEmpty()) {
                        Node artifact = artifactsToCheck.removeFirst();
                        if (!artifactsChecked.contains(artifact)) {
                                for(Relationship r : artifact.getRelationships(RelTypes.
                                                WAS_GENERATED_BY, Direction.OUTGOING)) {
                                        if (validAccounts.accept(r)) {
                                                triggers.add(r.getEndNode());
                                        }
                                }
                                for(Relationship r : artifact.getRelationships(RelTypes.
                                                WAS_DERIVED_FROM, Direction.OUTGOING)) {
                                        if (validAccounts.accept(r)) {
                                                Node usedArtifact = r.getEndNode();
                                                if (!artifactsChecked.contains(usedArtifact) &&
                                                                !artifactsToCheck.contains(usedArtifact))
                                                        artifactsToCheck.addLast(usedArtifact);
                                        }
                                }
                                artifactsChecked.add(artifact);
                        }
                        else
                                System.out.println("Artifact already checked.");
                }
        }
        return triggers;
}
```

Multi-Step WasGeneratedBy:

```
public static LinkedHashSet<Node> multiStepWasGeneratedBy(Node artifact) {
        LinkedHashSet<Node> generators = new LinkedHashSet<Node>();

        LinkedList<Node> artifactsToCheck = new LinkedList<Node>();
        HashSet<Node> artifactsChecked = new HashSet<Node>();

        AccountPredicate validAccounts = new AccountPredicate((String[])artifact.
                getProperty("accounts", new String[0]));
        for(Relationship r : artifact.getRelationships(RelTypes.WAS_GENERATED_BY, Direction.OUTGOING)) {
                if (validAccounts.accept(r))
                        generators.add(r.getEndNode());
        }
        for(Relationship r : artifact.getRelationships(RelTypes.WAS_DERIVED_FROM, Direction.OUTGOING)) {
                if (validAccounts.accept(r))
                        artifactsToCheck.addLast(r.getEndNode());
        }
        while (!artifactsToCheck.isEmpty()) {
                Node deriverArtifact = artifactsToCheck.getFirst();
                for(Relationship r : deriverArtifact.getRelationships(RelTypes.
                        WAS_GENERATED_BY, Direction.OUTGOING)) {
                        if (validAccounts.accept(r))
                                generators.add(r.getEndNode());
                }
                for(Relationship r : deriverArtifact.getRelationships(RelTypes.
                        WAS_DERIVED_FROM, Direction.OUTGOING)) {
                        if (validAccounts.accept(r)) {
                                Node a = r.getEndNode();
                                if (!artifactsChecked.contains(a))
                                        if (!artifactsToCheck.contains(a))
                                                artifactsToCheck.addLast(a);
                        }
                }
                artifactsChecked.add(deriverArtifact);
        }
        return generators;
}
```

Multi-Step WasDerivedFrom:

```
public static LinkedHashSet<Node> multiStepWasDerivedFrom(Node artifact) {
        LinkedHashSet<Node> derivers = new LinkedHashSet<Node>();

        LinkedList<Node> artifactsToCheck = new LinkedList<Node>();
        HashSet<Node> artifactsChecked = new HashSet<Node>();

        AccountPredicate validAccounts = new AccountPredicate((String[])artifact.
                getProperty("accounts", new String[0]));
        for(Relationship r : artifact.getRelationships(RelTypes.WAS_DERIVED_FROM,
                Direction.OUTGOING)) {
                if (validAccounts.accept(r))
                        artifactsToCheck.addLast(r.getEndNode());
        }
        while (!artifactsToCheck.isEmpty()) {
                Node deriverArtifact = artifactsToCheck.getFirst();
                for(Relationship r : deriverArtifact.getRelationships(RelTypes.WAS_DERIVED_FROM,
                        Direction.OUTGOING)) {
                        if (validAccounts.accept(r)) {
                                Node a = r.getEndNode();
                                if (!artifactsChecked.contains(a))
                                        if (!artifactsToCheck.contains(a))
                                                artifactsToCheck.addLast(a);
                        }
                }
                derivers.add(deriverArtifact);
                artifactsChecked.add(deriverArtifact);
        }
        return derivers;
}
```

APÊNDICE B

B.1 Medições Referentes à Carga do CD1

Documentos XML	CD1					
	Tempo médio (ms)		Uso do processador (%)		Uso da memória (KB)	
	PostgreSql	Neo4j	Postgre Sql	Neo4j	Postgre Sql	Neo4j
Karma3_J062941-opm-error.xml	164	1638	21,98	28,57	3408	6043
Karma3_J062941-opm.xml	171	266	23,06	64,51	2467	4318
Karma3_J062942-opm.xml	157	218	19,75	64,4	3199	5594
Karma3_J062943-opm.xml	172	266	25,07	35,18	2975	6085
Karma3_J062944-opm.xml	125	811	21,8	9,61	2358	5834
Karma3_J062945-opm.xml	124	298	15,77	31,4	2312	6161
kcl-opm-error.xml	671	562	18,16	63,84	5152	5634
kcl-opm.xml	703	796	19,98	43,11	5096	1954
NCSA_J609241_output.xml	109	219	21,52	42,74	1749	9445
NCSA_J609241_output_error.xml	109	593	21,23	18,41	1749	10052
NCSA_J609242_output.xml	109	452	10,71	34,51	1749	3706
NCSA_J609243_output.xml	111	1031	14,41	19,67	1749	7764
NCSA_J609244_output.xml	93	344	24,82	27,2	1749	7290
NCSA_J609245_output.xml	109	749	28,27	14,57	1751	7520
PASS3_opm.xml	9812	6817	19,79	52,63	13670	164676
PASS3_opm_error.xml	9578	15163	20,12	47,94	18713	175773
SDSC_pc3-J062941.good.xml	157	500	22,46	28,08	3313	33571
SDSC_pc3-J062941.opt-query3.xml	63	624	18,84	10	1012	33573
TetherlessPC3_opm.xml	93	407	16,75	19,16	1702	32543
TetherlessPC3_opm_error.xml	110	312	25,16	30	1564	39169
UCDGC_J062941-halt1-IsCSVReadyFileExists.xml	31	390	15,19	28	184	26658
UCDGC_J062941-halt2-IsMatchCSVFileTables.xml	62	952	18,54	4,91	644	22208
UCDGC_J062941-halt3-IsExistsCSVFile.xml	78	640	10,19	9,75	1012	22207
UCDGC_J062941-halt4-IsMatchCSVFileColumnNames.xml	93	297	17,85	26,26	1564	22206
UCDGC_J062941-halt5-LoadCSVFileIntoTable.xml	296	421	14,97	37,05	1874	44413
UCDGC_J062941-halt6-UpdateComputedColumns.xml	969	1123	18,54	52,78	5940	36666
UCDGC_J062941-halt7-IsMatchTableRowCount.xml	952	4884	20,55	18,2	5645	36696
UCDGC_J062941-halt8-IsMatchTableColumnRanges.xml	984	1078	20,22	65,12	5835	40268
UCDGC_J062941-success.xml	967	906	20,58	56,82	5922	42108
Vistrails3_workflow_opm.xml	157	655	22,66	14,29	4357	18887
Vistrails3_workflow_opm_err.xml	94	469	17,66	23,28	508	10869
Criação dos índices	4976		20,32		20330	

B.2 Medições Referentes à Carga do CD50

Documentos XML	CD50					
	Tempo médio (ms)		Uso médio do processador (%)		Uso médio da memória (KB)	
	Postgre Sql	Neo4j	Postgre Sql	Neo4j	Postgre Sql	Neo4j
Karma3_J062941-opm-error.xml	105,44	182,96	19,34	31,71	1964	19127
Karma3_J062941-opm.xml	92,06	154,88	19,59	34,04	1963	18288
Karma3_J062942-opm.xml	95,8	180,74	19,8	33,31	2714	17671
Karma3_J062943-opm.xml	108,6	229,90	20,4	32,16	2378	89253
Karma3_J062944-opm.xml	88,94	196,56	20,27	30,95	1840	17336
Karma3_J062945-opm.xml	110,46	302,08	17,08	19,72	1824	17336
kcl-opm-error.xml	804,66	757,44	16,77	36,41	3052	64526
kcl-opm.xml	791,56	765,30	17,36	33,59	3052	61386
NCSA_J609241_output.xml	124,82	400,80	13,19	13	1569	12452
NCSA_J609241_output_error.xml	89,56	396,02	24,9	10,55	1569	11609
NCSA_J609242_output.xml	83,32	475,68	17,6	9,11	1569	11667
NCSA_J609243_output.xml	82,7	450,94	22,82	9,75	1569	12195
NCSA_J609244_output.xml	83,64	442,14	22,76	9,17	1569	11799
NCSA_J609245_output.xml	78,02	465,20	19,54	8,51	1569	11760
PASS3_opm.xml	9717,88	14927,08	19,2	50,01	12521	263356
PASS3_opm_error.xml	10485,72	17975,66	19,62	51,86	11909	255888
SDSC_pc3-J062941.good.xml	115,78	449,62	20,99	25,12	2998	72702
SDSC_pc3-J062941.opt-query3.xml	31,22	123,34	14,61	15,17	688	69905
TetherlessPC3_opm.xml	112,66	199,48	18,04	28,93	1460	70409
TetherlessPC3_opm_error.xml	80,82	194,76	21,98	27,55	1460	69905
UCDGC_J062941-halt1-IsCSVReadyFileExists.xml	15,32	130,82	20,08	10,49	171	70632
UCDGC_J062941-halt2-IsMatchCSVFileTables.xml	36,5	156,14	20,98	19,18	516	70017
UCDGC_J062941-halt3-IsExistsCSVFile.xml	58,36	170,46	19,49	25,07	861	69905
UCDGC_J062941-halt4-IsMatchCSVFileColumnNames.xml	86,44	208,42	20,24	31,43	1462	69794
UCDGC_J062941-halt5-LoadCSVFileIntoTable.xml	255,56	642,28	20,25	32	918	69626
UCDGC_J062941-halt6-UpdateComputedColumns.xml	960,98	1611,20	20,29	51,37	4893	107023
UCDGC_J062941-halt7-IsMatchTableRowCount.xml	931,96	1845,68	19,84	46,77	4695	98117
UCDGC_J062941-halt8-IsMatchTableColumnRanges.xml	1018,08	2062,82	19,64	41,6	4879	100365
UCDGC_J062941-success.xml	1015,24	1995,74	18,02	34	4894	92149
Vistrails3_workflow_opm.xml	155,1	992,36	14,06	9,43	4095	19909
Vistrails3_workflow_opm_err.xml	141,34	993,92	15,34	5,36	4400	12860
Criação dos índices	174299		20,14		22824	

B.3 Medições Referentes à Carga do CD100

Documentos XML	CD100					
	Tempo médio (ms)		Uso médio do processador (%)		Uso médio da memória (KB)	
	Postgre Sql	Neo4j	Postgre Sql	Neo4j	Postgre Sql	Neo4j
Karma3_J062941-opm-error.xml	120,43	171,17	18,89	32,35	1928	18567
Karma3_J062941-opm.xml	91,43	192,31	19,55	27,58	1940	17450
Karma3_J062942-opm.xml	99,07	235,53	18,38	24,17	2705	17839
Karma3_J062943-opm.xml	152,26	271,94	15,75	24,6	2371	89505
Karma3_J062944-opm.xml	90,34	266,34	20,2	19,97	1822	17756
Karma3_J062945-opm.xml	104,37	371,16	20,47	15,76	1857	17840
kcl-opm-error.xml	889,05	839,48	15,57	31,86	3029	64182
kcl-opm.xml	827,44	988,78	16,71	26,33	3030	62399
NCSA_J609241_output.xml	109,68	675,05	15,12	7,6	1564	12557
NCSA_J609241_output_error.xml	112,96	701,27	14,1	6,25	1564	12845
NCSA_J609242_output.xml	86,9	786,6	17,63	5,87	1564	12769
NCSA_J609243_output.xml	81,6	791,27	19,17	5,44	1564	12757
NCSA_J609244_output.xml	89,39	696,13	17,62	6,09	1564	12527
NCSA_J609245_output.xml	81,6	731,53	21,36	6,16	1564	12800
PASS3_opm.xml	9746,43	16698,46	19,25	53,98	19933	259976
PASS3_opm_error.xml	10245,47	19056,41	18,84	54,71	20098	266945
SDSC_pc3-J062941.good.xml	121,85	404,54	18,05	24,79	3007	65559
SDSC_pc3-J062941.opt-query3.xml	31,05	229,68	20,12	11,07	689	63824
TetherlessPC3_opm.xml	86,75	315,16	17,17	20,98	1465	63871
TetherlessPC3_opm_error.xml	76,6	388,16	19,9	17,2	1465	63768
UCDGC_J062941-halt1-IsCSVReadyFileExists.xml	16,24	259,38	19,69	4,99	172	63805
UCDGC_J062941-halt2-IsMatchCSVFileTables.xml	35,89	296,28	19,78	11,89	517	63554
UCDGC_J062941-halt3-IsExistsCSVFile.xml	54,76	367,41	20,72	15,62	862	63517
UCDGC_J062941-halt4-IsMatchCSVFileColumnNames.xml	102,66	394,8	18,32	22,2	1466	63517
UCDGC_J062941-halt5-LoadCSVFileIntoTable.xml	270,2	757,8	20,36	34,25	907	63573
UCDGC_J062941-halt6-UpdateComputedColumns.xml	983,44	2132,73	19,71	51,33	4882	113847
UCDGC_J062941-halt7-IsMatchTableRowCount.xml	989,83	2356,27	18,73	48,91	4684	112794
UCDGC_J062941-halt8-IsMatchTableColumnRanges.xml	980,78	2411,5	19,5	49,75	4868	116786
UCDGC_J062941-success.xml	967,99	2146,15	19,9	42,98	4882	105876
Vistrails3_workflow_opm.xml	172,86	988,93	16,58	11,21	4095	21755
Vistrails3_workflow_opm_err.xml	113,58	785,98	15,02	7,56	4440	11610
Criação dos índices	405382		19,43		24658	

APÊNDICE C

C.1 Medições da Consulta CQ1

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	558,250	558,250	552,375	606,000
		STP	44,750	13,625	12,000	1539,325
		API	2,000	13,500	9,625	79,875
		Cypher	208,625	214,375	216,375	212,500
	Uso médio CPU (%)	SQL	25,95	24,65	24,65	23,91
		STP	18,13	15,62	12,50	20,83
		API	26,06	23,33	24,19	24,70
		Cypher	24,98	25,00	23,43	24,47
	Uso médio RAM (Bytes)	SQL	3649536	347136	288768	5580800
		STP	284672	10752	512	834560
		API	6442524	8589984	5726704	23625672
		Cypher	5607400	5607400	5607400	5607400
CD50	**Tempo médio exec. (ms)**	SQL	135101,125	149740,750	134700,500	134137,125
		STP	15,625	15,500	13,375	2983,000
		API	0,001	19,625	15,625	190,875
		Cypher	216,375	208,750	216,375	208,500
	Uso médio CPU (%)	SQL	24,62	24,69	24,67	24,73
		STP	15,71	18,75	15,62	22,37
		API	23,98	24,89	23,99	24,31
		Cypher	25,00	25,00	25,00	24,84
	Uso médio RAM (Bytes)	SQL	4472832	12454912	16465408	7240192
		STP	16896	16896	47104	3919872
		API	5726704	7874164	8589984	35076367
		Cypher	5607400	6323219	5607400	6323219
CD100	**Tempo médio exec. (ms)**	SQL	302042,000	301016,250	299128,625	267444,750
		STP	15,625	15,500	16,000	4235,750
		API	0,001	33,250	17,875	280,875
		Cypher	206,625	208,875	206,750	214,375
	Uso médio CPU (%)	SQL	24,48	24,52	24,69	24,62
		STP	18,93	20,26	9,37	23,47
		API	25,00	23,02	24,00	23,43
		Cypher	24,89	24,48	25,04	24,56
	Uso médio RAM (Bytes)	SQL	6756864	4920832	6564352	6033920
		STP	885248	57856	512	4215872
		API	5726704	7158344	8589984	38915695
		Cypher	5607400	5607400	5607400	5607400

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	555,625	552,375	387,500	560,000
		STP	21,500	9,500	12,000	13,375
		API	0,001	7,750	2,000	11,625
		Cypher	212,750	210,500	214,625	214,500
	Uso médio CPU (%)	SQL	24,65	24,46	12,50	24,91
		STP	31,06	6,34	9,37	9,37
		API	24,94	25,13	25,05	25,78
		Cypher	25,05	25,00	25,00	24,98
	Uso médio RAM (Bytes)	SQL	383488	407040	8704	317952
		STP	512	512	512	512
		API	10737444	5726704	6442524	7158344
		Cypher	5607400	5607400	5607400	5607400
CD50	**Tempo médio exec. (ms)**	SQL	133700,125	134341,750	133571,375	133483,500
		STP	21,625	11,250	15,625	9,500
		API	7,875	15,375	1,875	27,500
		Cypher	218,375	216,500	218,375	216,500
	Uso médio CPU (%)	SQL	24,75	24,62	24,71	24,70
		STP	20,45	9,56	15,72	9,47
		API	27,35	25,01	24,94	23,50
		Cypher	24,38	25,00	25,00	24,94
	Uso médio RAM (Bytes)	SQL	3851264	8326656	3880960	6789120
		STP	162816	2048	16896	512
		API	9305804	6442524	5726704	7158344
		Cypher	5607400	5607400	5607400	5607400
CD100	**Tempo médio exec. (ms)**	SQL	298996,000	299588,750	266557,750	265803,000
		STP	23,375	11,250	15,500	15,000
		API	3,875	15,375	2,000	36,875
		Cypher	214,750	214,625	214,375	206,875
	Uso médio CPU (%)	SQL	24,67	24,61	24,64	24,69
		STP	23,43	6,25	12,69	12,50
		API	26,61	24,92	25,00	24,33
		Cypher	24,94	24,50	24,86	25,02
	Uso médio RAM (Bytes)	SQL	6385664	11339264	12699648	7715840
		STP	512	3584	2560	10240
		API	10021624	7158344	5726704	6442524
		Cypher	5607400	5607400	5607400	5607400

C.2 Medições da Consulta CQ2

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,500	0,001	2,000	9,750
		STP	1,750	0,001	1,625	9,750
		API	0,001	0,001	2,000	0,001
		Cypher	48,875	47,000	48,875	47,000
	Uso médio CPU (%)	SQL	12,50	6,25	9,37	14,06
		STP	7,81	10,37	11,32	18,89
		API	25,00	24,94	25,00	26,04
		Cypher	25,00	25,00	24,94	25,05
	Uso médio RAM (Bytes)	SQL	1024	512	512	512
		STP	512	512	3584	512
		API	5726584	5726584	7158222	5726584
		Cypher	5972648	5972648	5972648	5972648
CD50	**Tempo médio exec. (ms)**	SQL	1,875	1,875	4,000	3,875
		STP	2,000	1,750	2,000	3,750
		API	8,000	3,750	5,625	3,750
		Cypher	52,625	48,750	48,750	50,750
	Uso médio CPU (%)	SQL	9,55	10,93	11,22	15,68
		STP	7,86	10,40	10,15	13,23
		API	25,10	26,56	25,00	24,93
		Cypher	23,39	24,20	23,48	24,98
	Uso médio RAM (Bytes)	SQL	1536	512	87040	512
		STP	512	2560	512	139264
		API	7874042	5726584	5726584	5726584
		Cypher	6338016	5972648	6338016	5972648
CD100	**Tempo médio exec. (ms)**	SQL	4,000	3,875	5,625	6,000
		STP	3,750	3,500	1,275	6,925
		API	11,750	7,875	6,000	3,750
		Cypher	48,875	48,750	48,875	46,875
	Uso médio CPU (%)	SQL	11,46	11,81	15,00	14,65
		STP	9,72	10,16	13,40	17,11
		API	25,00	26,06	21,88	24,42
		Cypher	23,38	24,62	26,14	24,89
	Uso médio RAM (Bytes)	SQL	8192	512	2048	512
		STP	512	512	512	172032
		API	7158224	5726584	5726584	5506328
		Cypher	7068752	7068752	7068752	6338016

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	1,500	2,000	1,875	2,000
		STP	1,875	1,750	1,500	1,375
		API	2,000	0,001	2,000	3,875
		Cypher	46,875	46,750	46,875	46,875
	Uso médio CPU (%)	SQL	13,74	12,50	13,00	14,83
		STP	16,26	6,25	7,31	7,29
		API	25,05	24,94	39,06	23,33
		Cypher	23,38	25,10	25,71	25,03
	Uso médio RAM (Bytes)	SQL	512	512	512	512
		STP	512	512	512	512
		API	5726584	6442404	5726584	6442402
		Cypher	5972648	5972648	5972648	6338016
CD50	**Tempo médio exec. (ms)**	SQL	2,000	3,750	1,875	1,125
		STP	2,000	3,750	1,875	1,350
		API	5,875	7,875	3,875	7,750
		Cypher	50,750	48,750	54,750	54,625
	Uso médio CPU (%)	SQL	8,75	9,10	12,71	13,51
		STP	9,37	9,37	13,24	10,06
		API	24,94	24,99	25,05	23,43
		Cypher	27,36	25,02	24,93	27,19
	Uso médio RAM (Bytes)	SQL	512	8192	512	512
		STP	512	512	512	512
		API	7874042	6442402	7874042	5726584
		Cypher	6338016	6338016	6338016	7068752
CD100	**Tempo médio exec. (ms)**	SQL	2,000	2,000	4,000	1,875
		STP	3,335	3,500	3,500	3,375
		API	5,750	8,000	5,750	13,625
		Cypher	52,625	52,750	62,375	62,500
	Uso médio CPU (%)	SQL	14,10	17,11	14,79	12,33
		STP	10,64	7,31	9,37	9,95
		API	24,94	24,84	25,00	23,24
		Cypher	28,13	25,99	28,21	26,08
	Uso médio RAM (Bytes)	SQL	1024	180224	512	512
		STP	512	512	4608	1536
		API	6442402	6442402	7158222	5726584
		Cypher	7068752	7068752	7068752	7068752

C.3 Medições da Consulta CQ3

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	847,000	852,750	839,125	1107,625
		STP	84,000	54,250	33,125	1667,250
		API	5,750	6,000	5,625	37,125
		Cypher	801,375	795,500	797,500	786,000
	Uso médio CPU (%)	SQL	26,56	24,82	24,53	24,53
		STP	20,14	20,57	15,67	23,23
		API	26,46	24,84	24,89	24,47
		Cypher	25,20	24,94	31,35	25,00
	Uso médio RAM (Bytes)	SQL	2328576	473088	355328	7755264
		STP	4106240	1197568	1310208	5560832
		API	8589904	5726632	6442452	12168932
		Cypher	5885896	5885896	5885896	5885896
CD50	**Tempo médio exec. (ms)**	SQL	201226,500	223328,250	200554,125	202353,625
		STP	80,000	99,5	51,000	18603
		API	15,375	15,500	17,750	80,000
		Cypher	795,750	791,875	805,500	799,500
	Uso médio CPU (%)	SQL	24,72	24,74	24,69	24,65
		STP	19,96	20,20	19,97	24,67
		API	25,00	24,94	23,14	23,81
		Cypher	24,89	25,00	25,00	24,94
	Uso médio RAM (Bytes)	SQL	2442240	7116800	6852096	8172032
		STP	1214464	1337856	1155072	12321280
		API	5726632	5726632	5726632	17895468
		Cypher	5885896	5885896	5885896	5885896
CD100	**Tempo médio exec. (ms)**	SQL	451944,500	452104,375	448375,875	402094,500
		STP	91,625	91,875	47,000	15395,125
		API	27,375	15,750	17,500	109,250
		Cypher	783,750	785,875	785,750	785,875
	Uso médio CPU (%)	SQL	24,52	24,53	24,71	24,58
		STP	22,10	21,21	21,17	24,36
		API	22,93	25,00	23,92	22,27
		Cypher	25,00	23,38	24,96	24,94
	Uso médio RAM (Bytes)	SQL	9089024	7907840	8273408	9573888
		STP	1515008	1462784	1186816	4359680
		API	5726632	5726632	5726632	22396837
		Cypher	5885896	5885896	5885896	5885896

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	840,875	838,875	946,250	831,375
		STP	47,000	37,125	156,000	38,875
		API	7,625	3,875	6,000	7,750
		Cypher	795,750	797,750	795,625	795,875
	Uso médio CPU (%)	SQL	24,41	24,65	24,68	24,83
		STP	17,51	20,02	21,42	19,96
		API	25,05	25,00	23,33	24,94
		Cypher	25,00	25,00	25,00	24,97
	Uso médio RAM (Bytes)	SQL	301056	499200	300032	293376
		STP	1233408	1173504	4466176	1624576
		API	7874084	5726632	8589904	5726632
		Cypher	5885896	5885896	5885896	5885896
CD50	**Tempo médio exec. (ms)**	SQL	200614,375	200267,250	200181,625	200204,875
		STP	52,5	50,625	315,875	52,625
		API	15,125	11,625	23,000	15,750
		Cypher	803,375	815,250	801,375	793,500
	Uso médio CPU (%)	SQL	24,72	24,69	24,74	24,73
		STP	18,90	19,74	24,56	17,04
		API	23,42	23,89	25,01	24,92
		Cypher	24,89	25,00	25,00	24,94
	Uso médio RAM (Bytes)	SQL	7719936	8380928	8491008	7663104
		STP	1350656	1221632	1039360	1186816
		API	5726632	5726632	7158264	5726632
		Cypher	5885896	5885896	5885896	5885896
CD100	**Tempo médio exec. (ms)**	SQL	447929,500	447523,625	398824,125	398861,375
		STP	51,000	48,750	390,000	52,625
		API	17,625	15,750	23,375	19,500
		Cypher	795,625	787,875	783,750	780,000
	Uso médio CPU (%)	SQL	24,71	24,67	24,66	24,68
		STP	16,38	21,30	24,63	16,49
		API	24,90	19,75	24,51	25,02
		Cypher	24,91	23,93	24,84	25,00
	Uso médio RAM (Bytes)	SQL	7901696	7666688	8508928	7665152
		STP	1494016	1295872	1028608	1194496
		API	5726632	5726632	7158264	5726632
		Cypher	5885896	5885896	5885896	5885896

C.4 Medições da Consulta OQ1

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,875	3,875	1,875	11,750
		STP	1,250	3,000	1,875	11,000
		API	2,000	0,001	0,001	2,000
		Cypher	282,875	280,625	281,000	280,750
	Uso médio CPU (%)	SQL	10,88	9,32	7,29	33,33
		STP	7,86	9,37	12,43	22,88
		API	24,94	21,92	25,00	25,04
		Cypher	24,89	24,84	24,94	24,94
	Uso médio RAM (Bytes)	SQL	2048	3584	512	10240
		STP	2048	1536	1536	10752
		API	5726584	5726584	5726584	5726584
		Cypher	6292464	6292464	6657832	6657832
CD50	**Tempo médio exec. (ms)**	SQL	2,000	7,750	7,625	7,875
		STP	1,750	6,875	6,875	7,500
		API	5,625	4,000	6,000	7,750
		Cypher	286,625	280,750	288,625	282,750
	Uso médio CPU (%)	SQL	14,08	14,71	7,80	16,97
		STP	7,76	7,05	13,54	13,09
		API	24,89	25,00	24,95	24,22
		Cypher	24,92	23,84	24,97	24,93
	Uso médio RAM (Bytes)	SQL	512	512	4608	89600
		STP	512	512	1024	6896
		API	5726584	5726584	5726584	5726584
		Cypher	5927096	5927096	6292464	7023200
CD100	**Tempo médio exec. (ms)**	SQL	7,875	4,000	5,875	15,875
		STP	5,625	7,875	3,875	7,500
		API	15,750	0,001	0,001	3,875
		Cypher	280,875	280,875	281,000	286,750
	Uso médio CPU (%)	SQL	11,94	13,63	17,66	18,17
		STP	9,80	6,25	15,60	15,31
		API	23,40	25,00	24,94	24,95
		Cypher	24,90	24,91	25,02	25,01
	Uso médio RAM (Bytes)	SQL	512	512	10240	221184
		STP	512	512	1536	5120
		API	5726584	5726584	5726584	5506328
		Cypher	5927096	5927096	5927096	6657832

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	3,500	2,000	2,125	2,500
		STP	3,000	1,875	1,750	2,875
		API	0,001	5,875	2,000	2,000
		Cypher	282,750	280,875	284,750	282,875
	Uso médio CPU (%)	SQL	10,93	10,93	10,43	12,50
		STP	17,08	7,26	16,26	12,50
		API	24,94	23,38	25,00	23,38
		Cypher	24,94	25,00	32,76	25,05
	Uso médio RAM (Bytes)	SQL	2560	2048	5632	4096
		STP	512	512	512	512
		API	5726584	5726584	5726584	5726584
		Cypher	6292464	6292464	6657832	5927096
CD50	**Tempo médio exec. (ms)**	SQL	3,875	4,000	5,750	3,875
		STP	3,500	3,750	5,750	3,875
		API	1,875	3,875	4,000	2,000
		Cypher	286,750	284,625	283,000	280,875
	Uso médio CPU (%)	SQL	9,49	13,98	13,12	12,00
		STP	12,50	10,43	12,93	10,97
		API	24,94	25,05	25,00	24,94
		Cypher	24,97	25,00	24,93	23,86
	Uso médio RAM (Bytes)	SQL	512	512	512	512
		STP	1536	512	512	512
		API	5726584	5726584	5726584	5726584
		Cypher	5927096	5927096	6292464	5927096
CD100	**Tempo médio exec. (ms)**	SQL	5,750	6,000	7,750	3,875
		STP	8,500	4,750	6,625	6,375
		API	5,625	3,875	11,750	9,750
		Cypher	296,000	282,750	282,750	280,750
	Uso médio CPU (%)	SQL	14,11	23,98	16,30	13,31
		STP	22,06	7,81	9,37	13,29
		API	25,00	25,00	24,94	25,00
		Cypher	23,91	23,89	25,03	25,00
	Uso médio RAM (Bytes)	SQL	1024	512	512	2560
		STP	2560	1024	512	512
		API	5726584	5726584	5726584	5726584
		Cypher	5927096	5927096	6292464	5927096

C.5 Medições da Consulta OQ3

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	3,875	3,750	4,000	15,750
		STP	3,000	3,500	4,125	14,625
		API	0,001	0,001	1,875	3,875
		Cypher	125,000	124,875	124,750	125,000
	Uso médio CPU (%)	SQL	15,63	15,52	17,13	14,01
		STP	12,50	7,86	6,25	23,36
		API	25,00	25,00	24,89	23,97
		Cypher	24,94	25,09	26,46	25,00
	Uso médio RAM (Bytes)	SQL	10752	512	2560	512
		STP	7680	1536	6656	407040
		API	5726592	5726592	5726592	5726592
		Cypher	5927104	5927104	5927104	5927104
CD50	**Tempo médio exec. (ms)**	SQL	4,500	9,750	11,625	21,375
		STP	3,625	9,375	11,000	20,375
		API	0,001	7,750	2,000	7,875
		Cypher	125,000	124,750	125,000	126,625
	Uso médio CPU (%)	SQL	17,23	13,02	17,61	15,82
		STP	10,88	12,44	8,60	23,25
		API	25,00	24,94	23,28	24,16
		Cypher	23,38	25,00	24,96	24,89
	Uso médio RAM (Bytes)	SQL	1024	2048	60416	70656
		STP	9728	2560	6144	129024
		API	5726592	5726592	6442412	5726592
		Cypher	5927104	5927104	5927104	5927104
CD100	**Tempo médio exec. (ms)**	SQL	9,625	9,750	9,750	15,875
		STP	4,500	4,625	5,625	12,250
		API	17,750	13,750	14,000	15,750
		Cypher	124,875	125,000	123,125	124,750
	Uso médio CPU (%)	SQL	17,01	17,69	14,51	18,64
		STP	11,48	10,88	9,37	22,48
		API	23,88	23,38	26,04	23,35
		Cypher	25,63	24,89	24,94	25,00
	Uso médio RAM (Bytes)	SQL	20992	3584	512	8192
		STP	4096	512	6144	131024
		API	5726592	5726592	6442412	5506336
		Cypher	5927104	5927104	5927104	5927104

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	3,875	3,875	3,750	3,875
		STP	3,875	4,000	4,000	3,375
		API	2,000	0,001	0,001	0,001
		Cypher	124,875	124,625	125,000	124,750
	Uso médio CPU (%)	SQL	15,62	14,58	12,44	15,62
		STP	18,75	7,81	9,37	7,81
		API	24,89	24,94	24,94	25,00
		Cypher	24,91	25,00	32,76	24,91
	Uso médio RAM (Bytes)	SQL	512	5632	512	2560
		STP	5120	512	2560	2048
		API	5726592	5726592	5726592	5726592
		Cypher	5927104	5927104	5927104	5927104
CD50	**Tempo médio exec. (ms)**	SQL	11,750	13,750	15,625	15,625
		STP	10,500	13,375	13,375	13,375
		API	15,750	5,875	0,001	7,750
		Cypher	125,000	124,750	124,750	124,625
	Uso médio CPU (%)	SQL	13,91	16,14	12,53	19,51
		STP	8,33	7,29	10,98	7,76
		API	25,00	24,89	25,00	23,41
		Cypher	24,94	25,00	24,99	25,00
	Uso médio RAM (Bytes)	SQL	15360	2048	3072	10240
		STP	9216	1536	512	512
		API	5726592	5726592	5726592	5726592
		Cypher	5927104	5927104	5927104	5927104
CD100	**Tempo médio exec. (ms)**	SQL	9,625	11,750	13,500	11,500
		STP	8,250	9,875	11,125	10,625
		API	11,500	11,875	9,875	13,750
		Cypher	124,750	124,875	123,250	125,000
	Uso médio CPU (%)	SQL	16,59	14,56	19,62	15,47
		STP	13,25	10,88	12,44	6,25
		API	25,01	24,95	24,94	26,08
		Cypher	25,00	24,99	24,89	24,96
	Uso médio RAM (Bytes)	SQL	23040	10240	11776	14336
		STP	1024	512	4096	3072
		API	5726592	5726592	5726592	5726592
		Cypher	5927104	5927104	5927104	5927104

C.6 Medições da Consulta OQ4

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	844,875	846,750	833,250	1121,250
		STP	78,000	54,375	46,750	1682,750
		API	1,875	9,750	1,875	79,875
		Cypher	296,625	296,375	296,250	292,500
	Uso médio CPU (%)	SQL	26,77	25,06	24,71	24,44
		STP	20,28	23,19	21,65	23,03
		API	24,94	24,87	23,12	23,78
		Cypher	25,00	29,73	24,99	23,38
	Uso médio RAM (Bytes)	SQL	4820992	375808	299520	7494144
		STP	3938816	1065984	1047552	5570560
		API	5726632	9305724	5726632	22906184
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	201007,500	223624,625	200631,750	208533,000
		STP	78,000	101,125	56,5	18615
		API	15,500	19,750	15,750	185,250
		Cypher	294,500	290,500	308,125	294,375
	Uso médio CPU (%)	SQL	24,74	24,76	24,71	24,69
		STP	20,89	21,88	19,90	24,82
		API	23,33	24,22	24,89	24,41
		Cypher	24,94	25,00	25,00	24,90
	Uso médio RAM (Bytes)	SQL	1833472	10485248	9456128	9164288
		STP	1516032	1081344	1158144	13837312
		API	5726632	8589904	5726632	34359256
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	452015,375	452016,625	448807,000	402069,625
		STP	87,750	91,625	58,250	15414,750
		API	17,875	27,375	15,625	273,000
		Cypher	286,750	284,750	290,375	284,875
	Uso médio CPU (%)	SQL	24,52	24,51	24,71	24,60
		STP	19,85	21,65	20,68	24,52
		API	25,05	25,78	23,89	23,51
		Cypher	25,00	25,04	24,85	25,00
	Uso médio RAM (Bytes)	SQL	7403520	8856064	8076800	7298048
		STP	900608	1025024	1274368	4993536
		API	5726632	8589904	5726632	38544000
		Cypher	5927144	5927144	5927144	5927144

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	834,875	833,000	948,250	835,000
		STP	41,000	33,125	154,125	31,125
		API	9,625	4,000	2,125	5,750
		Cypher	290,625	296,250	288,750	294,375
	Uso médio CPU (%)	SQL	25,00	24,76	25,05	24,65
		STP	17,11	19,06	21,83	24,32
		API	24,94	25,05	25,00	25,00
		Cypher	24,96	25,00	24,94	25,00
	Uso médio RAM (Bytes)	SQL	289280	351744	233472	358912
		STP	1000448	1303552	3927552	1157120
		API	7158272	5726632	7874092	5726632
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	200641,750	200148,375	200175,625	200113,125
		STP	46,875	47	317,625	48,625
		API	31,250	11,875	0,001	13,625
		Cypher	296,250	294,375	292,625	296,500
	Uso médio CPU (%)	SQL	24,71	24,70	24,71	24,72
		STP	20,87	22,04	23,43	21,99
		API	24,02	23,26	25,05	23,36
		Cypher	24,95	24,94	23,48	25,00
	Uso médio RAM (Bytes)	SQL	8045568	8540672	7970304	7860224
		STP	1372160	1385984	843264	1153536
		API	7158272	5726632	8589912	5726632
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	448282,250	447519,625	398584,750	398598,125
		STP	48,625	46,875	390,000	54,250
		API	39,125	10,000	0,125	15,375
		Cypher	296,375	292,375	282,750	286,750
	Uso médio CPU (%)	SQL	24,71	24,67	24,67	24,67
		STP	20,83	19,63	23,59	18,77
		API	23,42	24,90	24,94	23,44
		Cypher	24,87	24,94	24,89	25,00
	Uso médio RAM (Bytes)	SQL	8257024	7855104	8728576	8302592
		STP	1198592	1375744	686080	1044992
		API	7158272	5726632	7874092	5726632
		Cypher	5927144	5927144	5927144	5927144

C.7 Medições da Consulta OQ5

			Karma3	KCL	NCSA	PASS3
CD1	Tempo médio exec. (ms)	SQL	17,875	15,750	14,000	13,500
		STP	15,625	15,625	15,500	15,750
		API	9,375	3,875	6,000	5,875
		Cypher	66,750	72,250	68,125	64,500
	Uso médio CPU (%)	SQL	18,69	20,31	19,49	20,22
		STP	24,94	25,00	23,43	23,38
		API	24,90	21,87	26,46	24,92
		Cypher	25,05	26,61	25,05	25,00
	Uso médio RAM (Bytes)	SQL	7680	1536	512	512
		STP	4096	13312	11776	57344
		API	5726592	5726592	5726592	5726592
		Cypher	5968592	5968592	5968592	5968592
CD50	Tempo médio exec. (ms)	SQL	349,750	364,750	341,125	350,250
		STP	358,750	374,500	351,125	358,750
		API	21,500	13,750	17,875	15,625
		Cypher	68,125	70,500	70,500	70,000
	Uso médio CPU (%)	SQL	22,66	21,71	21,79	19,94
		STP	22,42	21,59	20,64	19,92
		API	25,06	23,31	23,95	24,27
		Cypher	24,89	24,89	25,00	25,00
	Uso médio RAM (Bytes)	SQL	183296	518656	558592	361472
		STP	211456	129024	547328	907264
		API	13600573	6442409	9305692	5726592
		Cypher	5968592	5968592	5968592	5968592
CD100	Tempo médio exec. (ms)	SQL	680,875	706,000	659,125	657,125
		STP	698,250	727,500	676,875	695,375
		API	31,125	15,500	19,500	24,000
		Cypher	68,875	66,250	66,125	68,000
	Uso médio CPU (%)	SQL	20,63	21,57	20,47	21,01
		STP	22,86	21,58	22,44	21,02
		API	22,43	21,41	22,92	23,06
		Cypher	24,90	24,91	25,00	24,00
	Uso médio RAM (Bytes)	SQL	3305472	422400	921088	720384
		STP	1259008	791040	901632	991744
		API	12884750	7158229	15032219	11384284
		Cypher	5927104	5927104	5927104	5927104

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	13,875	15,875	155,750	15,875
		STP	13,750	15,625	155,625	15,625
		API	0,001	4,000	0,001	5,625
		Cypher	72,250	68,250	70,125	68,375
	Uso médio CPU (%)	SQL	18,75	20,31	15,00	21,87
		STP	42,28	25,05	25,00	24,94
		API	25,00	23,38	25,00	25,00
		Cypher	25,00	25,00	24,94	23,38
	Uso médio RAM (Bytes)	SQL	512	1536	23040	512
		STP	2048	2048	512	4096
		API	5726592	5726592	6442409	5726592
		Cypher	5968592	5968592	5968592	5968592
CD50	**Tempo médio exec. (ms)**	SQL	265,375	341,375	1220,750	341,125
		STP	273,125	349,000	386,250	351,000
		API	2,000	15,625	17,500	17,875
		Cypher	70,000	68,000	68,375	70,375
	Uso médio CPU (%)	SQL	19,82	20,04	14,89	20,79
		STP	21,73	21,35	22,49	21,84
		API	23,33	25,04	22,33	24,95
		Cypher	24,94	25,05	23,43	24,94
	Uso médio RAM (Bytes)	SQL	173056	552960	8082432	225792
		STP	166400	299008	654848	228352
		API	5726592	9305689	12884753	10737332
		Cypher	5968592	5968592	5968592	5968592
CD100	**Tempo médio exec. (ms)**	SQL	512,875	668,750	2577,875	657,250
		STP	528,500	674,750	758,750	686,375
		API	1,875	21,500	29,250	19,625
		Cypher	66,625	68,625	64,125	64,875
	Uso médio CPU (%)	SQL	19,95	21,26	14,29	20,70
		STP	21,16	21,11	22,38	21,87
		API	25,00	25,20	23,61	26,38
		Cypher	25,00	23,93	24,89	25,00
	Uso médio RAM (Bytes)	SQL	516608	845824	6657536	411648
		STP	214016	305152	1005056	269312
		API	7874049	10737329	12168930	15032219
		Cypher	5927104	5927104	5927104	5927104

C.8 Medições da Consulta OQ6

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,500	1,875	2,875	9,750
		STP	1,875	1,750	2,750	9,500
		API	6,000	5,750	4,000	3,875
		Cypher	70,500	68,375	68,000	68,375
	Uso médio CPU (%)	SQL	12,44	13,51	10,98	16,32
		STP	9,37	9,34	7,76	22,24
		API	23,93	25,05	23,78	24,89
		Cypher	25,00	25,05	25,05	24,94
	Uso médio RAM (Bytes)	SQL	10240	2048	3072	512
		STP	512	512	512	34816
		API	6442452	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	1,875	15,750	3,750	8,125
		STP	1,750	15,000	3,750	8,375
		API	13,500	13,500	13,625	15,625
		Cypher	70,250	64,500	70,250	68,125
	Uso médio CPU (%)	SQL	9,37	18,75	12,05	18,14
		STP	12,50	16,56	6,25	20,50
		API	25,06	25,04	25,00	23,31
		Cypher	25,00	24,87	24,94	25,00
	Uso médio RAM (Bytes)	SQL	155648	8192	22528	38400
		STP	512	3072	8704	409600
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	5,875	5,875	5,750	15,625
		STP	3,750	5,750	4,750	10,625
		API	15,625	15,625	15,375	15,625
		Cypher	66,250	72,375	66,750	66,250
	Uso médio CPU (%)	SQL	17,79	21,82	14,16	18,60
		STP	12,44	13,56	7,79	16,64
		API	23,93	25,03	23,33	24,42
		Cypher	24,94	25,00	25,00	24,93
	Uso médio RAM (Bytes)	SQL	8192	512	512	512
		STP	512	512	512	409600
		API	5726632	5726632	5726632	5506376
		Cypher	5927144	5927144	5927144	5927144

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	3,875	2,125	15,500	3,125
		STP	3,750	1,750	13,875	2,750
		API	0,001	4,000	6,000	5,625
		Cypher	70,250	70,000	70,500	67,000
	Uso médio CPU (%)	SQL	12,50	7,81	25,00	12,50
		STP	17,68	7,81	12,50	7,76
		API	24,89	23,48	24,89	25,00
		Cypher	25,00	25,00	24,94	25,00
	Uso médio RAM (Bytes)	SQL	3072	2048	3072	3072
		STP	1536	1024	3584	4096
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	4,000	2,000	15,875	5,750
		STP	3,625	1,875	14,375	5,750
		API	2,000	11,750	11,750	9,750
		Cypher	70,375	70,000	70,500	70,125
	Uso médio CPU (%)	SQL	14,47	12,99	18,23	16,38
		STP	6,25	9,37	25,00	8,35
		API	22,33	25,00	22,33	25,00
		Cypher	25,00	23,43	24,94	24,94
	Uso médio RAM (Bytes)	SQL	8192	20480	512	16384
		STP	512	512	5120	512
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	3,875	7,750	5,750	3,875
		STP	2,750	6,750	4,625	2,750
		API	0,001	13,750	11,750	15,875
		Cypher	66,625	66,750	64,375	68,125
	Uso médio CPU (%)	SQL	10,94	11,38	11,75	11,57
		STP	9,37	4,68	10,18	7,00
		API	25,00	25,05	25,00	22,34
		Cypher	25,03	24,97	25,00	24,95
	Uso médio RAM (Bytes)	SQL	512	512	512	512
		STP	512	512	1536	512
		API	5726632	5726632	5726632	5726632
		Cypher	5927144	5927144	5927144	5927144

C.9 Medições da Consulta OQ7

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,875	11,750	3,875	8,625
		STP	1,625	9,750	3,750	8,115
		API	5,875	5,625	5,750	5,875
		Cypher	80,000	80,000	81,750	78,000
	Uso médio CPU (%)	SQL	9,37	19,96	15,62	21,64
		STP	7,81	10,16	4,63	24,51
		API	25,00	24,89	24,89	25,00
		Cypher	24,94	24,94	25,00	24,94
	Uso médio RAM (Bytes)	SQL	5632	4096	2560	31744
		STP	512	512	512	103936
		API	6442436	5726616	5726616	5726616
		Cypher	5968616	5968616	5968616	5968616
CD50	**Tempo médio exec. (ms)**	SQL	3,875	13,750	6,000	9,625
		STP	2,625	10,000	5,500	9,250
		API	13,625	13,625	11,625	9,750
		Cypher	80,000	78,000	78,000	78,000
	Uso médio CPU (%)	SQL	10,98	17,73	18,43	15,06
		STP	7,86	11,69	12,50	15,71
		API	25,05	23,93	25,02	24,28
		Cypher	25,05	24,94	24,89	24,94
	Uso médio RAM (Bytes)	SQL	25600	53248	24064	11264
		STP	512	512	512	37376
		API	5726616	5726616	5726616	5726616
		Cypher	5968616	5968616	5968616	5968616
CD100	**Tempo médio exec. (ms)**	SQL	5,875	5,625	5,875	15,500
		STP	4,750	4,375	4,625	12,500
		API	15,625	15,500	15,500	15,750
		Cypher	78,000	79,875	78,000	78,000
	Uso médio CPU (%)	SQL	17,16	12,18	14,13	16,44
		STP	9,87	9,37	13,13	14,84
		API	22,31	25,02	23,89	24,18
		Cypher	24,97	24,00	24,99	24,94
	Uso médio RAM (Bytes)	SQL	8192	1024	512	512
		STP	512	43008	1024	70144
		API	5726616	5726616	5726616	5506360
		Cypher	5927128	5927128	5927128	5927128

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	0,001	2,000	4,000	6,500
		STP	7,625	6,000	10,000	6,750
		API	0,001	1,875	7,625	3,750
		Cypher	85,875	78,000	80,000	82,000
	Uso médio CPU (%)	SQL	15,62	9,37	16,56	10,98
		STP	13,96	7,31	15,83	9,37
		API	25,00	24,84	24,89	24,89
		Cypher	25,00	23,43	25,05	24,92
	Uso médio RAM (Bytes)	SQL	1024	512	64000	2048
		STP	1024	512	512	512
		API	5726616	5726616	6442436	5726616
		Cypher	5968616	5968616	5968616	5968616
CD50	**Tempo médio exec. (ms)**	SQL	7,750	9,750	7,250	7,750
		STP	7,625	9,875	7,000	7,375
		API	0,001	13,625	15,625	9,625
		Cypher	81,875	79,875	78,000	78,000
	Uso médio CPU (%)	SQL	18,55	19,02	22,08	11,58
		STP	7,05	9,37	13,72	7,01
		API	24,94	24,90	25,00	24,95
		Cypher	24,96	25,00	25,00	24,89
	Uso médio RAM (Bytes)	SQL	512	5632	4096	512
		STP	512	6144	512	512
		API	5726616	5726616	5726616	5726616
		Cypher	5968616	5968616	5968616	5968616
CD100	**Tempo médio exec. (ms)**	SQL	7,750	7,875	7,750	5,875
		STP	6,000	7,875	7,500	4,250
		API	0,001	13,625	13,625	15,750
		Cypher	78,000	78,250	78,000	78,000
	Uso médio CPU (%)	SQL	15,57	17,71	16,62	15,65
		STP	6,25	10,88	12,50	3,12
		API	25,00	23,92	24,91	24,99
		Cypher	24,95	24,94	24,87	25,00
	Uso médio RAM (Bytes)	SQL	512	3072	512	512
		STP	512	512	5120	512
		API	5726616	5726616	5726616	5726616
		Cypher	5927128	5927128	5927128	5927128

C.10 Medições da Consulta OQ8

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	846,875	852,625	835,000	1097,750
		STP	80,000	52,625	33,125	1673,125
		API	6,000	3,750	6,000	6,000
		Cypher	292,375	292,625	294,375	290,625
	Uso médio CPU (%)	SQL	27,62	24,72	24,88	24,35
		STP	22,14	21,26	18,95	23,56
		API	25,00	23,43	25,00	24,97
		Cypher	25,00	24,94	25,04	24,16
	Uso médio RAM (Bytes)	SQL	4402688	578560	266240	7204352
		STP	4151808	1113600	1373696	5662720
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	201054,125	222101,125	200513,000	207690,750
		STP	76,125	80,000	48,75	16352,5
		API	13,625	11,875	13,750	15,625
		Cypher	294,625	294,625	294,500	294,250
	Uso médio CPU (%)	SQL	24,72	24,56	24,77	24,75
		STP	21,66	19,36	19,71	24,83
		API	25,09	24,99	23,48	24,92
		Cypher	23,88	23,38	24,89	24,89
	Uso médio RAM (Bytes)	SQL	2756096	10485760	7030784	10012672
		STP	850432	989184	1076736	9607680
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	451932,625	435215,500	448315,750	400979,250
		STP	81,750	60,625	46,750	11522,500
		API	15,625	17,125	16,000	15,625
		Cypher	288,750	282,750	284,750	282,750
	Uso médio CPU (%)	SQL	24,54	24,33	24,71	24,61
		STP	22,23	23,43	19,55	24,38
		API	23,91	24,32	24,92	24,94
		Cypher	24,91	24,95	24,89	25,00
	Uso médio RAM (Bytes)	SQL	7860224	7355884	7532544	9987584
		STP	1136128	1539072	1292800	5090304
		API	5726632	5726632	5726632	5506376
		Cypher	5927144	5927144	5927144	5968632

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	837,125	835,375	954,250	833,625
		STP	39,000	39,000	167,625	37,000
		API	1,000	2,000	5,750	5,875
		Cypher	296,375	292,750	292,375	294,500
	Uso médio CPU (%)	SQL	24,53	24,76	24,69	25,05
		STP	18,42	18,83	22,46	20,28
		API	23,48	25,05	24,00	24,84
		Cypher	25,00	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	428032	337920	424960	379904
		STP	1073664	1065472	4217344	1339392
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD50	**Tempo médio exec. (ms)**	SQL	200631,750	200114,875	200308,250	200136,875
		STP	46,875	46,875	236	46,75
		API	0,001	13,625	9,750	13,750
		Cypher	296,250	292,375	292,625	290,625
	Uso médio CPU (%)	SQL	24,70	24,71	24,74	24,75
		STP	19,13	18,81	23,84	19,46
		API	25,00	23,90	24,98	24,94
		Cypher	24,92	25,00	25,00	24,89
	Uso médio RAM (Bytes)	SQL	6934528	7546880	8893440	8052736
		STP	1212416	936960	1022976	1043456
		API	5726632	5726632	5726632	5726632
		Cypher	5968632	5968632	5968632	5968632
CD100	**Tempo médio exec. (ms)**	SQL	448220,000	447486,750	33,250	398945,500
		STP	47,000	47,000	202,750	46,750
		API	0,001	16,125	15,750	15,375
		Cypher	292,375	292,500	284,875	284,750
	Uso médio CPU (%)	SQL	24,71	24,69	18,41	24,67
		STP	21,15	20,57	22,57	21,71
		API	23,46	23,88	23,08	25,02
		Cypher	24,86	25,00	24,90	25,00
	Uso médio RAM (Bytes)	SQL	8326656	8051712	1771520	7012864
		STP	1400832	1082368	841216	1104384
		API	5726632	5726632	5726632	5726632
		Cypher	5927144	5927144	5927144	5927144

C.11 Medições da Consulta OQ10

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,875	1,875	5,000	61,375
		STP	1,875	1,875	4,750	58,625
		API	2,000	2,000	0,001	43,125
		Cypher	483,750	471,750	477,875	468,000
	Uso médio CPU (%)	SQL	9,37	13,51	10,98	18,51
		STP	9,37	10,41	9,38	22,99
		API	23,93	22,46	23,04	23,46
		Cypher	25,00	25,05	25,08	25,00
	Uso médio RAM (Bytes)	SQL	7680	4096	2048	50176
		STP	512	512	2048	168448
		API	5236431	2147460	2345412	6442356
		Cypher	5036254	5036254	5036254	5036254
CD50	**Tempo médio exec. (ms)**	SQL	1,875	15,875	5,750	124,000
		STP	1,625	8,000	6,625	104,250
		API	0,001	1,875	0,001	44,875
		Cypher	471,750	471,875	483,500	473,750
	Uso médio CPU (%)	SQL	10,93	8,83	19,59	22,42
		STP	7,86	15,57	12,50	22,32
		API	22,49	23,29	23,78	24,01
		Cypher	24,94	24,99	24,94	25,00
	Uso médio RAM (Bytes)	SQL	2560	512	2048	50688
		STP	3584	512	512	783872
		API	5036254	2147460	2345412	6442356
		Cypher	5036254	5036254	5036254	5036254
CD100	**Tempo médio exec. (ms)**	SQL	1,875	17,500	5,875	62,000
		STP	1,750	13,875	4,375	60,875
		API	0,001	1,875	1,875	44,875
		Cypher	473,750	468,000	473,875	468,250
	Uso médio CPU (%)	SQL	17,14	13,65	16,17	20,82
		STP	12,50	15,62	12,50	21,93
		API	23,43	25,00	25,00	24,98
		Cypher	24,94	25,00	24,84	25,00
	Uso médio RAM (Bytes)	SQL	512	512	512	1024
		STP	512	512	512	410112
		API	5726592	6442409	5726592	16890552
		Cypher	5883896	5883896	5883896	5869008

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	4,875	5,500	10,875	1,875
		STP	4,750	5,750	9,750	1,875
		API	0,001	0,001	3,875	4,000
		Cypher	479,750	477,750	473,875	473,875
	Uso médio CPU (%)	SQL	9,37	9,37	9,37	10,93
		STP	23,43	12,50	10,90	7,76
		API	23,48	25,00	24,10	25,99
		Cypher	25,00	24,95	24,91	25,00
	Uso médio RAM (Bytes)	SQL	2560	4096	512	4096
		STP	512	2048	2048	512
		API	715820	542522	1431640	715820
		Cypher	5036254	5036254	5036254	5036254
CD50	**Tempo médio exec. (ms)**	SQL	5,875	2,000	11,500	1,875
		STP	6,500	1,625	11,875	1,500
		API	1,875	0,001	2,000	0,001
		Cypher	471,750	475,500	483,500	471,875
	Uso médio CPU (%)	SQL	21,93	17,70	16,24	16,09
		STP	9,37	9,37	17,91	12,50
		API	23,65	23,95	24,23	23,84
		Cypher	24,94	25,08	25,00	25,05
	Uso médio RAM (Bytes)	SQL	512	512	512	1024
		STP	512	2048	2560	512
		API	715820	542522	1431640	715820
		Cypher	5036254	5036254	5036254	5036254
CD100	**Tempo médio exec. (ms)**	SQL	7,625	3,875	9,750	3,875
		STP	7,625	3,625	8,625	2,500
		API	0,001	1,875	4,000	0,001
		Cypher	475,625	473,750	469,875	464,125
	Uso médio CPU (%)	SQL	17,64	14,83	13,47	19,30
		STP	12,44	9,37	20,83	7,86
		API	25,00	25,00	24,97	25,00
		Cypher	24,93	25,00	24,84	24,97
	Uso médio RAM (Bytes)	SQL	512	512	10752	512
		STP	3072	1024	12800	4096
		API	6442412	5726592	7158214	5726592
		Cypher	5883896	5883896	5883896	5883896

C.12 Medições da Consulta OQ11

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1,875	12,875	1,875	112,250
		STP	1,175	10,000	1,875	109,000
		API	0,001	0,001	0,001	49,000
		Cypher	468,000	466,125	466,125	464,125
	Uso médio CPU (%)	SQL	14,06	17,23	12,50	15,74
		STP	10,88	18,75	10,93	23,70
		API	24,94	25,00	24,94	24,55
		Cypher	24,94	25,00	24,93	25,01
	Uso médio RAM (Bytes)	SQL	14848	3072	2560	1209344
		STP	1536	1024	512	1399808
		API	5726616	5726604	5726616	17895466
		Cypher	5855632	5855632	5840856	5855632
CD50	**Tempo médio exec. (ms)**	SQL	1,875	13,625	3,750	141,875
		STP	1,000	12,625	3,000	136,250
		API	0,001	2,000	0,001	52,875
		Cypher	468,000	466,000	475,750	468,000
	Uso médio CPU (%)	SQL	14,11	14,30	11,93	22,39
		STP	9,32	20,36	10,93	23,62
		API	24,94	25,00	24,89	25,37
		Cypher	23,28	25,00	24,97	25,00
	Uso médio RAM (Bytes)	SQL	512	8192	512	1832960
		STP	512	1024	512	1556480
		API	5726616	6442424	5726616	17895466
		Cypher	5855632	5855632	5855632	5855632
CD100	**Tempo médio exec. (ms)**	SQL	3,875	15,500	9,750	156,000
		STP	2,125	15,750	9,250	123,375
		API	0,001	1,875	0,001	52,875
		Cypher	458,250	460,250	462,125	458,250
	Uso médio CPU (%)	SQL	12,00	14,79	13,42	23,65
		STP	9,37	16,68	9,37	22,34
		API	25,04	24,94	24,93	25,00
		Cypher	25,00	24,97	23,90	23,46
	Uso médio RAM (Bytes)	SQL	512	512	512	744448
		STP	512	1024	512	1593856
		API	5726616	5726598	5726616	16890555
		Cypher	5855632	5855632	5855632	5845888

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	1,875	1,875	5,875	1,875
		STP	1,375	1,625	5,250	1,625
		API	0,001	0,001	3,875	0,001
		Cypher	468,000	468,000	460,125	468,000
	Uso médio CPU (%)	SQL	10,93	10,41	14,06	9,37
		STP	24,94	12,50	14,06	10,98
		API	24,94	25,00	25,00	25,00
		Cypher	25,00	25,00	24,97	25,06
	Uso médio RAM (Bytes)	SQL	512	3072	2048	512
		STP	512	512	512	2048
		API	6442436	5726616	5726574	5726616
		Cypher	5855632	5855632	5840856	5855632
CD50	**Tempo médio exec. (ms)**	SQL	3,750	5,750	9,500	5,750
		STP	4,000	6,000	9,000	6,000
		API	0,001	0,001	0,001	0,001
		Cypher	469,875	468,000	468,000	464,125
	Uso médio CPU (%)	SQL	12,45	12,85	18,80	15,01
		STP	12,50	10,93	14,58	9,37
		API	24,00	25,00	24,94	25,00
		Cypher	24,90	25,00	25,00	24,92
	Uso médio RAM (Bytes)	SQL	512	512	512	512
		STP	2560	1024	512	512
		API	5726616	5726616	5726574	5726616
		Cypher	5855632	5855632	5855632	5855632
CD100	**Tempo médio exec. (ms)**	SQL	13,500	3,750	15,625	7,750
		STP	12,875	2,625	15,500	7,875
		API	2,000	0,001	2,000	1,875
		Cypher	468,000	456,000	454,500	460,500
	Uso médio CPU (%)	SQL	14,71	17,85	13,05	17,24
		STP	10,93	12,50	20,81	9,37
		API	25,00	25,00	25,04	25,00
		Cypher	24,87	25,00	24,92	25,00
	Uso médio RAM (Bytes)	SQL	512	512	512	512
		STP	512	512	15872	1024
		API	5726616	5726616	5726574	5726616
		Cypher	5855632	5855632	5855632	5855632

C.13 Medições da Consulta OQ12

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	2,000	4,000	2,000	11,625
		STP	1,875	3,375	1,750	11,500
		API	0,001	2,000	0,001	2,000
		Cypher	76,125	76,000	74,250	72,250
	Uso médio CPU (%)	SQL	10,93	21,87	9,37	13,54
		STP	12,50	13,73	7,81	19,75
		API	25,00	24,92	24,94	26,51
		Cypher	24,84	25,00	25,00	24,94
	Uso médio RAM (Bytes)	SQL	9728	7168	2560	512
		STP	512	4096	512	9728
		API	6442436	6442436	6442436	6442436
		Cypher	5855632	5855632	5840856	5855632
CD50	**Tempo médio exec. (ms)**	SQL	3,875	15,500	5,875	191,000
		STP	3,375	15,625	6,000	185,375
		API	1,875	2,000	1,250	2,000
		Cypher	74,000	74,000	78,000	74,000
	Uso médio CPU (%)	SQL	15,62	24,93	15,06	16,07
		STP	10,88	19,30	10,93	23,28
		API	25,00	25,00	24,94	24,93
		Cypher	24,94	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	7168	21504	1024	3072
		STP	1024	5632	512	13824
		API	5726616	5726616	5726616	5726616
		Cypher	5855632	5855632	5855632	5855632
CD100	**Tempo médio exec. (ms)**	SQL	7,750	17,500	5,875	200,750
		STP	7,375	16,375	4,875	197,375
		API	2,000	3,875	5,875	2,125
		Cypher	74,375	64,375	76,125	72,125
	Uso médio CPU (%)	SQL	9,84	16,04	15,08	18,81
		STP	6,25	13,74	10,88	24,56
		API	24,94	25,00	26,56	24,89
		Cypher	24,94	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	512	512	1024	32154
		STP	512	512	512	151552
		API	5726616	5726616	5726616	5726616
		Cypher	5855632	5855632	5855632	5845888

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	2,000	4,750	5,225	3,875
		STP	1,625	4,750	5,500	3,750
		API	2,000	0,001	0,001	1,875
		Cypher	74,000	72,375	74,125	74,000
	Uso médio CPU (%)	SQL	10,93	15,62	7,76	10,93
		STP	15,63	10,98	12,50	7,81
		API	25,00	24,89	25,00	25,00
		Cypher	25,00	29,22	25,00	25,00
	Uso médio RAM (Bytes)	SQL	3584	2048	2560	512
		STP	7680	512	6656	512
		API	6442436	6442436	6442436	6442436
		Cypher	5855632	5855632	5840856	5855632
CD50	**Tempo médio exec. (ms)**	SQL	9,500	5,750	6,500	4,875
		STP	9,625	5,000	6,625	4,125
		API	1,875	2,000	1,250	1,875
		Cypher	74,125	76,000	72,250	70,250
	Uso médio CPU (%)	SQL	14,61	20,31	23,39	14,61
		STP	12,44	9,37	9,37	9,37
		API	24,94	24,92	24,94	25,00
		Cypher	25,00	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	2560	512	8192	512
		STP	2560	512	512	512
		API	5726616	5726616	5726616	5726616
		Cypher	5855632	5855632	5855632	5855632
CD100	**Tempo médio exec. (ms)**	SQL	3,750	5,875	7,625	5,750
		STP	3,000	5,625	7,625	5,000
		API	3,875	2,500	1,875	2,000
		Cypher	74,125	76,125	76,125	76,000
	Uso médio CPU (%)	SQL	16,09	15,55	15,01	18,06
		STP	7,76	7,76	9,37	6,25
		API	25,00	24,92	25,00	25,00
		Cypher	24,97	25,03	23,99	25,00
	Uso médio RAM (Bytes)	SQL	27136	512	2560	512
		STP	512	512	19456	512
		API	5726616	5726616	5726616	5726616
		Cypher	5855632	5855632	5855632	5855632

C.14 Medições da Consulta OQ13

			Karma3	KCL	NCSA	PASS3
CD1	**Tempo médio exec. (ms)**	SQL	1133,875	1110,250	1116,875	1386,125
		STP	93,625	62,375	37,000	2648,250
		API	5,625	5,875	2,000	175,875
		Cypher	306,250	310,250	304,250	296,375
	Uso médio CPU (%)	SQL	27,79	24,82	25,31	24,55
		STP	20,25	22,61	22,51	23,26
		API	25,05	25,96	24,94	24,81
		Cypher	24,94	24,95	24,89	25,00
	Uso médio RAM (Bytes)	SQL	2756608	550400	506880	7854080
		STP	4039168	751616	990208	5466112
		API	7158250	9305710	7158250	75876645
		Cypher	5896624	5896624	5922632	5896624
CD50	**Tempo médio exec. (ms)**	SQL	268083,625	294748,875	267374,625	278104,750
		STP	78,000	91,5	47	19268,125
		API	5,875	8,000	2,000	179,500
		Cypher	308,250	306,250	308,250	308,250
	Uso médio CPU (%)	SQL	24,79	24,64	24,73	24,68
		STP	22,51	21,49	19,90	24,72
		API	25,00	27,01	25,07	24,82
		Cypher	24,93	25,00	23,36	24,89
	Uso médio RAM (Bytes)	SQL	5237760	8056320	7601152	8746496
		STP	972800	651776	897024	9306624
		API	7158256	9305710	5726616	73574128
		Cypher	5896624	5896624	5896624	5896624
CD100	**Tempo médio exec. (ms)**	SQL	602473,125	616542,200	563274,300	536065,875
		STP	93,875	74,250	46,875	16009,750
		API	5,875	9,375	0,001	183,375
		Cypher	296,250	298,500	300,250	300,500
	Uso médio CPU (%)	SQL	24,52	24,64	24,16	24,59
		STP	21,55	19,90	20,29	24,43
		API	24,98	26,93	25,00	24,98
		Cypher	25,00	24,02	24,85	24,95
	Uso médio RAM (Bytes)	SQL	7408128	7456218	7976372	8086528
		STP	1000960	557056	903680	9142784
		API	7874076	9305710	6442436	76550393
		Cypher	5896624	5896624	5896624	5886808

			SDSCPc3	TetherlessPC3	UCDGC	VisTrails3
CD1	**Tempo médio exec. (ms)**	SQL	1108,875	1108,750	1175,375	1104,375
		STP	46,875	41,000	198,875	46,875
		API	4,000	6,000	18,000	5,625
		Cypher	306,250	312,125	304,375	300,625
	Uso médio CPU (%)	SQL	24,82	24,77	24,83	24,65
		STP	19,56	16,68	22,80	19,72
		API	25,05	25,05	25,00	24,94
		Cypher	25,00	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	709632	325632	249344	542208
		STP	813568	1027584	4315648	918528
		API	7158256	9305710	7874040	7874070
		Cypher	5896624	5896624	5922632	5896624
CD50	**Tempo médio exec. (ms)**	SQL	267526,750	266795,625	266756,375	266750,500
		STP	46,875	44,875	265,25	46,875
		API	5,875	2,000	15,625	2,000
		Cypher	304,250	302,250	304,375	304,250
	Uso médio CPU (%)	SQL	24,73	24,73	24,74	24,74
		STP	20,42	18,96	24,98	21,04
		API	25,00	24,97	25,06	25,00
		Cypher	24,94	25,00	25,00	25,00
	Uso médio RAM (Bytes)	SQL	9624576	9120768	7078912	8456704
		STP	1057280	964096	1145856	736768
		API	7158256	7874070	7874040	7158250
		Cypher	5896624	5896624	5896624	5896624
CD100	**Tempo médio exec. (ms)**	SQL	597272,375	596755,625	531682,125	531736,625
		STP	46,875	47,000	249,625	50,875
		API	1,875	5,875	17,625	3,875
		Cypher	306,375	304,250	302,250	296,250
	Uso médio CPU (%)	SQL	24,72	24,69	24,67	24,67
		STP	22,30	19,52	22,91	21,19
		API	24,95	25,05	25,70	25,00
		Cypher	24,96	25,00	24,87	24,95
	Uso médio RAM (Bytes)	SQL	7888384	7537152	7330304	8721408
		STP	1071616	1024512	947200	712704
		API	7874070	7874070	7874040	7874070
		Cypher	5896624	5896624	5896624	5896624